PRIFYSGOL
BANGOR
UNIVERSITY

School of Computer Science and Electronic Engineering
College of Environmental Sciences and Engineering

Generalized Ocean Simulation For Real-Time Applications

Ian V. Slesser

Submitted in partial satisfaction of the requirements for the
Degree of Bachelor of Science
in Computer Science

Supervisor Prof Jonathan C. Roberts

April 2020

Abstract

For many the ocean is a key and prominent part of their day to day lives, and most people have a familiarity with the ocean in some form, and so when it comes to simulating the ocean people have certain expectations of how it should look and behave.

For a graphical application, a realistically rendered and animated ocean can add a substantial amount of realism and immersion for a user, where it is typically easy to notice errors or unnatural patterns in the simulation, even as a background element.

Typical algorithms for simulating the ocean suffer can only simulate deep-/infinite depths, which is unsuitable for the finite/shallow areas such as the coastline, or are complex and non-performant which, for certain applications where the simulation is only a small part of the overall scene, is unsuitable.

By extending the existing Gerstner waves algorithm, this project introduces an algorithm for ocean simulation, that through highfields is able to incorporate shoreline waves and ocean waves in a single render pass suitable for real-time interactive applications viewed at moderate to far distance with a reasonable performance outlay.

The extended Gerstner waves algorithm also allows for future development to enhance the performance and fidelity of a simulation, with development paths offered as part of this project.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A realistically rendered and animated ocean can add a substantial amount of realism to an interactive graphical application [19], such as video games or interactive table-top simulations, as for many the ocean is something familiar and its behaviour is known [19], but for many of these real-time applications, the ocean simulation is only a small part of what the rest of the application would need to process and render [12].

Simulating the ocean can potentially include two distinct depths: infinite/deep and finite/shallow, corresponding to deep oceans and shorelines respectively, with this depth affecting the wavelength and magnitude of the waves [2, 8, 27]. Typically, algorithms that simulate the ocean only simulate deepwater [7, 13] or have a run-time complexity which leads to excessive-performance overhead [14, 28, 26] which can negatively impact the functionality and usability of interactive simulation.

The target type of application for this project is one in which the ocean is a background element, to add either atmosphere or context to a scene, and not intended to be viewed with great scrutiny. So to be able to simulate an ocean in this kind of real-time application an algorithm would need to only reach a certain threshold of realism, where an exact physically realistic model is not needed, likewise an accurate render of the ocean surface is not needed, in both areas an implementation needs to reach

The project aims to develop an ocean simulation algorithm which, as part of a complex scene, only takes a single, low-cost render pass to adequate realism for both deep and shallow water. The method should be capable of running within a real-time scene with other complex elements and calculations.

1.1 Objectives

The be considered successful the project must be able to meet the following objectives:

1. To provide an algorithm which simulates deep and shallow water ocean waves in a single render pass.

2. To provide an algorithm which cheaply simulates deep and shallow water ocean waves.

3. To provide an algorithm that gives a result that is adequate for mid to far viewing distances.

4. To provide an algorithm which is scalable and able to be modified and extended.

5. To demonstrate this algorithm as part of a complex real-time interactive scene

1.2 Methodology

This document will establish the background knowledge required for the project, research relevant algorithms and then present and review two different implementations of solutions to the project aim.

Two methods will be selected and each will be developed into a prototype which will then be assessed visually and through profiling, with the most suitable method for the objectives of the project will be integrated into a simple scene which includes complex interaction calculations.

The performance will then be presented and assessed visually and through profiling, allowing a conclusion to be drawn and any future development will be suggested. This paper will then conclude with a reflection as to whether or not the previously stated objectives were met.

1.3 Risk, Legal, Social, Ethical and Professional Issues

For advanced tasks where an external library is either required or makes sense to use, the project will integrate that library on the conditions that the library is licensed for use in this type of project, and that the project will adhere to the terms of the license of the library.

As part of the demonstration scene, users will have the option to use image recognition via a camera to control the scene, and as an image of the user can be classified as personal data, consideration must be given to the handling and storage of the images taken. This project addresses this problem in two ways, firstly data is not persistent and any webcam gathered data is discarded at the end of each frame. Secondly, this aspect of the project is only for demonstration purposes and so not part of the actual algorithm so any future development or release would not include this feature.

Chapter 2

Algorithm Research

There are various methods of simulating the ocean surface in computer graphics. This chapter will look at some of the prominent fluid simulation algorithms, presenting a brief overview of each algorithm. It will also briefly look at the basic theory behind ocean waves. The chapter will then end with a short conclusion of the chapter's findings.

2.1 Basic Ocean Wave Theory

The ocean is a constantly moving body of water, with waves on the surface caused by the friction of air molecules moving against the water's surface in the wind. A smaller wind force results in a smaller wave and a larger wind force results in a larger wave [1, 2]. Ocean waves are like other waves such as sound or light in that they have a wavelength (distance between crests) and an amplitude (height/depth of crests/troughs) and move in a circular orbit [1, 2, 8, 15, 27].

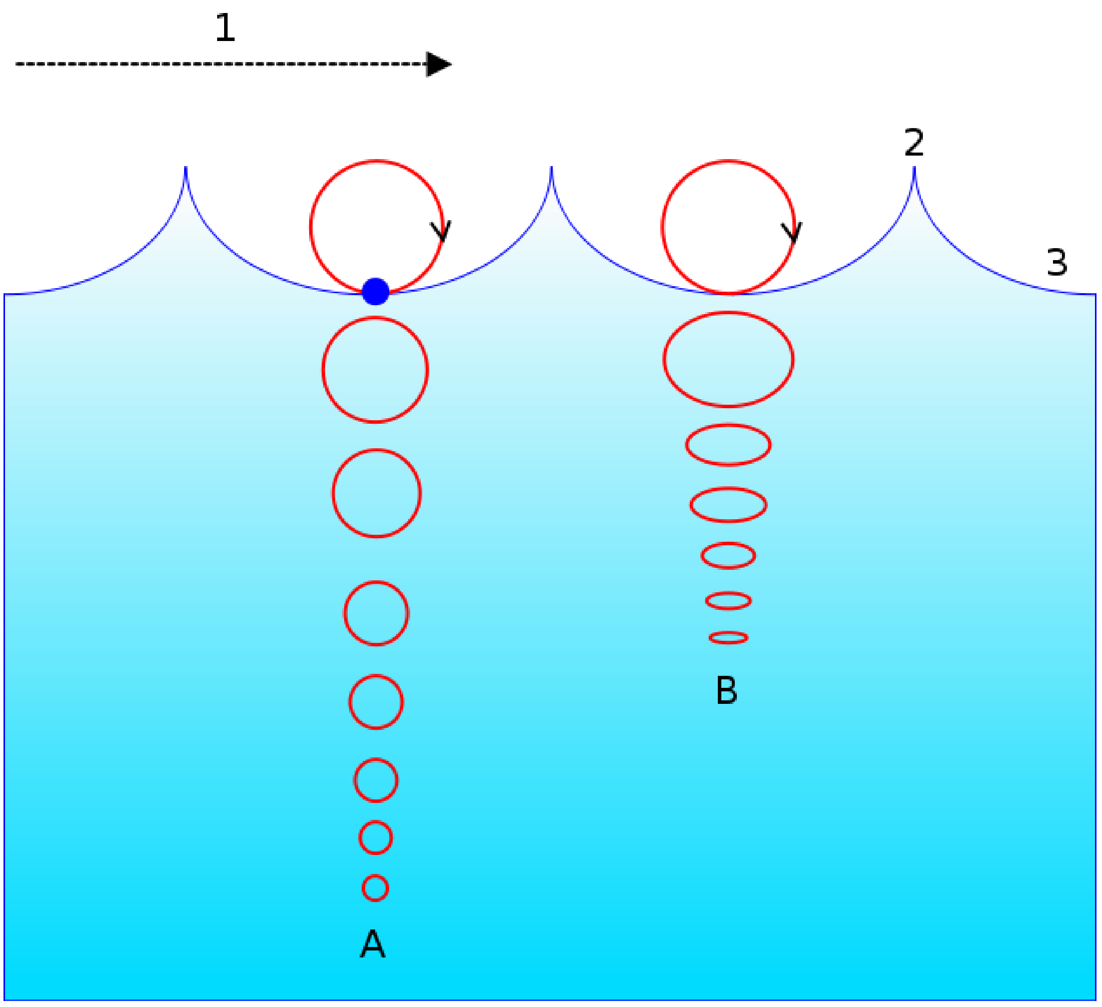


Figure 2.1: A particle motion in an ocean wave in deep water (A) and shallow water (B). The circular movement of a surface particle becomes elliptical with increasing depth by Vargklo at English Wikipedia / Public domain

A second frictional force is applied to the water when the water itself moves along the seabed [1, 15], the force of the water dissipates with depth so when the water is deep this friction is marginal, but as the water becomes shallow this force increases leading to orbit to become elliptical and an increase in amplitude, which cases the crest to approach the trough, and at some point, the crest collapses into a breaking wave [1, 2, 15, 27].

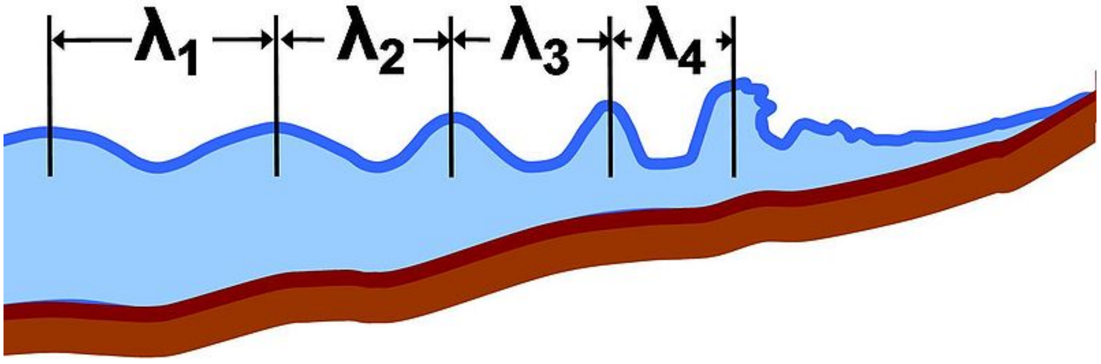


Figure 2.2: Local wavelengths defined as distances from crest to crest for an ocean wave over a varying depth of ocean bottom by Brews ohare / CC BY-SA

Taking D to be the depth of the ocean, L to be the length of the wave H to be the height of the wave, deep ocean waves are where $D\frac{1}{2}L$ and shallow ocean waves are where $D\frac{1}{20}L$ [27]. This ratio between the ocean depth and the wave's length gives instability and causes the change from circular motion to elliptical motion as discussed previously until they reach a point where the ratio is steep enough the waves collapse, or break into themselves giving the familiar shoreline wave effect.

2.2 Navier-Stokes Equations

The incompressible Navier-Stokes equations are a set of partial differential equations describing the flow of fluids [14, 26], named after Claude Navier and George Stokes who, in 1822 and 1845 respectively, formulated these equations. The Navier-Stokes equations are usually written [3, 4, 14]:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u + \frac{1}{\rho}\nabla p = g + \nu \nabla \cdot \nabla u$$

$$\nabla \cdot u = 0$$

Mathematically the state of a fluid at any given point in time is modelled as a velocity vector field, which is a function which assigns a velocity vector. The

Navier-Stokes equations define how the velocity vector field changes over time, given the current vector field state and a current set of forces [3].

Particles can then be moved using this velocity vector field by converting the surrounding velocities into forces that act on the object. Modelling individual objects within this system is computationally expensive, so an alternative is to replace individual particles with a continuous function which for every point in the field returns the exact density of objects at that point in space which can then be used to visualize the liquid flow [3].

The algorithm is unsuitable for the objective of this project as the algorithm is computationally expensive [14, 28], requiring two calculations each frame, one for the velocity field update and the second for the integration of the forces of the objects within the velocity field, and would require a vast amount of computation for a potential simulation of a large body of water such as the ocean while also being very sensitive to the time step [28] which makes it potentially unstable in real-time applications.

2.3 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) is a numerical method [10, 17, 23] for simulating a continuous field from a distribution of a small number of discrete particles with reasonable accuracy [6, 10, 18, 23], where each particle has properties such as mass, pressure, velocity and density. SPH was first proposed in 1977 by Lucy, Gingold and Monaghan [6, 10, 17] initially for applications in astrophysics, and in 1992 Monaghan described SPH's application in simulating incompressible flows such as waterfalls, bursting dams etc [18]. SPH is a gridless Langrangian method [18, 26], where each particle's position can be tracked over time [19] using statistical integration such as the Monte Carlo method [10, 17].

SPH is fairly complex, so this section will not be able to provide a full recount of it's complexities or nuances, but it will be able to provide a high-level simplified overview of the SPH algorithm. The SPH algorithm computes

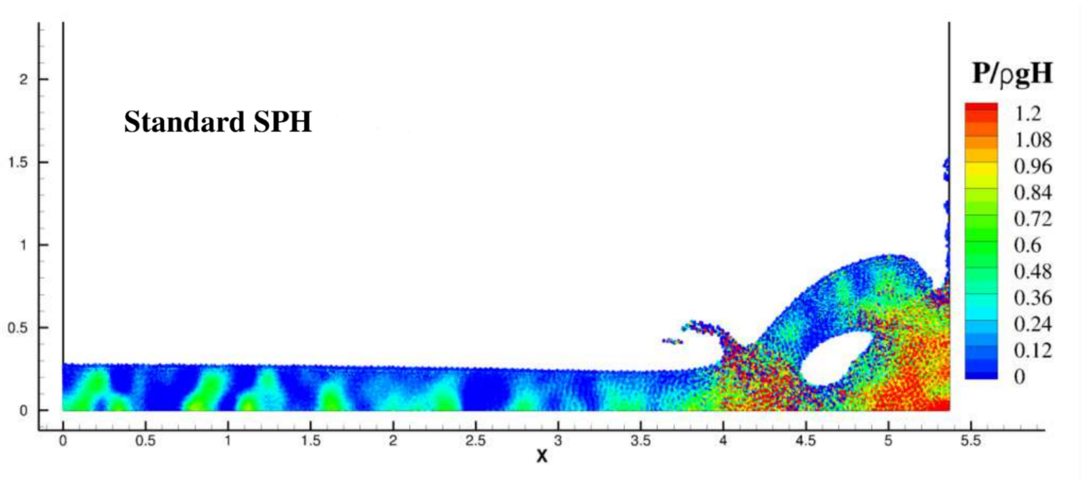


Figure 2.3: Smoothed Particle Hydrodynamics simulation: pressure field of a dam-break flow using standard SPH formulation by Salvatore Marrone / CC BY-SA

a density from a distribution of mass point particles, using a kernel weighted sum [6, 16, 18, 23]:

$$\rho(r) = \sum_{j=1}^{N_{neigh}} m_j W(|r - r_j|, h)$$

Where W is the smoothing kernel [16, 18, 23], a 3D Gaussian was preferred by Gingold and Monaghan [10, 17, 16]:

$$W(r,h) = \frac{1}{(\sqrt{\pi h})^3} exp(-\frac{r^2}{h^2})$$

To move the particles with the fluid velocity the following equation of motion can be applied [23, 18, 16]:

$$\frac{dv_i}{dt} = -\sum_j m_j \left(\frac{p_i}{p_i^2} + \frac{p_j}{p_j^2} \right) \nabla W(|r - r_j|, h)$$

Further to a physically realistic fluid simulation, SPH is also able to simulate different fluid densities, advection and dissipation [17, 23] which is beyond the scope of the intended simulations this project pertains to. In summary, while this algorithm provides a physically realistic simulation of a fluid, it is also computationally expensive which exponentially increases as the size of the simulation increases, as it would require more particles to be able to simulate a large body of water, such as an ocean, and so like Navier-Stokes would be unsuitable for this project's goals.

2.4 Gerstner Waves

Gerstner waves, also known as trochoidal waves, were discovered around 200 years ago in 1808 by Franz Josef Gerstner [5, 9, 13, 25] as an approximate solution to the fluid dynamic equations. An overlooked solution that was rediscovered independently in 1863 by Rankine [5, 13], still even in modern-day, regarded as impractical as the wave is not irrotational [5, 15] despite only being implemented in computer graphics by Fournier and Reeves in 1986 [7] An ocean surface is approximated as points that go through a circular motion as a wave passes by, note however that this solution only works for deep ocean waves [7, 13].

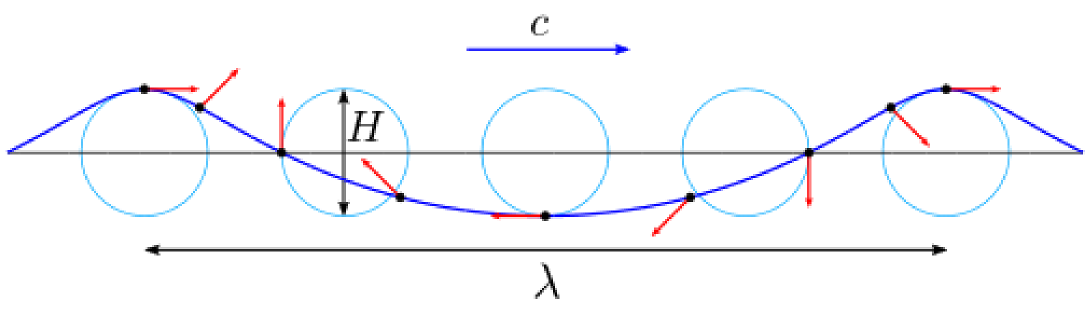


Figure 2.4: Surface elevation of a trochoidal wave in deep water (in dark blue). The wave height is denoted as H and the wavelength as λ. The particle trajectories are closed circles (in cyan), and the red vectors are the momentary velocities of the shown surface particles (black dots). The wave propagates to the right, with a phase speed c. by Kraaiennest / CC BY-SA

An undisturbed surface with a point $x_0 = (x_0, z_0)$ has a height $y = 0$, is displaced at time t by a wave with amplitude A by [7, 25]:

$$x = x_0 - (\frac{k}{k})A\sin{(k \cdot x_0 - \omega t)}y = Acos(k \cdot x_0 - \omega t)$$

Where the vector k is the wavevector, a horizontal vector that points in the direction of travel of the wave, with a magnitude k related to the wave's length λ by [25, 7]:

$$k = \frac{2\pi}{\lambda}$$

A Gerstner wave is a single sine wave, and for values where kA approaches 1 the wave forms a loop, which is not a desirable effect. To achieve a more complex wave structure, multiple Gerstner waves are calculated and summed [7, 25]:

$$x = x_0 - \sum_{i=1}^{N} (\frac{k_i}{k_i}) A_i \sin{(k_i \cdot x_0 - \omega_i t + \phi_i)}$$

$$y = \sum_{i=1}^{N} A_i cos(k_i \cdot x_0 - \omega_i t + \phi_i)$$

Gerstner waves appear suitable for this project, as they have a good level of fidelity with a relatively simple calculation that contains the components that need to be modified to affect the behaviour of the ocean at varying depths, despite being a deep or infinite depth preferred algorithm.

2.5 Fast Fourier Transform

In his 2001 paper "Simulating Ocean Water" Jerry Tessendorf [25] proposes an ocean simulation approach where multiple waves are layered together through a Fast Fourier Transform (FFT) to create a heightfield that produces more realistic waves than the Gerstner method, called 'linear waves' or 'gravity waves' which are typically fairly realistic when the weather is not too stormy [25].

FFT is an algorithm which computes the discrete Fourier transform (DFT), or it's inverse, of a sequence transforming the data from the time and space domain to frequency and vice versa. In Tessendorf's method, the input frequency is transformed via inverse DFT into the time domain, which is representative of the heightmap at a point in time, with a grid size in the range of 128 to 512 [25].

The wave height field expresses the wave height $h(x, t)$ at the horizontal position $x = (x, z)$ as the sum of sinusoids with complex, time-dependant amplitudes [25]:

$$h(x,t) = \sum_{k} \tilde{h}(k,t)e^{jkx}$$

Where t is time and k is represents the direction in which the waves are travelling, with n as the horizontal resolution and m the vertical resolution [25]:

$$k = (\frac{2\pi n}{L_x}, \frac{2\pi m}{L_x})$$

This process generates the height field at discrete points [25]:

$$x = (\frac{nL_x}{N}, \frac{mL_z}{M})$$

The wave field can then be tiled over a larger area to create an entire ocean surface, with the consequence that repeating wave patterns will occur along the field of vision [25].

This algorithm is very good, proven in many real-time and pre-computed simulations notably in video games but has a complex implementation which does not align with the project goals of making a generalised algorithm for both deep and shallow ocean water.

2.6 Implementation Choice

The algorithms explored in this chapter are some of the most well known in fluid or ocean simulation, and while others could have been explored the presented algorithms represent a cross-section of types and implementation complexity that meant that these chosen few were the best starting point, to begin with.

The implementation of this project will follow two paths. Firstly a particle-based approach will be investigated to check viability, as independent particles separate from a mesh would enable higher fidelity and realism in motion and behaviour, but with a potentially high cost that makes it unsuitable for the project goals. The objective of this approach will be to also implement it in such a way that GPU parallelisation is utilized to provide competitive performance.

Secondly, as actual ocean waves share similar characteristics and inputs (circular motion, amplitude and wavelength) to Gerstner waves, which when accounting for its relative simplicity and shared elements would allow for augmenting to provide varying depth behaviour, an enhanced Gerstner waves algorithm will be implemented, again using GPU parallelisation to ensure optimal performance.

Chapter 3

Development Framework

This chapter will define the surrounding framework that the project requires to be able to implement the chosen algorithms. It will define the project environment, APIs and libraries that will be used, along with some techniques that will be utilised to create inputs or situations for the algorithm.

3.1 Development environment

The project will be written in C++ using macOS High Sierra (10.13.6) using the JetBrains CLION IDE which will be used for debugging, while compilation will be done via the terminal with the Clang compiler, with multithreading enabled.

C++ was chosen as it is a de facto standard when it comes to graphical applications, and allows low-level access to the hardware, with minimal overhead compared with languages that are interpreted such as Java or C [12].

The machine used during all profiling will be a 27inch iMac, with a 2.7GHz Intel Core i5 processor, 8GB 1333MHz DDR3 RAM and an AMD Radeon HD 6770M 512MB graphics card, making it a baseline specification for this project that is comparative to lower-end hard hardware, allowing assumptions about performance scaling to be made when analyzing the results at a later stage.

3.2 OpenGL

The project will use OpenGL version 4.1, this is a limitation made due to Apple's macOS only supporting that version at the time of writing. OpenGL is a portable client-server API for accessing graphics hardware first released in 1994 by Silicon Graphics Computer Systems [24].

OpenGL was chosen as it is a cross-platform API [24], which allows the project to be eventually used on Microsoft Windows operating systems with minimal code refactoring, as opposed to Apple's Metal API or Microsoft's DirectX API. OpenGL is also well documented, feature-rich, and simple which makes it more suitable for use in the context of this project than more complex APIs like Vulkan.

In brief, OpenGL provides operations that allow the use of the graphics hardware of a machine but also has expectations in the format, order and use of these operations. In general, first data layouts are defined and stored along with the actual data in vertex array objects (VAO) and vertex buffer objects (VBO) respectively. Then a pipeline is created, which consists of two or many precompiled sub-programs known as shaders, with the vertex shader (processes input vertices) and the fragment shader (where a primitive is rasterized) being compulsory shader programs. Finally, the output from the pipeline is stored in a Framebuffer which, in its most basic usage, is then output to the screen [24]. The project will need to implement the following features, at a minimum, to be able to utilise the GPU of the hardware using OpenGL [24]:

- Define the layout of the data for both vertices and primitives.
- Compile shader files into shader programs and manage their lifecycle.
- Implement VAO and VBO setup and management.
- Manage entity to update and submission for rendering.

- Manage pre and post-render actions such as buffer clearing and buffer swapping.

Listing 3.1: Implemented OpenGL-related Classes

```
src/OpenGL
        BaseObjects
              Object.h
              RenderObject.h
              Renderable.h
        Buffers
              TextureBuffer.cpp
              TextureBuffer.h
              VBOLayout.h
              VertexArray.cpp
              VertexArray.h
              VertexBuffer.h
        Datastores
              Datastore.cpp
              Datastore.h
        GLLog.h
        OpenGL.hpp
        Primitives
              Cube.h
              Plane.cpp
              Plane.h
              Quad.h
              Vertex.h
        Programs
              Program.cpp
              Program.h
              ProgramManager.cpp
              ProgramManager.h
        Renderers
           Renderer.cpp
           Renderer.h
```

The project will possibly also need to implement more features depending on the needs discovered during development, for example, transform feed-back buffers, texture buffers. If advanced operations and concepts are used they will be detailed in their implementation chapters.

3.3 Additional Framework Libraries

The project will use GLFW [11] to handle window and context creation and GLAD [20] to handle OpenGL API bindings, these were both used for both cross-platform compatibility and speed of project start-up as both of these tasks are extremely complex and would have been entire projects to themselves.

For data structures and mathematics, the project will integrate GLM [21]. GLM is a proven, optimised library that is highly compatible with OpenGL while being cross-platform and cutting down on potentially hundreds of hours of work in this area.

The framework for this project will need to create various classes for similarly interacting with these libraries to the classes that wrap around OpenGL, and potentially implement functionality from the OS that these libraries do not implement already.

The project will also use OpenCV v4 which is a library aimed at real-time computer vision.OpenCV is used in this project in the demonstration scene to enable user input through their webcam, which by itself is a complex operation as the image has to be captured and processed in each frame, which makes it also an ideal to test the integration of the ocean simulation method in a non-trivial application.

3.4 User Interaction

The framework supports user interaction through two primary methods. The first is a simple keyboard and mouse event polling system allowing for a user to move the virtual camera around the world and control the look direction of the virtual camera.

The second is more complex but offers more advanced and novel forms of interaction and comes in two separate types of complex interaction. Firstly the application supports mouse picking through ray intersection, allowing the user to use the mouse to click an object in the scene and then drag the object around the scene. The method to achieve this takes the mouse coordinates which are in screen space and performs the reverse process to create a Model/View/Projection matrix to give the calculated world coordinates of the mouse cursor. These coordinates are then used as the second set of coordinates, along with the camera origin to project a ray in the scene and check intersections.

The last input method supported is webcam object tracking, implemented using OpenCV, which allows the user to use a small hand-held blue object, like a pen, which via the user's webcam is identified and tracked and it's position is used as the mouse position for ray intersection.

3.5 Perlin Noise

Perlin noise was developed in 1983 by Ken Perlin [22] to generate random values in a naturalistic pattern, with an initial use case of procedural texture generation. The algorithm is fairly straightforward, first, an N-dimensional grid of random gradient vectors is defined, the dot product between these vectors is then calculated and finally, the result is interpolated to give naturalistic noise useful in simulating slows (fluid and gas) and other natural phenomenon [22]

A prototype within this project used Perlin noise to generate a flow field which will be used to simulate the motion of the ocean. This project also uses Perlin noise as part of its demonstration scene to be able to generate interesting complex terrain, including the seafloor.

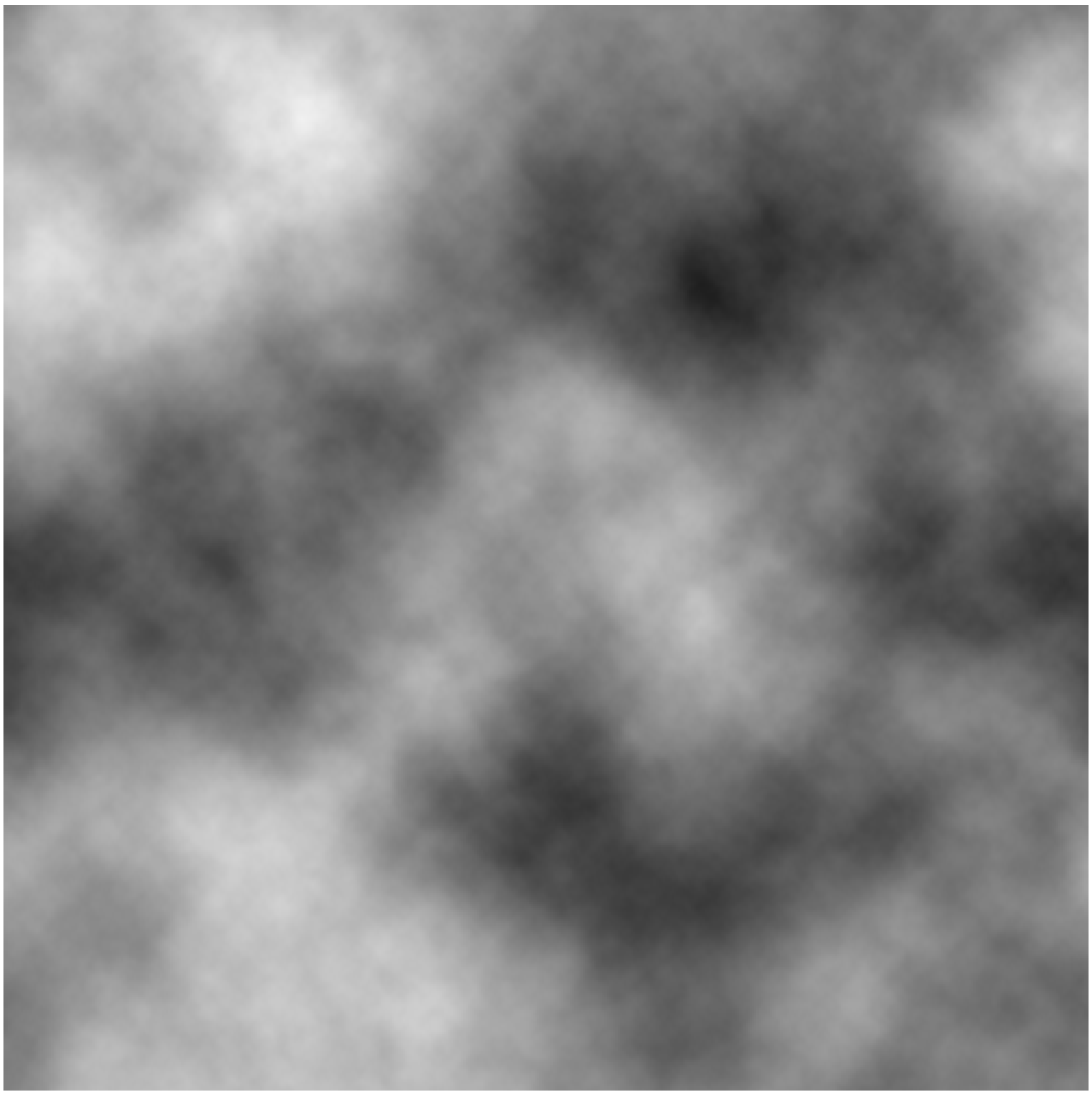


Figure 3.1: Two-dimensional Perlin noise function. The highest values are represented as white dots, and the lowest as the black ones (of course, various shades of gray represent all other values respectively) by Stevo-88 / Public domain

Chapter 4

Particle Prototype

The initial prototype was inspired by research conducted into particle-based approaches such as Navier Stokes or Smoothed Particle Hydrodynamics. While these algorithms were considered unsuitable for this project, the goal of the prototype was to simplify the algorithms presented, while maintaining fidelity and allowing for ease of customization in the simulation.

The approach taken was to extend key concepts used in particle systems such as data as textures, and transform feedback and add dynamism to the simulation through a grid of forces, a ForceGrid class, which would be used to adjust the particle positions over time.

4.1 ForceGrid Class

A class was created, designated ForceGrid, to encapsulate an array of 3-dimensional vectors which represent the force on a particle. These forces are then transferred to the GPU as an RGB texture through the use of a TBO, where on the GPU a point corresponding to the particle's position is sampled, as well as a weighted sampling around the point to be able to calculate the force to apply to the particle.

In this prototype, the ForceGrid is specifically tailored to initialize and update its values from Perlin noise, but the grid is accessible and structured in such a way that a user would be able to use any source by extending or wrapping this class, as long as they were to provide a 3-element vector for each grid entry.

4.2 Transform Feedback

Transform Feedback is a way of calculating data on the GPU [12, 24], where vertex shader or geometry shader outputs are captured into buffer objects allowing processing of data that persists over multiple frames, taking advantage of the parallel processing architecture of the GPU [12, 24].

In this prototype, we set up a single VAO which holds two VBO's with the second acting as an output location for the transform feedback, and the first supplying input data to the transform feedback as well as used as the data for the rending stage which happens later in the frame, so after each transform feedback operation the data in the two VBO are swapped so that the first VBO always contains the latest data.

Listing 4.1: Transform Feedback Execution

```
GLCheck(glEnable(GL_RASTERIZER_DISCARD));
GLCheck(glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0,
    datastore->getSubVertexBufferID()));
GLCheck(glBeginTransformFeedback(GL_POINTS));
GLCheck(glDrawArrays(GL_POINTS, 0, datastore->getPointerSize()));
GLCheck(glEndTransformFeedback());
GLCheck(glFlush());
GLCheck(glDisable(GL_RASTERIZER_DISCARD));

datastore->swapBuffers();
datastore->unbindUpdateArray();

// ....

programs->use(RENDER);
MVP();
datastore->bindRenderArray();
datastore->bindVertexBuffer();

GLCheck(glDrawArrays(GL_POINTS, 0, datastore->getPointerSize()));
```

This approach uses two buffers with the same data each frame, which gives the method a disadvantage when it comes to space, using twice the amount and also in complexity, both in the additional code for implementing it at an application level and in that it requires a separate program to be used that has additional attributes in the form of varyings, which describe the output data from the transform feedback.

4.3 Update Shader Implementation

The prototype uses only the vertex shader to perform the update calculations, without the need for the geometry shader. The code for the updated shader is found below.

Listing 4.2: Update Shader Extract

```
// Get our force value.
vec4 textureSample = SampleForceTexture();

// Calculate the drag force that will act on the particle.
vec3 drag = CalculateDrag();

// Calculate gravity, want it to act more on higher particles.
float gravityModifier = map(inPosition.y, 0.0, (SIZE_Y), 1.0, 5.0);

// Calculate the particle velocity, limit to a flat plane in this
    instance (y = 0)
vec3 calculatedVelocity = vec3(
    clamp(inVelocity.x + (textureSample.x * inMass), MIN_SPEED,
        MAX_SPEED) * dt,
    0.0,
    clamp(inVelocity.z + (textureSample.z * inMass), MIN_SPEED,
        MAX_SPEED) * dt
);

// Apply the drag to the particle.
calculatedVelocity = ApplyDrag(drag, calculatedVelocity);
```

```
// Apply the velocity to the position & check if the particle is out of
    bounds and handle.
outPosition = inPosition + calculatedVelocity;
```

As we are sampling a force from a passed in texture, the calculation is relatively simple, but there are still per-particle calculations that need to be performed, such as drag and gravity, before we integrate the velocity into the particle's position using semi-implicit Euler integration:

$$v' = a \cdot \Delta t$$

$$p' = v' \cdot \Delta t$$

Where v, a and p are velocity, acceleration and position respectively. For gravity, the prototype does not apply any gravitational force, but supports mapping a constant force of $[1 \cdots 5]$ based on the particles' y position. Drag is applied using the drag equation found in fluid dynamics:

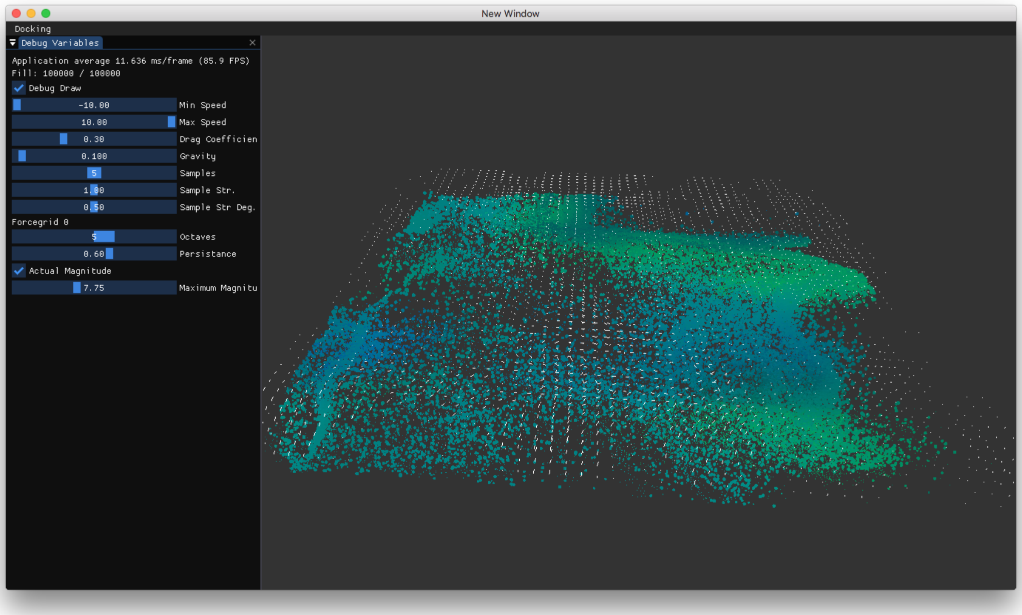
$$D = Cd \cdot \frac{\rho \cdot v^2}{2} \cdot A$$

Where Cd is the drag coefficient, ρ density, v velocity and A area.

4.4 Results

This prototype had a high degree of realism in the motion of the particles, with individual particles clustering, dispersing and moving in a naturalistic manner forming ever-changing points of interest on the potential surface over time.

However, the prototype also has key flaws which limit its intended use in performant, real-time applications. For example, as discussed previously there are space and complexity trade-offs which also result in redundant



```
Application average 11.636 ms/frame (85.9 FPS)
Fill: 100000 / 100000
[✓] Debug Draw
-10.00   Min Speed
10.00    Max Speed
0.30     Drag Coefficien
0.100    Gravity
5        Samples
1.00     Sample Str.
0.50     Sample Str Deg.
Forcegrid 0
5        Octaves
0.60     Persistance
[✓] Actual Magnitude
7.75     Maximum Magnitu
```

Figure 4.1: Simulation running with 100,000 particles. White lines visualize the ForceGrid class data

processing with each vertex being processed in the GPU twice, once for movement and positioning and again for rendering, with the consequence being that the simulation takes too long per frame to be suitable in performant real-time applications as seen in table 4.1.

Table 4.1: Profiling Results For The Particle Application (100,000 Particles)

	Average	Minimum	Maximum
MS/Frame	12.25	10.15	3.24
FPS	81.63	98.52	308.64

It is worth noting, however, that this result is for 100,000 particles, far beyond what would be required in most situations, furthermore, in many real-time applications some aspects are processed at fixed intervals [12] rather than each frame, so potential is there for efficiency improvements in this method.

However, this is still unsuitable for the goals of the project as the prototype rendered points to represent each individual vertex, so to be used in simulations a mesh must be constructed to form the ocean's surface each frame due to the nature of a vertex in the simulation essentially being able to go with the forces in the ForceGrid class, so no reliable method of linking each other together into triangles would exist except for a search on each vertex for its k nearest neighbours.

Chapter 5

Extended Gerstner Waves

As discussed after concluding the findings of the research presented in chapter 2, it was decided to proceed with a modified Gerstner wave algorithm that would enable it to handle both deep-water and shoreline simulations and a good-enough fidelity for the intended application.

This chapter will outline the key features of the implementation, including a surrounding structure to input and output data as well as the extension to the base Gerstner wave algorithm to enable it to apply to both deep and shallow ocean scenes.

5.1 Water

The water is composed of a plane of vertices whose position remains static in memory, but its vertices are displaced in the vertex shader stage to create the ocean surface. The waterplane is generated on application startup, generating vertices based on a specified size and resolution, to enable customization of fidelity, a sample of the code for the creation of the plane is shown in the following code block:

Listing 5.1: Water Plane Vertex Index Creation

```
for (float z = 0; z < cellsZ; ++z) {
    for (float x = 0; x < cellsX; ++x) {
        int xp = position.x + x * resolution;
        int zp = position.z + z * resolution;
        Vertex vertex = {{xp, position.y, zp}, COLOUR, {0.0f, 0.0f,
            0.0f}};
        vertices.push_back(vertex);
    }
}


for (int z = 0; z < cellsZ - 1; ++z) {
    int Z = z * (cellsX); // Row
    int ZN = (z + 1) * (cellsX); // Next Row

    for (int x = 0; x < cellsX - 1; ++x) {
        indices.push_back(Z + x); // A
        indices.push_back(ZN + x + 1); // C
        indices.push_back(ZN + x); // D
        indices.push_back(ZN + x + 1); // C
        indices.push_back(Z + x); // A
        indices.push_back(Z + x + 1); // B
    }
}
```

The vertices of the plane are then displaced by layering multiple Gerstner waves, which simulate fluid at infinite depth. To add wave data to the simulation each wave is passed into the GPU as a uniform variable in the form of a 4 element vector with x-direction, z-direction, wavelength and steepness represented as the x, y, z and w components respectively. This implementation supports 3 individual wave forces.

To calculate the Gerstner wave a function is used to return a new position for the vertex:

Listing 5.2: Gerstner Wave Function Within Vertex Shader

```
float k = 2 * PI / wave.w; // Wavelength
float c = sqrt(9.8 / k); // Phase speed
vec2 d = normalize(wave.xy); // Normalized wave direction
float f = k * (dot(d, p.xz) - c * time); // Wave slope
float a = steepness / k; // Amplitude
```

```
//

return vec3(
    d.x * (a * cos(f)),
    a * sin(f),
    d.y * (a * cos(f)) + (sin(time))
);
```

This is used on all input waves, with each result being added to the original vertex position to get it's final position once all waves have been processed.

5.2 Height Field (Terrain)

The project, as a prerequisite of the simulation of coastal and deep waters, needs both a method of displaying terrain and also a method of tracking a vertex's height above the terrain, so to do this a height field was implemented which allowed both terrain generation and depth calculation.

While a height field can be calculated in many ways, in this implementation Perlin noise is used to create naturalistic, random terrain, which is limited to a certain range based on the X/Y positions to be able to slope from the sea-floor through to mountain tops.

The vertices are displaced according to their height above the terrain, so a heightmap and displacement map are created which contain the height data of the terrain and objects in the scene respectively which is passed to the GPU as a sample, where it is then sampled to get the correct height data.

This height data is then converted into an OpenGL texture through the use of a Texture Buffer Object which is then used in the GPU to sample heights to get the depth of the ocean at that point, as shown below:

Listing 5.3: Terrain Height Map Sampling

```
float heightSample = texelFetch(THM, HMIndex(int(inPosition.x),
    int(inPosition.z))).x;
```

```
float depth = waterlevel - heightSample;
float normalizeddepth = Map(depth, 0.0, waterlevel, 0.0, 1.0);
float depthfactor = Map(depth, 0.0, waterlevel, max, min);
```

In real-time applications there are also objects that interact with the world, this could be structured within the water or vehicles such as boats. To account for these objects a second height field is created, the scene height field, which extends the terrain heightfield by also adding the object's height to the height field. This is then sent to the GPU as a texture where it can be sampled to get object height data which can be used, in this instance to implement a basic level of response from the ocean to the object.

5.3 Extending Gerstner Waves

The solution in this project extends the Gerstner wave algorithm by calculating a depth factor from the sampled depth as noted in the previous section. This alters the wavelength which then, in turn, affects the amplitude. At high depths, the depth will exceed a certain threshold, and so the factor will be 1.0 which means the Gerstner wave is unaltered, but as the depth decreases and the factor decreases the waves will gradually transition into shoreline waves. This is actually very simple to implement, by modifying some parts of the Gerstner wave function in the vertex shader:

Listing 5.4: Extension Applied To Existing Gerstner Function

```
float wavelength = wave.w * factor;
float k = 2 * PI / wavelength; // Wavelength

//

return vec3(
    d.x * (a * cos(f)),
    a * sin(f),
    d.y * (a * cos(f)) + (ndepth * sin(time))
);
```

This depth factor can also be used for secondary effects like determining fast-moving breaking waves for shading a shoreline foaming effect, which is completed outside of the Gerstner wave function and uses the scene height map to integrate similar effects around objects:

Listing 5.5: Scene Height Map Sampling

```
float sceneHeightSample = texelFetch(HM, HMIndex(int(p.x), int(p.z))).x;
float col = 1 - Map(p.y, 50.0, 100.0, 1.0, Map(depth, 0.0, waterlevel,
    1.0, 0.0));
passColour = vec4(col, col, col, Map(waterlevel - sceneHeightSample,
    0.0, waterlevel, 0.5, 1.0));

if (depth <= 0.0) passColour.a = 0.0;
if ((p.y - sceneHeightSample) < shorethreshold || p.y >= foamHeight)
    passColour = vec4(1.0, 1.0, 1.0, Map(waterlevel - sceneHeightSample,
        0.0, waterlevel, 0.8, 1.0));
```

5.4 Results

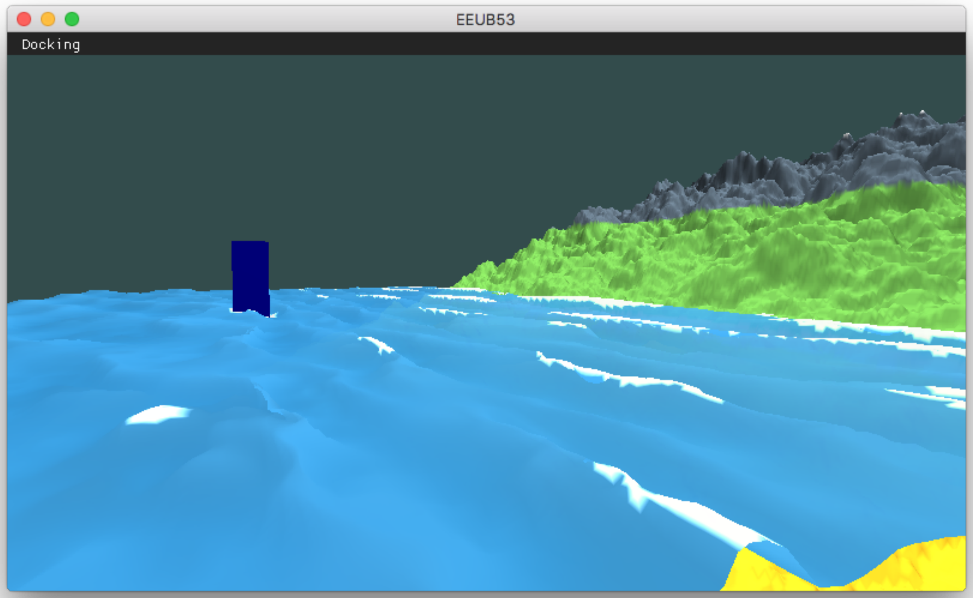


Figure 5.1: Simulation running viewed from a near distance at aside on angle. Displacement of the plane is visible from this angle.

The result of this algorithm can be measured in two ways, through visual fidelity and through performance. Visually the algorithm provides a degree of realism when viewed from mid to far distances as shown in 5.2, and surprisingly holds up when viewed from viewed closely from an unintended angle as shown in figure 5.1. Figure **??** also helps to demonstrate the effect that multiple layered Gerstner waves have on a plane's vertices that give the illusion of an ocean surface.

Table 5.1: Profiling results for the Extended Gerstner Waves Application

	Baseline	CPU (Sequential)	GPU (Parallel)
MS/Frame Average	2.57	125.94	7.18
MS/Frame Minimum	1.62	103.96	1.66
MS/Frame Maximum	188.24	358.75	225.62
FPS Average	389.10	7.97	139.27
FPS Minimum	5.31	2.81	4.43
FPS Maximum	617.20	10.45	602.40

Performance for this algorithm is also suitable for real-time applications even in its naive state, as shown in table 5.1 when compared to the baseline, which is a scene consisting only of the framework and the terrain, compares favourably with minimum and maximum time taken per frame, and within expectations for the average time taken per frame when accounting for the naive state of the implementation.

The algorithm itself is also highly favourable to parallelization, as demonstrated by comparing the CPU (Sequential) and GPU (Parallel) implementations, with the GPU implementation performing 62

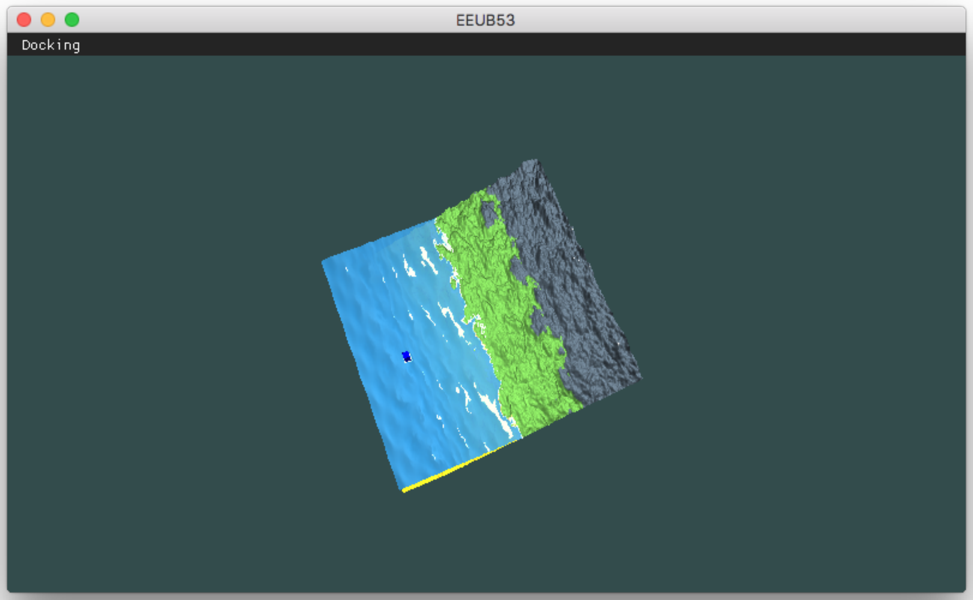


Figure 5.2: Simulation viewed from a far distance demonstrating the fidelity when used as a background element.

Chapter 6

Conclusion

This project set out to create a generalized algorithm for both deep and shallow ocean water simulations at the middle to far distances. As part of the project two implementations were produced, one based on particle flow algorithms and the second an extension of Gerstner waves.

Out of the two, the particle implementation had greater potential for fidelity and was certainly more visually impressive than the extended Gerstner waves algorithm, but the lower performance and lack of a tangible rendered surface led this algorithm to be unsuitable for the project's objectives.

The extended Gerstner wave algorithm than was chosen as the more suitable algorithm matching the objectives of the project, as the performance event in the naive implementation was acceptable for real-time applications while having acceptable fidelity for middle-far distance camera positioning and an acceptable base performance profile despite the naive implementation.

6.1 Future Work

While the extended Gerstner wave algorithm performs fairly well in the current context as seen in table 5.1, there are many improvements that could potentially be made to reduce the cost per frame. This chapter will look at some potential future enhancements that could be made to the algorithm. As the algorithm's implementation is highly naive, the main focus of the future work discussed will be on performance improvements, which are key to allow even further work on areas such as fidelity.

There are three definitive areas of optimization that would be immediately beneficial to the algorithm:

1. Mesh and calculation level of detail (LOD),
2. Object displacement integration; and
3. Bounding algorithm update to a fixed physics step.

6.1.1 Level Of Detail

The current implementation of the algorithm is considerably naive, a fixed resolution mesh where each vertex is operated on in the same way. This naivety affords opportunities for two possible levels of detail LOD related improvements. LOD is a technique where a lower resolution model or lower accuracy algorithm is used when further away from the viewer where details would occupy fewer pixels, which reduces wasted computation on detail that would not be able to be seen [14, 12].

First, as the implemented algorithm's key component is sine and cosine waves layered along with a function of time. In the implementation, we already integrate the camera position to the shaders for the purpose of lighting calculations, so it would be a wise step to also integrate a function of distance from the camera.

In this improvement, the algorithm could first check the distance and reduce the quality of the algorithm as the distance of a vertex from the camera increases. This could take the form of a stepped reduction in calculation granularity of the elements, such that at the lowest level of detail vertices would be moved with simple sine or cosine functions with little to no extra calculation, layered to retain some interesting surface topology, and further to this a look-up table of values of sine a cosine from $0 \cdots 360$ could be hardcoded in the shader which would reduce these calculations just to lookups, where the input angle is floored, losing granularity but at a great performance increase.

The second level of detail improvement would apply to the ocean surface mesh itself. Currently, the mesh is a fixed resolution mesh with no levels of detail, which if added means that the mesh would be higher resolution closer to the camera, meaning more vertices, but at the same time points on the mesh further from the camera would contain fewer vertices per mesh segment.

This would involve adding a tessellation stage to the current shader program [14]. Mesh tessellation means further subdividing the mesh to increase the resolution of it [12], while using the camera position, choosing where and what radius on the mesh to tessellate. This allows for the complex calculations in the vertex shader to only be performed on lower resolution meshes, where the tessellation stage can interpolate the positions to create the aforementioned smooth mesh segments [24].

6.1.2 Object Displacement

In the current implementation, there is support for interactable objects through the generation of a heightmap from the height of the object above the water plane's height, but this is limited to the water surrounding the object being recoloured while to imitate foaming water caused by the displacement.

To increase the visual fidelity the height map could be extended to include a displacement map, where each element in the map would contain a vector representative of the force outward from an object's centre. This could be pre-calculated by the object itself to save having to repeat the calculation.

In the shader stage this displacement could be passed in as a texture, and then sampled in much the same way as the heightmap and used to displace the vertices of the plane, potentially by adding a geometry shader stage into the pipeline where additional triangles could be added to wrap around the area of displacement and also fix any gaps created by displacement [12, 24].

This development, however, focuses solely on visual fidelity and would bring a relatively heavy cost, so would be a situational addition, particularly in certain mid to far distance applications where this level of fidelity is relatively wasted as would be unseen by the user, although this could be controlled by a level of detail system as described in the previous section.

6.1.3 Bounding To A Fixed Step

Many real-time applications such as games or physics simulations utilize a fixed time step [12], typically when calculating physics, where a fixed amount of updates are used, for example, 15 per second, which run separately from other updates, making them independent of frame rate, and allows for consistent, accurate results.

By utilizing this process the algorithm would be able to reduce the number of times it updates, for example by assuming that an application is rendering at 60 FPS and each iteration of the algorithm in its current form adds 4.61MS/F then bounding the algorithm to 30 FPS means that we would amortize the cost of the calculation over the 2 frames instead of a just one, improving the frame-rate over a period of time, while also handling sensitivity to time step change which then guarantees stability to the simulation [28, 14].

This solution would be trivial to implement, as the calculations within the shader utilize the current time as a parameter of sine and cosine, and also as a modifying value itself to control the position of the vertex at that point in time, which in the current implementation is each frame. Bounding to a fixed time step would simply require tracking of the elapsed time in the run loop of the application, and when a fixed threshold is reached then the program instructs the GPU to update and render the waves.

The potential downside to this method would be that larger fixed steps would lose some of the smoothness that is present with smaller time increments, however, this could be fixed by providing an offset time value as an input to the shader, either by a separate time value which increases at a slower rate, or interpolating between the previous and current time values, either solution would only require the minimum of extra space and complexity overhead [12].

References

[1] M. D. Adeyemo, ‘Wave transformation and velocity fields in the breaker zone’, 1969 (pp. 4, 5).

[2] F. Biesel, ‘Study of wave propagation in water of gradually varying depth’, in *Gravity Waves*, Nov. 1952, p. 243 (pp. 1, 4, 5).

[3] R. Bridson, *Fluid Simulation for Computer Graphics*, ser. Ak Peters Series. Taylor & Francis, 2008, ISBN: 9781568813264. [Online]. Available: https://books.google.co.uk/books?id=gFI8y87VCZ8C (pp. 6, 7).

[4] A. Chorin and J. Marsden, *A mathematical introduction to fluid mechanics*, ser. Universitext (1979). Springer-Verlag, 1979, ISBN: 9783540904069. [Online]. Available: https://books.google.co.uk/books?id=R17vAAAAMAAJ (p. 6).

[5] A. Craik, ‘The origins of water wave theory’, *Annual Review of Fluid Mechanics*, vol. 36, pp. 1–28, Jan. 2004 (p. 9).

[6] M. Desbrun and M.-P. Gascuel, ‘Smoothed particles: A new paradigm for animating highly deformable bodies’, in *Proceedings of the Eurographics Workshop on Computer Animation and Simulation ’96*, Poitiers, France: Springer-Verlag, 1996, pp. 61–76, ISBN: 3211828850 (pp. 7, 8).

[7] A. Fournier and W. T. Reeves, ‘A simple model of ocean waves’, in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’86, New York, NY, USA: Association for Computing Machinery, 1986, pp. 75–84, ISBN: 0897911962. DOI: 10.1145/15922.15894. [Online]. Available: https://doi.org/10.1145/15922.15894 (pp. 1, 9, 10).

[8] M. K. Gaughan and P. D. Komar, ‘The theory of wave propagation in water of gradually varying depth and the prediction of breaker type and height’, *Journal of Geophysical Research (1896-1977)*, vol. 80,

no. 21, pp. 2991–2996, 1975. DOI: 10.1029/JC080i021p02991. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/JC080i021p02991. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JC080i021p02991 (pp. 1, 4).

[9] F. Gerstner, ‘Theorie der wellen’, *Annalen der Physik*, vol. 32, no. 8, pp. 412–445, 1809. DOI: 10.1002/andp.18090320808. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/andp.18090320808. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.18090320808 (p. 9).

[10] R. A. Gingold and J. J. Monaghan, ‘Smoothed particle hydrodynamics: theory and application to non-spherical stars.’,, vol. 181, pp. 375–389, Nov. 1977. DOI: 10.1093/mnras/181.3.375 (pp. 7, 8).

[11] *Graphics library framework*, 2020. [Online]. Available: https://www.glfw.org/ (p. 16).

[12] J. Gregory and J. Lander, *Game Engine Architecture*, ser. Ak Peters Series. Taylor & Francis, 2009, ISBN: 9781568814131. [Online]. Available: https://books.google.co.uk/books?id=LJ20tsePKk4C (pp. 1, 13, 20, 23, 33–36).

[13] D. Henry, ‘On gerstner’s water wave’, *Journal of Nonlinear Mathematical Physics Volume Supplement*, vol. 15, pp. 87–95, Aug. 2008. DOI: 10.2991/jnmp.2008.15.s2.7 (pp. 1, 9).

[14] T. Kellomäki, *Large-Scale Water Simulation in Games*, English, ser. Tampere University of Technology. Publication. Tampere University of Technology, Dec. 2015, Awarding institution:Tampere University of Technology, ISBN: 978-952-15-3643-4 (pp. 1, 6, 7, 33–35).

[15] G. J. Komen, L. Cavaleri, M. Donelan, K. Hasselmann, S. Hasselmann and P. A. E. M. Janssen, *Dynamics and Modelling of Ocean Waves*. 1996 (pp. 4, 5, 9).

[16] G. Liu and M. Liu, *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. Jan. 2003. DOI: 10.1142/5340 (p. 8).

[17] J. J. Monaghan, ‘Smoothed particle hydrodynamics’, *Reports on Progress in Physics*, vol. 68, no. 8, pp. 1703–1759, Jul. 2005. DOI: 10.1088/0034-4885/68/8/r01. [Online]. Available: https://doi.org/10.1088%2F0034-4885%2F68%2F8%2Fr01 (pp. 7, 8).

[18] ——, ‘Smoothed particle hydrodynamics’, *Annual Review of Astronomy and Astrophysics*, vol. 30, no. 1, pp. 543–574, 1992. DOI: 10.1146/annurev.aa.30.090192.002551 (pp. 7, 8).

[19] M. Müller, D. Charypar and M. Gross, ‘Particle-based fluid simulation for interactive applications’, in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA ’03, San Diego, California: Eurographics Association, 2003, pp. 154–159, ISBN: 1581136595 (pp. 1, 7).

[20] *Opengl loading library*, 2020. [Online]. Available: https://glad.dav1d.de/ (p. 16).

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

[22] K. Perlin, ‘An image synthesizer’, *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 287–296, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325247. [Online]. Available: Perlin (p. 17).

[23] D. Price, ‘Smoothed particle hydrodynamics: Things i wish my mother taught me’, Nov. 2011 (pp. 7, 8).

[24] D. Shreiner, G. Sellers, J. M. Kessenich and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th. Addison-Wesley Professional, 2013, ISBN: 0321773039 (pp. 14, 20, 34).

[25] J. Tessendorf, ‘Simulating ocean water’, 2004 (pp. 9–11).

[26] J. Tompson, K. Schlachter, P. Sprechmann and K. Perlin, ‘Accelerating eulerian fluid simulation with convolutional networks’, in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML’17, Sydney, NSW, Australia: JMLR.org, 2017, pp. 3424–3433 (pp. 1, 6, 7).

[27] *Wave energy and wave changes with depth*, 2020. [Online]. Available: https://manoa.hawaii.edu/exploringourfluidearth/physical/waves/wave-energy-and-wave-changes-depth (pp. 1, 4–6).

[28] Youquan Liu, Xuehui Liu and Enhua Wu, ‘Real-time 3d fluid simulation on gpu with complex obstacles’, in *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, 2004, pp. 247–256 (pp. 1, 7, 35).

Appendix A

Tidal Application README

There are two projects contained within this directory. They both contain precompiled executables which are intended to be ran from scripts also contained within this folder so that the intended path to resources is maintained.

A.1 Tidal: Extended Gerstner Waves

From the root directory of this project run the ’Build.sh’ script, this will give multiple options, but as the project comes with precompiled binaries it is expected that option 0 will suffice. If for any reason you need to re-build the project then either option 1 or 2 will build with the former also launching the program.

A.1.1 Controls

W/S A/D: Move along the FORWARD/BACKWARD LEFT/RIGHT axes respectively. Q/E: Move along the UP axis. Right Mouse Button + Drag: Rotate Camera P: Toggle input modes (Mouse -> Camera)

1: Default perspective camera view. 2: Top-down camera view

Mouse input mode (default): Hover over an object and drag while holding the left mouse button. Camera input mode: Using a BLUE object hold within view of the camera, objects will be automatically ’picked’

A.2 Particle

From the ’Particle Prototype Complete Code’ directory of this project run the ’Build.sh’ script, this will give multiple options, but as the project comes with precompiled binaries it is expected that option 0 will suffice. If for any reason you need to re-build the project then either option 1 or 2 will build with the former also launching the program.

A.2.1 Controls

W/S A/D: Move along the X Z axes respectively. Q/E: Move along the Y axis. Mouse Wheel Scroll: Move along the forward/backward direction. Left Mouse Button + Drag: Rotate Camera

Appendix B

Project Source Trees

```
Tidal
 Resources
    Shaders
        Debug
           debug.basic.fragment.glsl
           debug.basic.vertex.glsl
        Mesh
           mesh.fragment.glsl
           mesh.vertex.glsl
           terrain.fragment.glsl
           terrain.vertex.glsl
        Water
            water.fragment.glsl
            water.vertex.glsl
 imgui.ini
 src
     ComputerVision
        WebcamCursor.h
        WebcamObjectTracker.cpp
        WebcamObjectTracker.h
     Core
        App.cpp
        App.h
        GUI
           GUILayer.cpp
```

```
      GUILayer.h
   Window
       Camera.cpp
       Camera.h
       InputCallbacks.h
       InputHandler.cpp
       InputHandler.h
       MouseSelector.cpp
       MouseSelector.h
       Window.cpp
       Window.h
Headers
   ImGUI.hpp
   glad.cpp
   imgui
   stb_image.h
OpenGL
   BaseObjects
      Object.h
      RenderObject.h
      Renderable.h
   Buffers
      TextureBuffer.cpp
      TextureBuffer.h
      VBOLayout.h
      VertexArray.cpp
      VertexArray.h
      VertexBuffer.h
   Datastores
      Datastore.cpp
      Datastore.h
   GLLog.h
   OpenGL.hpp
   Primitives
```

```
      Cube.h
      Plane.cpp
      Plane.h
      Quad.h
      Vertex.h
   Programs
      Program.cpp
      Program.h
      ProgramManager.cpp
      ProgramManager.h
   Renderers
       Renderer.cpp
       Renderer.h
PCHeader.cpp
PCHeader.h
Simulation
   Pillar.cpp
   Pillar.h
   SceneHeightMap.cpp
   SceneHeightMap.h
   Terrain.cpp
   Terrain.h
   Water.cpp
   Water.h
Utility
   Log.cpp
   Log.h
   Profiler.h
   Random
       Perlin.cpp
       Perlin.h
       Random.cpp
       Random.h
main.cpp
```

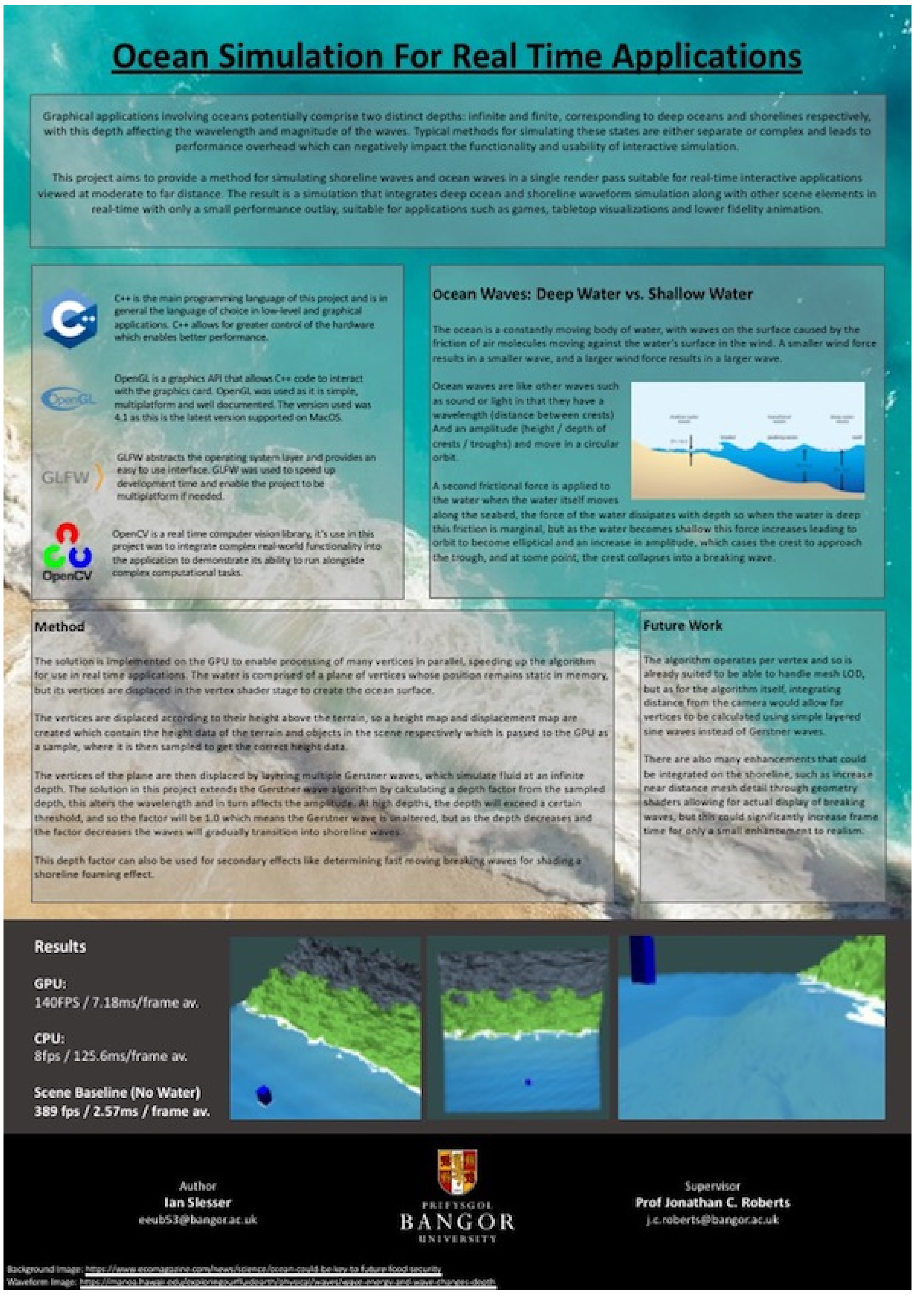
```
Particle
 Resources
    Definitions
       Error.def
       Input-Camera.def
    Shaders
       Debug
          debug.fragment.glsl
          debug.geometry.glsl
          debug.vertex.glsl
       mesh.fragment.glsl
       mesh.vertex.glsl
       particle_update.vertex.glsl
       point.fragment.glsl
       point.vertex.glsl
    config.ini
 imgui.ini
 src
     Core
        App.cpp
        App.h
        Camera
           Camera.cpp
           Camera.h
           CameraManager.cpp
           CameraManager.h
        GUI
           GUILayer.cpp
           GUILayer.h
        GlobalConfiguration.cpp
        GlobalConfiguration.h
        Input
           InputManager.cpp
           InputManager.h
```

```
   Window
       Window.cpp
       Window.h
Headers
   Colors.hpp
   ImGUI.hpp
   OpenGL.hpp
   glad.cpp
   imgui
      imconfig.h
      imgui.cpp
      imgui.h
      imgui_demo.cpp
      imgui_draw.cpp
      imgui_impl_glfw.cpp
      imgui_impl_glfw.h
      imgui_impl_opengl3.cpp
      imgui_impl_opengl3.h
      imgui_internal.h
      imgui_widgets.cpp
      imstb_rectpack.h
      imstb_textedit.h
      imstb_truetype.h
   stb_image.h
OpenGL
   Buffers
      TextureBuffer.cpp
      TextureBuffer.h
      VBOLayout.h
      VertexArray.cpp
      VertexArray.h
      VertexBuffer.h
   Datastores
      Datastore.cpp
```

```
        Datastore.h
        DebugDatastore.cpp
        DebugDatastore.h
        MeshDatastore.cpp
        MeshDatastore.h
    GLLog.h
    Primitives
        Cube.h
        Quad.h
        Vertex.h
    Programs
        DebugProgram.cpp
        DebugProgram.h
        Program.cpp
        Program.h
        ProgramManager.cpp
        ProgramManager.h
        UpdateProgram.cpp
        UpdateProgram.h
    Renderers
        DebugRenderer.cpp
        DebugRenderer.h
        MeshRenderer.cpp
        MeshRenderer.h
        Renderer.cpp
        Renderer.h
    Texture.cpp
    Texture.h
PCHeader.cpp
PCHeader.h
Particle
    Emitter
        Emitter.cpp
        Emitter.h
        EmitterManager.cpp
        EmitterManager.h
    Particle.cpp
    Particle.h
Simulation
    ForceGrid.cpp
    ForceGrid.h
    ForceGridData.h
    Simulation.cpp
    Simulation.h
    Vortex.cpp
    Vortex.h
Utility
    ConfigReader.h
    Log.cpp
    Log.h
    Math
        Mathf.cpp
        Mathf.h
    Profiler.h
    Random
        Perlin.cpp
        Perlin.h
        Random.cpp
        Random.h
    Timer.cpp
    Timer.h
main.cpp
```

Appendix C

CS Expo 2020 Poster



Ocean Simulation For Real Time Applications

Graphical applications involving oceans potentially comprise two distinct depths: infinite and finite, corresponding to deep oceans and shorelines respectively, with this depth affecting the wavelength and magnitude of the waves. Typical methods for simulating these states are either separate or complex and leads to performance overhead which can negatively impact the functionality and usability of interactive simulation.

This project aims to provide a method for simulating shoreline waves and ocean waves in a single render pass suitable for real-time interactive applications viewed at moderate to far distance. The result is a simulation that integrates deep ocean and shoreline waveform simulation along with other scene elements in real-time with only a small performance outlay, suitable for applications such as games, tabletop visualizations and lower fidelity animation.

C++ is the main programming language of this project and is in general the language of choice in low-level and graphical applications. C++ allows for greater control of the hardware which enables better performance.

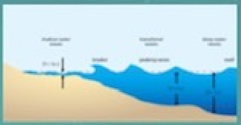
OpenGL is a graphics API that allows C++ code to interact with the graphics card. OpenGL was used as it is simple, multiplatform and well documented. The version used was 4.1 as this is the latest version supported on MacOS.

GLFW abstracts the operating system layer and provides an easy to use interface. GLFW was used to speed up development time and enable the project to be multiplatform if needed.

OpenCV is a real time computer vision library, it's use in this project was to integrate complex real-world functionality into the application to demonstrate its ability to run alongside complex computational tasks.

Ocean Waves: Deep Water vs. Shallow Water

The ocean is a constantly moving body of water, with waves on the surface caused by the friction of air molecules moving against the water's surface in the wind. A smaller wind force results in a smaller wave, and a larger wind force results in a larger wave.



Ocean waves are like other waves such as sound or light in that they have a wavelength (distance between crests) And an amplitude (height / depth of crests / troughs) and move in a circular orbit.

A second frictional force is applied to the water when the water itself moves along the seabed, the force of the water dissipates with depth so when the water is deep this friction is marginal, but as the water becomes shallow this force increases leading to orbit to become elliptical and an increase in amplitude, which cases the crest to approach the trough, and at some point, the crest collapses into a breaking wave.

Method

The solution is implemented on the GPU to enable processing of many vertices in parallel, speeding up the algorithm for use in real time applications. The water is comprised of a plane of vertices whose position remains static in memory, but its vertices are displaced in the vertex shader stage to create the ocean surface.

The vertices are displaced according to their height above the terrain, so a height map and displacement map are created which contain the height data of the terrain and objects in the scene respectively which is passed to the GPU as a sample, where it is then sampled to get the correct height data.

The vertices of the plane are then displaced by layering multiple Gerstner waves, which simulate fluid at an infinite depth. The solution in this project extends the Gerstner wave algorithm by calculating a depth factor from the sampled depth, this alters the wavelength and in turn affects the amplitude. At high depths, the depth will exceed a certain threshold, and so the factor will be 1.0 which means the Gerstner wave is unaltered, but as the depth decreases and the factor decreases the waves will gradually transition into shoreline waves.

This depth factor can also be used for secondary effects like determining fast moving breaking waves for shading a shoreline foaming effect.

Future Work

The algorithm operates per vertex and so is already suited to be able to handle mesh LOD, but as for the algorithm itself, integrating distance from the camera would allow far vertices to be calculated using simple layered sine waves instead of Gerstner waves.

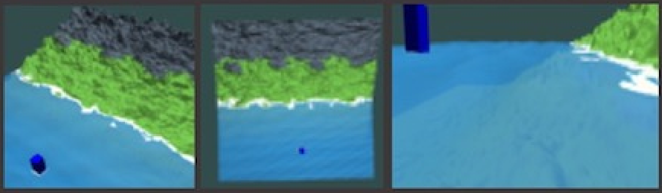
There are also many enhancements that could be integrated on the shoreline, such as increase near distance mesh detail through geometry shaders allowing for actual display of breaking waves, but this could significantly increase frame time for only a small enhancement to realism.

Results

GPU:
140FPS / 7.18ms/frame av.

CPU:
8fps / 125.6ms/frame av.

Scene Baseline (No Water)
389 fps / 2.57ms / frame av.



Author
Ian Slesser
eeub53@bangor.ac.uk

PRIFYSGOL BANGOR UNIVERSITY

Supervisor
Prof Jonathan C. Roberts
j.c.roberts@bangor.ac.uk

Background Image: https://www.ecomagazine.com/news/science/ocean-could-be-key-to-future-food-security
Waveform Image: https://manoa.hawaii.edu/exploringourfluidearth/physical/waves/wave-energy-and-wave-changes-depth