Improving the Accuracy of FFT-based GPGPU Ocean Surface Simulations

Master's thesis in Computer science and engineering

Jacob Eriksson
Joakim Wingård

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Improving the Accuracy of FFT-based GPGPU Ocean Surface Simulations

Jacob Eriksson
Joakim Wingård

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Improving the Accuracy of FFT-based GPGPU Ocean Surface Simulations
Jacob Eriksson
Joakim Wingård

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisor: Athanasios Papadimitriou, Rapid Images
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Two speedboats driving across the surface of the interactive ocean simulation that was implemented in Unreal Engine based on our proposed framework.

Typeset in LaTeX
Gothenburg, Sweden 2022

Improving the Accuracy of FFT-based GPGPU Ocean Surface Simulations

Jacob Eriksson
Joakim Wingård
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this paper, we explore how the current state of the art in real-time ocean simulations can be improved in terms of simulation accuracy, while preserving performance.

Current methods, both in academia and in the industry, simulate an ocean model in frequency space on the GPU, convert said model on an approximately frame-by-frame basis to the spatial domain using the Fourier transform, and then read back the resulting heightfield to the CPU as input to the application's physics engine. We propose a fully GPU-based simulation framework that eliminates these GPU readbacks, successfully eliminating the latency-induced simulation errors present in current solutions, while preserving both ocean interactivity and performance. Along this report we also present a prototype of our framework as an Unreal Engine project.

From comparing our proposed framework with the current state of the art, we find:

- a significant correction in simulation accuracy of boats and their wakes;
- near-equivalent GPU performance and improved CPU performance;
- the need to rewrite certain physics behaviors for the GPU that are commonly available as built-in functionality in modern CPU-based physics engines;
- an arguably more complicated implementation.

We conclude that the errors are significant enough to consider in related work and that the proposed approach is worthwhile investigating further in future work.

The prototype code is available at: `https://github.com/NeonSky/master-thesis`

Keywords: computer graphics, 3D graphics, real-time rendering, Fourier transform, FFT, simulation, GPGPU, interactive, ocean wave spectrum, oceanography

Acknowledgements

We would like to thank Athanasios Papadimitriou for acting as our industry advisor and Rapid Images for providing us with the 3D model that we use for our boats. We also thank Erik Sintorn and Ulf Assarsson for their feedback on our half-time presentation and everyone who has proofread and provided feedback on our report, including family and friends.

Finally, we would like to give a special thanks to our academic supervisor, Erik Sintorn, for his very valuable support and advice throughout this project.

Jacob Eriksson and Joakim Wingård, 2022

Contents

List of Figures

List of Tables

1

Introduction

The ocean is a common environment and scenario in real-time applications such as games and interactive simulations [61, 77, 79, 60, 57, 1, 70]. Unfortunately, real oceans are far too large and complex to be simulated and rendered accurately on consumer-grade hardware, especially every frame. Moreover, many real-time applications operate on a strict budget of ∼17 milliseconds of computation time per frame, equating to roughly 60 frames per second (FPS). Notably, this time budget is shared between all intricate systems of the application, leaving an even smaller time budget for ocean simulation. It would thus be of interest to achieve more realistic real-time ocean simulations while minimizing the performance concerns that follow.

An important simplification of the problem is to only render and simulate the surface of the ocean, effectively treating the ocean as a thin surface without any depth beneath it. This measurement naturally sacrifices a magnitude of realism, but it does not entirely exclude underwater behavior nor visuals, as those can be emulated through the use of special effects that are beyond the scope of this report. Using this surface model, it is possible to achieve a simplistic ocean surface by means of scrolling a 2D noise function (or texture) such as Perlin noise [63, 64], Simplex noise [65, 25], or fractional Brownian motion [51]. Although computationally efficient, these methods are not sophisticated enough to imitate the motion and structure of the ocean to the degree of realism that is usually expected from modern applications.

The current state of the art in real-time ocean simulation is to animate the ocean surface according to an empirically-derived model of a real-life ocean, known as an *ocean wave spectrum* (OWS), as described in the pioneering work by Jerry Tessendorf [81]. For physical reasons (see Chapter 2), an OWS (e.g. Phillips [66] or JONSWAP [28]) is an energy spectrum that exists in frequency space, which enforces the simulation of the ocean to also take place in frequency space. Consequently, to utilize this model in the spatial domain (in which the ocean surface is visually perceived), it is necessary to continuously apply the 2D inverse Fourier transform, most efficiently computed using a type of Fast Fourier Transform (FFT) algorithm [20, 23].

Due to hardware limitations, FFTs have traditionally been computed on the CPU, but with the advent of General Purpose GPU (GPGPU) programming, it has now become possible to efficiently compute these in parallel on the GPU, as shown in detail by Fynn-Jorin Flügge in late 2018 [20]. This greatly improves the performance of FFT-based ocean surfaces, but it also introduces a problem: interactivity. The

application usually stores its logical objects (e.g. boats and characters) and simulates physics between these objects on the CPU. The question is: how do we make these CPU-based objects efficiently and accurately interact with the GPU-based ocean, and vice versa?

One solution is to run an identical FFT-based ocean simulation in parallel on the CPU, but using a significantly lower resolution. With low enough resolution, the FFT becomes feasible to compute in real-time, but (1) it still consumes precious CPU cycles, (2) requires more memory and computations overall, (3) does not represent the elevations accurately enough for physics [79], and (4) may easily cause a mismatch between the logical ocean on the CPU and the visual ocean on the GPU. Consequently, this solution is generally avoided [79, 60, 57, 62]. Instead, the current state of the art is to continuously read the ocean state from the GPU *on demand*[1], while the state of external objects is continuously sent on demand from the CPU to the GPU [79, 60, 57, 62]. This is performed in an asynchronous and parallel manner, resulting in high performance even for large resolutions. Unfortunately, this solution introduces a different type of problem: simulation accuracy.

1.1 Problem Definition

The bidirectional interactions present between the ocean and external objects (e.g. boats) imply that the physical simulations of both parties need to occur in a sequential, alternating order:

- to simulate the external objects at time $t + 1$, their state and the ocean state at time t are both needed.
- to simulate the ocean at time $t + 1$, its state and the state of all external objects at time t are both needed.

In the current state-of-the-art solution, these two conditions are violated. The CPU and GPU asynchronously request results from each other on-demand, but the responses to these requests are not expected to arrive back within the same frame [91, 54, 3, 86, 55]. Consequently, the simulation on the CPU is expected to continuously update with respect to ocean states that are a few frames behind, and vice versa for the simulation on the GPU[2]. This introduces a systematic simulation error on both ends, every simulation step, resulting in a greater deviation from the correct simulation as time elapses[3]. Moreover, this accumulated simulation error is expected to become more severe the more frequent the interactions are and the more external objects there are that interact with the ocean simultaneously.

[1] For a non-interactive ocean, it is possible to simulate frames in advance (see [48]), but not when unpredictable (due to user input) bidirectional interactions are present, as considered here.

[2] In particular, speedboats can move quite far across a water surface in a few frames.

[3] Since these errors are correlated, they are not expected to cancel out each other like noise.

One possible solution to this problem is to only update each simulation by one step once its latest asynchronous readback has finished. However, this will cause the ocean and the external objects to update at a frequency limited by the round-trip latency between the CPU and GPU. That is, these parties will update noticeably slower than the rest of the application, resulting in an unnatural stop-motion effect that is not suitable for realistic real-time simulations.

Another possible solution is to send and read the simulation state results between the CPU and GPU in a synchronous manner. This also gets rid of the simulation errors, but may instead be detrimental to the overall performance of the application. The problem with synchronous reads is that they may cause a so-called *graphics pipeline stall* [2, 56, 86], which essentially blocks both the CPU and GPU until the data transaction has finished. This can cause noticeable and inconsistent frame freezes, especially for deeper graphics pipelines where many other systems besides the ocean simulation are competing for GPU resources. This is likely why many applications opt for the asynchronous approach instead [79, 60, 57, 62]. Naturally, the transfer of the simulation data itself also takes time, but judging from the throughput of current PCIe generations [26] and the amount of data to send [23, 79], this transfer speed should not be the bottleneck.

In summary, the current state of the art is efficient but sacrifices a degree of simulation accuracy (and thus realism) due to latency. What we seek to answer in this report are the following two research questions:

- What is the significance of this simulation error in practice?
- Could this simulation accuracy be recovered without hurting performance. And if so, then how?

1.2 Limitations

The topic of ocean simulation is vast and would be impossible to cover in its entirety in a single thesis. We have narrowed our scope to real-time interactivity with the ocean surface, and our simulation will have many limitations, notably:

- The scope is on oceans, not on other types of water bodies e.g. rivers or a glass of water. From a simulation perspective, this also implies that we will only consider the interface between water and air, thereby excluding other fluids and gases that might be of interest in general fluid simulations.
- Our model for simulating the ocean surface originates from a simplified version of the Navier-Stokes equations [81]. This simplification does not support certain extreme topology changes, such as breaking waves in shallow water.
- We will only consider the surface of deep oceans. That is, there will not be any underwater simulations or visuals, shores will not be present, and the depth of the ocean can be considered both uniform and very deep.

- In reality, there is a spectrum of different wave types, where each type is primarily affected by different forces [23]. In this work, similar to most previous work, we mainly focus on *gravitational wind waves*, which are waves whose primary disturbing force is wind and primary restoring force is gravity.
- We will primarily consider the interactions described in Figure 1.1 between the ocean and a speedboat traveling across its surface. But, if time permits, we will also investigate into the interactions that arise from several speedboats.
- In particular, the only ocean-interacting external objects that we will consider in this work are speedboats. However, the ideas that we present should extend to virtually any other object that would float well on the ocean.
- Advanced water rendering effects could easily be its own topic for a thesis, and as such, we do not aim to propose new solutions for that here. This will significantly limit the realism of our visuals.

Figure 1.1: Real-life examples of speedboats interacting with deep ocean water. The ocean and its waves affect the movement and orientation of the boat due to drag and buoyancy. Meanwhile, the boat produces foam, spray, and V-shaped Kelvin wakes as it travels across the ocean surface.

1.3 Goal

The goal and contributions of this thesis are:

- to propose a theoretical framework that improves the accuracy of FFT-based GPGPU ocean surface simulations, by avoiding the simulation errors incurred from latency in the current state of the art, while preserving performance.
- to demonstrate said framework by implementing a prototype in Unreal Engine [19] version 4.26 (C++ and HLSL) of user-controllable speedboats that interact with an ocean surface following our framework.

1.4 Ethical Considerations

After careful consideration, we have not recognized any societal, ethical, or ecological concerns that directly apply to this work, its results, nor its potential future. However, in the broader context, we do recognize that more realistic simulations could ease the creation of false scenarios for the purpose of deception. Realistic oceans by themselves might not have any obvious applications for such use, but they could be used in combination with other realistically simulated and rendered models. Another concern is that such realism would further entice humans away from the "real world", which may negatively affect both physical and mental health. Nevertheless, such concerns plague the entire field of computer graphics, in contrast to the small scope addressed in this report. Moreover, it is important to consider these concerns in the light of all the positive outcomes that realistic computer graphics bring – from medical research and auto-motives, to architecture and entertainment.

2

Theory

In this chapter, we introduce the fundamental theory that our work builds upon. The intention behind this chapter is to help readers that are unfamiliar with the current state of the art in real-time ocean simulations to understand:

- the underlying challenges that exist in the given problem (see Chapter 1);
- the decisions made in Chapter 3;
- the results presented in Chapter 4;
- and how to replicate our work for use in future research.

In particular, we will cover how an ocean can be simulated in real-time using a wave spectrum and how wakes (a type of waves formed by floating objects) can be simulated on top of this model using eWave [84]. We end the chapter with a look at GPU architecture to see why the GPU readbacks used in the current state of the art are either detrimental to performance or introduce latency.

2.1 Ocean Waves From Sinusoids

If we consider the ocean as a homogeneous water fluid of uniform mean depth, then we can model the gravitational wind waves that propagate on this surface using linear wave theory [81, 23]. In this subfield of fluid dynamics, the surface elevations formed by a single wave (component) are modeled using a sinusoid, while the total ocean surface is modeled as a sum of such sinusoids, with one term per wave component. This property will become highly relevant later, as the core idea of FFT-based ocean simulations is to represent the ocean surface as a Fourier series, which is defined as a sum of harmonically related[1] sinusoids.

[1]Harmonically related waves (a.k.a. a harmonic series) is a set of $n \in \mathbb{Z}^+$ waves (e.g. ocean or sound waves) whose frequencies are $\{f, 2f, \ldots, nf\}$, where $f \in \mathbb{R}$ is called the fundamental tone.

Before we attempt to model a 2D ocean surface in 3D space, let us first consider a 1D ocean in 2D space. The elevation $\eta \in \mathbb{R}$ of a single 1D sinusoid (a.k.a. wave component) at spatial coordinate $x \in \mathbb{R}$ and time $t \in \mathbb{R}$ can be described as

$$\eta(x,t) = A\cos(kx + \omega t + \phi) \tag{2.1}$$

where $A \in \mathbb{R}$ is the wave amplitude, $k \in \mathbb{R}$ is called the *wavenumber* (measured in radians per meter), $\omega \in \mathbb{R}$ is the angular frequency, and $\phi \in \mathbb{R}$ is the phase offset.

Similar to how ω acts as a temporal frequency that determines the time period T of the wave (as $T = \frac{2\pi \text{ rad}}{\omega}$), the wavenumber k acts as a spatial frequency that determines the wavelength λ of the wave (as $\lambda = \frac{2\pi \text{ rad}}{k}$). In linear wave theory, these two frequencies have a particular relationship with each other, known as the *dispersion relation* [81, 23, 31]. This relation states that

$$\omega^2 = gk\tanh(kd) \tag{2.2}$$

where $d \in \mathbb{R}$ is the ocean depth and $g \in \mathbb{R}$ is the gravitational acceleration constant. When we consider a deep ocean ($\frac{\lambda}{2} < d$) of uniform depth, this relation simplifies to

$$\begin{aligned} \omega^2 &\approx \lim_{d\to\infty}(gk\tanh(kd)) \\ &= gk \end{aligned} \tag{2.3}$$

Thus, given k, we can determine ω as

$$\omega \approx \sqrt{gk} \tag{2.4}$$

Consequently, when configuring the parameters of a particular wave component, we only need to decide: A, k, and ϕ. This idea also extends into the 3D realm.

When we enter the third dimension, the elevation η of a particular wave component at spatial (horizontal) coordinate $\boldsymbol{x} \in \mathbb{R}^2$ and time t can be described as

$$\eta(\boldsymbol{x},t) = A\cos(\boldsymbol{k}^\top\boldsymbol{x} + \omega t + \phi) \tag{2.5}$$

Comparing with Equation 2.1, the spatial coordinate x is now a vector $\boldsymbol{x}$ (given in m^2 instead of m) and the wavenumber k has been replaced by a *wavevector* $\boldsymbol{k}$. As the magnitude of $\boldsymbol{k}$ determines the spatial frequency, we have $k = ||\boldsymbol{k}||$. Moreover, the direction of $\boldsymbol{k}$ (computed as $\frac{\boldsymbol{k}}{k}$) determines the propagation direction of the wave. Thus, in the 3D case we instead need to decide: A, $\boldsymbol{k}$, and ϕ. Naturally, time remains 1-dimensional in 3D space, so the angular frequency ω remains a scalar.

We may identify each wave component in our ocean model using only its wavevector $\boldsymbol{k}$, because, as we will later see in Section 2.2, each wave component has a unique $\boldsymbol{k}$. We will therefore choose to denote the parameters of a particular wave component by using $\boldsymbol{k}$ as a subscript, e.g. its amplitude as $A_{\boldsymbol{k}}$ and its elevation function (Equation 2.5) as $\eta_{\boldsymbol{k}}$. We can then describe the total ocean elevation h at horizontal position $\boldsymbol{x}$ and time t as

$$h(\boldsymbol{x}, t) = \sum_{\boldsymbol{k}} \eta_{\boldsymbol{k}}(\boldsymbol{x}, t) = \sum_{\boldsymbol{k}} A_{\boldsymbol{k}} \cos(\boldsymbol{k}^\top \boldsymbol{x} + \omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}}) \tag{2.6}$$

which, assuming the $\eta_{\boldsymbol{k}}$ components are harmonically related, is a Fourier series.

Fourier analysis, introduced by Joseph Fourier [35], shows us that a finite interval (called the period) of any arbitrary function can be accurately approximated[2] as a Fourier series. Consequently, if the surface of a real-life ocean is approximated as a heightfield, we may accurately express any possible state of this ocean using Equation 2.6. The remaining challenges are thus:

- to ensure that the individual wave components $\eta_{\boldsymbol{k}}$ are harmonically related;
- and to configure the parameters of each $\eta_{\boldsymbol{k}}$ such that their sum accurately replicates the movement and appearance of a real-life ocean.

This is where the ocean wave spectrums come into play.

2.2 Ocean Wave Spectrums

The common goal of all ocean simulation models is to efficiently replicate the behavior of real-life oceans. FFT-based oceans accomplish this through a so-called Ocean Wave Spectrum (OWS) [23]. Formally, an OWS is defined as the power spectral density (PSD), a.k.a. power spectrum, of wave amplitudes observed from a real-life ocean [31]. That is, it describes the time-average energy of a particular ocean configuration as a continuous function distributed across wave components.

As an example, let $S(\boldsymbol{k})$ be an OWS in wavevector space. If the ocean configuration (a.k.a. state) that $S(\boldsymbol{k})$ was derived from can be expressed as a sum of the following two wave components:

- $\eta_{\boldsymbol{k}_1}(\boldsymbol{x}, t) = A_{\boldsymbol{k}_1} \cos({\boldsymbol{k}_1}^\top \boldsymbol{x} + \omega_{\boldsymbol{k}_1} t + \phi_{\boldsymbol{k}_1})$
- $\eta_{\boldsymbol{k}_2}(\boldsymbol{x}, t) = A_{\boldsymbol{k}_2} \cos({\boldsymbol{k}_2}^\top \boldsymbol{x} + \omega_{\boldsymbol{k}_2} t + \phi_{\boldsymbol{k}_2})$

and if $A_{\boldsymbol{k}_1} > A_{\boldsymbol{k}_2}$, then $S(\boldsymbol{k}_1) > S(\boldsymbol{k}_2)$ would follow, as the wave component $\boldsymbol{k}_1$ has a greater amplitude, and thereby greater time-average energy[3]. Moreover, $S(\boldsymbol{k}) = 0$ would hold for any $\boldsymbol{k} \notin \{\boldsymbol{k}_1, \boldsymbol{k}_2\}$ as no such component composed the ocean state.

[2] The accuracy of the approximation improves as the set of summed sinusoids grows.

[3] As sinusoids are periodic functions, the amplitude directly correlates with time-average energy.

By distributing the time-average energy across wavevectors, the OWS $S(\boldsymbol{k})$ of an ocean state will (as we will see in the following subsections) essentially allow us to reverse engineer the parameters of the individual wave components $\eta_{\boldsymbol{k}}$ that would best represent the surface of that ocean (as expressed by Equation 2.6).

We will now begin by explaining how an OWS can be derived from a real ocean and then proceed by explaining how it can be used to animate a virtual ocean.

2.2.1 Deriving an Ocean Wave Spectrum

Consider an ocean that has been subjected to a certain set of conditions (wind, temperature, etc.) for an extended period of time (e.g. a week). This ocean has entered a so-called *steady state* [23], where additional time subjected to the present conditions will no longer affect the shape of the ocean in a statistically significant manner [81]. That is, the ocean is a *fully developed ocean* with temporally invariant statistical properties (as long as the mentioned conditions remain). Furthermore, if we consider a homogeneous fluid with even depth (i.e. seabed), then the ocean surface's statistical properties are also spatially homogeneous. This kind of ocean state is of particular interest, as its statistical properties are relatively easy to capture.

By using an elevation-measurement instrument such as a buoy or a wave staff, it is possible to record the elevation $h(\boldsymbol{x}, t)$ of an ocean surface over a time span $t \in [t_s, t_e]$ at a fixed[4] horizontal position $\boldsymbol{x}$. If the measured ocean is a fully developed ocean, then $\boldsymbol{x}$ itself is irrelevant as long as it is fixed (due to the spatial homogeneity), while t_s and t_e are both irrelevant[5] as long as $[t_s, t_e]$ is within the time period during which the ocean exhibits its steady state (due to the temporal invariance). Then, according to linear wave theory and Fourier analysis [81], we can decompose this elevation function $h(\boldsymbol{x}, t)$ into a spectrum $\hat{h}(\boldsymbol{k}, t)$ in wavevector space, by applying the Fourier transform. In practice, this will be the discrete Fourier transform as the physical instruments used to record $h(\boldsymbol{x}, t)$ will produce discrete sequences of elevation measurements spaced by a fixed time interval[6], due to hardware limitations.

Let $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the discrete Fourier transform and the inverse discrete Fourier transform, respectively. Then, through an FFT algorithm, we can compute the Fourier components of $h(\boldsymbol{x}, t)$, and vice versa, as

$$\begin{aligned} \mathcal{F}\left(h(\boldsymbol{x}, t)\right) &= \hat{h}(\boldsymbol{k}, t) \\ &\Longleftrightarrow \\ h(\boldsymbol{x}, t) &= \mathcal{F}^{-1}\left(\hat{h}(\boldsymbol{k}, t)\right) \end{aligned} \tag{2.7}$$

[4] A fixed $\boldsymbol{x}$ can be ensured by strapping the buoy or wave staff to a fixed structure, like a pole, or a rope between two poles.

[5] The duration $t_e - t_s$ is still relevant though, as more samples (possibly averaged over multiple measurements) help reduce noise and capture long-frequency waves [23].

[6] According to the Nyquist-Shannon sampling theorem, this sampling frequency of the elevation-measuring instrument needs to exceed $2B$ samples per second, in order to perfectly capture and reconstruct wave components at frequencies as high as B hertz. In general, more advanced instruments with higher sampling rates will ultimately result in a more accurate OWS.

As this will simplify the upcoming derivation, we will assume that the continuous function $h(\boldsymbol{x}, t)$ was captured as a discrete sequence of $2N^2$ elevation samples, for some arbitrary $N \in \mathbb{Z}^+$. By now rewriting Equation 2.6 as an inverse discrete Fourier transform of $2N^2$ Fourier components (i.e. assuming that $h(\boldsymbol{x}, t)$ can be accurately represented with N^2 wave components), we find that:

$$\begin{aligned}
h(\boldsymbol{x}, t) &= \sum_{\boldsymbol{k}} \eta_{\boldsymbol{k}}(\boldsymbol{x}, t) \\
&= \sum_{\boldsymbol{k}} A_{\boldsymbol{k}} \cos(\boldsymbol{k}^\top \boldsymbol{x} + \omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}}) \\
&= \sum_{n=1}^{N^2} A_{\boldsymbol{k}_n} \cos({\boldsymbol{k}_n}^\top \boldsymbol{x} + \omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n}) \\
&= \sum_{n=1}^{N^2} A_{\boldsymbol{k}_n} \frac{e^{i(\boldsymbol{k}_n^\top \boldsymbol{x} + \omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})} + e^{-i(\boldsymbol{k}_n^\top \boldsymbol{x} + \omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})}}{2} \\
&= \sum_{n=-N^2}^{N^2} \frac{A_{\boldsymbol{k}_n} \cdot e^{i(\omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})}}{2} \cdot e^{i(\boldsymbol{k}_n^\top \boldsymbol{x})} \\
&= \frac{1}{2N^2} \sum_{n=-N^2}^{N^2} 2N^2 \cdot \frac{A_{\boldsymbol{k}_n} \cdot e^{i(\omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})}}{2} \cdot e^{i(\boldsymbol{k}_n^\top \boldsymbol{x})} \\
&= \frac{1}{2N^2} \sum_{n=-N^2}^{N^2} N^2 A_{\boldsymbol{k}_n} e^{i(\omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})} \cdot e^{i(\boldsymbol{k}_n^\top \boldsymbol{x})} \\
&= \frac{1}{2N^2} \sum_{n=-N^2}^{N^2} \hat{h}(\boldsymbol{k}_n, t) \cdot e^{i(\boldsymbol{k}_n^\top \boldsymbol{x})} \\
&= \mathcal{F}^{-1}\left(\hat{h}(\boldsymbol{k}, t)\right)
\end{aligned} \tag{2.8}$$

where the following notation is used:

- $A_{\boldsymbol{k}_0} = 0$
- $A_{\boldsymbol{k}_{-n}} = A_{\boldsymbol{k}_n}$
- $\boldsymbol{k}_{-n} = -\boldsymbol{k}_n$
- $\phi_{\boldsymbol{k}_{-n}} = -\phi_{\boldsymbol{k}_n}$
- $\hat{h}(\boldsymbol{k}_{-n}, t) = \overline{\hat{h}(\boldsymbol{k}_n, t)}$, where the overline symbol denotes the complex conjugate.

We also make use of the following law:

$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2} \tag{2.9}$$

which is derived from Euler's formula:

$$\begin{aligned}
e^{ix} &= \cos(x) + i \sin(x) \\
e^{-ix} &= \cos(-x) + i \sin(-x) = \cos(x) - i \sin(x)
\end{aligned} \tag{2.10}$$

The main takeaway from Equation 2.8, is that we now have the following expression for the Fourier components:

$$\mathcal{F}\left(h(\boldsymbol{x},t)\right) = \mathcal{F}\left(\mathcal{F}^{-1}\left(\hat{h}(\boldsymbol{k},t)\right)\right) = \hat{h}(\boldsymbol{k},t) = N^2 A_{\boldsymbol{k}} e^{i(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})} \tag{2.11}$$

As shown in Section 16.3 of *Introduction to Physical Oceanography* [76], the discrete wave spectrum P is defined as a discrete estimate of the $h(\boldsymbol{x},t)$ signal's true and continuous PSD, known as a periodogram. This discrete estimation results in the following definition:

$$\begin{aligned}
P(\boldsymbol{k}_n) &= \frac{1}{(2N^2)^2} \cdot (|\hat{h}(\boldsymbol{k}_n,t)|^2 + |\hat{h}(\boldsymbol{k}_{-n},t)|^2) \\
&= \frac{1}{4N^4} \cdot (|\hat{h}(\boldsymbol{k}_n,t)|^2 + |\hat{h}(\boldsymbol{k}_{-n},t)|^2) \\
&= \frac{1}{4N^4} \cdot (|\hat{h}(\boldsymbol{k}_n,t)|^2 + |\overline{\hat{h}(\boldsymbol{k}_n,t)}|^2) \\
&= \frac{1}{4N^4} \cdot (|\hat{h}(\boldsymbol{k}_n,t)|^2 + |\hat{h}(\boldsymbol{k}_n,t)|^2) \\
&= \frac{1}{4N^4} \cdot 2|\hat{h}(\boldsymbol{k}_n,t)|^2 \\
&= \frac{1}{2N^4} \cdot |\hat{h}(\boldsymbol{k}_n,t)|^2 \\
&= \frac{1}{2N^4} \cdot |N^2 A_{\boldsymbol{k}_n} e^{i(\omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})}|^2 \\
&= \frac{1}{2N^4} \cdot N^4 A_{\boldsymbol{k}_n}^2 |e^{i(\omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})}|^2 \\
&= \frac{A_{\boldsymbol{k}_n}^2}{2} \cdot |e^{i(\omega_{\boldsymbol{k}_n} t + \phi_{\boldsymbol{k}_n})}|^2 \\
&= \frac{A_{\boldsymbol{k}_n}^2}{2} \cdot 1^2 \\
&= \frac{A_{\boldsymbol{k}_n}^2}{2}
\end{aligned} \tag{2.12}$$

Then, to create the final (continuous) wave spectrum S, one fits a continuous function to P, e.g. by using the trapezoidal rule and then manually designing a computationally efficient function that fits the resulting shape. As such, we obtain

$$\forall \boldsymbol{k} \left[S(\boldsymbol{k}) \approx \frac{A_{\boldsymbol{k}}^2}{2} \right] \tag{2.13}$$

where the domain and image of $S(\boldsymbol{k})$ are both continuous. The important thing to note here is that the wave spectrum $S(\boldsymbol{k})$ gives us the square amplitude divided by two. This might seem odd, but it actually causes the wave spectrum to be proportional to the energy of the wave components.

In linear wave theory, the mean wave-energy density per unit horizontal area ($\mathrm{kg/s^2} = \mathrm{J/m^2}$) for a single wave component is [47, 38]:

$$\begin{aligned} E &= E_p + E_k \\ &= \frac{\rho g a^2}{4} + \frac{\rho g a^2}{4} \\ &= \frac{\rho g a^2}{2} \\ &\propto \frac{a^2}{2} \end{aligned} \tag{2.14}$$

where

- E_p is the potential energy contribution (in $\mathrm{J/m^2}$).
- E_k is the kinetic energy contribution (in $\mathrm{J/m^2}$).
- $\rho \approx 1000$ is the density of water (in $\mathrm{kg/m^3}$).
- $g \approx 9.82$ is the gravity acceleration constant (in $\mathrm{m/s^2}$).
- a is the wave amplitude (in meters).

This tangent about mean wave energy is of interest to us as it ensures that the total energy of the system (i.e. the ocean) will be preserved over space and time if we simulate the ocean using a static wave spectrum S (assuming fixed ρ and g).

With the ocean wave spectrum $S(\boldsymbol{k})$ now at hand, it is time to apply it.

2.2.2 Applying an Ocean Wave Spectrum

Up to this point we have considered how a real-life ocean can be analyzed using Fourier transforms in order to derive an OWS model of it in the frequency domain (more specifically wavevector space). Now, we will consider how we can construct and animate our own stochastic, but statistically equivalent, virtual ocean using this model. In other words, we are henceforth solely concerned with constructing our own $h(\boldsymbol{x}, t)$ function through Fourier synthesis, as opposed to analyzing an existing $h(\boldsymbol{x}, t)$ function through Fourier analysis (as in Section 2.2.1). This means that we can now simplify our expression of $h(\boldsymbol{x}, t)$ as:

$$
\begin{aligned}
h(\boldsymbol{x},t) &= \mathcal{R}(h(\boldsymbol{x},t)) \\
&= \mathcal{R}(\sum_{\boldsymbol{k}} \eta_{\boldsymbol{k}}(\boldsymbol{x},t)) \\
&= \sum_{\boldsymbol{k}} \mathcal{R}(\eta_{\boldsymbol{k}}(\boldsymbol{x},t)) \\
&= \sum_{n=1}^{N^2} \mathcal{R}(A_{\boldsymbol{k}_n} \frac{e^{i(\boldsymbol{k}_n^\top \boldsymbol{x}+\omega_{\boldsymbol{k}_n} t+\phi_{\boldsymbol{k}_n})} + e^{-i(\boldsymbol{k}_n^\top \boldsymbol{x}+\omega_{\boldsymbol{k}_n} t+\phi_{\boldsymbol{k}_n})}}{2}) \\
&= \sum_{n=1}^{N^2} \frac{A_{\boldsymbol{k}_n}}{2} \mathcal{R}(e^{i(\boldsymbol{k}_n^\top \boldsymbol{x}+\omega_{\boldsymbol{k}_n} t+\phi_{\boldsymbol{k}_n})} + e^{-i(\boldsymbol{k}_n^\top \boldsymbol{x}+\omega_{\boldsymbol{k}_n} t+\phi_{\boldsymbol{k}_n})}) \\
&= \sum_{n=1}^{N^2} \frac{A_{\boldsymbol{k}_n}}{2} (2\mathcal{R}(e^{i(\boldsymbol{k}_n^\top \boldsymbol{x}+\omega_{\boldsymbol{k}_n} t+\phi_{\boldsymbol{k}_n})})) \\
&= \sum_{n=1}^{N^2} A_{\boldsymbol{k}_n} \mathcal{R}(e^{i(\boldsymbol{k}_n^\top \boldsymbol{x}+\omega_{\boldsymbol{k}_n} t+\phi_{\boldsymbol{k}_n})}) \\
&= \mathcal{R}\left(\sum_{n=1}^{N^2} A_{\boldsymbol{k}_n} e^{i(\boldsymbol{k}_n^\top \boldsymbol{x}+\omega_{\boldsymbol{k}_n} t+\phi_{\boldsymbol{k}_n})})\right) \\
&= \mathcal{R}\left(\sum_{\boldsymbol{k}} A_{\boldsymbol{k}} e^{i(\boldsymbol{k}^\top \boldsymbol{x}+\omega_{\boldsymbol{k}} t+\phi_{\boldsymbol{k}})}\right) \\
&= \mathcal{R}\left(\sum_{\boldsymbol{k}} A_{\boldsymbol{k}} e^{i(\omega_{\boldsymbol{k}} t+\phi_{\boldsymbol{k}})} \cdot e^{i\boldsymbol{k}^\top \boldsymbol{x}}\right) \\
&= \mathcal{R}\left(\sum_{\boldsymbol{k}} \tilde{h}(\boldsymbol{k},t) e^{i\boldsymbol{k}^\top \boldsymbol{x}}\right)
\end{aligned}
\tag{2.15}
$$

which is now composed of only N^2 Fourier components, instead of $2N^2$. Here we use $\mathcal{R}(z)$ to denote the real part of $z \in \mathbb{C}$, i.e. $\mathcal{R}(x+iy) = x$, and we use $\tilde{h}(\boldsymbol{k},t)$ to distinguish from $\hat{h}(\boldsymbol{k},t)$ used in Section 2.2.1.

To ensure that the individual wave components are harmonically related, one may choose the set of wavevectors $\boldsymbol{k}$ as described by Gamper et al. [81, 20, 23]. First, consider a finite square plane of side length L meters that spans along the XZ-plane and is centered at the origin. This surface will represent the ocean surface at rest. Then, divide this surface into an evenly spaced grid of N^2 vertices[7] (where N is a power of 2) whose individual elevations (Y-axis coordinates) will be animated[8].

[7] In practice, we can not simulate a continuous spectrum (i.e. infinite amount) of wave components. Consequently, the spatial samples will be limited by the number of discrete wavevectors.

[8] The elevation $h(\boldsymbol{x},t)$ of horizontal positions $\boldsymbol{x}$ between these grid points may be interpolated based on the grid points that are closest to $\boldsymbol{x}$.

In the spatial domain, we then have the N^2 grid points

$$\boldsymbol{x} = \langle x, z \rangle \in \{(\alpha\Delta x, \beta\Delta z) \mid (\frac{-N}{2}, \frac{-N}{2}) \leq (\alpha \in \mathbb{Z}, \beta \in \mathbb{Z}) < (\frac{N}{2}, \frac{N}{2})\} \tag{2.16}$$

for which we associate the following N^2 wavevectors in the frequency domain:

$$\boldsymbol{k} = \langle k_x, k_z \rangle \in \{(\alpha\Delta k_x, \beta\Delta k_z) \mid (\frac{-N}{2}, \frac{-N}{2}) \leq (\alpha \in \mathbb{Z}, \beta \in \mathbb{Z}) < (\frac{N}{2}, \frac{N}{2})\} \tag{2.17}$$

where

- $\Delta x = \Delta z = \frac{L}{N}$
- $\Delta k_x = \Delta k_z = \frac{2\pi}{L}$

As FFT-based oceans inherit the periodic properties of Fourier series, it follows that

$$\begin{aligned} h(\boldsymbol{x}, t) &= h(\boldsymbol{x} + \langle N\Delta x, 0 \rangle, t) = h(\boldsymbol{x} + \langle L, 0 \rangle, t) \\ h(\boldsymbol{x}, t) &= h(\boldsymbol{x} + \langle 0, N\Delta z \rangle, t) = h(\boldsymbol{x} + \langle 0, L \rangle, t) \end{aligned} \tag{2.18}$$

which makes the ocean seamlessly and infinitely tileable, as well as well-defined for all possible values of $\boldsymbol{x} \in \mathbb{R}^2$, assuming interpolation is used.

The choice of N and L has implications both in terms of performance and reconstruction quality. A greater L results in larger periodic ocean patches and thereby less apparent repetitions. However, simply increasing L will reduce local quality as the N^2 grid points will be stretched apart, akin to naively upscaling a 2D image. Meanwhile, increasing N results in greater visual fidelity, but it also hurts performance as the number of wavevectors to simulate scales by $\mathcal{O}(N^2)$.

As motivated in pages 29-31 of *Ocean Surface Generation and Rendering* [23], the choice of N and L implicitly determines the range of fully reconstructable wavelengths according to the Nyquist-Shannon sampling theorem. Knowing this range is useful as it determines the sub-image of the OWS, whose domain is $\mathbb{R}^2$, that will be preserved in the simulation[9]. In particular, the smallest fully reconstructable wavelength will be

$$\lambda_{\min} = 2\frac{L}{N} \tag{2.19}$$

which by $\lambda = \frac{2\pi \text{ rad}}{k}$ implies that the range of simulated wavevectors will be $[-k_{\max}, k_{\max}]$, where

$$k_{\max} = \frac{\pi N}{L} \tag{2.20}$$

[9]The range also reveals at what camera distances the ocean may suitably be viewed from.

Equation 2.17 gives us a harmonic series of wavevectors to be used in Equation 2.15. Hence, the remaining challenge is to configure $A_{\boldsymbol{k}}$ and $\phi_{\boldsymbol{k}}$ of each wave component such that the result resembles the appearance and motion of a real ocean[10]. In the case of a stochastic ocean, the phase offsets $\phi_{\boldsymbol{k}}$ may be uniformly sampled from the range $[0, 2\pi)$, as done in Gamper et al. [81, 23]. As for $A_{\boldsymbol{k}}$, one might expect $\sqrt{2S(\boldsymbol{k})}$ to be a reasonable choice (see Equation 2.13), but there are actually two problems with this approach:

1. It is not stochastic, which reduces the flexibility[11] of the simulation.
2. $S(\boldsymbol{k})$ is continuous, representing the energy associated with $\boldsymbol{k}$ and no other wavevectors, while $A_{\boldsymbol{k}}$ represents the amplitude, or mean-wave energy, associated with $\boldsymbol{k}$ and any nearby wavevectors[12], due to the grid discretization of the simulation space outlined by Equation 2.16 and Equation 2.17.

According to Tessendorf [81], statistical analysis of ocean measurements has shown that the elevation components $\tilde{h}(\boldsymbol{k}, t)$ are approximately temporally-stationary i.i.d. circularly-symmetric complex central Gaussian random variables. Naturally, their mean can then be approximated as 0. The variance, on the other hand, is a bit more involved, but can be derived as follows:

$$\begin{aligned}
\mathrm{Var}[\tilde{h}(\boldsymbol{k}, t)] &= \mathrm{Var}[A_{\boldsymbol{k}} e^{i(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})}] \\
&= A_{\boldsymbol{k}}^2 \cdot \mathrm{Var}[e^{i(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})}] \\
&= A_{\boldsymbol{k}}^2 \cdot (\mathrm{Var}[\cos(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})] + \mathrm{Var}[\sin(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})]) \\
&= A_{\boldsymbol{k}}^2 \cdot (\mathbb{E}[\cos^2(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})] + \mathbb{E}[\sin^2(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})]) \\
&= A_{\boldsymbol{k}}^2 \cdot (\mathbb{E}[\frac{\cos(2(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})) + 1}{2}] + \mathbb{E}[1 - \cos^2(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})]) \\
&= A_{\boldsymbol{k}}^2 \cdot (\frac{1}{2} + (1 - \frac{1}{2})) \\
&= A_{\boldsymbol{k}}^2 \cdot 1 \\
&= A_{\boldsymbol{k}}^2
\end{aligned} \tag{2.21}$$

[10] Recall that $\omega_{\boldsymbol{k}}$ can be directly derived from $\boldsymbol{k}$ using the dispersion relation.

[11] In particular, with a stochastic model, we obtain a population of oceans, which makes it possible to present a new ocean each time the application is run. It also becomes possible to explore said population in order to hand-pick oceans of desired appearance.

[12] Nearby, as in a small Euclidean distance between them.

Consequently, the variance of the entire ocean can then be expressed as

$$\begin{aligned}
\mathrm{Var}[h(\boldsymbol{x},t)] &= \mathrm{Var}[\mathcal{R}(\sum_{\boldsymbol{k}} \tilde{h}(\boldsymbol{k},t)e^{i\boldsymbol{k}^\top \boldsymbol{x}})] \\
&= \sum_{\boldsymbol{k}} \mathrm{Var}[\mathcal{R}(\tilde{h}(\boldsymbol{k},t)e^{i\boldsymbol{k}^\top \boldsymbol{x}})] \\
&= \sum_{\boldsymbol{k}} \mathrm{Var}[\mathcal{R}(A_{\boldsymbol{k}}e^{i(\omega_{\boldsymbol{k}}t+\phi_{\boldsymbol{k}})}e^{i\boldsymbol{k}^\top \boldsymbol{x}})] \\
&= \sum_{\boldsymbol{k}} \mathrm{Var}[A_{\boldsymbol{k}}\mathcal{R}(e^{i(\boldsymbol{k}^\top \boldsymbol{x}+\omega_{\boldsymbol{k}}t+\phi_{\boldsymbol{k}})})] \\
&= \sum_{\boldsymbol{k}} A_{\boldsymbol{k}}^2 \cdot \mathrm{Var}[\mathcal{R}(e^{i(\boldsymbol{k}^\top \boldsymbol{x}+\omega_{\boldsymbol{k}}t+\phi_{\boldsymbol{k}})})] \\
&= \sum_{\boldsymbol{k}} \frac{A_{\boldsymbol{k}}^2}{2}
\end{aligned} \tag{2.22}$$

By Plancherel's theorem [95], and the definition of the PSD, the variance of $h(\boldsymbol{x},t)$ is equal to the energy captured by its PSD (i.e. the OWS). Thus, by using Equation 2.22, we can now derive an approximation for $A_{\boldsymbol{k}}$ that approximately preserves the energy of the OWS, as shown below:

$$\begin{aligned}
\mathrm{Var}[h(\boldsymbol{x},t)] &= \int_{\omega=0}^{\infty} S(\omega)\ \mathrm{d}\omega \\
&= \iint_{\boldsymbol{k}} S(\boldsymbol{k})\ \mathrm{d}\boldsymbol{k} \\
&\approx \iint_{\boldsymbol{k}=\langle -k_{\max}, -k_{\max}\rangle}^{\langle k_{\max}, k_{\max}\rangle} S(\boldsymbol{k})\ \mathrm{d}\boldsymbol{k} \\
&= \sum_{\alpha,\beta} \iint_{\boldsymbol{k}=\langle \alpha\Delta k_x, \beta\Delta k_z\rangle}^{\langle (\alpha+1)\Delta k_x, (\beta+1)\Delta k_z\rangle} S(\boldsymbol{k})\ \mathrm{d}\boldsymbol{k} \\
&\approx \sum_{\alpha,\beta} S(\langle \alpha\Delta k_x, \beta\Delta k_z\rangle)\Delta k_x \Delta k_z \\
&= \sum_{\boldsymbol{k}_{\alpha,\beta}} S(\boldsymbol{k}_{\alpha,\beta})\Delta k_x \Delta k_z
\end{aligned} \tag{2.23}$$

where α and β are omitted for brevity to formulate

$$\begin{aligned}
\frac{A_{\boldsymbol{k}}^2}{2} &= S(\boldsymbol{k})\Delta k_x \Delta k_z \\
&\Longrightarrow \\
A_{\boldsymbol{k}} &= \sqrt{2S(\boldsymbol{k})\Delta k_x \Delta k_z}
\end{aligned} \tag{2.24}$$

That is, we sum the patches of area between the grid points as our approximation. We limit ourselves to the domain $[-k_{\max}, k_{\max}]^2$ instead of infinities (see Equation 2.20), but we do not expect infinite wavevectors in a real-life ocean anyway.

As seen in Equation 2.23, the OWS may also be expressed in angular frequency space. This form is common in oceanographic literature, while wavevector space is more common in the field of computer graphics. As shown by Horvath [31], these two forms are related by

$$S(\boldsymbol{k}) = \frac{S(\omega) \cdot \frac{\delta\omega}{\delta k}}{k} \tag{2.25}$$

where the derivative may be evaluated using the dispersion relation (Equation 2.4):

$$\frac{\delta\omega}{\delta k} = \frac{g}{2\sqrt{gk}} \tag{2.26}$$

Finally, with all the wave component parameters at hand, we may express and compute a stochastic ocean based on $S(\boldsymbol{k})$ through an FTT algorithm as

$$\begin{aligned} h(\boldsymbol{x}, t) &= \mathcal{R}\left(\sum_{\boldsymbol{k}} \tilde{h}(\boldsymbol{k}, t) e^{i\boldsymbol{k}^\top \boldsymbol{x}}\right) \\ &= \mathcal{R}\left(N^2 \cdot \frac{1}{N^2} \sum_{\boldsymbol{k}} \tilde{h}(\boldsymbol{k}, t) e^{i\boldsymbol{k}^\top \boldsymbol{x}}\right) \\ &= \mathcal{R}\left(N^2 \cdot \mathcal{F}^{-1}\left(\tilde{h}(\boldsymbol{k}, t)\right)\right) \end{aligned} \tag{2.27}$$

where the Fourier components are defined as

$$\tilde{h}(\boldsymbol{k}, t) = \frac{1}{\sqrt{2}}(\xi_{\boldsymbol{k},r} + i\xi_{\boldsymbol{k},i}) A_{\boldsymbol{k}} e^{i\omega_{\boldsymbol{k}} t} \tag{2.28}$$

and where $\xi_{\boldsymbol{k},r}$ and $\xi_{\boldsymbol{k},i}$ are random variates from the standard normal distribution.

To derive Equation 2.28, first consider a Gaussian random variable $\mathcal{A}_{\boldsymbol{k}}$ where

- $\mathbb{E}[\mathcal{A}_{\boldsymbol{k}}] = 0$
- $\mathrm{Var}[\mathcal{A}_{\boldsymbol{k}}] = \mathbb{E}[\mathcal{A}_{\boldsymbol{k}}^2] = A_{\boldsymbol{k}}^2$

Then,

$$\mathcal{A}_{\boldsymbol{k}} e^{i(\omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})} = \frac{1}{\sqrt{2}}(\xi_{\boldsymbol{k},r} + i\xi_{\boldsymbol{k},i}) A_{\boldsymbol{k}} e^{i\omega_{\boldsymbol{k}} t} \tag{2.29}$$

holds as they both follow the same probability distribution, a circularly-symmetric complex central Gaussian distribution with expected value 0 and variance $A_{\boldsymbol{k}}^2$.

Taking a step back, one may question the necessity of animating the ocean in wavevector space, which (as shown by Equation 2.28) requires us to compute the rather expensive Fourier transform to obtain a representation of the ocean in the spatial domain every frame. It would certainly be simpler to just derive a set of sinusoids in the spatial domain once and then evaluate these at each time step. However, this approach would yield a time complexity of $\mathcal{O}(N^4)$ as each of the N^2 sinusoids would have to be evaluated for each of the N^2 points in the spatial domain. In contrast, computing the 2D FFT yields a time complexity of only $\mathcal{O}(N^2 \log N^2)$.

2.3 Extensions To Ocean Wave Spectrums

The FFT-based ocean model presented in Section 2.2 provides a solid foundation for simulating realistic-looking oceans in real-time. Unfortunately, due to several approximations made and the limitation of only being able to represent a finite set of wavevectors in hardware, there are a number of shortcomings that follow. However, a number of solutions have been proposed to mitigate these problems, and we will present some of them in this section. Notably, we will look into how the following concerns are commonly dealt with:

- The current model only moves the surface vertically, but the surface of a real ocean also moves horizontally. For instance, a rubber duck floating on top of an ocean surface would swing back and forth in rhythm with the waves.
- The current model produces somewhat round wave crests and troughs due to a finite number of sinusoids being used. This may be reasonable for fair weather conditions, but it does not work well for stormy weather where ocean waves typically form sharp peaks and flattened valleys [81].
- The current model does not take into account wind direction. The model mimics observations made from a real ocean using an OWS, but the instruments used in those observations only measure wave elevation, not wind direction.

2.3.1 Horizontal Displacements

The problems of horizontal movement and rounded wakes are closely connected and usually solved together through the use of *choppy waves* [81, 23, 85, 20]. Formerly, each 2D grid position $\boldsymbol{x} = \langle x, z \rangle$ was displaced at time t into a 3D position $\boldsymbol{X}$ as

$$\boldsymbol{X} = \langle x, h(\boldsymbol{x}, t), z \rangle \tag{2.30}$$

Now, the idea is to instead define $\boldsymbol{X}$ as

$$\boldsymbol{X} = \langle x + \boldsymbol{D}_x, h(\boldsymbol{x}, t), z + \boldsymbol{D}_z \rangle \tag{2.31}$$

where

$$\begin{aligned} \langle \boldsymbol{D}_x, \boldsymbol{D}_z \rangle &= \boldsymbol{D}(\boldsymbol{x}, t) \\ &= \sum_{\boldsymbol{k}} (-i\frac{\boldsymbol{k}}{k}) \tilde{h}(\boldsymbol{k}, t) e^{i\boldsymbol{k}^\top \boldsymbol{x}} \\ &= \sum_{\boldsymbol{k}} \left(-i\hat{\boldsymbol{k}} \cdot \tilde{h}(\boldsymbol{k}, t) \right) e^{i\boldsymbol{k}^\top \boldsymbol{x}} \end{aligned} \tag{2.32}$$

which has a rather complicated derivation that can be found in Chapter 7 of *Improved linear representation of ocean surface waves* [9]. Note the similarity to Equation 2.27 and that $\hat{\boldsymbol{k}}$ is the normalized direction of $\boldsymbol{k}$.

In practice, Equation 2.31 is often extended with a *choppiness factor* $\lambda \in \mathbb{R}$ that controls the intensity as

$$\boldsymbol{X} = \langle x + \lambda \boldsymbol{D}_x, h(\boldsymbol{x}, t), z + \lambda \boldsymbol{D}_z \rangle \tag{2.33}$$

and called *choppy waves*, a term coined by Tessendorf [81]. The λ factor is not physically motivated, but rather introduced in order to provide more artistic control.

As an aside, coming into the field of computer graphics, it might seem strange to revolve movement around a static grid whose vertices are individually displaced. One could argue for and against whether or not a shared origin for the vertices would have been more convenient, but the main benefit of this grid is actually in terms of performance. By using a static reference, the (often large amount of) vertex data used only needs to be uploaded to the GPU once, instead of every frame.

2.3.2 Directional Spreading Functions

As an OWS does not take into account wind direction by itself, it is common in simulation purposes [31, 23, 85] to extend the definition of $S(\boldsymbol{k})$ with a directional spreading function $D(\omega, \theta)$. Such a function attenuates the energy of waves based on their propagation direction in relation to the direction of the wind. If we recall Equation 2.25 from earlier, then the full set of conversions would now become [31]:

$$\begin{aligned} S(\boldsymbol{k}) &= S(\langle k\cos(\theta), k\sin(\theta)\rangle) \\ &= \frac{S(\omega,\theta)\cdot\dfrac{\delta\omega}{\delta k}}{k} \\ &= \frac{S(\omega)D(\omega,\theta)\cdot\dfrac{\delta\omega}{\delta k}}{k} \\ &= \frac{S(f)D(\omega,\theta)\cdot\dfrac{\delta\omega}{\delta k}\cdot\dfrac{\delta f}{\delta\omega}}{k} \end{aligned} \tag{2.34}$$

For all wave propagation directions (angles) θ, the value of $D(\omega, \theta)$ is typically maximized when θ aligns with the direction of the wind. In the literature, directional spreading functions are typically defined based on the assumption that the wind direction has angle 0. Consequently, $\arg\max_\theta D(\omega, \theta) = 0$ is typically the case.

In general, a directional spreading function has to satisfy

$$\forall\omega, \theta\,[D(\omega, \theta) \geq 0] \tag{2.35}$$

and

$$\forall\omega\left[\int_{-\pi}^{\pi} D(\omega, \theta)\ d\theta = 1\right] \tag{2.36}$$

because then

$$S(\omega) = \int_{-\pi}^{\pi} S(\omega, \theta)\ d\theta \tag{2.37}$$

is also satisfied, implying that the total energy amount is not disturbed [23].

2.4 Linearized Equations of Fluid Motion

The FFT-based ocean model presented so far has been non-interactive with respect to external objects such as boats. Simulating such behavior can be achieved through the Navier-Stokes equations, which is a set of partial differential equations (PDE) describing the motion of fluids. Unfortunately, these equations are too computationally demanding to be directly used in real-time simulation of oceans, even for our simplified ocean surface. Nevertheless, through a number of assumptions and approximations, the Navier-Stokes equations can actually be simplified to a system that can efficiently be computed in real-time for large-scale fluid simulations.

In this section, we give a brief introduction to the Navier-Stokes equations and how the aforementioned simplification of them can be derived. The linearized Bernoulli equations presented at the end of this section (Equation 2.50 and Equation 2.51) is the result of this process, and what can ultimately be used for interactive simulation of wakes formed by objects moving across the ocean surface. The equations are expressed in the frequency domain, making them particularly compatible with the FFT-based ocean model, as demonstrated later in Section 3.1.2.

For an incompressible fluid, the Navier-Stokes equations are

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \frac{\mu}{\rho}\nabla^2 \boldsymbol{u} = -\nabla \frac{p}{\rho} + \boldsymbol{F} \tag{2.38}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2.39}$$

where $\boldsymbol{u}(\boldsymbol{X}, t)$ is a 3D vector field of velocities, $p(\boldsymbol{X}, t)$ is a 3D scalar field of pressure levels, μ is the viscosity coefficient of the fluid (high for sticky fluids, $\sim$0 for water), ρ is the density of the fluid, and $\boldsymbol{F}$ is the sum of external forces (such as gravity). The ∇^2 operator is called the Laplace operator and is defined as $\nabla^2 f = \nabla \cdot \nabla f$.

The first equation comes from Newton's second law of motion; force equals mass times acceleration. In this case, there are two internal forces involved, one force due to the pressure gradient $-\nabla p$ (the fluid flows from higher pressure areas to lower pressure areas) and one due to the kinematic viscosity $\frac{\mu}{\rho}$ (each fluid particle's velocity is affected more by its neighbors when μ is high). The second equation (Equation 2.39) implies that mass is conserved through space. More specifically, at any given point $\boldsymbol{X}$, if the velocity $\boldsymbol{u}$ changes positively in one direction, then it must change negatively by the same magnitude in the opposite direction. In essence, the shape of the fluid body may change, but not the density (which is considered fixed).

By assuming that ocean water is both incompressible and completely inviscid (i.e. $\mu = 0$), we may simplify Equation 2.38 into

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\nabla \frac{p}{\rho} + \boldsymbol{F} \tag{2.40}$$

which, when paired with Equation 2.39, is commonly referred to as *Euler's equations* in the field of fluid dynamics.

Going further, we may also simplify Euler's equations, by restricting the fluid motion to a model type known as *potential flow*. Under this assumption, the fluid velocity $\boldsymbol{u}$ is defined as the gradient of a scalar field $w(\boldsymbol{X}, t)$, representing *velocity potential*. That is, $\boldsymbol{u} = \nabla w$. This is a quite significant assumption, as it assumes that the curl across the velocity field is $\mathbf{0}$ (irrotational flow), making the formation of vortices impossible. However, it is an important simplification from a computational perspective, as it reduces the spatial degrees of freedom of $\boldsymbol{u}$ from 3 to only 1 (w). Again, this might seem like an extreme simplification, but it has been empirically shown to produce good-looking results in practice [81, 85, 60].

Furthermore, as we mainly focus on gravitational wind waves in this report, the sum of external forces $\boldsymbol{F}$ is approximately dominated by gravity, the primary restoring force of such waves. Gravity is a conservative force, implying that $\boldsymbol{F}$ may also be defined as the gradient of some potential function, e.g. $\boldsymbol{F} = -\nabla U(\boldsymbol{X}, t)$. A natural choice for U would then be potential energy ($U = gh$), as chosen by Tessendorf [81].

With these additional constraints, Equation 2.40 may now be expressed as

$$\frac{\partial \nabla w}{\partial t} + (\nabla w \cdot \nabla)\nabla w = -\nabla \frac{p}{\rho} - \nabla U \tag{2.41}$$

which may be further simplified by applying the following derivation:

$$\begin{aligned}
\frac{\partial \nabla w}{\partial t} + (\nabla w \cdot \nabla)\nabla w &= -\nabla \frac{p}{\rho} - \nabla U \\
\frac{\partial \nabla w}{\partial t} + \frac{1}{2}\nabla(\nabla w \cdot \nabla w) - \nabla w \times (\nabla \times \nabla w) &= -\nabla \frac{p}{\rho} - \nabla U \\
\frac{\partial \nabla w}{\partial t} + \frac{1}{2}\nabla(\nabla w \cdot \nabla w) &= -\nabla \frac{p}{\rho} - \nabla U \\
\frac{\partial \nabla w}{\partial t} + \frac{1}{2}\nabla(\nabla w \cdot \nabla w) + \nabla \frac{p}{\rho} + \nabla U &= \mathbf{0} \\
\nabla(\frac{\partial w}{\partial t} + \frac{1}{2}(\nabla w \cdot \nabla w) + \frac{p}{\rho} + U) &= \mathbf{0} \\
\nabla(\frac{\partial w}{\partial t} + \frac{1}{2}||\boldsymbol{u}||^2 + \frac{p}{\rho} + U) &= \mathbf{0} \\
\frac{\partial w}{\partial t} + \frac{1}{2}||\boldsymbol{u}||^2 + \frac{p}{\rho} + U &= f(t) \\
\frac{\partial w}{\partial t} + \frac{1}{2}||\boldsymbol{u}||^2 + \frac{p}{\rho} + U &= \frac{\partial(\int f(t)\ \mathrm{d}t)}{\partial t} \\
\frac{\partial(w - \int f(t)\ \mathrm{d}t)}{\partial t} + \frac{1}{2}||\boldsymbol{u}||^2 + \frac{p}{\rho} + U &= 0 \\
\frac{\partial v}{\partial t} + \frac{1}{2}||\boldsymbol{u}||^2 + \frac{p}{\rho} + U &= 0 \\
\frac{\partial v}{\partial t} + \frac{1}{2}||\boldsymbol{u}||^2 &= -\frac{p}{\rho} - U
\end{aligned} \tag{2.42}$$

where $f(t)$ is unknown and v is defined as $v = w - \int f(t)dt$.

If we now substitute $\boldsymbol{u} = \nabla w = \nabla v$ in both Equation 2.39 and Equation 2.42, then our resulting pair of equations is:

$$\frac{\partial v}{\partial t} + \frac{1}{2}||\nabla v||^2 = -\frac{p}{\rho} - U \tag{2.43}$$

$$\nabla^2 v(\boldsymbol{X}, t) = 0 \tag{2.44}$$

where Equation 2.43 is called *Bernoulli's equation* for unsteady potential flow [81].

These two equations form the theoretical foundation on which the soundness of the simulation will rest, but there are yet two additional simplifications that will be made in order to obtain a computationally efficient system suitable for our ocean model. The first simplification is to linearize the equations, as suggested by Tessendorf [81], which simply amounts to dropping the quadratic term[13] in Equation 2.43, resulting in:

$$\frac{\partial v}{\partial t} = -\frac{p}{\rho} - U \tag{2.45}$$

The second simplification is to limit the fluid simulation to the surface of a heightfield $h(\boldsymbol{x}, t)$, as opposed to a 3D volume. Fortunately, this measure does not hurt generality when paired with the surface-limited ocean model presented in Section 2.2. Moreover, as all fluid particles along an ocean surface experience virtually the same atmospheric pressure (and by definition no hydrostatic pressure), the pressure function $p(\boldsymbol{X}, t)$ can now be treated as a constant, conveniently defined as $p = 0$.

With y representing our vertical axis, the equations 2.45 and 2.44 now become

$$\begin{aligned} \frac{\partial v}{\partial t} &= -U \\ &= -gh \end{aligned} \tag{2.46}$$

and

$$\left(\nabla_\perp^2 + \frac{\partial^2}{\partial y^2}\right) v = 0 \tag{2.47}$$

respectively, where the horizontal gradient $\nabla_\perp$ is defined as $\nabla_\perp = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial z^2}$.

By now we know how the velocity potential should change over time (Equation 2.46), but we still do not know how the heightfield should be affected because of this. Intuitively, the vertical velocity of the fluid at the surface is equivalent to the rate at which the elevation changes over time, i.e. $\frac{\partial h}{\partial t} = \frac{\partial v}{\partial y}$ (assuming free surface motion [83, 80]). By rearranging the terms of Equation 2.47, we get:

[13] This restriction will, in principle, limit the motion of extraordinarily intense fluid motions.

$$\begin{aligned}\left(\nabla_\perp^2 + \frac{\partial^2}{\partial y^2}\right) v &= 0 \\ \frac{\partial^2}{\partial y^2} v &= -\nabla_\perp^2 v \\ \frac{\partial}{\partial y} &= \pm\sqrt{-\nabla_\perp^2}\end{aligned} \tag{2.48}$$

of which any solution is valid, so we pick $\frac{\partial}{\partial y} = \sqrt{-\nabla_\perp^2}$. Consequently,

$$\begin{aligned}\frac{\partial h(\boldsymbol{x}, t)}{\partial t} &= \frac{\partial v(\boldsymbol{x}, t)}{\partial y} \\ &= \sqrt{-\nabla_\perp^2} v(\boldsymbol{x}, t)\end{aligned} \tag{2.49}$$

The linearized Bernoulli equations 2.46 and 2.49 outline the steps used to move the simulation forward, which, given initial values for height and velocity potential, can be used to perform (interactive) fluid simulation of wakes on top of an ocean surface.

In order to solve these equations in practice, one may use matrix exponentials to derive a computationally exact and efficient solution, such as the FFT forms (equations 19 and 20) derived and presented in eWave [84]. The derivation from the linearized Bernoulli equations to the FFT forms is purely numerical, without introducing any additional constraints or assumptions. Therefore, we simply present the resulting equations themselves here and refer readers to the original eWave paper [84] for further details:

$$\tilde{v}(\boldsymbol{k}, t + \Delta t) = \cos(\omega_{\boldsymbol{k}} \Delta t)\tilde{v}(\boldsymbol{k}, t) - \frac{g}{\omega_{\boldsymbol{k}}} \sin(\omega_{\boldsymbol{k}} \Delta t)\tilde{h}(\boldsymbol{k}, t) \tag{2.50}$$

$$\tilde{h}(\boldsymbol{k}, t + \Delta t) = \cos(\omega_{\boldsymbol{k}} \Delta t)\tilde{h}(\boldsymbol{k}, t) + \frac{k}{\omega_{\boldsymbol{k}}} \sin(\omega_{\boldsymbol{k}} \Delta t)\tilde{v}(\boldsymbol{k}, t) \tag{2.51}$$

The above equations are solved in wavevector space, just like the non-interactive ocean model presented in Section 2.2. Consequently, they may also be efficiently converted to and from the spatial domain using GPU-accelerated FFT algorithms, making the two simulations particularly easy to combine.

On a final note, we will use the linearized Bernoulli equations to motivate the dispersion relation that was taken for granted in Section 2.1. To begin with, equations 2.46 and 2.49 can be rewritten as a single equation, as shown below (see Section 3.2 in Tessendorf [81] for the derivation):

$$\frac{\partial^4 h(\boldsymbol{x}, t)}{\partial t^4} = g^2 \nabla_\perp^2 h(\boldsymbol{x}, t) \tag{2.52}$$

If we substitute $h(\boldsymbol{x}, t)$ in Equation 2.52 with the elevation for a single wave component, i.e. $h(\boldsymbol{x}, t) = A_{\boldsymbol{k}} e^{i(\boldsymbol{k}^\top \boldsymbol{x} + \omega_{\boldsymbol{k}} t + \phi_{\boldsymbol{k}})}$ (see Equation 2.15), then we get

$$A_{\boldsymbol{k}}(\omega_{\boldsymbol{k}}^4 - g^2 k^2) = 0 \tag{2.53}$$

which has two possible solutions. Either $A_{\boldsymbol{k}} = 0$, implying a wave component with 0 mean-energy, which can be disregarded as such a wave component would be pointless to simulate. The other possible solution is that $\omega_{\boldsymbol{k}}^4 = g^2 k^2$, which would yield the dispersion relation $\omega_{\boldsymbol{k}}^2 = gk$ (Equation 2.3) that we have assumed so far.

The linearized Bernoulli equations that we have presented in this section form a first-order PDE system. In general, the sum of specific solutions to such a system is itself also a valid solution [85]. Hence, not only is each wave component simulated in Section 2.2 a solution to the linearized Bernoulli equations, but so is their sum, i.e. the Fourier series. In conclusion, combining the FFT forms of the linearized Bernoulli equations (equations 2.50 and 2.51) and the FFT-based non-interactive ocean model (Equation 2.27) is not only straightforward, but also valid.

2.5 GPU Readbacks

In general, performing readbacks of data from the GPU to the CPU can be quite problematic. When performing synchronous readbacks, we run the risk of causing a graphics pipeline stall, which can be detrimental to performance [2, 56, 86]. If we instead opt for asynchronous readbacks, we avoid most of the former performance concerns but consequently introduce latency [91, 54, 3, 86, 55].

Shawn Hargreaves, Direct 3D development lead at Microsoft (as of writing), explains the details of a pipeline stall in one of their blog posts [27], and this description is further expanded upon in the official documentation of Direct 3D [54]. In summary, the CPU and the GPU should ideally run in parallel in order to achieve maximum performance. However, if they are at some point in time forced to run sequentially (i.e. one device works while the other waits), then that occurrence is referred to as a pipeline stall. In the case of a synchronous readback, this stall follows from the following situation:

1. The CPU provides batches of work (command buffers) to the GPU. The GPU begins to process this incoming work.

2. The CPU requests a synchronous readback in order to read results based on previously dispatched command buffers. The GPU can not immediately process this readback as the previous commands might affect the result.

3. The CPU now **waits** until the GPU reaches the readback. The GPU eventually completes the readback, allowing the CPU to resume.

4. The GPU now **waits** as it has no more work left to do until the CPU has filled up and dispatched new command buffers[14].

[14]The CPU could not record command buffers while waiting as the readback was synchronous.

When performing synchronous readbacks of a 256×256 texture (with 32 bits per pixel) in Unreal Engine, we found from our own measurements[15] that the overall cost imposed by such stalls was typically in the range of 5-15 milliseconds per frame, which is near unusable when considering a frame budget of $\sim$17 milliseconds.

In the case of an asynchronous readback, the CPU does not wait for the readback to finish like in step 3 above. Instead, the CPU will continue to work and occasionally poll the target staging resource[16] until the readback has finished [54]. In the case of the Unity game engine, this readback status is automatically polled once every frame [92]. In Direct 3D, this process is not automatic, but Microsoft recommends that the application should wait "At least two frames because this will enable parallelism between the CPU(s) and the GPU to be maximally leveraged." [53]. This 2-frames rule is motivated by the way that the CPU and GPU work asynchronously in most real-time applications. In particular, the following pattern usually occurs [53]:

- Frame 1: the CPU records command buffers for the current frame and then dispatches these to the GPU.
- Frame 2: the CPU records command buffers for the current frame and then dispatches these to the GPU. **Meanwhile**, the GPU processes the work received from frame 1.
- Frame 3: the CPU records command buffers for the current frame and then dispatches these to the GPU. Meanwhile, the GPU processes the work received from frame 2 **because** the work received from frame 1 is now finished.
- ...
- Frame N: the CPU records command buffers for the current frame and then dispatches these to the GPU. Meanwhile, the GPU processes the work received from frame $N-1$ because the work received from frame $N-2$ is now finished.

Naturally, there is no guarantee that a readback will have finished after 2 frames. Depending on recent workload and hardware, it could finish either earlier or later, sometimes in a fluctuating manner. However, it should be noted that the latency (when counted in frames) has an upper limit defined by the size of the swapchain[17]. But, if this limit is reached, then a pipeline stall will occur, so a swapchain size of at least 2 is preferred. The takeaway is that there will invariably be latency when performing asynchronous readbacks (unless a stall occurs), and the results can not be expected to be available in the same frame as the corresponding readback request.

[15] See Chapter 4 for a list of the GPUs used.

[16] A staging resource exists in a dedicated area of CPU memory, sometimes called AGP memory [53], that is accessible by the GPU. The *target* in this case simply refers to the memory destination to which the readback will copy the requested data.

[17] Essentially, how many frames the application allows the CPU to work ahead of the GPU.

On a final note, currently, most high-end consumer-grade GPUs are discrete as opposed to integrated GPUs, as suggested by Steam's hardware surveys [94]. The key difference between these two types is that discrete GPUs have dedicated RAM (called video RAM) that is physically separate from the CPU's (called system RAM). In the case of integrated GPUs, it is possible that by following a unified memory architecture [53], such as in the Apple M1 chip [73], one could potentially reduce the latency or at least the stall performance concerns related to GPU readbacks. This could make GPU readbacks a more viable option in future consumer-grade hardware setups, but the bottleneck of reliably passing data between CPU and GPU control (without introducing race conditions) would remain.

3

Methods

In this chapter, we present the design of our proposed theoretical framework that aims to improve the accuracy of FFT-based GPGPU ocean surface simulations by avoiding latency-induced errors. We also cover how said framework was implemented as a prototype in Unreal Engine, as well as how this prototype was used to produce the results presented in Chapter 4. We begin with explaining how a modest (but faithful[1]) version of the current state of the art in real-time ocean simulation was implemented, then proceed with presenting the modifications that our framework introduces, and then conclude with the methods used to compare the two solutions.

3.1 Implementing the State of the Art

The implemented prototype is comprised of three different simulations that interact with each other in unidirectional and bidirectional ways. These simulations are:

- a non-interactive ocean surface simulation that mimics the movement of a real-life ocean whose shape is mainly dominated by gravitational wind waves;
- an interactive fluid simulation that acts on top of the ocean surface and is responsible for propagating wakes induced by external objects;
- and the simulation of a set of user-controllable boats that float and drive along the surface of the ocean, generating wakes in the process.

In this section, we will describe the above simulations individually in the order listed, followed by a brief overview of the rendering process used.

3.1.1 Non-interactive Ocean Surface Simulation

The non-interactive part of the ocean simulation is heavily based on the theory presented in Section 2.2. In particular, the spatial grid defined by Equation 2.16 is used (with $L = 100$ meters and $N = 2^8 = 256$) as a static reference of vertices, whose positions are then dynamically displaced each frame using Equation 2.33 (with $\lambda = 1$) in order to animate the surface in a way that mimics the motion of a real-life ocean. The set of wavevectors that was used is defined by Equation 2.17.

[1] As in further model improvements should not affect the design of the proposed framework.

As the resulting surface, which we will refer to as an *ocean patch*, is periodic (i.e. seamlessly tileable), we use Equation 2.18 to repeat it over a finitely large area. This form of tiling is suitably accompanied by a Level of Detail (LOD) solution[2] in order to significantly reduce the overall triangle count to draw. Alternatively, a projected grid [30, 40] could be used to achieve a truly infinitely-spreading ocean at a fixed cost[3]. In this work, neither of these optimizations were prioritized enough to be implemented as neither of them were expected to affect the final comparison.

One deviation made from the displacement formula presented in Equation 2.33, is that we for the implementation follow the advice of Thomas Gamper [23] in changing the definition of $\tilde{h}(\boldsymbol{k}, t)$ in Equation 2.28 to

$$\tilde{h}(\boldsymbol{k}, t) = \frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t} + \overline{\frac{1}{2}\tilde{h}_0(-\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}} \tag{3.1}$$

where the time-independent[4] random variates $\tilde{h}_0(\boldsymbol{k}) \in \mathbb{C}$ are defined as

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_{\boldsymbol{k},r} + i\xi_{\boldsymbol{k},i})A_{\boldsymbol{k}} \tag{3.2}$$

This new definition of $\tilde{h}(\boldsymbol{k}, t)$, which we will denote $\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)$, is preferred over the definition given in Equation 2.33, which we will denote $\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)$, because it is a Hermitian function with respect to $\boldsymbol{k}$. That is,

$$\tilde{h}_{\mathrm{P}}(-\boldsymbol{k}, t) = \overline{\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)} \tag{3.3}$$

In general, the (inverse) Fourier transform of a Hermitian function is a real-valued function, and vice versa [23]. Consequently, $h(\boldsymbol{x}, t)$ may be computed as

$$\begin{aligned} h(\boldsymbol{x}, t) &= \mathcal{R}\left(N^2 \cdot \mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)\right)\right) \\ &= N^2 \cdot \mathcal{R}\left(\mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)\right)\right) \\ &= N^2 \cdot \mathcal{R}\left(\mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)\right)\right) \\ &= N^2 \cdot \mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)\right) \end{aligned} \tag{3.4}$$

The benefit gained from this modification is that $\mathcal{F}$ may now be implemented with an FFT algorithm adapted to real-valued signals. In such an FFT, one can omit all memory and computations related to the imaginary dimension (as the imaginary parts will cancel out to 0 anyway), effectively halving the overall computation and memory cost. This optimization can not be applied to Equation 2.27, because evaluating that equation would require the computation of a regular (complex) FFT of which half of the result is then ignored as we only extract the real part.

[2]For an overview, see: `https://en.wikipedia.org/wiki/Level_of_detail_(computer_graphics)`

[3]The simulation itself would still need to be finite though.

[4]Consequently, the $\tilde{h}_0(\boldsymbol{k})$ values only need to be computed once during the simulation.

For the FFT implementation itself, we decided to implement an adapted version of the GPGPU-based algorithm presented by Flügge [20]. There are well-tested libraries available for computing FFTs, such as FFTW [22] and cuFFT [58], but these are restricted to CPU-only and NVIDIA-cards-only, respectively. Regardless, a custom implementation was desired as it would ensure (1) that we could adapt the code to work with Unreal, (2) that we could completely avoid GPU readbacks, (3) transparency, and (4) flexibility in how the FFT would be called from other GPU-based algorithms.

The necessity and validity of Equation 3.4 follows from the following observations:

- $\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)$ is not Hermitian.
- $\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)$ is Hermitian.
- $\mathcal{R}\left(\mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)\right)\right) = \mathcal{R}\left(\mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)\right)\right)$

Let us tackle these three prerequisites in the order listed. First off, $\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)$ is not Hermitian because $\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t) = \frac{1}{\sqrt{2}}(\xi_{\boldsymbol{k},r} + i\xi_{\boldsymbol{k},i})A_{\boldsymbol{k}}e^{i\omega_{\boldsymbol{k}}t}$, where the random variates $\xi_{\boldsymbol{k},r}$ and $\xi_{\boldsymbol{k},i}$ are unique to each wavevector $\boldsymbol{k}$. In general, $\xi_{\boldsymbol{k},r} \neq \xi_{-\boldsymbol{k},r}$ and $\xi_{\boldsymbol{k},i} \neq \xi_{-\boldsymbol{k},i}$. Thus, in general

$$\tilde{h}_{\mathrm{T}}(-\boldsymbol{k}, t) \neq \overline{\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)} \tag{3.5}$$

Next, we show the correctness of Equation 3.3 as follows:

$$\begin{aligned}
\tilde{h}_{\mathrm{P}}(-\boldsymbol{k}, t) &= \frac{1}{2}\tilde{h}_0(-\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t} + \overline{\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}} \\
&= \overline{\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}} + \frac{1}{2}\tilde{h}_0(-\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t} \\
&= \overline{\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t} + \overline{\frac{1}{2}\tilde{h}_0(-\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}}} \\
&= \overline{\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)}
\end{aligned} \tag{3.6}$$

Finally, for the last concern, we show:

$$\begin{aligned}
\mathcal{R}\left(\mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{T}}(\boldsymbol{k}, t)\right)\right) &= \mathcal{R}\left(\mathcal{F}^{-1}\left(\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}\right)\right) \\
&= \mathcal{R}\left(\mathcal{F}^{-1}\left(\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t} + \frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}\right)\right) \\
&= \mathcal{R}\left(\mathcal{F}^{-1}\left(\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}\right) + \mathcal{F}^{-1}\left(\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}\right)\right) \\
&= \mathcal{R}\left(\mathcal{F}^{-1}\left(\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}\right) + \overline{\mathcal{F}^{-1}\left(\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}\right)}\right) \\
&= \mathcal{R}\left(\mathcal{F}^{-1}\left(\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}\right) + \mathcal{F}^{-1}\left(\overline{\frac{1}{2}\tilde{h}_0(-\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}}\right)\right) \\
&= \mathcal{R}\left(\mathcal{F}^{-1}\left(\frac{1}{2}\tilde{h}_0(\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t} + \overline{\frac{1}{2}\tilde{h}_0(-\boldsymbol{k})e^{i\omega_{\boldsymbol{k}}t}}\right)\right) \\
&= \mathcal{R}\left(\mathcal{F}^{-1}\left(\tilde{h}_{\mathrm{P}}(\boldsymbol{k}, t)\right)\right)
\end{aligned} \tag{3.7}$$

Recall from Section 2.2.2 that an OWS is required to configure the parameters of all the wave components. In this work, we did not have any strict requirements in the choice of an OWS, other than that the resulting ocean should "look convincing". Thus, we simply implemented and tried a small selection of OWS functions, from which we then picked the spectrum that was deemed the most adequate. For a thorough discussion on different OWS functions, we refer readers to the papers by Horvath et al. [31, 23].

The following lists the non-directional OWS functions $S(\omega)$ that were implemented:

- the Phillips spectrum [66]
- the JONSWAP spectrum [28]

The following lists the directional spreading functions $D(\omega, \theta)$ that were implemented:

- the flat spreading function [31]
- the Donelan-Banner spreading function [31]

From this selection, we ended up choosing the Phillips spectrum in combination with the Donelan-Banner spreading function as the defaults for our prototype, simply based on our subjective visual preference.

The implemented directional spreading functions above are all defined based on the assumption that the wind direction has angle 0. Thus, to support a user-configurable wind direction, we choose to evaluate the computation of any $D(\omega, \theta)$ value instead as $D(\omega, \theta - \theta_w)$, where θ is the propagation angle of wave and θ_w is the angle of the wind. Consequently, we evaluate $D(\omega, 0)$ whenever θ and θ_w align, as expected.

To realize the described simulation in code, we heavily rely on the use of textures and compute shaders. On startup, we initialize:

- a 4-channel $\log(N) \times N$ *butterfly texture* needed for the FFT algorithm [20];
- a 2-channel $N \times N$ texture containing the $\tilde{h}_0(\boldsymbol{k})$ values;
- and a 2-channel $N \times N$ texture containing the $\overline{\tilde{h}_0(-\boldsymbol{k})}$ values.

Then, every frame, we compute:

- $\tilde{h}(\boldsymbol{k}, t)$ (using Equation 3.1);
- and $-i\hat{\boldsymbol{k}} \cdot \tilde{h}(\boldsymbol{k}, t)$ (see Equation 2.32).

which are stored in two separate 2-channel textures. The FFT algorithm is then run in parallel on these two textures, resulting in the displacement $\boldsymbol{X}$ from Equation 2.33. For an overview, see Figure 3.1.

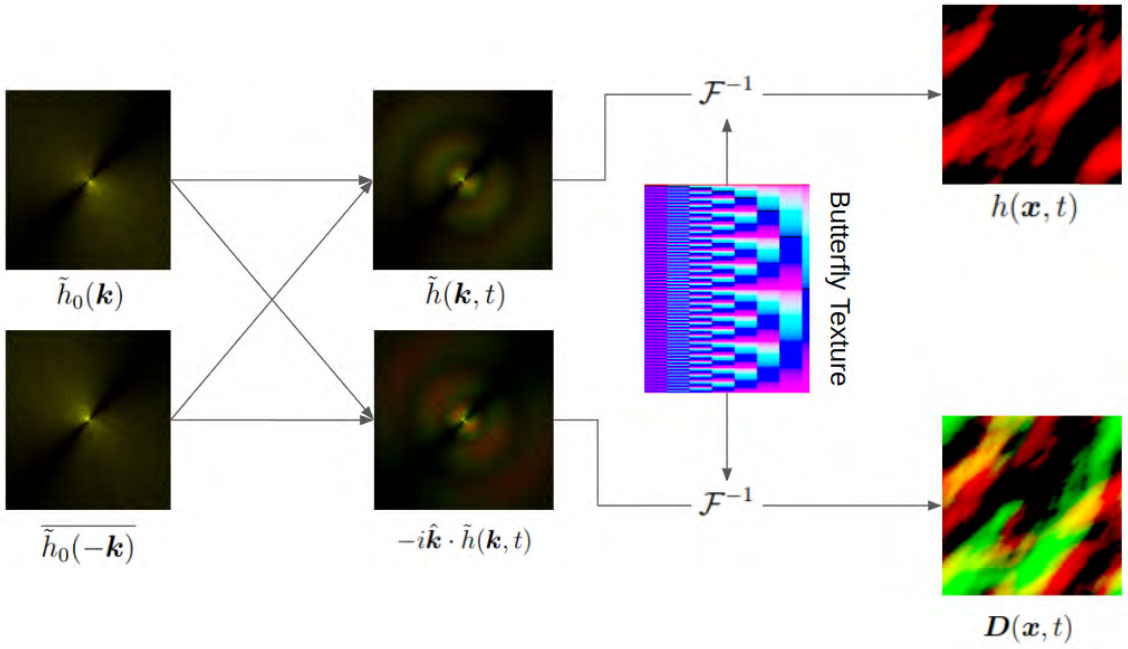


Figure 3.1: Schematic of the data flow in the non-interactive ocean surface simulation. The four textures that depend on time t are computed every frame.

Throughout our implementation, we use the FFT optimization described in chapter 2.4 of *Handbook of Real-Time Fast Fourier Transforms: Algorithms to Product Testing* [72] to efficiently compute the Fourier transform of pairs of signals using a single texture, such as with $-i\hat{\boldsymbol{k}}\cdot\tilde{h}(\boldsymbol{k},t)$ in Figure 3.1. This technique also requires the Hermitian property, which further signifies the importance of Equation 3.1.

On a final note, the FFT algorithm presented by Flügge [20], that we adapted in this work, is based on the 2D inverse Fourier transform. For the non-interactive part of the ocean simulation presented in this section, the inverse transform is sufficient. However, in the next section we will come to need the forward Fourier transform as well. To achieve this, we employed the data-swapping technique presented by Rick Lyons in one of their blog posts (see method #3) [50]. This technique is actually presented as a method to compute the inverse Fourier transform using the (forward) Fourier transform, but since the operation is symmetric, it can also be used for the reverse problem that we face.

3.1.2 Wake Simulation

In this section, we describe how the ocean simulation presented in Section 3.1.1 was made interactive. This interactive part of the ocean simulation is heavily based on the theory presented in Section 2.4. However, this theory only supplies us with a means to propagate existing wakes, which by itself does not allow for interactivity. Specifically, we will in this section focus on how objects that float on or move across the ocean surface can be made to both generate new and obstruct existing wakes.

The algorithm we use for this simulation is called eWave, and was presented by Tessendorf in 2014 [84]. eWave by itself is actually an extension of an algorithm called iWave [82], which was also presented by Tessendorf. The main contribution of this extension is the derivation of a set of exact[5] solutions to the wave propagation problem described in the original iWave paper. In the context of this report, this wave propagation problem is described by equations 2.46 and 2.49, while the chosen eWave solution is described by equations 2.50 and 2.51. Nevertheless, for the sake of discussion, we will in this report refer to the ideas jointly described by these two papers as simply "eWave".

eWave utilizes a surprisingly simple, yet powerful, model to generate sources of local disturbances and obstructions on the ocean surface. A simple 2D masking operation is all that is required to let objects of arbitrary shape both act as obstructions for incoming waves and to add disturbances for generating new waves. This is a straightforward and fast operation that (empirically-tested) produces good-looking results [60, 85, 81], but it is not physically accurate [81, 82]. Most notably, by representing all ocean-surface intersections as a binary mask, one effectively ignores the variation in depth by which external objects intersect the surface – intuitively, a deeper intersection should result in a greater volume of water to displace.

A promising alternative to eWave is presented in the recently published paper *Water Surface Wavelets* [38]. From a technical standpoint, the wavelet-based water propagation model described in this paper appears to be preferable to eWave, because it allows the visual fidelity of the simulation to scale faster than the computational complexity of the simulation itself, at least when compared to eWave. More specifically, it is able to represent a greater range of wavelengths (in particular small wavelengths, such as ripples) at the same computational cost. The algorithm's implementation is noticeably more complicated though, as it is essentially a hybrid of different simulations. As the two solutions should be interchangeable for our purposes, we decided to choose eWave due to its relative simplicity. Our decision was also motivated by the fact that eWave has recently been adopted with good results in both the industry [60] and in academic prototypes [85].

Unlike the non-interactive ocean simulation, the simulation of interactive wakes through eWave is not periodic[6]. Consequently, the interactive wake simulation can only cover a finite subregion of the possibly infinite ocean, implying that only a finite portion of the ocean surface will actually be interactive. Fortunately, there are ways to hide this limitation, as we will discuss in sections 3.1.2.3 and 3.1.2.5.

[5] iWave originally had to resort to artificial damping in order to stabilize the simulation [82].

[6] Which is reasonable, since a boat should only affect the part of the ocean surface that is locally next to it, and not deform the ocean in a repeating pattern across the entire spatial domain.

Similarly to the non-interactive ocean simulation, the entire simulation that we describe in this section is performed on the GPU using compute shaders. The required simulation data is *conceptually* maintained in the following set of $N \times N$ 1-channel textures (where t is the time of the current frame):

- H, containing the $h(\boldsymbol{x}, t) = \mathcal{F}^{-1}(\tilde{h}(\boldsymbol{k}, t))$ values (see Equation 2.50);
- V, containing the $v(\boldsymbol{x}, t) = \mathcal{F}^{-1}(\tilde{v}(\boldsymbol{k}, t))$ values (see Equation 2.51);
- H_{prev}, containing the $\tilde{h}(\boldsymbol{x}, t - \Delta t)$ values;
- V_{prev}, containing the $\tilde{v}(\boldsymbol{x}, t - \Delta t)$ values;
- and B, which masks any objects that intersect the surface.

However, in practice H and V, as well as H_{prev} and V_{prev}, are actually paired into the same 2-channel texture in order to alleviate the same FFT optimization as used in Section 3.1. Nevertheless, for ease of discussion, we will refer to these values as separate textures, whose purposes will be made clear in the upcoming subsections.

3.1.2.1 Obstructions and Disturbances

To implement the 2D masking operation that is used by eWave [84] in a way that integrates well with the FFT forms of the linearized Bernoulli equations (see Equation 2.50 and Equation 2.51) [85], we represent the set of obstructions (and disruptions) caused by each boat as a texture B. The pixel values of this texture are binary, more specifically either 0 or 1. A value of 0 indicates the presence of an object that intersects the surface, while a value of 1 indicates that nothing obstructs the surface at the corresponding location. We generate this texture, or obstruction map, every frame by projecting the submerged parts of our boats (see Section 3.1.3) onto the XZ-plane. The result is a black-white texture, like the ones shown in Figure 3.2.

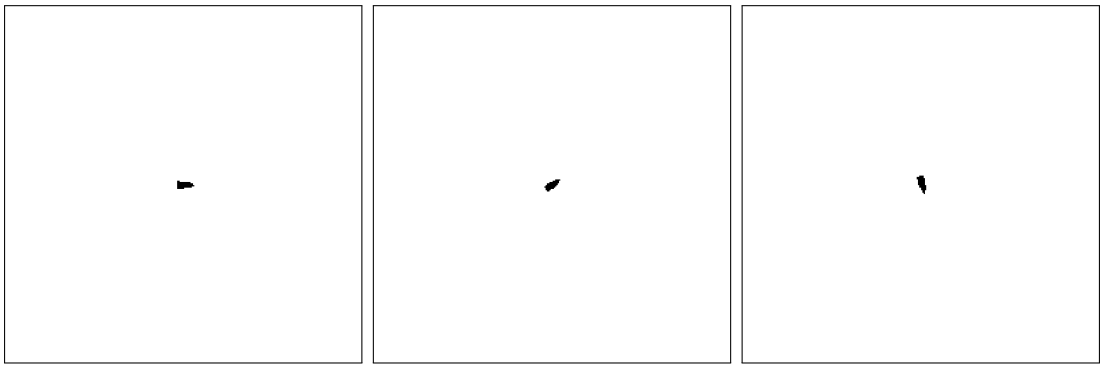


Figure 3.2: The obstruction map B for three different boat states. Regardless of the boat's position and velocity, the boat will always be centered in its corresponding obstruction map. However, any rotation will cause the boat's projection to deform.

In order to stop waves from unrealistically propagating inside of solid objects, the wake heightfield H is simply pixel-wise multiplied by the obstruction map B. Through this operation, the wake heights are multiplied by 0 everywhere an object

intersects the ocean surface. Conveniently, a side-effect of this simple operation is that it forms the reflection and refraction waves that result from the incoming waves hitting the submerged object. In the related papers [81, 82, 84, 85], Tessendorf does not appear to elaborate on the reasoning or intuition behind this side-effect, but our hypothesis is that it is a consequence of the mass conservation equation (see Equation 2.39) inherited from the Navier-Stokes equations. Essentially, the "removed" water mass has to go somewhere, and that "somewhere" is not inside the obstruction. Figure 3.3 shows a few examples of these reflections and refractions.

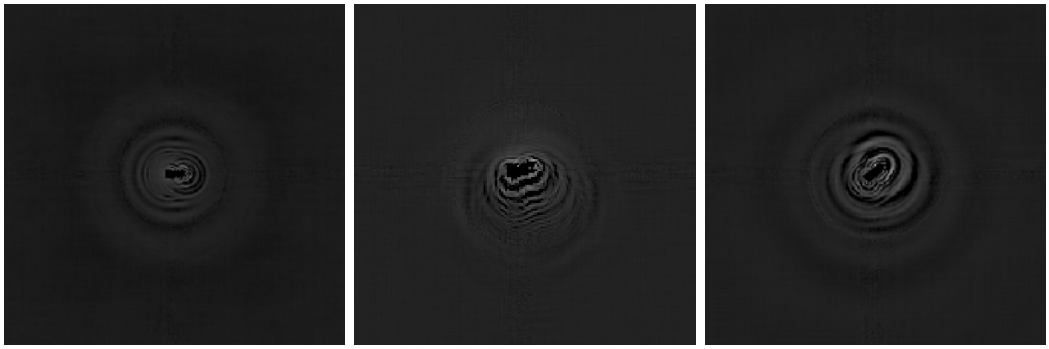


Figure 3.3: Three examples of wave reflections and refractions formed around the hull of a boat. From left to right: the boat is stationary, the boat drifts slowly upwards, and the boat bounces vertically.

A similar masking operation is used to create sources of disturbance that generate new waves. By adding the complement of the obstruction map B (scaled by an artistically-chosen factor $s \in \mathbb{R}$) to the heightfield H, we can selectively raise or lower the surface at specific locations. These elevation-modifications, or disturbances, then give rise to ripple and wake effects once the wave propagation step has been resolved.

For the wakes formed by our speedboats, we use a positive scaling factor s proportional to the velocity of the boat, such that greater wakes are formed by a greater velocity. Intuitively, a boat pushes water downward and sideways, so a negative scaling factor s, might seem appropriate. However, since disturbances and obstructions are being applied in an alternating fashion (see Section 3.1.2.4), it is in practice only the outline of the submerged geometries that will effectively disturb the surface. And if the boat's center pushes water down, then that water must go up elsewhere. In our case, that "elsewhere" is around the outline (i.e. hull) of the boat.

Despite being binary[7], the obstruction and disturbance masking operations are together enough to create the V-shaped Kelvin wake pattern observed from real-life speedboats (recall Figure 1.1) as well as the wakes running along the side of the boat that originate from the front (i.e. bow) of the boat. An example of these wake patterns being formed by the described algorithm can be found in Figure 3.4.

[7]Notably, the depth by which an object is submerged is being unrealistically disregarded.

Figure 3.4: Wake patterns formed by a boat driving straight northwest. The leftmost image shows the height potentials in H while the rightmost image shows the velocity potentials in V. As with B, each boat remains centered in its textures.

3.1.2.2 Wave Propagation

The eWave simulation described by equations 2.50 and 2.51 solves the linearized Bernoulli equations for the height and velocity potential of the waves. The height is what we are mainly interested in to animate the surface, but both of these potentials need to be maintained as they are required in every simulation update. In our case, we store them in the two textures H and V.

The mentioned equations are solved in the frequency domain, while the propagated wakes are realized in the spatial domain. Thus, to take the current state of the spatial domain as input to these equations, we first perform a forward FFT to both H and V, giving us the $\tilde{h}(\boldsymbol{k}, t)$ and $\tilde{v}(\boldsymbol{k}, t)$ values. From these, we obtain the next frame's values ($\tilde{h}(\boldsymbol{k}, t+\Delta t)$ and $\tilde{v}(\boldsymbol{k}, t+\Delta t)$) through equations 2.50 and 2.51, which then are realized back in to the spatial domain (H and V) using an inverse FFT.

3.1.2.3 Boundary Conditions

As the eWave simulation is inherently finite, it is possible for waves to propagate towards the border of the simulation, where they would then suddenly disappear. This behavior would look quite jarring when combined with a substantially larger non-interactive ocean simulation, because it would result in some prominent waves suddenly disappearing in the open sea. One way to mitigate this problem is to apply a smooth transition along the border of the simulation in the form of a gradual decay. We apply this technique through the linear 2D trim function $T(\boldsymbol{x})$, presented in Gilligan [85], as follows:

$$\begin{aligned} h(\mathbf{x}, t) &\leftarrow T(\mathbf{x}) \cdot h(\mathbf{x}, t) \\ v(\mathbf{x}, t) &\leftarrow T(\mathbf{x}) \cdot v(\mathbf{x}, t) \end{aligned} \tag{3.8}$$

where $T(\boldsymbol{x}) = T(\langle x, z \rangle) = T(x)T(z)$ and the 1D trim function $T(x)$ is defined as:

$$T(x) = \begin{cases} x/D & \text{if } x/D < 1 \\ |x - W|/D & \text{if } |x - W|/D < 1 \\ 1 & \text{otherwise} \end{cases} \tag{3.9}$$

where W is the width (and height) of the simulation texture (in our case $W = N$) and D is the thickness (in pixels) of the taper region. This gives us a linear, smooth decay of the wave heights and velocities at the borders of our simulation texture.

For clarity, we visualize Equation 3.9 using our parameters in Figure 3.5.

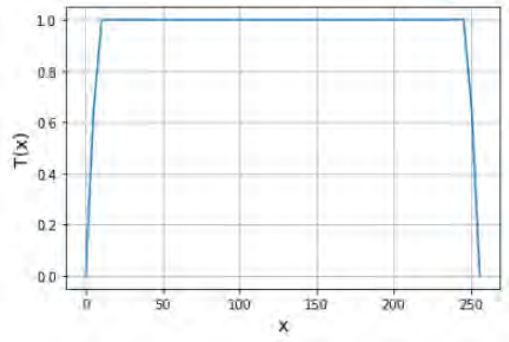


Figure 3.5: The tapering function used to fade out wakes on the border of the wake simulation. In our case, we use $D = 8$ (arbitrary choice) and $W = N = 256$.

3.1.2.4 Update Algorithm

The update algorithm performed in each simulation step of our eWave implementation is largely the same as the one presented in Gilligan [85]. However, due to time constrains, we chose to omit the Semi-Lagrangian advection scheme that allows for a drift current to affect the simulated area. We have also added a step that moves the simulation itself horizontally across the spatial domain, as described in Section 3.1.2.5. Each full simulation step can thus be summarized as follows:

1. Move the simulation (see Section 3.1.2.5)
2. Update obstruction maps: $B \leftarrow Project$(submerged geometry)
3. Apply obstructions: $H \leftarrow H * B$
4. Disturbance sources: $H \leftarrow H + s \cdot (1 - B)$
5. Boundary conditions: $H \leftarrow T * H$, $V \leftarrow T * V$
6. Compute FFT(H) and FFT(V)
7. Wave propagation (equations 2.50 and 2.51)
8. Compute iFFT(H) and iFFT(V)
9. Apply obstructions: $H \leftarrow H * B$
10. Boundary conditions: $H \leftarrow T * H$, $V \leftarrow T * V$
11. $H_{\text{prev}} \leftarrow H$, $V_{\text{prev}} \leftarrow V$

where $+$ and $*$ denote pixel-wise addition and multiplication, respectively.

From the above description of the update algorithm, it might seem like the textures H_{prev} and V_{prev} go unused, but they are actually used beyond the eWave simulation in order to move the simulation area along with the boat.

3.1.2.5 Moving the Simulation

The eWave simulation described thus far only covers a fixed finite area (of size 100 m^2, in our case). However, we want the speedboats to be able to move around freely across the ocean. In order to overcome this limitation, we dedicate a simulation area to each boat and then continuously move each such area along with its corresponding boat. With this approach, we need 1 simulation area per boat, regardless of the positions of the boats. Another option would be to dedicate a static simulation area per ocean patch, and then dynamically activate and inactivate these areas depending on the positions of the boats. In the worst-case scenario, this requires 4 simulation areas per boat (when each boat is next to the corner of a unique intersection of ocean patches), and in the best-case scenario, only 1 simulation area in total (because boats can share simulation area). Visually, the two options should yield equivalent results. Thus, the preferable option will depend on how many external objects there will be and how tightly they will be packed.

To artificially move the simulation area each frame, we copy each $h(\boldsymbol{x}, t - \Delta t)$ value of H_{prev} over to $h(\boldsymbol{x} + \boldsymbol{\varepsilon}, t)$ of H (and similarly for V_{prev} and V), where $\boldsymbol{\varepsilon} \in \mathbb{Z}^2$ is a 2D offset computed based on the movement of the boat[8]. Pairs of textures (e.g. H_{prev} and H) are used to avoid race conditions between threads when these copy operations are performed. Moreover, there are also two edge cases to address here:

- If $\boldsymbol{x}+\boldsymbol{\varepsilon}$ is an out-of-bounds pixel coordinate, then the copied value is discarded.
- If a pixel in the target texture is never copied to, its value is initialized to 0 (for both H and V).

As a result, when a boat travels in a particular direction, it will be met by freshly initialized height and velocity potentials, while the wakes it leaves behind will be slowly phased out from the simulation area. Inevitably, there will be discontinuities along the simulation border, but these are mitigated to a visually acceptable extent thanks to the tapering function (see Equation 3.9). We have found that this solution works surprisingly well for our purposes, even though speedboats can traverse quite long distances across the ocean surface in relatively short time frames.

3.1.3 Boat Physics Simulation

The boat simulation implemented in this work is largely based on a pair of articles written by Jacques Kerner [44, 45], who at the time had worked on implementing boat physics for the AAA game Just Cause 3. Unlike a proper scientific vehicle simulation, the ideas proposed by Jacques Kerner aim to capture the most important

[8]For the details of this offset computation, we refer readers to the compute shader `Obstruction.usf` found in the prototype's source code.

forces that act on boats at a reasonable computation cost per frame, which, as we have established by now, is a necessary trade-off imposed by real-time applications.

The first step of any simulation strategy is to first decide how to represent the object being simulated. In 3D computer graphics, some common strategies include:

- representing the object as a point cloud, where a ray is cast from each point to check for collisions;
- representing the object as the volume formed by a collection of 3D primitives, e.g. spheres, cubes, or columns;
- representing the object as a closed surface, typically composed of triangles.

The ray casting approach works well for accurate collision detection, but it inherently does not define a volume, which is problematic when simulating buoyancy. The volume-composition approach can give a compact representation (especially when using spheres) that allows for continuous intersection precision for collisions. However, the individual primitives used may overlap or leave gaps when packed together, consequently misrepresenting the true volume of the object [44]. Moreover, unless the primitives closely resemble the shape of the simulated object, one will end up with incorrect normals, which is problematic when pushing the object backward due to a collision. With this in mind, we chose the surface-based approach. For further discussion on the volumetric and surface-based approaches, we refer readers to the first of the two articles [44].

For rendering, we have a high-detail model of a boat that was provided to us by Rapid Images (see Figure 3.6). As this model forms a closed surface, it could theoretically be used as-is for the boat simulation as well, but simulating the roughly 2 million triangles it is made of would not be particularly cheap. With this in mind, we imported the model into Blender [5] and from there created a low-poly model (see Figure 3.7) that would represent the boat as perceived by the simulation.

Figure 3.6: The speedboat model used in our prototype as the visual representation of the boats. This 3D model was provided to us by Rapid Images.

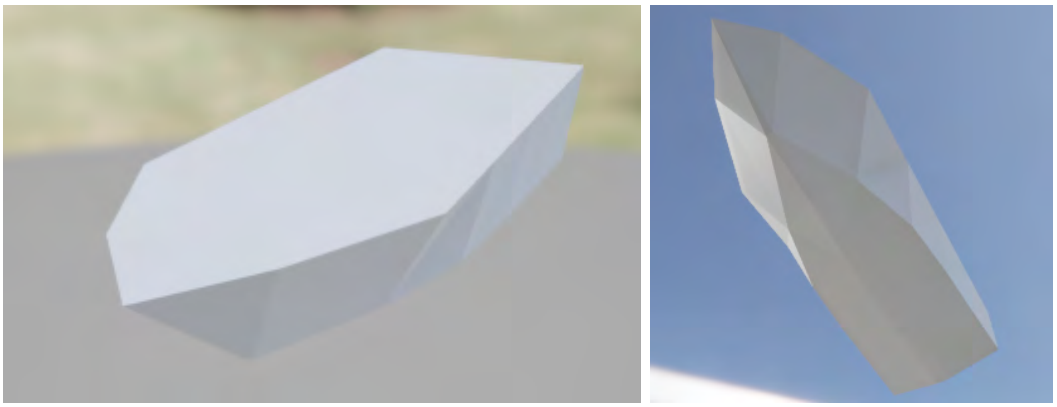


Figure 3.7: The low-poly (70 triangles) boat model used for the boat simulations.

In the simulation, each boat is treated as a rigid body[9]. Conveniently, Unreal Engine has built-in functionality for simulating rigid bodies. However, this functionality is limited to the CPU, which makes it unsuitable for our purposes. We have therefore chosen to instead implement our own rigid body (and related quaternion) functionality. For now, we will consider the commonalities between the CPU-based and GPU-based boat simulations, while Section 3.2.1 will focus on the GPU implementations along with the proposed framework.

Although the laws of rigid bodies and mechanical physics are beyond the scope of this report, we will still give an overview of our rigid body simulation and how it connects back to the triangles specified by our low-poly boat model. To begin with, each rigid body has:

- a 3D position in space;
- a linear velocity (affecting position);
- mass (resistance against linear momentum);
- an orientation (represented as a quaternion);
- an angular velocity (affecting orientation);
- and moment of inertia (resistance against angular momentum);

The position and orientation can be initialized arbitrarily, while the velocities will be assumed to always equal $\mathbf{0}$ initially. Meanwhile, mass and moment of inertia are fixed properties[10], which in our case are defined by the weight and shape of the simulated boat. To distribute the mass M across the boat, we simply compute the total surface area A_T of our model's triangles, and then assign triangle i, whose surface area is $A_{T,i}$, a mass of

$$M_i = M \cdot \frac{A_{T,i}}{A_T} \tag{3.10}$$

[9]Essentially, a 3D object restricted to translation and rotation (e.g. no shearing nor scaling).
[10]Unless phenomenons such as deformations and fuel consumption are considered.

Naturally, assigning mass to infinitely thin triangles is merely an approximation, and not physically accurate. Moreover, it does not easily take into account the uneven weight distribution of a boat (e.g. due to the heavy engine), which might disrupt steering. However, it does preserve both mass and volume. More importantly, it allows us to apply forces on the rigid body with respect to the centroid and normal of every triangle. This makes it possible for external forces (such as waves and gravity) to push and pull different parts of the boat in separate directions and by varying amounts, resulting in quite fine-grained torque computations.

At every frame, a given rigid body will experience a set of external forces (e.g. gravity and buoyancy), which will amount to a total force and torque. By Newton's second law of motion, the rigid body will then accelerate in the given frame. This acceleration will then contribute to the linear and angular velocities for the (fixed) time difference Δt that each discrete frame represents. In turn, the velocities affect the position and orientation of the boat, which is what ultimately affects the visuals.

Next, we will cover the external forces that act on each of the boats.

3.1.3.1 External Forces

First off, we have gravity, which is particularly simple to model, as it acts on the center of mass. Consequently, we apply it once, as opposed to once for each triangle. We calculate this force as:

$$\boldsymbol{F}_g = -gM \tag{3.11}$$

Next, we will consider buoyancy. As different parts of the boat might be submerged under the ocean surface by varying amounts, especially when considering an animated ocean, we calculate a buoyancy force per triangle instead of once per boat.

A prerequisite for computing these forces is a measurement of how deeply submerged (if at all) the various parts of the boat are. For this, we employ the (triangle) cutting algorithm proposed by Kerner [44], adapted to our FFT-based ocean model. The details of this algorithm are quite lengthy, but the basic idea is that we sample the elevation of the ocean surface at a few discrete points per triangle, and then subdivide triangles until we obtain a list of fully submerged triangles[11] that together approximate the submerged part(s) of the boat.

Once we have a list of fully submerged triangles, we can apply buoyancy to them. Based on Archimedes' principle, we compute the following buoyancy force for each fully submerged triangle i:

$$\boldsymbol{F}_{b_i} = -(g \cdot \rho \cdot h_{T,i} \cdot S_{T,i})\hat{\boldsymbol{n}}_{T,i} \tag{3.12}$$

where ρ denotes the density of water, $h_{T,i}$ denotes the depth beneath the ocean surface at which the fully submerged triangle's centroid is situated, $S_{T,i}$ denotes the surface area of said triangle, and $\hat{\boldsymbol{n}}_{T,i}$ denotes the normal[12] of said triangle.

[11] As discussed by Kerner [44], these triangles are not guaranteed to be fully submerged, but the cutting algorithm's approximation is quite accurate, especially when more triangles are used.

[12] The normal that points outward the boat, typically from the hull into the ocean.

In practice, only the vertical component of $\boldsymbol{F}_{b_i}$ is of interest, as the horizontal components should cancel out for a closed surface, such as our boat. The inclusion of $S_{T,i}$ in Equation 3.12 deviates from the force suggested by Kerner [44], however, we believe that this modification is necessary as dimensional analysis of the right-hand side of the equation would not yield a force otherwise.

So far, we have gravity and buoyancy, two forces that act against each other. If we simulate the boat as-is, then it will rest at the ocean surface, which is desired, but it will also bounce up and down in perfect oscillation as there is no energy loss. To resolve this issue, we need to introduce a set of damping forces. In a scientific physics simulation, these forces can quickly become quite complicated and expensive to model. However, in the real-time industry, it is quite common to instead employ a set of artificial forces that empirically give reasonable results [59, 18, 89].

In this work, we use the following linear and quadratic damping forces [45]:

$$\boldsymbol{F}_{d_1} = -(c_d \cdot \frac{S_T}{A_T})\boldsymbol{v} \tag{3.13}$$

$$\boldsymbol{F}_{d_2} = -(c_d \cdot \frac{S_T}{A_T} \cdot ||\boldsymbol{v}||)\boldsymbol{v} \tag{3.14}$$

where $c_d = 500$ is a fixed constant, $S_T = \sum_i S_{T,i}$, and $\boldsymbol{v}$ is the current linear velocity of the rigid body. To damp angular velocity we apply the idea described in NVIDIA's PhysX documentation [59], which seemingly is also used in Unreal Engine [18] and Unity [89], but scale the result with respect to $\frac{S_T}{A_T}$.

Furthermore, with the analogy of a human belly flop, if a boat was to be dropped onto the ocean, then we would expect the high density of the water (as compared to air) to result in a dramatic damping of the boat's vertical velocity upon impact. Kerner proposes a set of forces to achieve this sense of rigidness [45], however, we settled with a single, simpler force as we found it satisfactory for our purposes. This force, $\boldsymbol{F}_v$, is defined as follows:

$$\boldsymbol{F}_v = \boldsymbol{F}_{\text{stop}} \cdot ||\Delta(\frac{S_T}{A_T})|| \tag{3.15}$$

where $\boldsymbol{F}_{\text{stop}}$ is the boat's negative vertical momentum (i.e. the force that by itself would completely halt the boat's vertical velocity) and $\Delta(\frac{S_T}{A_T})$ is the difference in $\frac{S_T}{A_T}$ compared to the previous frame.

Finally, to let users control the boat, we apply two imaginary forces. The first force originates from the center-back (a.k.a. stern) of the boat (roughly where the engine's propeller would go), and pushes the boat forward[13]. The second force originates from the sides (i.e. below the gunwales), and pushes the boat horizontally to allow for basic steering.

[13]It will also push the boat's front (a.k.a. bow) a bit upward due to torque, as the engine is placed a bit below the boat's center of gravity.

3.1.4 Rendering

So far, we have presented the ocean, wakes, and boats; the three simulations involved in the implemented prototype. Now we will briefly cover how this simulation data, in combination with some additional rendering-specific data, is used to interactively render the full simulation in Unreal Engine. As physically-accurate ocean rendering is not the focus of this report, we use a fairly simple setup that aims to bring somewhat believable results at a relatively low implementation-time cost. For a detailed discussion on realistic real-time ocean rendering, we refer readers to the work conducted by NVIDIA et al. [60, 23, 85, 79].

In Unreal Engine, the prototype is structured as a single scene containing:

- a set of lighting and rendering related objects (such as cameras and skylight);
- one actor responsible for the ocean simulation (including wakes);
- and a set of actors (pawns) each responsible for the simulation of a boat.

As for the lighting and camera configurations, we mostly used the default settings available in Unreal Engine. This minimal need for configuration helped alleviate most of the workload related to performing decent rendering, which represents one of the reasons behind why we chose to implement the prototype in Unreal Engine to begin with. Although, this alleviation is not our main motivation (see Section 3.2).

To generate and animate the shape of the ocean, each ocean patch is represented as a shared `UProceduralMeshComponent` (effectively a buffer on the GPU) that stores the static grid of vertices outlined by Equation 2.16. These vertices are then individually displaced (i.e. animated) using a custom `UMaterial`, which is effectively a GPU program (more specifically a combined vertex/fragment shader). The `UProceduralMeshComponent` is filled with initial vertex data from the CPU side (C++) while the `UMaterial` mostly maps simulation data (represented as render targets on the GPU) to Unreal Engine's coordinate system[14].

Besides shape, the appearance of 3D objects is largely determined by their shading. In computer graphics, shading largely relies on the use of *normals* (i.e. normalized normal vectors of the surface to render), which are typically computed and cached per mesh vertex. In this work, we derive the vertex normals of the ocean surface using finite differentiation in the spatial domain, and interpolate between these normals for any intermediate points. This is efficient both in terms of computation and memory costs, but it does sacrifice a degree of visual quality compared to analytical methods, especially when representing waves of small wavelengths [81, 85].

Besides normals and basic PBR[15] properties, which do most of the heavy lifting, we also color the surface of the ocean with a dark blue color that is slightly saturated

[14] Unreal Engine's coordinate system differs from ours both in terms of units and axis labels.

[15] Physically Based Rendering (PBR) is a popular rendering model used in modern computer graphics. Typical PBR properties include material metalicness, specularity, and roughness. For an overview, see: `https://learnopengl.com/PBR/Theory`

by a multi-layered simplex noise function. This noise, although not important, does help give the appearance of the ocean a little more variety, especially across patches. A similar white-colored simplex noise is used to visualize the foam that the wakes would generate. This foam-related noise is not applied across the entire ocean surface though, only where the wake velocity field (see Equation 2.50) is sufficiently active.

Another way to add more variety to the ocean's appearance is to render foam wherever the Jacobian of the horizontal displacement (see Equation 2.33) is negative, as explained by Tessendorf et al. [81, 23]. This approach is usually accompanied by a greater choppiness factor λ to better mimic the shape and appearance of highly agitated oceans. We did not implement this idea into our work partially because of time and partially to help visually clarify the distinction between the base ocean surface and the wakes.

3.2 The Proposed Framework

In Section 3.1.2 we described why the GPU-based wake simulation requires access to the CPU-based boats to create its obstruction and disturbance maps. Meanwhile, in Section 3.1.3 we described why the CPU-based boats require access to the GPU-based ocean (including the wake simulation results) in order to sample correct elevations for their buoyancy simulations. This communication dependency between the CPU and the GPU either creates a performance problem (see Figure 3.8) or a latency problem (see Figure 3.9), as explained in sections 1.1 and 2.5.

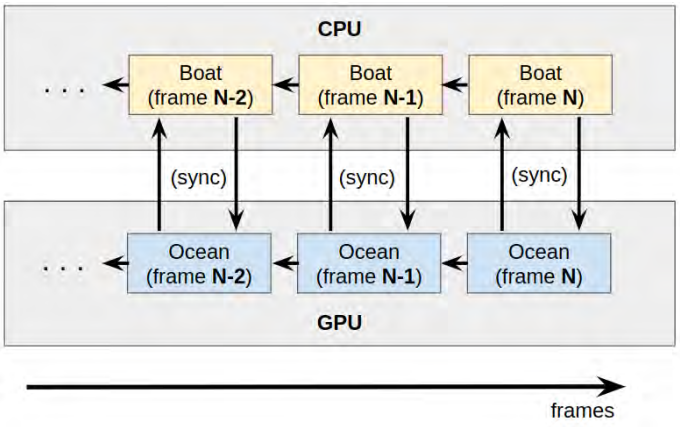


Figure 3.8: The state-of-the-art architecture in interactive real-time FFT-based GPGPU ocean surface simulations when synchronous readbacks are used. Each update step causes a graphics pipeline stall, which hurts performance.

The framework that we propose circumvents this problem by moving the external objects that interact with the ocean (i.e. the speedboats) to the GPU (where the ocean simulation resides), handling any interaction logic and implementation complications that follow. An overview of this framework is presented in Figure 3.10.

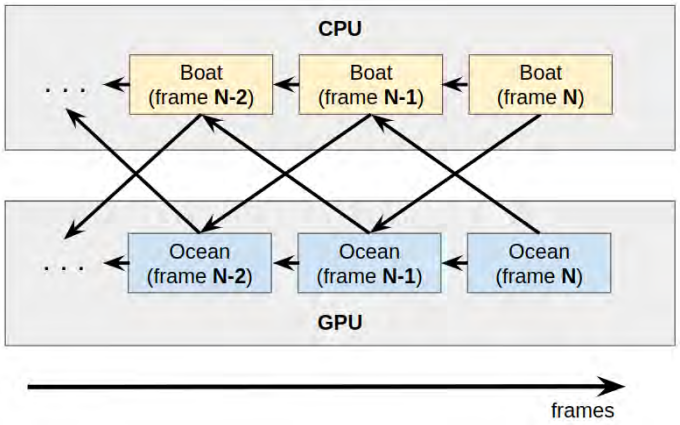


Figure 3.9: The state-of-the-art architecture in interactive real-time FFT-based GPGPU ocean surface simulations when asynchronous readbacks are used. States are updated asynchronously, causing latency (here depicted as a consistent 1-frame delay) that results in simulation updates with respect to old states.

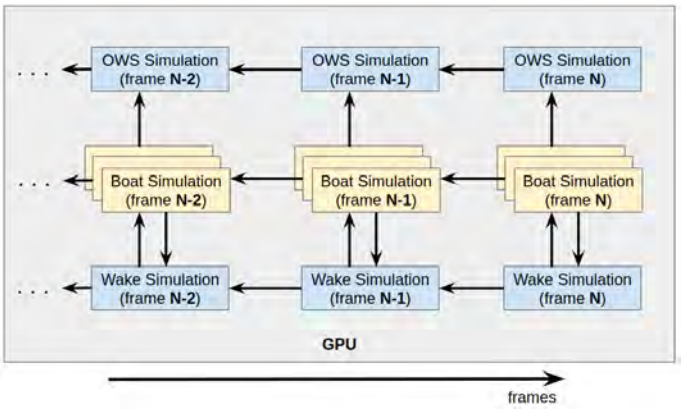


Figure 3.10: The architecture of our proposed framework. Unlike figures 3.8 and 3.9, all parts related to the ocean simulation are now stored and computed on the GPU. The ocean is here split into its non-interactive and interactive parts to highlight the related dependencies.

The main benefit of this architecture is that it completely removes the need for GPU readbacks, which were previously needed to simulate the ocean in interaction with the boats. In comparison with synchronous GPU readbacks, the proposed architecture should greatly improve performance, as a graphics pipeline stall is now avoided. In comparison with asynchronous GPU readbacks, the proposed architecture should noticeably improve simulation accuracy, as the latency concern is now completely removed (since there are no longer any asynchronous memory accesses involved).

The act of GPU-accelerating the physics simulation of the boats in itself may also bring some beneficial properties. Notably, the frame-by-frame cutting of triangles into fully submerged triangles and the computation of any related forces can now be processed in parallel. For a small triangle count, this performance gain might be negligible, but it could become quite significant for a large triangle count, which

might be desired in more accurate simulations. Moreover, since the boats now have access to the continuous ocean model, it might be possible to perform much more accurate (continuous) solutions, but this is beyond the scope of this report.

Moving the boats to the GPU is not entirely unproblematic though. To start with, rather than solving the original problem of latency-free communication between the CPU and the GPU, it circumvents it. In other words, the problem is still present, but it now exists between any objects still simulated on the CPU (e.g. characters) and the GPU-based boats (or the GPU-based ocean). Two possible solutions are:

- to handle these interactions using the traditional approach of GPU readbacks, effectively making a trade-off prioritizing the accuracy of the ocean simulation.
- to transition CPU-based objects to and from the GPU whenever they enter, respectively exit, a volume of space where these interactions are expected. For instance, a CPU-based object could be transitioned to the GPU whenever it is about to make contact with the GPU-based ocean.

The first option is straightforward to implement, but retains a trade-off in interaction accuracy. However, as many modern vehicle-related games aim for a high degree of realism [34, 90, 71, 52], this trade-off might be preferable for realistic ocean-focused applications. The second option mitigates the interaction accuracy concern altogether, but would involve a significantly more complicated implementation. In particular, any objects that should support such transitions would require both a CPU-based and a GPU-based implementation (as well as the transition-performing code itself). As the CPU and the GPU are fundamentally different pieces of hardware, these implementations could easily desynchronize. Moreover, having multiple implementations would also result in a lot of code to both write and maintain[16]. These challenges pose an interesting problem though, and it might be worthwhile exploring this option further in future work (see Section 5.3).

Regardless of the option chosen, there remains a final[17] concern when considering the proposed framework, and that is built-in (or library) support. Notably, most of Unreal Engine's built-in physics is CPU-based [93], which means that it can not be used for external objects simulated on the GPU. For instance, we had to write custom GPGPU implementations of rigid bodies and quaternions when realizing boat physics on the GPU (see Section 3.2.1). Strictly speaking, this concern is not a limitation in itself, but it is generally seen as a good practice to rely on well-tested libraries whenever possible, and those options might be fewer on the GPU side than on the CPU side. We think that this lack of convenience might be the main reason behind why the proposed framework is not already in use, but it is therefore important to explore in this direction to see how significant the increased simulation accuracy, and possibly improved performance, truly are.

In a broader context, one may question why programs like Unreal Engine refrain

[16] Unless there are only a few types of simple objects that needs to support such transitions, in which case this option might be perfectly viable.

[17] To our knowledge. There might be more, possibly application-dependent, concerns to consider.

from performing all physics simulations on the GPU instead of on the CPU. Apart from the design and memory complications discussed so far, one major disadvantage of such an approach is that the inherent architecture of current GPUs make them unsuitable for efficiently performing long sequential tasks with complex branching [68]. In particular, most GPUs are designed after a Single Instruction Multiple Data (SIMD) scheme, which allows for superior performance over CPUs when the given workload can be evenly distributed into identical sequences of operations that can be executed in parallel threads at a large scale [68]. Although some problems can be adapted to suit these conditions, such as by splitting a boat's body into many identically-operating triangles (whose forces require no branching), there are also many physics problems that do not fit this category, especially when paired with tree-based data structures (such as a bounding volume hierarchies) that cause branching control flow. And for these problems, the CPU will typically excel.

Next, we will cover the major modifications that were made from Section 3.1 to adapt the simulation to the proposed framework.

3.2.1 Adapting The Implementation

The most obvious adaption required is the boat model itself. Previously the boat state was stored on the CPU, in the form of a C++ `struct`. Now, we instead store the boat as a texture[18] of two rows, the first row storing the current state and the second row storing the previous frame's state (see Section 3.1.2.5). These state values are stored as 32-bit floating-point numbers, which matches the precision used on the CPU side.

When updating a boat, we first run a compute shader that processes each triangle of the boat's simulation mesh (one triangle per thread) and produces a structured buffer of submerged triangles as its result. As the list of submerged triangles from the cutting algorithm is at most twice the number of triangles of the original simulation mesh [44], we chose to keep the structured buffer fixed in size and denote empty slots by assigning an area of 0 to those triangles. Once the submerged triangles are ready, we provide them to a compute shader that updates the state of the boat by computing and applying the forces experienced during that frame.

To continuously update the visual model of the boat (see Figure 3.6) with respect to the simulation results, we displace the vertices of that model in scene space using a custom `UMaterial` (much like how the ocean is animated). Having a camera then follow this boat should in theory not be a problem, but we experienced some issues when implementing this in Unreal Engine.

In computer graphics, we typically represent the camera as a set of matrices (possibly compressed into one). These matrices are usually modified on the CPU (where the camera can be moved along with CPU-based objects) and continuously uploaded to

[18] A structured buffer could have been used instead, but we experienced a few issues (unrelated to the framework) when manipulating these through Unreal Engine's API. We therefore kept the use of structured buffers in general to a minimum.

the GPU (where the camera is needed to render the next frame). This flow appears to be the model that Unreal Engine expects. However, once uploaded to the GPU, the camera matrices can normally still be modified freely (e.g. using compute shaders), which can be used to track GPU-based objects, create special effects, and more. This freedom does not appear to be available in Unreal Engine without modifying (or possibly overloading) some of the engine's source code, because the engine does not inherently expose direct access to the matrices, prohibiting us from manipulating them further. This constraint highlights one of the two main reasons for why we decided to implement our prototype in a modern game engine to begin with.

In a custom application, unrestricted by the constraints of an external framework or library, it is always possible to adapt the flow of data according to some theoretically plausible framework, because the application owns the memory of all the data that is used. In contrast, modern commercial applications are typically built using such frameworks and libraries[19], limiting access to certain borrowed data. In the realm of 3D applications, one popular such choice is Unreal Engine [19, 1, 60]. Consequently, Unreal Engine was chosen because it would enforce similar restrictions on the design of our framework to those typically imposed on modern 3D applications. The other main reason behind the choice was that it would give us realistic performance measurements, as the engine overhead would prevent the application from being solely optimized to only perform ocean simulations efficiently.

Going back to the camera issue, if it is not possible to gain direct access to the camera matrices on the GPU, and if the camera should follow a GPU-based boat, then we suggest performing an asynchronous readback of the boat's transform that the camera can then follow. This approach reduces the responsiveness of the camera slightly, but preserves the accuracy of the simulation, which is why we settled with it in the prototype[20]. If direct access to the camera matrices on the GPU is possible, then we suggest to first process them through a compute shader that applies the desired camera logic[21], before passing the matrices further to any other shaders.

As for any interactions between the ocean and the boats, the implementation does not alter significantly, except that any GPU readbacks are omitted. Previously, the CPU state of the boat needed to be uploaded to and represented on the GPU in order for the wake simulation to process it. Also previously, the GPU state of the ocean needed to be downloaded to and represented on the CPU (in the form of elevation samples) in order for the boat simulation to process it. In the proposed framework, everything is represented on the GPU, which means that no duplicate representations (i.e. copies of data) have to be transferred or made. Thus, the proposed framework may complicate the design of some aspects, but it actually simplifies the interaction logic between the boats and the ocean[22].

[19] For several legitimate reasons, such as decreased development costs, well-tested code, faster time to market, separation of concern, security aspects, support, thorough documentation, etc.

[20] Although, we made the camera follow the GPU-based boat an optional setting, making the asynchronous readback also optional.

[21] Similar to how it would be done on the CPU.

[22] A pleasant property if the interaction between the ocean and external objects is the core of the application.

As a final touch, we also added support for interactions between the GPU-based boats such that they can collide with each other (as seen in Figure 4.5). For these interactions, each boat's collision box is simplified as a spherocylinder, and collisions are simulated by applying repelling rigid body forces. See `GPUBoat.usf` for details.

3.3 Evaluation Methodology

To evaluate our proposed framework, we compare it with a state of the art implementation that uses CPU-based boats in combination with either synchronous or asynchronous readbacks of the GPU-based ocean. As alluded to in Chapter 1, this comparison will consider both simulation accuracy and performance. Notably, from our discussion so far, we should expect (when compared to GPU-based boats):

- worse performance but accurate simulation from CPU-based boats with synchronous readbacks;
- and similar performance but worse simulation accuracy from CPU-based boats with asynchronous readbacks.

Next, we describe how simulation accuracy and performance are measured for the different solutions, while the comparison results will later be presented in Section 4.2.

3.3.1 Simulation Accuracy

To measure the simulation accuracy of a particular simulation, we first need to determine a ground truth against which it can be compared. For this baseline, we may either use the GPU-based boats or the CPU-based boats that use synchronous readbacks, as both experience no latency-induced errors. By then running the target simulation (i.e. the one based on asynchronous readbacks) and the baseline in isolation for identical user-input sequences, we can measure when and by how much the simulation states deviate.

Unfortunately, as we found out during development, Unreal Engine version 4.26 does not expose an option to perform asynchronous readbacks without partially causing a stall [14, 16], even though this is supported by the underlying graphics APIs (e.g. Direct 3D and Vulkan) and by the Unity game engine [91]. Furthermore, this problem appears to persist in Unreal Engine version 5 [15, 17], which officially released toward the end of this report's development. Fortunately, this limitation in Unreal Engine does not affect the GPU-based boats as they do not require any readbacks, but it does complicate the evaluation, as stalls greatly affect latency.

In response to the discovered limitations, we decided on a compromise where the latency itself is artificial, fixed, and customizable. Essentially, we run the baseline simulation but mock "readbacks" to have a fixed amount of frames in delay. Naturally, an artificial delay of 0 is identical to the original baseline. With this strategy, we can measure the effect that latency has on simulation accuracy in a controlled manner, by running the simulation with 1 frame of latency, 2 frames of latency, etc.

A limitation of measuring accuracy using a fixed artificial latency is that it does not faithfully capture the fluctuations in latency present in true asynchronous readbacks. To also cover this aspect of the comparison, we decided to make a simple project in the Unity game engine (also included in the source code of our prototype) that performs true asynchronous readbacks and records their latency. The resulting recordings form a discrete distribution of latencies (see Figure 4.16) from which we can then sample in Unreal Engine to also obtain an "organic" delay for readbacks.

Finally, to ensure consistent and reproducible simulations, we initialize the prototype with a random number generation (RNG) seed that can be configured within the Unreal Engine editor. This step is required as the ocean is stochastic.

3.3.2 Performance

When measuring and comparing performance, we also encountered problems related to limitations in Unreal Engine. In order to properly utilize the potential of modern graphics APIs (e.g. Vulkan), the rendering hardware interface exposed by Unreal Engine has been migrating toward a graph-based execution model called the Render Dependency Graph (RDG). Given the novelty of the RDG, its documentation and supporting infrastructure (including profiling) in Unreal Engine version 4.26 [12] is rather sparse. However, with the release of Unreal Engine version 5.0, proper performance profiling has finally been supported through the Unreal Insights program [13]. Unfortunately, as of writing, the Unreal Insights program either crashes or displays corrupted data if one attempts to profile the RDG[23]. This is problematic in our case, as all interactions with custom compute shaders go through the RDG, which accounts for the majority of our codebase.

In conclusion, the lack of profiling support for the RDG combined with the stall-inducing asynchronous readbacks (see Section 3.3.1) leaves us with less fine-grained means of measuring performance. We decided to measure performance of individual shaders by using RenderDoc [42], which is able to record the Vulkan commands that Unreal Engine internally submits to the GPU, allowing us to then replay and measure the cost of those commands (within RenderDoc). Unfortunately, RenderDoc cannot measure graphics pipeline stalls, which presumably Unreal Insights could. Thus, to see the effect of such stalls and the performance of complete simulations, we also measure the overall CPU and GPU costs of a selection of scenes (see tables 4.2 and 4.3). Moreover, as asynchronous readbacks ideally should not cause any pipeline stalls, we will for comparison purposes assume that their cost is negligible, which in reality will underestimate the cost of the CPU-based boats that use asynchronous readbacks (as CPU-GPU data transfer itself is not completely negligible).

Finally, to gather the performance data itself, we ran the prototype and the various simulation types on the hardware setups presented in Table 4.1.

[23] At least on Windows. This crash is unrelated to our project as it also occurs on official Unreal Engine sample projects and even empty projects. It is possible to avoid the crash, but then Unreal Insights will display corrupted data instead, suggesting a greater bug in the engine source code.

4

Results

In this chapter, we present some visual results of our implementation (described in Chapter 3) followed by a comparison between the state-of-the-art solution and our proposed framework, first by simulation accuracy and then by performance.

The prototype code is available at: `https://github.com/NeonSky/master-thesis`

4.1 The Implementation

A prototype was implemented of our proposed framework as an application made in Unreal Engine (primarily using C++ and HLSL). This prototype is primarily intended to be interacted with through the Unreal Engine editor, as that is where all user configuration takes place. The application itself only runs and renders the simulations themselves (without providing any heads-up display), aside from handling user inputs for the boats (which are steered through the keyboard).

Figure 4.1 shows the ocean in isolation (i.e. the non-interactive OWS-based ocean simulation), Figure 4.2 highlights the interactive wake simulation for a single speedboat, while Figure 4.3 demonstrates the buoyancy part of the boat physics simulation. Figures 4.4-4.8 clarify the problem of latency in simulations, and to what degree this can immediately affect the visual outcome in practical applications. Finally, Figure 4.9 shows a screenshot of the application in typical use and is intended as a summary of how the various simulations fit together.

Figure 4.1: Three different instances of our non-interactive ocean simulation, produced using identical parameters but different random seeds. Half of what makes ocean simulations form convincing illusions of real-life oceans lies in the movement of the waves, which unfortunately is lost when presented as static images like this.

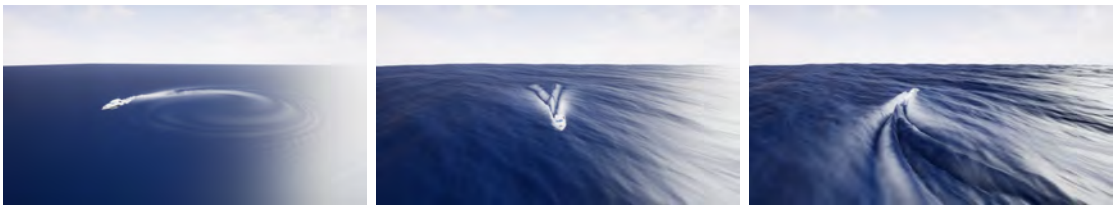


Figure 4.2: The ocean simulation when combined with a GPU-based speedboat. In the leftmost image, the amplitudes of the waves formed by the non-interactive ocean have been reduced to zero, leaving only the wake simulation visible. The other two images include the non-interactive ocean with varying degrees of agitated water. Notice the V-shaped Kelvin wakes formed behind the boat.

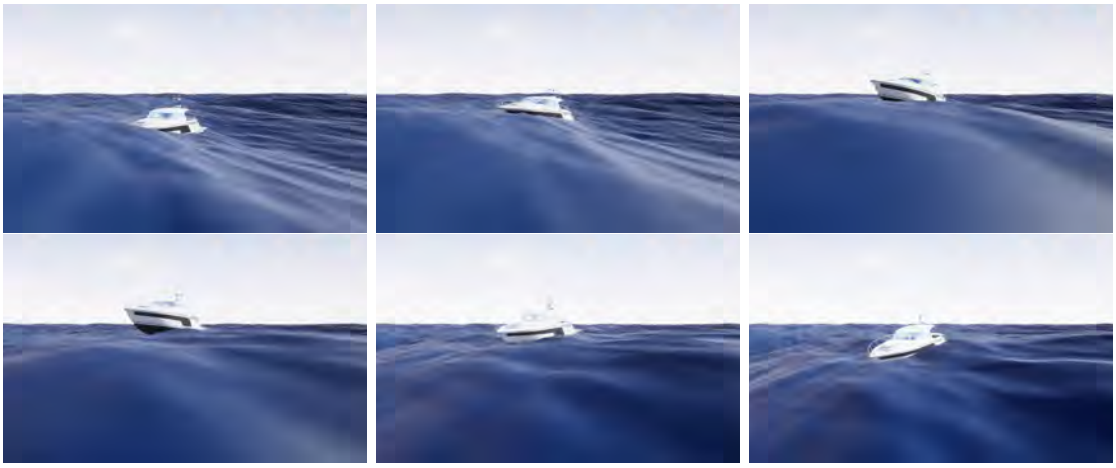


Figure 4.3: A sequence of six non-consecutive frames (from top-left to top-right, then from bottom-left to bottom-right) demonstrating a boat following the shape of the ocean surface due to buoyancy. In this case, the boat is not subjected to any user input, which means it is simply floating along the ocean surface based on the gravitational wind waves.

Figure 4.4: A sequence of three non-consecutive frames (from left to right) showing how wakes formed by one boat also affect the elevations perceived by other boats.

Figure 4.5: A sequence of three non-consecutive frames (from left to right) demonstrating a collision between two GPU-based boats as one drives into the other.

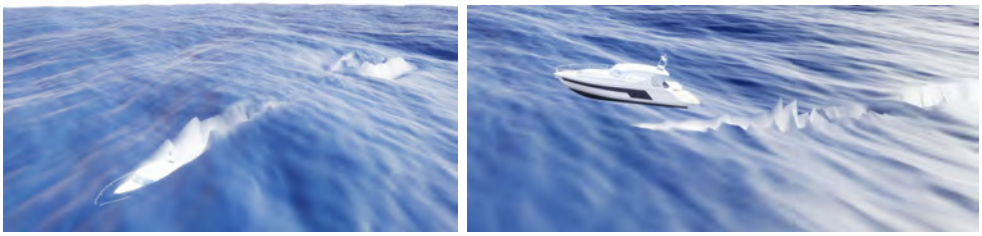


Figure 4.6: Examples of problems that occur when a speedboat experiences an exaggerated artificial readback latency of 60 frames. In the left image, the boat drives a tunnel through the big wave as it still has not noticed the increased elevation. In the right image, the boat drives in the air as it still thinks that it is partially submerged (consequently also forming wakes beneath it) after driving off a wave.

The intention behind Figure 4.6 is simply to clarify the kinds of problems that become immediately visible once latency is introduced. Meanwhile, figures 4.7 and 4.8 are presented to show the degree to which these problems materialize in practice.

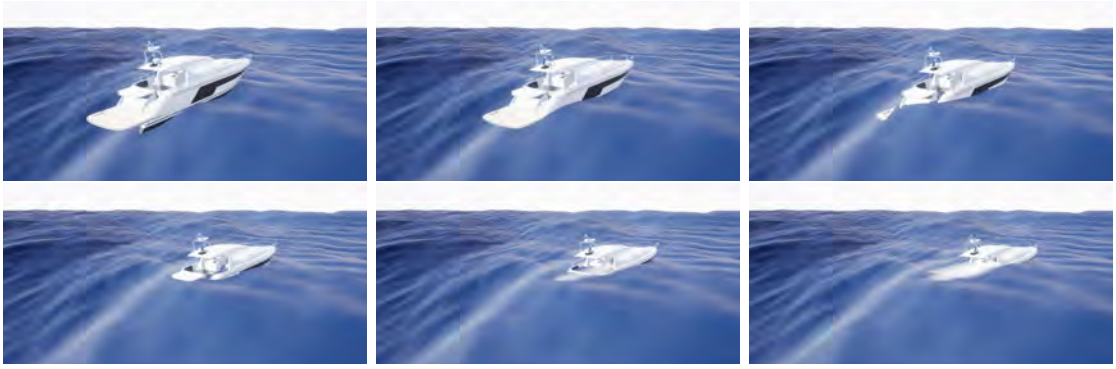


Figure 4.7: A sequence of six frames (from top-left to top-right, then from bottom-left to bottom-right) where a boat, experiencing an artificial readback latency of 3 frames, is driving into a wave. On frame 1, the boat has not started to intersect the ocean surface yet. On frames 2-4, the boat is intersecting the ocean surface but wakes and foam are not forming as the boat still does not realize it has started to submerge. On frame 5, the boat receives the elevations requested on frame 2. As the boat here detects that it is submerged, it now starts to form wakes and foam from this frame and onward.

Figure 4.8: A sequence of six frames (from top-left to top-right, then from bottom-left to bottom-right) where a boat, experiencing an artificial readback latency of 3 frames, is driving off a wave. On frame 1, the boat is still intersecting with the ocean surface. On frames 2-4, the boat still believes it is partially submerged, causing foam to still expand beneath it. On frame 5, the boat receives the elevations requested on frame 2. As the boat here detects that it is no longer submerged, it now stops forming wakes and foam from this frame and onward.

Figure 4.9: A screenshot of our prototype under normal operation.

4.2 Comparison Against State of the Art

The hardware setups used in the following comparisons are presented in Table 4.1.

Alias	CPU	GPU	RAM
$\mathfrak{A}$	Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz	NVIDIA GTX 1070 Ti	16 GB
$\mathfrak{B}$	AMD Ryzen 7 5800H 3.20GHz	NVIDIA GeForce RTX 3060 Laptop GPU	16 GB

Table 4.1: The hardware setups used to gather the results presented in this chapter, as well as the aliases used to refer to these individual setups.

4.2.1 Simulation Accuracy

In this section, we compare our proposed framework with the state of the art (using asynchronous readbacks) in terms of simulation accuracy. For the reasoning behind the way that this comparison is conducted, we refer readers to Section 3.3.1.

To form a baseline, we first drive a boat around the ocean in an arbitrary manner for a random RNG seed, recording all user inputs in the process. In our case, this recording was 1000 frames long with a fixed delta time of 0.02 seconds for the simulations (i.e. 50 simulation updates per second), equating to $1000 \cdot 0.02 = 20$ seconds of elapsed time in the simulation. By then performing a playback of this user input, we can obtain a baseline for any RNG seed. In Figure 4.10, we visualize how a boat experiencing zero latency would travel on a completely flat ocean, if subjected to the user input that we recorded for comparison purposes in this section.

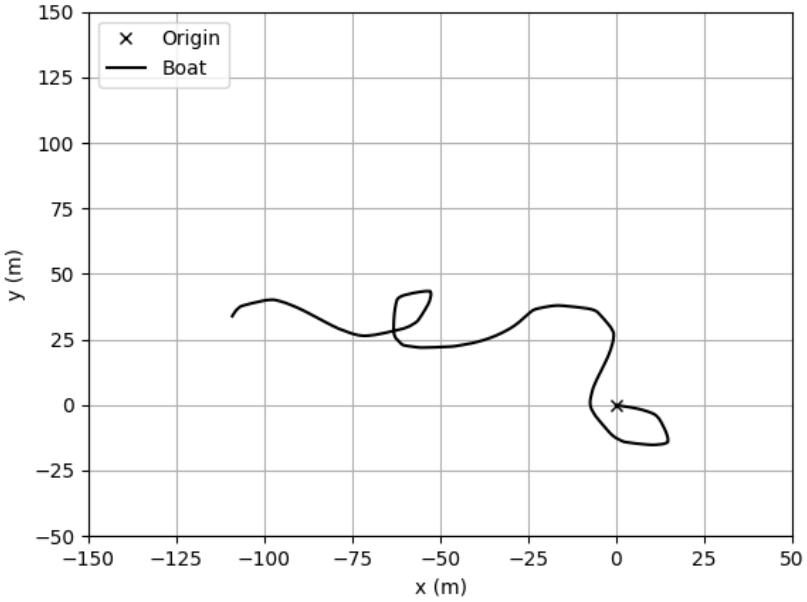
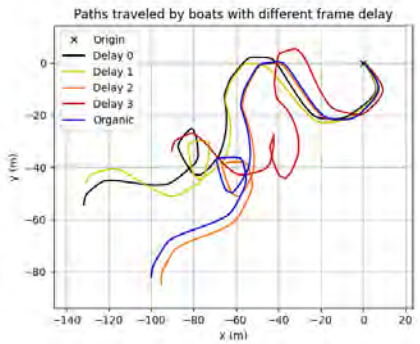
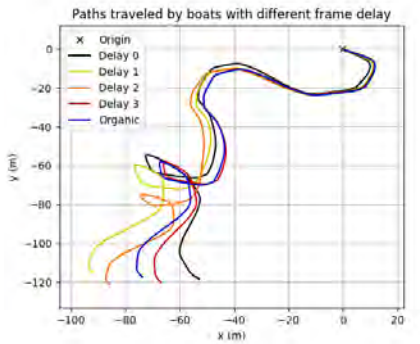


Figure 4.10: A speedboat driving on a flat ocean when subjected to an arbitrary sequence of user inputs. This plot presents a 2D top-down perspective of the world space, in which the boat starts at the origin on frame 0. Due to buoyancy and gravity, the boat also fluctuates vertically, but that axis is omitted from this view.

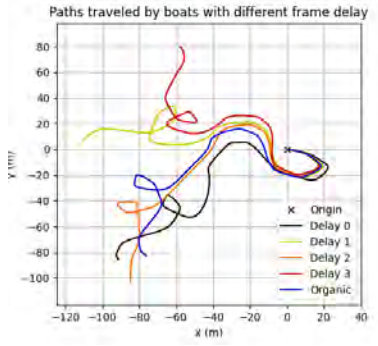
By playing back the user input (visualized in Figure 4.10) for boats traveling on oceans of varying RNG seeds, we obtain slightly different paths. For each such setting (i.e. RNG seed), we can then compare how the baseline simulation model of zero latency compares to those with fixed artificial latency, as well as to a boat experiencing "organic" latency (see Section 3.3.1). In Figure 4.11, we present path comparisons for a few such settings. The chosen set of artificial latencies was motivated by an experiment performed in the Unity game engine, whose results can be found in Figure 4.16, presented at the end of this section.



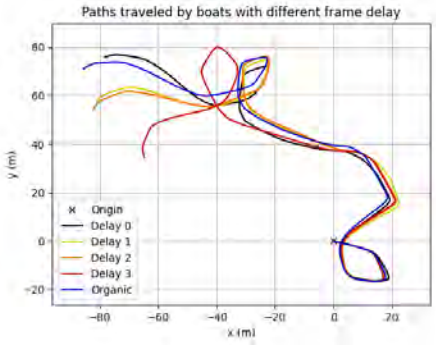
(a) Traveled paths for RNG seed = 11.



(b) Traveled paths for RNG seed = 12.



(c) Traveled paths for RNG seed = 32.

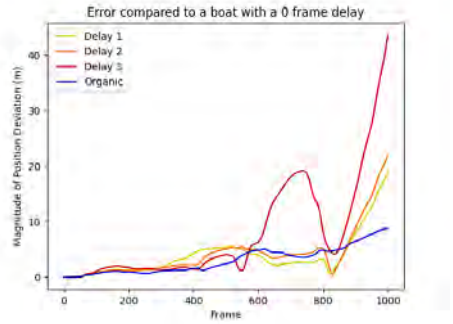


(d) Traveled paths for RNG seed = 9.

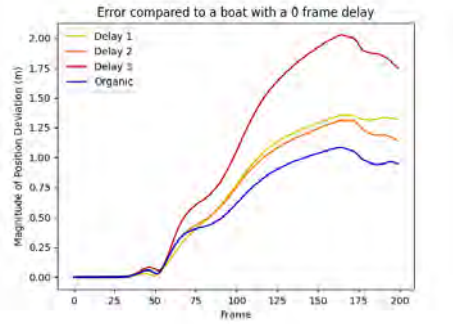
Figure 4.11: The paths taken by the boat when subjected to 0-3 frames of artificial latency, as well the "organic" latency, for four different RNG seeds. Notably, a higher delay does not always result in a greater deviation and the paths become more unpredictable as time elapses, suggesting a chaotic system. Each boat instance was simulated in isolation, preventing multiple boats from affecting each other's path.

As we can see in Figure 4.11, the boats that experience any form of latency eventually deviate from the baseline, and this deviation typically worsens over time. In particular, the "Delay 3" track of Figure 4.11a shows how a deviation early on can lead to a great deviation for the remaining path. To measure the magnitude of the deviations observed in these plots, we may measure the distance in world space by which the various boats differ from the baseline at any given frame. In Figure 4.12, we present the evolution of this error for the paths presented in Figure 4.11.

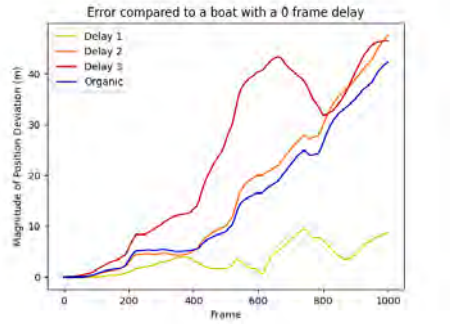
When observing the evolution of the deviation magnitudes for individual examples, as in Figure 4.12, it does not become apparent whether the severity of the delay affects the long-term deviation error. In particular, the behavior following the initial deviation tends to be rather chaotic. However, if we pay close attention to the initial frames, it appears that the severity does affect the magnitude and occurrence of the initial deviation. Moreover, this pattern is not limited to the examples presented in Figure 4.12, and instead generalizes to all of the individual RNG seeds that we have measured. If we instead average the error plots for many different RNG seeds, the patterns become much more clear, as shown in Figure 4.13.



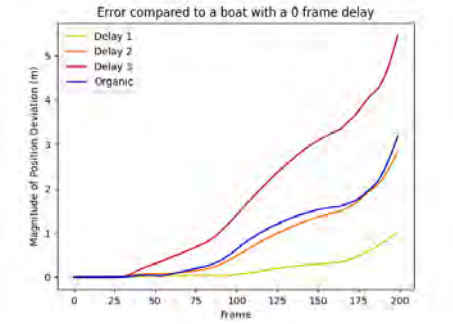
(a) First 1000 frames of RNG seed 9.



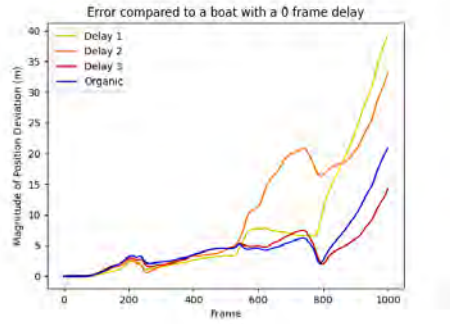
(b) First 200 frames of RNG seed 9.



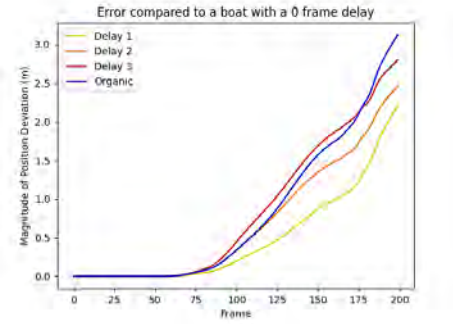
(c) First 1000 frames of RNG seed 11.



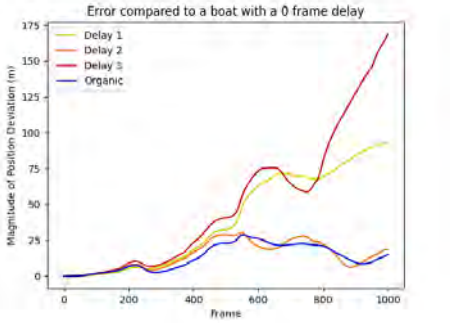
(d) First 200 frames of RNG seed 11.



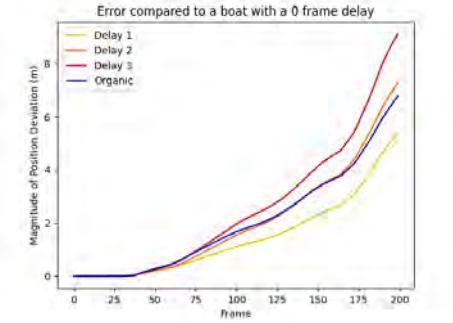
(e) First 1000 frames of RNG seed 12.



(f) First 200 frames of RNG seed 12.

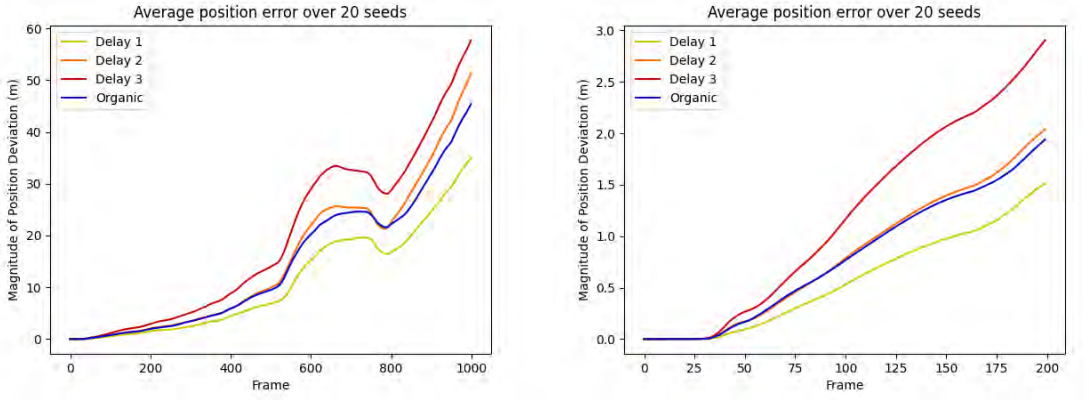


(g) First 1000 frames of RNG seed 32.



(h) First 200 frames of RNG seed 32.

Figure 4.12: Magnitude of deviation errors measured as the distance in world space between the baseline boat experiencing zero latency and various boats experiencing different degrees of latency. The four RNG seed examples presented here coincide with those presented in Figure 4.11. Note the different y-axis scales.



(a) 1000 frames of simulation. **(b)** The first 200 frames.

Figure 4.13: The deviation errors for 20 different RNG seeds, averaged on a frame-by-frame basis into a single plot. The left image shows the evolution of the deviation errors for all measured 1000 frames, while the right image shows a zoomed-in version of the first 200 frames. A set of 20 arbitrary RNG seeds was deemed sufficient, as averaging over more seeds did not appear to change the plot significantly.

The averaged errors presented in Figure 4.13 suggest not only that the magnitude of the delay affects the initial deviation, but also the long-term deviation. However, as suggested by Figure 4.12, these long-term patterns are only probabilistic, as individual settings can still appear quite unpredictable. This chaotic behavior is not unexpected though, since each boat will be subjected to increasingly different ocean conditions the more that they deviate from the baseline.

Besides the transform of the boats, one can also investigate by how much the interactive wake simulation deviates over time. In Figure 4.14, we present a similar error plot where the deviation in wakes is measured by computing the root-mean-square error (RMSE) of the wake heightfield texture H against the baseline.

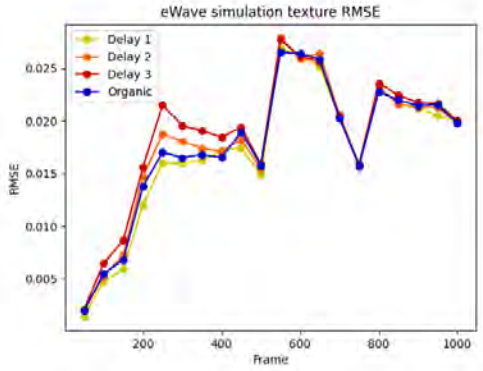


Figure 4.14: RMSE of the wake heightfield texture H when averaged over the same 20 RNG seeds as Figure 4.13 and measured at 50-frame intervals. Normalization is performed such that an RMSE of 0.025 represents an average pixel-difference of 0.025 (where pixel values are normalized to $[0.0, 1.0]$).

It is worth noting that the transforms of the boats and the outcome of the wake simulation are highly correlated. In Figure 4.15 we present screenshots from the prototype that demonstrate how these deviations may affect the simulation outcome for a particular instance.

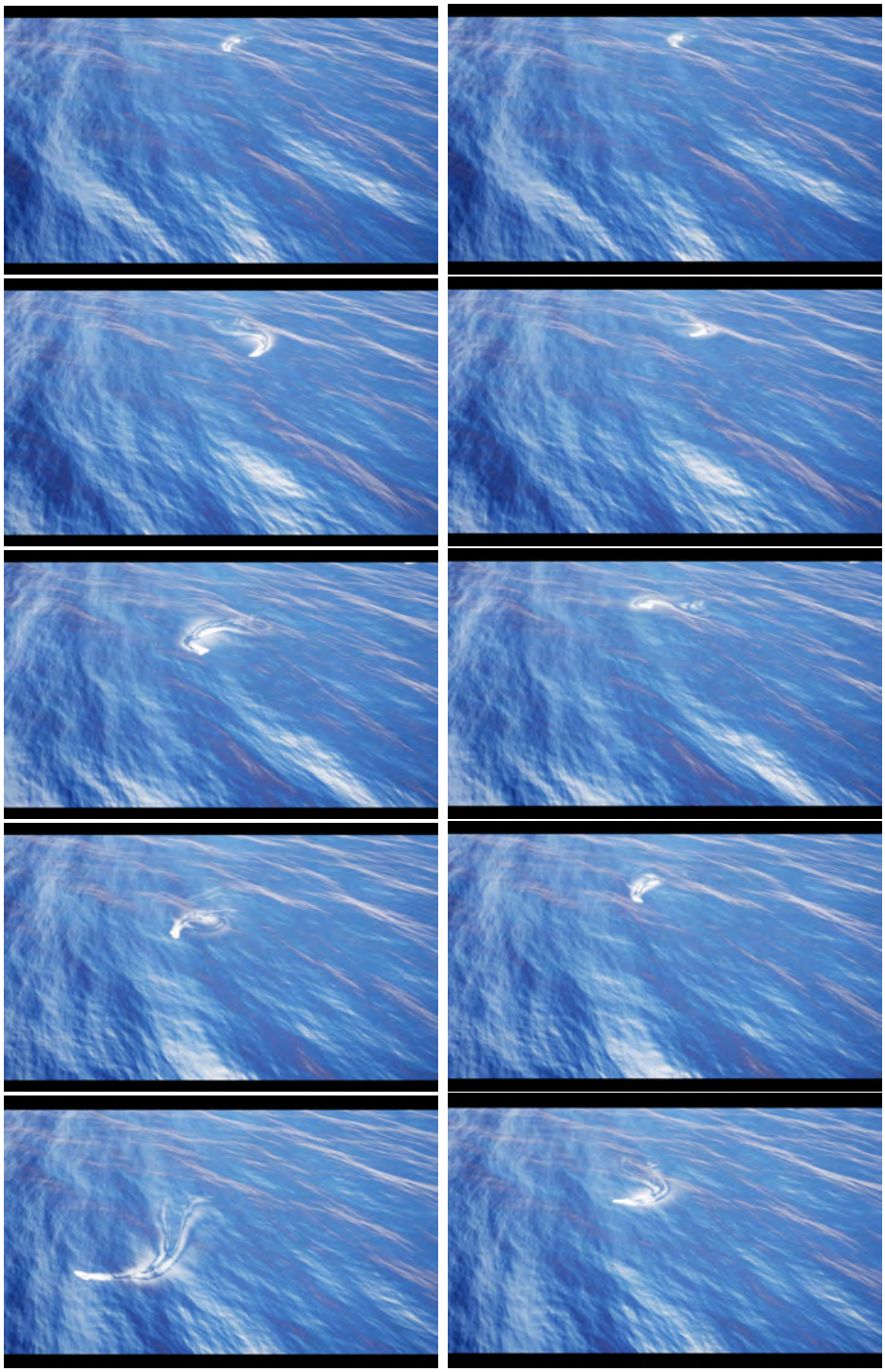


Figure 4.15: Screenshot comparisons between the baseline (left column) and a boat experiencing an artificial delay of 3 frames (right column) for RNG seed = 11. From top to bottom, each row compares the simulation state at frames 200, 400, 600, 800, and 1000, respectively. As expected, the deviation worsens over time.

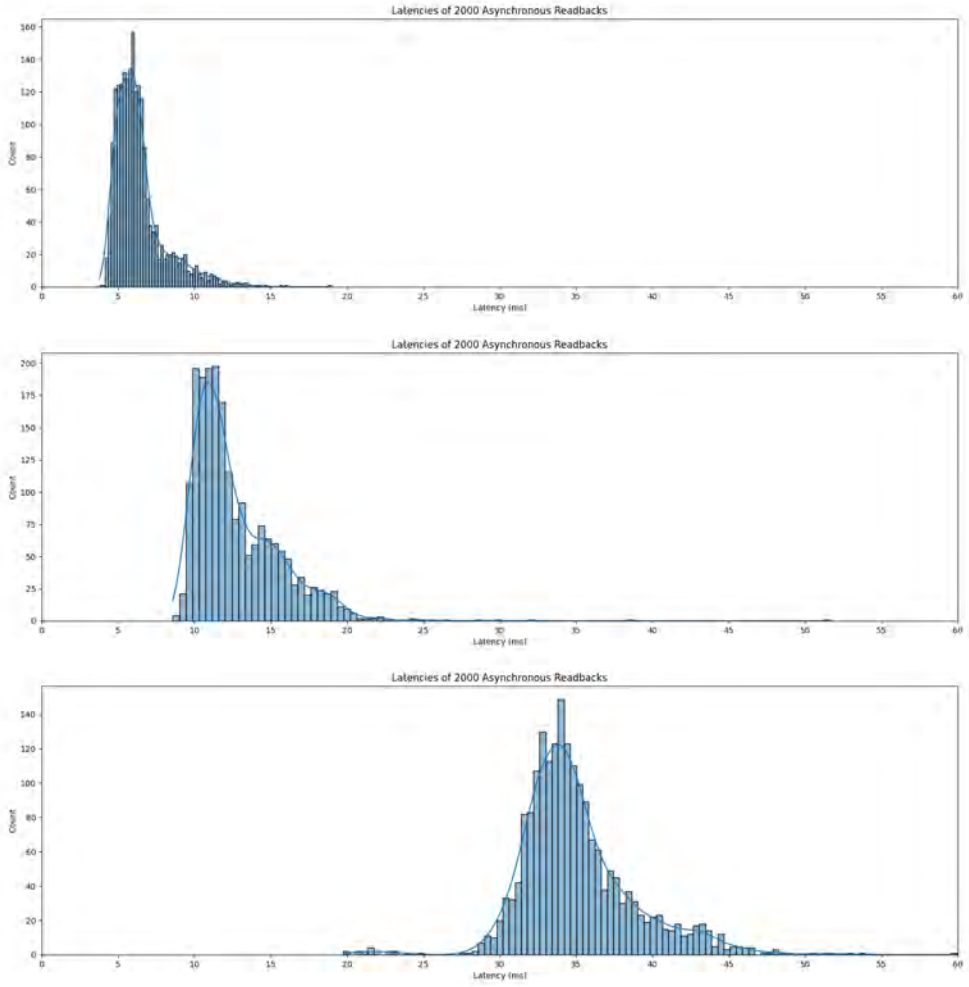


Figure 4.16: Discrete distributions obtained from performing asynchronous readbacks in the Unity game engine and measuring their latencies. Each graph represents measurements taken in scenes of varying complexity. From top to bottom: an empty scene (performing at ∼300 FPS), a default particle system with a particle emission rate of 10000 particles per frame (performing at ∼140 FPS), a default particle system with a particle emission rate of 15000 particles per frame (performing at ∼60 FPS). The bottom-most distribution is used as our "organic" latency, since it closely resembles the expected workload. The distributions themselves are plotted using a maximum of 80 bars/bins. The data presented here was collected on setup $\mathfrak{A}$.

4.2.2 Performance

See Section 3.3.2 for the reasoning behind our approach to compare the state of the art with our proposed framework. Furthermore, see `PerformanceMeasuring.md` in the prototype source code for detailed instructions on how to reproduce the performance measurements presented in this section using the Unreal Engine editor.

First, we present the GPU and CPU costs for various settings (or scenes) of the prototype. In Table 4.2, we examine the average frame-by-frame costs of running the simulation when rendering an empty scene. These measurements are intended to highlight the costs of the simulation itself, in disregard to how it is rendered. Meanwhile, Table 4.3 presents the same settings, but for fully rendered simulations, resulting in more realistic costs for practical purposes.

	Setup $\mathfrak{A}$		Setup $\mathfrak{B}$	
Setting	**GPU**	**CPU**	**GPU**	**CPU**
Empty scene	2.215	0.000	3.087	0.000
Only the (non-interactive) ocean	2.843	0.015	3.250	0.010
Ocean + 1 CPU-based boat (sync)	5.879	3.051	5.195	2.237
Ocean + 1 CPU-based boat (async)	3.111	0.406	3.940	0.326
Ocean + 1 GPU-based boat	2.924	0.354	3.801	0.252
Ocean + 2 CPU-based boats (sync)	8.389	5.587	6.252	3.773
Ocean + 2 CPU-based boats (async)	3.256	0.663	4.771	0.452
Ocean + 2 GPU-based boats	3.052	0.516	5.120	0.387

Table 4.2: The performance of the prototype when simulating various settings, but always only rendering an empty scene. All numbers are presented in milliseconds and represent the average cost of moving the simulation one frame forward. “sync” denotes synchronous readbacks, while “async” denotes asynchronous readbacks.

	Setup $\mathfrak{A}$		Setup $\mathfrak{B}$	
Setting	**GPU**	**CPU**	**GPU**	**CPU**
Empty scene	2.837	0.000	3.525	0.000
Only the (non-interactive) ocean	3.079	0.015	3.448	0.010
Ocean + 1 CPU-based boat (sync)	8.199	5.510	7.429	4.641
Ocean + 1 CPU-based boat (async)	4.633	0.457	5.345	0.313
Ocean + 1 GPU-based boat	4.523	0.350	4.574	0.251
Ocean + 2 CPU-based boats (sync)	12.722	10.222	9.876	7.691
Ocean + 2 CPU-based boats (async)	6.495	0.616	7.202	0.428
Ocean + 2 GPU-based boats	5.555	0.531	6.375	0.377

Table 4.3: The performance of the prototype when simulating and rendering various settings. All numbers are presented in milliseconds and represent the average cost of moving the simulation one frame forward. “sync” denotes synchronous readbacks, while “async” denotes asynchronous readbacks.

When comparing these tables, we rest assured that the GPU costs in Table 4.2 do not appear to exceed their counterparts in Table 4.3. From both tables, we note that the CPU costs appear to be small for all settings except for those including synchronous readbacks. In particular, due to causing pipeline stalls, the synchronous readbacks lead to considerably worse performance on the GPU, but especially on the CPU. Moreover, the reason why "Empty scene" differs between the two tables is that in Table 4.3, the ocean mesh is still visible (but not animated) in this setting.

From Table 4.2, we note that the CPU-based boats that use asynchronous readbacks have similar GPU costs as the GPU-based boats. However, in Table 4.3, the GPU-based boats appear to have consistently smaller costs on the GPU. In contrast, the GPU-based boats have a distinctly smaller footprint on the CPU for both tables. Though, in general it appears that rendering has a higher performance footprint than increased simulation complexity, which is further implied by Table 4.4.

In Table 4.4, we approximate the average GPU costs of the three core simulations that make up our prototype, as well as the cost of performing a typical FFT pass.

Feature	Setup $\mathfrak{A}$	Setup $\mathfrak{B}$
Update non-interactive ocean	195.584	285.008
Update GPU boat	44.032	60.416
Update wakes	324.608	535.552
FFT (256×256)	84.992	145.408

Table 4.4: From row 1 to row 4: the average GPU execution times of updating the non-interactive ocean simulation, updating the GPU-based boat simulation, updating the wake simulation, and performing an FFT on a 256×256 texture (with 4 color channels and 32 bits per channel). All costs are presented in microseconds.

For comparison, we used the cuFFT [58] library's 2D FFT implementation and measured a noticeably smaller FFT (256×256) cost of 47.2 microseconds on setup $\mathfrak{A}$ and 17.4 microseconds on setup $\mathfrak{B}$. However, this cuFFT benchmark was performed in complete isolation of any Unreal Engine code and used CUDA functionality to measure the performance (as opposed to RenderDoc).

5

Discussion

In this chapter, we begin by recalling the two research questions that were established in Chapter 1, and discuss these with regard to the results of our study, presented in Chapter 4. In summary, we find that there indeed is a significant simulation error introduced by latency. We also find that this error is avoidable by running the entire simulation on the GPU, which does not hurt performance compared to performing asynchronous readbacks to the CPU. In Section 5.1, we focus the discussion on our results from Chapter 4, followed by a brief discussion of related work in Section 5.2, and conclude with a discussion on implications for future research in Section 5.3.

5.1 Our Work

We now discuss the two open research questions (RQ) we set out to investigate:

- RQ1: What is the significance of this simulation error in practice?
- RQ2: Could this simulation accuracy be recovered without hurting performance. And if so, then how?

5.1.1 Research Question 1

The most obvious significance of the simulation error is that it undesirably causes the simulation to deviate from its intended sequence of states, as seen in Figure 4.11. We can also appreciate the average growth of this error's magnitude in Figure 4.13, which suggests that a speedboat may reasonably deviate by ∼40 meters in world space after only 20 seconds of simulation, at least when comparing identical user input sequences. Such great deviations can likely be attributed to minor deviations to the boat's rotation, which then causes the boat to head in different directions. Thus, if a human controls the boat, as opposed to a fixed input sequence, then the overall deviation may substantially decrease, as the user can reactively compensate for the simulation errors. However, with ∼60 uncorrelated simulation errors per second, there will invariably be a limitation to such manual aid.

In contrast to the artificially fixed latencies used as references, an organic readback delay not only causes the simulation to deviate, but also to become nondeterministic. This is an unwelcome property in the context of competitive games, as equally

performing players should be rewarded equally. In particular, if players are given identical starting conditions in a boat race, then it would be unfair if identical user inputs from these players did not result in a draw, simply due to hardware limitations. With a nondeterministic simulation, this concern becomes prevalent regardless if the players compete simultaneously or in sequence (e.g. using a leaderboard or high score system). Consequently, both online play and offline time attacks (a.k.a. speedrunning) are negatively affected. Moreover, an exciting boat race is usually longer than 20 seconds, which only leaves more room for the latency-induced errors to become noticeable.

For certain applications, the believability of the simulation might be preferred over its correctness. For such applications, the significance of the simulation errors is much harder to objectively evaluate, at least at this stage of research. But from our personal experience in testing our own prototype, we hardly perceive any difference in the responsiveness of controlling the boat or the shape of the generated wakes. Based on this observation, we infer that it is unlikely that a user would be able to distinguish between the latencies presented in Figure 4.13, if only given limited experience with the different latency severities in isolation. However, when presented with the various severities simultaneously, the differences become much more apparent. Moreover, in esports it is desired to have minimal response times [78], so it is possible that some users could distinguish the severities regardless.

In addition to deviations in the simulation state, the presence of latency also introduces visual artifacts, most notably, when an external object enters or exits the surface of the ocean (see figures 4.7 and 4.8). Consequently, the generation of wakes starts and stops a few frames later than it is supposed to, which is likely a major contributor to the errors shown in Figure 4.14. Nevertheless, these artifacts are barely visible even with a keen eye, especially at high frame rates. The theoretical problems become obvious in Figure 4.6, but such scenarios are impossible unless the application is so demanding that it is running at non-interactively low frame rates.

5.1.2 Research Question 2

From tables 4.2 and 4.3, we infer that the proposed framework can recover the simulation accuracy without hurting performance (on either the CPU or the GPU side). In particular, we measure near-equivalent (if not better) performance for the GPU-based boats compared to the CPU-based boats that utilize asynchronous readbacks. Meanwhile, synchronous readbacks result in major performance hits, especially as GPU workload grows. Nevertheless, as explained in Section 3.3, the employed performance evaluation is not completely accurate. Thus, the individual costs are less trustworthy than the overall pattern that they suggest. Additionally, use of more complex collision detection would likely have favored CPU-based boats since associated data structures, namely bounding volume hierarchies (e.g. R-trees) [29], require frequent branching of control flow due to their tree-based hierarchies.

As shown in Table 4.4, the performance cost of our prototype is dominated by the interactive wake simulation (aside from rendering). This observation is expected since the wake simulation is composed of many sequential steps (see Section 3.1.2.4), requires several FFTs, and also needs to generate an obstruction map. Unfortunately, the number of wake simulations also scales with the number of boats to simulate, unlike the non-interactive simulation, which only intensifies the issue. However, this performance concern plagues both CPU-based and GPU-based boats, and is thus unrelated to our proposed framework. Nonetheless, it is of interest to mitigate this concern as our framework is mainly of relevance when the ocean is interactive.

A significant portion of the present GPU workload appears to come from the many FFT passes that need to be executed each frame. As seen in Table 4.4, each FFT pass costs roughly 100 microseconds, however, the performance measured from cuFFT [58] suggests that this implementation could be greatly optimized (see Section 4.2.2). Thus, the FFT implementation used in this work (and presented by Flügge [20]) seems to be less efficient than it could be. It is also possible that cuFFT is partially efficient because it only supports NVIDIA cards, but vkFFT [88] is another FFT library that promises similar performance to cuFFT [87] while also being cross-platform. However, the performance demonstrated by these libraries might not perfectly translate to an integration with Unreal Engine's RDG [13] system, and this integration in itself might be non-trivial to implement. Nevertheless, the potential performance gains are compelling enough to experiment with different FFT options.

There are a couple of results from our performance measurements that surprise us. The most surprising result is the (supposedly) smaller GPU cost of GPU-based boats than CPU-based boats using asynchronous readbacks (see Table 4.3). This pattern is less evident in Table 4.2, but potentially still present. We speculate that either Unreal Engine's RDG system or the GPU driver might have an easier time optimizing (e.g. performing commands in parallel) a deeper graphics pipeline when all tasks are on the GPU, as opposed to when the CPU occasionally interrupts with new read and write requests. For instance, the CPU-based boats have to upload a representation of their state to the GPU every frame. Another surprising result is that setup $\mathfrak{B}$, which hosts a more powerful CPU and GPU than setup $\mathfrak{A}$, on average performed significantly worse than setup $\mathfrak{A}$. An exception to this pattern is the cuFFT benchmark, where setup $\mathfrak{B}$ performed remarkably well. Our speculation here is that setup $\mathfrak{B}$ might suffer from thermal throttling when stress-tested to simulate and render scenes as fast as possible in Unreal Engine (see tables 4.2 and 4.3). This speculation is based on the fact that setup $\mathfrak{B}$ is a laptop with limited cooling, while setup $\mathfrak{A}$ resides inside a spacious desktop computer chassis. However, this speculation does not clearly justify Table 4.4, which is not a stress test.

Finally, as the proposed framework demonstrates that the simulation accuracy, lost from latency-induced errors, can be successfully recovered without hurting performance, we conclude that its usage should be considered within the context of real-time interactive ocean simulations. Furthermore, as we find no simulation limitations imposed by the proposed framework, we deem that it should be compatible with virtually any related work (whenever applicable).

5.2 Related Work

Although the two research questions that we set out to answer in this report have, to our knowledge, not been addressed in prior work, there have been plenty of research conducted on simulation and rendering of oceans. In this section, we highlight some options and extensions to the solutions mentioned and presented so far in this report.

5.2.1 Simulating Oceans

While FFT-based GPGPU oceans (animated using an OWS) is considered the state-of-the-art model for real-time performance budgets, there are several domains in which this budget is either much stricter or much more lenient.

If the budget is much stricter than real-time[1], then the arguably most efficient option would be to compound only a handful of sinusoids (in the spatial domain, without FFT). An example of this approach can be seen in the critically acclaimed game *The Legend of Zelda: The Wind Waker* [24], which employs a highly stylized ocean. For improved realism, one may instead replace the sinusoids with Gerstner waves (a.k.a. Trochoidal waves), which have been used in computer graphics as early as 1986, as presented by Fournier and Reeves [21]. These waves are similar in complexity, but while sinusoids only vertically animate individual fluid particles, Gerstner waves also animate them along the horizontal axes according to additional sinusoids. Consequently, each particle travels in a trochoidal orbit around its resting point, which sharpens peaks and flattens valleys. This concept eventually inspired Tessendorf's idea of *choppy waves* (see Equation 2.33) for FFT-based oceans [81].

Regardless of the wave model, it is possible to both save performance and realize an infinitely-spreading ocean by using a projected grid [30, 40]. The idea is to continuously project a fixed grid of points from the camera's lens into the scene, and then only evaluate the ocean simulation for those points. This way, we only simulate the part of the non-interactive ocean that is actually visible to the camera, and at a fixed visual fidelity. The main disadvantage of this approach is the visual aliasing artifacts that it produces [40, 8], especially near the horizon, as shown in Jérémy Bouny's WebGL demo [6]. Another option is to use a rough ocean mesh and then dynamically add detail through tessellation [67]. This option suffers less from artifacts, but is also more expensive since the performance cost is no longer fixed.

If instead the performance budget is much more lenient than real-time, then one might consider restricted forms of traditionally offline-focused solutions. Such solutions generally belong to Computational Fluid Dynamics (CFD), which is a vast research field whose applications in computer graphics is roughly categorized into two groups: Eulerian and Lagrangian methods [7]. These methods tend to be quite expensive, as they model 3D volumes of fluids rather than just their 2D surfaces.

[1] E.g., high-performance games where the ocean is part of the scenery rather then the action.

Eulerian methods model fluid bodies by using stationary 3D grid-based data structures (called *staggered grids*), where each grid vertex has associated properties, such as density, pressure, and velocity. A fluid can then be simulated within this staggered grid (e.g. using Navier-Stokes) and be visualized through a volume rendering algorithm, like Marching Cubes [49] or ray marching. Unfortunately, as all space within the grid (regardless if it is empty) needs to be simulated, Eulerian methods can become computationally expensive unless this space is properly utilized. Moreover, as the fluid is confined by this static grid, and the grid itself needs to be fixed in space, Eulerian methods can lack the flexibility that is sometimes desired when working with deforming fluids. The pioneering papers by Jos Stam [74, 75] provide an excellent introduction to the topic and uses of Eulerian fluid simulations.

Lagrangian methods, in contrast, model fluid bodies using a set of mobile fluid parcels[2] that are not constrained in space. Consequently, Lagrangian methods are more flexible than Eulerian methods and they do not need to simulate empty space. A less desired consequence is that each of the N fluid parcels can overlap and interact with any other fluid parcel, yielding a $\mathcal{O}(N^2)$ time complexity. Smoothed Particle Hydrodynamics (SPH) [4, 43] can help reduce the expected time complexity, but not the worst-case time complexity[3] of $\mathcal{O}(N^2)$. Moreover, since the fluid parcels may overlap (i.e. the fluid is treated as compressible [33]), one can expect the need to simulate a lot of them to realize a fluid that fills a certain amount of space. For further discussion on CFD-based methods, we refer readers to other work [33, 7].

As for interactive wake simulations, there are both alternatives and extensions to the eWave model that we employed in our work. One notable extension is the inclusion of ambient drift currents, described in Gilligan [85], which would disturb the overall advection of the wakes accordingly (e.g. along the wind direction). However, in the context of speedboats on an ocean, the generated wakes are often large enough to label the contribution of such drift as negligible. Instead, a far more useful extension would have been the use of a more sophisticated representation of obstructions. As-is, the binary representation used by eWave (see Section 3.1.2.1) works well for non-interactive ripples (as used in Gilligan [85]), but not for wakes that are powerful enough to affect the elevation of (other) boats. Either a more granular obstruction representation is needed or a different model altogether.

Other alternatives to eWave include: wave particles [96], wave packets [39], and wavelets [38]. These three papers loosely build on top of each other in the order presented, arguably forming the current state of the art in real-time wake simulations (see Section 3.1.2 for our motivation behind choosing eWave in spite of this). Unfortunately, these methods suffer from the same obstruction representation concern as eWave, however, the concern should be easy to mitigate in (at least) the former two [96, 39]. In these two methods, the lifetime of individual wave units (particles or packets) is tracked. Thus, one could transition the wave units from fully interactive to fully non-interactive as they emit from a boat. Once the wakes have become

[2] A fluid parcel is essentially a particle representing a small mass, and thereby volume, of fluid.

[3] Without sacrificing simulation accuracy. Otherwise, one can limit the number of neighbors that a fluid parcel can interact with to a constant, and thus get a linear (albeit error-prone) solution.

non-interactive, they are no longer affected (and thereby limited) by the obstruction representation, but would still affect the elevations perceived by external objects. A gradual transition into this state would help reduce any potential visual artifacts.

5.2.2 Rendering Oceans

Our work is primarily focused on the accuracy and interactivity of ocean simulations, but such simulations are ideally combined with impressive rendering, of which there is plenty of research on. In our work, we mostly relied on Unreal Engine's built-in rendering pipeline for PBR-based materials (see Section 3.1.4). Although this rendering approach is neither the most efficient nor the most visually stunning, it was deemed suitable for our work. However, in other contexts, a different balance between performance and visual quality may be desired.

If higher performance is required, then one might consider using scrolling textures, which involves moving a set of tileable water textures across the ocean surface (see Isidoro et al. for implementation details [41]). This technique has been used at least since *Super Mario 64* in 1996 and, with some further additions like displacement maps, is still used in more recent titles such as *Super Mario Galaxy 2* [36]. To achieve a more realistic effect, the scrolling textures may be combined with noise, such as Perlin noise [63, 64], Simplex noise [65, 25], or fractional Brownian motion (fBm) [51]. An example of such use can be seen in the work by Schneider et al. [69], where fBm is used to generate surface disruptions, reflections, and refractions.

If a more lenient performance budget is available and higher visual quality is desired, then there are at least three immediate improvements that can be made to the approach described in our work. Firstly, one can analytically compute the surface normals to capture waves of smaller wavelengths [81, 85]. Secondly, one can approximate subsurface scattering by employing a bidirectional surface scattering reflectance distribution function (BSSRDF) [10], which extends a bidirectional reflectance distribution function (BRDF) by modeling scattering of light within material surfaces [37]. Thirdly, one can split the OWS used by the non-interactive ocean simulation into frequency bands (a.k.a. cascades), which increases computational demand, but also significantly increases the visual fidelity [23, 62, 60]. For instance, the ocean used in our prototype has a mesh with $\sim$40 cm between neighboring vertices, while cascades could easily reduce this number to $\sim$2 cm [62]. Theoretically, a downside with cascades is that the higher frequency bands have easily noticeable tiling artifacts, but this issue is mitigated once the bands are summed up [60].

Besides the semi-transparent water surface itself, there are many more effects that go into rendering a realistic ocean. One such effect is foam. On the open ocean, foam occurs naturally in the form of whitecaps, which can be simulated using the Jacobian determinant of the horizontal vertex displacements [11]. Foam also forms in conjunction with wakes, in which case, noise, scrolling textures, and (possibly SPH-based) particle systems may be appropriate rendering options. One might even consider adding foam, along with splashes and spray, as a lightweight post-process [32]. Other notable effects include: caustics [46], god rays [81], and distorted

refractions [40]. However, great care should be taken when implementing effects involving transparency, because order-dependent alpha blending would require the need to sort fragments, of which there would be many for an expansive ocean surface.

5.3 Future Work

First and foremost, it would be valuable to utilize the Unreal Insights tool [13], once stable enough (see Section 3.3.2), to verify the GPU performance results presented in Section 4.2.2. Subsequently, it would be interesting to see how the performance of GPU-based boats scale when more accurate collision detection is performed.

Moreover, as the proposed framework is primarily of interest when the GPU-based ocean is interactive, it would be worthwhile to further explore improvements and alternatives to the interactive wake simulation used in this work. Such alternatives and improvements have been discussed in Section 5.2.1 in terms of simulation quality, but there still remains some concern regarding performance. As-is, the eWave-based wake simulation appears to be the performance bottleneck of the prototype (see Section 4.2.2), but as discussed in Section 5.1.2, this performance could potentially be significantly improved with a faster FFT implementation.

Finally, a remaining and more open-ended question is how CPU-based objects could be transitioned into GPU-based objects, and vice versa, in a maintainable and efficient manner. Because, if such a solution would exist, then one could dynamically group and distribute objects to the CPU and GPU, such that minimal (if any) asynchronous readbacks would be needed. Ideally, one would only need to express an object's behavior once, and then have this description compiled to both a CPU-based and a GPU-based implementation (as well as any transition logic required between the two). Albeit a massive undertaking, such a solution might become more feasible to explore if CPU and GPU architectures continue to become more similar.

6

Conclusion

In this thesis, we investigated: (1) the significance of the latency-induced simulation errors present in state-of-the-art solutions to real-time and interactive ocean surface simulations, and (2) how these errors could be eliminated without hurting performance i order to maintain simulation accuracy. To carry out this investigation, we (1) designed and proposed a GPU-focused framework that circumvents the use of any latency-inducing GPU readbacks within the simulation, and (2) implemented a prototype of said framework in order to compare it with the state of the art.

From our investigation, we found that the presence of latency within the simulation invariably results in a deviation from the simulation's expected outcome. More precisely, we found that speedboats subjected to identical conditions, including user input, can deviate in position by as much as 40 meters after only 20 seconds of simulation, simply due to latency. We also demonstrated that existing solutions relying on asynchronous readbacks, unless locked to a fixed latency, are subject to nondeterministic behavior, which can be troublesome in competitive contexts. Fortunately, we also observed that our proposed framework successfully avoids the latency-induced errors without hurting performance, consequently removing these concerns associated with latency away from the simulation.

In conclusion, with this work we contribute with (1) detailed discussions and concrete measurements of the latency-induced simulation errors, (2) a framework that avoids the introduction of these errors, and (3) an open-source prototype in Unreal Engine that performs such measurements and demonstrates said framework. The greater implications of this research is an increased awareness of the addressed concerns and the availability of a solution that resolves them. Going forward, one could further validate the presented results and potentially commercialize the existing prototype.

Bibliography

[1] Nigel Ang, Andrew Catling, Francesco Cifariello Ciardi, and Valentine Kozin. The technical art of sea of thieves. In *ACM SIGGRAPH 2018 Talks*, SIGGRAPH '18, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Arm. Avoid calls that stall the graphics pipeline, 2021. `https://developer.arm.com/documentation/dui0555/a/optimization-checklist/avoid-calls-that-stall-the-graphics-pipeline`, last accessed on 2022-01-26.

[3] Kirill Bazhenov. Gpu driven occlusion culling in life is feudal, 2017-02-01. `https://bazhenovc.github.io/blog/post/gpu-driven-occlusion-culling-slides-lif/`, last accessed on 2022-01-26.

[4] Jan Bender and Dan Koschier. Divergence-free smoothed particle hydrodynamics. In *Proceedings of the 2015 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 2015.

[5] Blender Foundation. Blender, 2022. `https://www.blender.org/`, last accessed on 2022-03-16.

[6] Jérémy Bouny. Screen space projected grid with glsl and webgl, 2015-05-10. `https://jeremybouny.fr/en/articles/screen_space_grid/`, last accessed on 2022-05-09.

[7] Robert Bridson and Matthias Müller. Fluid simulation: Siggraph 2007 course notes video files associated with this course are available from the citation page. pages 1–81, 08 2007.

[8] Eric Bruneton, Fabrice Neyret, and Nicolas Holzschuch. Real-time Realistic Ocean Lighting using Seamless Transitions from Geometry to BRDF. *Computer Graphics Forum*, 29(2):487–496, May 2010. EUROGRAPHICS 2010 (full paper) - Session Rendering I.

[9] Dennis B. Creamer, Frank Henyey, Roy Schult, and Jon Wright. Improved linear representation of ocean surface waves. *Journal of Fluid Mechanics*, 205:135–161, 1989.

[10] Eugene d'Eon and Geoffrey Irving. A quantized-diffusion model for rendering

translucent materials. *ACM transactions on graphics (TOG)*, 30(4):1–14, 2011.

[11] Jonathan Dupuy and Eric Bruneton. Real-time animation and rendering of ocean whitecaps. In *SIGGRAPH Asia 2012 Technical Briefs*, pages 1–3. Association for Computing Machinery, 2012.

[12] Epic Games. Unreal engine 4.26 documentation: Rendering dependency graph. https://docs.unrealengine.com/4.26/en-US/render-dependency-graph-in-unreal-engine/, last accessed on 2022-04-27.

[13] Epic Games. Unreal engine 5.0 documentation: Render dependency graph. https://docs.unrealengine.com/5.0/en-US/render-dependency-graph-in-unreal-engine/, last accessed on 2022-04-27.

[14] Epic Games. Unrealengine/engine/source/runtime/rhi/private/rhicommandlist.cpp. https://github.com/EpicGames/UnrealEngine/blob/99b6e203a15d04fc7bbbf554c421a985c1ccb8f1/Engine/Source/Runtime/RHI/Private/RHICommandList.cpp#L2698, last accessed on 2022-04-27.

[15] Epic Games. Unrealengine/engine/source/runtime/rhi/private/rhicommandlist.cpp. https://github.com/EpicGames/UnrealEngine/blob/46544fa5e0aa9e6740c19b44b0628b72e7bbd5ce/Engine/Source/Runtime/RHI/Private/RHICommandList.cpp#L2705, last accessed on 2022-04-27.

[16] Epic Games. Unrealengine/engine/source/runtime/vulkanrhi/private/vulkanrendertarget.cpp. https://github.com/EpicGames/UnrealEngine/blob/99b6e203a15d04fc7bbbf554c421a985c1ccb8f1/Engine/Source/Runtime/VulkanRHI/Private/VulkanRenderTarget.cpp#L554, last accessed on 2022-04-27.

[17] Epic Games. Unrealengine/engine/source/runtime/vulkanrhi/private/vulkanrendertarget.cpp. https://github.com/EpicGames/UnrealEngine/blob/46544fa5e0aa9e6740c19b44b0628b72e7bbd5ce/Engine/Source/Runtime/VulkanRHI/Private/VulkanRenderTarget.cpp#L613, last accessed on 2022-04-27.

[18] Epic Games. Physics damping, 2022. https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Physics/Constraints/DampingAndFriction/, last accessed on 2022-03-17.

[19] Epic Games. Unreal engine, 2022. https://www.unrealengine.com/, last accessed on 2022-03-16.

[20] Fynn-Jorin Flügge. Realtime gpgpu fft ocean water simulation. Technical report, Eingebettete Systeme E-13, 2017.

[21] Alain Fournier and William T Reeves. A simple model of ocean waves. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 75–84, 1986.

[22] Matteo Frigo and Steven Johnson. Fftw: Fastest fourier transform in the west, 2022. https://www.fftw.org/, last accessed on 2022-03-14.

[23] Thomas Gamper. Ocean surface generation and rendering. Master's thesis, Wien, 2018.

[24] Nathan Gordon. The ocean: Wind waker graphics analysis series, 2017-01-23. https://medium.com/@gordonnl/the-ocean-170fdfd659f1, last accessed on 2022-05-09.

[25] Stefan Gustavson. Simplex noise demystified, 2005-03-22. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf, last accessed on 2022-02-11.

[26] Scharon Harding. What is pcie? a basic definition, 2021-02-08. https://www.tomshardware.com/reviews/pcie-definition,5754.html, last accessed on 2022-01-26.

[27] Shawn Hargreaves. Stalling the pipeline, 2008. https://shawnhargreaves.com/blog/stalling-the-pipeline.html, last accessed on 2022-03-08.

[28] Klaus Hasselmann, T. Barnett, E. Bouws, H. Carlson, D. Cartwright, K Enke, J Ewing, H Gienapp, D. Hasselmann, P. Kruseman, A Meerburg, Peter Muller, Dirk Olbers, K Richter, W. Sell, and H. Walden. Measurements of wind-wave growth and swell decay during the joint north sea wave project (jonswap). *Deut. Hydrogr. Z.*, 8:1–95, 01 1973.

[29] Herman Johannes Haverkort. *Results on geometric networks and data structures*. PhD thesis, Utrecht University, 2004.

[30] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. Interactive animation of ocean waves. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, page 161–166, New York, NY, USA, 2002. Association for Computing Machinery.

[31] Christopher J. Horvath. Empirical directional wave spectra for computer graphics. In *Proceedings of the 2015 Symposium on Digital Production*, DigiPro '15, page 29–39, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Markus Ihmsen, Nadir Akinci, Gizem Akinci, and Matthias Teschner. Unified spray, foam and air bubbles for particle-based fluids. *The Visual Computer*, 28:669–677, 2012-06.

[33] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. Sph fluids in computer graphics. In *EUROGRAPHICS 2014/S. LEFEBVRE AND M. SPAGNUOLO*. Citeseer, 2014.

[34] Ivory Tower. The crew 2, 2018. https://www.ubisoft.com/en-gb/game/the-crew/the-crew-2, last accessed on 2022-01-26.

[35] J J O'Connor and E F Robertson. Jean baptiste joseph fourier, 1997. https://mathshistory.st-andrews.ac.uk/Biographies/Fourier/, last accessed on 2022-02-28.

[36] Jasper. How scrolling textures gave super mario galaxy 2 its charm, 2019. https://youtu.be/8rCRsOLiO7k, last accessed on 2022-03-17.

[37] Henrik Wann Jensen, Stephen R Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, 2001.

[38] Stefan Jeschke, Tomáš Skřivan, Matthias Müller-Fischer, Nuttapong Chentanez, Miles Macklin, and Chris Wojtan. Water surface wavelets. *ACM Trans. Graph.*, 37(4), jul 2018.

[39] Stefan Jeschke and Chris Wojtan. Water wave packets. *ACM Trans. Graph.*, 36(4), jul 2017.

[40] Claes Johanson. Real-time water rendering: Introducing the projected grid concept. Master's thesis, Lund University, 2004.

[41] Alex Vlachos John Isidoro and Chris Brennan. Rendering ocean water. In Wolfgang F. Engel, editor, *Direct3D ShaderX Vertex and Pixel Shader Tips and Tricks*, pages 347–356. Wordware Publishing, Inc., Plano, Texas, 2002.

[42] Baldur Karlsson. Renderdoc, 2018. https://renderdoc.org/, last accessed on 2022-05-19.

[43] Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. *University of Copenhagen: Department of Computer Science*, 2, 2006.

[44] Jacques Kerner. Water interaction model for boats in video games, 2015-02-27. https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games, last accessed on 2022-01-26.

[45] Jacques Kerner. Water interaction model for boats in video games: Part 2, 2016-01-08. https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games-part-2, last accessed on 2022-01-26.

[46] Gerasimos Kougianos and Konstantinos Moustakas. Large-scale ray traced water caustics in real-time using cascaded caustic maps. *Computers & Graphics*, 98:255–267, 2021.

[47] Harald Krogstad and Øivind Arnsten. Linear wave theory: Part a, 2000. https://folk.ntnu.no/oivarn/hercules_ntnu/LWTcourse/lwt_new_2000_Part_A.pdf, last accessed on 2022-03-11.

[48] Fredrik Larsson. Deterministic ocean waves. Master's thesis, Lund University Institute of Technology, LTH, 2012.

[49] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169,

aug 1987.

[50] Rick Lyons. Four ways to compute an inverse fft using the forward fft algorithm, 2015-07-07. https://www.dsprelated.com/showarticle/800.php, last accessed on 2022-03-15.

[51] Benoit B. Mandelbrot and John W. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM Review*, 10(4):422–437, 1968.

[52] Microsoft. Microsoft flight simulator, 2020. https://www.flightsimulator.com/, last accessed on 2022-01-26.

[53] Microsoft. Resource management best practices, 2020-08-19. https://docs.microsoft.com/en-us/windows/win32/dxtecharts/resource-management-best-practices, last accessed on 2022-03-08.

[54] Microsoft. Copying and accessing resource data (direct3d 10), 2021-08-19. https://docs.microsoft.com/en-us/windows/win32/direct3d10/d3d10-graphics-programming-guide-resources-mapping?redirectedfrom=MSDN#Accessing, last accessed on 2022-01-26.

[55] NavyFish. Asynchronously getting data from the gpu (directx 11 with rendertexture or computebuffer), 2015. https://forum.unity.com/threads/asynchronously-getting-data-from-the-gpu-directx-11-with-rendertexture-or-computebuffer.281346/, last accessed on 2022-01-26.

[56] NVIDIA. Chapter 28. graphics pipeline performance, 2007. https://developer.nvidia.com/gpugems/gpugems/part-v-performance-and-practicalities/chapter-28-graphics-pipeline-performance, last accessed on 2022-01-26.

[57] NVIDIA. NVIDIA WaveWorks, 2019. https://developer.nvidia.com/waveworks, last accessed on 2021-10-29.

[58] NVIDIA. cufft, 2022. https://docs.nvidia.com/cuda/cufft/index.html, last accessed on 2022-03-14.

[59] NVIDIA. Rigid body dynamics: Nvidia physx sdk, 2022. https://documentation.help/NVIDIA-PhysX-SDK-Guide/RigidDynamics.html#damping, last accessed on 2022-03-17.

[60] NVIDIA and Grapeshot Games. Wakes, explosions and lighting: Interactive water simulation in atlas, 2019. https://www.youtube.com/watch?v=Dqld965-Vv0, last accessed on 2021-10-29.

[61] Lisa Overing. Maritime simulation & training: a partnership that pays off, 2019-07-03. https://www.marinelink.com/news/maritime-simulation-training-a-468007, last accessed on 2022-03-22.

[62] Ivan Pensionerov. gasgiant/fft-ocean, 2020. https://github.com/gasgiant/

FFT-Ocean, last accessed on 2022-01-26.

[63] Kenneth Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 287–296. Association for Computing Machinery, 1985.

[64] Kenneth Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, 2002.

[65] Kenneth Perlin. Noise hardware, 2002. https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf, last accessed on 2022-02-11.

[66] O. M. Phillips. On the generation of waves by turbulent wind. *Journal of Fluid Mechanics*, 2(5):417–445, 1957.

[67] Anna Puig-Centelles, Francisco Ramos, Oscar Ripolles, Miguel Chover, and Mateu Sbert. View-dependent tessellation and simulation of ocean surfaces. *The Scientific World Journal*, 2014, 2014.

[68] RasterGrid Kft. Simd in the gpu world, 2022-02-04. https://www.rastergrid.com/blog/gpu-tech/2022/02/simd-in-the-gpu-world/, last accessed on 2022-04-12.

[69] Jens Schneider and Rüdiger Westermann. Towards real-time visual simulation of water surfaces. In *VMV*, volume 1, pages 211–218. Citeseer, 2001.

[70] Mike Seymour. Assassin's creed iii: The tech behind (or beneath) the action, 2012-12-11. https://www.fxguide.com/fxfeatured/assassins-creed-iii-the-tech-behind-or-beneath-the-action/, last accessed on 2021-11-26.

[71] Slightly Mad Studios. Project cars, 2020. https://www.projectcarsgame.com/, last accessed on 2022-01-26.

[72] Winthrop W. Smith and Joanne M. Smith. *Handbook of real-time fast Fourier transforms : algorithms to product testing*. New York : IEEE Press, 1995.

[73] Jeff Springer. What is unified memory and how does it work on apple silicon?, 2021-06-02. https://www.xda-developers.com/apple-silicon-unified-memory/, last accessed on 2022-03-08.

[74] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, 1999.

[75] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.

[76] Robert H. Stewart. *Introduction to Physical Oceanography*. University Press of Florida, 2008.

[77] Fabio Suriano. An introduction to realistic ocean rendering through fft - fabio suriano - codemotion rome 2017, 2017. https://www.slideshare.net/Codemotion/an-introduction-to-realistic-ocean-rendering-through-fft-f

abio-suriano-codemotion-rome-2017, last accessed on 2022-01-26.

[78] Tony Tamasi. Why does high fps matter for esports?, 2019-12-03. https://www.nvidia.com/en-us/geforce/news/what-is-fps-and-how-it-helps-you-win-games/, last accessed on 2022-05-11.

[79] Tim Tcheblokov. Ocean simulation and rendering in war thunder, 2015. https://developer.download.nvidia.com/assets/gameworks/downloads/regular/events/cgdc15/CGDC2015_ocean_simulation_en.pdf, last accessed on 2021-11-26.

[80] Alexandra Techet. Free-surface waves, 2005. http://web.mit.edu/2.016/www/handouts/Free-Surface-Waves.pdf, last accessed on 2022-03-11.

[81] Jerry Tessendorf. Simulating ocean water. *Simulating nature: realistic and interactive techniques. SIGGRAPH*, 1(2):5, 2001.

[82] Jerry Tessendorf. Interactive water surfaces. *Game Programming Gems 4*, 2004. http://jtessen.people.clemson.edu/papers_files/Interactive_Water_Surfaces.pdf, last accessed on 2021-10-29.

[83] Jerry Tessendorf. Simulation of interactive surface waves, 2008-11-20. http://jtessen.people.clemson.edu/papers_files/SimInterSurfWaves.pdf, last accessed on 2022-03-10.

[84] Jerry Tessendorf. ewave: Using an exponential solver on the iwave problem. *Technical Note*, 2014.

[85] Jerry Tessendorf. Gilligan : A prototype framework for simulating and rendering maritime environments, 2017.

[86] The_Fiddler, Xmas, MalcolmB, Alfonse_Reinheart (Jason L. McKesson). Why is gpu-cpu transfer slow?, 2009. https://community.khronos.org/t/why-is-gpu-cpu-transfer-slow/58708/3, last accessed on 2022-01-26.

[87] Dmitrii Tolmachev. Vkfft: Performant, cross-platform and open-source gpu fft library using vulkan api, 2021-04-13. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31300/, last accessed on 2022-05-13.

[88] Dmitrii Tolmachev. Dtolm/vkfft, 2022-04-11. https://github.com/DTolm/VkFFT, last accessed on 2022-05-13.

[89] Unity Technologies. Rigidbody, 2022. https://docs.unity3d.com/ScriptReference/Rigidbody.html, last accessed on 2022-03-17.

[90] Unity Technologies. Unity-technologies/boatattack, 2022. https://github.com/Unity-Technologies/BoatAttack, last accessed on 2022-01-26.

[91] Unity Technologies. Asyncgpureadback, 2022-01-22. https://docs.unity3d.com/ScriptReference/Rendering.AsyncGPUReadback.html, last accessed on 2022-01-26.

[92] Unity Technologies. Asyncgpureadbackrequest, 2022-02-25. https://docs.unity3d.com/ScriptReference/Rendering.AsyncGPUReadbackRequest.html, last accessed on 2022-03-08.

[93] Unreal Engine. Physics-based simulations & effects | live training | unreal engine, 2015-09-03. https://youtu.be/-lFKZRX1K_0?t=185, last accessed on 2022-01-26.

[94] Valve. Steam hardware & software survey: February 2022, 2022-04. https://store.steampowered.com/hwsurvey/videocard/, last accessed on 2022-03-08.

[95] Eric Weisstein. Plancherel's theorem. https://mathworld.wolfram.com/PlancherelsTheorem.html, last accessed on 2022-03-07.

[96] Cem Yuksel, Donald H. House, and John Keyser. Wave particles. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, page 99–es, New York, NY, USA, 2007. Association for Computing Machinery.