F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Bachelor's Thesis

Procedural Water Waves

Viktor Shubert
Study programme: Open Informatics
Specialization: Computer Games and Graphics

May 2024
Supervisor: Ing. Jaroslav Sloup

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Shubert** Jméno: **Viktor** Osobní číslo: **499315**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Procedurální generování vln na vodní hladině

Název bakalářské práce anglicky:

Procedural Water Waves

Pokyny pro vypracování:

Proveďte rešerši metod procedurálního generování vln na vodní hladině používaných v počítačové grafice. Detailně se seznamte s metodou [2] a jejím rozšířením [3].
Navrhněte a naimplementujte aplikaci simulující vlny na vodní hladině pomocí metody využívající diskretizaci funkce vyjadřující amplitudy vln v závislosti na pozici, vlnovém vektoru a čase [2]. Aplikaci doplňte o uživatelské rozhraní umožňující interaktivní změnu parametrů ovlivňujících způsob diskretizace funkce amplitudy.
Vyjděte z veřejně dostupné CPU implementace zvolené metody [4] a přesuňte podstatné části simulace na GPU tak, aby simulace běžela v reálném čase. Implementaci proveďte v OpenGL s využitím compute shaderů.
Funkčnost implementovaného řešení ověřte alespoň na třech scénách různé geometrické složitosti. Zhodnoťte vliv počtu vzorků použitých pro diskretizaci funkce amplitudy na výslednou kvalitu vygenerovaných vln a rychlost simulace i vykreslování.

Seznam doporučené literatury:

[1] T. Kellomäki: Fast Water Simulation Methods for Games. Computers in Entertainment (CIE), Vol.16, No.1, p.1-14, 2017.
[2] S. Jeschke, T. Skřivan, M. Müller-Fischer, N. Chentanez, M. Macklin, C. Wojtan: Water Surface Wavelets. ACM Transactions on Graphics, Vol.37, No.4, p.1-13, 2018.
[3] S. Jeschke, C. Hafner, N. Chentanez, M. Macklin, M. Müller-Fischer, C. Wojtan: Making Procedural Water Waves Boundary-aware. Computer Graphics Forum, Vol.39, No.8, p.47-54, 2020.
[4] Water Surface Wavelets webpage: https://visualcomputing.ist.ac.at/publications/2018/WSW/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jaroslav Sloup Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **15.02.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

Ing. Jaroslav Sloup podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

Acknowledgement / Declaration

First of all, I would like to thank my supervisor Ing. Jaroslav Sloup for his patience and helpful pieces of advice throughout my work on the the thesis.

I would also like to express gratitude to my family and friends for their encouragement.

I hereby declare that this thesis represents my own work and that I have cited all the sources in accordance with the Methodical guideline no. 2/2024 for adhering to ethical principles when elaborating an academic final thesis.

Prague, May 24, 2024

.....

Abstrakt / Abstract

Tato práce se zaměřuje na analýzu různých metod simulace vodní plochy a implementaci jedné z nich. Začíná teoretickým porovnáním různých způsobů simulace vody, zhodnocením jejich výhod a nevýhod. Vybraná metoda je potom vysvětlena podrobněji a zároveň je upřesněna motivace, která stála za jejím výběrem. Poté je v kapitole o návrhu vyložena struktura aplikace, zatímco dále jsou vysvětleny podrobnosti o implementaci. Nakonec jsou porovnány různé konfigurace, změřen jejich výkon a zhodnocena jejich vizuální věrnost.

Klíčová slova: procedurální vlny, fyzikální simulace, výpočty na GPU, návrh aplikace, implementace aplikace

Překlad titulu: Procedurální generování vln na vodní hladině

This thesis focuses on analyzing different methods of water surface simulation and implementing one of them. It starts with the theoretical comparison between different ways of simulating water, evaluating their pros and cons. The chosen method is then explained more in-depth, while also elaborating on the motivation behind the choice. After that, in the design chapter, the application structure is laid out, while the details of the implementation are explained next. Finally, different configurations are compared, measuring their performance and evaluating their visual fidelity.

Keywords: procedural waves, physical simulation, GPU computation, application design, application implementation

Contents /

Tables /

Chapter 1
Introduction

Water surface wave simulation is an essential component in various fields such as computer graphics, virtual reality, video games (see 1.1), visual effects, and so on. The ability to physically and accurately model and render water surfaces, while also minimizing computational resource requirements, is particularly important for creating realistic and immersive virtual environments.

1.1 Goals

The goal of this thesis is to explore various methods for simulating water surfaces, and evaluating their advantages and disadvantages. Furthermore, the thesis will involve the implementation of the water surface simulation method described in [1] using C++ and OpenGL, a widely-used API for rendering 2D and 3D graphics and light GPU computation. This practical component will demonstrate the feasibility and performance of the chosen method.

1.2 Structure

In the following chapters, we will examine the theoretical foundations of water wave simulation, exploring key principles and equations such as the Navier-Stokes equations. We will also review existing literature and methodologies, comparing their computational efficiency and accuracy. In the Theory and Design chapters, we will go a bit deeper into explaining the underlying physics used in the method this project is focused on and also outline the main aspects of the final application. The Implementation chapter will detail the process of developing the water surface simulation with OpenGL compute shaders, including the challenges that it might entail. We will continue with the Results chapter demonstrating the results and metrics of the performance of the application. Finally, in the Conclusion chapter, we will discuss the outcome, potential improvements, shortcomings, and possible future enhancements for the prototype application.

Figure 1.1. Example of waves in a video game called Sea of Thieves taken from [2].

Chapter 2
Method overview

Fluid motion is governed by a combination of different physical forces, like gravity, pressure, or surface tension. These forces together shape the behavior of fluids. For the past few decades, Navier-Stokes equations have been the most widely used way to predict this behavior. These equations, which describe how the velocity of a fluid evolves over time, are fundamental to fluid dynamics.

Methods that utilize Navier-Stokes equations are normally divided into three categories: analytical approximations, numerical solutions, and hybrid approaches. Analytical approximations aim to simplify the equations to make them solvable in a closed form, offering quick results, but often at the cost of generality and inability to simulate complex boundary conditions. On the other hand, numerical approaches discretize the equations and directly solve them, providing accurate results, but require significant computational resources. Hybrid approaches aim to balance the two, seeking to achieve a compromise between computational efficiency and solution accuracy.

In this chapter, we will explore different methods and examine how each approach works, its theoretical foundations, and the specific contexts they excel or fall short in. By understanding the strengths and limitations of these methods, we can make informed choices about which techniques to use for different applications.

2.1 Analytical methods

Analytical methods mostly make different theoretical assumptions to the Navier-Stokes equations to simplify and make them more manageable. In computer graphics, these assumptions are waves in deep water, periodic boundary conditions, and small amplitudes. These methods have been very effective in creating height field waves by utilizing analytical solutions to the differential equations on a 2D grid, while also taking into account the wave dispersion. Examples of such methods could be [3–5].

Figure 2.1. Example of an analytical approach from [3].

Some of the approaches try to mitigate the limitations and can simulate shallow waves by allowing interactions with a predefined heightfield-based static environment [6], but still lack the ability to interact with dynamic and more complex environments.

These methods are very effective at simulating the waves regardless of the frequency without any impact on the stability of the method and are not affected by the CFL condition [7], since they do not depend on the time step size. However, due to their limitations and assumptions, it becomes difficult and resource-demanding to compute complex and non-periodic boundaries.

2.2 Numerical methods

To address the shortcomings of the analytical approach, it is possible to numerically solve the Navier-Stokes equations. There are particle-based approaches like [8], that aim to volumetrically simulate water as a series of particles. While this has its own benefits, it is unnecessarily computationally intensive for simulating the ocean surface in many cases. Other methods [9–10] do this by simulating wave motions over a 2D grid representing height field which reduces the computational cost compared to the 3D particle representation.

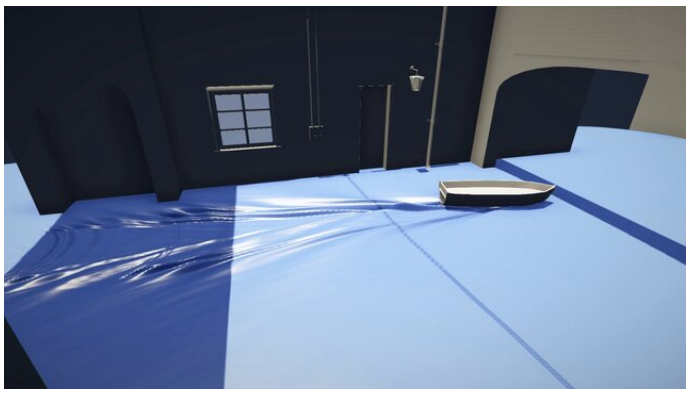

Figure 2.2. Example of an numerical approach from [10].

Due to having minimal assumptions about the environment, these direct numerical approaches are more flexible, than analytical, can have more elaborate environments, and generally are a more accurate representation of the surface. However, since they simulate wave propagation through time by iterating over a grid, the resulting level of detail is dependent on the resolution of the grid, which directly impacts performance and is subject to CFL condition [7].

This is a necessary convergence condition when solving certain partial differential equations. It defines a so-called **Courant number**, which you can obtain like so:

$$C = \frac{u\Delta t}{\Delta x} \tag{1}$$

where:

- u is the maximum velocity of propagation
- Δt is the time step
- Δx is the spatial interval we are working with

We will discuss how to work with it in the implementation section.

2.3 Hybrid methods

Finally, there are hybrid approaches that try to combine the best of the two and have good visual fidelity, flexibility in the choice of boundaries, and aren't very computationally demanding. Notable examples might be [11–12]. They have numerical stability and relative accuracy from the analytical approach but also divide the waves into smaller localized packets that can more easily interact with boundaries. The method this project aims to implement [1] also falls into this category and we will discuss it more in the next chapter.

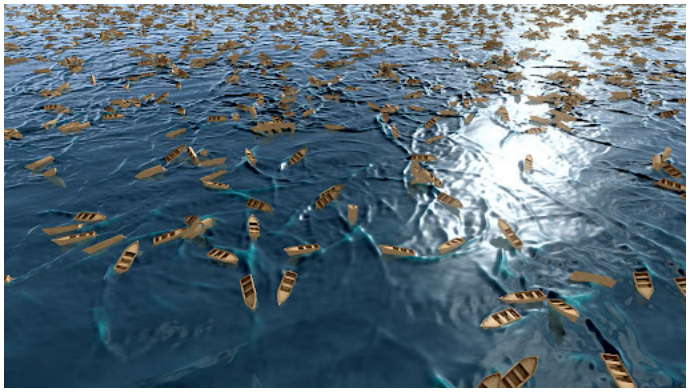

Figure 2.3. Example of a hybrid approach from [11].

Chapter 3
Theory

In this chapter, we will go through the motivation behind the chosen method to implement, its theoretical underpinnings and physical concepts that will be used in the implementation, and detail the sections of the application and how they will work together.

At the beginning we will focus on the advantages of this method and why it was chosen. Then we will also describe the main theoretical foundation it is based on and finally go through some important physical concepts that it employs.

3.1 Motivation

The chosen has made several improvements for its time compared to what had been done before, namely:

- **Spatial and frequential degrees of freedom of the simulation grid.** This allows for more localized control over the result
- **Lower frequency simulation variable other than the height of water.** Since water height changes rapidly over time, it requires a higher resolution simulation grid to accurately represent the surface, compared to amplitude. This decision, on the other hand, makes even less detailed grids yield comparably good results

Additionally, according to the authors, this method is much more GPU-friendly, meaning it is easier to parallelize, which makes it a good candidate for our project.

3.2 Theoretical foundation

Here we will explore the theory behind the method, based on the specification explained in [1]. The main idea of the method is to represent the height of the waves with the complex function:

$$\eta_c(\boldsymbol{x}, t) = \int_{\mathbb{R}^2} \mathcal{A}(\boldsymbol{x}, \boldsymbol{k}, t) e^{i(\boldsymbol{k}\cdot\boldsymbol{x} - \omega(k)t)} d\boldsymbol{k} \tag{1}$$

and the resulting water height is the real part of it: $\eta(\boldsymbol{x}, t) = \text{Re}\ \eta_c(\boldsymbol{x}, t)$. In this function $\boldsymbol{x}$ is a 2D spatial coordinate, $\boldsymbol{k}$ is a wave vector, its magnitude $k = |\boldsymbol{k}|$ is a wave number, $\hat{\boldsymbol{k}} = \frac{\boldsymbol{k}}{|\boldsymbol{k}|}$ is the normalized wave vector and $\omega(k)$ is angular speed of the waves with the wave number k.

$\mathcal{A}(\boldsymbol{x}, \boldsymbol{k}, t)$ is the amplitude function that varies over space, wave frequencies, and time. Compared to the previous works that used $\mathcal{A}(\boldsymbol{k})$ that varies only over the wave parameters, this approach provides more freedom with localized control over the change in wave shapes.

The time evolution of this amplitude function is then represented by the following differential equation:

$$\frac{\partial \mathcal{A}}{\partial t}(\boldsymbol{x}, \boldsymbol{k}, t) = -\omega'(k)(\hat{\boldsymbol{k}} \cdot \nabla_x)\mathcal{A}(\boldsymbol{x}, \boldsymbol{k}, t) \tag{2}$$

This is the advection of the amplitude in the direction $\hat{\boldsymbol{k}}$ with the group velocity $\omega'(k)$. We will also consider boundary conditions as explained in the method specification, with:

$$\begin{aligned} \mathcal{A}(\boldsymbol{x}, \boldsymbol{k}, t) &= \mathcal{A}_{ambient}(\boldsymbol{x}, \boldsymbol{k}, t) \\ \mathcal{A}(\boldsymbol{x}, \boldsymbol{k}, t) &= \mathcal{A}(\boldsymbol{x}, \boldsymbol{k}_{reflected}, t) \end{aligned} \tag{3}$$

where $\mathcal{A}_{ambient}(\boldsymbol{x}, \boldsymbol{k}, t)$ is going to be the transmitting boundary at the edges of the simulation domain and $\mathcal{A}(\boldsymbol{x}, \boldsymbol{k}_{reflected}, t)$ is the behavior when reflecting a boundary with $\boldsymbol{k}_{reflected} = \boldsymbol{k} - 2(\boldsymbol{k} \cdot \boldsymbol{n}) \cdot \boldsymbol{n}$ as the reflected wave vector.

3.2.1 Discretization over a grid

For the discretization of this amplitude function we will use the same 4D grid from section 4.1 of [1], with 2 spatial dimensions x and y, Θ angle from the equation $\boldsymbol{k} = (k \cdot cos\Theta, k \cdot sin\Theta)$, that can be represented by this diagram:

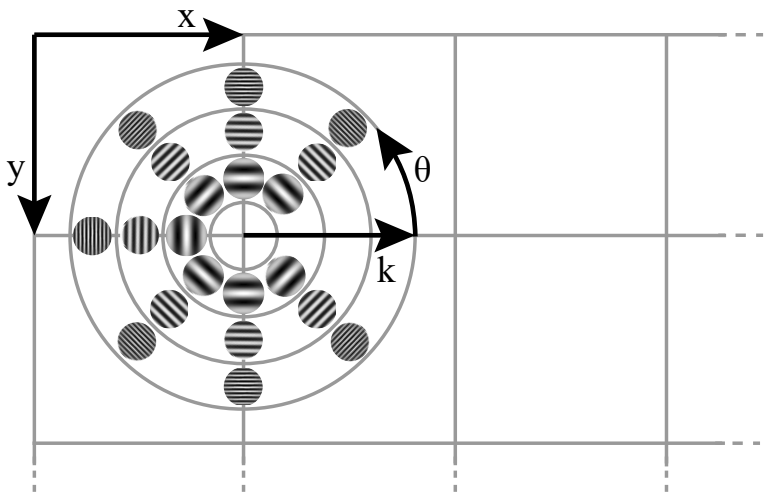

Figure 3.1. Grid representation diagram taken from [1], Figure 2.

This grid ranges between $[x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [0, 2\pi] \times [k_{min}, k_{max}]$. The continuous amplitude can then be obtained by interpolating between neighboring nodes of the grid. This relation can be represented as the combination of basis functions in all dimensions:

$$\mathcal{A}(\boldsymbol{x}, \boldsymbol{k}, t) = \sum_{a,b,c} \mathcal{A}_{abc}(t)\phi_a(\boldsymbol{x})\theta_b(\Theta)\psi_c(k) \tag{4}$$

where $\mathcal{A}_{abc}(t)$ are values stored in grid with the position $\boldsymbol{x}_a$, angle Θ_b and wave number k_c in a given time t. While $\phi_a(\boldsymbol{x})$, $\theta_b(\Theta)$, $\psi_c(k)$ are basis functions interpolating between the values of $\mathcal{A}_{abc}$ over space, angle and wave number respectively. Since $\psi_c(k)$ represents the dimension of frequency, we can use an actual ocean spectrum for it, which will make the result more realistic. We will be using a so-called Pierson–Moskowitz

spectrum, described in [13] but also compare it to other spectra to see how the result differs.

The authors mention using either linear or cubic interpolation for the first two terms, while we will be using only linear interpolation, which has a good reason that will be explained later in the implementation section.

3.2.2 Advection

Wave vector advection is the main way amplitudes propagate through our grid, it is solved by semi-Lagrangian advection and is represented by the equation:

$$\mathcal{A}_{abc}(t + \Delta t) = \mathcal{A}_{bc}(\boldsymbol{x}_a - \Delta t \omega'(k_c)\hat{\boldsymbol{k}}, t) \tag{5}$$

where, as previously mentioned, $\hat{\boldsymbol{k}} = (cos\Theta_b, sin\Theta_b)$ direction of the wave vector and $\mathcal{A}_{bc}$ is the interpolated amplitude with a fixed angle Θ_b and wavenumber k_c $\omega'(k_c)$ is the group speed. In this step, we also apply relevant boundary conditions if we leave the simulation domain or interact with a boundary.

3.2.3 Diffusion

Additionally, there are also diffusion terms to smooth out the adjacent discrete wave vectors to make them less spread out. With those terms, the partial derivative of the function looks like this:

$$\frac{\partial \mathcal{A}}{\partial t} = -\omega'(k)(\hat{\boldsymbol{k}} \cdot \nabla_x)\mathcal{A} + \delta(\hat{\boldsymbol{k}} \cdot \nabla)^2 \mathcal{A} + \gamma \frac{\partial^2 \mathcal{A}}{\partial \Theta^2} \tag{6}$$

this equation is dependant on the sizes of steps in each direction Δx, $\Delta\Theta$ and Δk, where $\delta = 10^{-5}\Delta x^2 \Delta k^2 |\omega''(k)|$ controls diffusion in the direction of travel, and $\gamma = 0.025\omega'(k)\Delta\Theta^2/\Delta x$ control diffusion in the angle. It is discretized with second order finite differencing.

3.2.4 Height calculation

For the final height calculation, we will use the formula:

$$\eta(\boldsymbol{x}, t) = \int_0^{2\pi} \sum_{a,b,c} \mathcal{A}_{abc}(t)\phi_a(\boldsymbol{x})\theta_b(\Theta)\overline{\Psi}_c(\hat{\boldsymbol{k}} \cdot \boldsymbol{x} + \xi(\hat{\boldsymbol{k}}), t) d\Theta \tag{7}$$

where we integrate over all the directions. Additionally, this equation introduces the new concept, that the authors called *profile buffer*. This buffer is essentially responsible for the local details of the waves. $\xi(\hat{\boldsymbol{k}})$ term is a random number that depends on the angle Θ and adds variability to the the waves. In the results section we will try to display what happens if we don't use it and how the surface is affected by it. Profile buffer is calculated as follows:

$$\overline{\Psi}_c(p, t) = \int_0^{\infty} \psi_c(k) cos(kp - \omega(k)t) k \, dk \tag{8}$$

where we integrate over the entire spectrum of wave lengths we simulate on, $\psi_c(k)$ a localized amplitude function of the waves and is represented by an ocean spectrum here, and the *cos* term the vertical displacement of the waves. However, here we will also compute a horizontal displacement to more closely resemble a so-called trochoidal or Gerstner [14] waves, as has been also pointed out by the authors of the method.

3.3 Physical concepts

In this section, we will discuss the necessary underlying physical concepts that we will work with while implementing and how they relate to each other. Also, keep in mind that many of these parameters have different formulas depending on the depth of the ocean, but since we are working with deep water, we might as well assume that the depth is approaching infinity: $h \to \infty$. This in turn simplifies many of the following equations.

3.3.1 Dispersion relation

One of the concepts is *dispersion relation* or angular frequency, which relates the wave number of a wave and its frequency. It is calculated as follows:

$$\omega(k) = \sqrt{g \cdot k} \tag{9}$$

where $g = 9.81 m/s^2$ is the gravitational constant.

3.3.2 Group velocity

Another one is *group velocity*. It depends on the dispersion of the wave and represents the speed of the entire packet of the waves propagating through space. The formula for calculating group velocity is:

$$\omega'(k) = \frac{1}{2}\sqrt{\frac{g}{k}} \tag{10}$$

3.3.3 Wave length/number relation

In addition, we will also need this relation that connects the wave number to its wave length and is calculated according to the equation:

$$k = \frac{2\pi}{k_{length}} \tag{11}$$

3.3.4 Gerstner waves

Gerstner wave is an exact solution of the Euler equations for periodic gravity waves first discovered by Franz Gerstner [14]. This concept is often used in computer graphics to model waves because it doesn't have high computational requirements and is generally a visually realistic representation. As we have discussed earlier, [3] has shown how to do it. These are the equations for the calculation of the final absolute position of a point, taken from [15]:

$$\begin{aligned}
\xi &= \alpha - \sum_{m=1}^{M} \frac{k_{x,m}}{k_m} \frac{a_m}{tanh(k_m h)} sin(\Theta_m), \\
\eta &= \beta - \sum_{m=1}^{M} \frac{k_{z,m}}{k_m} \frac{a_m}{tanh(k_m h)} sin(\Theta_m), \\
\zeta &= \sum_{m=1}^{M} a_m cos(\Theta_m), \\
\Theta_m &= k_{x,m}\alpha + k_{z,m}\beta - \omega_m t - \phi_m
\end{aligned} \tag{12}$$

In these equations terms ξ, η and ζ show functions that represent the absolute final position of points in x, z and y dimensions respectively (with y being the vertical component). α and β are initial horizontal positions. a_m is the amplitude of the given wave and k_m is the wave number. The term Θ_m is related to the term $kp - \omega(k)t$ from the equation (8) and roughly represents the phase of our wave.

Subsequently, we will need a normal vector to accurately apply lighting to the resulting fragments and to do that we will use the formula [15]:

$$\begin{aligned} \boldsymbol{n} &= \frac{\partial \boldsymbol{s}}{\partial \alpha} \times \frac{\partial \boldsymbol{s}}{\partial \beta} \\ \boldsymbol{s}(\alpha, \beta, t) &= \begin{pmatrix} \xi(\alpha, \beta, t) \\ \zeta(\alpha, \beta, t) \\ \eta(\alpha, \beta, t) \end{pmatrix} \end{aligned} \tag{13}$$

Chapter 4
Application design

This chapter will explain all the foundational blocks of the application and explain how we are going to implement specific parts of the method in the application, what constructs are going to be used to save and pass the data around, etc.

4.1 Main points

The application prototype will be written for PC using cross-platform libraries, so in theory, it should work on all the systems that support x86-64 architecture. It will be written using C++ for the application itself and OpenGL API for rendering and main computation tasks.

4.1.1 Libraries

- **GLFW** - cross-platform library to create and manage windows, contexts receiving and processing events [1]
- **GLM** - GLSL style mathematics library [2]
- **DevIL** - library to load and manage images [3]
- **Dear ImGui** - lightweight gui library [4]
- **Glad** - OpenGL loader [5]

4.1.2 Application structure

The application will consist of several main components, namely:

- **Simulation** - will store the data and manage our simulation grid and profile buffers,
- **Visualization** - will store the data about the meshes and objects we are rendering and render them,
- **Gui** - won't store anything, just provides functions to easier initialize ImGui and start/end its frames, render the gui data.

Each component will have its initialization function called when it is added to the application. After that, each component will have an update function that will be called during the update stage, a GUI function, where it can submit data to ImGui frame to expose its variables to the user, and a render function, which will be called during the render stage. The application itself will also have all of these functions for the unified approach.

The application itself will first initialize all the necessary resources, and then enter the main loop. States of the application can roughly be represented by the diagram:

[1] https://www.glfw.org/
[2] https://github.com/g-truc/glm
[3] https://openil.sourceforge.net/
[4] https://github.com/ocornut/imgui
[5] https://glad.dav1d.de/

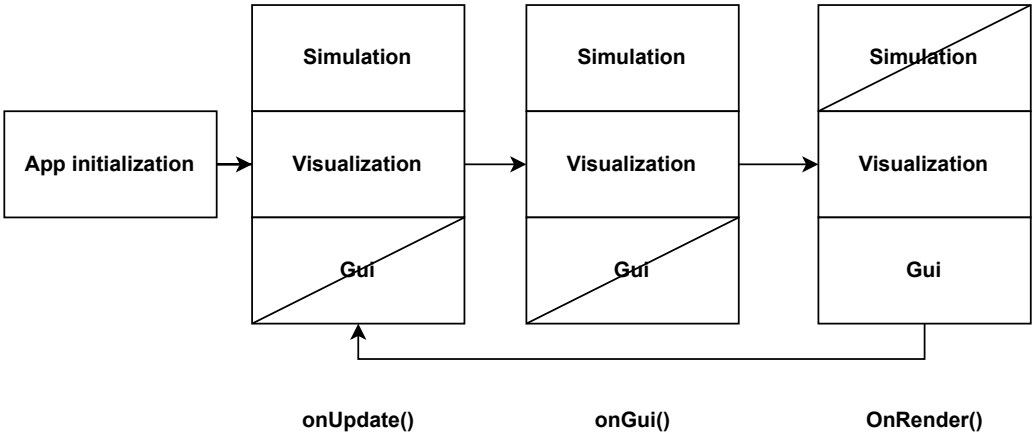

Figure 4.1. Diagram of the flow of the application.

Components that are crossed on the diagram do not participate in the respective stage pointed out below.

4.2 Simulation

This section will go over the necessary parts of the simulation part of the application. We will first explain the way the grid itself will be represented, and then go through each step of the simulation algorithm, elaborating on how they are going to work with the input/output data and give some general information regarding the compute shaders for each step of the simulation.

4.2.1 Amplitude grid

Our simulation variable, namely the amplitude, will be stored in an array of 3D textures, where each 3D texture represents $\mathcal{A}_{bc}$ interpolated amplitude over spatial and angle coordinates from the equation (5). This is also the reason we are using linear interpolation as has been mentioned before, since we can use natively implemented interpolation by just sampling a texture, which in theory should be faster than devising our own interpolation function. Subsequently, each entry in the texture array is going to be indexed by the wavenumber k, the remaining degree of freedom of our grid.

It is worth mentioning, that internally k values in the grid will represent linear wavelength to have a more understandable range of the simulation. Those wavelengths are then going to be converted back to wave numbers when needed according to the formula (11). The idea for this is taken from the CPU implementation mentioned in [1].

This approach is limited by the number of image units we can use, which has to be at least 16 according to the OpenGL specification, meaning it should be possible to simulate up to 15 discrete values of k, since we will have to bind all of these at the same time in the last stage of height evaluation in vertex and fragment shader. The remaining 1 slot is going to be used by the profile buffer texture. While this is a limitation, according to the authors, even **1** discrete k produces more than respectable results, which makes this limitation very unlikely to be an issue.

The class for amplitude grid will also store other relevant information:

- $delta_x$ - step between the discrete nodes of the grid,
- min_x, max_x - minimum and maximum values over the real domain,
- dim_x - resolution of the grid,

x is used here just as an example, and in fact, each of these is a 4D vector, where each entry represents its own dimension.

The class will also have a function that will check for the CFL condition (1) for a given dt and if the condition is met it will return that same dt, otherwise, it will return the maximum possible time step for a given resolution of the grid and group speeds.

4.2.2 Time step

Each time step of the simulation can be represented by the following pseudo-code:

```
1  function TimeStep(t, dt)
2      Advection(dt)
3      Diffusion(dt)
4      Ψ̄ ← precomputeProfile(t)
5  end function
```

Figure 4.2. Algorithm to calculate time step

Note how advection and diffusion require a time step to simulate the propagation of waves, while the profile buffer is not dependent on the change in time, but rather is calculated at an absolute point in time.

4.2.3 Profile buffer

As mentioned in the method specification by the authors [1], the profile buffer is a 1D texture and there is its own profile buffer associated with each wavenumber k. This makes it possible to use a 1D array texture (`GL_TEXTURE_1D_ARRAY`), which is essentially continuous in one dimension and discrete in the other, which is perfect for our use case. As mentioned before, the buffer will precompute 2 terms for spatial displacement and 2 derivatives for normal calculation, 4 float values in total, which we store as RGBA values in the texture. Additionally, we will calculate two phases of the wave function and interpolate between them to achieve better results. In the results section we will compare the outcome with and without interpolation between the phases.

The way we will be calculating the buffer is roughly like this:

```
1  function precomputeProfile(t)
2      for p_i ← 0, p_max
3          p ← p_i · period/p_max + k_min
4          Ψ̄(p_i) ← Integrate(k_min, k_max, gerstner(k))
5      end for
6  end function
7
8  function gerstner(k)
9      k_num ← τ/k
10     phase ← k_num · p − ω(k_num) · t
11     return k · ψ(k)· cos(phase)
12 end function
```

Figure 4.3. Pseudo code for profile buffer calculation

Here `Integrate` is a function that calculates integral between k_{min} and k_{max} over the function `gerstner`, p_{max} is the resolution of the profile buffer and k_{length} is the

wavelength related to the wave number k. The *period* in this case is a multiple of k_{max} for a given profile buffer (the idea was taken from the CPU implementation [1]). The multiplier will be tweakable by the use user in GUI. These two functions together represent the equation (8).

4.2.4 Water height

The final displacement calculation will be done in the vertex shader and normal vector calculation in the fragment shader associated with water mesh, which will be explained more in-depth later. Here it is worth mentioning, that the rough way we will do it is according to Algorithm 1 from [1]:

```
1  function WaterHeight(x, t)
2      η ← 0
3      for b ← 1, Θ_η do
4          θ_b ← b · 2π/Θ_η
5          k ← (cosθ, sin θ)
6          p ← k · x+ rand(b)
7          for k ← 1, K do
8              η ← η + A(x, k_c k) · Ψ̄_c(p, t)
9          end for
10     end for
11 end function
```

Figure 4.4. Pseudo code for calculating water height.

where Θ_η is number of integration nodes for angle Θ and K is number of discrete wave numbers.

4.2.5 Compute shaders

The main workforce for computing all the necessary simulation data will are going to be compute shaders and the data should remain on the GPU at all times. Passing data between CPU and GPU every simulation step is unacceptable since it will produce a substantial performance overhead.

Compute shaders we will have are:

- **Advections Compute** - will take all of the necessary data as uniforms, including the amplitude textures, and calculate the resulting amplitude according to the equation (5). For each discrete wavenumber k we will dispatch its own compute shader. Input textures are going to be bound to `sampler3D` units, so we can interpolate between values, while output textures will be `image3D` to access and write specific values.
- **Diffusion Compute** - here it is much simpler, since we will be working with discrete values, so no need for samplers, both input and output textures will be `image3D`, and we will need multiple inputs, since we have to deal with other k. As in the previous step, we will dispatch separate compute shaders for each k. and also bind image textures for $k-1$ and $k+1$.
- **Disturbance Compute** - to more easily interact with the surface for the user, we will also add a so-called disturbance compute shader, that will create disturbances on the water surface by increasing amplitude in all discrete directions Θ_b by some constant values. The shader will take as uniforms 2 `image3D` textures: input and output, and a position where to create a disturbance. It will be dispatched when the user clicks on a point on the surface. The idea for this was taken from the CPU implementation of [1].

- **Profile Compute** - compute shader that will calculate profile buffer entries according to the equation (8) and store them in a `image2D` to which we will bind a `GL_TEXTURE_1D_ARRAY` as mentioned previously. we will have to dispatch this shader only once each simulation step since it will work on the entire profile buffer at once. It will accept uniform values for the spectrum that we want to choose (the function will be implemented in the shader itself, we will just have to choose it), upper/lower bounds of k we want to integrate over, absolute time and wind speed.

4.3 Other details

There are going to be other useful parts that don't directly relate to simulation but still contribute to the application.

4.3.1 Other shaders

We will certainly also need shaders to render the scene, these are going to be vertex/fragment shader pairs:

- **Lighting shader** - a generic lighting shader that implements the Phong Illumination Model [16] and will be used mainly for boundary objects.
- **Skybox shader** - a shader that we will use to render the sky box around the scene, will contain `samplerCube` in the fragment shader.
- **Water shader** - an extension to the lighting shader above, that will also take amplitudes and profile buffer and calculate the spatial displacement of a vertex in the vertex shader and a normal, that will be used for lighting in fragment shader.

4.3.2 Abstraction classes

These is an overview of what abstraction classes we will have to hide `gl*` commands in the main application and instead interact with objects only:

- **Base shader** - for the ease of interacting with shader programs, we will associate each program with an object and have functions to change its uniforms by directly interacting with this object.
- **Compute shader** - extension of the base shader, that will only add way to read from file and compile a compute shader.
- **Shader** - an extension of the base shader, that will be used for rendering and will have a constructor that reads both vertex and fragment shaders from files and compiles them into one program.
- **Mesh** - generic mesh object that will have a shader associated with it, that will be used to render it. Will store a reference to buffers associated with the data of its mesh, which will be initialized in the constructor.
- **Object** - generic object that will have a mesh associated with it, and also store additional data related to a specific instance, like its position, rotation and scale.
- **Camera** - a way to abstract the view and projection matrices used to transform the scene into viewport coordinates. The framework we are building will allow us to have more cameras, but we will be operating with only one.

In all the cases, except Camera, these are the classes we will inherit from to express a specific case like advection compute shader, water or skybox object, etc, but we will detail all of these in the next section.

Chapter 5
Implementation

This chapter is going to be dedicated to the details about the implementation of different parts of the application, while also explaining the steps and design choices. At the beginning, we will explain everything related to simulating the wave vector advection, diffusion, and precomputation of the profile buffer. Then, we will go over the way the resulting water mesh is going to be constructed and updated for each frame.

5.1 Simulation

The class that handles all the simulation is called `AmplitudeGrid`. Parameters for the grid testing are going to be set in a separate file as `#define` macros. The grid resolution we will be using to test is going to be:

Dimension	Macro	Resolution
X - spatial	`N_SPATIAL`	128
Θ - angle	`N_THETA`	16
K - wavenumber	`N_K`	1
N - profile samples	`P_RES`	4096

Table 5.1. Testing parameters

5.1.1 Shaders

It is worth mentioning a few details about how we are going to abstract shader initialization and interaction since that information will be useful later.

Firstly, all the shaders have functions to set uniform values associated with its program ID in a form `set[Type](std::string name, Type value)` where `name` is the name of the uniform variable, value is its value and `[Type]` can be any type like `Integer`, `Float`, `Vec3`, `IVec2`, etc.

Also, to abstract the `glUseProgram` command, we have functions like `bind()` and `unbind()`.

Secondly, we will be using the extension `GL_ARB_shading_language_include`, which enables a functionality, similar to C language `#include`, with the difference, that it doesn't allow us to read files from the PC, but includes so-called *named strings*.

To include a named string we first need to register it with the function `glNamedStringARB`, which associates a path with a string, where path is going to be an internal way for the GPU to find the file and the string is what will be included in the shader source code in place of `#include` during compilation.

To facilitate this process, we introduce a static function in the `ShaderBase` class `ShaderBase::addIncludeFile(const std::string& fpath);` which reads contents of a file at `fpath` and adds them as a named string.

Additionally, **if** the path has substring **macros** in it, the function will remove the last two lines and add the macros of our current `N_THETA` and `N_K`. This might be a very convoluted process, but it allows us to easily pass parameters to the shader during compilation. This also means that shader files that define macros should have `N_THETA` and `N_K` macros at the end of the file.

The reason we cannot pass these as uniform values lies in the fact, that we need to use them as a size for sampler and image arrays, which has to be known at compile time.

5.1.2 Grid initialization

We first need to construct the grid and initialize all of its internal structures. The grid will be constructed from the following function :

```
12  AmplitudeGrid(float size, float waveNumberMin, float waveNumberMax);
```

where we set the physical size of the simulation domain, which will be mapped directly onto our mesh and min/max values for wavenumber since we cannot simulate infinitely big range.

In the constructor, we set parameters for the min/values, grid resolution and step in the physical domain between the grid nodes:[1]

```
1  m_dim = (N_SPATIAL, N_SPATIAL, N_THETA, N_K),
2  m_min = (-size/2, -size/2, 0.0f, waveNumberMin),
3  m_max = (size/2, size/2, TAU, waveNumberMax)
4  for (int d = 0; d < 4; d++)
5  {
6      m_delta[d] = (m_max[d] - m_min[d]) / m_dim[d];
7  }
```

After that we are initializing amplitude textures, IDs for textures reserved for each k are stored in `std::vector`, and we will need twice as much, also including output textures, since we need to store output data somewhere, without copying data, meaning we will have 2 vectors. Initialization is then done as follows:

```
1  float borderColor[] = { 0.0f, 0.0f, 0.0f, 0.0f };
2  m_inTextures.resize(N_K);
3  m_outTextures.resize(N_K);
4  glCreateTextures(GL_TEXTURE_3D, N_K, m_inTextures.data());
5  glCreateTextures(GL_TEXTURE_3D, N_K, m_outTextures.data());
6  for (int i = 0; i < N_K; ++i)
7  {
8      GLuint currTexture = m_inTextures[i];
9      glTextureParameteri(currTexture, GL_TEXTURE_WRAP_S,
10     GL_CLAMP_TO_BORDER);
11     glTextureParameteri(currTexture, GL_TEXTURE_WRAP_T,
12     GL_CLAMP_TO_BORDER);
13     glTextureParameterfv(currTexture, GL_TEXTURE_BORDER_COLOR,
14     borderColor);
15     glTextureParameteri(currTexture, GL_TEXTURE_WRAP_R, GL_REPEAT);
16     glTextureParameteri(currTexture, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

[1] prefix `m_*` is there for class members, to distinguish them from local variables

```
17      glTextureParameteri(currTexture, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
18      glTextureStorage3D(currTexture, 1, GL_R32F, m_dim[X], m_dim[Z],
19      m_dim[Theta]);
20
21      currTexture = m_outTextures[i];
22      glTextureParameteri(currTexture, GL_TEXTURE_WRAP_S,
23      GL_CLAMP_TO_BORDER);
24      glTextureParameteri(currTexture, GL_TEXTURE_WRAP_T,
25      GL_CLAMP_TO_BORDER);
26      glTextureParameterfv(currTexture, GL_TEXTURE_BORDER_COLOR,
27      borderColor);
28      glTextureParameteri(currTexture, GL_TEXTURE_WRAP_R, GL_REPEAT);
29      glTextureParameteri(currTexture, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
30      glTextureParameteri(currTexture, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
31      glTextureStorage3D(currTexture, 1, GL_R32F, m_dim[X], m_dim[Z],
32      m_dim[Theta]);
33  }
```

where we first allocate enough space in vectors and create `N_K` 3D textures in each vector. After that we initialize all the parameters: set wrapping for spatial coordinates to clamp to the border, which is `{0, 0, 0, 0}` to not affect simulation if we by any chance try to sample outside of it. However, for the angle dimension, we use `GL_REPEAT`, since if we get out of the direction range $(0, 2\pi)$ we just have to wrap around.

Even though we set these wrappings, they are mostly to ensure that we don't get arbitrary results if something goes wrong, but we will be checking for boundaries in the code either way, so, during the normal operation, it won't get to sampling outside.

Additionally, we set down and upsampling to `GL_LINEAR`, to have the necessary linear interpolation when sampling the values.

Subsequently, we allocate space for $X \cdot \Theta \cdot K$(5.1) values calling `glTextureStorage3D`.

We then use `std::vector::swap` after each step of the simulation, to ensure each subsequent step has updated data to work with.

For the profile buffer, we won't have any special class and it will be represented by the function associated with the grid. The initialization of the profile buffer texture is as follows:

```
1  glCreateTextures(GL_TEXTURE_1D_ARRAY, 1, &m_profileTexture);
2  glTextureParameteri(m_profileTexture, GL_TEXTURE_WRAP_S, GL_REPEAT);
3  glTextureParameteri(m_profileTexture, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
4  glTextureParameteri(m_profileTexture, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
5  glTextureStorage2D(m_profileTexture, 1, GL_RGBA32F, P_RES, N_K);
```

which, as mentioned before, is 1D array texture. Here we again need linear up/down-sampling, and `GL_REPEAT` for wrapping over the wave phases. After that we allocated data for $N \cdot K$ (5.1) values.

Additionally, we need to initialize shaders, and all of them are placed in the `shaders/` folder:

```
1  ShaderBase::addIncludeFile("shaders/compute_macros.glsl");
2  ShaderBase::addIncludeFile("shaders/compute_common.glsl");
3
4  m_advectionCompute = std::make_unique<TimeStepCompute>("shaders/
5  advection.comp");
```

```
m_diffusionCompute = std::make_unique<TimeStepCompute>("shaders/
diffusion.comp");
m_disturbanceCompute = std::make_unique<DisturbanceCompute>("shaders/
disturbance.comp");
m_profileCompute = std::make_unique<ProfileCompute>("shaders/
profile.comp");
```

Since we are using `std::unique_ptr` [2] instead of pure pointers, we have to construct them using `std::make_unique`. Shader class constructors read the contents of files that are passed as a parameter and compile them, saving the program ID to its internal variable.

5.1.3 Time Step

Time step function is almost exactly the same as in 4.2:

```
void AmplitudeGrid::timeStep(float dt, bool updateAmps)
{
    m_time += dt;
    if (updateAmps)
    {
        advectionStep(dt);
        wavevectorDiffusion(dt);
    }
    precomputeProfileBuffers();
}
```

with a notable exception like `bool updateAmps` which defines whether to update the grid. If we just compute the profile buffer without updating the grid, the ocean will have steady waves that circle around periodically.

Additionally, the code to calculate a time step according to the condition (1), as explained at the end of section 4.2.1, we have a function:

```
double AmplitudeGrid::cflTimeStep(float dt, float timeMultiplier) const
{
    dt = dt * pow(10, timeMultiplier);
    float u = groupSpeed(m_dim[K] - 1);
    float dx = std::min(m_delta[X], m_delta[Z]);

    if (u * dt / dx > 0.5f)
        return (0.5 * dx / u);
    else
        return dt;
}
```

here we also have a `timeMultiplier` parameter which allows us to speed up or slow down the simulation.

5.1.4 Common shader functions

In a separate file `compute_common.glsl` we have a bunch of functions, that are going to help us recover the position in the physical simulation domain from a grid position and vice versa. These have both vector and individual value versions:

[2] `https://en.cppreference.com/w/cpp/memory/unique_ptr`

```
float realPos(int idx, int dim)
{
   return u_min[dim] + (idx + 0.5) * u_delta[dim];
}
vec3 realPos(int ix, int iz, int itheta)
{
   return vec3(realPos(ix, X), realPos(iz, Z), realPos(itheta, Theta));
}
float gridPos(float val, int dim)
{
   return (val - u_min[dim]) / u_delta[dim] - 0.5;
}
vec3 gridPos(vec3 realPosition)
{
   return vec3(gridPos(realPosition.x,X), gridPos(realPosition.y, Z),
gridPos(realPosition.z, Theta));
}
```

Figure 5.1. Position conversion functions

Additionally, in the `compute_macros.glsl` file we have a way to index values in a grid position vector based on their

5.1.5 Advection

Advection compute shader is associated with an instance of `TimeStepCompute` class, where it calls the function `dispatchAdvection`.

Since the maximum number of invocations in a local group is 1024, we define local group size as:

```
layout (local_size_x = 16, local_size_y = 16, local_size_z = 4) in;
```

which exactly comes out to be $16 \cdot 16 \cdot 4 = 1024$. Spatial coordinates were chosen only because Θ is 4, which will be the lowest possible value for our `N_THETA`. Also, to ensure that workgroup sizes are whole numbers, we add static assertions to check for that in the parameter file:

```
static_assert(N_SPATIAL % 16 == 0 && N_SPATIAL >= 16);
static_assert(N_THETA % 4 == 0 && N_THETA >= 4);
```

these ensure that spatial dimensions of out grid are divisible by 16 and the angle dimension is divisible by 4. This way when we dispatch compute shaders, we can safely divide dimensions by the local group size like so:

```
glDispatchCompute(N_SPATIAL / 16, N_SPATIAL / 16, N_THETA / 4);
```

Advection itself is done in the main function:

```
ivec3 pos = ivec3(gl_GlobalInvocationID);
vec3 realPosition = realPos(pos[X], pos[Z], pos[Theta]);
float result;
vec2 waveVector = vec2(cos(realPosition[Theta]),
sin(realPosition[Theta]));
realPosition = realPosition - u_dt * u_groupSpeed *
vec3(waveVector, 0.0);
realPosition = reflection(realPosition);
```

```
9   vec3 gridPosition = gridPos(realPosition);
10
11  if (gridPosition[X] <= 0.0 || gridPosition[X]+1 >= u_dim[X] ||
12  gridPosition[Z] <= 0.0 || gridPosition[Z]+1 >= u_dim[Z])
13  {
14     result = defaultAmplitude(realPosition[Theta]);
15  }
16  else
17  {
18     vec3 texPos =  gridPosition/(u_dim.xyz) + 0.5/u_dim.xyz;
19     result = texture(in_Grid, texPos).r;
20  }
21  imageStore(out_Grid, pos, vec4(result, 0, 0, 0));
```

Figure 5.2. Advection compute shader

Here we first get a position on the grid, by referencing the built-in compute shader variable `gl_GlobalInvocationID`, which is equal to `gl_WorkGroupID * gl_WorkGroupSiz + gl_LocalInvocationID`. It is an integer position where this invocation is on our 3D grid. From this int position on the grid, we calculate the real position in our simulation domain with the function `realPosition` explained above. Then we retrieve the normalized wave vector from the angle Θ and calculate the semi-Lagrangian advection.

During this stage, we also have to consider boudary reflections. The way we implement them will work only on cubes and the first thing we need is a function to check whether given coordinates are inside of a cube, where `cubepos` is the position of the cube and `cubescale` is its scale such that each side then as length `2 * cubescale`:

```
1   bool insideCube(vec2 position)
2   {
3      return (position.x > cubepos.x - cubescale &&
4            position.x < cubepos.x + cubescale &&
5            position.y > cubepos.y - cubescale &&
6            position.y < cubepos.y + cubescale);
7   }
```

With a way to check if a spatical coordinate is inside of a cube, we can now calculate the reflection using the following function:

```
1   vec3 reflection(vec3 realPosition)
2   {
3      if (!insideCube(realPosition.xy))
4         return realPosition;
5
6      vec2 pos = realPosition.xy;
7      vec2 posc = realPosition.xy - cubepos;
8      vec2 n;
9      if (abs(posc.x) > abs(posc.y))
10        n = sign(posc.x) * vec2(1, 0);
11     else
12        n = sign(posc.y) * vec2(0, 1);
13     n *= 2;
14
```

```
    vec2 kdir = vec2(cos(realPosition[Theta]),
 sin(realPosition[Theta]));
    kdir = kdir - 2.0 * (kdir * n) * n;

    float reftheta = atan(kdir.y, kdir.x);
    return vec3(realPosition.xy, reftheta);
}
```

In the function, we first check if we are inside of the cube since we have to do reflections only in that case. If so, we then calculate the normal of the side of the cube we are the closest to, recover the simulation domain position of θ, and finally perform reflection according to the equation (3).

After the advection, we check if we are still in the simulation domain, and if no, we apply the ambient value for the amplitude.

However, if we still are in the domain, we then calculate texture coordinates ranging `(0, 1)` from the grid coordinates and grid dimensions, and sample the input texture for the resulting value.

5.1.6 Diffusion

As with advection, diffusion compute shader is represented by the `TimeStepCompute` class and has the same local workgroup size. It is also dispatched for each discrete k we have to update their respective textures. Here, however, we need to pass both textures for $k-1$ and $k+1$ as inputs in addition to the texture for k, as well as the current k_i index for k in the 4D grid we are dealing with.

For this reason, the function to dispatch diffusion compute is slightly different and accepts the entire vector of textures and then binds them as needed. Before binding the texture to the image unit it first checks whether we are still within `(0, N_K)` dimensions.

The diffusion itself is done in the shader as follows:

```
ivec3 pos = ivec3(gl_GlobalInvocationID);

float gamma = 0.05 * u_groupSpeed * u_dt  * u_delta[Theta] *
u_delta[Theta] / u_delta[X];

float delta = 0.00002 * u_dt * u_delta[K] * u_delta[K] * u_delta[X] *
u_delta[X] * u_groupSpeed;

float result = (1.0 - gamma) * value(pos[X], pos[Z], pos[Theta]) +
gamma * 0.5 * (value(pos[X], pos[Z], pos[Theta]+1) +
value(pos[X], pos[Z], pos[Theta]-1)) ;

result -= delta * (value(pos[X], pos[Z], pos[Theta], u_ik) + 0.5 *
(value(pos[X], pos[Z], pos[Theta], u_ik + 1) +
value(pos[X], pos[Z], pos[Theta], u_ik -1)));

imageStore(out_Grid, pos, vec4(result, 0, 0, 0));
```

Figure 5.3. Diffusion compute shader

We first calculate values for δ and γ as described in 3.2.3 and then apply the finite differencing method, so that the gamma term is calculated like:

$$\mathcal{A}(a,b,c) = \mathcal{A}(a,b,c) - 2\gamma\mathcal{A}(a,b,c) + \gamma\mathcal{A}(a,b+1,c) + \gamma\mathcal{A}(a,b-1,c) \tag{1}$$

where $\mathcal{A}(a,b,c)$ is the amplitude value with position $\boldsymbol{x}_a$ is position, Θ_b is the angle and k_c is the wave number. By incorporating 2 into γ and simplifying the terms we get:

$$\mathcal{A}(a,b,c) = (1-\gamma)\mathcal{A}(a,b,c) + \frac{\mathcal{A}(a,b+1,c) + \mathcal{A}(a,b-1,c)}{2} \tag{2}$$

The δ term is then calculated in the same fashion, but instead takes adjacent values of k.

Here we do not use samplers, so to deal with out of bounds texture access we check them in the `value()` function before accessing the values in the texture. To deal with the bigger angle we just wrap it to get back into our grid and return 0 if we access outside of the k dimension:

```
float value(int ix, int iz, int itheta, int ik)
{
    itheta = (itheta + u_dim[Theta]) % u_dim[Theta];
    if (ik < 0 || ik >= N_K)
    {
        return 0.0;
    }
    vec4 val = imageLoad(in_Amps[ik], ivec3(ix, iz, itheta));
    return val.r;
}
```

5.1.7 Profile Buffer

The next step is the calculation of the profile buffer. For this purpose, we will dispatch a shader for each layer of the 1D array texture that represents the buffer. Additionally, we will also provide its current k and integral bounds as uniforms. To calculate the bounds we use the same functions 5.1 to get the position in the simulation domain and then subtract and add half of the step Δk to get lower and upper bounds respectively:

```
float kmin = realPos(ik, K) - 0.5 * m_delta[K];
float kmax = realPos(ik, K) + 0.5 * m_delta[K];
```

In the shader itself, `main()` function just integrates over the function:

```
vec4 compute(float k, float p, float c)
{
   float knum = TAU / k;
   float phase1 = knum * p - dispersionRelation(knum) * u_time;
   float phase2 = knum * (p + u_period) - dispersionRelation(knum) *
u_time;
   return k * spectrum(k) * mix(calculateDisplacement(phase2, knum),
calculateDisplacement(phase1, knum), c);
}
```

where `knum` is the wave number calculated from the wavelength (11) and then, as mentioned before, we also interpolate between individual phases of the wave using `mix`

with coeficient $c = \frac{p}{period}$. The function `calculateDisplacement` returns calculated gerstner waves, according to the section 3.3.4.

We need to store only necessary parts in the profile buffer, and there are a few simplifications we can make to the equations (12):

- discard α and β, which will leave us just the relative displacement of the points in horizontal and vertical dimensions.
- since we are working in deep water, the tangent term approaches one: $tanh(k_m\ h) \rightarrow 1$.
- the amplitude a_m is the `specrtrum(k)` function.
- $k_{x,m}$ and $k_{z,m}$ are x and y components of out wave vector $\boldsymbol{k} = (k \cdot cos(\Theta_b), k \cdot sin(\Theta_b))$, we can simplify both $\frac{k_{*,m}}{k_m}$ to $cos(\Theta_b)$ and $sin(\Theta_b)$. These are going to be included in the final height summation.
- the sum $\sum_{m=1}^{M}$ is what we will do during rendering and calculating the final height, so we discard it as well.

In the end, we are left with values $-sin(\Theta)$ and $cos(\Theta)$, where Θ signifies the phase of the wave and should not be confused with the angle on our grid. These are displacements in horizontal and vertical coordinates respectively.

Additionally, we need to calculate partial derivatives $\frac{\partial s}{\partial \alpha}$ and $\frac{\partial s}{\partial \beta}$ as pointed out in (13), which are going to be used to calculate the normal vector.

After evaluating the derivatives we get:

$$\begin{aligned} \frac{\partial s}{\partial \alpha} &= k_{x,m} \begin{pmatrix} -cos(\Theta) \\ -sin(\Theta) \\ -cos(\Theta) \end{pmatrix} \\ \frac{\partial s}{\partial \beta} &= k_{z,m} \begin{pmatrix} -cos(\Theta) \\ -sin(\Theta) \\ -cos(\Theta) \end{pmatrix} \end{aligned} \tag{3}$$

which means we only need to store $k \cdot -cos(\Theta)$ and $k \cdot -sin(\Theta)$ in the profile buffer to be able to calculate the normal.

Taking all the above into consideration, the function `calculateDisplacement()` looks as follow:

```
1 vec4 calculateDisplacement(float phase, float k)
2 {
3     float s = sin(phase);
4     float c = cos(phase);
5     return vec4(-s, c, - k * c, - k * s );
6 };
```

The values in the profile buffer, interpreted as colors, look like this:

Figure 5.4. Profile buffer contents.

5.2 Water surface

To use our simulation data, we need to render a water surface. In this section, we will go over the details of how is water mesh constructed, what are the parameters of shaders related to the water and how it is then rendered.

5.2.1 Water mesh

To start over, we create a separate class extending the generic mesh from 4.3.2. As with the simulation grid, our water mesh will be centered at `(0,0)`. The function to construct water mesh object is:

```
WaterMesh(WaterShader* shader, uint32_t size, float scale);
```

Here `size` is the the size of the mesh in a number of vertices in each dimension, while `scale` is how big the mesh will be as a whole. `WaterShader` is the shader we will use to calculate the final position of the vertex and it will be explained later.

The constructor first initialized the necessary buffers:

```
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glGenBuffers(1, &ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);

glEnableVertexAttribArray(shader->attributes.position);
glVertexAttribPointer(shader->attributes.position, 3, GL_FLOAT,
GL_FALSE, 0, 0);
glBufferData(GL_ARRAY_BUFFER, vertexSetSize * 3, nullptr,
GL_DYNAMIC_DRAW);

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

We will be using `GL_ELEMENT_ARRAY_BUFFER` to draw the mesh, to not need several copies of the same vertices and a vertex array to facilitate the interaction with the buffers.

Before filling the buffers, we have to generate all the positions and indices. Positions are generated as follows:

```
float dx = scale / size;
float dy = scale / size;
for (int ix = 0; ix <= size; ++ix)
{
   for (int iy = 0; iy <= size; ++iy)
   {
      glm::vec3 pos(-scale / 2 + ix * dx, 0.0f, -scale / 2 + iy * dy);
      positions.emplace_back(pos);
   }
}
```

The mesh is created at the vertical coordinate $y = 0$ and ranges between $(-scale/2; scale/2)$ and each triangle length is $scale/size$ww.

The indices are then generated as:

```
for (int ix = 0; ix < size; ++ix)
{
   for (int iy = 0; iy < size; ++iy)
   {
      int idx = iy + ix * (size + 1);
      int Ix = size + 1;
      int Iy = 1;
      indices.push_back(idx);
      indices.push_back(idx + Ix);
      indices.push_back(idx + Iy);

      indices.push_back(idx + Ix);
      indices.push_back(idx + Ix + Iy);
      indices.push_back(idx + Iy);
   }
}
```

And the way individual primitives are indexed can be represented by the diagram:

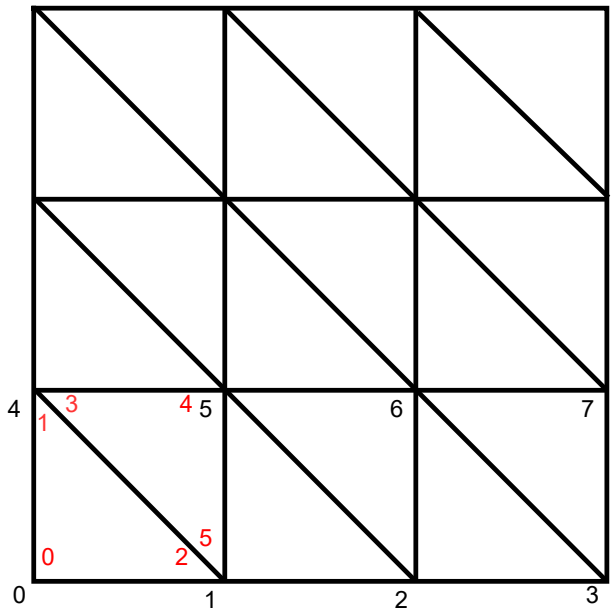

Figure 5.5. Mesh vertex indexing.

where red digits are indices for the primitives and black digits are numbers of individual vertices.

To render this mesh as needed we can then use its `draw` function, where we can also specify the polygon mode (mainly used to render a wireframe):

```
void WaterMesh::draw(GLenum polygonMode) const
{
   glBindVertexArray(vao);
   glPolygonMode(GL_FRONT_AND_BACK, polygonMode);
   glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
   glBindVertexArray(0);
}
```

Additionally, we have a class for the water object instance, but it only stores the color of the water surface and saves the mesh and its model matrix.

5.2.2 Water shader

The final calculations defining the shape of the water surface are going to be done inside the shader, associated with water mesh. To not repeat the code, here we also have some common include files between vertex and fragment shader, which are `water_macros.glsl` and `water_common.glsl`. While the former defines some shared constants as well as the grid parameters, the latter has a shared function, that is used to sample the right texture:

```
float getAmp(float theta, vec3 pos, vec2 posScaled, int ik)
{
   if (pos.x < u_min.x || pos.z < u_min.y || pos.x > u_max.x ||
 pos.z > u_max.y)
   {
      return defaultAmplitude(theta, defDirection, u_defaultAmp);
   }
   vec3 tPos = vec3(posScaled, theta /TAU);
   return texture(u_Amps[ik], tPos).r * u_multiplier;
}
```

which also checks for bounds and returns a default amplitude value if we are sampling outside. This will allow us to see some waves outside of the simulation domain. Here we also apply a user-modifiable parameter `u_multiplier` that won't have any effect on the simulation, but will allow us to exaggerate amplitude size during the rendering stage. Also worth mentioning, is that since we are going to call this function in a loop, we also pass the already scaled-down spatial position as `posScaled`, add a scaled-down `theta` and sample from the amplitude texture at index `ik`.

Here in the common file, we also define a function to generate pseudo-random numbers from a seed, which will be our angle `theta`:

```
float rand(int seed)
{
   return 23.34 * (fract(sin(seed * 123.432) * 5354.53));
}
```

With all of these in our arsenal, we can now get to the main calculation, which is done in both vertex and fragment shaders. In the vertex stage, we first apply the model matrix. The resulting coordinate is then scaled down and saved in the variable which is then passed to the fragment stage and used during amplitude sampling. Then we calculate the final sum (see 4.4) for the displacement of a vertex in both horizontal and vertical coordinates:

```
vec3 calculateDisplacement(vec2 position)
{
   vec3 result = vec3(0);
   for (int b = 0; b < INTEGRATION_SAMPLES; ++b)
   {
      float theta = TAU / INTEGRATION_SAMPLES * b;
      vec2 k = vec2(cos(theta), sin(theta));
      float p = dot(k, position)
```

```
        +
        rand(b);
        p = p / u_profilePeriod;

        for (int ik = 0; ik < N_K; ++ik)
        {
            vec4 val = getAmp(theta, vPosition, vPosScaled, ik) *
    texture(u_profileBuffer, vec2(p, ik));
            result += vec3(k.x * val.x, val.y, k.y * val.x);
        }
    }
    return result;
}
```

This function almost exactly matches the one from the pseudocode mentioned above, with a few notable exceptions. Firstly, we scale down the value of p to the range `(0, 1)` to be able to sample the texture correctly. Additionally, here we can see the terms $\frac{k_{*,m}}{k_m}$ that we left out during the simplification of Gerstner wave equations discussed in the section 5.1.7, and those are `k.x` and `k.y`.

Unfortunately, right now we also do not have the integration over several wave numbers. Instead, the values from each `ik` amplitude grid are being summed up. To implement the integration however, we would also need to pass the min/max values of your k, then calculate its real value similar to how θ is calculated and finally convert the simulation domain position to float grid position and interpolate between the values of adjacent textures.

In the fragment shader, we then calculate the normal for each fragment, using a similar approach, but by utilizing the latter 2 terms from the profile buffer instead:

```
vec3 calculateNormal(vec2 position)
{
    vec3 dx = vec3(1.0, 0.0, 0.0);
    vec3 dz = vec3(0.0, 0.0, 1.0);
    for (int b = 0; b < INTEGRATION_SAMPLES; ++b)
    {
        float theta = TAU / INTEGRATION_SAMPLES * b;
        vec2 k = vec2(cos(theta), sin(theta));
        float p = dot(k, position) + rand(b);
        p = p / u_profilePeriod;
        for (int ik = 0; ik < N_K; ++ik)
        {
            vec4 val = getAmp(theta, vPosition, vPosScaled, ik) *
    texture(u_profileBuffer, vec2(p, ik));

            dx += k.x * val.zwz;
            dz += k.y * val.zwz;
        }
    }
    return normalize(-cross(dx, dz));
}
```

With those 2 terms, we recover the vector of partial derivatives from the equation (3). We then sum them up with the unit vectors, in both spatial dimensions, since we also have to take into consideration the derivatives of α and β in functions ξ and ζ from the equation (12).

With this normal we can now apply the lighting to each fragment, according to he Phong Illumination model [16]:

```
    vec3 normal = calculateNormal(vPosition.xz);

    vec3 lightColor = vec3(1.0);
    vec3 outAmbient = vec3(0.0);
    vec3 outDiffuse = vec3(0.0);
    vec3 outSpecular = vec3(0.0);

    vec3 diffuse = u_diffuse;
    if (vPosition.x < u_min.x || vPosition.z < u_min.y ||
  vPosition.x > u_max.x || vPosition.z > u_max.y)
    {
       diffuse = vec3(0.5);
    }

    vec3 L = normalize(u_lightPosition);
    vec3 R = reflect(-L, normal);
    vec3 V = normalize(u_cameraPosition - vPosition);

    float diff = max(dot(normal, L), 0.0);
    float spec = pow(max(dot(R, V), 0.0), u_shininess);

    outAmbient = u_ambient;
    outDiffuse = diffuse * lightColor * diff;
    outSpecular = u_specular * spec;

    fColor = vec4(outAmbient + outDiffuse + outSpecular, 1.0);
```

It is worth mentioning, that we also check for the position of the fragment, and set the diffuse color of the fragments outside of the simulation domain to gray color, to see where the waves are simulated more easily.

Chapter 6
Results

In this chapter we will present the final results of the project and compare some of the configurations, measuring how much time each of the compute shader stages takes, how long it takes to render, etc. We are also going to compare how some of the parts of the algorithm affect the simulation, as has been mentioned throughout the previous chapters. Finally, we will compare the performance to the CPU implementation on the same configurations.

First of all, in the figure 6.1 you can see the result of the simulation with testing parameters from 5.1 and ambient amplitude $\mathcal{A}_{ambient} = 0.1$ using Pierson-Moskowitz spectrum.

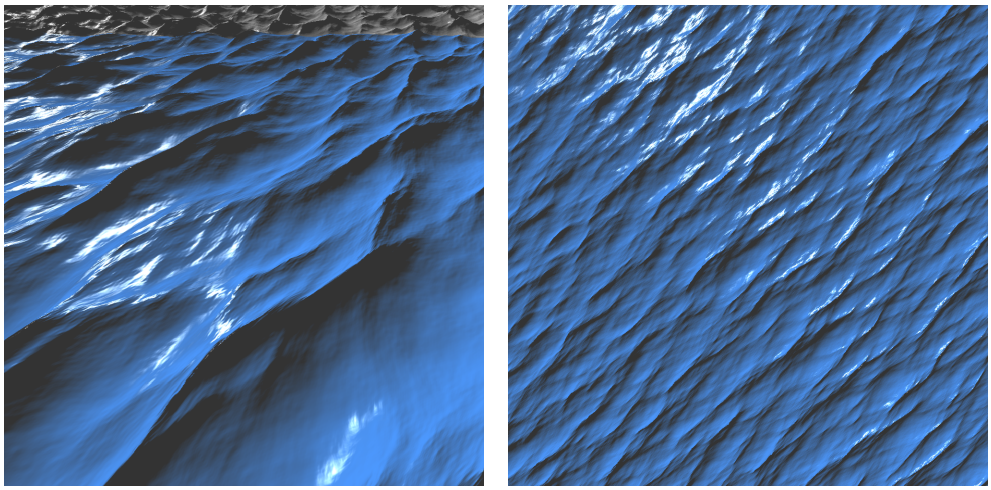

Figure 6.1. Simulation result with the grid resolution $128 \times 16 \times 1$.

Since the amplitude is increased in a single direction, we get this shape of water moving in a certain direction. Ideally, we would want to have more sophisticated boundaries, as well as a precalculated initial grid for $\mathcal{A}$ to have an already established ocean. Nonetheless, this application is still good to show what the method is capable of.

6.1 Performance metrics

In this section, we will try out a few different configurations and measure their performance. The measurements are going to be done in the software RenderDoc [1], which proved to be very useful throughout the work on the thesis, facilitating the debugging of individual frames. The results are presented in the table 6.1.

There we can see how, as expected, profile buffer computation is largely independent of the spatial and angle dimensions, and depends only on the resolution of

[1] `https://renderdoc.org/`

No.	Res. X	Res. θ	Res. K	Advection	Diffusion	Profile
1	128	16	1	0.0266	0.0367	0.0238
2	256	16	1	0.0921	0.1269	0.0228
3	512	16	1	0.3673	0.5313	0.0241
4	1024	16	1	1.4452	2.0746	0.0245
5	2056	16	1	5.4242	7.9929	0.0228
6	128	32	1	0.0515	0.0696	0.0227
7	128	16	2	0.05983	0.0872	0.0471

Table 6.1. Time taken by each compute shader stage in milliseconds.

wavenumber k. We can also note, that the diffusion step is more computationally intensive, compared to the advection step roughly by a factor of 1.4. Then also it is not a surprise, that increasing k the time increases by roughly 2 times.

6.2 Visual comparison

While it is important to have information about the performance of the method using different configurations, we also need to know whether it is even needed to have higher-resolution grids. Figures 6.2, 6.3 and 6.4 show the difference with different spatial, angle and wavenumber resolutions respectively.

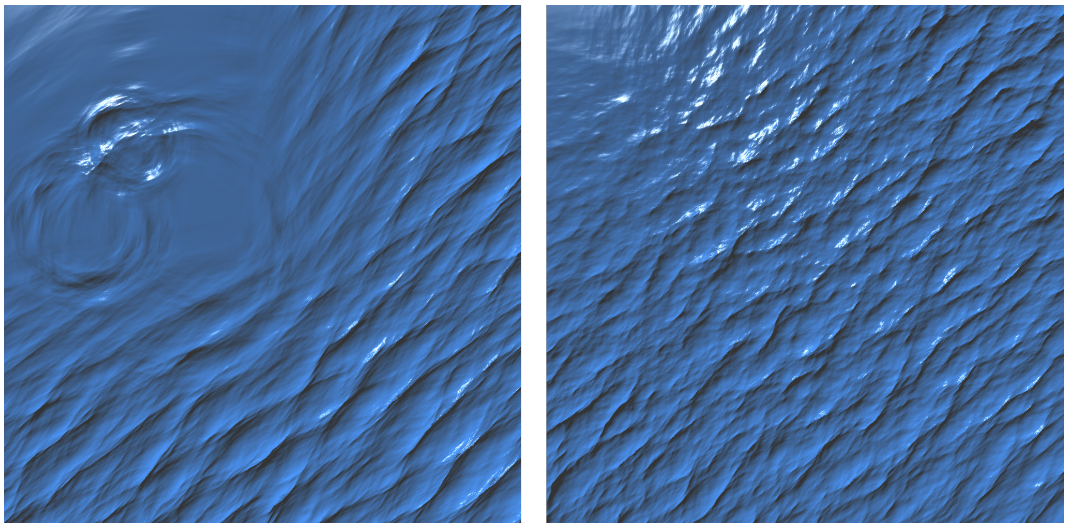

Figure 6.2. Comparison of grids with different spatial resolution, configurations 2 and 3 from the table.6.1.

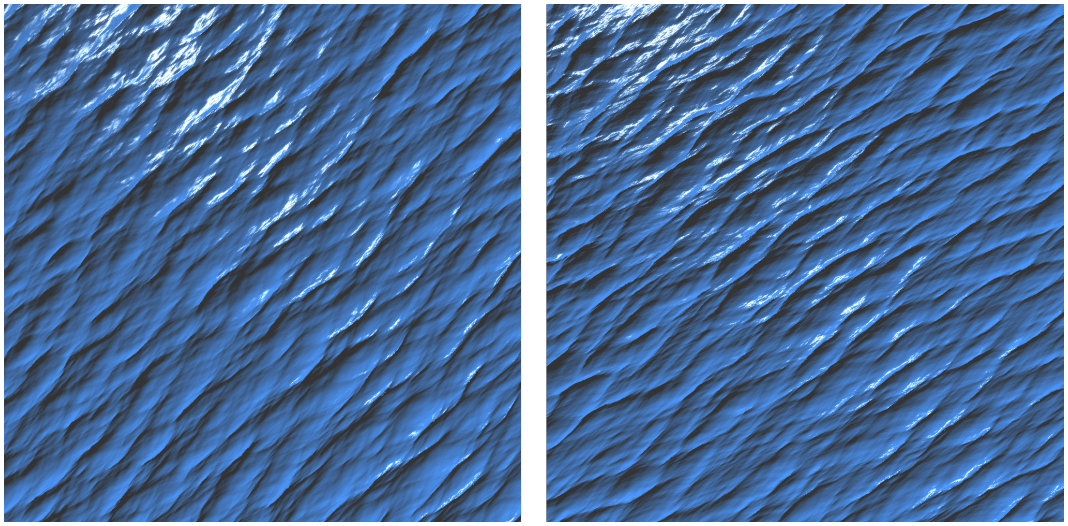

Figure 6.3. Comparison of grids with different angle resolution, configurations 1 and 6 from the table.6.1.

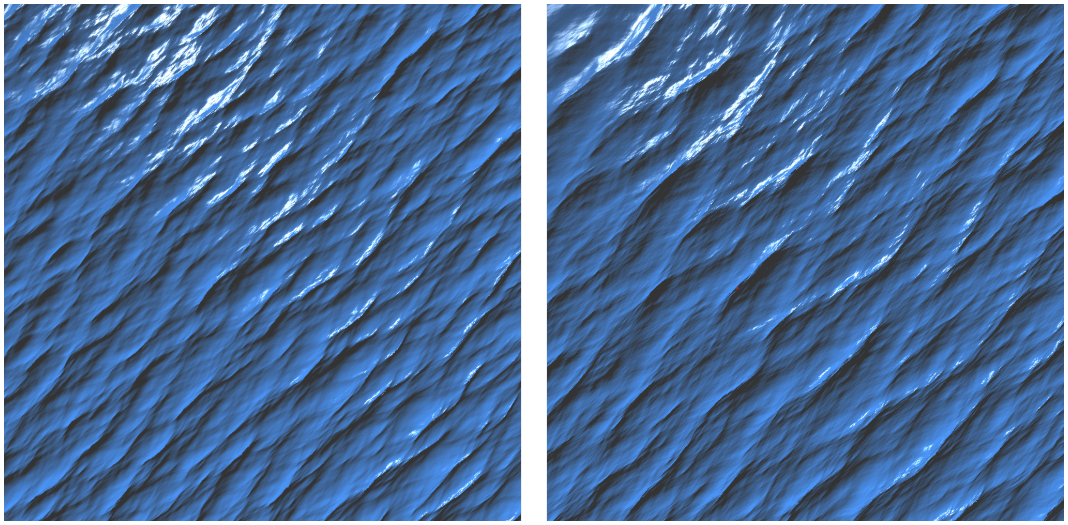

Figure 6.4. Comparison of grids with different k resolution, configurations 1 and 7 from the table.6.1.

The figures show, that increasing the resolution doesn't yield big improvements, while definitely increases simulation time.

6.3 Spectrum comparison

Ocean wave spectrum controls the local details of the waves that we have and, theoretically, we can use any non-directional spectrum in this implementation. The reason we are including non-directional only is that our profile buffer computation relies solely on the wind speed and k wavenumber variables.

In the figures 6.5 and 6.6 you can see the examples of the results produced with the JONSWAP and Tessendorf spectra, discussed and researched in [4], and while it is possible to use any spectra, the results came out not very realistic: the waves

seem too smooth and don't have much variability. Tweaking different constants in spectrum value calculation might help to mitigate this issue.

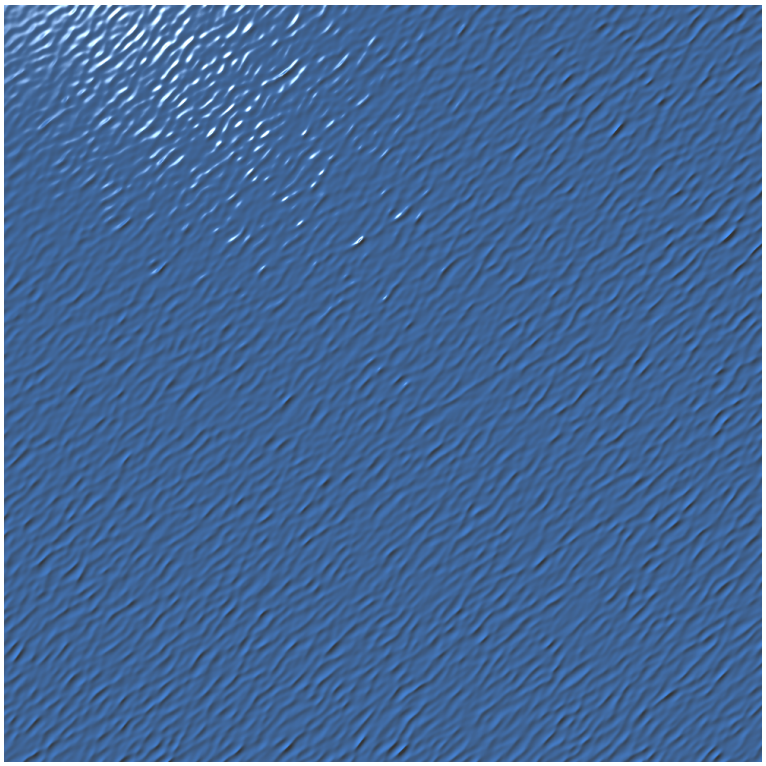

Figure 6.5. Result produced with the Tessendorf spectrum.

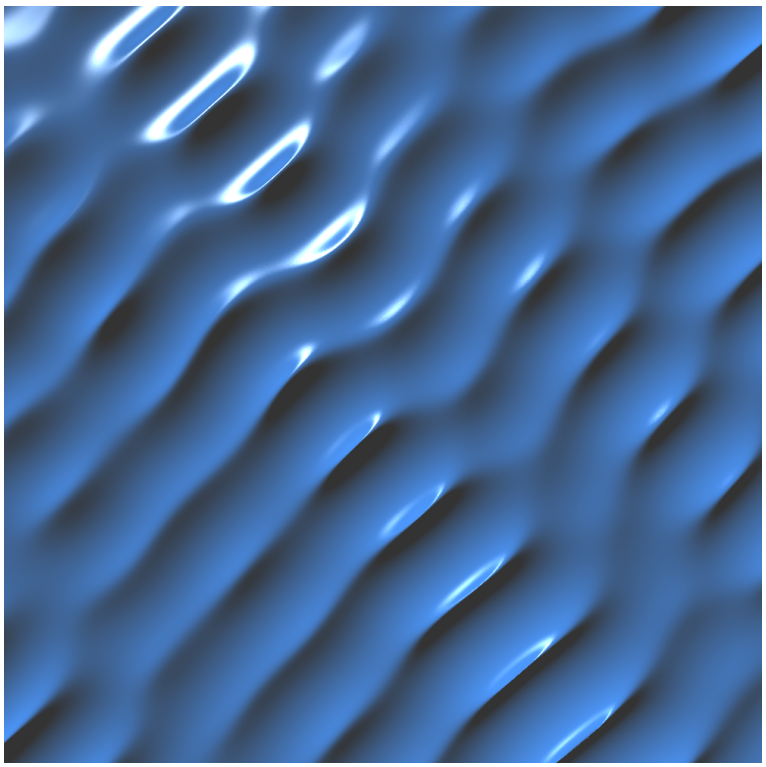

Figure 6.6. Result produced with the JONSWAP spectrum.

6.4 Boundaries

We also have to check how the resulting algorithm deals with the boundaries and, as explained earlier, due to a bad choice of the way to represent boundaries. So with the current approach, we have a scene with a cube in the middle and the results can be seen in the figure 6.7.

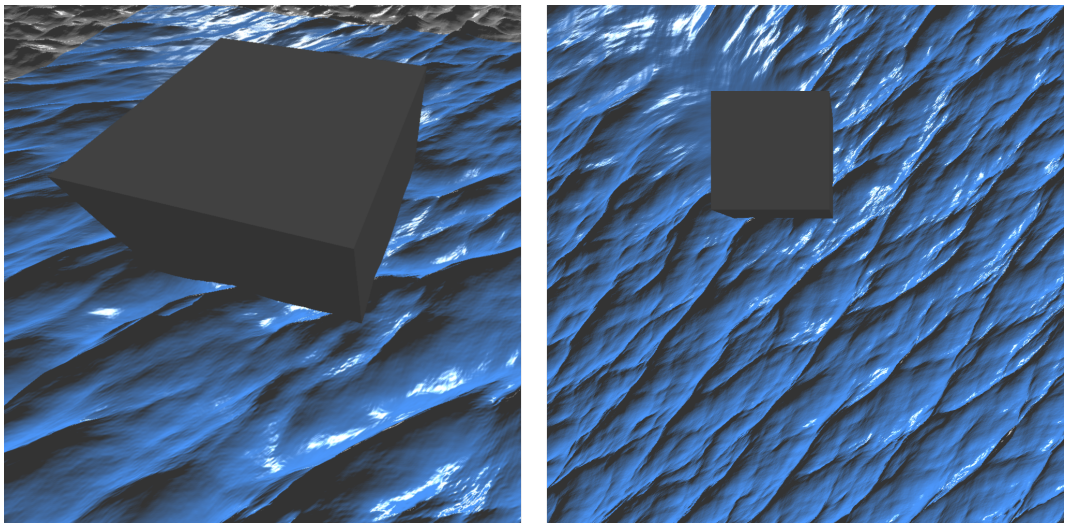

Figure 6.7. Example of boudary reflections.

While it is possible to see slight signs of reflection there, the result is by no means satisfactory and further work will be required. It will be much more beneficial to change the way boundaries are represented altogether. For example, to represent a static environment, a heightmap will be a good choice since we can then compare the height of an environment point with the height of our mesh and additionally estimate a normal vector of the environment in a given point by calculating the gradient in that point. For dynamic boundaries, on the other hand, we can directly modify the values on our amplitude grid, by taking the speed and direction of movement of an object and estimating by how much we have to increase the values in a given direction.

Because of this poor design choice, the requirement for 3 different scenes from the assignment wasn't entirely completed, and there are technically only 2: with a cube boundary and without it.

6.5 Certain parameter significance

As has been mentioned throughout the Design and Implementation chapters, in this section we will demonstrate the significance of some of the parts of the algorithm and how they contribute to the visual quality and how realistic the final result is.

In the figure 6.8 you can see how without the random factor when sampling the profile buffer, the result is very much deterministic and the waves tend to circle around the point `(0, 0)`.

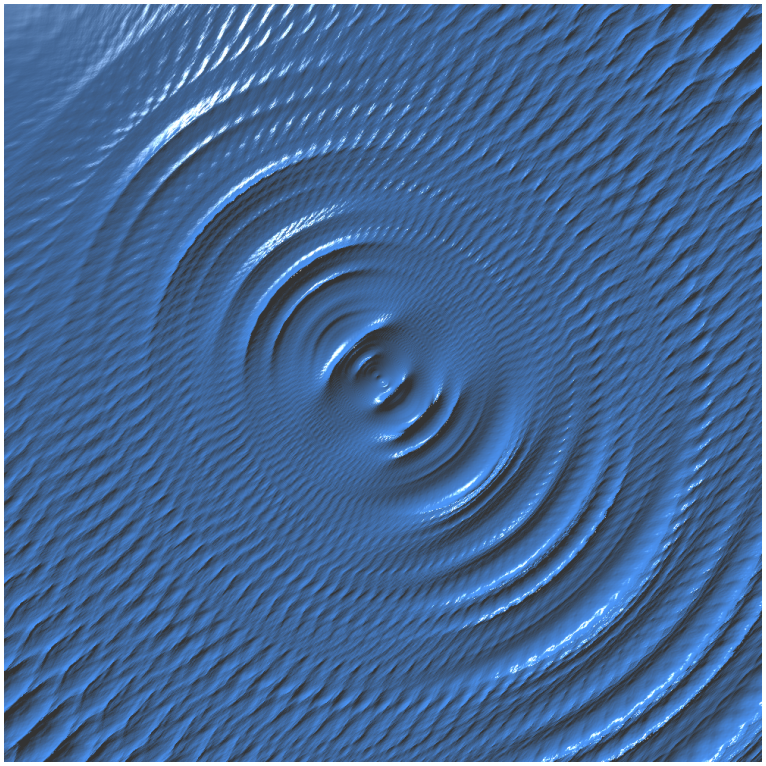

Figure 6.8. The effect produced without random number in profile buffer sampling.

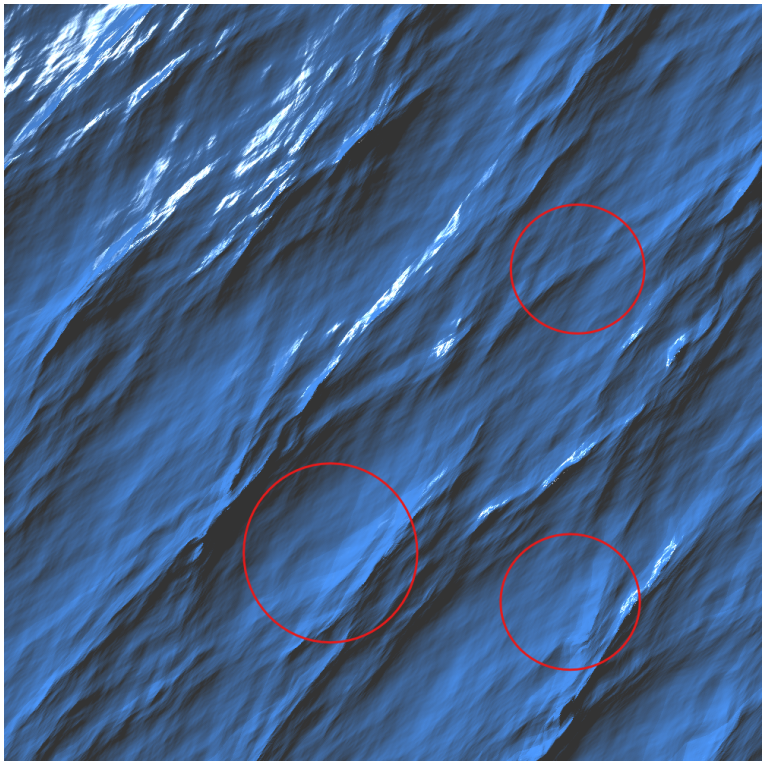

Figure 6.9. The effect produced without interpolation between wave phases.

Another thing we discussed earlier was the interpolation between phases of Gerstner waves during the precomputation of the profile buffer. In the figure 6.9 you can see some regions with sharp edges, that do not show up on the other pictures.

Additionally, while it is not visible in the figure, this way the wave height periodically pulsates, which is far from the ideal result.

6.6 Comparison to the CPU implementation

Finally, in this section, we will compare the previous results to the CPU implementation of this same method. It is worth mentioning, that only the simulation portion of the method was taken from the publicly available code, so the comparison should be fairly reliable since we will be working with the same scene and the same rendering code. The time taken by each simulation stage can be seen in the table 6.2.

No.	Res. X	Res. θ	Res. K	Advection	Diffusion	Profile
1	128	16	1	8.244	3.224	20.889
2	256	16	1	37.442	12.206	19.271
3	512	16	1	81.966	43.603	20.867
4	128	32	1	15.017	6.485	21.292

Table 6.2. Time taken by each stage of CPU implementation in milliseconds.

The most striking difference is that all the values are orders of magnitude higher on the same configurations. Additionally, out of all the stages, the profile buffer seems to have been slowed down the most. We can see that on the base configuration, its computation takes almost 1000 times as much time as it takes on GPU (6.1). From this data, we can deduce that profile buffer , and the grid simulation itself for that matter, were designed to greatly benefit from the parallelization GPU provides.

Chapter 7
Conclusions

In this thesis, we investigated various methods for simulating water surfaces, with a particular focus on the context of computer graphics. We examined the theoretical foundations of fluid dynamics, including the Navier-Stokes equations, and explored analytical, numerical, and hybrid approaches that had been produced by various researchers. We implemented a method using C++ and OpenGL, utilizing compute shaders to achieve efficient real-time performance. We also conducted a series of tests and evaluations to assess the performance and visual quality of the simulated waves.

In conclusion, we relatively successfully developed a procedural water wave simulation application that provides respectable results, while not requiring significant computational resources.

7.1 Ways of improvement

Despite the partial success of the current implementation, there are some areas where it is lacking, or produces unsatisfactory results, requiring further refinement:

- in the final calculation of water displacement and normal vector, the entire range of simulated wave numbers has to be integrated over,
- different spectrum parameters have to be tweaked to produce more realistic results,
- the boundary checking and reflection system has to be reworked, as explained in the Results chapter.

7.2 Future work

And finally, there are ways we can expand this method further, that will add to the overall result but were not in the scope of this thesis.

The amplitude grid can be precomputed and loaded into the textures to have an initial state. With this state of already established ocean, it will be much easier to get the desired results from the very beginning. Also, if we do not need boundary reflections, this method allows us to discard the advection and diffustion altogether. In that case, the wave propagation will not be simulated, and the profile buffer will solely be responsible for the visual quality.

Additionally, the visual quality can be improved in a way, that doesn't directly relate to the simulation, like implementing realistic water lighting and employing reflections and refractions of the light rays.

References

[1] Stefan Jeschke, Tomas Skrivan, Matthias Muller-Fischer, Nuttapong Chentanez, Miles Macklin, and Chris Wojtan. Water surface wavelets. *ACM Trans. Graph.* 2018, 37 (4). DOI 10.1145/3197517.3201336.

[2] Sam Bishop. Sea of Thieves' oceans detailed in latest dev diary. 2016.

[3] Jerry Tessendorf. Simulating Ocean Water. *SIG-GRAPH'99 Course Note*. 2001.

[4] Christopher J. Horvath. *Empirical directional wave spectra for computer graphics*. In: *Proceedings of the 2015 Symposium on Digital Production*. New York, NY, USA: Association for Computing Machinery, 2015. 29–39. ISBN 9781450337182. `https://doi.org/10.1145/2791261.2791267`.

[5] Stefan Jeschke, and Chris Wojtan. Water Wave Animation via Wavefront Parameter Interpolation. *ACM Trans. Graph.* 2015, 34 (3). DOI 10.1145/2714572.

[6] S. Jeschke, C. Hafner, N. Chentanez, M. Macklin, M. Müller-Fischer, and C. Wojtan. Making Procedural Water Waves Boundary-aware. *Computer Graphics Forum*. 2020, 39 (8), 47-54. DOI https://doi.org/10.1111/cgf.14100.

[7] Richard Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM J. Res. Dev.* 1967, 11 215–234. DOI 10.1147/rd.112.0215.

[8] R. Bridson. *Fluid Simulation for Computer Graphics (2nd ed.)*. 2015. `https://doi.org/10.1201/9781315266008`.

[9] Michael Kass, and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. 1990, 24 (4), 49–57. DOI 10.1145/97880.97884.

[10] José Canabal, David Miraut, Nils Thuerey, Theodore Kim, Javier Portilla, and Miguel Otaduy. Dispersion kernels for water wave simulation. *ACM Transactions on Graphics*. 2016, 35 1-10. DOI 10.1145/2980179.2982415.

[11] Cem Yuksel, Donald H. House, and John Keyser. *Wave particles*. In: *ACM SIGGRAPH 2007 Papers*. New York, NY, USA: Association for Computing Machinery, 2007. 99–es. ISBN 9781450378369. `https://doi.org/10.1145/1275808.1276501`.

[12] Stefan Jeschke, and Chris Wojtan. Water wave packets. *ACM Trans. Graph.* 2017, 36 (4). DOI 10.1145/3072959.3073678.

[13] Willard J. Pierson Jr., and Lionel Moskowitz. A proposed spectral form for fully developed wind seas based on the similarity theory of S. A. Kitaigorodskii. *Journal of Geophysical Research (1896-1977)*. 1964, 69 (24), 5181-5190. DOI https://doi.org/10.1029/JZ069i024p05181.

[14] Franz Gerstner. Theorie der Wellen. *Annalen der Physik*. 1809, 32 (8), 412-445. DOI https://doi.org/10.1002/andp.18090320808.

[15] Wikipedia contributors. *Trochoidal wave — Wikipedia, The Free Encyclopedia*. 2023.

https://en.wikipedia.org/w/index.php?title=Trochoidal_wave&oldid=1130928648.

[16] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*. 1975, 18 (6), 311–317. DOI 10.1145/360825.360839.

Appendix A
Instructions

To run the prototype application, testing PC GPU has to support OpenGL 4.6 and extension `GL_ARB_shading_language_include`.

A.1 Compilation

The source codes are located in the attached zip archive.

To compile the application first open solution file `/WaveSimulation.sln` in Microsoft Visual Studio (preferably 2022), choose `Release` configuration and press the build button.

To compile and run press `Ctrl + F5`.

A.2 Useful files and folders

`/WaveSimulation.sln`	Microsoft Visual Studio solution file related to the project.
`/WaveSimulation/src`	Source code for the application.
`/WaveSimulation/assets`	Shaders and textures used in the scene
`/WaveSimulation/src`	
`/utils/parameters.h`	File for setting the grid resolution parameters.

A.3 GUI description

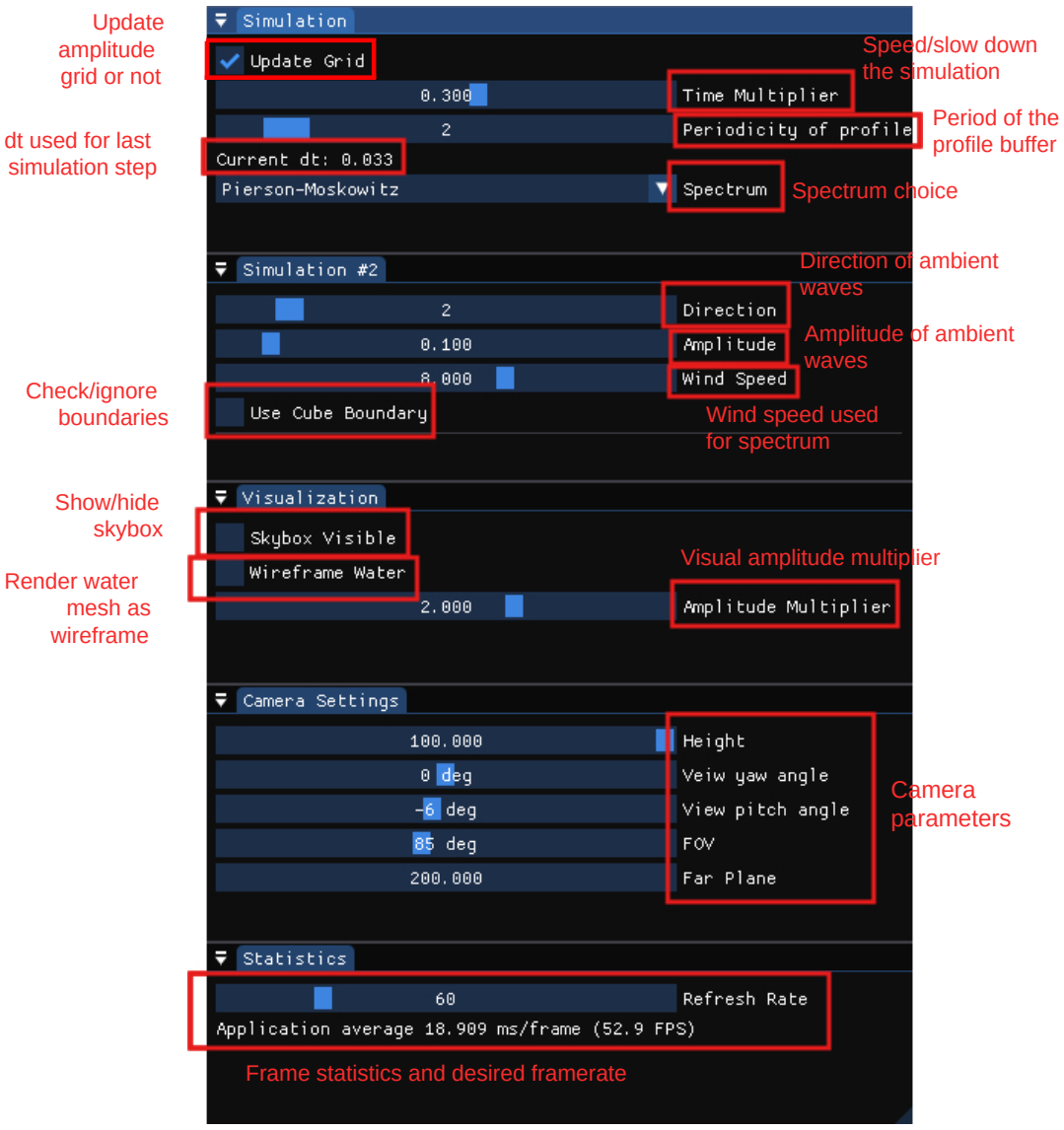

Figure A.1. Description of GUI elements.