



EÖTVÖS LORÁND UNIVERSITY

COMPUTER SCIENCE

ALGORITHMS AND DATA STRUCTURES

Ocean Rendering with Fast Fourier Transform for Real-Time Applications

Supervisor:

Csaba Bálint

Adjunct professor

Author:

Gergely Gugi

Computer science BSc student

Budapest, 2024

Contents

1	Abstract	3
2	Introduction	4
3	Simulation	6
3.1	Oceanography	6
3.1.1	Gerstner waves	6
3.1.2	Analytical description of waves	7
3.2	Function series and sums	9
3.2.1	Fourier series	9
3.3	Fourier transformation	11
3.3.1	Discrete Fourier transform	12
3.3.2	2-Dimensional Fast Fourier transform	12
3.4	Implementation	14
3.4.1	Data representation	14
3.4.2	Pipeline	15
3.5	Level of Detail	17
4	Rendering	19
4.1	Method 1: Using regular geometry	19
4.1.1	Quadtree	20
4.1.2	Tessellation	22
4.1.3	Gap issue	23
4.2	Method 2: Per-pixel displacement mapping	24
4.2.1	Step size optimization	25
4.2.2	Cone mapping	26
4.3	Shading	27
5	Results	29

6 Conclusion	31
6.1 Future work	32
6.1.1 Axis aligned min max boxes for rendering	32
A Appendix	34
Bibliography	39
List of Figures	41
List of Algorithms	43

Chapter 1

Abstract

In this paper, we present findings on simulating and rendering ocean surfaces using modern graphics APIs. The simulation is based on Jerry Tessendorf’s model, adapted for efficient parallel execution on the GPU. It employs analytical approximations of ocean wave spectra, generating a realistic ocean surface through Fourier transformations. For rendering, we compare two approaches: one relies on patch tessellation and world partitioning to produce high-resolution geometry, while the other utilizes per-pixel displacement mapping to mimic fine details. We demonstrate a technique with a preprocessing cone mapping step that mitigates artifacts inherent to the latter algorithm. We discuss the efficiency and visual clarity trade-offs for each method, with particular attention to scenarios where one approach outperforms the other. Additionally, we address the tiling artifacts caused by the simulation stage by combining multiple simulations with differing areas, creating a more dynamic surface. The implementation is written in C++20 with the DirectX 12 backend, enabling efficient rendering with GPU compute and graphics shaders.

Chapter 2

Introduction

Ocean simulation is an important component in the field of computer graphics, encompassing applications from virtual simulation to cinematic visual effects. Near the turn of the millennium, significant progress was made in describing dynamic and visually convincing water surfaces. A foundational contribution to future research in this field is Jerry Tessendorf’s work [1], which remains the cornerstone for simulating vast water bodies. His usage of spectral wave models and ocean surface dynamics has set a high standard for subsequent advancements in this domain.

Building on this essential work, as well as more recent examples [2, 3], this paper explores modern real-time ocean simulation and rendering techniques. It aims to balance visual fidelity and computational efficiency while maintaining the flexibility of artistic creativity to create differing conditions. Using the DirectX 12 backend, combining the Fourier transform with compute shaders, we leverage the parallel processing capabilities of the GPU to handle large-scale computations in real-time.

In addition to the simulation, we investigate approaches to rendering a height map. First, we present a space partitioning solution that employs a quadtree structure and GPU tessellation of nodes to generate highly detailed geometry for believable ocean surface visualization. We address the key challenges with discontinuities in the rasterized image by using the neighboring data constructable from the hierarchical structure of the quadtree. Furthermore, we demonstrate a per-pixel displacement mapping approach as an alternative method to the geometry-based rendering technique. By rendering a flat plane and employing ray marching in the pixel shader to compute intersections with the height map, this technique can create the most visually accurate depiction of the simulation results with minimal geometry. We aimed

to enhance this approach through cone mapping, seeking to improve performance. However, our results indicate that, in its current state, it is not a viable alternative to the traditional pipeline.

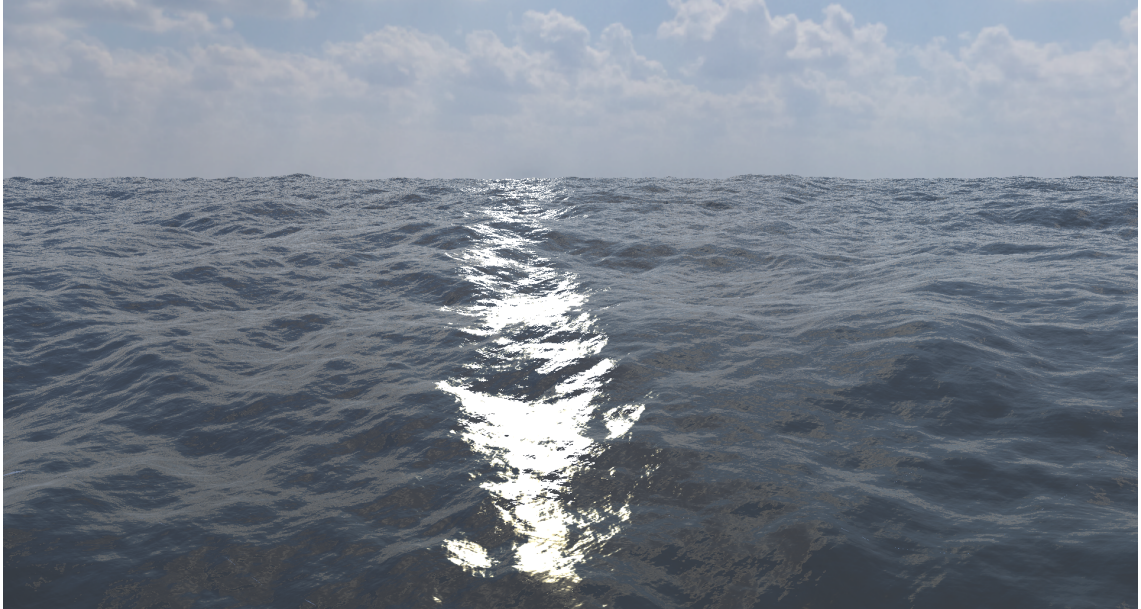


Figure 2.1: See simulation running in: <https://youtu.be/i8zhWRCqHss>

Chapter 3

Simulation

3.1 Oceanography

Oceanography is the study of the physical, geographical, chemical, and biological characteristics of the ocean. It is a multidisciplinary field that aims to understand the largest ecosystem on Earth and its effect on it. Specifically, physical oceanography focuses on describing and analyzing the properties and movement of energy through waves, tides, and currents.

Ocean waves may be categorized by their origin and behaviors, ranging from small wind-originated ripples to swells and even tsunamis. In order for researchers to analyze and predict the behavior of the sea, they had to break down the complex currents of the ocean into their smaller parts and model the spectrums of waves. This enables the study of the energy distribution and dominance across different wave frequencies and wavelengths.

Many models exist to describe the wave spectra of the ocean, such as the Pierson-Moskowitz and Phillips spectrums [4]. These models are the basis for modern applications such as ocean simulation and weather forecasts. Wave spectra can be derived from observations offering a statistical representation between waves, winds, and currents.

3.1.1 Gerstner waves

Wave analysis begins with a two-dimensional example of the effect of a single wave. To a good approximation, points of the surface (in this case line) follow a circular motion. Their movement can therefore be expressed as a parametric function

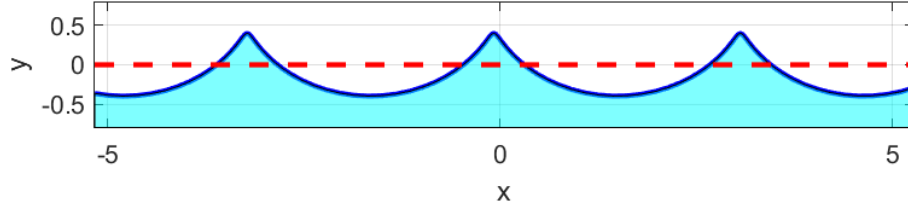


Figure 3.1: A single Gerstner wave in dark blue. The displaced line is marked in red.

of time [5]:

$$\begin{aligned} x &= x_0 - A \cdot \sin(k \cdot x_0 - \omega t + \phi) \\ y &= y_0 + A \cdot \cos(k \cdot x_0 - \omega t + \phi), \end{aligned} \quad (3.1)$$

where (x_0, y_0) is the starting point, A is the amplitude of the wave variable, and k, ω and ϕ describe the wave's magnitude, frequency, and phase accordingly. This function produces the Gerstner waves. This correspondence is trivially adaptable to three dimensions:

$$\begin{aligned} x &= x_0 - A(k_0/k) \cdot \sin(k_0 x_0 + k_1 z_0 - \omega t + \phi) \\ y &= y_0 - A(k_1/k) \cdot \sin(k_0 x_0 + k_1 z_0 - \omega t + \phi) \\ z &= z_0 + A \cdot \cos(k_0 x_0 + k_1 z_0 - \omega t + \phi), \end{aligned} \quad (3.2)$$

where $\mathbf{k} = (k_1, k_2) \in \mathbb{R}^2$ is the magnitude of the wave, and k_0 and k_1 describe its direction along the XY plane respectively. In deep water, the value of the frequency ω is equal to $\sqrt{gk}$, where g is the gravitational constant. The basic principle of ocean simulation is to calculate the effect of multiple sinusoidal-based Gerstner waves for each point and add them together to get the height at each point of the surface.

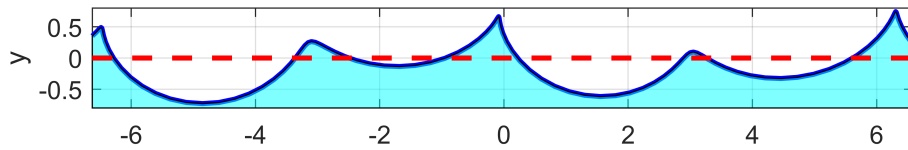


Figure 3.2: The result of adding together three Gerstner waves already creates an interesting surface. The wave is dark blue, the displaced line is marked in red.

3.1.2 Analytical description of waves

We based our simulation on previous work by Tessendorf [1], which describes an efficient method for ocean simulation and a deep insight into oceanography. In his work, the height field is expressed as the sum of many sinusoidal components. At any given point $\mathbf{x} = (x, y)$ on the 2-dimensional surface and an elapsed time t , the

height can be expressed as:

$$h(\mathbf{x}, t) := \sum_{\mathbf{k}_j} \tilde{h}(\mathbf{k}_j, t) e^{i\mathbf{k}_j^T \mathbf{x}}, \quad (3.3)$$

where $\mathbf{k}_j = (k_0, k_1) \in \mathbb{R}^2$ represent wave vectors on the XY plane. This complex-valued function representation simplifies the calculation of wave interactions described in equation 3.2, as it encapsulates the phase and frequency information of waves into a single function $\tilde{h}(\mathbf{k}, t) \in (\mathbb{R}^2 \times \mathbb{R}_0^+) \rightarrow \mathbb{C}$. This function can be computed as:

$$\tilde{h}(\mathbf{k}, t) := \tilde{h}_0(\mathbf{k}) e^{i\omega(\mathbf{k})t} + \tilde{h}_0(-\mathbf{k}) e^{-i\omega(\mathbf{k})t}, \quad (3.4)$$

where $\tilde{h}_0 \in \mathbb{R}^2 \rightarrow \mathbb{C}$ is the initial spectrum for all $\mathbf{k}$ waves. This function is capable of describing the characteristics of the wave and simulation parameters. We will employ the Phillips spectrum statistical model, which is particularly well-suited for calculating the starting spectrum of wind-driven waves in a fully developed ocean [6, 4].

$$P_h(\mathbf{k}) = A \frac{e^{(-1/(kL)^2)}}{k^4} \left| \frac{\mathbf{k}}{k} \cdot \frac{\mathbf{w}}{w} \right|^2, \quad (3.5)$$

where L is the length of the largest wave, derived from the wind vector $\mathbf{w}$ as $L := w^2/g$, and A is a global scaling constant. To initialize the simulation, we create a realization of this statistical model using two random numbers ξ_r, ξ_i from the Gaussian normal distribution.

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P_h(\mathbf{k})}. \quad (3.6)$$

This statistical model provides a robust and versatile simulation while offering flexibility for artistic expression through adjustable parameters. By fine-tuning these parameters, it is possible to control the dynamics of the waves and the characteristics of the resulting surface. For a more comprehensive explanation of the principles behind these choices, please refer to the original work [1].

With the mathematical foundation established, the next step is to discretize the model for GPU implementation. The equation 3.3 will be computed using fast Fourier transforms for efficient summation of wave components.

3.2 Function series and sums

Let (f_n) be a function series if, $\forall n \in \mathbb{N} : f_n : \mathbb{R} \rightarrow \mathbb{R}$. Its partial sum can be defined as the function: $F_n := \sum_{i=0}^n f_i$. The analysis of such series is often concerned with whether the f_n and F_n functions converge to a limit function uniformly, or point-wise as n goes to infinity.

For example, the trigonometric series is defined as:

$$\alpha_n \in \mathbb{R}(n \in \mathbb{N}), \beta_n \in \mathbb{N}(1 \leq n \in \mathbb{N})$$

$$f_0 \equiv \alpha_0$$

$$f_n(x) := \alpha_n \cos(nx) + \beta_n \sin(nx), (1 \leq n \in \mathbb{N})$$

$$F_n = \alpha_0 + \sum_{k=0}^n (\alpha_k \cos(kx) + \beta_k \sin(kx)), (1 \leq n \in \mathbb{N}).$$

This sum has some nice properties such as, it is infinitely derivable, it is a natural way to approximate periodic functions, and the set $\{1, \sin(nx), \cos(nx)\}$ forms a basis for space of integrable functions, and is therefore useful in functional analysis, and due to it being comprised of a trigonometric function, it is easy to work with to determine other facts about the series.

3.2.1 Fourier series

A specific type of trigonometric series are Fourier series. They are a method to dissect a periodic function into the sum of trigonometric series. For any Riemann integrable and periodic function there is an accompanying Fourier series representation, where the sum of the series (given the right conditions) converges to the original function.

Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a periodic function with period p . Let P be the representative period, usually $[-\frac{p}{2}, \frac{p}{2}]$ or $[0, p]$. Let f be Riemann integrable over this period: $f|_P \in R(P)$.

For example, consider a sawtooth function illustrated in Figure 3.3:

$$f(x) := \frac{x}{\pi} \quad (x \in [-\pi, \pi]),$$

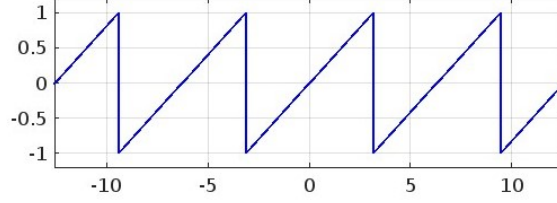


Figure 3.3: A sawtooth function from -4π to 4π

this function has 2π period and is implicitly $f : \mathbb{R} \rightarrow \mathbb{R}$ as

$$f(x + 2k\pi) = f(x) \quad (-\pi \leq x \leq \pi, k \in \mathbb{Z} \setminus \{0\}).$$

Then we can calculate the trigonometric coefficients, also known as the Fourier coefficients, for any f function as:

$$a_0 := \frac{2}{p} \int_P f(t) dt$$

$$a_n := \frac{2}{p} \int_P f(t) \cos\left(\frac{2\pi n}{p} t\right) dt \quad (1 \leq n \in \mathbb{N})$$

$$b_n := \frac{2}{p} \int_P f(t) \sin\left(\frac{2\pi n}{p} t\right) dt \quad (1 \leq n \in \mathbb{N}).$$

For simplicity, we use the angular period instead: $\omega = \frac{2\pi n}{p}$. Using this notation the Fourier sum of f can be written as:

$$s_n := \frac{a_0}{2} + \sum_{k=1}^n (a_k \cos(\omega k x) + b_k \sin(\omega k x)).$$

Given that the Dirichlet conditions are met, this sum is a perfect approximation as it converges to f [7]. Taking the 16th partial sum of the Fourier series for the sawtooth function is visible in Figure 3.4. Through the formula for complex numbers,

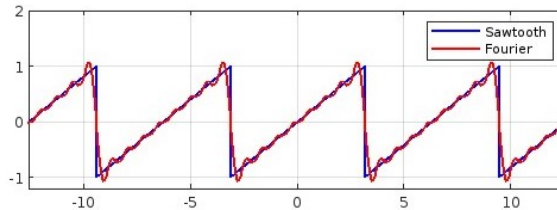


Figure 3.4: 16th partial Fourier series sum approximating the sawtooth function.

we can turn (a_n) and (b_n) series into a single series (c_n) .

$$e^{ix} = \cos(x) + i \sin(x) \iff \begin{cases} \cos(x) = \frac{e^{ix} + e^{-ix}}{2} \\ \sin(x) = \frac{e^{ix} - e^{-ix}}{2i} \end{cases}$$

$$s_n(x) = \frac{a_0}{2} + \sum_{k=1}^n \left(a_k \frac{e^{i\omega kx} + e^{-i\omega kx}}{2} + b_k \frac{e^{i\omega kx} - e^{-i\omega kx}}{2i} \right) =$$

$$= \frac{a_0}{2} + \sum_{k=1}^n \frac{a_k - ib_k}{2} e^{i\omega kx} + \sum_{k=1}^n \frac{a_k + ib_k}{2} e^{-i\omega kx} =$$

$$= \sum_{k=-n}^n c_k e^{i\omega kx},$$

as

$$c_n = \begin{cases} \frac{a_n - ib_n}{2}, & n > 0 \\ \frac{a_0}{2}, & n = 0 \\ \frac{a_n + ib_n}{2} = c_{-n}^*, & n < 0 \end{cases} \implies c_n = \frac{1}{p} \int_P f(t) \cdot e^{-i\omega n t} dt.$$

Where c_{-n}^* is the complex conjugate operator. However, this only works given that f is a periodic function over some interval P . This complex Fourier coefficient can also be applied for complex-valued functions.

3.3 Fourier transformation

The Fourier series is an excellent tool for decomposing periodic functions into their frequency components. However, many functions of interest are not periodic. To address this, we turn to the Fourier transform, which generalizes the Fourier series to handle non-periodic functions.

The previous (c_n) , $(n \in \mathbb{N})$ is converging from a series into a continuous function as $p \rightarrow \infty$. This $\hat{f}$ transform of $f : \mathbb{C} \rightarrow \mathbb{C}$, $f \in R$ is defined as:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) \cdot e^{-i2\pi x \xi} dx$$

This is a transformation from the spatial domain, often notated as x being the free-moving variable, to the frequency domain, notated as ξ being the variable of $\hat{f}$. This transformation enables the study of the function's periodicity and harmonic characteristics.

Fourier analysis does not stop here; it is a vast and intricate field [8]. However, for this paper, understanding the fundamentals provided above will suffice. We now shift our focus to applying this concept in a manner compatible with computational methods.

3.3.1 Discrete Fourier transform

Consider a sequence of N sampled values $(f_0, f_1, \dots, f_{N-1})$ taken at uniform intervals from a function $f \in \mathbb{C} \rightarrow \mathbb{C}$. Let the uniform sampling interval be Δx , and the sampled positions be notated as x_n , where $n = 0, 1, \dots, N-1$. The DFT of this sequence is defined as:

$$X_n = \sum_{k=0}^{N-1} f_k \cdot e^{-i2\pi \frac{k}{N}n},$$

Where $X_n \in \mathbb{C}$ now represents discrete frequencies. These frequencies correspond to harmonics of the underlying spatial domain, allowing direct analysis of dominant wavelengths and patterns. The inverse discrete Fourier transform reconstructs the original sampled data:

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi \frac{k}{N}n}$$

See the transformation for sine and cosine in Figure A.4, Figure A.5.

3.3.2 2-Dimensional Fast Fourier transform

Fast Fourier transform is a computationally efficient algorithm that calculates the inverse¹ discrete Fourier transform. James W. Cooley and John W. Tukey's method leverages the divide-and-conquer paradigm to reduce the complexity from $\mathcal{O}(N^2)$ to two $\mathcal{O}(N \log_2 N)$ FFT passes per element of the two-dimensional input.

The definition of a two-dimensional discrete Fourier transform is the following:

$$f_{j_1, j_0} = \frac{1}{N^2} \sum_{k_0=0}^{N-1} \sum_{k_1=0}^{N-1} X_{k_1, k_0} \cdot W_N^{k_0 \cdot k_1},$$

where W_N is the N th complex root of unity $W_N := e^{i \frac{2\pi}{N}}$, X is the input and f is the output, both two dimensional arrays.

¹The same method can be applied to the discrete Fourier transform without change in the approach. There W_N is equal to: $e^{-i \frac{2\pi}{N}}$

The first observation is that it is possible to break down this DFT into two stages:

1. Row-wise DFT: transform each row of the texture independently.
2. Column-wise DFT: apply the same method for each column of the intermediate result.

The intermediary step is:

$$A_{j_0, k_0} = \sum_{k_1=0}^{N-1} X_{k_1, k_0} \cdot W_N^{k_1 \cdot k_0}.$$

To simplify the explanation let's assume that N is a highly composite number, specifically a power of two. The original authors have shown how it works otherwise, but they also outline why the performance is best in this scenario.

The authors have observed that there is still an overlap between the calculations in each row. Using the periodicity of $x \mapsto W_N^x$ leads to DFTs of size $N/2$:

$$A_{j_0, k_0} = \underbrace{\sum_{m=0}^{N/2-1} X_{2m, k_0} \cdot W_{N/2}^{m \cdot k_0}}_{X_{even, k_0}} + W_N^{k_0} \underbrace{\sum_{m=0}^{N/2-1} X_{2m+1, k_0} \cdot W_{N/2}^{m \cdot k_0}}_{X_{odd, k_0}}.$$

Then using the symmetry of W_N : $W_N^{x+N/2} = -W_N^x$, it is possible to further reduce the required calculations:

$$\begin{aligned} A_{j_0, k_0} &= X_{even, k_0} + W_N^{k_0} X_{odd, k_0}, \quad (k_0 < N/2) \\ A_{j_0, k_0+N/2} &= X_{even, k_0} - W_N^{k_0} X_{odd, k_0}, \quad (k_0 < N/2) \end{aligned},$$

where $W_N^{k_0}$ is called the twiddle factor. This operation is called the butterfly operation, due to the way it is usually visualized.

Performing this step recursively to calculate the even and odd variants of X , leads to the final FFT algorithm. The shown algorithm 1 is the efficient parallelized code, which utilizes bit manipulation of the indexes to preemptively place data close and to linearize the recursive algorithm.

Algorithm 1 FFT parallelized implementation for rows of inputTexture.

Input: groupId row, threadID column.

Output: The result of the FFT at position (groupId, threadID).

```

1:  $a := \text{reverseBits}_{\log_2(N)}(\text{threadID})$  ▷ Reverse significant bits
2:  $\text{buffer}[0][a] := \text{inputTexture}[(\text{groupId}, \text{threadID})]$ 
3:  $\text{state} := 0$  ▷ Indicates the current buffer index
4: for  $s := 1$  to  $\log_2(N)$  do
5:    $m := 2^s$  ▷ Height of butterfly group
6:    $mh := 2^{s-1}$ 
7:    $k := (\text{threadID} \cdot (N/m)) \bmod N$ 
8:    $i := \text{threadID} - (\text{threadID} \bmod m)$  ▷ Starting row of butterfly group
9:    $j := \text{threadID} \bmod mh$  ▷ Butterfly index in group
10:   $\theta := \frac{2\pi \cdot k}{N}$ 
11:   $W_{N,k} := (\cos(\theta), \sin(\theta))$  ▷ Complex number
12:   $\text{GroupMemoryBarrier}()$  ▷ Ensure all threads have finished writing
13:   $\text{input1} := \text{buffer}[\text{state}][i + j + mh]$ 
14:   $\text{input2} := \text{buffer}[\text{state}][i + j]$ 
15:   $\text{state} := 1 - \text{state}$  ▷ Flip state
16:   $\text{buffer}[\text{state}][\text{threadID}] := \text{input2} + W_{N,k} \cdot \text{input1}$ 
17: end for
18:  $\text{resultTexture}[(\text{groupId}, \text{threadID})] := \text{buffer}[\text{state}][\text{threadID}]$ 

```

3.4 Implementation

Armed with the understanding of oceanography and the methodology of how the equations will be calculated, we must translate the theory outlined in Section 3.1 into a discrete model suitable for execution on the highly parallelized GPU.

3.4.1 Data representation

The inverse FFT calculations require a uniformly distributed set of points containing the Fourier coefficients in equation 3.4 as a texture. To discretize the abstract description of the ocean, we have to define parameters,

- N the texture size. This matches the previous notation, as $N \times N$ textures have $N \times N$ data points to sum over. For performance reasons textures are square, and N is a power two since FFTs are proven to work best over them. This property is also inherent to how computers work on datasets.
- L , the extent of the calculated ocean patch contained in the texture. This is uniform in both directions for the same reason as above.

- The parameters defined in the Phillips spectrum definition 3.5: $\mathbf{w}$ wind, A the global scalar.

Our textures up until the FFT will contain wave spectrum information, where the coordinate of the texel in correlation to the center is the vector of which wave's data it holds. To prevent precision errors, for each texture the highest layouts are used that fit in 128 bits per texel. That is, 2 component textures are 32-bit per component, and 4 component textures are 16-bit per. All illustrations will be using these texture definitions.

3.4.2 Pipeline

The overview of the pipeline is visible in figure A.1

Initial spectrum

The initial spectrum of the waves has to be calculated at startup or when changes occur to the simulation parameters. Therefore these are calculated on the CPU using an XorShifter as a fast pseudo-random number generator. Along with the spectrum, the corresponding frequencies are uploaded as a separate single-component 32-bit texture.

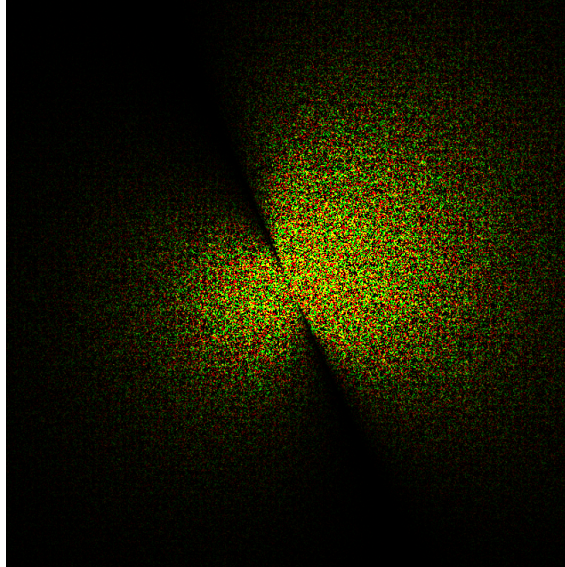


Figure 3.5: Initial starting spectrum, generated with a $(+0.9,+0.4)$ XZ wind vector.

Specific time spectrum

From now on, all tasks must be run in each frame for the simulation to feel fluid and lifelike as their outputs are time-dependent. These tasks are therefore run on the GPU as compute shaders, which allow us to operate on each texel in parallel.

In this task, we calculate the spectrum $\tilde{h}$ for each wave (texel). While this might initially seem like a straightforward step, it's essential to focus on optimizing all processes for better performance. Specifically, this step requires loading the initial spectrum for both $\mathbf{k}$ and $-\mathbf{k}$, which are symmetric wave components.

To remove an unnecessary texture read, we can modify the arrangement in which the dispatched threads correspond to the texture coordinates. By reordering the threads so that symmetrical texels (those corresponding to $\mathbf{k}$ and $-\mathbf{k}$) are grouped, we can take advantage of group-shared memory, to cache the value of the texture in faster memory.

Fast Fourier Transform

The fast Fourier transform computation is executed using the previously shown algorithm 1. The method leverages shared memory for each row processing of the texture, which must fit into the group-shared memory blocks of the GPU. DirectX 12 compatible GPUs are guaranteed to provide 32KB of group-shared memory. As a result, this method can only support up to 1024x1024 textures.

The algorithm is defined to apply the Fourier transform to the row of the input texture. To correctly apply the Fourier transform to the rows and then columns, instead of creating a second compute shader, we modify the existing algorithm to output a transposed result. This output is then transformed, which applies the transform to the columns of the original textures, and transposes the data back to normal orientation.

Gradients

Tessendorf's approach [1] to calculating surface gradients involves performing additional FFTs to compute the normal map. While this method achieves the highest possible visual fidelity, we chose the finite difference method. This approach calculates gradients by taking the differences between adjacent heightmap values and computing their cross-product. The finite difference method is computationally more

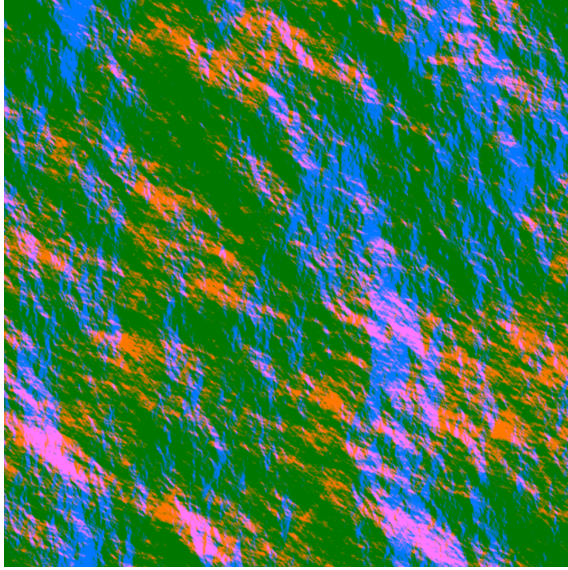


Figure 3.6: Gradients texture generated via the finite differential algorithm, the red, green, and blue components correspond to the XYZ axes respectively.

efficient than FFT-based calculations and provides sufficient accuracy for real-time applications.

Foam

An additional advantage of the finite difference method is its ability to approximate the derivative of the surface directly from the neighboring heightmap differences. The determinant can be calculated by constructing the Jacobian matrix from these derivatives. When the determinant becomes negative, the matrix is no longer invertible, suggesting that the surface has undergone self-intersection. This condition often corresponds to phenomena such as ripples or wave breaking.

To simulate the presence of foam, we introduce an exponential decay method to ensure that foam appears, persists on the surface, and dissipates over time. The determinant can be stored alongside the gradient in a single texture as a four-component vector to compact the data in memory.

3.5 Level of Detail

The simulation is constrained by the finite size of the textures that store the per-vertex representation of the surface. Additionally, performing an FFT on these textures requires compatibility with the compute shader, which imposes limitations

on the texture extent. Nevertheless, increasing the memory capacity alone does not address the challenge of simulating larger surfaces.

To visualize a vast ocean surface, the generated displacement map has to be repeated horizontally. This creates a continuous surface due to the textures being periodic along their dimensions, a property resulting from the FFT. However, this tiling is easily observable in the scene, ruining the illusion of the whole simulation. Throughout previous papers [9, 2], the most adapted solution is to apply Perlin noise in the vertex shader. Multiple papers also mention an interpolation between the two constructions based on distance from the viewing point to achieve better fidelity close to it. Thankfully, due to the advancements mentioned earlier, GPUs now can run these simulations in real-time in a fraction of a second.

Therefore, this paper proposes a multi-scale solution, where the complete simulation pipeline is executed multiple times, each at a differing scale of influence. During rendering, special care must be given to ensure each level is sampled correctly. Due to using multiple textures with differing periods, vertexes have to use their location on the plane to calculate the texture coordinates for each texture. Importantly, the scales of the detail levels must not be simple fractions of one another to prevent reintroducing repetition to our surface. We found that three simulations at the following varying scales work best quality-wise.

- The lowest level covers the largest area at around 200-400 meters.
- The medium level covers around one-quarter to half the extent of the lowest.
- The highest level covers a much smaller area, around 10-30 meters.

The highest level is intentionally quite small. This is to counteract blocking artifacts, which occur when stretching a low-resolution texture on an object, or when a texture is applied to a huge object. By maintaining a high resolution-to-space ratio for the smallest waves, the finer details remain sharp.

Chapter 4

Rendering

Our ocean simulation has already made multiple simplifications that made the calculations more straightforward and doable in real-time but inevitably caused a physically less accurate result. Sadly, utilizing the displacement map proves to be a challenge as well.

4.1 Method 1: Using regular geometry

The intended method to display a height map in compliance with the regular graphics pipeline is to draw a subdivided plane and then displace the vertices in world space while rendering.

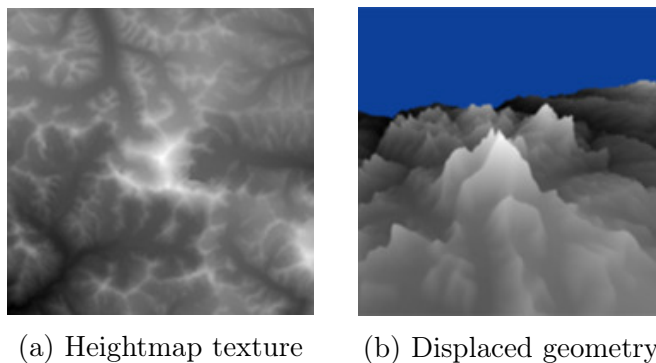


Figure 4.1: Illustration of a heightmap texture and displaced geometry. Height map generated by <https://manticorp.github.io/unrealheightmap>

This method would work for small puddles, but my goal is to create the illusion of an ocean by rendering a surface of multiple kilometers in size. To visualize a large surface we have to repeatedly render a small mesh to cover the whole ocean surface. This approach creates the problem that it is impossible to increase the level of detail

close to the camera where it matters, without forcing overdraw on triangles close to the far clipping plane. This is due to rendering with a uniform distribution of a statically defined mesh.

4.1.1 Quadtree

The solution to over- and underdraw involves dynamically subdividing the desired plane. A quadtree is an efficient hierarchical structure for this purpose, dividing a patch of the plane into four equal smaller patches. Each patch is then tested against a stop condition to determine whether further subdivision is necessary. This condition ensures that the distance from the center of each patch to the camera is proportional to the largest extent of the patch.

The power of the quadtree is even more pronounced when combined with frustum culling as an additional condition. This optimization enables the patches near the camera to be subdivided to a high level of detail, without wasting time on invisible patches outside the viewport.

Furthermore, significant performance improvements can be achieved by optimizing the traversal of the quadtree during the collection of the required rendering data of the nodes. Our method determines the best optimal order to explore the child of each node. Since the quadtree is guaranteed to be axis-aligned, and reside on the XY plane, this order can be determined by only the viewing direction of the camera. See the desired order for a given camera angle in Figure 4.3. This is a rough approximation, with two complications.

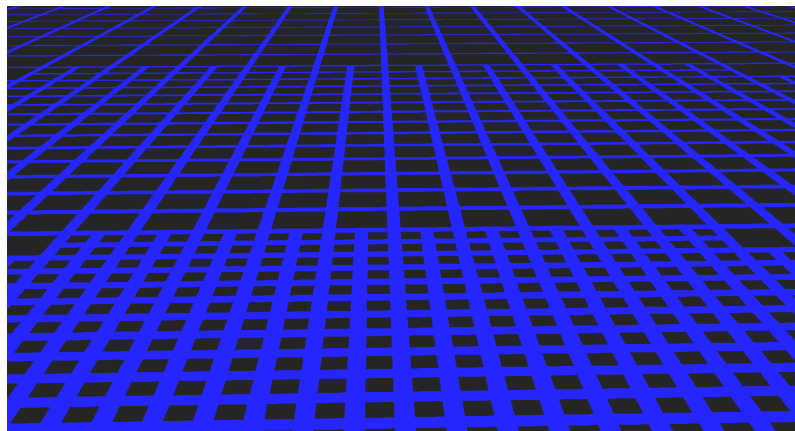


Figure 4.2: Quadtree visualization.

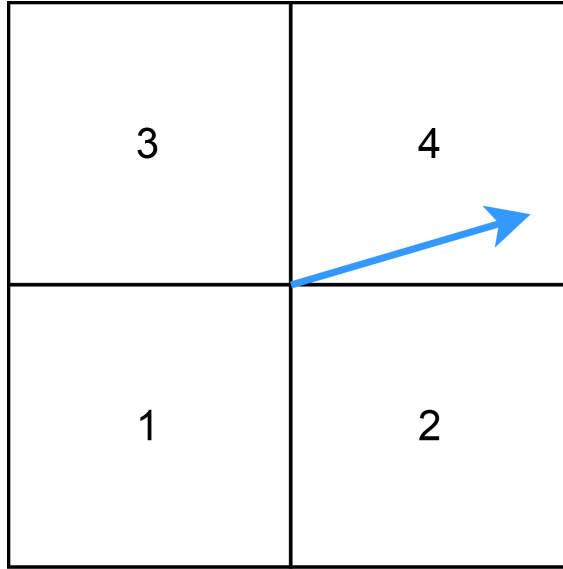


Figure 4.3: The order in which quadtree nodes children are explored given a camera angle in blue.

1. RAM access caching

Without a specialized traversal order, the quadtree can be explored sequentially only returning the child nodes, leveraging the underlying continuous memory layout of the data structure. This approach is highly efficient due to CPU memory prefetching. When accessing data from main memory, the CPU reads a predetermined cache-line size, which can result in cache hits for subsequent memory reads. In the naive method, this leads to an unexpected but significant performance boost. In the improved method, the index in the array has to jump ahead and backward, which results in non-sequential reads. This disrupts caching and causes a 5-10% increase in traversal time. Nonetheless, the reduction in overdraw more than compensates for this overhead, making our approach faster overall.

2. Position of the camera on the plane

This method does not account for the relative position of the camera to the plane. Therefore when the camera is near a corner of the plane looking outwards, the traversal will initially process incorrect nodes first. However, due to those nodes being frustum culled, the method will rapidly reach its optimal situation as seen in Figure 4.4.

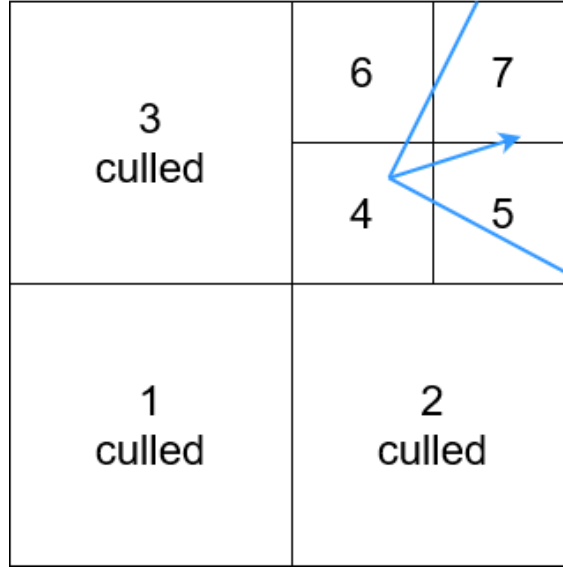


Figure 4.4: Visualizing how the method traverses the quadtree given a camera position, view, and frustum in blue.

4.1.2 Tessellation

The previous method adds detail by repeatedly rendering the same mesh at varying scales and positions to display the surface, using instance rendering across multiple draw calls. While effective, this method is highly sequential and thus calculated on the CPU. In contrast, tessellation increases detail by subdividing triangles - in this case quad patches - directly in the graphics pipeline on the GPU. This stage occurs in between the vertex and pixel shaders and involves the following steps:

- 1: Vertex Shader In this method, the vertex shader does not sample the height map. It only outputs the necessary variables for each vertex for future steps.
- 2: Hull Shader ¹ Using the world positions from the previous step, the hull shader decides the tessellation factor for the quad patch. Each edge is assigned a factor from 1 to 64, along with two inner factors for the interior of the quad.
- 3: Domain Shader ² Operating on the vertices generated based on the specification by the hull shader, the domain shader computes the world position of the new vertex using the UV coordinates. Using the interpolated texture coordinates, it samples the height map to displace the vertex appropriately.

¹In OpenGL, this is referred to as Tessellation Control.

²In OpenGL, this is referred to as Tessellation Evaluation.

4.1.3 Gap issue

The previous approaches occur at different stages of the rendering process, presenting an opportunity to integrate them to address their downsides.

- The quadtree approach can lead to an explosion in the number of instances that are required to render in high fidelity close to the camera, creating excessive traffic between the CPU and GPU.
- The tessellation approach suffers from the fact that the base mesh is repeated across the entire surface, forcing a trade-off between overdraw for distant patches or low precision close to the camera, as there is a limit to the tessellation factor.

This combination also resolves an unmentioned problem with the quadtree: gaps. Since neighboring patches of the quadtree may have different parent nodes and even different depths, their extent may be different. This leads to mismatched vertex counts along shared edges. When displaced in the domain shader, this causes visible discontinuities in the surface.

To address this, the tessellation pipeline can align the vertex count of the neighboring smaller patch along the shared edge. This ratio has to be precomputed and provided to the pipeline, as the tessellation stage is unaware of the quadtree structure. During the traversal of the quadtree, we can efficiently calculate these ratios, by backtracking at each child node.

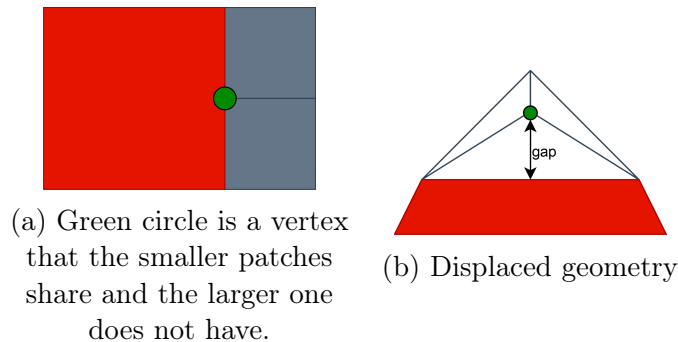


Figure 4.5: Visualizing how differing sized quadtree nodes can create gaps.

Algorithm 2 Determine relative size by traversing neighbors and children in a quad-tree

```

1: Input: tree: A Quadtree,
           directions: an array containing the neighboring indices and whether it
           shares parents. This variable depends on which neighbor we are checking,
           node: current node,
           path: the path to node, where -1 indicates the start of the tree.
2: Output A Boolean indicating whether the neighbor is smaller.
3: function ISSMALLER(directions, tree, path, node)
4:   buff := empty FIFO container
5:   for all elem in path in reverse do
6:     if elem = -1 then                                     ▷ Outside of tree, no neighbor
7:       return false
8:     end if
9:     neighborIndex, sharesParent := directions[elem]
10:    buff.push(neighborIndex)
11:    node := tree.GetAt(node.parent)
12:    if sharesParent then
13:      break
14:    end if
15:  end for
16:  for all elem in buff in reverse do
17:    node := tree.GetAt(node.children[elem])
18:    if not node.HasChildren() then
19:      return false
20:    end if
21:  end for
22:  return node.HasChildren()
23: end function

```

4.2 Method 2: Per-pixel displacement mapping

Rendering with the previous method produces high-fidelity images by generating unnecessarily detailed geometry, often resulting in overdraw. That approach requires thousands of vertices and triangles to approximate the ocean surface accurately. *Per pixel displacement mapping* offers an alternative to improve clarity with minimal geometry.

This method renders a large, flat plane and relies on ray marching in the pixel shader to determine the intersection of the viewing ray with the height map. The surface is then shaded according to the intersection point. When done correctly, the resulting surface will create the illusion of depth, mimicking the wavy surface of the ocean.

When properly tuned, this approach delivers the most visually accurate representation of the surface. If the minimum step size is too large, the algorithm risks overshooting an intersection point, resulting in visible artifacts such as gaps and inconsistencies in the surface. Conversely, reducing the step size or setting an exorbitantly large maximum travel distance may increase quality at the cost of increased computational cost.

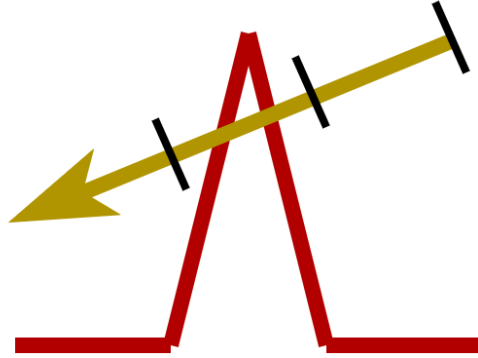


Figure 4.6: A naive ray marcher missing an intersection point in a fluctuating heightmap.

Artifacts

A notable limitation of this method becomes apparent at low viewing angles. In such cases, the method may fail to detect an intersection, defaulting to the last tested point within the maximum distance. This behavior produces visible artifacts, where certain pixels display incorrect shading. However, these mostly only happen over valleys in the height map and are therefore usually invisible. An illustration of this problem is shown in Figure A.3

4.2.1 Step size optimization

Given any point in the ray marching algorithm, a minimum step size can be declared as the minimum distance to reach new data holding texel in our heightmap. Any step smaller will result in redundant calculations, as no new data will be read. This value can be computed based on the current UV coordinates, viewing angle, and extent of the texture. [10]

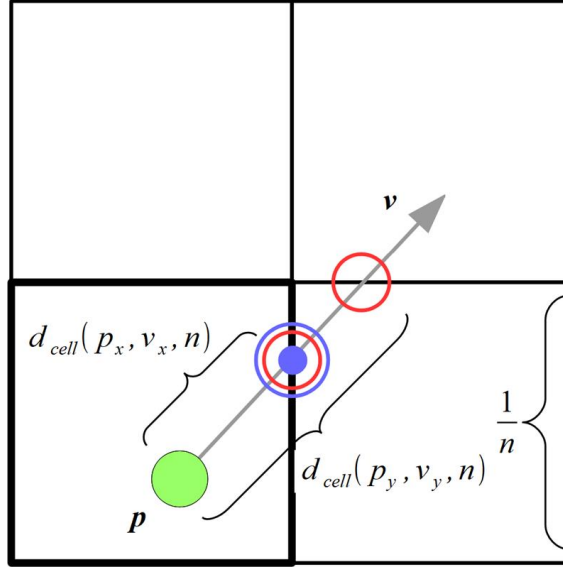


Figure 4.7: For a given point p , visualize the minimum distance to reach new data holding texel. Illustration taken from [10].

4.2.2 Cone mapping

Through preprocessing tasks, heuristics can be applied to improve the traversal of the height map. In this approach, each texel in the height map is assigned a cone, that does not intersect the height map. These cones guide the ray marcher process by ensuring that steps efficiently traverse the heightmap without missing an intersection.

Relaxed cone maps

Our chosen approach creates relaxed cone maps, where the cone is allowed to intersect the surface such that, within the cone, the height map remains level or increases. This guarantees that if our next step intersects the cone, we have not accidentally skipped over a sudden peak.

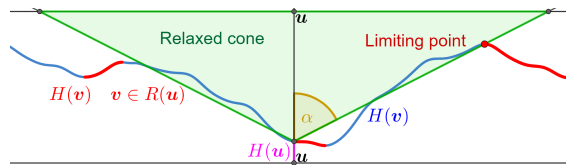


Figure 4.8: A 1D slice of the height map (blue curve) with the relaxed cone (green triangle). From the cone apex, the cone angle α is the largest such that the cone still excludes the red back-facing terrain. Illustration taken from [10].

The code for generating these textures and the illustration were adapted from [10]. Their approach also requires a min-max mipmap texture for rapid computation

of the cone map.

Calculating the intersection

The intersection of the cone with the viewing ray can be computed using the following equation. The cone is defined by its height h , and tangent $\tan \alpha$, while the ray is represented by a parametric equation $\mathbf{p} + t \cdot \mathbf{v}$, ($t > 0$). The step adjustment Δt can be computed as:

$$\Delta t = \frac{(\mathbf{p} \cdot \mathbf{y} + t \cdot \mathbf{v} \cdot \mathbf{y}) - h}{\sqrt{\mathbf{v} \cdot \mathbf{x}^2 + \mathbf{v} \cdot \mathbf{z}^2 + \tan \alpha}}. \quad (4.1)$$

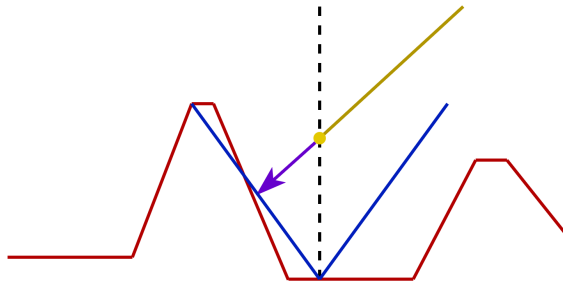


Figure 4.9: Ray intersecting a cone. The ray is colored yellow, the relaxed cone is blue, the height map is red, and the next step is purple.

4.3 Shading

To achieve high-quality realistic rendering, we implemented a physically-based Cook-Torrance shading model. This versatile approach provides realistic reflectance calculations for a wide range of materials with varying characteristics. By approximating microfacets - surface details smaller than the human eye can discern - it statistically models how the surface would interact with visible light. This method simulates multiple optical phenomena, including how these microfacets reflect light toward the camera, how they obscure and block each other, and the Fresnel effect, which dictates how the reflected intensity changes based on the incident angle. Water, being a highly reflective material, benefits significantly from this approach. This shader enhances specular highlights and contributes to the overall realism of our ocean simulation.

To synthesize the final image, we employ a deferred shading pipeline, this method divides the geometry and lighting calculations into distinct passes. This separation ensures that all geometric occlusion and visibility checks are completed before invoking the expensive shading process. This technique can significantly reduce redundant computations, making it ideal for scenes with complex lighting such as our model. However, this approach requires large texture buffers to store intermediary geometric data such as world position, normals, and material properties.

Chapter 5

Results

The implementation was done using the framework made by Péter Mayor¹. It provides a high-level abstraction over the DirectX 12 backend making GPU resource management automatic, although direct C API calls are still necessary. All measurements were taken on an Nvidia Geforce RTX 2080 super video card at FullHD resolution. Our source code is available at: <https://www.github.com/Reun85/WaterRendering>

Table 5.1 illustrates the execution times of each stage in the simulation pipeline measured in milliseconds, under the highest precision settings.

Settings	High	Medium
Texture size	1024x1024	512x512
Task	Execution time (μ s)	
$\tilde{h}$ computation	66,085	18,797
FFT	111,264	23,571
Gradient calculation	56,715	14,863
Min-Max mip-map	28,990	8,433
Cone map generation	190,660	50,699

Table 5.1: Performance of each simulation pipeline stage (EOP to EOP) measured in microseconds. Note, that the FFT calculations only show a single pass of two and are performed three times as per section 3.5. The min-max and cone map generation only has to run if our chosen rendering method is per-pixel-based displacement.

Comparing the two rendering pipelines presents significant challenges due to their distinct strengths and limitations. Initially, we anticipated that the per-pixel rendering method would provide unparalleled visual clarity by enabling each pixel

¹Template available at: <https://github.com/axodox/axodox-graphics>

to independently determine the best intersection point on the surface. However, this clarity was expected to come at the cost of significant computational overhead, both in the preprocessing tasks required to create the cone map and during the intersection-finding process.

In contrast, we predicted that the tessellation-based approach would prioritize performance, producing great images with exceptional speed. The addition of a normal map to the displaced geometry was anticipated to further enhance the lighting calculations. This approach seemed particularly promising for close-up views, where the per-pixel approach was expected to struggle from not converging within a reasonable amount of steps.

Our actual findings reveal that the tessellation approach could match the visual fidelity of the per-pixel displacement while still upholding the significant performance lead. By setting our subdivision limit for the quadtree excessively high, this approach displays pixel-sized rasterized triangles, which with precisely sampled displacements match the per-pixel-based approach in visual accuracy. We initially expected that this would cause a massive overdraw for the tessellation pipeline. Performance testing showed that the tessellation approach is so efficient, that it matches the time it takes the per-pixel method to complete two steps. This is due to high-resolution texture reads per step in the ray marcher being significantly slower than the large-scale, compute-intensive calculations that modern GPUs are architected to handle. Considering that in scenarios that adhere to the ray-marchers advantages it usually has to take 8 steps, which explains our measured best-case 4 times performance difference.

	Per-pixel	Tessellation	
Scenario	Execution time (ms)		Figure
Steep angle	7.758	1.570	A.2a
Medium angle	4.540	1.184	A.2b
Close-up	6.756	1.273	A.2c

Table 5.2: Performance of both rendering techniques measured in milliseconds, given specific situations. Note, that the per-pixel rendering requires additional precomputed helper textures. Those costs are visible in table 5.1. All calculations were done with the highest simulation preset.

A playlist of the simulation with certain simulation settings is available at: <https://youtube.com/playlist?list=PL6wugOnNJsFfMEhkeLJh0Sk1vam4cVFIg&si=X2CZ93vTSzZZsBF7>

Chapter 6

Conclusion

In this work, we explored modern techniques for real-time ocean simulation, focusing on balancing visual fidelity, computational efficiency, and flexibility for artistic expression. Beginning with foundational two-dimensional descriptions of Gerstner waves, to employing statistical wave spectrum models, we achieved a visually compelling ocean surface driven by real-world oceanography research. The integration of Fourier transforms into our simulation pipeline through GPU compute shaders demonstrated the power of large-scale parallelism. We demonstrated our solution to the periodicity problem caused by FFTs, through multi-scale simulations.

The paper explored difficulties with rendering height maps efficiently. Starting with the tessellation-based approach which subdivides a vast plane via a quadtree structure implemented on the CPU. The subdivided geometry is then processed through GPU tessellation to achieve detailed geometry, enabling displacement using the result of the simulation. We explained the performance optimizations of this approach and the challenges with maintaining surface continuity.

As an alternative we have implemented a per-pixel-based displacement, which leverages cone maps for efficient pixel shader ray marching converging to an intersection point on the height map. Despite our efforts, while highly accurate at depicting the ocean surface, this approach struggles with performance constraints, limiting its suitability for real-time rendering of highly fluctuating ocean surfaces.

6.1 Future work

Disturbances

One significant limitation is the lack of interactivity between the ocean surface and external objects. Currently, the system cannot simulate or respond to disturbances, such as the wake of a ship. This limitation is inherent to Fourier transform-based approaches, as that method computes the entire energy flow across the entire surface, not in any specific region of the ocean. Therefore modern simulations utilize a different method to achieve interactivity dubbed iWave [11, 12].

Applying forces to bodies submerged in water

Implementing buoyancy requires an understanding of the underlying forces acting upon an object submerged. However, the most challenging aspect of this simulation lies in the fact that our ocean simulation is confined to the GPU. Calculating the solution for systems of equations for buoyancy is not efficient. It would not also allow for other forces to influence the submerged objects that are processed on the CPU.

Self shadowing

The current shading method does not allow for the surface to create shadow on external objects, or itself, which would be critical in achieving realistic visuals. Expanding our cone mapping solution to follow shadow rays would be a potential solution [13].

6.1.1 Axis aligned min max boxes for rendering

As outlined in Section 5, our per-pixel displacement did not yield the expected results. While the optimization steps detailed in Section 4.2 are apparent, we believe there are additional strategies that could further enhance our outcomes. Inspired by “Hardware Accelerated Per-Pixel Displacement Mapping” we propose subdividing the plane using our Quadtree 4.1.1 implementation similarly to the tessellation approach, but rasterizing cubes at the nodes instead of quads. By stretching the cubes to accommodate the maximum height of our surface within each cube, the ray marcher should require fewer steps to converge. This method should also alleviate the difficult scenarios at low angles. This method reduces the likelihood of reaching

the ray marcher's depth limit; instead, the ray marcher would exit the space defined by the cube and defer the fragment calculation to the neighboring cube's side.

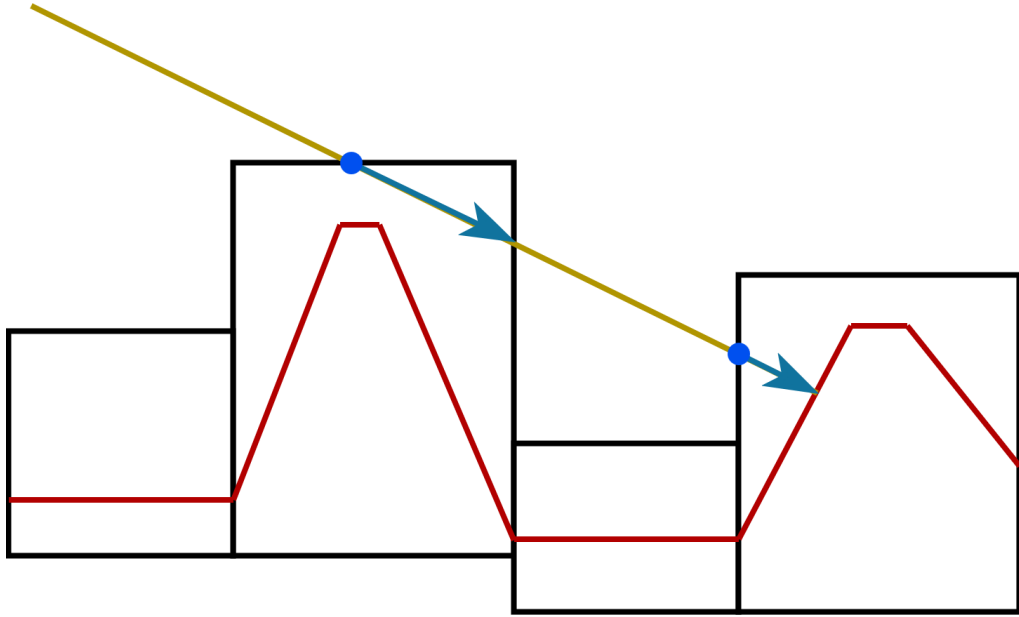


Figure 6.1: Prism rendering being applied to exaggerated min max cubes. The height map is red, the view ray is yellow, and ray marched parts are marked in blue.

Appendix A

Appendix

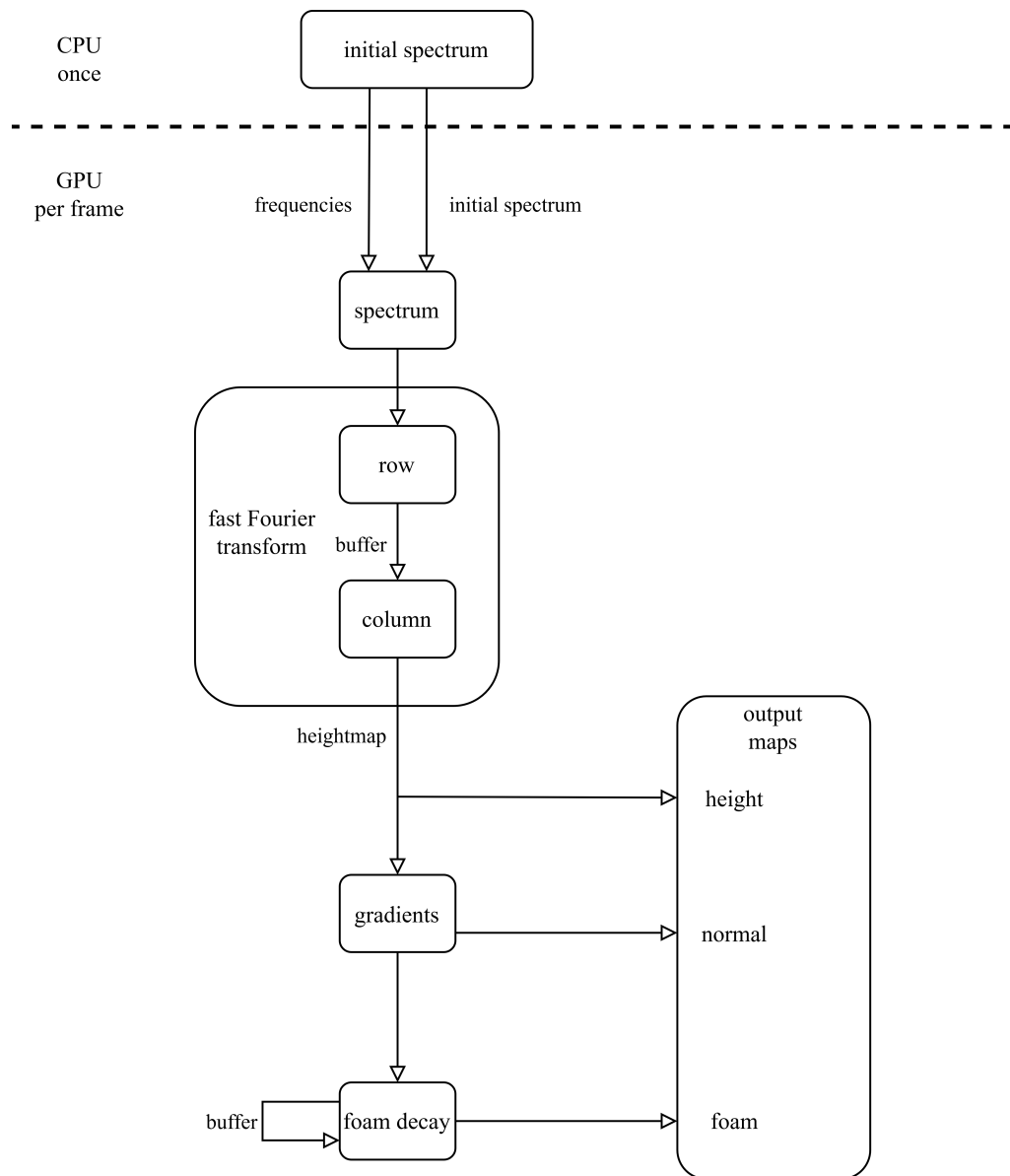
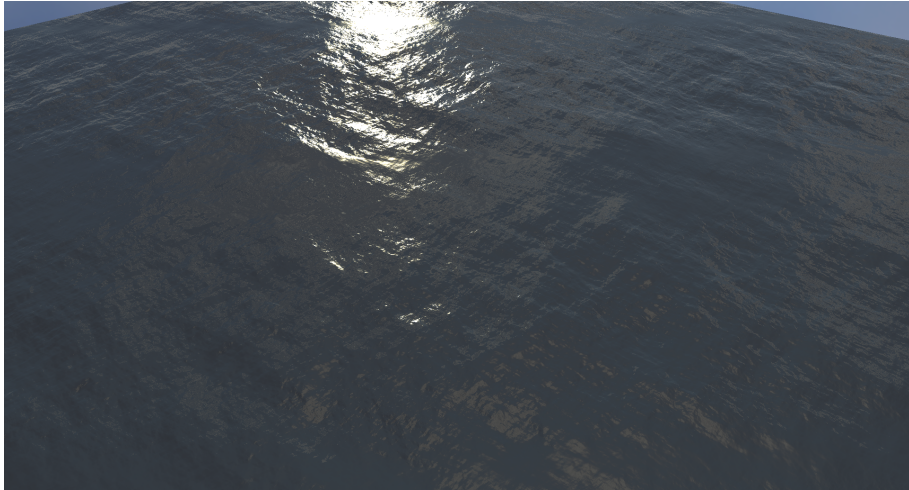
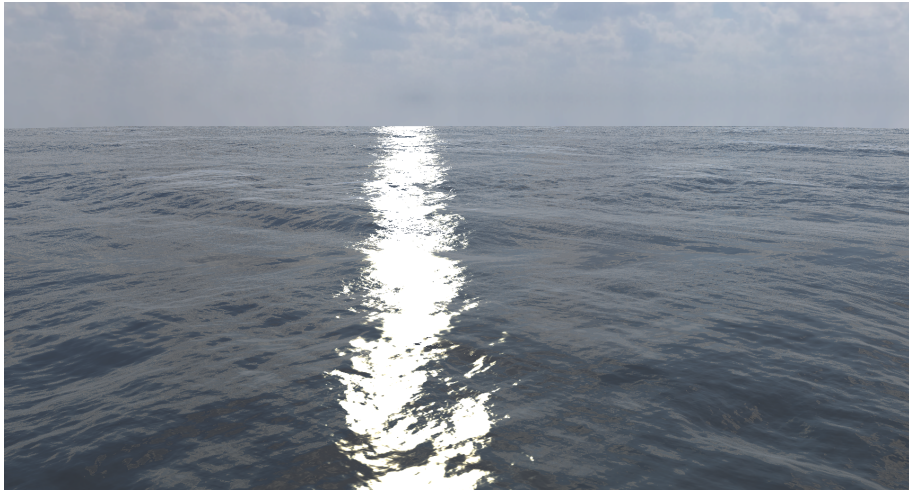


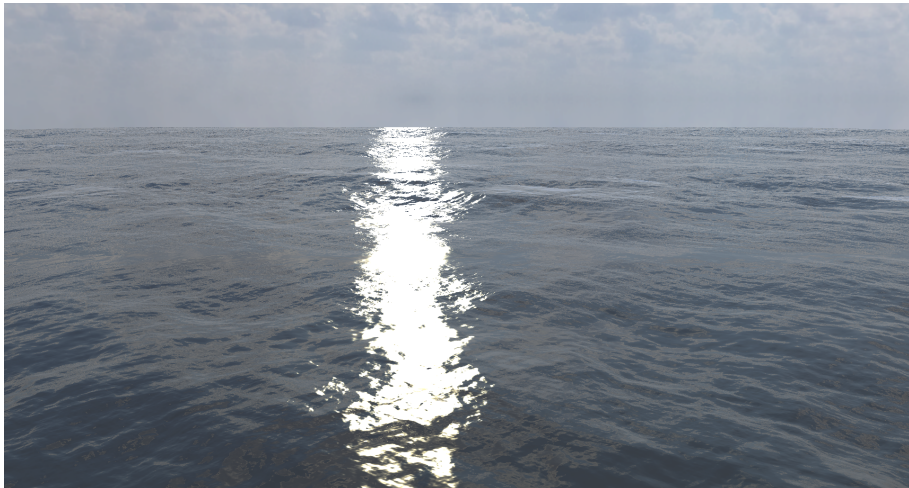
Figure A.1: The simulation pipeline.



(a) Steep angle.

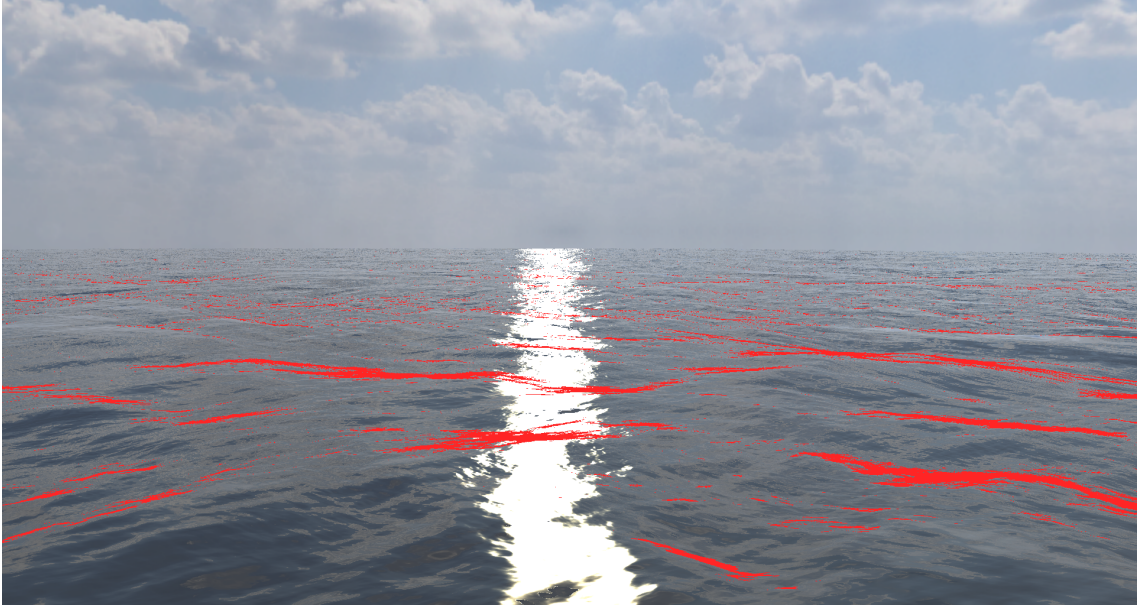


(b) Medium angle.



(c) Close-up

Figure A.2: The ground truth, rendered by per-pixel displacement with maximum 10000 depth.

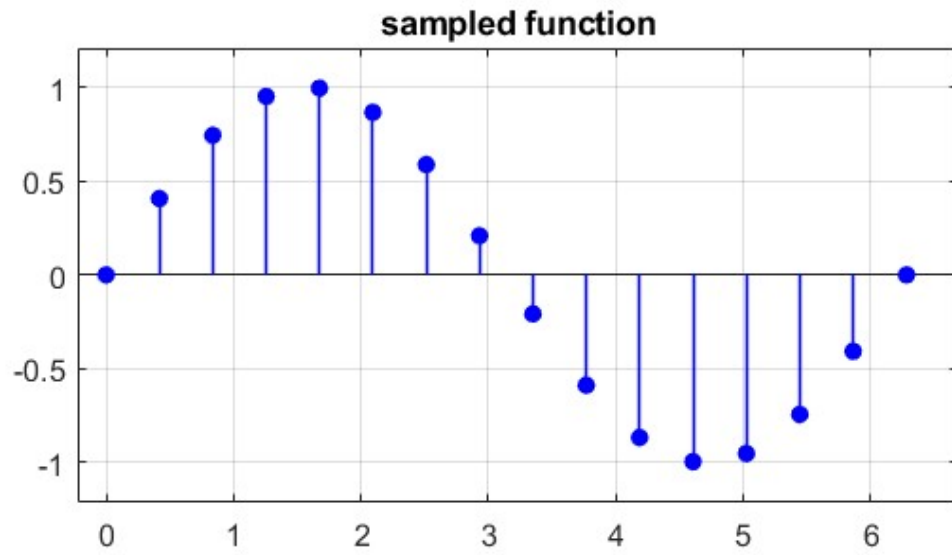


(a) Missed intersections pixels highlighted in red

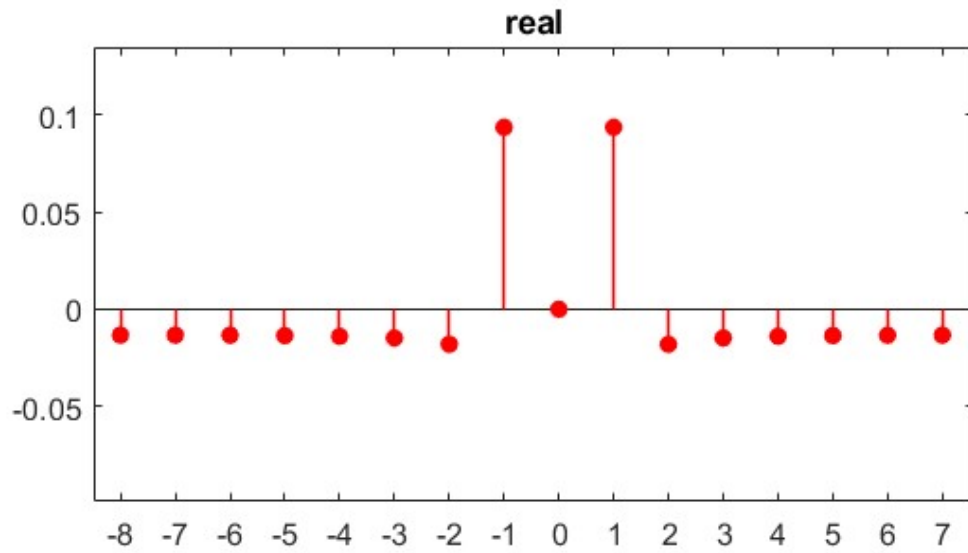


(b) Missed intersections pixels shaded based on the last position tested

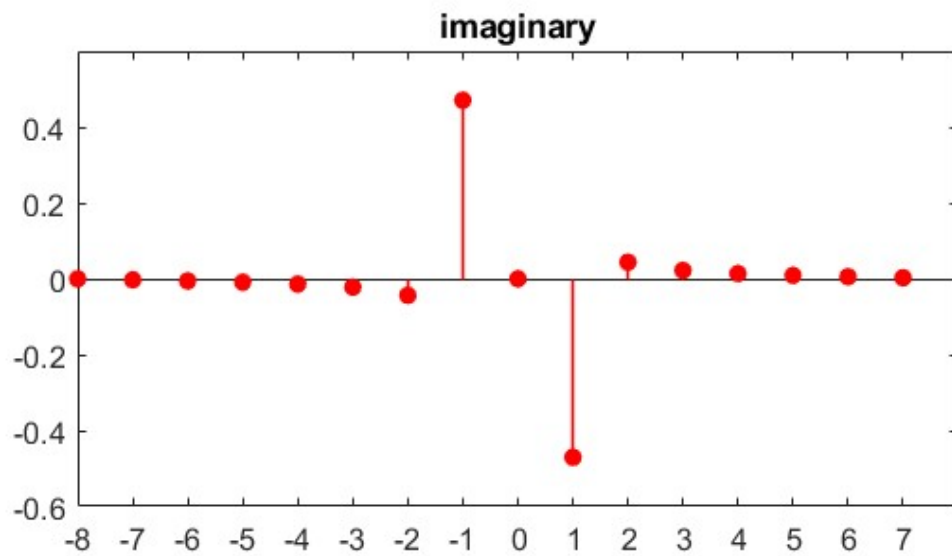
Figure A.3: Artifacts at low angles, where the ray marching method fails to find the intersection



(a) $\sin(x)$ function

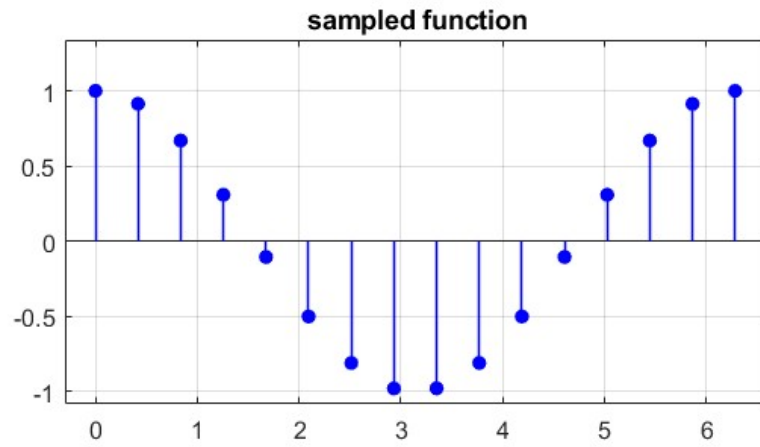


(b) Real part of the DFT

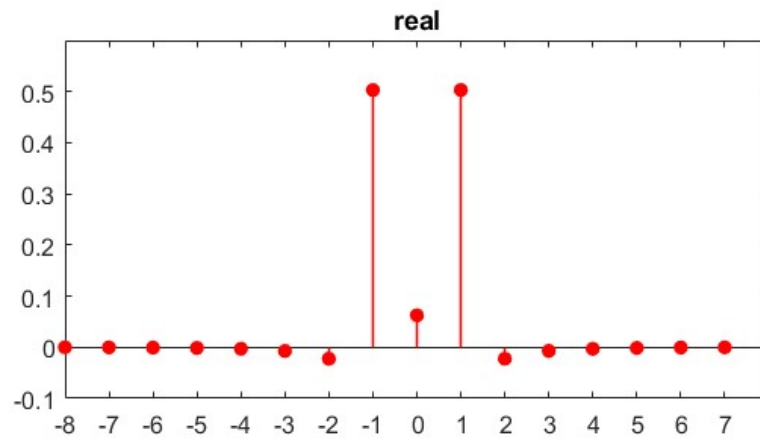


(c) Imaginary part of the DFT

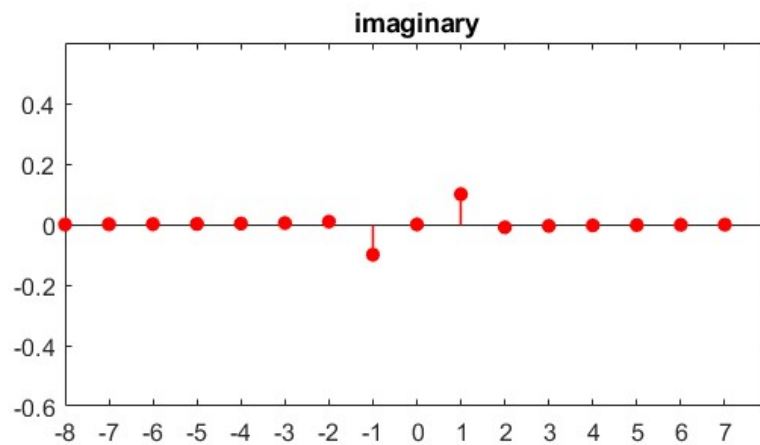
Figure A.4: Sinus wave sampled 16 times, over its period alongside the DFT results.



(a) $\cos(x)$ function



(b) Real part of the DFT



(c) Imaginary part of the DFT

Figure A.5: Cosinus wave sampled 16 times, over its period alongside the DFT results.

Bibliography

- [1] Jerry Tessendorf. “Simulating Ocean Water”. In: *SIG-GRAPH’99 Course Note* (Jan. 2001).
- [2] darthasylum. *Ocean rendering*. Accessed: 2024-11-21. 2018. URL: https://m.cdn.blog.hu/da/darthasylum/tutorials/C++/ch56_ocean.html.
- [3] James W. Cooley and John W. Tukey. “An Algorithm for the Machine Calculation of Complex Fourier Series”. In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842. URL: <http://www.jstor.org/stable/2003354> (visited on 11/22/2024).
- [4] Marina Ryabkova et al. “A Review of Wave Spectrum Models As Applied to the Problem of Radar Probing of the Sea Surface”. In: *Journal of Geophysical Research: Oceans* 124.10 (2019). Accessed 24 Nov. 2024, pp. 7104–7134. DOI: 10.1029/2018JC014804. URL: <https://doi.org/10.1029/2018JC014804>.
- [5] Alain Fournier and William T. Reeves. “A simple model of ocean waves”. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 75–84. ISSN: 0097-8930. DOI: 10.1145/15886.15894. URL: <https://doi.org/10.1145/15886.15894>.
- [6] O. M. Phillips. *The theory of wave spectra*. Vol. 246. Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, 1958, pp. 234–250. DOI: 10.1098/rspa.1958.0151.
- [7] Simon Péter. *Bevezetés az analízisbe II*. ELTE Eötvös, 2014.
- [8] Simon Péter. *Fourier-transzformáció*. ELTE Eötvös, 2019.
- [9] Li Tian. “Ocean wave simulation by the mix of FFT and Perlin Noise”. In: *WSCG 2014 Conference on Computer Graphics* (2015). URL: <https://dspace5.zcu.cz/bitstream/11025/11943/1/Tian.pdf>.
- [10] Róbert Bán et al. “Robust Cone Step Mapping”. In: July 2024. DOI: 10.2312/sr.20241146.

- [11] Jerry Tessendorf. *Interactive Water Surfaces*. https://people.computing.clemson.edu/~jtessen/reports/papers_files/Interactive_Water_Surfaces.pdf. Accessed: 2024-11-27.
- [12] Björn Ottosson. “Real-time Interactive Water Waves”. PhD thesis. Jan. 2011. DOI: 10.13140/RG.2.2.30935.60325.
- [13] Fabio Policarpo and Manuel Oliveira. “Relaxed Cone Stepping for Relief Mapping”. In: (Jan. 2007).
- [14] Johannes Hirche et al. “Hardware Accelerated Per-Pixel Displacement Mapping”. In: (May 2004).

List of Figures

2.1	See simulation running in: https://youtu.be/i8zhWRCqHss	5
3.1	A single Gerstner wave in dark blue. The displaced line is marked in red.	7
3.2	The result of adding together three Gerstner waves already creates an interesting surface. The wave is dark blue, the displaced line is marked in red.	7
3.3	A sawtooth function from -4π to 4π	10
3.4	16th partial Fourier series sum approximating the sawtooth function.	10
3.5	Initial starting spectrum, generated with a $(+0.9,+0.4)$ XZ wind vector.	15
3.6	Gradients texture generated via the finite differential algorithm, the red, green, and blue components correspond to the XYZ axes respectively.	17
4.1	Illustration of a heightmap texture and displaced geometry. Height map generated by https://manticorp.github.io/unrealheightmap	19
4.2	Quadtree visualization.	20
4.3	The order in which quadtree nodes children are explored given a camera angle in blue.	21
4.4	Visualizing how the method traverses the quadtree given a camera position, view, and frustum in blue.	22
4.5	Visualizing how differing sized quadtree nodes can create gaps.	23
4.6	A naive ray marcher missing an intersection point in a fluctuating heightmap.	25
4.7	For a given point p , visualize the minimum distance to reach new data holding texel. Illustration taken from [10].	26

4.8	A 1D slice of the height map (blue curve) with the relaxed cone (green triangle). From the cone apex, the cone angle α is the largest such that the cone still excludes the red back-facing terrain. Illustration taken from [10].	26
4.9	Ray intersecting a cone. The ray is colored yellow, the relaxed cone is blue, the height map is red, and the next step is purple.	27
6.1	Prism rendering being applied to exaggerated min max cubes. The height map is red, the view ray is yellow, and ray marched parts are marked in blue.	33
A.1	The simulation pipeline.	34
A.2	The ground truth, rendered by per-pixel displacement with maximum 10000 depth.	35
A.3	Artifacts at low angles, where the ray marching method fails to find the intersection	36
A.4	Sinus wave sampled 16 times, over its period alongside the DFT results.	37
A.5	Cosinus wave sampled 16 times, over its period alongside the DFT results.	38

List of Algorithms

1	FFT parallelized implementation for rows of inputTexture.	14
2	Determine relative size by traversing neighbors and children in a quad- tree	24