How to create an efficient real-time Ocean Simulation in Rust?

Ilya Rasputin
Candidate Number: 5092

August 2025 - April 2026

Contents

1 Introduction

1.1 Overview

There are these grand, gorgeous and spectacular oceans in many video-games like *Sea of Thieves* [1], or even in famous movies like *Titanic* [2]. But how is it possible to create such an ocean and what mathematics, as well as computer science go into this?
This discussion will explore the technology and mathematics behind real-time ocean simulations, building up its own interpretation in this field. First, this project will discuss the theory, and afterwards this project will look at more practical and efficient approximation methods more commonly used in the real world. Using various lighting techniques and building on top of Tessendorf's implementation [3], which includes the use of the Phillips spectrum [4] and Inverse Fast Fourier Transform algorithms to achieve a realistic and visually compelling result.

Figure 1: *Image from the video game* Sea of Thieves *[1]*

1.2 Applications

Some of the most common use cases for an ocean simulation is the testing of hardware, as well as improving the graphical standpoint of a project, to attract a larger customer base [5]. In addition, it could serve as a great lesson on computational shaders, lighting and optimisation techniques.

1.3 Aims & Goals

By far, the most important goal of this project is to explore all the details and dive deep into maths rabbit-holes, where the end goal is producing a fully functional simulation. Secondly, another important goal for this discussion is to explore the possibilities of creating an efficient simulation using minimal external ready-to-use tools and engines, such as *Unreal Engine* [6] and *Unity* [7], in Rust. This helps to develop the skills and knowledge about lighting, 3D rendering as well as providing an opportunity contribute to the Rust community by acting as guidance for fellow ambitious programmers.

1.4 Prerequisites

Before the start of this discussion, the key words, ideas and mathematical knowledge have to be mentioned.

1.4.1 Ocean

What is an ocean? An ocean is a large continuous body of water filled with sea salt. Oceans cover 71% of the whole earth surface and most of this water has a depth of 3,668 meters [8]. Primarily the ocean waves are generated by the transfer of energy from the wind onto the water's surface, forming waves. In this study the Phillips Spectrum [4] will be used to account for wind direction and any other factors to make the ocean seem more vivid and realistic.

1.4.2 Efficient Real-Time Simulation

Real-time simulations refer to the type of simulations and computer programs that produces an output at the same rate as the real world time, allowing for seamless effect and direct interactions with the user. While in "offline" [9] or "non-real-time" simulations the output is usually predetermined and perhaps rendered and computed beforehand, since the algorithms used there are usually slower. The challenge of this paper is to find a sweet spot that would still look vivid and realistic, but provides a suitable frame rate of about $\sim$ 60FPS, which also correlates to $\sim$ 16.7ms of frame time.

1.4.3 Programming Language and External Libraries

First, a programming language has to be chosen determining at what pace the development will go, as well as the available tools. For this case study, the programming language of choice is Rust [10], having a great typing system, being memory safe and having an emerging ecosystem of useful libraries like `wgpu` [11] providing a graphics API. It is important to consider a game engine, like *Unity* [7] or *Unreal Engine* [6] which allows for faster development speed and ease of use, however the challenge of beginning from scratch and having more control is more beneficial.

1.4.4 Open Source & Git

As stated previously, driving the success of others is an important aim of this discussion. Therefore the whole repository [12] is available under an *MIT* [13] license on GitHub, a cloud based platform where you can store, share, and work together with others to write code [14].

1.4.5 Glossary of Mathematical Terms

Note: Help box

Since this is a highly technical and mathematics based project, the reader may see these "Note" boxes appear throughout. These act as a guide, providing hints and useful information on the topic currently being discussed.

Notation	Description
$f(x)$	Function of a real variable
$f : A \to B$	Function mapping set A to set B
$\sin x, \cos x, \tan x, \sinh x, \cosh x, \tanh x$	Trigonometric functions
$\arcsin x, \arccos x, \arctan x$	Inverse trigonometric functions
$\sinh x, \cosh x, \tanh x$	Hyperbolic functions
$\sinh^{-1} x, \cosh^{-1} x, \tanh^{-1} x$	Inverse hyperbolic functions
$\mathbb{R}, \mathbb{Z}, \mathbb{Q}$	Real, integer, rational numbers
$\subseteq, \in, \notin$	Subset, element of, not element of
$\{ x \in A : P(x) \}$	Set-builder notation
$\lim_{x \to 0}, x \to 0$	Limit as x approaches 0
$\vec{v}$	Vector (general)
$\hat{v}$	Unit vector
$\|\vec{v}\|$	Magnitude (norm)
$\nabla \cdot \vec{F}$	Divergence of vector field
$\nabla \times \vec{F}$	Curl of vector field
$\frac{d}{dx}, \frac{df}{dx}, \frac{df(x)}{dx}$	Derivative
$\frac{\partial}{\partial x}, \frac{\partial f}{\partial x}, \frac{\partial f(x)}{\partial x}$	Partial derivative
$\int_a^b f(x)\,dx$	Definite integral
$\int_\Omega f(x)\,dx$	Integral over the domain Ω
∇f	Gradient of scalar field
$\nabla \vec{F}$	Gradient of a vector field
$\nabla^2 f$	Laplacian

Figure 2: *Table of standard mathematical notation.*

In addition to the mathematical terms, the following `inline_code_blocks` will be used throughout the discussion, showcasing a connection to the code and following `wgsl` terms will also come up, as shown in Figure 3.

Function	Description
normalise($\vec{v}$)	Normalisation of a vector
mix(a, b, t)	Linear interpolation
smoothstep(a, b, t)	Cubic Hermite interpolation

Figure 3: *WGSL and programming function notation.*

2 Literature Review

2.1 Sinusoidal functions

2.1.1 Parameters of a sine wave

Sinusoidal waves are special type of a functions, oscillating up and down with a periodical frequency of 2π. Originating from a simple unit circle ($\cos^2\theta + \sin^2\theta = 1$), as illustrated in Figure 5.

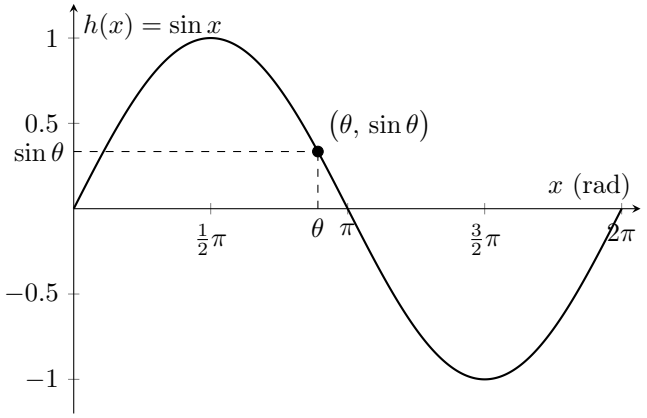


Figure 4: *Sine wave at* $(\theta, \sin\theta)$

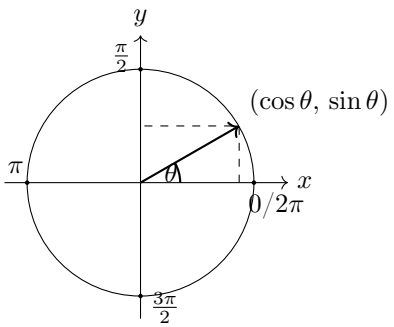
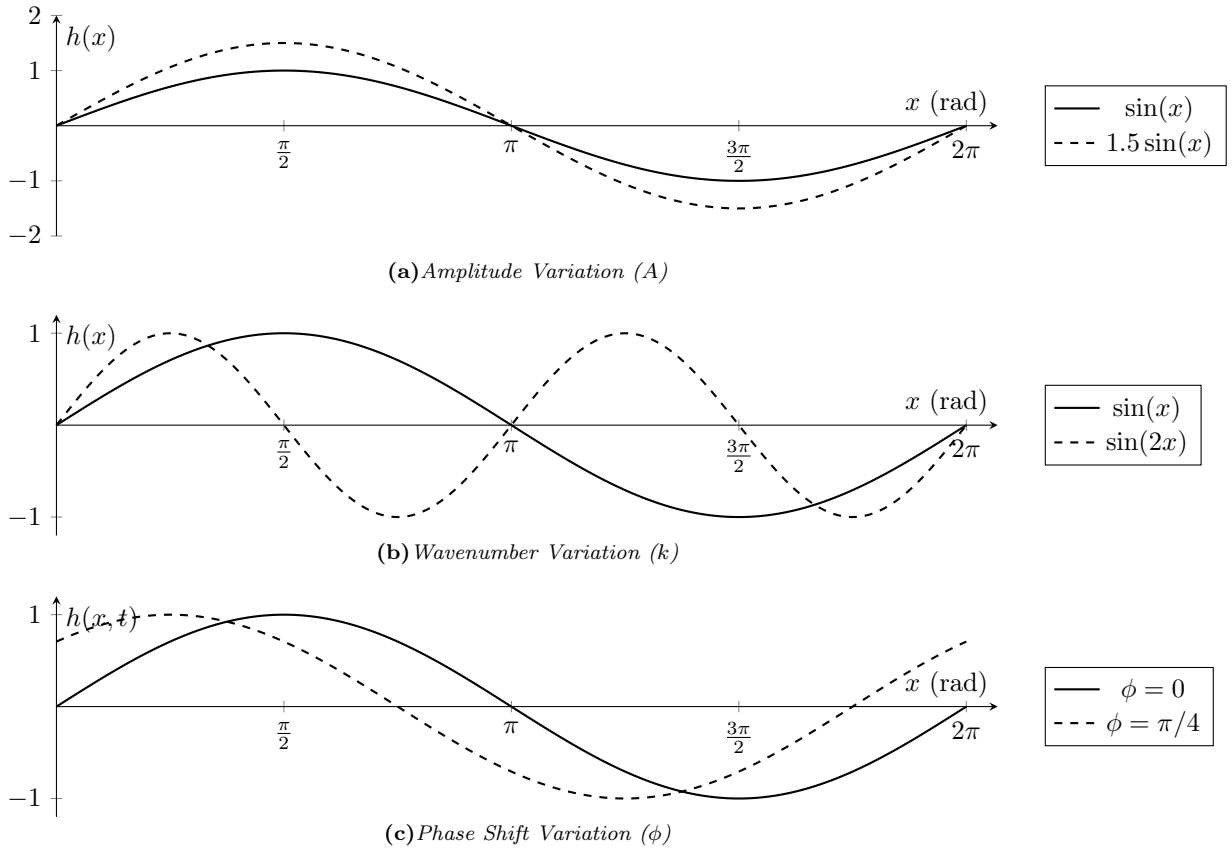


Figure 5: *This unit circle presents a natural and oscillating output as the angle increases, producing the waveform seen in Figure 4*

In a sine wave its amplitude A refers to the maximum displacement, wavenumber k governs the spatial frequency. While the angular frequency ω, and phase shift ϕ allow to encode the initial random phase, as well as fake movement. By add time t, and using a spatial variable x instead of an angle θ the general form of a sine wave including all these parameters is as shown in Equation (1).

$$h(x,t) = A \cdot \sin(k \cdot x + \omega \cdot t + \phi) \tag{1}$$



(a) *Amplitude Variation (A)*

(b) *Wavenumber Variation (k)*

(c) *Phase Shift Variation (ϕ)*

Figure 6: *Visual breakdown of sine wave parameters: (a) demonstrates amplitude affecting vertical peak height, (b) shows wavenumber controlling spatial frequency, and (c) illustrates the horizontal displacement caused by phase shifting.*

2.1.2 Dispersion relation

Wave dispersion tells us that waves with different wavelengths travel at different speeds and the relationship is represented in Equation (2).

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

The depth of water h is assumed to be very large, then as $h \to \infty$, $\tanh(kh) \to 1$, implying Equation (3).

$$\omega = \sqrt{g \cdot ||\vec{k}||} \tag{3}$$

Deep water usually refers to $h > \lambda/2$ [3]

2.2 Gerstner waves

Gerstner waves have a much more real-world approach to modelling the ocean, being first derived by Franz Josef Von Gerstner in 1809 [15]. Gerstner waves are great at conveying the sharpness of peaks and the flatness of troughs, because they incorporate horizontal displacement.

However, this method will not be used in this discussion. Firstly, Gerstner waves break at scale since the parameters for amplitude, direction and phase have to be generated and matched by hand. Secondly, while the computational power may be cheap for a few waves, the algorithm scales poorly when trying to match

the detail provided by other FFT methods. Lastly, Gerstner Waves have the ability to fold over themselves, as shown in Figure 7.

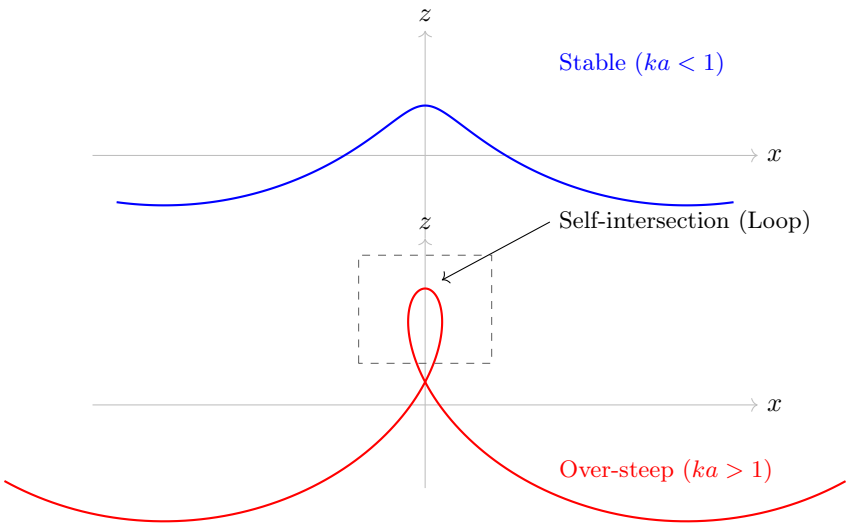


Figure 7: *Visual representation of Gerstner wave folding over itself*

2.3 Phillips spectrum

The ocean surface waves can exist across many different kinds of shapes and scales, therefore a statistically accurate model is used, like the Phillips spectrum. Phillips Spectrum being a well known spectra of sea waves in the presence and in the absence of wind [4]. This section will explore how the equation is formed following Tessendorf's method [3].

2.3.1 Theoretical Derivation

Phillips (1958) [16] showed that at high wavenumbers the equilibrium range is maintained by wind and breaking, making wind speed negligible so the spectrum $h(k)$ depends only on g and $\vec{k}$, as confirmed by dimensional analysis.

$$h(k) \propto g^a k^b \tag{4}$$

Using $[h(k)] = \mathrm{L}^2$, $[g] = \mathrm{LT}^{-2}$ and $[k] = \mathrm{L}^{-1}$:

$$[g^a k^b] = (\mathrm{LT}^{-2})^a (\mathrm{L}^{-1})^b = \mathrm{L}^{a-b}\mathrm{T}^{-2a} \tag{5}$$

Matching dimensions with $h(k)$ gives $a = 2$ and $b = -4$, hence

$$h(k) = \alpha_P \frac{g^2}{k^4} \tag{6}$$

where $\alpha_P \approx 0.0081$ is the Phillips Constant, which was determined from empirical ocean measurements [16].

2.3.2 Directionality and Scale

Since the k^{-4} term diverges to ∞ as $k \to 0$, and the wind speed V can only sustain waves up to the scale at which wave phase speed $c = \sqrt{\frac{g}{k}}$ equals to V. The maximum wavelength is as shown in Equation (7).

$$L = \frac{V^2}{g} \tag{7}$$

Waves larger than L are not possible to be generated by the wind, therefore will be suppressed by the factor $\exp\left(\frac{-1}{k^2L^2}\right)$, which falls to zero for $k \ll 1/L$ and approaches unity for $k \gg 1/L$.

By following Tessendorf's [3] implementation, a directional weighting, D, is introduced.

$$D(\vec{k}, \hat{w}) = \left|\hat{k} \cdot \hat{w}\right|^2 = \cos^2\theta \tag{8}$$

where $\hat{w}$ is the wind unit vector and θ is the angle between $\vec{k}$ and $\hat{w}$.

At high wave numbers $k \gg 1/l$, the theory starts to break down, as surface tension starts to dominate. These tiny changes can easily cause aliasing issues, therefore a Gaussian low-pass filter is implemented.

$$G(k, l) = \exp\left(-k^2\, l^2\right) \tag{9}$$

This filter eliminates this energy above the cut-off scale l [3].

Hence, by combining the equations for the range (6), wind-scale cut-off (7), the directional factor (8) and the small scale damping (9), the Phillips spectrum will be as described in Equation (10) [3].

$$P(\vec{k}) = A\,(\hat{k} \cdot \hat{w})^2\, \exp\left(\frac{-1}{(||\vec{k}||^2\, L^2)}\right) \exp\left(-||\vec{k}||^2\, l^2\right) ||\vec{k}||^{-4} \tag{10}$$

where the dominant scale $L = ||\vec{V}||^2/g$.

2.3.3 Statistics and Time Evolution

However, this is a power spectrum density, it specifies the variance, but not the phase of each Fourier Mode. To generate a surface, the ocean has to be treated as Gaussian Random Field [17]. Therefore, for each vector $\vec{k}$, two independent samples are required using the Box-Muller Transform is used, which results in two samples of the Gaussian Random Field, ξ_i, ξ_r. This forms the complex Fourier amplitude as described in Equation (11).

$$\tilde{h}_0(\vec{k}) = \frac{\xi_r + i\,\xi_i}{\sqrt{2}} \sqrt{P(\vec{k})} \tag{11}$$

The $1/\sqrt{2}$ term ensures that the complex magnitude has the correct variance $P(\vec{k})$, instead of $2P(\vec{k})$. In addition to $\tilde{h}_0(\vec{k})$, this guarantees that the ocean surface will be composed of only the real values after the IFFT. This requirement is known as the Hermitian Symmetry condition. Time Evolution is then controlled by the deep-water dispersion relation, covered in Section 2.1.2. This shifts the phase of each mode over time, as governed in Equation (12)

$$\tilde{h}(\vec{k}, t) = \tilde{h}_0(\vec{k})\, \exp\left(i\omega(\vec{k})\, t\right) + \tilde{h}_0^*(-\vec{k})\, \exp\left(-i\omega(\vec{k})\, t\right) \tag{12}$$

An Inverse Fast Fourier Transform (IFFT) over the full $N \times N$ grid produces a spatial surface $\eta(x, t)$ at each timestep.

2.4 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform is a technique used to convert data from time or space domain to frequency domain, splitting up data into smaller chunks that it's made up of as shown in Figure 8. This transform maps the complex vector $\vec{x}$ with N elements $(x_0, x_1, x_2, \ldots, x_{N-1}) = x_N$ onto a different vector $\vec{X}$ also with N elements $(X_0, X_1, X_2, \ldots, X_{N-1}) = X_k$. Where X_k is to be defined as in Equation (13). [18]

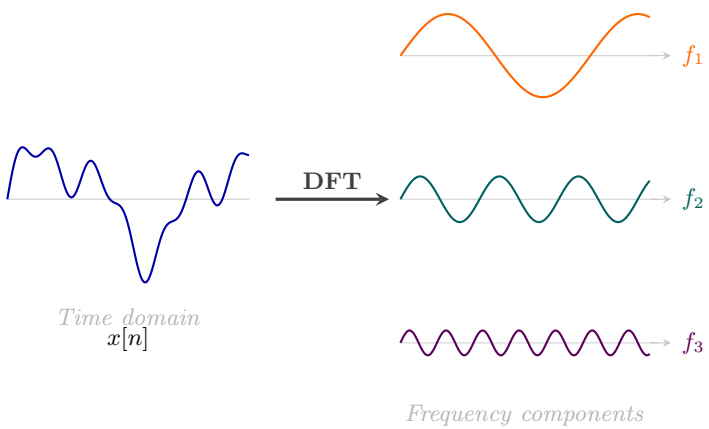


Figure 8: *The DFT decomposes a composite discrete signal $x[n]$ into its sinusoidal components at distinct frequencies f_1, f_2, f_3, each with its own amplitude and phase.*

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} kn} \tag{13}$$

Substituting $\exp\left(\frac{-2\pi i}{N}\right) = \nu$, results in Equation (14) [18]

$$X_k = \sum_{n=0}^{N-1} x_n \nu^{kn} \tag{14}$$

ν can be expressed in terms of trigonometric functions as shown in Equation (15).

$$\begin{aligned}
\nu &= \exp\left(\frac{-2\pi i}{N}\right) \\
\nu &= \cos\left(\frac{-2\pi}{N}\right) + i\sin\left(\frac{-2\pi}{N}\right) \\
\nu &= \cos\left(\frac{2\pi}{N}\right) - i\sin\left(\frac{2\pi}{N}\right) \\
\nu^{kn} &= \left[\cos\left(\frac{2\pi}{N}\right) - i\sin\left(\frac{2\pi}{N}\right)\right]^{kn} \\
\nu^{kn} &= \cos\left(kn\,\frac{2\pi}{N}\right) - i\sin\left(kn\,\frac{2\pi}{N}\right)
\end{aligned} \tag{15}$$

Calculating a single frequency X_k, requires $N \times N$ operations, therefore a Fast Fourier Transform (FFT) is used.

2.4.1 Fast Fourier Transform (FFT)

Most common example of a FFT is the Cooley-Tukey Radix-2 Decimation-In-Time. Radix-2 works by computing separate DFT's of even inputs $(x_0, x_2, x_4, \dots)$ and odd inputs $(x_1, x_3, x_5, \dots)$ [18] as shown in Equation (16). This is the core principle, as it is able to recursively split up a DFT of size N into 2 separate DFT's of size $N/2$, referred to as butterfly passes.

$$
\begin{aligned}
X_k &= \sum_{m=0}^{N/2-1} x_{2m}\, e^{\frac{-2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1}\, e^{\frac{-2\pi i}{N}(2m+1)k} \\
X_k &= \sum_{m=0}^{N/2-1} x_{2m}\, e^{\frac{-2\pi i}{N}(2m)k} + e^{\frac{-2\pi i}{N}} \sum_{m=0}^{N/2-1} x_{2m+1}\, e^{\frac{-2\pi i}{N}(2m)k}
\end{aligned}
\tag{16}
$$

Equation (16) consists of the even indexed part DFT E_k and odd indexed part DFT O_k, rewritten as (17).

$$
X_k = E_k + e^{\frac{-2\pi i}{N}k}\, O_k \tag{17}
$$

Functions of k, E_k and O_k repeat every $N/2$, Equation (17) can be rewritten as shown in Equation (18).

$$
\begin{aligned}
E_{k+N/2} &= E_k \\
O_{k+N/2} &= O_k \\
X_k &= \begin{cases} E_k + e^{\frac{-2\pi i}{N}k}\, O_k, & \text{for } 0 \le k < N/2 \\ E_{k-N/2} + e^{\frac{-2\pi i}{N}k}\, O_{k-N/2}, & \text{for } N/2 \le k < N \end{cases}
\end{aligned}
\tag{18}
$$

Using the following property of the twiddle factor [18],

$$
e^{\frac{-2\pi i}{N}(k+N/2)} = e^{\frac{-2\pi ik}{N} - \pi i} = e^{-\pi i}\, e^{\frac{-2\pi ik}{N}} = -e^{\frac{-2\pi ik}{N}} \tag{19}
$$

X_k can be rewritten as shown in Equation (20) [18].

$$
\begin{aligned}
X_k &= E_k + e^{\frac{-2\pi ik}{N}}\, O_k \\
X_{k+N/2} &= E_k - e^{\frac{-2\pi ik}{N}}\, O_k
\end{aligned}
\tag{20}
$$

2.4.2 Bit Reversal

For a transform to be a Decimation in Time (DIT), where the array is split into even and odd indexed values, once the algorithm reaches the end, data becomes completely scrambled, as shown in Figure 9.

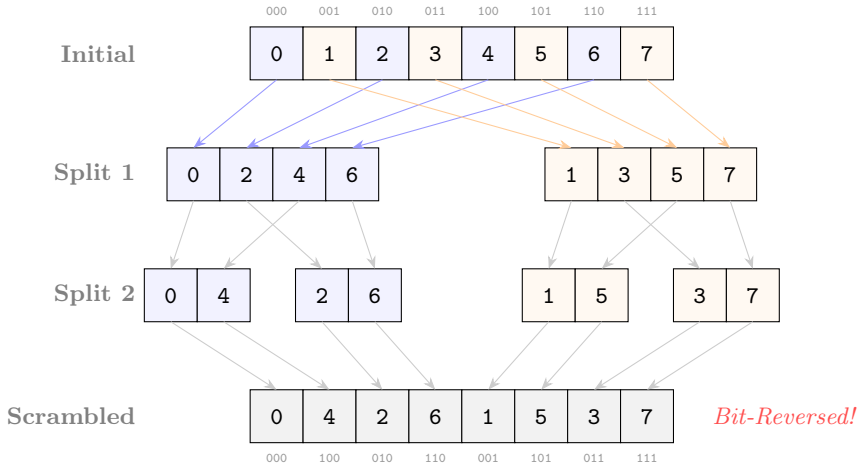


Figure 9: *Recursive DIT of an $N = 8$ array with bitwise representations.*

This allows for the clever use of bitwise manipulation to flip the bits around for the original array, and let the algorithm sort everything into its correct place.

2.4.3 Inverse Fast Fourier Transform (IFFT)

The Inverse Discrete Fourier Transform does the opposite by converting the frequency domain back to spatial domain. The definition of the IDFT is as shown in Equation (21)

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} \tag{21}$$

The sign on the twiddle factor is now positive and a $1/N$ scaling factor is added. The same radix-2 logic can be applied for both the forward and inverse transforms as shown in the following steps.

1. Conjugate the complex input
2. Perform standard radix-2 FFT.
3. Conjugate the complex input again
4. Add a scaling factor of $1/N$

2.5 Lighting Models

For user to see an image, a lighting model has to be in place. Any pipeline works in a similar fashion, where the CPU does all the setup and logic, and after sending draw calls to the GPU. Due to the parallelism power of the GPU, it can use shaders to efficiently display an image to the screen. Shaders are programs which run on the GPU [19], such as the Vertex Shader, being responsible for the movement of vertices, and the Fragment Shader, used for colouring in individual pixels. The general architecture of this rendering pipeline is as shown in Figure 10.

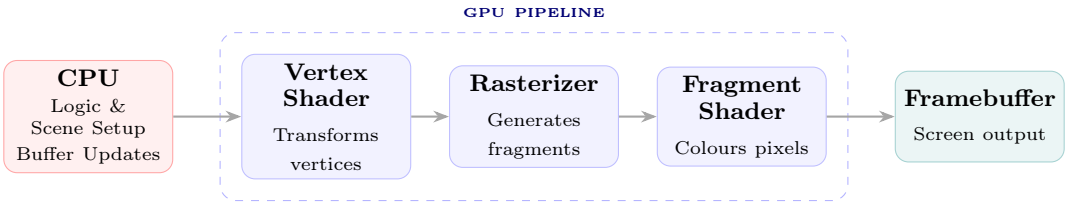


Figure 10: *CPU prepares the scene and issues draw calls to the GPU, which processes geometry through the shader stages before writing pixel colours to the framebuffer.*

2.5.1 Surface Normals

The shader must first determine the surface normal $\hat{n}$ for every point on the wave. From vector calculus, the normal to such a surface is given by the gradient of the implicit function $f(\vec{x}, z, t) = z - \eta(\vec{x}, t) = 0$, [3] as shown in Equation (22).

$$\vec{n} = \vec{\nabla} f = \vec{\nabla}(z - \eta(\vec{x}, t)) = \begin{bmatrix} -\frac{\partial \eta}{\partial x} \\ -\frac{\partial \eta}{\partial y} \\ 1 \end{bmatrix} = \begin{bmatrix} -\vec{\nabla}\eta \\ 1 \end{bmatrix} \tag{22}$$

The unit normal $\hat{n}$ used in the shader is computed by normalizing the vector:

$$\hat{n} = \frac{\vec{n}}{||\vec{n}||} \tag{23}$$

Inside of shaders, an approximation method is used for the gradient $\vec{\nabla}\eta$, like the central difference method. For a point on the grid with spacing Δs the partial derivative is estimated as shown in Equation (24).

$$\begin{aligned} \frac{\partial \eta}{\partial x} &\approx \frac{\eta(x + \Delta s, y, t) - \eta(x - \Delta s, y, t)}{2\Delta s} \\ \frac{\partial \eta}{\partial y} &\approx \frac{\eta(x, y + \Delta s, t) - \eta(x, y - \Delta s, t)}{2\Delta s} \end{aligned} \tag{24}$$

2.5.2 Cook-Torrance PBR model

Physically Based Rendered (PBR) model is a realistic and efficient approach towards lighting. The ground truth for PBR is the rendering equation as shown in Equation (25) which states that the total radiance L_0 reflected from a point p in direction ω_o is the integral of the incoming light L_i from all directions ω_i over the hemisphere Ω.

$$L_o(p, \omega_o) = \int_\Omega f_r(p, \omega_i, \omega_o)\, L_i(p, \omega_i)\, (\hat{n} \cdot \omega_i)\, d\omega_i \tag{25}$$

Where f_r is the Bidirectional Reflectance Distribution Function (BRDF). Because calculating this integral in real-time is computationally impossible, the Cook-Torrance microfacet model is used as an approximation, being composed of primarily three parts: the Normal Distribution Function D, the Geometry Function G and the Fresnel term F [20], which are then combined into a single BRDF as shown in equation (26).

$$\text{Final} = (\text{Diffuse} + \text{Specular}) \cdot (\text{LightColor} \cdot \text{LightDiffuseIntensity}) \cdot (\hat{n} \cdot \vec{d_L}) \tag{26}$$

The diffuse term follows a Lambertian model, where f_{lambert} is simply the surface albedo colour [21]. While the specular term uses a Cook-Torrance BRDF, $f_{\text{CookTorreace}}$, ensuring energy conservation using $K_s = (1 - K_d)$, meaning the sum of the diffuse and specular contributions never exceed incoming energy.

$$\begin{aligned} \text{Diffuse} &= K_d \cdot \frac{f_{\text{lambert}}}{\pi} \\ \text{Specular} &= (1 - K_d) \cdot f_{\text{CookTorrance}} \end{aligned} \tag{27}$$

The Cook-Torrance BRDF will be defined as shown in Equation (28), the denominator ensuring that energy remains conserved.

$$f_{\text{CookTorrance}} = \frac{D \cdot G \cdot F}{4 \cdot (\hat{n} \cdot \vec{d_L}) \cdot (\hat{n} \cdot \hat{v})} \tag{28}$$

The Normal Distribution Function D, models the microscopic structure of the surface since real surfaces are made up of tiny microfacets each with their own normal. This function determines what fraction of the microfacets are orientated towards the halfway vector $\hat{h} = \text{normalize}(\vec{d_L} + \vec{v})$, and therefore contribute to the specular highlight as shown in Figure 11. The GGX distribution [22] is also used here, as shown in Equation (29), where α is the surface roughness coefficient.

$$D = \frac{\alpha^2}{\pi \cdot \Big((\hat{n} \cdot \hat{h})^2 \cdot (\alpha^2 - 1) + 1\Big)^2} \tag{29}$$

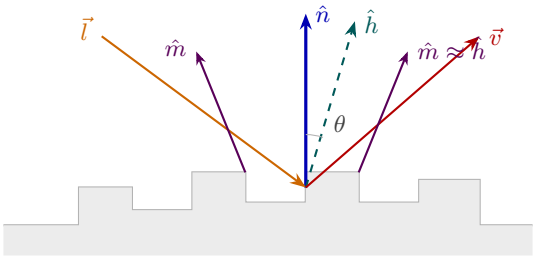


Figure 11: *Microfacet surface model, facets whose normal $\hat{m}$ aligns with the halfway vector $\hat{h}$ contribute to specular reflection (Simplified diagram)*

However, the light path may still be partially blocked due to neighbouring microfacets casting shadows on it (shadowing), or block the reflected ray from reaching the viewer (masking) as shown in Figure 12. The Geometry Function G, accounts for this using the Smith method [23] as shown in Equation (30) where S is the Schlick-GGX approximation as shown in Equation (31).

$$G = S(\hat{n} \cdot \vec{v}) \cdot S(\hat{n} \cdot \vec{d_L}) \tag{30}$$

$$S(d_p) = \frac{d_p}{d_p \cdot (1-k) + k}, \qquad k = \frac{(\alpha+1)^2}{8} \tag{31}$$

This being the core term preventing PBR materials from appearing unrealistically bright at grazing angles, commonly seen in Blinn-Phong.

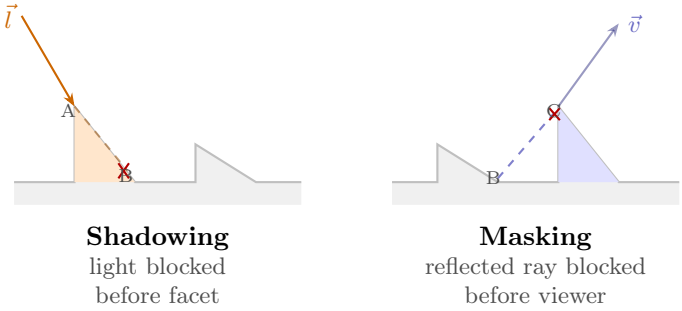


Figure 12: *Shadowing and masking. The geometry function G attenuates the contribution of affected facets.*

The Fresnel term describes how the viewing angle changes the reflectivity term as shown in the Equation (32), since at grazing angles all surfaces become highly reflective regardless of the material. The F_0 term represents the base reflectivity at normal incidence.

$$F = F_0 + (1 - F_0) \cdot (1 - (\vec{v} \cdot \hat{h}))^5 \tag{32}$$

After combining these terms, the final PBR is defined by the Equation (33).

$$\text{PBR} = \left(\frac{K_d \cdot \text{Color}}{\pi} + \frac{(1-K_d) \cdot D \cdot G \cdot F}{4 \cdot (\hat{n} \cdot \vec{l}) \cdot (\hat{n} \cdot \vec{v})} \right) \cdot (\text{LightColor} \cdot \text{LightDiffuseIntensity}) \cdot (\hat{n} \cdot \vec{l}) \tag{33}$$

3 Discussion

The knowledge about Fourier Transforms, Phillips spectrum and buffers will now be applicable in practice.

3.1 Initial Setup

First, the window has to be created, therefore the `winit` library will be used. The `main` function will be responsible for the creation of the event loop, as well as creating the `App` struct and initialising default states. The `App` struct implements methods needed for a window to work. The `resumed` method will be called once the application is ready, it will create the window and call the startup sequence of the `State` struct as shown in Figure 13.

```rs
fn resumed(&mut self, event_loop: &ActiveEventLoop) {
    let window_attributes = Window::default_attributes()
        .with_title(format!("Ocean Simulation, build {VERSION}"))
        .with_inner_size(LogicalSize::new(2560, 1440));
    let window = Arc::new(event_loop.create_window(window_attributes).unwrap());

    window.set_cursor_visible(false);
    if window.set_cursor_grab(CursorGrabMode::Locked).is_err() {
        log::warn!("Could not lock cursor")
    }
    self.state = Some(pollster::block_on(State::new(window)).unwrap());
}
```

Figure 13: *The* `resumed` *method which defines the needed window attributes, creates a window and calls* `State::new()`

Afterwards, the State struct will initialise the graphics pipeline and setup initial variables. The hardware API library is `wgpu` [11], allowing to create shaders, send buffers and create graphical pipelines. Figure 14 displays the startup sequence, defined in the `State::new()`.

```rs
// Get the size of window passed in
let size = window.inner_size();
// creates an instance specifying the backend (Vulkan, OpenGL, etc..)
let instance = wgpu::Instance::new(&wgpu::InstanceDescriptor {
    backends: (wgpu::Backends::PRIMARY),
    ..Default::default()
});
```

(a) *Creating the instance and retrieving window size*

```rs
// Creates a surface for wgpu to display onto, which maps onto the window screen.
let surface = instance.create_surface(window.clone()).unwrap();

// An adapter for hardware & software to communicate
let adapter = instance
    .request_adapter(&wgpu::RequestAdapterOptionsBase {
        power_preference: wgpu::PowerPreference::default(),
        force_fallback_adapter: false,
        compatible_surface: Some(&surface),
    })
    .await?;

// Get the device and queue variables, which act as the main driving force
let (device, queue) = adapter
    .request_device(&wgpu::wgt::DeviceDescriptor {
        label: None,
        required_features: wgpu::Features::empty(),
        required_limits: wgpu::Limits::defaults(),
        experimental_features: wgpu::ExperimentalFeatures::disabled(),
        memory_hints: Default::default(),
        trace: wgpu::Trace::Off,
    })
    .await?;

// Setup the surface rules & capabilities
let surface_caps = surface.get_capabilities(&adapter);
let surface_format = surface_caps
    .formats
    .iter()
    .find(|format| format.is_srgb())
    .copied()
    .unwrap_or(surface_caps.formats[0]);

let surface_config = wgpu::SurfaceConfiguration {
    usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
    format: surface_format,
    width: size.width,
    height: size.height,
    present_mode: wgpu::PresentMode::Mailbox, // uncapped FPS
    alpha_mode: surface_caps.alpha_modes[0],
    view_formats: vec![],
    desired_maximum_frame_latency: 2,
};
```

(b) *Surface creation, device setup, and configuration*

Figure 14: *Fragment of the* `new` *method responsible for the initialisation of* ***wgpu***

3.2 Mesh Formation

3.2.1 Vertex Structure and Data Flow

An ocean simulation works by creating a mesh, whose vertices are being vertically displaced inside the vertex shader. Figure 15 shows the Vertex lifecycle

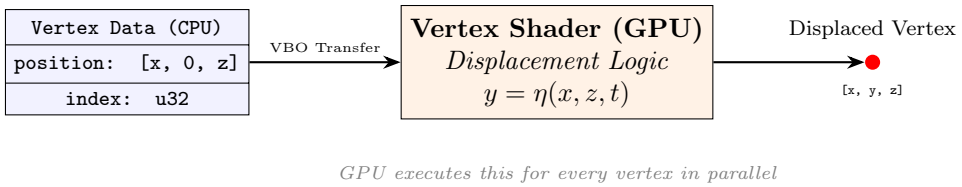


Figure 15: *The flow of a single vertex packet, where each vertex is treated as a discrete data packet containing its coordinates and a unique identifier*

3.2.2 Grid Generation and Indexing

The mesh, being a flat plane, is to be created, centred at $(0, 0)$ with physical size L and resolution defined by the number of subdivisions N. To ensure the mesh tiles and scales correctly, the simulation generates $N + 1$ vertices for N subdivisions, following the logic shown in Equation (34).

$$x = (\text{col} \times \text{step}) - \frac{L}{2}, \quad z = (\text{row} \times \text{step}) - \frac{L}{2} \tag{34}$$

The VBO stores the raw data of the vertices, while the Index Buffer (Object) defines how the vertices are connected to form fragments stored in winding order. Figure 16 shows how each square is made up of two triangles. Back-face culling is ensured by maintaining counter clockwise.

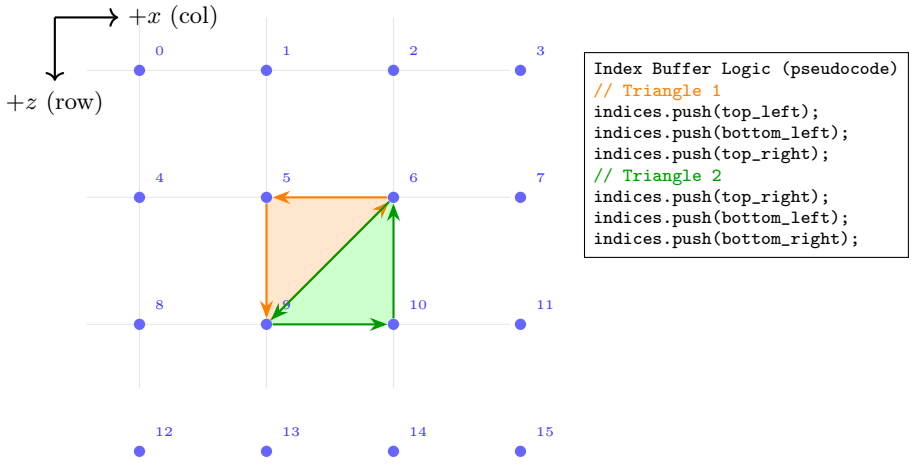


Figure 16: *Vertices are stored in a 1D array using row-major indexing (row × N + col)*

3.3 Vertex Displacement

3.3.1 Temporal Updates and Uniform Buffers

The vertex shader can now do displace the y coordinate of each vertex based on the height function. To create a moving sine wave, the GPU will need to access the time, therefore a Uniform Buffer is created, capable of storing and passing values from CPU with the help of bind group layouts and bind bind group bindings. Figure 17 shows the `update` method, called every frame.

```rs
// ...creation and initialisation of Time Uniform Buffer is not shown
pub fn update(&mut self) {
    // Use the standard library instant definition
    let now = Instant::now();
    // Acquire the difference
    let dt = (now - self.last_frame_time_instant).as_secs_f32();
    // Update latest frame instant
    self.last_frame_time_instant = now;
    // Internal method which increments time
    self.time_uniform.increment_time(dt);
    // Updated uniform has be cast onto the buffer to be sent to the GPU
    self.queue.write_buffer(
        &self.time_buffer,
        // 0 Is the byte offset
        0,
        bytemuck::cast_slice(&[self.time_uniform]),
    );
}
```

Figure 17: *A modified snippet from the code, where dt is calculated and the time inside the uniform buffer is incremented.*

After this, the render pipeline has to be configured, acting as a "global state" for the draw call. Now the vertex shader can be configured to accept a `TimeUniform` bind group, and apply the displacement onto the vertices as shown in Figure 18.

$$\eta(x, z, t) = A \cdot \sin(x \cdot f + t) + A \cdot \cos(z \cdot f + t) \tag{35}$$

```wgsl
// In this example, the uniform is not accounted to be aligned with 16 bytes.
struct TimeUniform {
    time: f32,
};

// The vertex buffer input
struct VertexInput {
    position: vec3<f32>,
}

// Matches the Bind Group created in Rust
@group(0) @binding(0) var<uniform> time_uniform: TimeUniform;

@vertex
fn vs_main(input: VertexInput) -> VertexOutput {
    var out: VertexOutput;

    // Simple displacement function for testing
    let amplitude = 2.0;
    let frequency = 0.5;
    let displacement = amplitude * sin(input.position.x * frequency + time_uniform.time)
                     + amplitude * cos(input.position.z * frequency + time_uniform.time);
    // Output the displaced vertex
    out.clip_position = vec4<f32>(input.position.x, displacement, input.position.z, 1.0);
    return out;
}
```

Figure 18: *To verify the functionality of the code, a simple sine wave across x axis and a cosine wave along z axis is applied following the Equation* (35)*, producing an output as seen in Figure 19 (with a texture stretched over the mesh)*

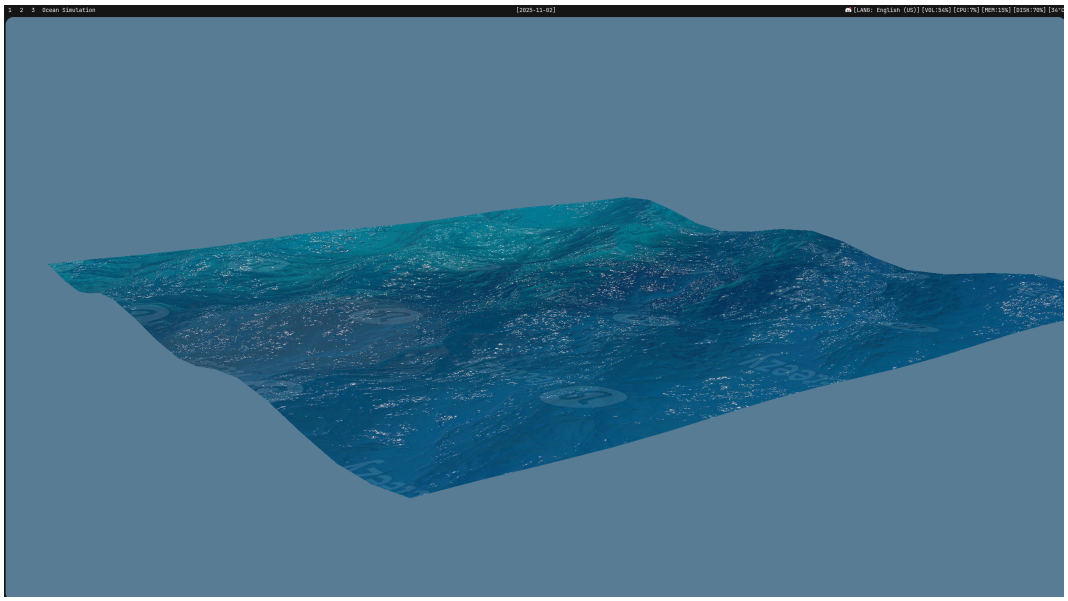


Figure 19: *Simple sine & cosine wave displacement of the vertex with an arbitriary texture stretched over it. Build v0.1.1*

3.4 The Skybox

3.4.1 Cubemapping and Depth Manipulation

The skybox, being a $1 \times 1 \times 1$ cube, but rendered at the furthest possible depth, is important to implement as it acts as the primarily light source used in reflections. The skybox is kept static as the camera moves, achieved by stripping the translation data from the view matrix before rendering the skybox as shown in Equation (36).

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{36}$$

3.4.2 Procedural Atmospheric Simulation

A procedural shader can be written to dynamically generate the sky, allowing for more freedom and customisability. The render pipeline is as shown in Figure 20.

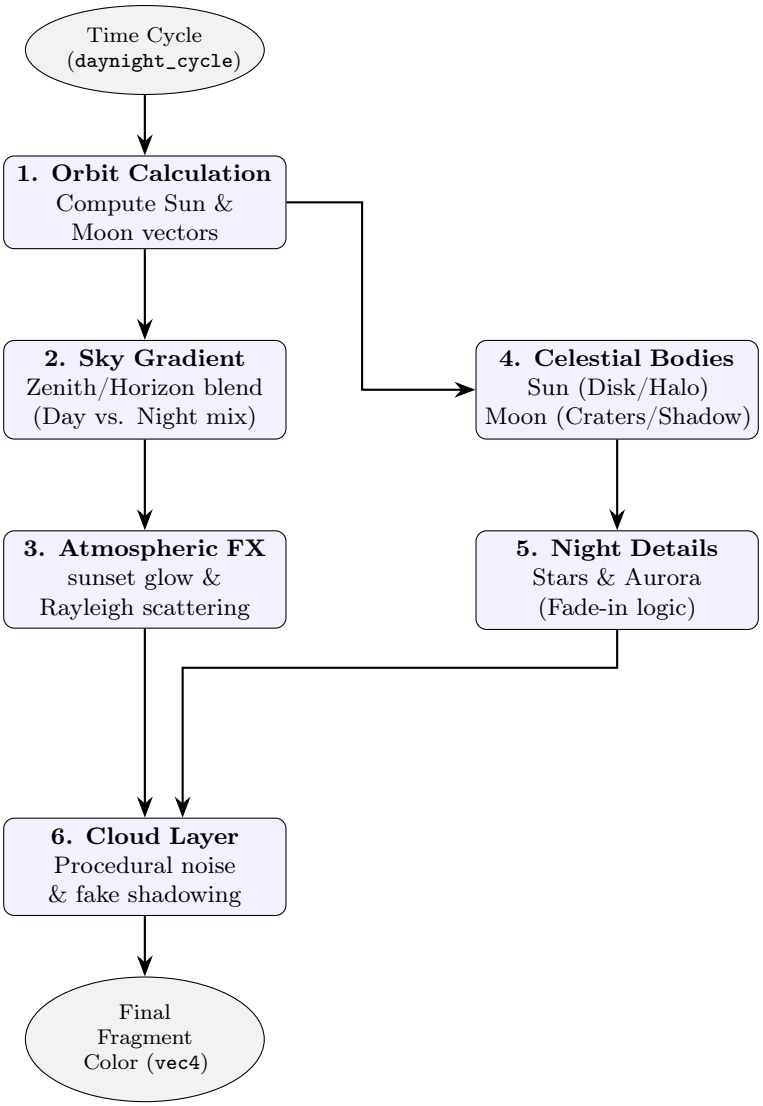


Figure 20: *Procedural Skybox Shader Pipeline. The shader calculates celestial positions before layering atmospheric gradients, light scattering, physical bodies, and clouds.*

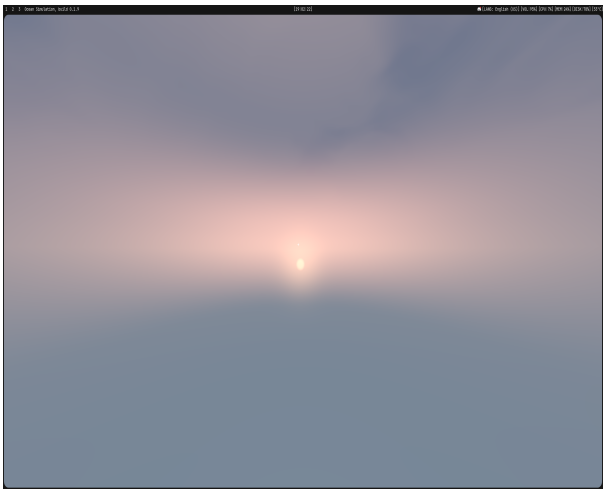
The `daynight_cycle` value is read from the settings struct, covered in Section **??**, used to calculate the angle and sun's position as shown in Equation (37), ϕ being the sun offset in the z direction.

$$\vec{d_L} = \langle \sin(\theta), \cos(\theta), \phi \rangle \tag{37}$$

Afterwards, atmospheric sky gradient is to be generated. The vertical gradient is done by blending the Zenith Sky colour and the Horizon sky colour based on the pixel height (`dir.y`), which are determined using Equation (38).

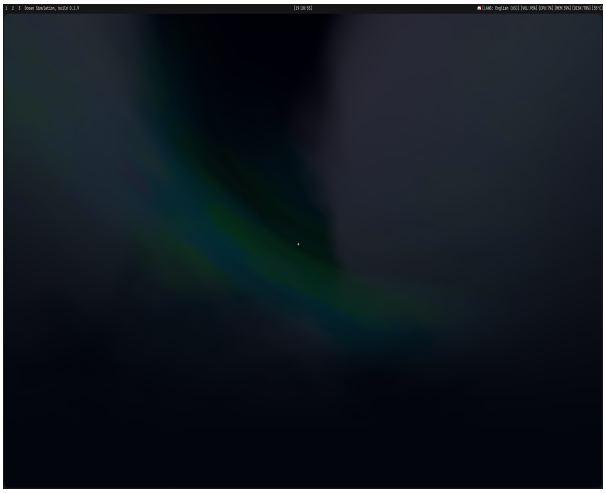
$$\begin{aligned} I &= \text{smoothstep}(-0.3,\, 0.3,\, \cos(\theta)) \\ Z &= \text{mix}(\text{night}_{\text{zenith}},\, \text{day}_{\text{zenith}},\, I) \\ H &= \text{mix}(\text{night}_{\text{horizon}},\, \text{day}_{\text{horizon}},\, I) \end{aligned} \tag{38}$$

Figure 21 shows the different sky effects implemented.

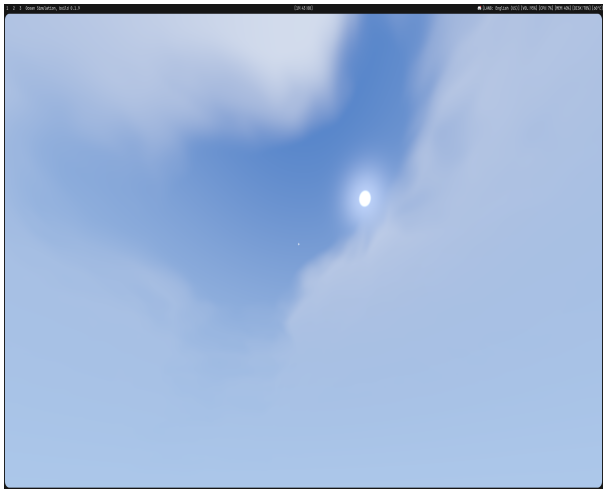


(a) *Sunset sky near the horizon showing Rayleigh scattering, producing orange-pink tint and horizontal glow*

(b) *Procedural moon surface generated via spherical UV mapping and fractional Brownian motion*



(c) *Aurora borealis formed using oscillating sine waves of varying frequencies with added noise*



(d) *Planar-mapped clouds projected onto a 2D plane, with density from noise functions and fake volumetric shadows computed using offset sampling towards the sun*

Figure 21: *Sky rendering features produced by the shader, including atmospheric scattering, celestial bodies, aurora effects, and volumetric cloud approximation*

3.5 Surface Normal Reconstruction

First, the method of central differences, covered in section 2.5.1, has to be implemented. As shown in Figure 22, the height map has to be sampled at four neighbouring points: $h_{i,j+1}$, $h_{i,j-1}$ and $h_{i+1,j}$, $h_{i-1,j}$.

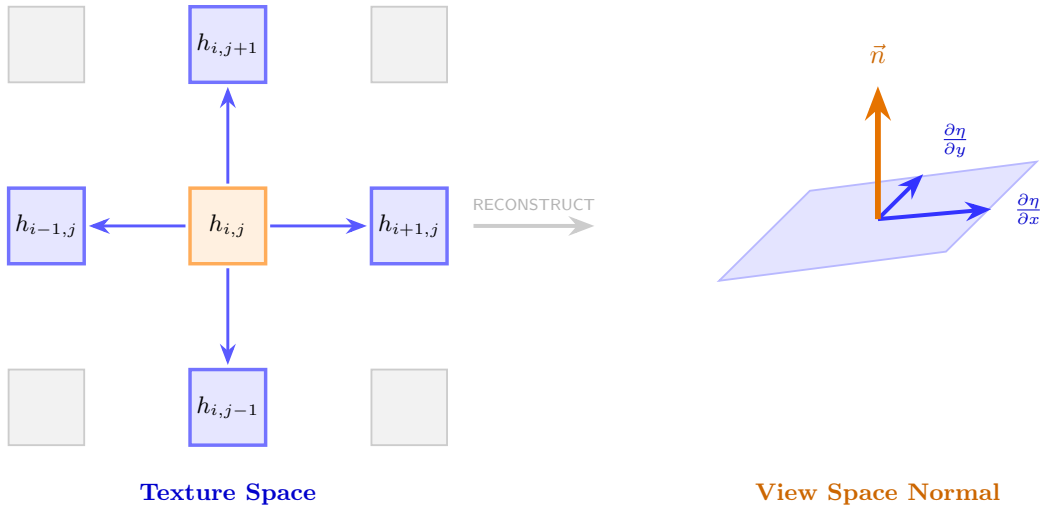


Figure 22: *Neighbour heights are sampled from the heightmap (shown on left) to compute the spatial gradient, which is then transformed into the surface normal $\vec{n}$ (right).*

From the slopes acquired in Equation (24), two tangential vectors can be constructed.

$$\vec{t_x} = \langle\, 1,\, \frac{\partial h}{\partial x},\, 0 \,\rangle, \quad \vec{t_z} = \langle\, 0\,,\, \frac{\partial h}{\partial z},\, 1 \,\rangle \tag{39}$$

To account for horizontal displacement the x and z components of the vectors are also influenced by the derivatives of the displacement maps (D_x, D_z), arriving at Equation (40).

$$\vec{t_x} = \langle\, 1 + D_x,\, \frac{\partial h}{\partial x},\, D_z \,\rangle, \quad \vec{t_z} = \langle\, D_x\,,\, \frac{\partial h}{\partial z},\, 1 + D_z \,\rangle \tag{40}$$

```wgsl
// ..amp, chop, subdivisions definition
let texel_uv = 1.0 / subdivisions;
// This defines the step size that we will take
let sample_offset_uv = texel_uv * 1.5;
// Master scale is set to 1024, meaning tilling happens every 512 units from the origin.
let run_meters = (sample_offset_uv * 2.0) * master_scale;

// now since packed -> single texture has .r as height, .g as dx and .b as dz
// this samples the neighbouring texels for data
let data_right = textureSampleLevel(texture_packed, sampler_ocean, uv + vec2(
    sample_offset_uv, 0.0), 0.0);
let data_left = textureSampleLevel(texture_packed, sampler_ocean, uv - vec2(
    sample_offset_uv, 0.0), 0.0);
let data_up = textureSampleLevel(texture_packed, sampler_ocean, uv + vec2(0.0,
    sample_offset_uv), 0.0);
let data_down = textureSampleLevel(texture_packed, sampler_ocean, uv - vec2(0.0,
    sample_offset_uv), 0.0);

// partial derivatives using central difference
let dDx_h = (data_right.r - data_left.r) * amp / run_meters;
let dDz_h = (data_up.r - data_down.r) * amp / run_meters;

let dDx_du = (data_right.g - data_left.g) * chop / run_meters;
let dDx_dv = (data_up.g - data_down.g) * chop / run_meters;

let dDz_du = (data_right.b - data_left.b) * chop / run_meters;
let dDz_dv = (data_up.b - data_down.b) * chop / run_meters;

// Acquire the tangent vectors with the horizontal displacements
let tangent_u = vec3<f32>(1.0 + dDx_du, dDx_h, dDz_du);
let tangent_v = vec3<f32>(dDx_dv, dDz_h, 1.0 + dDz_dv);

// The normal will be the cross product
let normal_geometry = normalize(cross(tangent_v, tangent_u));
```

Figure 23: *Modified segment of the fragment shader, responsible for calculating the surface normals*

Mathematically the normal is equivalent to taking the cross product between two tangent vectors as shown in Equation (41).

$$\vec{n} = \vec{t_z} \times \vec{t_x} = \det \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ 0 & \frac{\partial h}{\partial z} & 1 \\ 1 & \frac{\partial h}{\partial x} & 0 \end{vmatrix} \tag{41}$$

By expanding the determinant Equation (42) is acquired.

$$\begin{aligned} \vec{n} &= \hat{i}\left[0 - \frac{\partial h}{\partial x}\right] - \hat{j}\left[0 - 1\right] + \hat{k}\left[0 - \frac{\partial h}{\partial z}\right] \\ \vec{n} &= \langle -\frac{\partial h}{\partial x}, 1, -\frac{\partial h}{\partial z} \rangle \end{aligned} \tag{42}$$

3.6 Tessendorf's Model Implementation

The simulation following the Tessendorf's model can be broken down into multiple stages as shown in Figure 24.

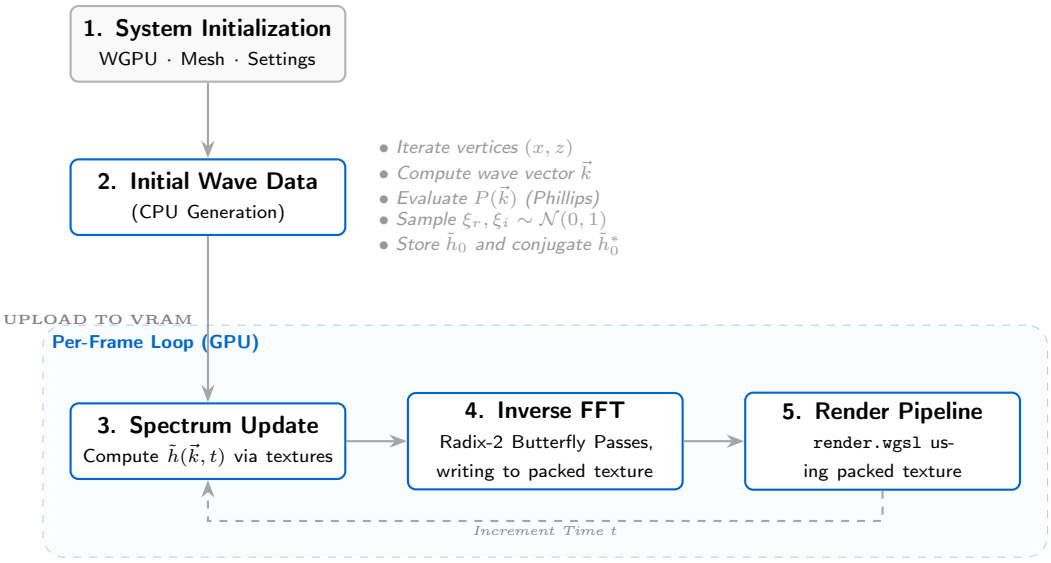


Figure 24: *The Tessendorf Ocean Simulation Pipeline, detailing the CPU initialization and GPU Fourier computations.*

3.6.1 Initial Data Generation

First, initial data is generated once at startup. The code will call `InitialData::generate_data()` which will loop through each vertex in the mesh, create a new instance of `InitialData` and push it to a vector. Where the data is acquired by first constructing the vector $\vec{k} = \langle k_x, k_y \rangle$, as shown in Figure 25.

```rs
// m is the rows, while n is the columns
let k_x = (2.0 * PI * (f32(n) - f32(subdivisions) / 2.0)) / fft_size;
let k_y = (2.0 * PI * (f32(m) - f32(subdivisions) / 2.0)) / fft_size;
let k_vec = [k_x, k_y];

if k_vec == [0.0, 0.0] {
    // return a zero-d out struct
}
```

Figure 25: *Modified section of the code responsible for generating the vector k from position in the grid. The* `- f32(subdivisons) / 2.0` *statement ensures that the grid is centered around the origin, where the vector with components* $(0, 0)$ *sits in the middle*

The Phillips spectrum uses this $\vec{k}$ value to generate a value, where the full equation for acquiring the Phillips Spectrum Value is as shown in Equation (43).

$$P(\vec{k}) = A \cdot (\hat{k} \cdot \hat{W})^2 \cdot \exp\left(\frac{-1}{||\vec{k}||^2 L^2}\right) \cdot \exp\left(-||\vec{k}||^2 l^2\right) \cdot ||\vec{k}||^{-4} \tag{43}$$

```rs
// ...k_vec is converted into a cgmath vector, k_len calculated
// The code checks if the length is higher than the allowed amount specified in the
    settings. This limits unrealistic waves breaking through.
if k_len > max_w || k_len < 0.001 {
    return 0.0;
}

let k2 = k_len * k_len;
let k4 = k2 * k2;
let k_hat = k.normalize();

// Wind vectors setup, provided by the ocean settings
let w: Vector2<f32> = wind_vector.into();
let w_len = w.magnitude();
let w_hat = w.normalize();

// The directional weighting
let align = cgmath::dot(k_hat, w_hat);
// Squaring the value, but also allowing negative values to break through, allowing for
    realism.
let align2 = if align > 0.0 {
    align.powi(2)
} else {
    align.powi(2) * 0.07
};

// l dampening value
let l = w_len * w_len / 9.81;
let l2 = l * l;

let exp_term = f32::exp(-1.0 / (k2 * l2));
let damp = f32::exp(-k2 * l_small * l_small);

// The 0.001 is added to avoid a devision by zero error
(align2 * amplitude * exp_term * damp) / (k4 + 0.001)
```

Figure 26: *Modified section of the* `::generate_phillips_spectrum_value()` *method inside of* `InitialData` *struct. The directional weighting is calculated using the dot product from the* `cgmath` *package*

Two random values, ξ_r, ξ_i, are drawn from the Gaussian field, using an `ocean_seed` to keep the ocean waves consistent upon rebuilding data as shown in Figure 27.

```rs
// ... seed provided from the settings struct
let mut rng = StdRng::seed_from_u64(seed as u64);

// Generate two values
let xi_r = Self::box_muller(rng.random::<f32>().max(1e-6), rng.random::<f32>());
let xi_i = Self::box_muller(rng.random::<f32>().max(1e-6), rng.random::<f32>());
```

Figure 27: *Sample of the code showing the generation of the two random values for the initial data. Where* `box_muller` *is a new method depicting the Box Muller transformation required to transform a uniform field to a Gaussian Distribution, as seen in Figure 28.*

```rs
pub fn box_muller(u1: f32, u2: f32) -> f32 {
    (-2.0 * u1.ln()).sqrt() * (2.0 * std::f32::consts::PI * u2).cos()
}
```

Figure 28: *Box-Muller transformation to form a Gaussian distribution*

Next, frequency domain is constructed by first computing `let sqrt_phk = (phk / 2.0).sqrt()`, where further multiplying by the two random numbers will result in a real component and an imaginary component.

Note: Bit Shift

Bit shifting is a low level operation performed in computing, involving moving the bits in a binary number a specified number of times in a particular direction, effectively performing multiplication and division by a factor of 2, as shown in Figure 29.

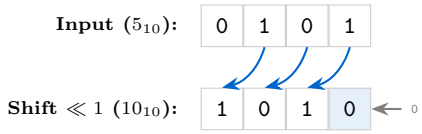


Figure 29: *A logical left bit shift ($\ll 1$). Each position move left doubles the value.*

The future FFT butterfly passes will require twiddle factors. The $\exp\left(\frac{-2\pi ik}{N}\right)$ definition of a twiddle factor can be expressed in terms of sin and cos, since $e^{i\theta} = \cos\theta + i\sin\theta$, which will harm the efficiency of the program, by being constantly recomputed on the GPU. Therefore, it is standard practice to pre-compute the twiddle factors and pass it into the GPU as shown in Figure 30.

```rs
// If the subdivisions is 2048, the max butterfly passes an FFT shader can do is 2^11,
    therefore a logarithm is taken
let max_stages = fft_subdivisions.ilog2();
let mut twiddles = Vec::<[f32; 2]>::with_capacity((fft_subdivisions - 1) as usize);

// For each stage, a twiddle factor has to be generated
for stage in 0..max_stages {
    // Left bit shift by the number of stages.
    let s = 1u32 << stage;
    for offset in 0..s {
        // Generate the angle required
        let angle = 2.0 * std::f32::consts::PI * (offset as f32) / (2.0 * s as f32);
        // Extract sine and cosine values and push it onto the vector.
        twiddles.push([angle.cos(), angle.sin()])
    }
}
// In rust, the 'return' keyword is not needed, if at the end of a function, but suits
    nicely here
return twiddles;
```

Figure 30: *Twiddle code generation computed prematurely on the CPU*

The angle calculation can be rearranged as shown in Figure 44, where $N_{\text{local}} = 2s$ is the size of the sub-problem at that stage.

$$\theta = \frac{2\pi \cdot \text{offset}}{2s} = \frac{2\pi \cdot \text{offset}}{N_{\text{local}}} \tag{44}$$

In the end, values for `k_vec`, `h_0` and `w` along with twiddle factors are obtained.

3.6.2 Time Evolution

Storage textures are used to send the initial data to the compute shaders and since it's read-only, the data will not change between frames, only evolved on the GPU. Compute shaders work differently to normal shaders by not producing an output to the screen, and are launched in a grid of workgroups, each containing threads, working in parallel, allowing for fast and efficient calculations. The compute shader entry point `@compute @workgroup_size(16,16)` defines 256 threads per workgroup, and with `pass.dispatch_workgroups(N/16, N/16, 1)` the GPU covers the entire $N \times N$ texture so each pixel is processed in parallel. Since each FFT butterfly stage operates on independent input pairs, the GPU computes all N^2 elements simultaneously, unlike the CPU which would process them largely serially.

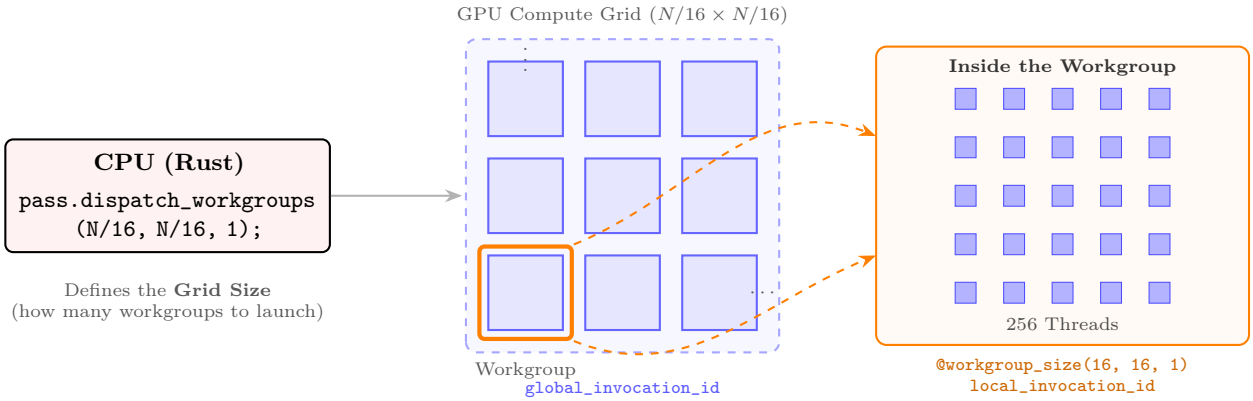


Figure 31: *The Compute Dispatch hierarchy. The CPU defines the total number of workgroups to be processed (the Grid), while the Shader code defines the internal density of threads per workgroup. This 2D mapping allows the GPU to process an $N \times N$ texture by assigning exactly one thread to every pixel.*

The spectrum has to be evolved in time, run once per frame, where the code takes a static `h_0` passed in at initialisation and uses the the Equation (12) to evolve it. The shader code will be as shown in Figure 32.

```wgsl
let w_i = initial_data[index].angular_frequency;
let wt = w_i * camera.time * ocean_settings.time_scale;
let cos_wt = cos(wt);
let sin_wt = sin(wt);

// Using Euler's formula to represent the iwt in terms of sine and cosine
// Could possibly also be pre-computed...
// e^{iwt} = cos(wt) + i*sin(wt) = <cos(wt), sin(wt)>
let exponent = vec2<f32>(cos_wt, sin_wt); // e^{iwt}
let exponent_neg = vec2<f32>(cos_wt, -sin_wt); // e^{-iwt}

// Now do some complex multiplication.
let h_tilda: vec2<f32> = (complex_multiplication(h_0, exponent)
                        + complex_multiplication(h_0_mirrored_conjugate, exponent_neg));
```

Figure 32: *Modified section of the* `wgsl` *code responsible for evolving the time spectrum*

Now the horizontal displacements dx and dz have to be calculated. The calculation involved multiplication by i, but it can be modelled as a rotation of the vector by $\pi/2$. Code responsible is shown in Figure 33.

```wgsl
// Multiplying by i: (a + bi) * i = -b + ai
let h_dx = vec2(-h_tilda.y * k_norm.x, h_tilda.x * k_norm.x);
let h_dz = vec2(-h_tilda.y * k_norm.y, h_tilda.x * k_norm.y);
```

Figure 33: *Modified section of the code detailing the* `h_dx` *and* `h_dz` *calculation.*

After the spectrum has been updated, the three complex signals: `h_tilda`, `h_dx` and `h_dz` are generated; they have to be packed into textures. Each texture however, is only able to store 4 values (r, g, b, a). Which correlates to storing only 2 complex numbers per texture, implying the need of 2 separate textures, as shown in Figure 34.

```wgsl
// R,G -> h_tilda. B,A -> h_dx
textureStore(dst_h_dx, vec2<i32>(id.xy), vec4<f32>(h_tilda, h_dx));
// R,G -> h_dz, B,A -> empty
textureStore(dst_dz, vec2<i32>(id.xy), vec4<f32>(h_dz, 0.0, 0.0));
```

Figure 34: *Modified section of the code, representing how the values are laid out between the two textures.*

3.6.3 IFFT Implementation

To recap Section 2.4.1, but to recap, for an N-size IFFT, the number of passes required would be $\log_2(N)$. First the horizontal ones, then the vertical. Rather than implementing a separate IFFT, same radix-2 algorithm can be used with only two changes required: a positive twiddle sign change and the normalisation $1/N$. In practice, this means the four steps discussed earlier can be reduced as shown in Figure 35.

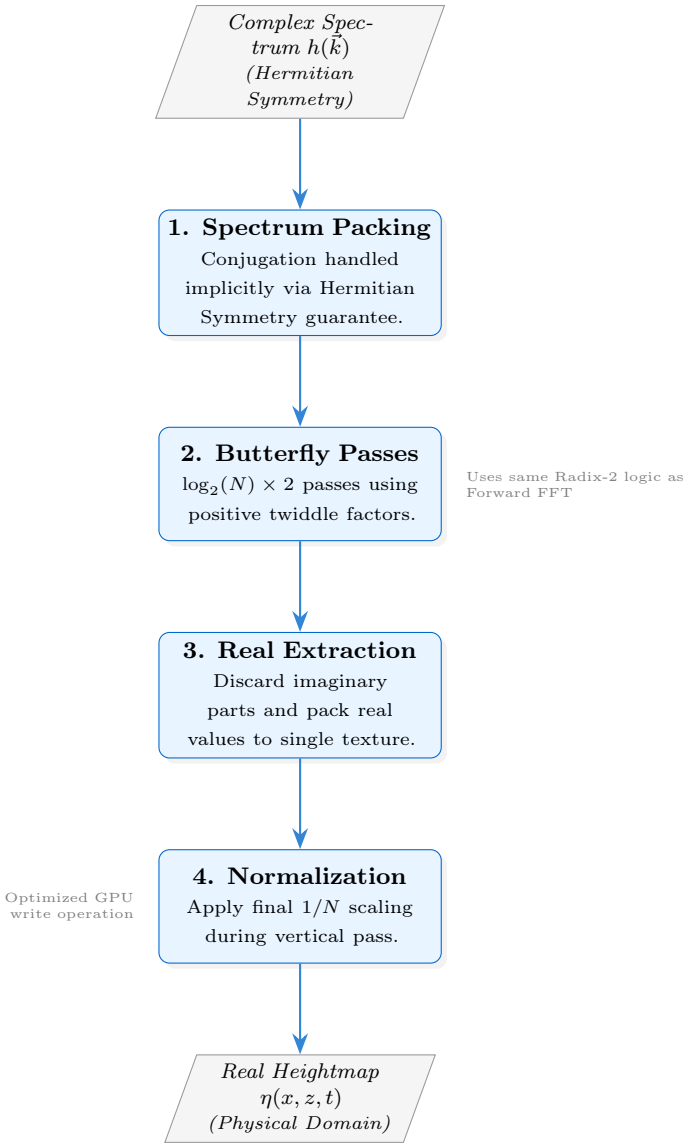


Figure 35: *The optimized IFFT pipeline for ocean surface generation, illustrating the reduction from a standard inverse transform to a symmetry-aware butterfly sequence.*

However, a texture cannot be read from and written to within the same call, as this would bring up race conditions and lots of bugs. This calls for a simple solution, the ping-pong architecture. This involves the use of two textures, one for read and one for write, where with each successive pass the textures alternate their roles. Figure 36 represents this process.

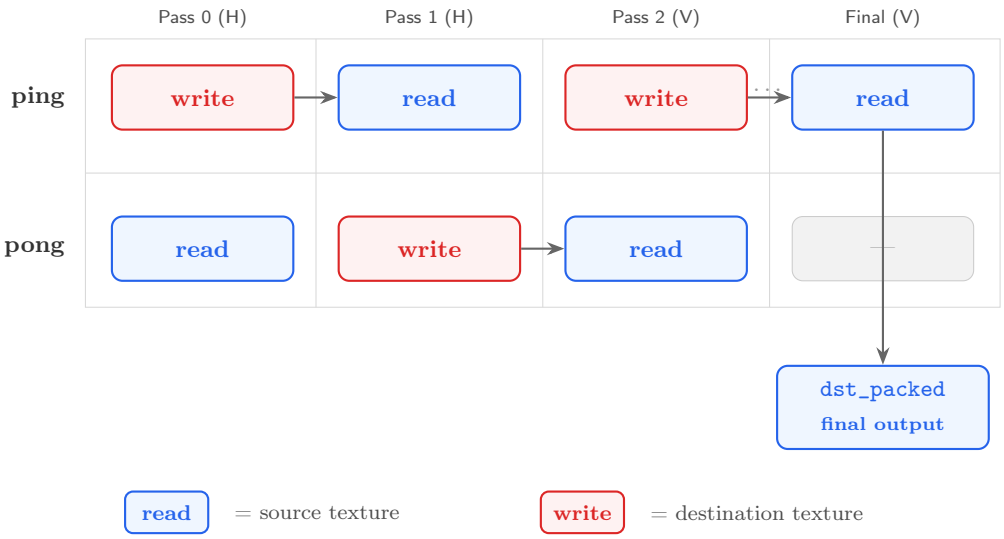


Figure 36: *Ping-pong texture swapping across butterfly passes.*

All the bind groups are prebuilt with the textures already wired in opposing roles, meaning there is zero allocation overhead at runtime, as shown in Figure 37.

```rs
let mut current_reads_from_ping = true;

for i in 0..passes {
    let mut pass = encoder.begin_compute_pass(&Default::default());
    pass.set_pipeline(&self.fft_compute_pipeline);

    // Select pre-built bind group based on which texture is currently source
    let bind_group = if current_reads_from_ping {
        &cascade.bind_groups_ping[i]   // reads ping, writes pong
    } else {
        &cascade.bind_groups_pong[i]   // reads pong, writes ping
    };

    pass.set_bind_group(1, bind_group, &[]);
    pass.dispatch_workgroups(N / 16, N / 16, 1);

    // Flip for next pass
    current_reads_from_ping = !current_reads_from_ping;
}

// Track which texture holds the final result for the renderer
cascade.output_is_ping = !passes.is_multiple_of(2);
```

Figure 37: *Modified section of* `compute_fft`*, showing the ping-pong dispatch loop and the* `output_is_ping` *flag switching.*

Every pass dispatches the compute shader, handling both horizontal and vertical directions, swapping the required axis as shown in Figure 38.

```wgsl
let t = select(x, y, config.is_vertical == 1u);
let other = select(y, x, config.is_vertical == 1u);
```

Figure 38: *A single shader handles both horizontal and vertical passes via the* `is_vertical` *flag.*

Left logical bit shift sued to find threads own position inside butterfly structure using $s = 2^{\text{stage}}$ as shown in 39.

```wgsl
// Sub-problem size using logical left bit shift.
let s = 1u << config.stage;

// Threads position gets converted into indices
let group_idx = t / (2u * s); // which butterfly group
let butterfly_idx = t % (2u * s); // position within that group
let pair = butterfly_idx / s; // 0 is top wing and 1 is bottom wing
let offset = butterfly_idx % s; // index within the half-group

// The two elements this thread will combine
let base_idx = group_idx * (2u * s) + offset;
let idx0_t = base_idx; // top element
let idx1_t = base_idx + s; // bottom element
```

Figure 39: *Modified section of the code, responsible for decomposing the global ID into its butterfly position.*

Figure 40 illustrates the process in which the two outputs can be generated depending on the `pair` and how the twiddle factor comes into play.

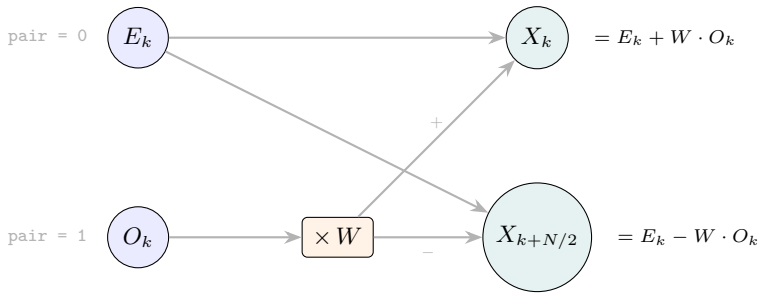


Figure 40: *A single radix-2 DIT butterfly, where the twiddle factor W rotates the lower element in the complex plane before the addition and subtraction that produces the two outputs that can be seen*

This diagram can be compared with the `wgsl` code as shown in Figure 41.

```wgsl
let rotated_h_dx_xy = complex_multiplication(twiddle, src1_h_dx.xy);
let rotated_h_dx_zw = complex_multiplication(twiddle, src1_h_dx.zw);
let rotated_dz_xy = complex_multiplication(twiddle, src1_dz.xy);

if (pair == 0u) {
    // Top, E + W*O
    res_h_dx = vec4(src0_h_dx.xy + rotated_h_dx_xy, src0_h_dx.zw + rotated_h_dx_zw);
} else {
    // Bottom, E - W*O
    res_h_dx = vec4(src0_h_dx.xy - rotated_h_dx_xy, src0_h_dx.zw - rotated_h_dx_zw);
}
```

Figure 41: *Modified section of the code, showing the butterfly addition and subtraction for* `h_tilda` *and* `h_dx`*. The same pattern repeats identically for* `h_dz`*.*

Each stage s contributes 2^s entries in the buffer, so the starting index for s in the array would correspond to $2^s - 1$ inside the twiddle array as seen in 42.

```wgsl
// base is the starting index for the current stage in the flat array
let base = (1u << config.stage) - 1u;
let twiddle = twiddle_array[base + offset];
```

Figure 42: *Twiddle factor lookup inside the code. Where the positive angle makes the transform an inverse.*

On the last vertical pass, the shader will apply the $1/N$ normalisation and discard the imaginary components, allowing to write the real components directly into a single `packed_texture`, saving extra VRAM, as shown in Figure 43.

```wgsl
if (config.stage == ocean_settings.pass_num - 1u && config.is_vertical == 1u) {
    // normalisation
    if (pair == 0u) {
        res_h = (src0_h_dx.xy + rotated_h_dx_xy) / f32(n);
        res_dx = (src0_h_dx.zw + rotated_h_dx_zw) / f32(n);
        res_dz = (src0_dz.xy + rotated_dz_xy) / f32(n);
    } else {
        res_h = (src0_h_dx.xy - rotated_h_dx_xy) / f32(n);
        res_dx = (src0_h_dx.zw - rotated_h_dx_zw) / f32(n);
        res_dz = (src0_dz.xy - rotated_dz_xy) / f32(n);
    }
    // Only .x (real part) is stored
    textureStore(dst_packed_final, vec2<i32>(id.xy),
                 vec4<f32>(res_h.x, res_dx.x, res_dz.x, 1.0));
}
```

Figure 43: *Modified section of the final butterfly pass, where the normalisation is done and the real displacement is packed into a single* `rgba16float` *texture.*

The channel layout of the packed output texture is as shown in Figure 44.

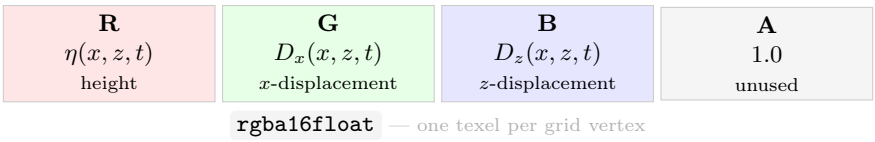


Figure 44: *layout of* `packed_texture`*. Three independent IFFTs (height, dx, dz) share the same butterfly passes and are packed into a single texture, to be read by the vertex shader to render.*

The IFFT is now complete, however the vertex shader now has to use this data, meaning the `packed_texture` has to be passed to the main render pipeline. Figure 45 shows the vertex shader responsible for applying the displacements.

```wgsl
// ... uv and scaling calculations
let world_pos = model.position.xyz;
let sample_uv = world_pos.xz / 1000.0;

// Sample the texture at mip level 0
let displacement = textureSampleLevel(texture_packed, sampler_ocean, sample_uv, 0.0);
let amp = ocean_settings.amplitude_scale;
let chop = ocean_settings.chop_scale;

let h = displacement.r * amp;
let dx = displacement.g * chop;
let dz = displacement.b * chop;

let displaced_pos = vec3(
    model.position.x + dx,
    h,
    model.position.z + dz
);
out.tex_coords = sample_uv;
out.world_pos = displaced_pos;
```

Figure 45: *The vertex shader code responsible for sampling the texture and applying the displacement values onto the vertex*

3.7 PBR Implementation

As discussed earlier in Section 2.5.2, this model is based on the statistical approach that the surface of any object is made up of many microfacets.

3.7.1 Mapping Equations to Shader Code

The Fresnel function, geometry function and normal distribution function, F, G, D respectively are now modelled directly to `wgsl` code as seen in Figure 46.

```wgsl
// The dot products are being passed in as parameters to avoid re-calculation in each
    function.
fn normal_func(n_dot_half: f32, alpha: f32) -> f32 {
    let alpha2 = alpha * alpha;
    let denom = (n_dot_half * n_dot_half * (alpha2 - 1.0) + 1.0);
    return alpha2 / (pi * denom * denom);
}

fn geometry_func(n_dot_view: f32, n_dot_light: f32, alpha: f32) -> f32 {
    // Smiths function accept the dot product, as well as alpha value
    return smiths(n_dot_view, alpha) * smiths(n_dot_light, alpha);
}
// A helped function for the geometry function
fn smiths(dot_product: f32, alpha: f32) -> f32 {
    let k = ((alpha + 1.0) * (alpha + 1.0)) / 8.0;
    let denom = dot_product * (1.0 - k) + k;
    // A small value added on to avoid a division by zero error.
    return dot_product / (denom + 1e-6);
}

fn fresnel_func(view_dot_half: f32, f_0: f32) -> f32 {
    let scale = pow(1.0 - view_dot_half, 5.0);
    return f_0 + (1.0 - f_0) * scale;
}
```

Figure 46: *A modified section of the code, showing the mathematical functions for fresnel, geometry and normal distribution.*

The `alpha` value is calculated dynamically as shown in Figure 47, using the base value of `roughness`, and is scaled by multiple parameters, such as the distance and foam coverage.

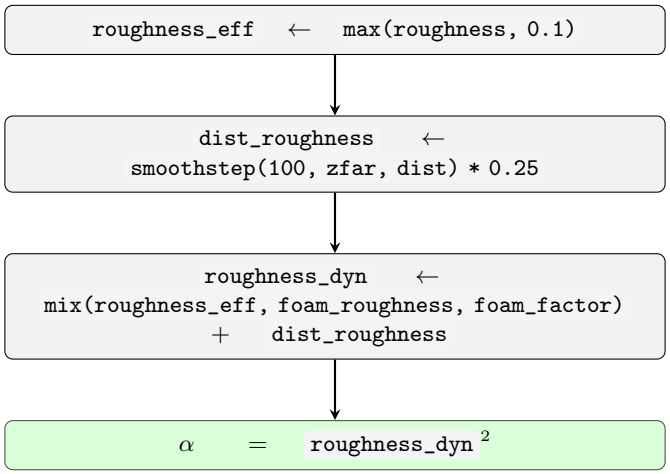


Figure 47: *Dynamic calculation of* `alpha`.

As previously shown in Equation 26, these functions combined make up the Cook-Torrence BRDF, and can be represented in code as shown in Figure 48.

```wgsl
fn cook_torrance(n_dot_light: f32, n_dot_view: f32, n_dot_half: f32, alpha: f32, fresnel:
    f32) -> f32 {
    let normal_function = normal_func(n_dot_half, alpha);
    let geometry_function = geometry_func(n_dot_view, n_dot_light, alpha);
    // Small value added to stop division by zero error.
    return (normal_function * geometry_function * fresnel) / (4.0 * n_dot_light *
    n_dot_view + 1e-4);
}
```

Figure 48: *A modified section of the code, displaying the Cook-Torrence BRDF implementation in* `wgsl`.

This function is used to acquire the specular intensity value which is used in combination with the light colour and other various dampeners. These dampeners include `specular_scale`, `reflection_dampener` and `1.0 - foam_factor`. The `reflection_dampener` terms are required to limit the amount of reflections occurring when a wave is near the point of collapse or creases, calculated by interpolating the Jacobian between the minimum and maximum reflection value. The Jacobian will be discussed more in Section 3.8.1.

3.7.2 Sky Reflections and Tone Mapping

The Fresnel term quantifies the physical value of reflectance, and is used in two distinct cases. One of them is being passed into the BRDF model using the `view_dot_half` parameter, governing specular highlights. While the second value is calculated using the `n_dot_view` and is used to find how much of the procedural sky will get blended over the water colour. This is what controls the mirror-like reflections seen in the water.

Afterwards, ambient light is applied which is sampled from the sky's current sky colour. After all of the factors are combined, the brightness may often exceed 1.0. Clamping the brightness would cause lose all the detailed highlights. Therefore, Academy Colour Encoding System (ACES) [24] filmic tone mapping curve will be used as seen in Figure 49. Afterwards, a gamma correction is applied by raising colour to power of 1/2.2.

```wgsl
fn aces_tone_map(color: vec3<f32>) -> vec3<f32> {
    let a = 2.51;
    let b = 0.03;
    let c = 2.43;
    let d = 0.59;
    let e = 0.14;
    // ACES approximation is used
    return clamp((color * (a * color + b)) / (color * (c * color + d) + e),
                 vec3<f32>(0.0), vec3<f32>(1.0));
}
```

Figure 49: *The* `wgsl` *implementation of the Narkowicz 2015 ACES approximation.*

The complete flow of the new PBR system is summarised in Figure 50.

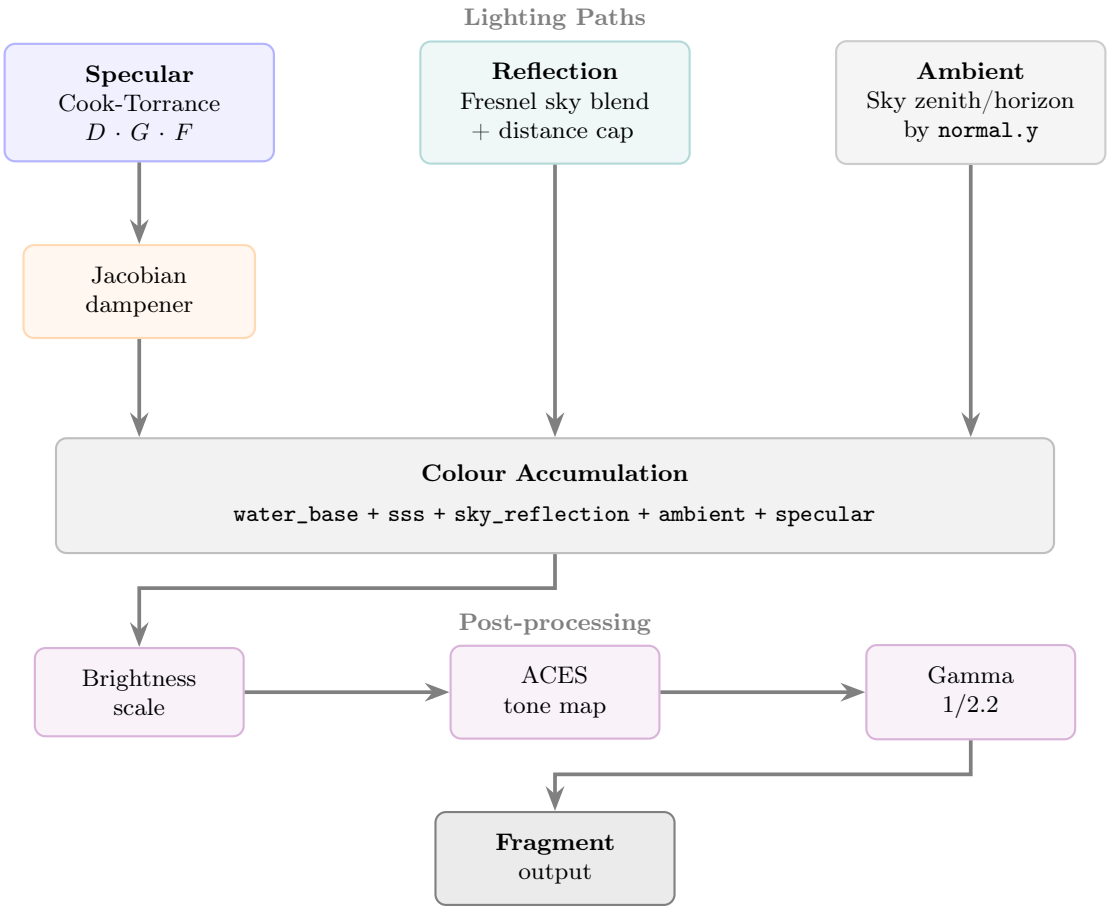


Figure 50: *Three light paths add up onto the water base colour and then get post-processed.*

3.8 Jacobian Foam

Waves constantly battle against one another, often clashing and creasing in various ways. When this happens, often foam is generated. This can be modelled into our simulation by first identifying these positions where the foam should generate, then advecting it through time.

3.8.1 The Jacobian

To identify where the foam should be generated, the Jacobian value is used. The Jacobian, more specifically the Jacobian matrix, consists of first order partial derivatives of a vector function, This represents the best linear approximation at a given point in space time, acting as a multivariable equivalent of a derivative. Definition being shown in Equation (45).

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{45}$$

By taking the determinant of the Jacobian matrix we can identify the disturbed areas of the surface by compression or stretching. Recall that earlier the position of a vertex was defined as shown in Equation (46).

$$v = \langle\, u + D_x(u,v),\ \eta(u,v),\ v + D_z(u,v) \,\rangle \tag{46}$$

This means that the Jacobian can be applied onto the position of the vertex and its determinant can be calculated as shown in Equation (47).

$$\begin{aligned}\mathbf{J} &= \begin{bmatrix} \frac{\partial}{\partial u}\left[u + D_x(u,v)\right] & \frac{\partial}{\partial v}\left[u + D_x(u,v)\right] \\ \frac{\partial}{\partial u}\left[v + D_z(u,v)\right] & \frac{\partial}{\partial v}\left[v + D_z(u,v)\right] \end{bmatrix} \\ &= \begin{bmatrix} 1 + \frac{\partial D_x(u,v)}{\partial u} & \frac{\partial D_x(u,v)}{\partial v} \\ \frac{\partial D_z(u,v)}{\partial u} & 1 + \frac{\partial D_z(u,v)}{\partial v} \end{bmatrix} \\ J = \det \mathbf{J} &= \left(1 + \frac{\partial D_x}{\partial u}\right)\left(1 + \frac{\partial D_z}{\partial v}\right) - \frac{\partial D_x}{\partial v}\frac{\partial D_z}{\partial u}\end{aligned} \tag{47}$$

Where when $J = 1$ the surface is undisturbed, when $J > 1$ the surface is being stretched, while when $J \to 0$ the vertices are converging to a single point. J is calculated using the central differences method as shown in Figure 51.

```wgsl
// .. sample_offset_uv and run_meters calculation
// First the points are sampled, extracting the data.
let data_right = textureSampleLevel(texture_packed, sampler_ocean, uv + vec2(
sample_offset_uv, 0.0), 0.0);
let data_left = textureSampleLevel(texture_packed, sampler_ocean, uv - vec2(
sample_offset_uv, 0.0), 0.0);
let data_up = textureSampleLevel(texture_packed, sampler_ocean, uv + vec2(0.0,
sample_offset_uv), 0.0);
let data_down = textureSampleLevel(texture_packed, sampler_ocean, uv - vec2(0.0,
sample_offset_uv), 0.0);

// The partial derivatives are computed and scaled to appropriate size
let dDx_du = (data_right.g - data_left.g) * chop / run_meters;
let dDx_dv = (data_up.g - data_down.g) * chop / run_meters;

let dDz_du = (data_right.b - data_left.b) * chop / run_meters;
let dDz_dv = (data_up.b - data_down.b) * chop / run_meters;

// Final  jacobian calculation
let jacobian = (1.0 + dDx_du) * (1.0 + dDz_dv) - (dDx_dv * dDz_du);
```

Figure 51: *Modified section of the code from the main fragment shader calculating the Jacobian using central differences method.*

The result is shown in Figure 52 where the ocean surface has the full spectrum of colours.

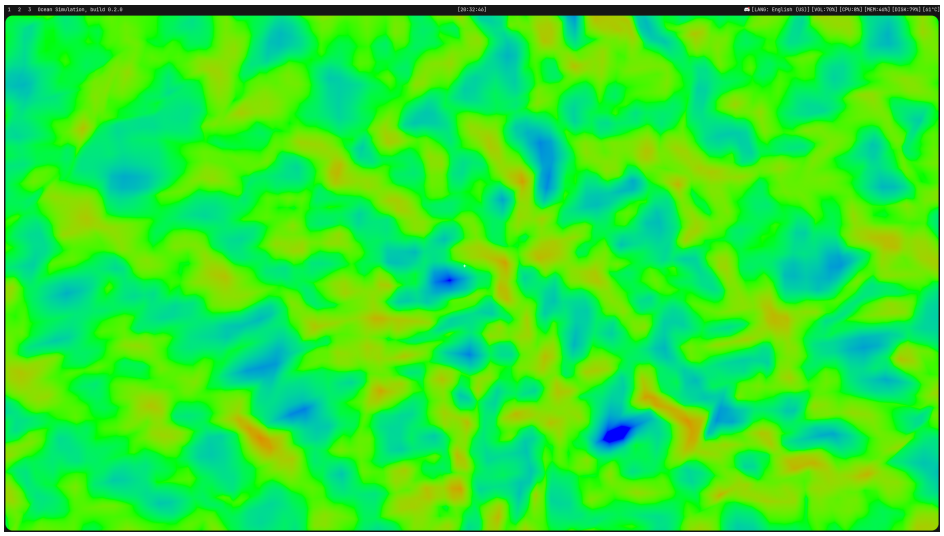


Figure 52: *To double check the values, the Jacobian can be mapped to the colour spectrum, where green represents $J \approx 1$, red represents when $J \to 0$ and blue is shown when $J > 1$ and outputted as the primary fragment colour.*

3.8.2 Foam Generation and Advection

Foam generation and advection will work by utilising texture to store the foam strength, and will be handled by a separate compute shader using a similar ping-pong model to the IFFT implementation. Now the calculation shown in Figure 51 can be translated to the compute shader code as shown in Figure 53.

```wgsl
// Breaking value calculated using a threshold and jacobian
let breaking = clamp(ocean_settings.foam_threshold - jacobian, 0.0, 1.0);
// Generation power is calculated
let generated = pow(breaking, ocean_settings.foam_power);

// Previous value from the texture is read
let prev = textureLoad(foam_texture_read, global_id.xy).r;
// We decay the old data
let decayed = prev * ocean_settings.decay_factor;

// Write new data
let result = max(decayed, generated);
textureStore(foam_texture_write, global_id.xy, vec4<f32>(clamp(result, 0.0, 1.0), 0.0,
    0.0, 1.0));
```

Figure 53: *Modified section of the* `compute_foam` *function inside the compute shader*

This shader is called every frame, storing the foam strength inside the texture. Afterwards, another shader moves the foam across the ocean surface over time, making it flow with the water instead of sitting still. This is referred to as advection, the process of transporting quantity along a velocity field. To achieve this result, the velocity of each texel is read directly from the D_x and D_z stored in the green and blue channel of the `packed_texture` respectively. The displacement of the foam in pixels can be then calculated as shown in Equation (48).

$$s_{\text{px}} = v \cdot \text{foam\_speed} \cdot \Delta t \cdot \frac{N}{\text{size}} \tag{48}$$

Afterwards, the previous position of foam is calculated instead of the next position, as it ensures that every single pixel gets covered equally and no holes are left behind. An analogy to this would be that instead of

blowing snow away, which would create areas of high snow density, the snow should be vacuumed in. Before the data is stored inside the texture, `dissipation_factor` is used to ensure that foam doesn't accumulate infinitely over time. The final foam texture is now fully calculated and updated every frame based off the Jacobian value and the velocity of water at that specific time is as shown in Figure 54

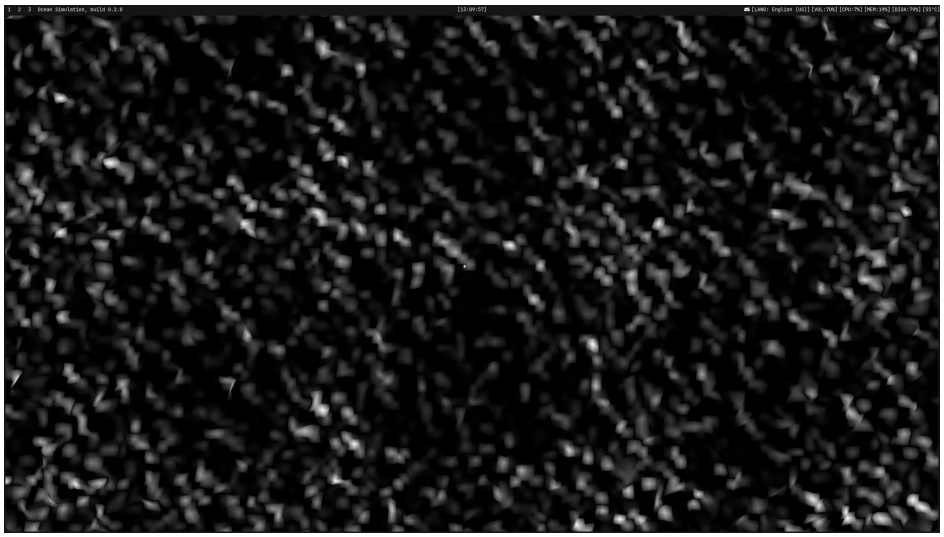


Figure 54: *Fragment shader output when* `vec3<f32>(foam_strength)` *is the output, acknowledging the fact the output is not perfect and could be improved, however this satisfies the simulation requirements.*

3.8.3 Foam Rendering

Inside the fragment shader, the texture is first sampled at the appropriate `uv` coordinates and data is extracted from the red channel. As previously seen in Figure 54 the output is still blocky and missing sharper detail. Therefore the fragment shader is also responsible for amplifying the detail on the texture by calculating the `foam_drift` value using the wind speed and time. This `foam_drift` value is then used to acquire two `uv` coordinates, used to generate the random noise. This noise is then smoothed out and applied onto the foam texture, as seen in Figure 55.

```wgsl
// Acquire foam drift value
let foam_drift = wind_norm * camera.delta_time * wind_mag * 0.00035;
// A lot of random constants
let foam_uv_a = in.world_pos.xz * 2.0 + foam_drift;
let foam_uv_b = in.world_pos.xz * 4.5 + foam_drift * 1.4 + vec2(17.3, 4.8);
let detail_noise = noise(foam_uv_a) * 0.6 + noise(foam_uv_b) * 0.4;
// Apply generated noise
let foam_shaped = advected_foam * smoothstep(0.2, 0.8, detail_noise);
```

Figure 55: *Modified section of the code responsible for introducing detail. Note that the* `noise` *function is not a standard* `wgsl` *function, but rather a custom implemented smooth noise.*

The output from this more detailed foam is shown in Figure 56, by mapping `foam_shaped` onto the output colour.

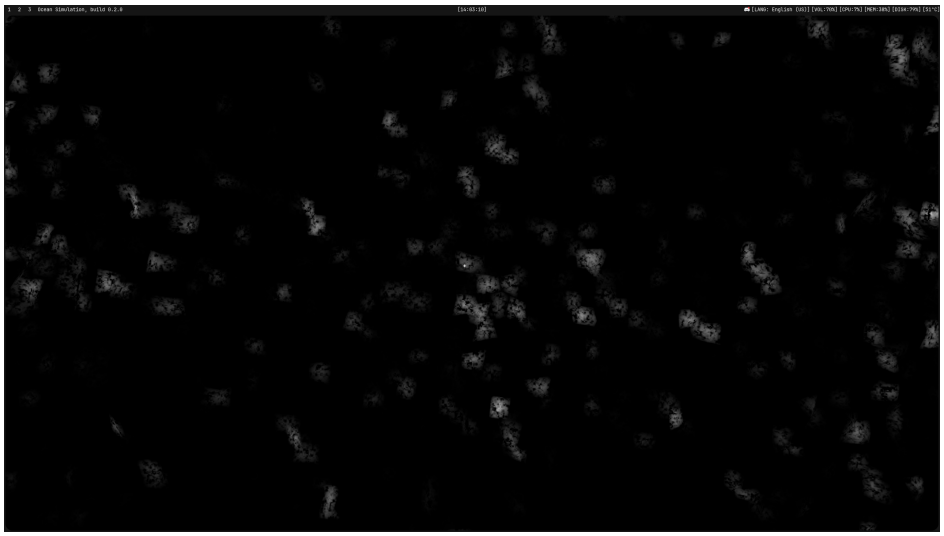


Figure 56: *Image of* `foam_shaped` *mapped to the output colour, where bright spots represent foam present in that area.*

The rendering of the foam is to be computed, using the half-Lambert model. The foam is given an off-white albedo color. The diffuse term will remap the surface normal dot product from $[-1, 1]$ to $[0, 1]$ using an `n_dot_light` $\times 0.5 + 0.5$ warp. Then a small ambient occlusion factor is added to darken the areas with foam of high density as shown in Figure 57.

```wgsl
let foam_albedo = vec3(1.0, 0.98, 0.96);
// THe ambient occlusion.
let foam_ao = 1.0 - foam_factor * 0.2;
let foam_lit = foam_albedo * (n_dot_light * 0.5 + 0.5) * foam_ao;
```

Figure 57: *Modified section of the fragment shader computing the foam diffuse term using a half-Lambert model*

A `sky_fill` term is calculated using the current `horizon` and `zenith` sky colour, which will make foam not as dark under certain angles. The direct and sky terms are summed up to give the complete foam diffuse, and on top of this a small specular glint is added using a simple Blinn-Phong with an exponent of 150. Finally, the accumulated foam light is blended into the water colour using `total_foam` and a specular term added on top as shown in Figure 58.

```wgsl
let foam_sky = mix(horizon, zenith, 0.5) * 0.3;
let foam_diffuse = foam_lit * light_color + foam_sky;
let foam_spec = pow(max(dot(normal, half_dir), 0.0), 150.0) * foam_factor * 0.08 *
    intensity;

color = mix(color, foam_diffuse, smoothstep(0.0, 0.18, total_foam) * total_foam);
color += foam_spec * foam_albedo;
```

Figure 58: *Modified section of the fragment shader combining the sky fill, specular glint and final blend of foam*

3.9 Subsurface Scattering

The simulation does not account yet for the most important factor - that Water is transparent, not opaque. Therefore whenever the ocean waves go over and cover the light source, the absorbed light has to scatter

inside the surface at various angles and exit the surface at a different point in space, while changing the colour hue. This is referred to as Subsurface Scattering (SSS), and it is what gives those turquoise, vibrant glowy look to the thin crests of a wave. Figure 59 shows how the light undergoes SSS.

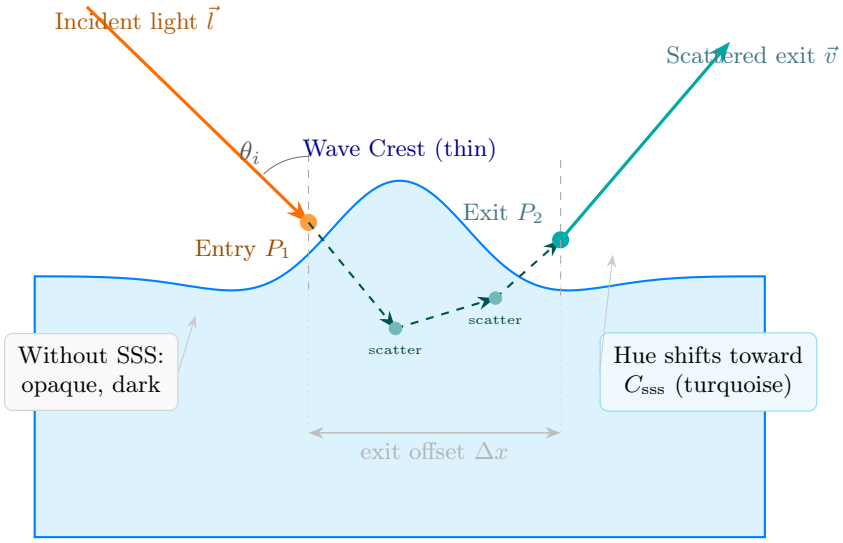


Figure 59: *Subsurface scattering on a wave crest. The thinner the crest, the shorter the internal path and the more visible the turquoise tint*

A translucency approximation has to be used since true SSS would require expensive volumetric lighting calculations. The technique used in the simulation will be the Wrapped Back Lighting approximation [25]. It works by checking if the light is behind a wave relative to camera, then by adding `normal + sss_distortion_scale`, the shader simulates the refraction of light as it enters the water, bending towards the viewer. Figure 60 represents the `wgsl` code implementing this technique.

```wgsl
let trans_light_dir = normalize(light_dir + normal * ocean_settings.sss_distortion_scale)
    * sss_crest_mask;
let trans_dot = max(dot(view_dir, -trans_light_dir), 0.0);
let p_back = pow(trans_dot, ocean_settings.sss_power);
```

Figure 60: *Code responsible for generating the light effect generated based off the* `sss_crest_mask`

The crest mask is made by utilising the height of the wave, as well as the Jacobian value to identify areas of compression ($J > 1$) and smoothening between the two values as shown in Figure 61.

```wgsl
let sss_thickness_mask = 1.0 - smoothstep(ocean_settings.sss_min_height, ocean_settings.
    sss_max_height, in.height);
let sss_crest_mask = smoothstep(0.7, 0.1, jacobian);
```

Figure 61: *The SSS crest mask calculation involving height of wave and the jacobian*

Afterwards, all the variables can be combined to form a strength factor. Using this strength factor, the final SSS color can be formed by mixing the `sss_color` from the settings and account for these strength factors as shown in Figure 62.

```wgsl
let sss_strength = p_back * sss_thickness_mask * sss_crest_mask * ocean_settings.
    sss_intensity * sss_dist_fade;
let wave_peak_sss = smoothstep(0.8, 0.1, jacobian) * ocean_settings.sss_intensity;
let sss = mix(ocean_settings.sss_color.rgb, light_color, p_back) * (sss_strength +
    wave_peak_sss);
```

Figure 62: *The final SSS calculations*

3.10 Cascading multiple FFT's

Even with the Tessendorf's model set in place, the ocean may still look repetitive and bland. This is a common issue, which involves a genius fix where instead of increasing the complexity of the spectra itself to generate a more precise and random ocean, by overlaying multiple FFT cascades running in real-time with different settings, a realistic model of the ocean can be created, where big swells and small ripples can exist co-united. 63.

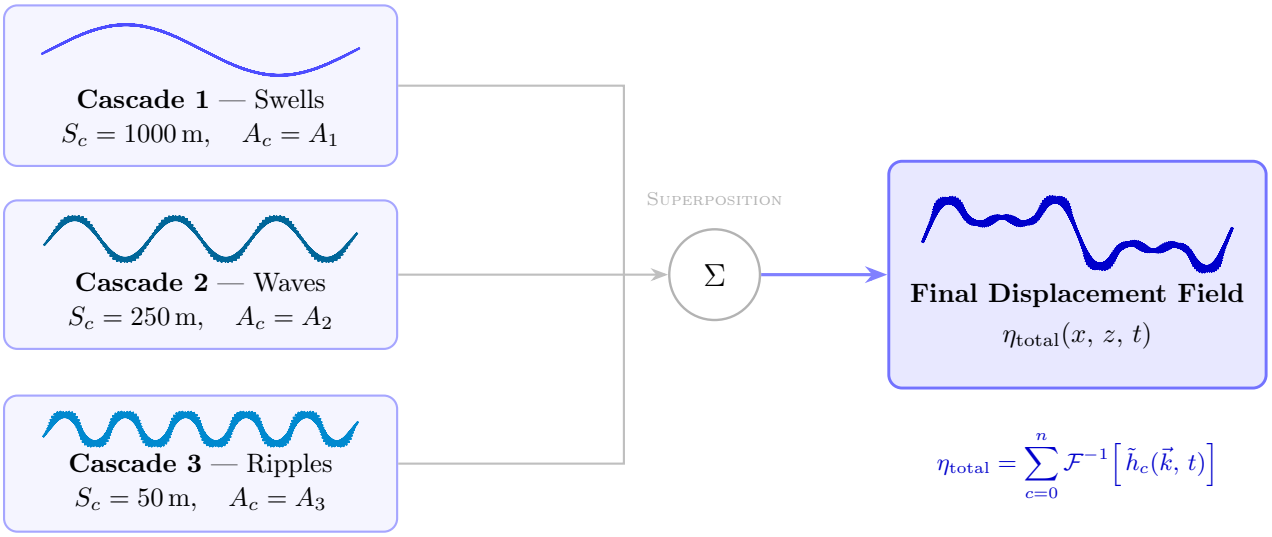


Figure 63: *Multiple independent IFFT simulations run in parallel, each with its own tile size S_c and amplitude A_c. Their displacement fields are summed via superposition to produce a single combined displacement field.*

To allow the user to freely adjust the cascades, ocean settings now carry the cascade data where at index 0 the cascade size is stored and at index 1 the amplitude is stored as shown in Figure 64.

```rs
ocean_settings_uniform.cascade_data[cascade_index][0]   // physical size
ocean_settings_uniform.cascade_data[cascade_index][1]   // amplitude
```

Figure 64: *An example section of the code, showcasing the two dimensional array for a single cascade*

The `State` struct is adjusted to instead of holding a single ping-pong framebuffer and a single `heigh_field`, to holding a vector of `CascadeResources`. This `CascadeResources` struct will now be responsible for holding the necessary textures and bind groups.

Now, instead of passing out one of the ping or pong textures to the main `render.wgsl` shader, all the cascades have to be added up. Therefore, a new compute pipeline `cascade.wgsl` has to be created. Every frame, each cascade is handled individually first, updating the spectrum and performing butterfly passes, and generating

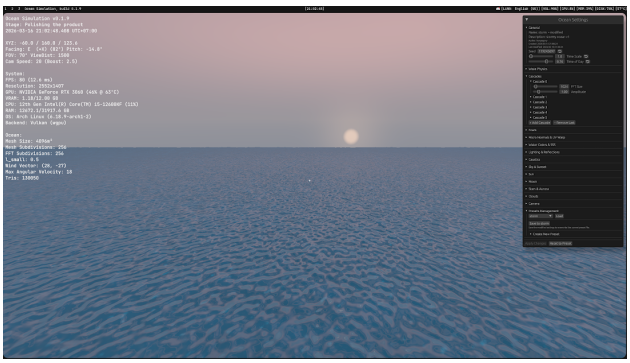
`texture_packed`. Afterwards, the newly created pipeline will loop through each cascade and write its final outputs onto a combined texture. This is also done using a ping-pong architecture to allow the GPU to work in parallel without overwriting data and avoiding timing issues. This is done as shown in Figure 65.

```wgsl
@compute @workgroup_size(16, 16)
fn combine_cascades(@builtin(global_invocation_id) gid: vec3<u32>) {
    let n = ocean_settings.fft_subdivisions;
    let coords = vec2<i32>(gid.xy);
    let master_scale = 1000.0;
    let uv = (vec2<f32>(gid.xy) + 0.5) / f32(n);

    let this_fft_size = ocean_settings.cascade_data[combine_config.cascade_index].x;
    let scaled_uv = uv * (master_scale / this_fft_size);

    let cascade_data = textureSampleLevel(in_packed, in_sampler, scaled_uv, 0.0);
    let combined_data = textureLoad(combined_read, coords, 0);

    // now everything is packed again.
    textureStore(combined_write, coords, combined_data + cascade_data);
}
```

Figure 65: *Modified code from* `cascade.wgsl` *showing the combination of the multiple cascades*

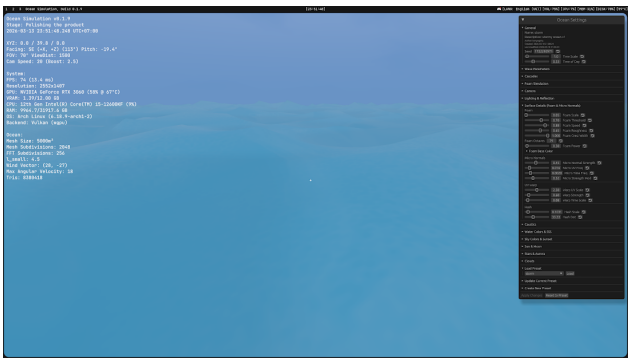
This new cascade system now allows for endless possible combinations of the ocean, where some examples can be seen in Figure 66.

High Frequency



FFT SIZE	AMPLITUDE
1024.0	1.000
256.0	0.500
64.0	0.180
16.0	0.060
4.0	0.015
1.0	0.003

Broad swells



FFT SIZE	AMPLITUDE
12000.0	0.700
1500.0	1.200
250.0	0.800
10.0	0.040

Figure 66: *Comparison of Cascade presets. High-frequency detail on left and Broad-scale swells on right.*

3.11 Limitations and issues

Any project will have its own set of issues. Since this discussion values the open-source model and honesty, all the issues are to be publicly shared.

3.11.1 The Blob Object

First the code is not structured to its best potential. To be more specific, the `State` struct mentioned throughout is doing too much work. The struct handles all the fields, contains most of the methods related to FFTs, Cascades, Foam and so on. This creates a so-called "god object", which violates the single responsibility principle necessary for clean and organised code. This is however, a relatively straightforward fix of refactoring code, but a fix that requires time which project is not able to afford. The struct has to be split up into sub-modules, as shown in Figure 67.

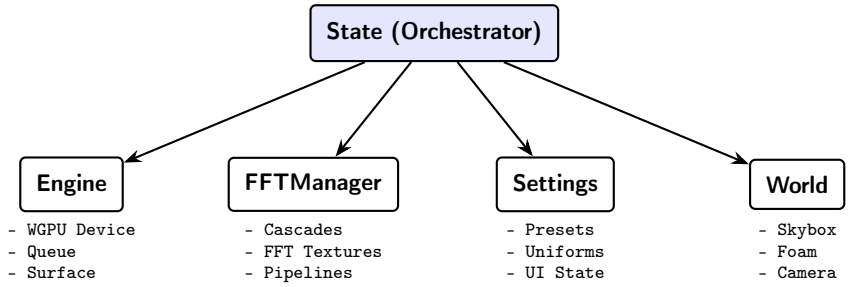


Figure 67: *The modular hierarchy*

3.11.2 Unnecessary re-calculations

The whole simulation is packed with unnecessary code which is possible to reduce by implementing methods and functions that will suit multiple modules at once, instead of repeating some preset code. For example, the Jacobian value is calculated twice in two different places, once in the foam shader and second time in the fragment shader. However, this can be simplified to actually be computed inside of the combine shader and stored in the free alpha channel. By introducing more of these tweaks, the frame time may be reduced and code complexity decreasing, allowing for more collaboration between peers since less codebase learning would be required.

3.11.3 Visual Fidelity

Second, the simulation ultimately falls slightly behind the set expectations, with the foam not working out as well as expected and having noticeable tiling which appears due to the fact that all the cascades have to be compressed into a single packed texture in the `combine_cascades` compute shader. The proposed solution was to discard the combined texture, and just have an array of textures which will hold each cascades individual data, and then in the rendering shaders a simple `for` loop would accumulate the displacements. However, this turned out to be a complex task, introduced new issues. This should not be treated as a failure, but rather as an exciting learning opportunity, which was taken full advantage of. This discussion has brought up many topics that otherwise would've gone unnoticed and no skills would be gained. In addition, the development process will not be fully terminated after this discussion is finished, therefore the visual artefacts, bugs, and any new additions can still be implemented and adjusted. After all, this is an impressive piece of work no matter what the result is, which is to be proud of.

4 Conclusion

In conclusion, it is entirely possible to build an efficient Real-Time Ocean Simulation in Rust with minimal external libraries. This is possible by using `wgpu` and `winit` libraries to set up the window, Afterwards, the mesh is generated with N subdivisions and size L and passed into the vertex and index buffer. Then, by using the `fft_size` and `amplitude` from each cascade, the initial frequency spectrum is generated using the Phillips Spectrum which follows Hermitian Symmetry.

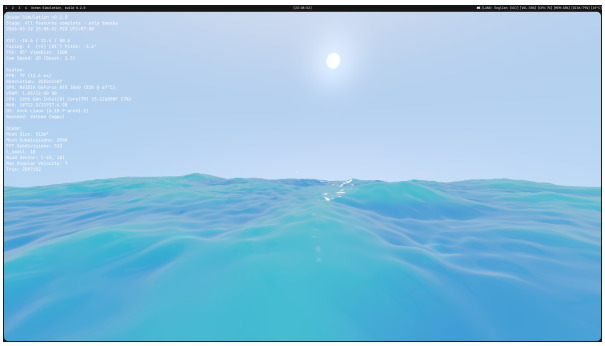
This data along with pre-calculated twiddle values are passed into the GPU compute shader, which uses a ping-pong texture model to compute an Inverse Fast Fourier Transform. This IFFT uses a DIT Tukey-Cooley

Radix-2 butterfly passes to transform the spectrum from frequency domain into a time domain, and store the resultant vertical displacement η, horizontal displacements dx, dz into a single `rgba16float` texture due to the Hermitian Symmetry discussed earlier. These textures from each cascade are then summed up using a compute shader to increase the detail on the ocean surface and are stored in a combined texture. This combined texture is then used inside of the foam compute shader, which calculates the Jacobian value to find places to generate the foam. The Jacobian value is a measure of compression in the ocean surface. This foam is advected every frame using the velocity of the ocean by implementing the central differences method. This method samples the neighbouring points around a texel and approximates the derivative.

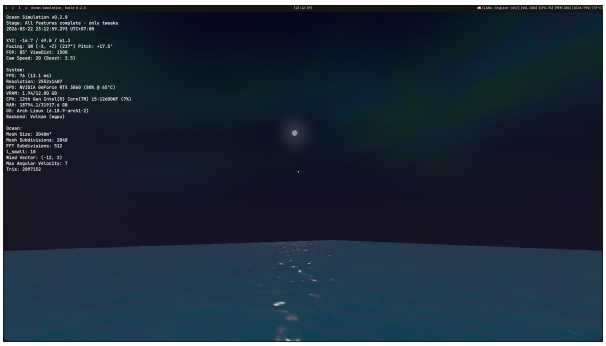
Afterwards, the vertex shader uses the combined textures to extract the vertical and horizontal displacements at each texel and apply those displacements to the mesh surface. Then, a skybox shader is used to generate a procedurally generated sky which can be fine tuned to fit various conditions, such as time of day, sky visibility, aurora strength and much more. The skybox shader is also responsible for providing a light source for reflections to appear in the water. Next, the fragment shader uses a Cook-Torrance BRDF model to implement Physical Based Rendering by utilising the idea of microfacets, as well as apply additional effects on the water surface, like Subsurface Scattering (SSS). SSS is implemented using the Wrapped Back Lighting approximation and the Jacobian to find thin areas of a water crest and display a more turquoise colour by approximating the random scattering inside the water surface. In addition to that, slight caustics were implemented but not mentioned in the discussion, as the effect was not pronounced and are not fit due to the depth of the water. Then, inside the fragment shader tone mapping and gamma correction is applied onto the water surface before finally displaying it on the screen. Final results can be seen in Figure 68, which does intact resemble an ocean surface.

In the end, the ocean scales nicely performance wise as size and subdivisions grow. The simulation runs at approximately 70FPS, correlating to an average 14.5ms frame time, while using an *NVIDIA RTX 3060 12GB* [26] graphics card running at *2560x1440* and *32GB of DDR4 3200MHz RAM*. This aligns nicely with the aims and goals of this project, actually surpassing the expected performance mark. The goal of using minimal external libraries is also met, since no read-to-use game engines were used, only pure Rust, WGSL, `winit`, `wgpu`, `egui` and other small non-essential crates were used. And lastly, there was a lot of experience gained by constantly debugging the ocean surface and introducing new mechanics and features never seen before, forcing to explore topics of mathematics and computer science which were not familiar before.

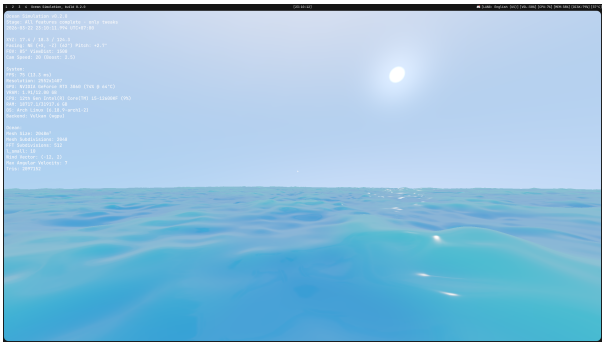
With that, the discussion comes to an end.



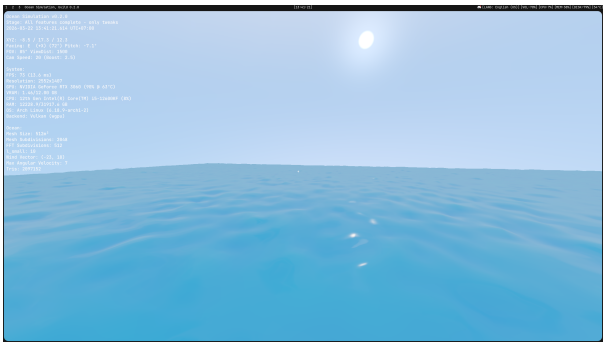
(a) *Big swells*



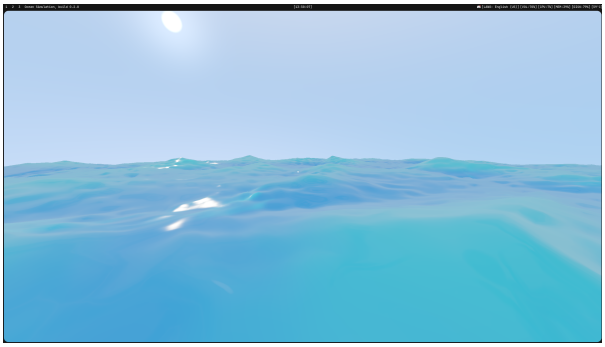
(b) *Calm night*



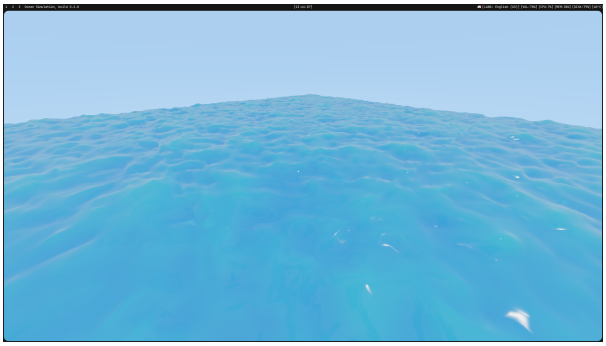
(c) *Medium swells*



(d) *Calm day*



(e) *Medium swells*



(f) *Big swells up-top*

Figure 68: *The resultant images with all effects combined, some containing debug information*

References

[1] Rare. *Sea of Thieves*. 2018. URL: https://www.seaofthieves.com/ (visited on 03/22/2026).

[2] James Cameron. *Titanic*. Motion Picture. 1997. (Visited on 03/22/2026).

[3] Jerry Tessendorf. *Simulating Ocean Water*. 2004. URL: https://people.computing.clemson.edu/~jtessen/reports/papers_files/coursenotes2004.pdf (visited on 09/30/2025).

[4] Sergei I. Badulin and Vladimir E. Zakharov. *The Phillips Spectrum and a Model of Wind-Wave Dissipation*. 2019. URL: https://arxiv.org/pdf/1912.03945 (visited on 03/03/2026).

[5] Francisco Manuel Morales-Rodríguez et al. "Outstanding Videogames on Water: A Quality Assessment Review". In: *Sustainability* 10.10 (2018), p. 3521. DOI: 10.3390/su10103521. (Visited on 03/22/2026).

[6] Epic Games. *Unreal Engine*. URL: https://www.unrealengine.com/en-US (visited on 12/09/2025).

[7] Unity Technologies. *Unity Game Engine*. URL: https://www.unity.com (visited on 12/09/2025).

[8] Encyclopædia Britannica. *Ocean: Definition and Science*. URL: https://www.britannica.com/science/ocean (visited on 12/02/2025).

[9] Khronos Group. *Real-Time vs. Offline Rendering: Discussion*. URL: https://community.khronos.org/t/real-time-vs-offline-rendering/70305 (visited on 03/22/2026).

[10] The Rust Foundation. *The Rust Programming Language*. URL: https://rust-lang.org/ (visited on 11/07/2025).

[11] wgpu Project. *wgpu Rust Documentation*. URL: https://docs.rs/wgpu/latest/wgpu/ (visited on 11/07/2025).

[12] Ilya Rasputin. *Ocean Simulation Project Repository*. 2025. URL: https://github.com/konyogony/ocean (visited on 10/28/2025).

[13] Open Source Initiative. *The MIT License*. URL: https://opensource.org/license/mit (visited on 03/22/2026).

[14] GitHub. *About GitHub and Git*. URL: https://docs.github.com/en/get-started/start-your-journey/about-github-and-git (visited on 01/22/2026).

[15] Franz Gerstner. "Theorie der Wellen". In: *Annalen der Physik* 32.8 (1809). DOI: 10.1002/andp.18090320808. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.18090320808 (visited on 03/10/2026).

[16] O. M. Phillips. "The equilibrium range in the spectrum of wind-generated waves". In: *Journal of Fluid Mechanics* 4.4 (1958), pp. 426–434. DOI: 10.1017/S0022112058000550. (Visited on 03/04/2026).

[17] G. A. Mastin, P. A. Watterberg, and J. F. Mareda. "Fourier Synthesis of Ocean Scenes". In: *IEEE Computer Graphics and Applications* 7.3 (1987), pp. 16–23. DOI: 10.1109/MCG.1987.276961. (Visited on 03/06/2026).

[18] Michael G. Rozman. *Radix-2 Fast Fourier Transform*. 2019. URL: https://www.phys.uconn.edu/~rozman/Courses/m3511_19s/downloads/radix2fft.pdf (visited on 03/10/2026).

[19] Karl Bittner. *Introduction To Shaders*. 2024. URL: https://hexaquo.at/pages/introduction-to-shaders/ (visited on 03/13/2026).

[20] Robert L. Cook and Kenneth E. Torrance. "A Reflectance Model for Computer Graphics". In: *ACM Transactions on Graphics* 1.1 (1982), pp. 7–24. (Visited on 03/14/2026).

[21] Joey de Vries. *Physically Based Rendering: Theory*. 2023. URL: https://learnopengl.com/PBR/Theory (visited on 03/14/2026).

[22] Bruce Walter et al. "Microfacet Models for Refraction through Rough Surfaces". In: *Proceedings of the Eurographics Symposium on Rendering*. 2007, pp. 195–206. (Visited on 03/14/2026).

[23] Brian Karis. *Real Shading in Unreal Engine 4*. SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice. 2013. URL: https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf (visited on 03/14/2026).

[24] Academy of Motion Picture Arts and Sciences. *S-2014-004: Academy Color Encoding System (ACES) Output Transform*. Tech. rep. The Academy of Motion Picture Arts and Sciences, 2014. URL: https://www.oscars.org/science-technology/sci-tech-projects/aces (visited on 03/22/2026).

[25] Colin Barré-Brisebois and Marc Bouchard. "Approximating Translucency for a Fast, Cheap and Convincing Subsurface-Scattering Look". In: *Game Developers Conference (GDC)*. 2011. URL: https://colinbarrebrisebois.com/2011/03/07/gdc-2011-approximating-translucency-for-a-fast-cheap-and-convincing-subsurface-scattering-look/ (visited on 03/22/2026).

[26] NVIDIA. *GeForce RTX 3060 Family Product Overview*. URL: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3060-3060ti/ (visited on 03/22/2026).