How to create an efficient real-time Ocean Simulation?

Ilya Rasputin

August 2025 - April 2026

Contents

1 Introduction . . . **4**
1.1 Overview . . . 4
1.2 Applications . . . 4
1.3 Aims & Goals . . . 4
1.4 Prerequisites . . . 4
1.4.1 Ocean . . . 4
1.4.2 Efficient Real-Time Simulation . . . 5
1.4.3 Programming Language and External Libraries . . . 5
1.4.4 Open Source & Git . . . 5
1.4.5 Glossary of Mathematical Terms . . . 5

2 Literature Review . . . **7**
2.1 Sinusoidal functions . . . 7
2.1.1 Parameters of a sine wave . . . 7
2.1.2 Sines in higher dimensions . . . 9
2.1.3 Dispersion relation . . . 10
2.2 Gerstner waves . . . 11
2.3 Fluid Kinematics . . . 13
2.3.1 Acceleration Vector Field . . . 13
2.3.2 Fluid motion . . . 15
2.3.3 Navier-Stokes . . . 17
2.4 Phillips spectrum . . . 17
2.4.1 Theoretical Derivation . . . 17
2.4.2 Directionality and Scale . . . 18
2.4.3 Statistics and Time Evolution . . . 19
2.5 Discrete Fourier Transform (DFT) . . . 19
2.5.1 Fast Fourier Transform (FFT) . . . 20
2.5.2 Bit Reversal . . . 21
2.5.3 Inverse Fast Fourier Transform (IFFT) . . . 22
2.6 Lighting Models . . . 22
2.6.1 Surface Normals . . . 23
2.6.2 Blinn-Phong model . . . 23
2.6.3 Cook-Torence PBR model . . . 25

3 Discussion . . . **27**
3.1 Initial Setup . . . 28
3.2 Mesh Formation . . . 30
3.2.1 Vertex Structure and Data Flow . . . 30
3.2.2 Grid Generation and Indexing . . . 31
3.3 Vertex Displacement . . . 31
3.3.1 Temporal Updates and Uniform Buffers . . . 31
3.3.2 Data Transmission and Shader Logic . . . 32
3.4 The Camera . . . 35
3.4.1 Direction and Spherical Projections . . . 35
3.4.2 Viewing Frustum and Matrices . . . 35
3.5 The Skybox . . . 36
3.5.1 Cubemapping and Depth Manipulation . . . 36
3.5.2 Procedural Atmospheric Simulation . . . 37
3.5.3 Celestial Bodies and Volumetric Effects . . . 39
3.6 Blinn Phong implementation . . . 41
3.6.1 Surface Normal Reconstruction . . . 41
3.6.2 Fragment Shader Calculations . . . 43

3.7 Tessendorf's Model Implementation . . . 44
3.7.1 Initial Data Generation . . . 45
3.7.2 Time Evolution . . . 49
3.7.3 IFFT Implementation . . . 52
3.8 Settings and Presets . . . 56
3.8.1 Preset, Builder and Uniform . . . 56
3.8.2 User Interface . . . 56
3.9 PBR Implementation . . . 58
3.9.1 Mapping Equations to Shader Code . . . 58
3.9.2 Sky Reflections and Tone Mapping . . . 59
3.10 Jacobian Foam . . . 61
3.10.1 The Jacobian . . . 61
3.10.2 Foam Generation and Advection . . . 62
3.10.3 Foam Rendering . . . 64
3.11 Subsurface Scattering . . . 66
3.12 Cascading multiple FFT's . . . 67
3.13 Limitations and issues . . . 70
3.13.1 The Blob Object . . . 70
3.13.2 Unnecessary re-calculations . . . 71
3.13.3 Visual Fidelity . . . 71

4 Conclusion . . . 71

1 Introduction

1.1 Overview

There are these grand, gorgeous and spectacular oceans in many video-games like *Sea of Thieves* [24], or even in famous movies like *Titanic* [5]. But how is it possible to create such an ocean? What mathematics and physics goes into this?
This discussion will explore the technology and mathematics behind real-time ocean simulations, and build up its own interpretation in this field. First, this project will discuss the theory, which will cover fluid dynamic equations, like Navier-Stokes, as well as go over lighting calculations and different ocean model interpretations. This will provide the reader with an understanding of the ideal model of fluids, if there was infinite computational power.
As a result, this project will look at more practical and efficient approximation methods more commonly used in the real world. Using simplified lighting calculations and building on top of Tessendorf's implementation [31], which includes the use of the Phillips spectrum [2] and Inverse Fast Fourier Transform algorithms to achieve a realistic and visually compelling result.

Figure 1: *Image from the video game* Sea of Thieves *[24]*

1.2 Applications

There are many applications for a real-time ocean simulation. In most use cases, the model is efficient enough to be applied in games and movies. This drives the user to play and explore the game, or be more compelled to watch certain media containing the ocean, generally increasing the revenue of the company as well as user satisfaction [18]. In other cases, the model may be really heavy and would require heavy computing power, so it could theoretically be used to stress test and compare different hardware options, such as Graphic Processing Units (GPUs). This however will not be studied in the discussion, as currently access to a range of hardware is limited. Last but not least, a use case for an ocean simulation is of course the learning experience. 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 intricacies and edge cases, as well as 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 a simulation using minimal external ready-to-use tools and engines, such as *Unreal Engine* [9] and *Unity* [34]. This project aims to build the whole simulation, the whole "game" engine, from complete scratch in Rust, excluding the use of libraries required to communicate with the hardware and other small non-essential crates. This helps to develop the skills and knowledge about lighting, 3D rendering as well as contribute to the Rust community. Lastly, this discussion aims to act as guidance for enthusiasts and fellow ambitious programmers, or any other readers to follow if they wish to learn more. The project aims to provide clear explanations, discussions and helpful ideas that will drive the success of others in this area of computing.

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 [7]. Primarily the ocean waves are generated by the transfer of energy from the wind onto the water's surface, forming waves, but rarely could

be caused by seismic activity. In this study, the simulation will consist of randomised waves, which will later be transformed using the Phillips Spectrum [2] 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. This allows for seamless effect and direct interactions with the user, hardware, as well allowing for proper physics computations. There is a notable difference between other types of simulations, where in "offline" [16] or "non-real-time" simulations the output is usually predetermined and perhaps rendered and computed beforehand, since the algorithms used there are usually slower.

It is also important to understand the trade-off a program like this may have. This simulation should be efficient, meaning efficient and intricate algorithms have to be in place to minimise the heavy load, however which of course would not provide as much realism as a Navier-Stokes computation, but that just is not physically possible. 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. This value is chosen since it's the baseline for most tests, aligning with the most common V-Sync frame rate and allowing for maintained visual fidelity.

1.4.3 Programming Language and External Libraries

To create the simulation, a programming language has to be chosen. This will determine at what pace the development will go, as well as the available tools, libraries and community help. For this case study, the programming language of choice is Rust [32]. While C [14], or other variations such as C++ [29] may seem like the perfect option, with a mature ecosystem and widespread adoption in the industry. However, the complexity of code may lead to the program having memory leaks and weird bugs, which would degrade the performance and prolong the development cycle. Rust on the other hand, has a great typing system, is memory safe and has an emerging ecosystem of useful libraries. It is also important to consider that many would've went with a game engine, like *Unity* [34] or *Unreal Engine* [9] which is not a bad choice, allowing for way faster development speed and ease of use with the help of blueprints [8], however having more control and taking on the challenge of starting from scratch may drive this area of development further, as well as being more interesting.

It is also important to mention the library powering this project, `wgpu` [38]. WebGPU (wgpu), is a JavaScript, Rust, C++ and C API for cross-platform efficient graphic processing unit access [11], initially developed by Mozilla [19] as a solution for a safe and native implementation of a graphics API. When working with pure JavaScript, wgpu creates a canvas in the browser to draw the image, while on other platforms like JavaScript runtimes (Bun), or Rust, respective libraries are present to achieve the same result. The `wgpu` crate also supports all modern backends such as Vulkan / DX12 / DX11 / Metal [37] which would be really useful for efficient distribution of the program.

1.4.4 Open Source & Git

As stated previously, guiding fellow creators and driving the success of others is an important aim of this discussion. Therefore the whole repository [25] is available on GitHub, a cloud based platform where you can store, share, and work together with others to write code [12]. Open Source licensing is the core principle of development for the community, where the code written is shared and available for the whole public to see, dissect, interact and use. The *MIT* license [21] provides a good support of that, allowing for any modifications and user engagement to occur, that is what the project will be using. *Git* is also an important tool in the development process, being a version control system that intelligently tracks changes in files [12]. This tool allows to check previous changes, rollback to old versions and take any missing images for use in the discussion.

1.4.5 Glossary of Mathematical Terms

This section contains the information needed to understand this discussion, which includes a list of the standard mathematical notation, programming functions and terms that will be used throughout this study.

Note: Help box

First, it is important to note that this is a heavily Mathematics based project, however the study will do to the best of its abilities to explain every concept seen here. Be it partial derivatives, sine waves or even any computer science related terms, like uniform buffers or compute shaders. 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. This is an example of their use.

The following is a list of the standard mathematical notation used throughout this study.

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
$\lVert \vec{v} \rVert$	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

In addition to the mathematical terms and the note box, the following `inline_code_blocks` will be used throughout the discussion, showcasing a connection to the code. The following `wgsl` terms will also come up in mathematical equations and figures.

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

2 Literature Review

2.1 Sinusoidal functions

2.1.1 Parameters of a sine wave

A sine wave, also sometimes referred to as a sinusoidal wave, is a special type of a function, often used in mathematics, physics, and other areas. The sine wave is a smooth function that oscillates up and down with a periodical frequency of 2π. These waves are very predictable, and we can use them in real-time computations. This is the core foundation on which our whole simulation is based. Therefore, it is important to understand how all the different parts of the sine-wave function work. To understand this, look at where the sine wave originates, a simple unit circle (circle with an equation $\cos^2\theta + \sin^2\theta = 1$), as illustrated in Figure 3. We can see how this unit circle presents a natural and oscillating output as the angle increases, where the point is moving round the circle, producing the waveform seen in Figure 2.

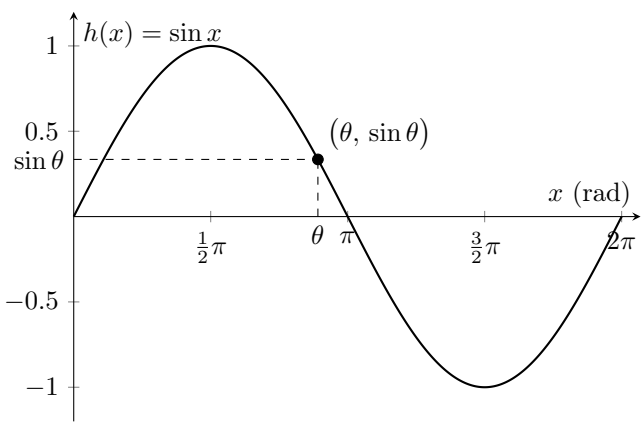


Figure 2: *Sine wave at* $(\theta, \sin\theta)$

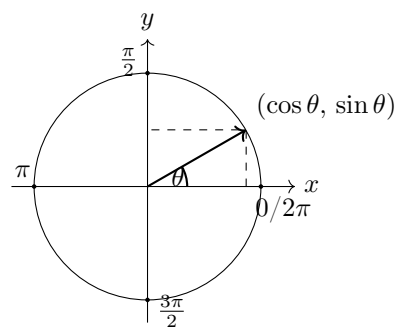


Figure 3: *The unit circle.*

Certain parameters can modify a sine wave to make it behave differently. In a sine wave, its amplitude (A), wavenumber (k), angular frequency (ω), and phase shift (ϕ) are adjustable. First, add a second argument to our height function - time (t), and use a spatial variable x instead of an angle θ. The general form of a sine wave including all these parameters is:

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

Now, this section will explore and explain step by step what each parameter means. A represents the amplitude of the wave - basically the peak height from the baseline, as visualized in Figure 4.

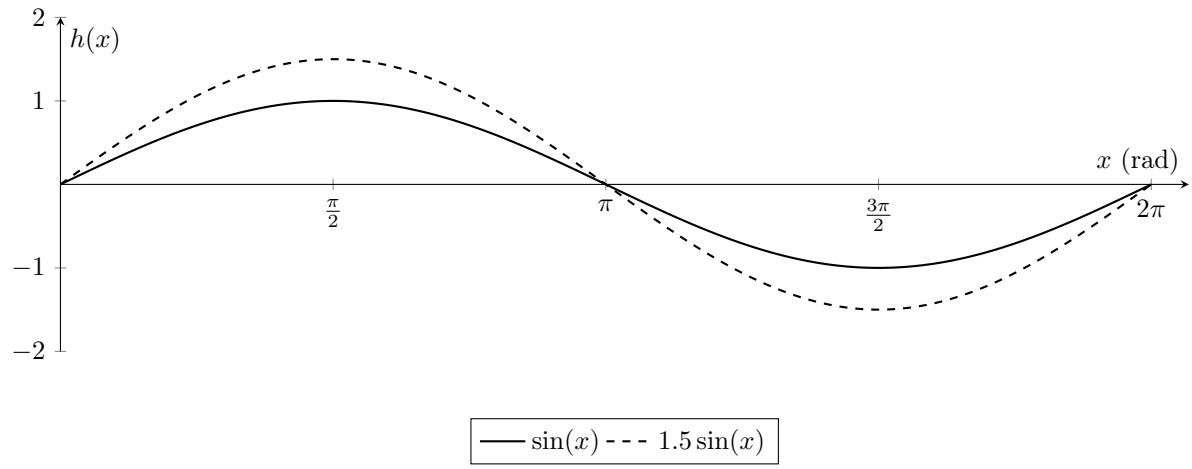


Figure 4: *Comparison of two sine waves with different amplitudes.*

Next, k (wavenumber) changes the spatial frequency of the wave, which stretches or compresses it horizontally. This determines how many full oscillations the wave completes over a distance of 2π, as shown in Figure 5. The equation $T = \frac{2\pi}{k}$ can also be used to get the period, meaning how long it takes to complete a full oscillation.

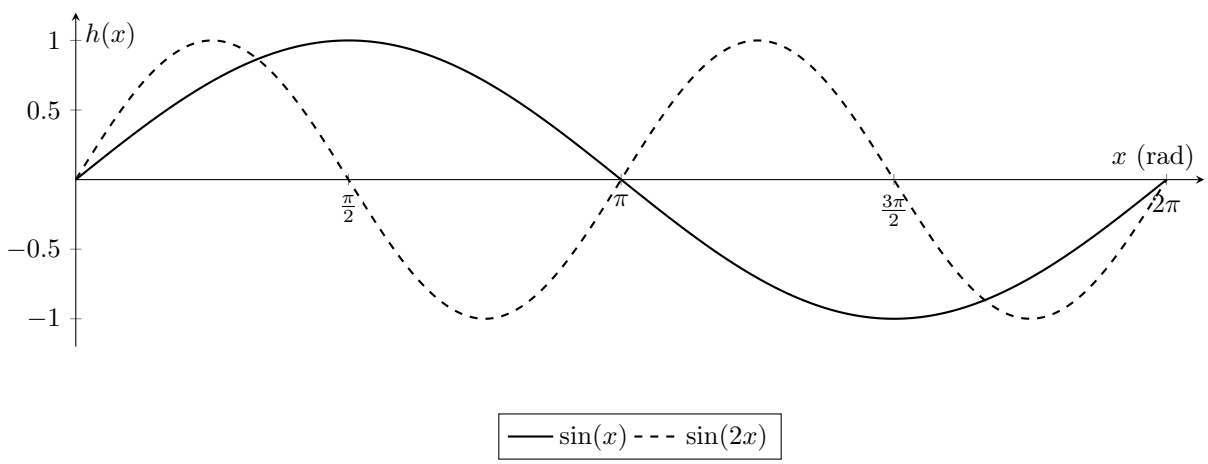


Figure 5: *Comparison of two sine waves with different spatial frequencies (wavenumbers).*

Finally, the angular frequency ω (omega) and time t shift the wave continuously along the horizontal axis, allowing it to "move" when time is incremented. The phase shift ϕ determines the starting position of the wave at $t = 0$, as well as being able to encode the initial random phase, which is crucial for use in Phillips Spectrum 2.4. Together, these parameters control the motion of the wave. The effect of the phase shift is illustrated in Figure 6.

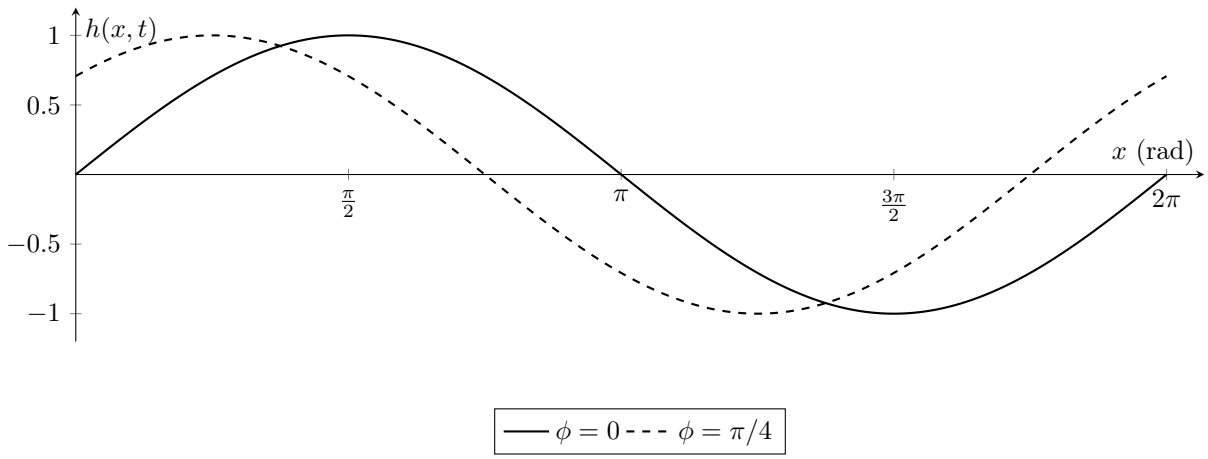


Figure 6: *Illustration of phase shift.*

2.1.2 Sines in higher dimensions

Note: Vectors

A vector is a type of variable which has both magnitude and direction, for example velocity (a scalar comparison would be speed). They are denoted by an arrow $\vec{v}$ and can be represented in multiple different formats, may it be a column vector, a component vector or other forms.

$$\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = v_x\hat{i} + v_y\hat{j} + v_z\hat{k} = \langle\, v_x,\, v_y,\, v_z \,\rangle \tag{2}$$

It is also important to understand the dot product. The dot product is a special type of operation performed on two vector quantities to produce a scalar number, denoted by a $\cdot$ between the variables. The dot product usually indicates how similar the two vectors are, if they are pointing in the same direction. There are multiple formulas for the dot product, here is a few:

$$\vec{a} \cdot \vec{b} = ||\vec{a}||\ ||\vec{b}|| \cos\theta = a_x b_x + a_y b_y + a_z b_z \tag{3}$$

So far this discussion examined a one-dimensional travelling sine wave. In a real-time ocean simulation the world is three-dimensional, with a large two-dimension surface, so we change the single spatial coordinate x to a 2D position vector $\vec{x} = (x, y)$. To define the wave's direction of travel on this 2D surface, the scalar wave number k is changed to a 2D wave vector $\vec{k} = (k_x, k_y)$. The direction of this vector is the direction the wave travels, and its magnitude $||\vec{k}||$ is the wave number from the 1D case, as shown in Figure 7.

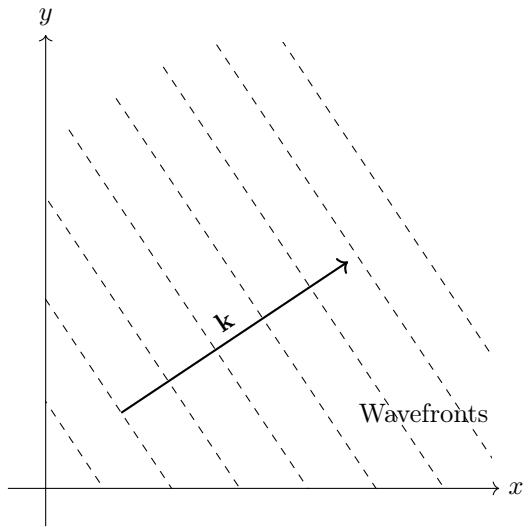


Figure 7: *Top-down view of planar wavefronts in 2D. The wave vector $\vec{k}$ indicates the travel direction, and the wavefronts (dashed lines) are perpendicular to $\vec{k}$*

In the 1D equation, the term $k\,x$ determines the phase of the wave at a given position. To get the phase at a 2D point $\vec{x}$, we need to know how far along the wave's direction the point is. This is accomplished by using the vector dot product, $\vec{k} \cdot \vec{x}$. The dot product projects the position vector $\vec{x}$ onto the wave vector $\vec{k}$, giving the correct phase contribution. By replacing the term $k\,x$ with $\vec{k} \cdot \vec{x}$, we arrive at the final equation for a single, planar travelling wave on a 2D surface is as shown in Equation (4).

$$h(\vec{x}, t) = A \cdot \sin\left(\vec{k} \cdot \vec{x} + \omega t + \phi\right), \tag{4}$$

Where α is the amplitude, $\vec{k} = (k_x, k_y)$ is the wave vector, ω is the angular frequency, ϕ is a phase offset, and $\vec{k} \cdot \vec{x} = k_x x + k_y y$ is the usual dot product.

2.1.3 Dispersion relation

Note: Hyperbolic functions

In addition to the standard trigonometric functions such as $\sin x, \cos x, \tan x, \sec x, \cot x, \csc x$, there exists a hyperbolic set of analogous functions. These functions show up in many areas of physics, mathematics and are usually defined in terms of exponentials.
The two fundamental hyperbolic functions are as shown in Equation (5), while the rest can be defined same as the traditional trigonometric functions, for example $\tanh x = \sinh x / \cosh x$

$$\begin{aligned} \sinh x &= \frac{e^x - e^{-x}}{2} \\ \cosh x &= \frac{e^x + e^{-x}}{2} \end{aligned} \tag{5}$$

These functions also satisfy identities similar to trigonometric identities, such as

$$\cosh^2 x - \sinh^2 x = 1, \tag{6}$$

which is analogous to the identity $\cos^2 x + \sin^2 x = 1$.

In a real ocean, all waves travel at different speeds. Wave dispersion tells us that waves with different wavelengths travel at different speeds. This relationship is represented in Equation (7).

$$\omega^2 = gk \tanh(kh) \tag{7}$$

In this equation, ω describes the angular frequency, g is the acceleration due to gravity constant ($9.8ms^{-2}$), and $k = ||\vec{k}||$. In this discussion, the depth of water h is assumed to be very large, so as $h \to \infty$, $\tanh(kh) \to 1$. Therefore, a simplified formula can be used, as shown in Equation (8).

$$\omega = \sqrt{g \cdot ||\vec{k}||} \tag{8}$$

Deep water usually refers to $h > \lambda/2$ [31]

2.2 Gerstner waves

In contrast to simple sine-wave approximation of the ocean, where the surface is only displaced vertically, Gerstner waves have a much more real-world approach to modelling the ocean. It was first derived by Franz Josef Von Gerstner in 1809 [10], where the waves would be an exact solution to the incompressible Euler equations in deep water gravity waves. Gerstner waves are great at conveying the sharpness of peaks and the flatness of troughs, because they incorporate horizontal displacement.

This section will now look at the governing equations and their derivation. Start by taking a two dimensional cross section of the wave perpendicular to the crest line, where using Cartesian coordinates (x, y), x would signify the direction of wave propagation and y being the axis pointing upwards (See Figure 8), it would also be reasonable to assume that there is no surface tension, since the amplitude in a deep ocean is non-negligible [13]. Now define the fluid domain to be a semi-infinite region $\{(x, y) : x \in \mathbb{R}, y < \eta(t, x)\}$ [13] bounded from the top by a free surface $y = \eta(x)$.

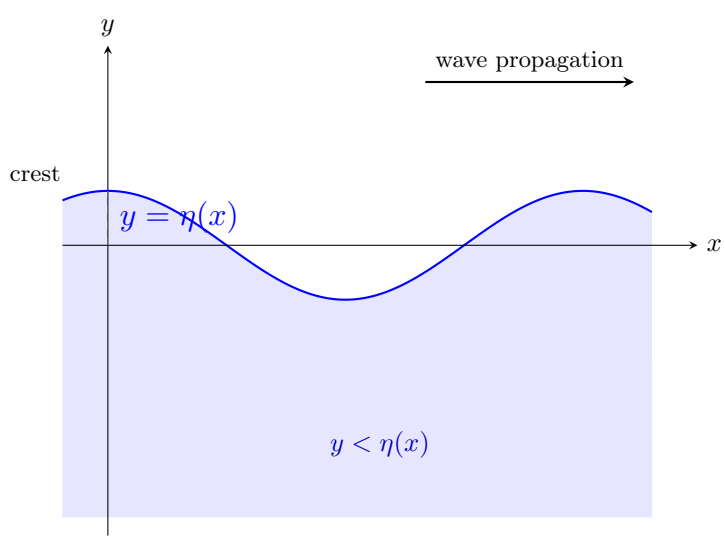


Figure 8: *Two-dimensional cross section of a Gerstner surface gravity wave*

Now define the velocity field in the fluid domain to be $(u(t, x, y), v(t, x, y))$ as well as being at a constant density, implying the equation of mass conservation $v_x + v_y = 0$. By making further assumptions of inviscid flow, the equations for motion will be given by Euler's equations where $P(t, x, y)$ denotes the pressure and g is gravitational constant of acceleration [13]:

$$\begin{cases} u_t + uu_x + vu_y = -P_x \\ v_t + uv_x + vv_y = -P_y - g \end{cases} \tag{9}$$

To decouple the motion of air from that of the free surface particles [13], introducing the dynamic boundary condition is necessary, where P_0 is the constant atmospheric pressure:

$$P = P_0 \text{ on } y = \eta(t, x) \tag{10}$$

And since the free-surface is always composed of the same particles, the kinematic boundary condition is present [13]:

$$v = \eta_t + u\eta_x \text{ on } y = \eta(t, x) \tag{11}$$

Not to forget the boundary condition at the bottom of the ocean, showing that at great depths there is particularly no movement of particles, given by:

$$(u, v) \longrightarrow (0, 0) \text{ as } y \longrightarrow -\infty \tag{12}$$

After assigning the boundaries, a description of a Gerstner wave requires 2 parameters, $a \in \mathbb{R}$ and $b \leq b_0$ for a fixed $b_0 \leq 0$, and so the lower half-plane represents the still water body [13]. Meaning if a and b are constants, the Gerstner wave would be given by the following equation, where $m > 0$ is fixed and $m = 2\pi/\lambda$ (wavenumber).

$$\begin{aligned} x &= a + \frac{e^{mb}}{m} \sin m \left(a + \sqrt{\frac{g}{m}} t \right) \\ y &= b - \frac{e^{mb}}{m} \cos m \left(a + \sqrt{\frac{g}{m}} t \right) \end{aligned} \tag{13}$$

This equations says that the path of the particle is centred around (a, b) with radius $\frac{e^{mb}}{m}$ and angular velocity of $\sqrt{\frac{g}{m}}$. Now obtaining a description of another particle is done simply by changing the values of a and b in (13). [13] Calling the set prescribed by $(a, b_0) : a \in \mathbb{R}$, means that the profile of the surface as time evolves is given by a smooth curve obtaining by setting $b = b_0$ in Equation (13) as shown in Equation (14) [13]

$$\begin{aligned} x &= a + \frac{e^{mb}}{m} \sin m \left(a + \sqrt{\frac{g}{m}} t \right) \\ y &= b_0 - \frac{e^{mb}}{m} \cos m \left(a + \sqrt{\frac{g}{m}} t \right) \end{aligned} \tag{14}$$

The equation seen in (14) is the equation for a trochoid for $b < b_0$. At the extreme case with the still water surface being at $(a, 0) : a \in \mathbb{R}$ leads to a surface wave having the profile of a cycloid, a continuous curve with upward cusps [13] (a pointed tip or juncture where two waves meet).

However, in order to prove that equation in (13) provides a general solution to the governing equations, the following properties have to be met:

- The equation of continuity $v_x + v_y = 0$ holds;
- a suitable pressure exists enabling Euler's equation in (9) and the boundary condition (10) to be satisfied;
- a particle on the free surface stays on the free surface, as governed by Equation (11);
- the limiting boundary condition in (12) is satisfied. [13]

Luckily, this process has been proven before by *D Henry* in publication [13] by utilising degree theory, starting by applying a transformation, where the map is a local diffeomorphism via the Inverse Function Theorem. This has shown a positive Jacobian determinant and established global injectivity using the Mean Value Theorem. The proof also utilizes the Invariance of Domain theorem, to demonstrate surjectivity and verify that the particle paths fill the entire fluid region beneath the trochoidal surface.

This approach works well for some simulations, not requiring heavy computation and looking visually appealing. However, this method will not be used in this discussion, as issues with Gerstner Waves start to pile up. Firstly, Gerstner waves simply break at scale. The parameters for amplitude, direction, phase and much more have to be generated and matched by hand, which first of all will be long, hence inefficient, and second of all will yield an unrealistic result. Another reason why Gerstner waves are not considered in this discussion is because while the computational power is cheap for a few waves, but the algorithm scales poorly when trying to match the detail provided by FFT methods. Another one of the disturbing issues that Gerstner Waves present is their ability to fold over themselves, as represented in Figure 9. Hence with all of these issues enumerated, the Tessendorf's [31] approach is more highly motivated.

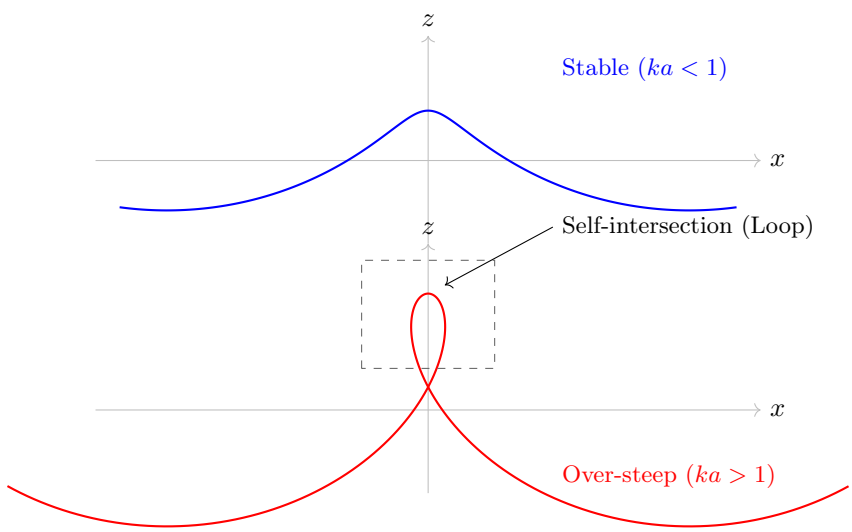


Figure 9: *Visual representation of Gerstner wave folding over itself*

2.3 Fluid Kinematics

It is important to now consider how the fluid particles behave in the real world, as it motivates the Navier-Stokes equation in Section 2.3.3 and explains why incompressibility is assumed throughout.

2.3.1 Acceleration Vector Field

Hence, consider a position vector,

$$\vec{r}(x, y, z, t) = r_x(x, y, z, t)\hat{\imath} + r_y(x, y, z, t)\hat{\jmath} + r_z(x, y, z, t)\hat{k} \tag{15}$$

meaning that the velocity at point (x, y, z, t) would be

$$\vec{v}(x, y, z, t) = v_x(x, y, z, t)\hat{\imath} + v_y(x, y, z, t)\hat{\jmath} + v_z(x, y, z, t)\hat{k} \tag{16}$$

To get the acceleration of a particle in this field, differentiate the velocity with respect to time. Using the chain rule is necessary to account for both, how the velocity is changing in time and how the velocity is changing in space.

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{\partial \vec{v}}{\partial t} + \frac{\partial \vec{v}}{\partial x}\frac{dx}{dt} + \frac{\partial \vec{v}}{\partial y}\frac{dy}{dt} + \frac{\partial \vec{v}}{\partial z}\frac{dz}{dt} \tag{17}$$

It is important to observe that

$$v_x = \frac{dx}{dt},\ v_y = \frac{dy}{dt},\ v_z = \frac{dz}{dt}. \tag{18}$$

Meaning the acceleration equation simplifies down to

$$\vec{a} = \frac{\partial \vec{v}}{\partial t} + \frac{\partial \vec{v}}{\partial x}v_x + \frac{\partial \vec{v}}{\partial y}v_y + \frac{\partial \vec{v}}{\partial z}v_z \tag{19}$$

Note that this acceleration is not the rate of change of velocity at the fixed point in space, but actually the change of velocity of a fluid particle as it moves through space.

Note: Gradient Operator

Gradient Operator (∇) can be applied onto scalar fields to measure how steeply the field rises in each direction. Consider $f(x, y, z)$, then the gradient of f will be,

$$\begin{aligned} \vec{\nabla} &= \frac{\partial}{\partial x}\hat{\imath} + \frac{\partial}{\partial y}\hat{\jmath} + \frac{\partial}{\partial z}\hat{k} \\ \vec{\nabla} f &= \frac{\partial f}{\partial x}\hat{\imath} + \frac{\partial f}{\partial y}\hat{\jmath} + \frac{\partial f}{\partial z}\hat{k} \end{aligned} \tag{20}$$

The gradient operator can also be used to acquire the divergence of a vector field. This returns a scalar value which basically signifies the relationship between the amount coming in and the amount going out of the single point. Where positive divergence indicates a "source" or an outward flow from a point in a vector field, while negative direction is referred to as a "sink", acting oppositely to the "source".

$$\begin{aligned} \vec{\nabla} \cdot \vec{v} &= \left(\hat{\imath}\frac{\partial}{\partial x} + \hat{\jmath}\frac{\partial}{\partial y} + \hat{k}\frac{\partial}{\partial z}\right) \cdot (v_x\hat{\imath} + v_y\hat{\jmath} + v_z\hat{k}) \\ &= \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \end{aligned} \tag{21}$$

Recall the gradient partial differential operator, ∇. Now, make the connection that

$$\begin{aligned} (\vec{v} \cdot \vec{\nabla}) &= v_x\frac{\partial}{\partial x} + v_y\frac{\partial}{\partial y} + v_z\frac{\partial}{\partial z} \\ (\vec{v} \cdot \vec{\nabla})\vec{v} &= v_x\frac{\partial \vec{v}}{\partial x} + v_y\frac{\partial \vec{v}}{\partial y} + v_z\frac{\partial \vec{v}}{\partial z} \end{aligned} \tag{22}$$

Hence, the acceleration equation simplifies even further,

$$\vec{a}(x, y, z, t) = \frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \vec{\nabla})\vec{v} = \frac{\partial \vec{v}}{\partial t} + v_x\frac{\partial \vec{v}}{\partial x} + v_y\frac{\partial \vec{v}}{\partial y} + v_z\frac{\partial \vec{v}}{\partial z} \tag{23}$$

Note: Material Derivative Operator

From the previously known facts, it is important to establish a connection to the material derivative. Material derivative describes the rate of change of a property, as we follow the particle through space and time.

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + (\vec{v} \cdot \vec{\nabla}) \tag{24}$$

The $\partial/\partial t$ term is the local derivative, while the $\vec{v} \cdot \vec{\nabla}$ is the convective derivative term, measuring the change in a property as we move through space.

Thus, after considering the new material derivative, the final form of acceleration of a particle as it moves through space and time can now be defined as,

$$\vec{a} = \frac{D\vec{v}}{Dt} = \frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \vec{\nabla})\vec{v} = \frac{\partial \vec{v}}{\partial t} + v_x \frac{\partial \vec{v}}{\partial x} + v_y \frac{\partial \vec{v}}{\partial y} + v_z \frac{\partial \vec{v}}{\partial z} \tag{25}$$

2.3.2 Fluid motion

Consider the movement of a small cubic volume of a fluid element, from time t to $t + \delta t$. The fluid may have undergone translation, rotation, deformation or even shape change. This next section of the Literary Review will study the motions briefly.

For translational motion, consider a point P at time t, undergoing a uniform flow $\vec{v}(x, y, z, t)$. In this case the velocity field is spatially uniform around the element so all spatial derivatives of the velocity vanish. The fluid will only translate in the direction of the velocity without changing its shape. Mathematically, this can be expressed as,

$$\nabla \vec{v} = \mathbf{0} \quad \Longrightarrow \quad \frac{\partial v_i}{\partial x_j} = 0 \quad (i, j = x, y, z). \tag{26}$$

However, if the velocity field is non-uniform, things get complicated. For now, assume that the flow will change only in the x direction, meaning $\vec{v}(x, y, z, t) = v_x(x, t)\hat{i}$. Now, consider a small fluid element, with a volume $\delta V = \delta x \delta y \delta z$. Since deformation is occurring, the fluid element will undergo change in volume. As shown in Figure 10, consider the segments AB and CD of the fluid at time t. At time $t + \delta t$ the fluid element has both translated and deformed. As it moves, segment AB has moved to the position $(x) + v_x(x)\delta t$, and segment CD has moved to the position $(x + \delta x) + v_x(x + \delta x)\delta t$.

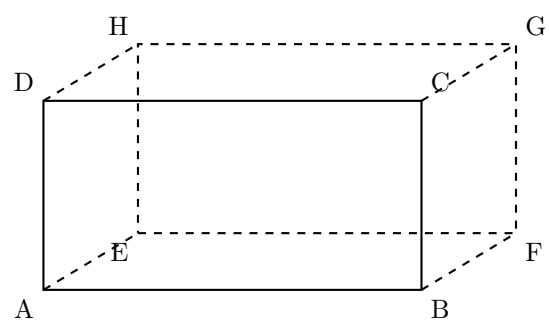


Figure 10: *3D schema of a fluid element*

Which means that the change in volume of the fluid element will be:

$$\begin{aligned}\Delta(\delta V) &= \Big[\big((x+\delta x + v_x(x+\delta x)\delta t) - (x + v_x(x)\delta t)\big)\delta y \delta z\Big] - \delta x \delta y \delta z \\ &= \big[v_x(x+\delta x) - v_x(x)\big]\delta t \delta y \delta z\end{aligned} \tag{27}$$

Now after applying the first order Taylor expansion terms on this equation, the function simplifies as shown in Equation (**??**)

$$v_x(x+\delta x) \quad \Longrightarrow \quad v_x(x) + \frac{\partial v_x}{\partial x}\delta x$$

$$\begin{aligned}\big[v_x(x+\delta x) - v_x(x)\big]\delta t \delta y \delta z &= \frac{\partial v_x}{\partial x}\delta x \delta t \delta y \delta z \\ &= \frac{\partial v_x}{\partial x}\delta t \delta V\end{aligned} \tag{28}$$

Note: Taylor Expansion

It is crucial to grasp the Taylor Expansion of $f(x+h)$. First, start with an ordinary Taylor Series expansion,

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 \tag{29}$$

Now, what does the $x - x_0$ terms mean? For convergence to occur, usually this needs to be a small value, giving it an appropriate name h. Substituting $x - x_0 = h$, the equation becomes,

$$f(x_0 + h) \approx f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2}h^2 \tag{30}$$

Now, if we use this new formula for our example we would get:

$$v_x(x+\delta x) \approx v_x(x) + \frac{\partial v_x}{\partial x}\delta x + \frac{\partial^2 v_x}{\partial x^2}\frac{(\delta x)^2}{2} \tag{31}$$

By only taking the first-order linear terms in this expansion, the simplified formula shown in Equation (28).

The rate at which the volume is changing divided by the original volume of the fluid is called the volumetric dilation rate.

$$\frac{1}{\delta V}\lim_{\delta t \to 0}\frac{\Delta(\delta V)}{\delta t} = \frac{\partial v_x}{\partial x} \tag{32}$$

It is important to bring up that currently only the x direction is being considered. If there are non-zero changes in other components of the velocity, then the volume dilation rate can be generalised to this equation,

$$\begin{aligned}\frac{1}{\delta V}\lim_{\delta t \to 0}\frac{\Delta(\delta V)}{\delta t} &= \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \\ &= \vec{\nabla} \cdot \vec{v}\end{aligned} \tag{33}$$

Also, recall that the density of an incompressible fluid cannot change, therefore no volume change, only transitional displacement. Therefore, the velocity field for an incompressible fluid must satisfy the condition that the divergence of it is 0.

$$\vec{\nabla} \cdot \vec{v} = 0 \tag{34}$$

2.3.3 Navier-Stokes

The Navier-Stokes equations are a complicated set of Partial Differential Equations which describe the fundamental mechanics of fluids, acting as a statement of conservation of momentum. While they are too complex to compute in real-time for a simulation, they represent the physical ground truth. The general form is as shown in Equation (35).

$$\rho\left(\frac{\partial\vec{v}}{\partial t} + (\vec{v}\cdot\vec{\nabla})\vec{v}\right) = -\vec{\nabla}p + \mu\nabla^2\vec{v} + \vec{F}, \tag{35}$$

Where ρ (rho) is the fluid density, $\vec{v}$ is the velocity vector and p denotes the scalar fluid pressure. μ (mu) is the dynamic viscosity of the fluid. $\vec{F}$ represents external body forces per unit volume, which in our case is gravity, $\vec{F} = \rho\vec{g}$.

The term on the left, $\rho\left(\frac{\partial\vec{v}}{\partial t} + (\vec{v}\cdot\vec{\nabla})\vec{v}\right)$, represents the inertia of the fluid element (mass times acceleration). The expression in the parenthesis is the material derivative of velocity, which describes the acceleration of a fluid particle.

On the right side, we have the forces acting on the fluid. The term $-\nabla p$ is the pressure gradient force, which shows that fluid moves from areas of high pressure to areas of low pressure. The term $\mu\nabla^2\vec{v}$ represents the viscous force, accounting for internal friction within the fluid [30]. The Laplacian operator (∇^2) measures how the velocity at a point differs from its neighbours, smoothing out sharp velocity changes.

2.4 Phillips spectrum

The ocean surface waves can exist across many different kinds of shapes and scales. Ranging from wind driven swells which span hundreds of metres to tiny ripples. To account for all of these details, a statistically accurate model is used, a spectrum. This spectrum will be able to assign an energy value to each wave component.

The Phillips spectrum provides exactly this, it is a well known spectra of sea waves in the presence and in the absence of wind [2] which will allow us to simulate the ocean accurately. This spectra is a 2D wave vector domain, where each wave vector is given by $\vec{k} = (v_x, v_y)$ and magnitude $k = ||\vec{k}||$. This section will explore how the equation is formed following Tessendorf's method [31].

2.4.1 Theoretical Derivation

Note: Dimensional Analysis

Dimensional Analysis is a tool often used in Physics and Engineering where by comparing units an equation is checked for consistency. The square brackets around a variable signify that the units are extracted. Common examples of units are listed, but not limited to, the ones in Equation (36)

$$\begin{aligned} T &\Longrightarrow \text{time}(s) \\ M &\Longrightarrow \text{mass}(kg) \\ L &\Longrightarrow \text{length}(m) \\ K &\Longrightarrow \text{temperature}(K) \\ I &\Longrightarrow \text{current}(A) \\ [\vec{v}] = \frac{[\vec{s}]}{[t]} &\Longrightarrow \frac{\mathrm{L}}{\mathrm{T}} = \mathrm{LT}^{-1} \end{aligned} \tag{36}$$

Phillips (1958) [23] has observed that at high wavenumbers (low wavelengths), the equilibrium range is often continuously supplied by the wind and removed by breaking, or leaving the spectrum in a ready state. Since wind speed has no effect, the only physically relevant properties that will act are the acceleration due to

gravity g and the wave number $\vec{k}$ on the surface elevation variance spectrum $h(k)$. Dimensional analysis can confirm the form, as shown here:

$$h(k) \propto g^a k^b \tag{37}$$

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

$$[g^a k^b] = (\mathrm{LT}^{-2})^a (\mathrm{L}^{-1})^b = \mathrm{L}^{a-b}\mathrm{T}^{-2a} \tag{38}$$

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

$$h(k) = \alpha_P \frac{g^2}{k^4} \tag{39}$$

where $\alpha_P \approx 0.0081$ is the Phillips Constant, which was determined from empirical ocean measurements [23], meaning it was not derived, but made to fit observations.

2.4.2 Directionality and Scale

However, there is an issue. The k^{-4} diverges to ∞ as $k \to 0$, which implies infinite energy at really large wavelengths. Additionally, the wind speed V can only sustain coherent waves up to the scale at which wave phase speed $c = \sqrt{\frac{g}{k}}$ equals to V, giving a maximum wavelength as shown in Equation (40).

$$L = \frac{V^2}{g} \tag{40}$$

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

The original formulation of Phillips' in Equation (39) is isotropic, meaning energy is distributed in all directions equally. This is not the case for real life, as the wind usually drives the direction of the waves. Hence, by following Tessendorf's [31] implementation, a directional weighting, D, is introduced.

$$D(\vec{k}, \hat{w}) = \left|\hat{k} \cdot \hat{w}\right|^2 = \cos^2\theta \tag{41}$$

where $\hat{w}$ is the wind unit vector and θ is the angle between $\vec{k}$ and $\hat{w}$. This helps to keep most of the energy pointing in the direction of the wind when $\theta = 0$, and suppress other waves pointing perpendicular to wind when $\theta = \pi/2$.

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.

$$G(k, l) = \exp\left(-k^2 l^2\right) \tag{42}$$

This filter eliminates this energy above the cut-off scale l [31].

Hence, by combining the equations for the range (39), wind-scale cut-off (40), the directional factor (41) and the small scale damping (42), the Phillips spectrum will be as described in Equation (43). This formulation follows the Tessendorf [31], who was able to unify these theories together into a practical model.

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

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

2.4.3 Statistics and Time Evolution

Although the model is complete, it serves no use just yet. 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]. Justified by the Central Limit Theorem (CLT), the surface elevation is the superposition of many individual waves trains, which by CLT converge to Gaussian statistics.

Therefore, for each vector $\vec{k}$, two independent samples are required. To accurately model a Gaussian Random Field, the Box-Muller Transform is used. This transform takes two uniform random numbers $u_1, u_2 \in (0, 1]$ and maps them to a standard normal distribution $\mathcal{N}(0, 1)$ as seen in Equation (44).

$$\xi = \sqrt{-2\ln(u_1)}\cos(2\pi u_2) \tag{44}$$

By utilising this transform, two samples of the Gaussian Random Field are drawn, ξ_i, ξ_r. This forms the complex Fourier amplitude as described in Equation (45).

$$\tilde{h}_0(\vec{k}) = \frac{\xi_r + i\,\xi_i}{\sqrt{2}}\sqrt{P(\vec{k})} \tag{45}$$

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 $\tilde{h}_0(\vec{k})$, a complex conjugate $\tilde{h}_0^*(-\vec{k})$ is stored. 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, meaning for the resultant surface to be made up of purely real values ($\eta(x, z, t) \in \mathbb{R}$), the Fourier coefficients must satisfy the relation $\tilde{h}(-\vec{k}) = \tilde{h}^*(\vec{k})$.

Time Evolution is then controlled by the deep-water dispersion relation, covered in Section 2.1.3, repeated in Equation (46)

$$\omega(\vec{k}) = \sqrt{g\,||\vec{k}||} \tag{46}$$

This shifts the phase of each mode over time, as governed in Equation (47)

$$\tilde{h}(\vec{k}, t) = \tilde{h}_0(\vec{k})\,\exp\Big(i\omega(\vec{k})\,t\Big) + \tilde{h}_0^*(-\vec{k})\,\exp\Big(-i\omega(\vec{k})\,t\Big) \tag{47}$$

An Inverse Fast Fourier Transform (IFFT) over the full $N \times N$ grid produces a spatial surface $\eta(x, t)$ at each timestep, which is already usable for rendering inside shaders. This discussion expands more on the meaning of an IFFT in Section 2.5

2.5 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform is a technique used to convert data from time or space domain to frequency domain. The DFT is also able to split up data into smaller chunks that it's made up of as shown in Figure 11. 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 $\vec{X}$ with also N elements $(X_0, X_1, X_2, \ldots, X_{N-1}) = X_k$. Where X_k is to be defined as in Equation (48). [26]

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N}kn} \tag{48}$$

Which by substituting $\exp\left(\frac{-2\pi i}{N}\right) = \nu$, a simplified form is acquired as shown in Equation (49) [26]

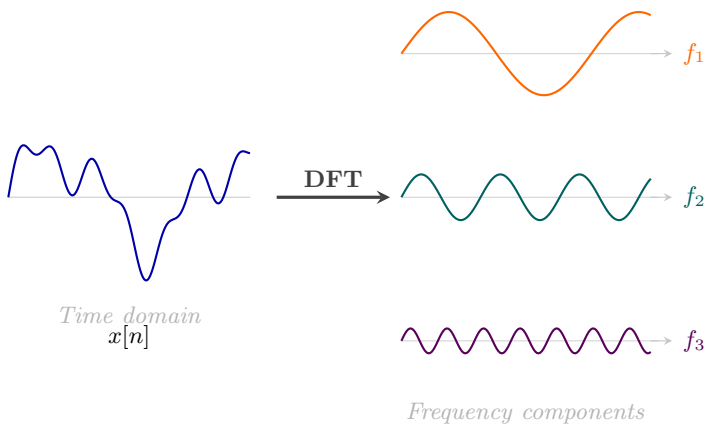


Figure 11: *The DFT decomposes a composite discrete signal $x[n]$ into its sinusoidal components at distinct frequencies f_1, f_2, f_3, each with its own amplitude and phase.*

$$X_k = \sum_{n=0}^{N-1} x_n \nu^{kn} \tag{49}$$

where the ν term can be expressed in terms of trigonometric functions using Euler's formula and De Moivre's theorem as shown in Equation (50).

$$\begin{aligned}
\nu &= \exp\left(\frac{-2\pi i}{N}\right) \\
\nu &= \cos\left(\frac{-2\pi}{N}\right) + i\sin\left(\frac{-2\pi}{N}\right) \\
\nu &= \cos\left(\frac{2\pi}{N}\right) - i\sin\left(\frac{2\pi}{N}\right) \\
\nu^{kn} &= \left[\cos\left(\frac{2\pi}{N}\right) - i\sin\left(\frac{2\pi}{N}\right)\right]^{kn} \\
\nu^{kn} &= \cos\left(kn\,\frac{2\pi}{N}\right) - i\sin\left(kn\,\frac{2\pi}{N}\right)
\end{aligned} \tag{50}$$

To calculate a single frequency X_k, the algorithm has to sum up N multiplications and since there is N amount of elements, the number of operations becomes $N \times N$. The complexity of this transform is $O(n^2)$, which is very inefficient for larger grid sizes, therefore a different approach has to be taken, commonly referred to as a Fast Fourier Transform (FFT).

2.5.1 Fast Fourier Transform (FFT)

To solve the issue of polynomial growth described earlier, a different algorithm is introduced, whose most common example is a Cooley-Tukey Radix-2 Decimation-In-Time. Radix-2 term refers to a FFT algorithm which cuts down the big DFT by a factor of 2 every step it takes. This requires for the number of elements to be a power of 2 ($N = 2^k, k \in \mathbb{Z}$). This reduces the complexity of the algorithm to $O(n \log n)$, and works by computing separate DFT's of even inputs $(x_0, x_2, x_4, \dots)$ and odd inputs $(x_1, x_3, x_5, \dots)$ [26]. This section will now derive the formula for the Radix-2 algorithm by splitting up the DFT shown in Equation (49) into a sum as shown in Equation (51).

$$
\begin{aligned}
X_k &= \sum_{m=0}^{N/2-1} x_{2m}\, e^{\frac{-2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1}\, e^{\frac{-2\pi i}{N}(2m+1)k} \\
X_k &= \sum_{m=0}^{N/2-1} x_{2m}\, e^{\frac{-2\pi i}{N}(2m)k} + e^{\frac{-2\pi i}{N}} \sum_{m=0}^{N/2-1} x_{2m+1}\, e^{\frac{-2\pi i}{N}(2m)k}
\end{aligned}
\tag{51}
$$

Now, one can see that in Equation (51) consists of two sums which are the DFTs of even indexed part (E_k) and odd indexed part respectively (O_k), which can be rewritten as (52).

$$
X_k = E_k + e^{\frac{-2\pi i}{N}k}\, O_k \tag{52}
$$

Since the functions of k, E_k and O_k are periodic with periodicity of $N/2$, Equation (52) can be rewritten as shown in Equation (53).

$$
\begin{aligned}
E_{k+N/2} &= E_k \\
O_{k+N/2} &= O_k \\
X_k &= \begin{cases} E_k + e^{\frac{-2\pi i}{N}k}\, O_k, & \text{for } 0 \leq k < N/2 \\ E_{k-N/2} + e^{\frac{-2\pi i}{N}k}\, O_{k-N/2}, & \text{for } N/2 \leq k < N \end{cases}
\end{aligned}
\tag{53}
$$

Using the following property of the twiddle factor [26],

$$
e^{\frac{-2\pi i}{N}(k+N/2)} = e^{\frac{-2\pi ik}{N} - \pi i} = e^{-\pi i}\, e^{\frac{-2\pi ik}{N}} = -e^{\frac{-2\pi ik}{N}} \tag{54}
$$

X_k can be rewritten as shown in Equation (55) [26].

$$
\begin{aligned}
X_k &= E_k + e^{\frac{-2\pi ik}{N}}\, O_k \\
X_{k+N/2} &= E_k - e^{\frac{-2\pi ik}{N}}\, O_k
\end{aligned}
\tag{55}
$$

This is the core principle of the Radix-2 transform, as it is able to recursively split up a whole DFT of size N into 2 separate DFT's of size $N/2$, reducing the mathematical complexity as N grows larger. This is also often called butterfly passes.

2.5.2 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 12.

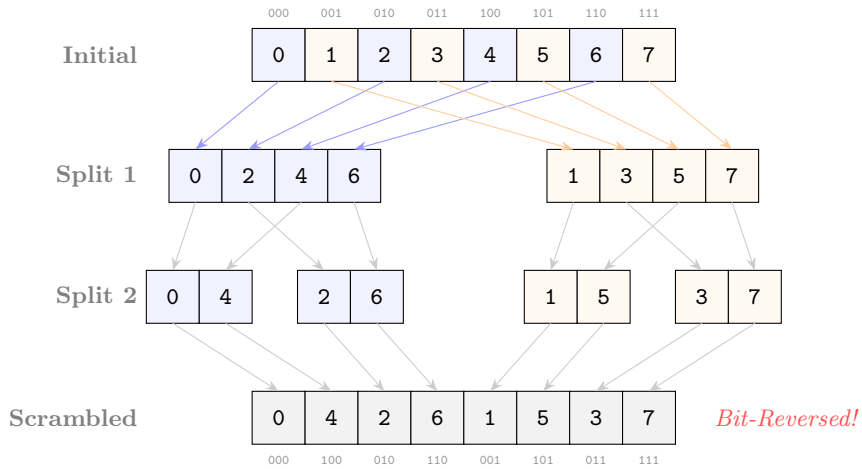


Figure 12: *Recursive DIT of an $N = 8$ array with bitwise representations.*

The binary entries for the data in the scrambled array will actually be bit reversed when compared to the original binary entries in the same position. 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. This eliminates the need for a large additional array by allowing butterfly passes to write onto the same memory location.

2.5.3 Inverse Fast Fourier Transform (IFFT)

Unlike the standard DFT/FFTs which convert spatial or time domain to frequency domain, the Inverse Discrete Fourier Transform does the opposite by converting the frequency domain back to spatial domain. This is especially useful when dealing with oceanographic spectra, like the Phillips spectrum, to acquire the needed height map $\eta(x, t)$ to be used by the vertex shader. The definition of the IDFT is as shown in Equation (56)

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} \tag{56}$$

There are only two main differences to note between the DFT and IDFT: the positive sign switch in the twiddle factor (representing a rotation in the opposite direction on the complex plane), as well as a $1/N$ scaling factor which normalises the signal. The advantage of the Cooley-Tukey algorithm discussed in Section 2.5.1 is that the same radix-2 logic can be applied for both the forward and inverse transforms. Therefore, the IFFT can be calculated using the standard FFT function by following the steps as follows:

1. Conjugate the complex input (swapping the real and imaginary parts)
2. Perform standard radix-2 FFT.
3. Conjugate the complex input again
4. Add a scaling factor of $1/N$

2.6 Lighting Models

Although the physics and maths behind wave propagation may be in place, an important topic is yet to be discussed. The user will have to see an output on the screen, therefore a lighting model has to be in place and this section will be about that. However, before exploring different strategies, it is important to understand how any rendering system works.

First and foremost, the CPU does all the logic, it sets up the scene, loads up necessary data and initialises any processes. The general architecture of this rendering pipeline is as shown in Figure 13. Once the CPU

has prepared the environment, it sends a draw call to the GPU. The GPU is able to use vertex and fragment shaders to do calculations on the data sent by the CPU and display an image to the screen. Some readers may question, "Why is the GPU doing the rendering and not the CPU?". And the truth is that the CPU is horrible at parallelism, it can only perform a single task, a single calculation at a time, but really fast. Meanwhile the GPU is able to handle lots of simpler calculations in parallel, therefore being much more efficient for rendering frames with millions, or billions of pixels. Shaders are, in essence, just programs which run on the GPU [4]. The Vertex Shader is responsible for the movement of vertices, while the Fragment Shader is used for colouring in individual pixels. Now, there are more types of shaders, such as the compute shader, but that will be explained in more detail in a later stage.

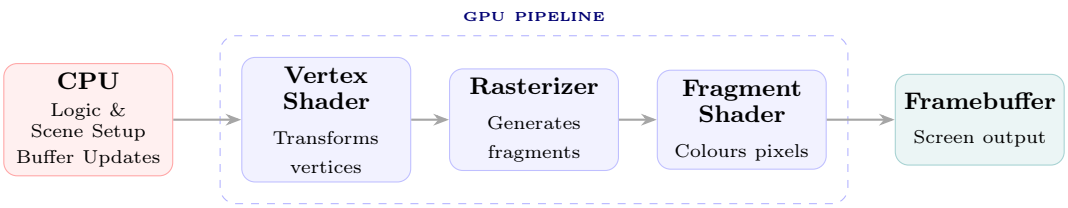


Figure 13: *CPU prepares the scene and issues draw calls to the GPU, which processes geometry through the shader stages before writing pixel colours to the framebuffer.*

2.6.1 Surface Normals

For any lighting model the shader must first determine the surface normal $\hat{n}$ for every point on the wave. This is required since any lighting model will use that to calculate the amount of sunlight reflecting. The surface normal can be defined implicitly by the function $f(\vec{x}, z, t) = z - \eta(\vec{x}, t) = 0$, where $\vec{x} = (x, y)$ represents the horizontal coordinates and η is the vertical displacement. From vector calculus, the normal to such a surface is given by the gradient of the implicit function [31] as shown in Equation (57).

$$\vec{n} = \vec{\nabla} f = \vec{\nabla}(z - \eta(\vec{x}, t)) = \begin{bmatrix} -\frac{\partial \eta}{\partial x} \\ -\frac{\partial \eta}{\partial y} \\ 1 \end{bmatrix} = \begin{bmatrix} -\vec{\nabla}\eta \\ 1 \end{bmatrix} \tag{57}$$

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

$$\hat{n} = \frac{\vec{n}}{||\vec{n}||} \tag{58}$$

In practice, the analytical partial derivatives may not be available, therefore an approximation method has to be 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 (59), where Δs is typically set to the world-space distance between texels.

$$\begin{aligned} \frac{\partial \eta}{\partial x} &\approx \frac{\eta(x + \Delta s, y, t) - \eta(x - \Delta s, y, t)}{2\Delta s} \\ \frac{\partial \eta}{\partial y} &\approx \frac{\eta(x, y + \Delta s, t) - \eta(x, y - \Delta s, t)}{2\Delta s} \end{aligned} \tag{59}$$

2.6.2 Blinn-Phong model

The Blinn-Phong is a great and very efficient lighting technique to convey the feeling of 3D look and feel of an object. Phong lighting is primarily composed from 3 main components: Ambient, Diffuse and Specular light as shown in Figure 14

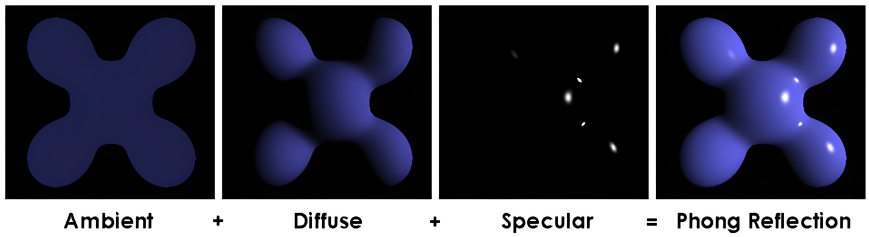


Figure 14: *Breakdown of the Blinn-Phong reflection model components. Adapted from [28].*

Ambient light is the fundamental layer, representing the indirect light coming onto the object. This helps to illuminate and give a base shape to the object even in the dark. This can be thought of as constant light applied to parts of an object, no matter the data.

Diffuse light, on the other hand, requires a light source and will determine how much of the light reflects based on the rotation of the unit surface normal. For example, when the incident light is directly head-on (90 degrees) to the surface, the amount of light reflected will be at its maximum. Whereas, when the surface is tilted to the incident light, the amount reflected will be less. This relationship can be represented using a dot product between the unit surface normal vector and the incident light vector as shown in Equation (60), where $\vec{d_L}$ is the direction of the light source.

$$K_d = \max(\hat{n} \cdot \vec{d_L},\ 0) \tag{60}$$

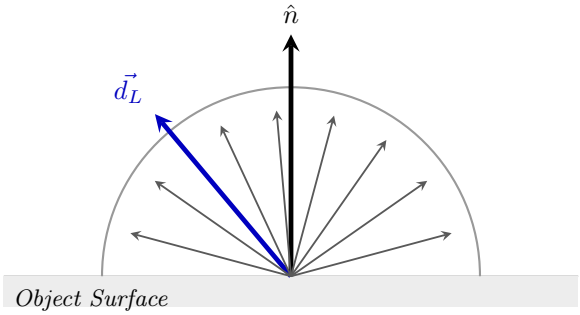


Figure 15: *Diffuse reflection. Incident light $\vec{d_L}$ scatters uniformly across the hemisphere, independent of view direction.*

Therefore, the diffuse component would be the product of K_d and light colour. Lastly, specular lighting refers to the component of light that is responsible for the bright, mirror-like spot of light commonly appearing on metallic and glossy objects. This component requires two inputs: the light source direction $\vec{d_L}$ and the view direction $\vec{v}$. Unlike standard Phong, the Blinn-Phong model avoids computing the reflection vector entirely, and instead uses a halfway vector $\hat{h}$, defined as the normalised bisector between $\vec{d_L}$ and $\vec{v}$. Where the specular strength could then be determined by taking the dot product between the halfway vector and the surface normal as shown in Equation (61), where α is the shininess exponent controlling the size of the highlight.

$$\begin{aligned} \hat{h} &= \frac{\vec{d_L} + \vec{v}}{\|\vec{d_L} + \vec{v}\|} \\ K_s &= \max(\hat{n} \cdot \hat{h},\ 0)^{\alpha} \end{aligned} \tag{61}$$

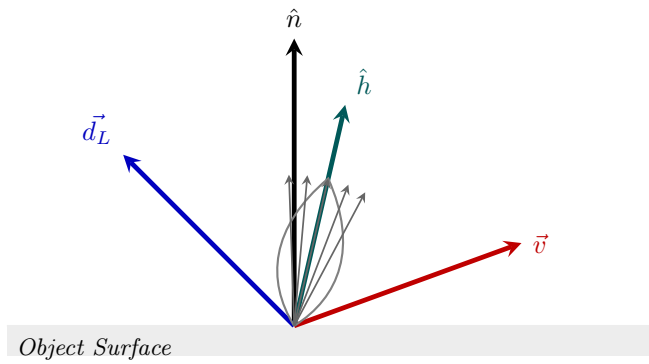


Figure 16: *Blinn-Phong specular geometry. The halfway vector $\hat{h}$ bisects $\vec{d_L}$ and $\vec{v}$; specular intensity is determined by the angle between $\hat{h}$ and the surface normal $\hat{n}$.*

In the end, all the different components are combined together to produce a single output colour as shown in Equation (62), where I_a, I_d and I_s are the ambient, diffuse and specular light intensities, and K_a, K_d and K_s are the corresponding material reflectance coefficients.

$$I = I_a K_a + I_d K_d + I_s K_s \tag{62}$$

2.6.3 Cook-Torence PBR model

Although Blinn-Phong is an excellent efficient model, it is not capable of producing such a realistic result as a Physically Based Rendered (PBR) model would. The ground truth for PBR is the rendering equation as shown in Equation (63), stating that 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.

$$L_o(p, \omega_o) = \int_\Omega f_r(p, \omega_i, \omega_o)\, L_i(p, \omega_i)\, (\hat{n} \cdot \omega_i)\, d\omega_i \tag{63}$$

Where f_r is the Bidirectional Reflectance Distribution Function (BRDF). Because calculating this integral in real-time for every pixel is computationally impossible, the Cook-Torrance microfacet model is used as an approximation. This technique models light at a microscopic level, where by using real physics it calculates how light would behave on rough surfaces. The model is also composed of primarily three parts: the Normal Distribution Function, the Geometry Function and the Fresnel term, which are then combined into a single BRDF as shown in equation (64).

$$\text{Final} = (\text{Diffuse} + \text{Specular}) \cdot \text{LightIntensity} \cdot (\hat{n} \cdot \vec{d_L}) \tag{64}$$

where LightIntensity = LightColor · LightDiffuseIntensity. The diffuse term follows a Lambertian model, where f_{lambert} is simply the surface albedo colour [35]. While the specular term uses a Cook-Torrance BRDF, $f_{\text{CookTorreace}}$, but to ensure energy conservation the specular coefficient is defined as $K_s = (1 - K_d)$, meaning the sum of the diffuse and specular contributions never exceed incoming energy.

$$\begin{aligned} \text{Diffuse} &= K_d \cdot \frac{f_{\text{lambert}}}{\pi} \\ \text{Specular} &= (1 - K_d) \cdot f_{\text{CookTorrance}} \end{aligned} \tag{65}$$

The Cook-Torrance BRDF will be defined as shown in Equation (66), where D, G and F represent the Normal Distribution Function, the Geometric function and the Fresnel function respectively [6]. While the denominator is responsible to ensure that energy remains conserved.

$$f_{\text{CookTorrance}} = \frac{D \cdot G \cdot F}{4 \cdot (\hat{n} \cdot \vec{d_L}) \cdot (\hat{n} \cdot \hat{v})} \tag{66}$$

The Normal Distribution Function D, models the microscopic structure of the surface, as not all surfaces are perfectly smooth, but instead 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 contributte to the specular highlight as shown in Figure 17. The GGX distribution [36] is also used here, as shown in Equation (67), where α is the surface roughness coefficient.

$$D = \frac{\alpha^2}{\pi \cdot \left((\hat{n} \cdot \hat{h})^2 \cdot (\alpha^2 - 1) + 1\right)^2} \tag{67}$$

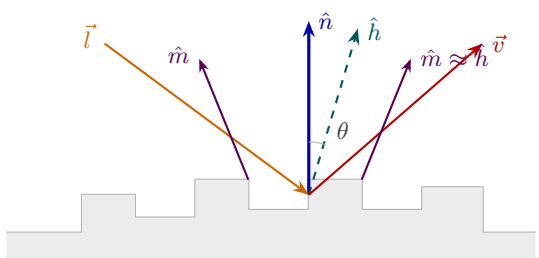


Figure 17: *Microfacet surface model, facets whose normal $\hat{m}$ aligns with the halfway vector $\hat{h}$ contribute to specular reflection (Simplified diagram)*

Even when the microfacet may be orientated towards the half-way vector $\hat{h}$, it may still be partially blocked, since neighbouring microfacets can cast a shadow on it from the light direction (shadowing), or block the reflected ray from reaching the viewer (masking) as shown in Figure 18. The Geometry Function G, accounts for the shadowing and masking effect using the Smith method [15], which evaluates them independently and multiplies the results shown in Equation (68).

$$G = S(\hat{n} \cdot \vec{v}) \cdot S(\hat{n} \cdot \vec{d_L}) \tag{68}$$

where S is the Schlick-GGX approximation as shown in Equation (69).

$$S(d_p) = \frac{d_p}{d_p \cdot (1 - k) + k}, \qquad k = \frac{(\alpha + 1)^2}{8} \tag{69}$$

This is the core term that prevents PBR materials from appearing unrealistically bright at grazing angles, which can be seen more commonly in simpler models like the Blinn-Phong, previously discussed.

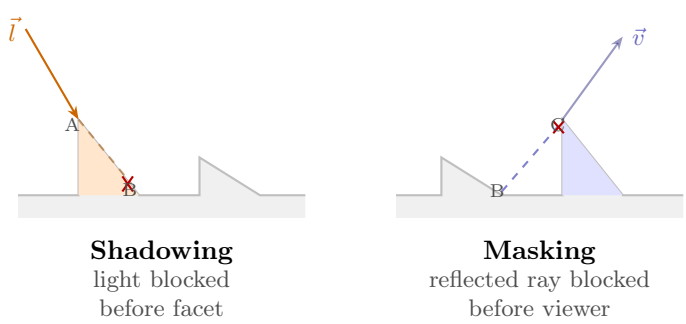


Figure 18: *Shadowing and masking. The geometry function G attenuates the contribution of affected facets.*

Lastly, the Fresnel term describes how the viewing angle changes the reflectivity term. At grazing angles all surfaces become highly reflective regardless of the material. The F_0 term represents the base reflectivity at normal incidence. The Schlick approximation [27] is used again, as shown in the Equation (70).

$$F = F_0 + (1 - F_0) \cdot (1 - (\vec{v} \cdot \hat{h}))^5 \tag{70}$$

The F term will drive reflectivity towards 1 as the angle between $\vec{v}$ and $\hat{h}$ increases, which produces that realistic edge highlight.

After combining all three terms, the final full PBR is defined by the Equation (71)

$$\text{PBR} = \left(\frac{K_d \cdot \text{Color}}{\pi} + \frac{(1 - K_d) \cdot D \cdot G \cdot F}{4 \cdot (\hat{n} \cdot \vec{l}) \cdot (\hat{n} \cdot \vec{v})} \right) \cdot \text{LightIntensity} \cdot (\hat{n} \cdot \vec{l}) \tag{71}$$

3 Discussion

This dissertation will now cover the practical part of the project, where the knowledge gained about Fourier Transforms, Oceanographic spectrums and buffers becomes applicable. It is important to understand that although theoretical concepts like the Navier-Stokes or Gerstner Waves were covered, they are not considered for the use in the simulation. It is beneficial to understand the underlying principles and core mechanisms that make up reality, as well as alternatives to other wave forms, even if there isn't a clear pathway of them being used. With that out of the way, this section will now cover the initial setup of the environment and the simulation, outlining the core rendering pipeline and the intent behind every decision.

3.1 Initial Setup

Note: Structs and Methods in Rust

A struct is a data type which can hold multiple related values. Each value has a unique identifier, making it clear what each value is [33]. Each of those fields can then be referenced by `struct_name.field_name`. Figure 19 shows definition and initialisation of a struct.

```rs
struct User {
    id: String,
    active: bool,
    points: u64
}

let user1 = User {
    id: String::from("id123"),
    active: true,
    points: 300
}
let points = user1.points;
```

Figure 19: *An example code defining a struct.*

Now, methods are similar to functions, where they are declared with the same keyword `fn` and can take in parameters and return values [33]. Unlike functions, methods' first parameter will always be `self`. This allows the method to reference and use the fields and other methods available inside the struct. For example deactivating a user as shown in Figure 20

```rs
// Same struct definition
struct User {
    id: String,
    active: bool,
    points: u64
}

// Implement the following methods for a struct...
impl User {
    // A public method (accessible from outside the struct), which takes in a
    mutatable parameter &self
    pub fn deactivate(&mut self) {
        self.active = false;
    }
}

// The struct now has been declared as mut, to allow fields to be changed and
    mutated.
let mut user1 = User {
    id: String::from("id123"),
    active: true,
    points: 300
}
// Calling the method is done similar to fields
user1.deactivate()
```

Figure 20: *An example code defining a struct and a method*

First and foremost, the window has to be created. For that, a library under the name of `winit` will be used. Any Rust application follows the same logic, after the code is compiled, the `main` function in the `main.rs` file will be called. In this case, this function will be responsible for the creation of the event loop, as well as

creating the `App` struct and initialising the state to `None`. The `App` struct implements methods needed for a process to work, such as `resume`, `user_event`, `window_event`, etc.. The resumed method will be called once the application is ready, and is responsible for the creation of the window, as well as calling the startup sequence of the `State` struct as shown in Figure 21.

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

Afterwards, the State struct is responsible for the creation and initialisation of all the variables, the graphics pipeline and initial data. The library responsible for drawing graphics onto the window is `wgpu` [38]. This library allows to initialise shaders, send buffers and control in what way the GPU will work. This is an important part of the abstraction layer, as it allows the programmer to focus on creating main logic rather than designing system calls. Using this library also satisfies the goals of this discussion, as the library doesn't provide any ready physics or graphical engine, only an interface to the hardware itself. Figure 22 displays the startup sequence which is 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()
});

// 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, // For uncapped fps but idle less power
    alpha_mode: surface_caps.alpha_modes[0],
    view_formats: vec![],
    desired_maximum_frame_latency: 2,
};
```

Figure 22: *Fragment of the* `new` *method, which is responsible for the initialisation of wgpu*

3.2 Mesh Formation

3.2.1 Vertex Structure and Data Flow

An ocean simulation works by generating a horizontal grid of vertices, which are then being vertically displaced in the vertex shader in accordance with ocean rules. Each vertex is treated as a discrete data packet containing its coordinates and a unique identifier, as shown in Figure 23. This packet is stored in a Vertex Buffer (Object).

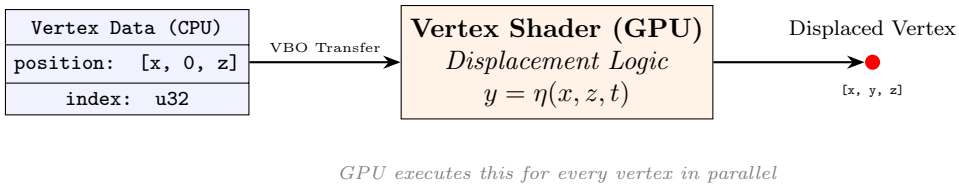


Figure 23: *The flow of a single vertex packet*

3.2.2 Grid Generation and Indexing

First, the mesh, which is a flat plane, has to be created, centred at $(0, 0)$ with physical size L and resolution defined by the number of subdivisions N. The mesh creation process is handled by the `generate_plane` function. To ensure that the mesh tiles and scales correctly, the simulation generates $N + 1$ vertices for N subdivisions. This accounts for the boundary edge that closes the final quad in the grid. The vertices are produced using the logic shown in Equation (72) where col and row iterate through values 0 to N inclusive, preventing a gap at positive x and z boundaries of a tile.

$$x = (\text{col} \times \text{step}) - \frac{L}{2}, \quad z = (\text{row} \times \text{step}) - \frac{L}{2} \tag{72}$$

The VBO stores the raw data of the vertices, while the Index Buffer (Object) defines how the vertices are connected to form a visible surface. In `wgpu`, geometry is rendered as a series of triangles, also called fragments. As shown in figure 24, each square in the grid is decomposed into two triangles. The order in which these indices are stored, such as the winding order, determines the face normal of the triangle. A consistent counter clockwise is maintained to ensure back-face culling can take place, an optimisation technique automatically done by `wgpu` which doesn't render the triangles facing away from the camera.

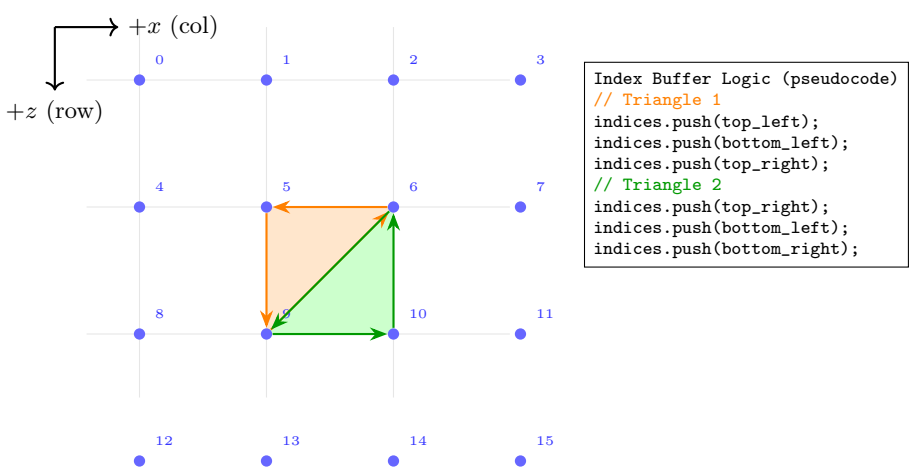


Figure 24: *Vertices are stored in a 1D array using row-major indexing (row × N + col)*

3.3 Vertex Displacement

3.3.1 Temporal Updates and Uniform Buffers

Now that the mesh is generated, the vertex shader can do its job of displacing the y coordinate of each vertex based on a function. However, currently there is not enough data to produce an output. The sine wave has to be constantly shifted by the time, which the GPU does not have access to. To solve this issue, a Uniform Buffer is created, which is capable of storing values to be passed into the GPU each render call. Therefore, the State struct will now contain an `update` method, which is called every frame. The method will have to

know how much time passed since the last call, *dt*, therefore the struct now also contains an additional field `last_frame_time_instant`. Figure 25 shows this code in action, where `time_uniform` and `time_buffer` are new fields with an `increment_time` method in the state struct.

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

3.3.2 Data Transmission and Shader Logic

To transfer the data from the CPU side to the GPU side of the program, `wgpu` utilises bind groups and pipeline layouts. A bind group layout acts as a template, which defines the slots (bindings) that a shader will expect with their data types. For this section, the bind group will be responsible to pass along the `time_uniform` buffer as shown in Figure 26.

```rs
// Buffer that will hold the uniform struct.
let time_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("time_buffer"),
    contents: bytemuck::cast_slice(&[time_uniform]),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

// Layout defining what data to expect, as well as the capabilities.
let time_bind_group_layout =
    device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
        entries: &[wgpu::BindGroupLayoutEntry {
            binding: 0,
            visibility: wgpu::ShaderStages::VERTEX
                | wgpu::ShaderStages::FRAGMENT
                | wgpu::ShaderStages::COMPUTE,
            ty: wgpu::BindingType::Buffer {
                ty: wgpu::BufferBindingType::Uniform,
                has_dynamic_offset: false,
                min_binding_size: None,
            },
            count: None,
        }],
        label: Some("time_bind_group_layout"),
    });

// Bind group itself, what will be passed onto the GPU.
let time_bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
    layout: &time_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: time_buffer.as_entire_binding(),
    }],
    label: Some("time_bind_group"),
});
```

Figure 26: *A modified section of code responsible for creating a bind group*

Now, that the data is accessible by the GPU, the render pipeline has to be configured. It acts as a "global state" for the draw call, outlining the shaders, configuration on the vertex buffer and what way to handle the depth stencil. Now the vertex shader can be configured to accept a `TimeUniform` bind group, and apply the displacement onto the vertices. To verify the functionality of the code, a simple sine wave across x axis and a cosine wave along z axis can be applied as shown in Equation (73), where A is the amplitude and f is the frequency.

$$\eta(x, z, t) = A \cdot \sin(x \cdot f + t) + A \cdot \cos(z \cdot f + t) \tag{73}$$

The vertex shader will be defined as shown in Figure 27.

```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 27: *Mock WGSL code that will be used in the vertex shader*

The output produced will be similar to the early build of the program, as shown in Figure 28. It is important to note that this mesh includes a texture stretched over the mesh, to give more depth, however this is a temporary fix.

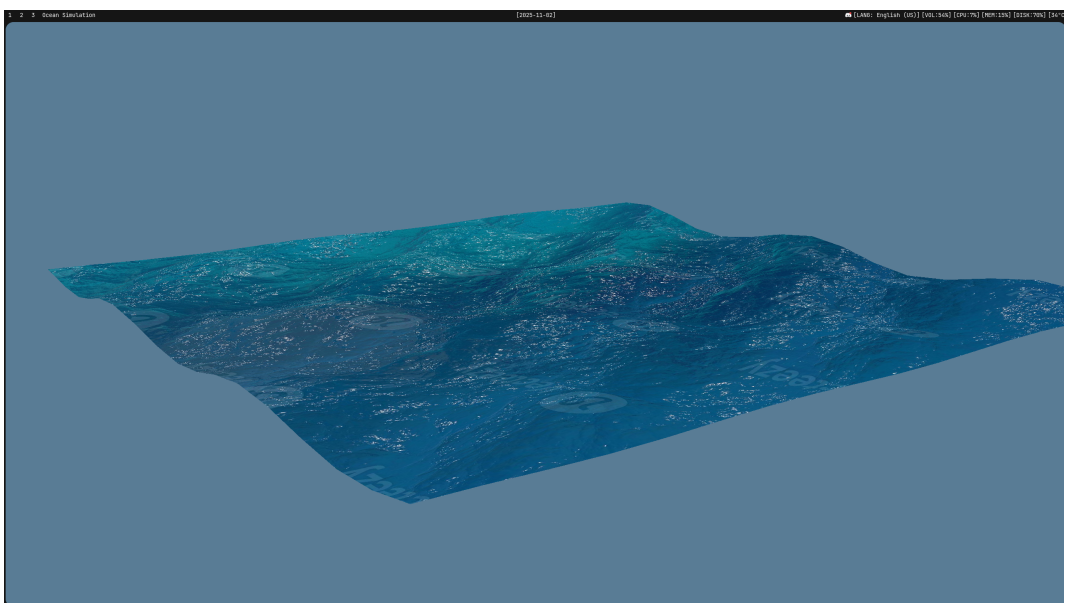


Figure 28: *Simple sine & cosine wave displacement of the vertex. Build v0.1.1*

3.4 The Camera

Currently, the output of the program will produce an image, but it is not visually compelling and hard to use. This can be improved by implementing a skybox, which is the sky of the whole scene, as well as creating a camera which the user can move.

3.4.1 Direction and Spherical Projections

First, and foremost, a camera has to be implemented. A camera can be modelled essentially as a drone. For this, three pieces of information have to be known, the position of the camera in world space, the yaw and pitch of the camera (combined give direction where the camera is looking), and how wide the projection is (FOV). The direction (forward) vector is derived by converting the spherical rotation into the Cartesian coordinates. Yaw (α) is responsible for the rotation around the y axis (left and right), while the pitch (θ) is the rotation around the x axis (up or down). This vector can be thought of as being enclosed in a sphere as shown in Figure 29, and calculated as in Equation (74).

$$\vec{v} = \langle \cos\theta \cdot \cos\alpha,\ \sin\theta,\ \cos\theta \cdot \sin\alpha \rangle \tag{74}$$

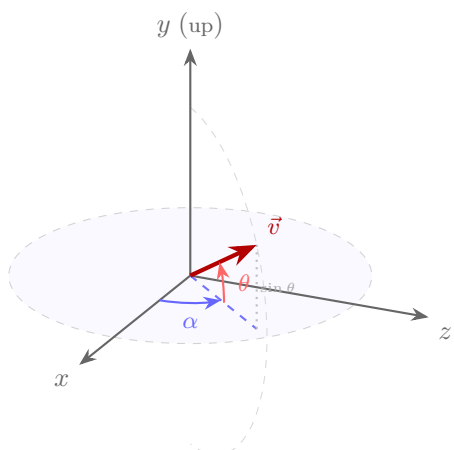


Figure 29: *Camera direction vector $\vec{v}$ where θ represents the elevation (pitch) from the xz-plane.*

3.4.2 Viewing Frustum and Matrices

A projection matrix is required. This matrix is responsible for transforming the world space to view space. Normally, every object has a world position, for example a wave at $(20, 0)$. However, the GPU needs to know where that wave is relative to the user (view space). It is a common trick in the game industry that if the user moves by 5 units forward, it's mathematically easier to shift the whole world 5 units backwards. This is where the view matrix comes in, it is basically the inverse of the camera's transformation matrix. It translates and transforms the world in accordance to make the camera always be at $(0, 0, 0)$ looking forward. A camera can also be represented as a frustum, which captures anything that falls within it. This is the projection matrix, which is calculated using the `perspective` function provided by `cgmath` and contains linear algebra beyond the scope of this research. This function takes in multiple parameters, such as the FOV_y, aspect ratio, z_{near} and z_{far}. These parameters dictate how wide the viewing angle is, how square the image is, how close and how far the user can see, as shown in Figure 30.

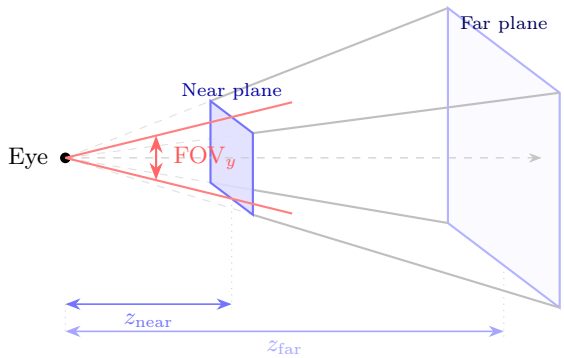


Figure 30: *The viewing frustum defined by the projection matrix. Only geometry between the near and far planes, inside the truncated pyramid, is rendered. FOV_y sets the vertical viewing angle; the aspect ratio scales the horizontal spread. z_{near} and z_{far} clip geometry that is too close or too distant.*

An example of the final code responsible for building the view matrix can be seen in Figure 31.

```rs
pub fn build_view_projection_matrix(&mut self) -> Matrix4<f32> {
    // extract sin & cos components for both pitch and yaw
    let (sin_pitch, cos_pitch) = self.pitch.0.sin_cos();
    let (sin_yaw, cos_yaw) = self.yaw.0.sin_cos();

    // construct the forwards pointing vector as shown
    self.forward =
        Vector3::new(cos_pitch * cos_yaw, sin_pitch, cos_pitch * sin_yaw).normalize();

    // target will be the current position of the camera, plus the facing direction
    let target = self.eye + self.forward;

    // View matrix is built using the right hand method provided by cgmath library
    let view = Matrix4::look_at_rh(self.eye, target, self.up);

    // The view frustum, what the user will see
    let proj = perspective(Deg(self.fovy), self.aspect, self.znear, self.zfar);

    proj * view
}
```

Figure 31: *Modified code snippet of the building process of the view projection matrix*

Now, the `CameraUniform` struct is complete with a method for building the projection matrix and values to update, such as the position of the camera, the controls have to be done. For that, a `CameraController` struct is created, containing methods for updating the camera each frame. User input is handled by the method `handle_window_event` and `handle_user_event`, which passes the events to the corresponding handlers inside `CameraController`. This code will not be displayed, as it is considered trivial, however it is important to note that the pitch axis should be restricted to a range of $[-89°, 89°]$ to prevent the camera from flipping upside down.

3.5 The Skybox

3.5.1 Cubemapping and Depth Manipulation

After the camera, a skybox is really important to implement, as it is the primarily light source for the ocean. This is because the frensel, and reflections will be calculated using the skybox colour. The skybox is primary a $1 \times 1 \times 1$ cube, but rendered at the furthest possible depth, which creates the illusion of a vast atmosphere.

To render a convincing sky, a cubemap is applied. Unlike a traditional two dimensional texture, a cubemap consists of 6 individual faces, all laid out on a single image, each representing the faces of a cube. Sampling from a cubemap does not use standard two dimensional coordinates (u, v) either, a three dimensional vector $\vec{r}$ is used. The GPU determines which face of the cube the vector points towards and samples the texels accordingly. An example of a cubemap is present in Figure 32

Figure 32: *Example of a cubemap (Sorsele). Reproduced from [22].*

The skybox is kept static as the camera moves, this is achieved by stripping the translation data from the view matrix before rendering the skybox. In other terms, the 4×4 matrix M is reduced to only the upper left 3×3 submatrix as shown in Equation (75). This sets the translational components to zero, while still maintaining the rotation.

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{75}$$

To ensure that the skybox is always behind the ocean geometry, the rendering has to be done with a special depth setting. By setting the vertex output z component to its w component in the vertex shader ($z = w$), the depth value will always result in a value of 1.0. This will place the skybox as far as possible, where in combination with a `LessEqual` depth test, the skybox will only be drawn on pixels with no other geometry has been rendered.

3.5.2 Procedural Atmospheric Simulation

However, there is another approach to bringing content to a skybox - procedural shaders. A custom shader can be written to dynamically generate the sky, the sun, clouds and simulate the time of day. This is the more standard approach used by programmers, allowing for much more freedom and customisability. This can be done by creating a new `Skybox` struct. This struct, initialised inside of `State::new()`, will be responsible for the creation of the skybox shader, the skybox render pipeline and needed buffers. It is important to make sure that the `front_face` property inside of the `RenderPipelineDescriptor` is set to `FrontFace::Cw` (clockwise). This will effectively flip the mesh inside out, and allow the user inside of it to see the rendered

skybox. Afterwards, the fragment shader will be responsible for drawing the image on this canvas. The pipeline for the skybox shader is as shown in Figure 33.

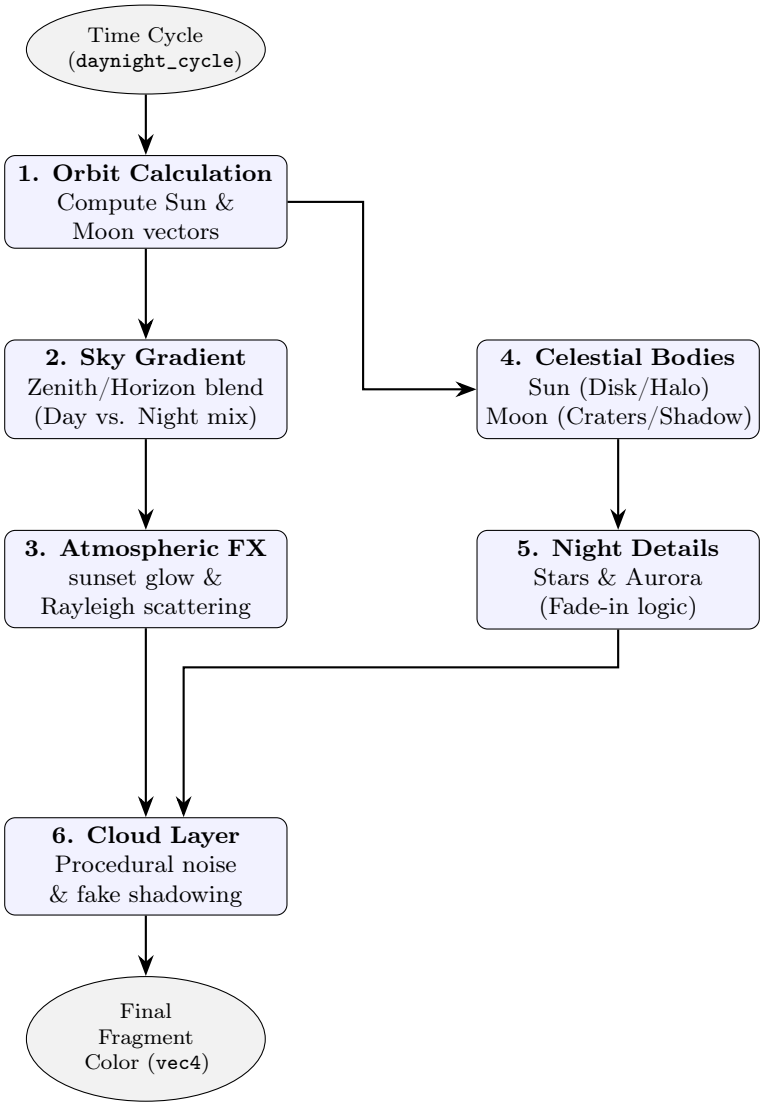


Figure 33: *Procedural Skybox Shader Pipeline. The shader calculates celestial positions before layering atmospheric gradients, light scattering, physical bodies, and clouds.*

Firstly, the `daynight_cycle` value is read from the settings struct (which will be covered in Section 3.8) determining the current time of day. The angle and sun's position is calculated as shown in Equation (76), where $\vec{d_L}$ is the sun direction, θ is the calculated angle from the `daynight_cycle` variable, ϕ is the sun offset in the z direction. The position of the moon is also placed directly opposite the sun, ensuring consistency in the placement of bodies and lighting conditions update automatically.

$$\vec{d_L} = \langle \sin(\theta), \cos(\theta), \phi \rangle \tag{76}$$

Afterwards, the atmospheric sky gradient has to be generated. This process will be procedural, as there is no pre-determined texture or image used, the algorithm will purely rely on the position of the sky and a few base colors to generate an appropriate gradient. The vertical gradient is done by blending the Zenith Sky color and the Horizon sky color based on the pixel height (`dir.y`). These zenith and horizon colors are

determined by blending in the preset values for the day and night version in relation to the position of the sky as shown in Equation (77).

$$\begin{aligned} I &= \text{smoothstep}(-0.3,\, 0.3,\, \cos(\theta)) \\ Z &= \text{mix}(\text{night}_{\text{zenith}},\, \text{day}_{\text{zenith}},\, I) \\ H &= \text{mix}(\text{night}_{\text{horizon}},\, \text{day}_{\text{horizon}},\, I) \end{aligned} \tag{77}$$

When the sky is near the horizon, a scatter tint and horizontal glow is applied onto the sky. This simulates Rayleigh scattering, when the sky turns orange-pink, as shown in Figure 34

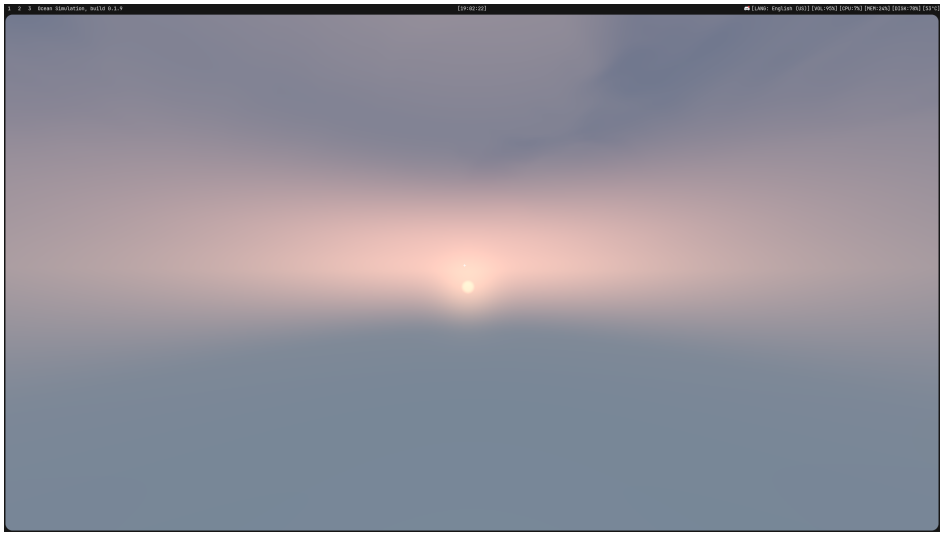


Figure 34: *sunset image of the final program, with* `daynight_cycle = 0.76`

3.5.3 Celestial Bodies and Volumetric Effects

Now, the image in Figure 34 also contains the sun, meaning this section will now cover celestial objects. Instead of using textures for the objects, they are drawn mathematically on the screen. The sun is a simple mathematical disk with a "halo", the glow around it, based on the distance between the pixel and the sun's direction (which was calculated earlier in Equation 76). The moon is a more complex hit-test, where if the pixel looks at the moon, the shader will calculate the UV coordinates on a sphere to apply procedural craters as seen in Figure 35, which was implemented using fractional Brownian motion.

Figure 35: *Image of the moon craters more up-close from the final program, with* `daynight_cycle = 0.95` *and clouds turned off*

Next, the shader is also capable of adding randomly generated stars, using polar coordinates and adding a blinking animation to them. Aurora Borealis can also be activated, adjusting its strength and brightness. The aurora is made by generating a ribbon of light mathematically by using the angle around the horizon and combines 3 sine waves of different frequencies and shifting them as time goes on, creating an oscillatory shape. Fractional Brownian Motion is also used to introduce noise, which creates the large wisps, along with a fine noise creating vertical streaks and causing that ionising feeling as seen in Figure 36

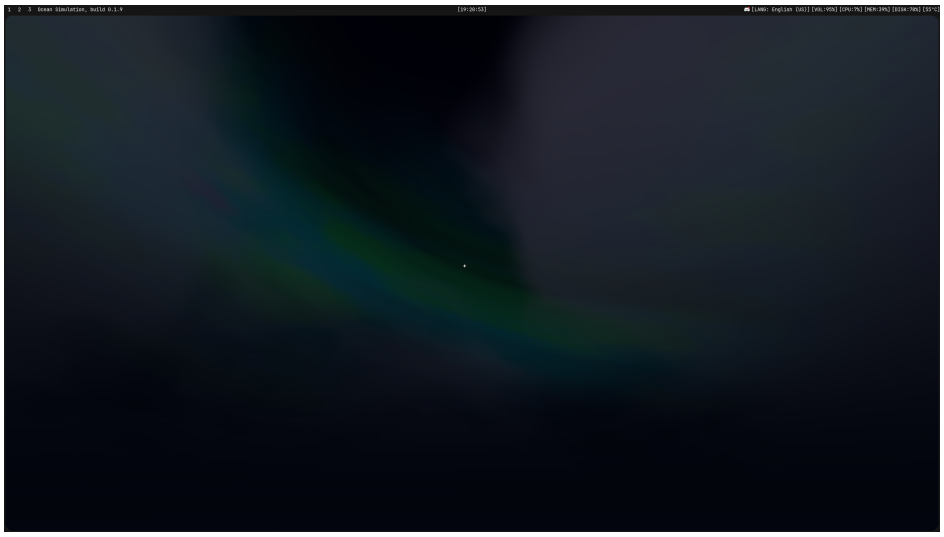


Figure 36: *Image of the aurora borealis*

Lastly, the clouds have to be generated. The common trick for this is Planar Mapping, where instead of wrapping around a sphere, the shader projects the sky onto a 2D plane high above the observer (`dix.xy/dir.y`), creating the illusion of flat, horizontally moving clouds. A noise function is used to determine the density of where the clouds exist, using the `smoothstep` function, which will create soft/hard edges. The volumetric shadows are faked into two stages to avoid expensive light calculation. First, by taking a sample of density at the current pixel, and second taking a sample of density at a small offset towards the sun. By taking the

difference between the samples, shader calculates which parts of the clouds are blocking the sun, creating dark undersides as shown in Figure 37.

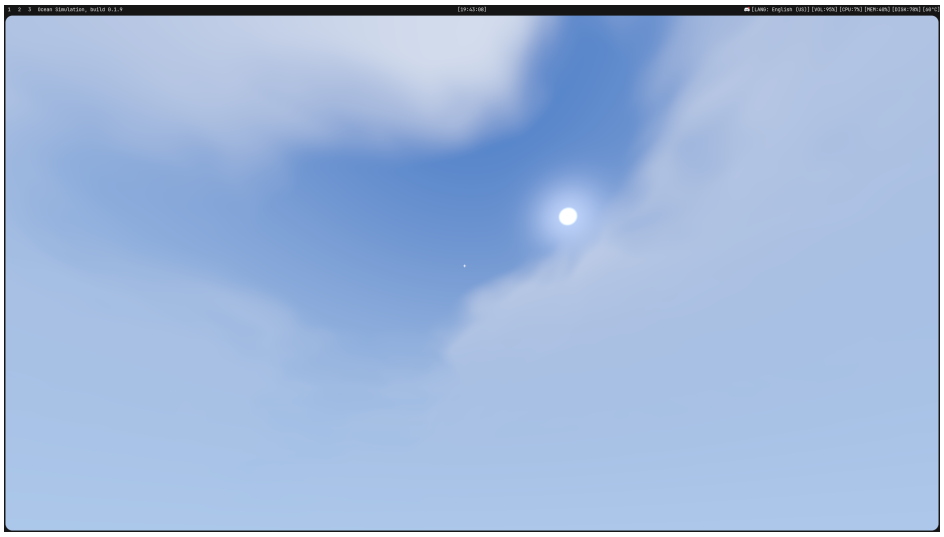


Figure 37: *Image of the clouds*

3.6 Blinn Phong implementation

3.6.1 Surface Normal Reconstruction

With the camera system and skybox in place, the main focus will now shift onto implementing the most basic lighting model - Blinn Phong. This model was discussed previously in Section 2.6.2, where the main formulas and derivations were established. This section in contrast, will focus on the practical implementation of those mentioned theorems, in the vertex and fragment shader. First, the previously mentioned method of central differences, covered in section 2.6.1, has to be set in place. As shown in Figure 38, the heightmap has to be sampled at four neighbouring points: $h_{i,j+1}$, $h_{i,j-1}$ (Vertical neighbours) and $h_{i+1,j}$, $h_{i-1,j}$ (Horizontal neighbours).

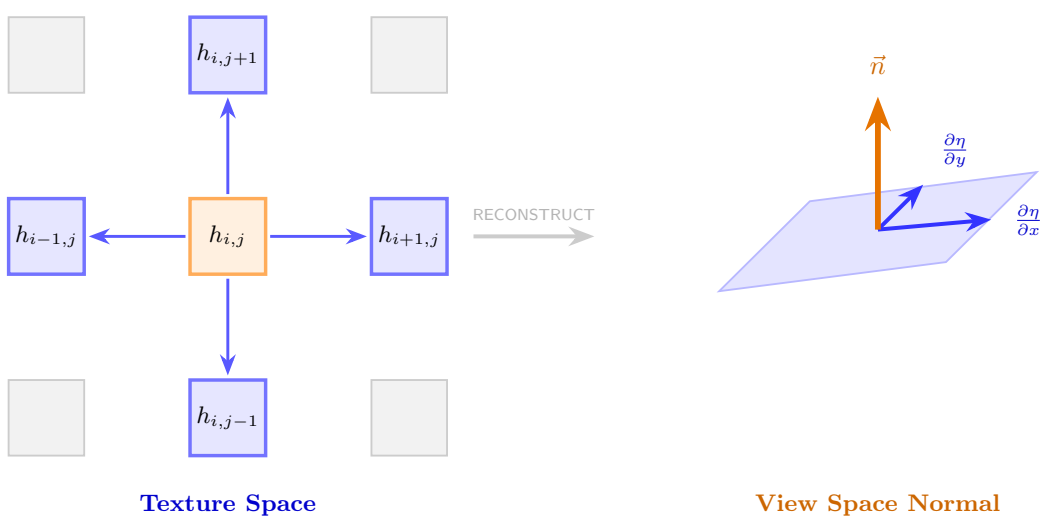


Figure 38: *Neighbour heights are sampled from the heightmap (shown on left) to compute the spatial gradient, which is then transformed into the surface normal $\vec{n}$ (right).*

In the `wgsl` implementation, a `delta_uv` is defined based on the world-space step size of the simulation. From the slopes acquired in Equation (59), two vectors can be constructed, which will be perfectly tangent to the surface of the water at the sampling point.

$$\vec{t_x} = \langle 1, \frac{\partial h}{\partial x}, 0 \rangle, \quad \vec{t_z} = \langle 0, \frac{\partial h}{\partial z}, 1 \rangle \tag{78}$$

However, as seen in the implementation code shown in Figure 39, the simulation has to account for horizontal displacement present (Choppiness, which will be discussed more in Section 3.7.2). This is a more complex tangent construction, where the x and z components of the vectors are also influenced by the derivatives of the displacement maps (D_x, D_z), constructing the Equation (79).

$$\vec{t_x} = \langle 1 + D_x, \frac{\partial h}{\partial x}, D_z \rangle, \quad \vec{t_z} = \langle D_x, \frac{\partial h}{\partial z}, 1 + D_z \rangle \tag{79}$$

```wgsl
let uv = in.tex_coords;

let amp = ocean_settings.amplitude_scale;
let chop = ocean_settings.chop_scale;
let subdivisions = f32(ocean_settings.fft_subdivisions);

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);

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

The surface normal will be defined as perpendicular to the surface. Mathematically, this is equivalent to taking the cross product between two tangent vectors, as a cross product will always result in an orthogonal vector. This is shown in Equation (80).

$$\vec{n} = \vec{t_z} \times \vec{t_x} = \det \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ 0 & \frac{\partial h}{\partial z} & 1 \\ 1 & \frac{\partial h}{\partial x} & 0 \end{vmatrix} \tag{80}$$

Where, if the determinant is expanded, the resulting normal vector would be as shown in Equation (81)

$$\begin{aligned} \vec{n} &= \hat{i}\left[0 - \frac{\partial h}{\partial x}\right] - \hat{j}\,[0-1] + \hat{k}\left[0 - \frac{\partial h}{\partial z}\right] \\ \vec{n} &= \langle -\frac{\partial h}{\partial x}, 1, -\frac{\partial h}{\partial z} \rangle \end{aligned} \tag{81}$$

3.6.2 Fragment Shader Calculations

To recall the previous sections, Blinn Phong consists of Ambient Light, Diffuse Light and Specular Highlights. In the simulation, ambient light is constant and applied to the whole surface of the object, therefore a simple constant, `ambient_intensity = 0.3` has to be added to represent I_a.

Afterwards, the diffuse light is to be calculated. To recall, the diffuse light calculation involves the dot product between the direction of light, which as previously discussed is the sun's position from the skybox shader mentioned in Section 3.5.2. This means that the sun direction has to be recalculated in the main fragment shader as well, which is not too efficient, however `wgsl` and `wgpu` do not yet support module importing and shared code. Therefore, the diffuse light code will be as shown in Figure 40.

```wgsl
// ... calculations for the surface normal, as shown in the previous section.
// Repeated code from skybox.wgsl
let angle = (ocean_settings.daynight_cycle - 0.5) * 6.28318;
let d_L = normalize(vec3(sin(angle), cos(angle), ocean_settings.sun_offset_z));

// Intensity using normal and d_L
let diffuse_intensity = max(dot(normal_geometry, d_L), 0.0);
// Combine the color of incoming light, intensity and scale by 0.4
let diffuse_component = ocean_settings.sun_color.rgb * diffuse_intensity * 0.4;
```

Figure 40: *An example code of the diffuse lighting present in Blinn Phong*

Next, the specular highlights are calculated using the half-vector trick. It is important to understand that the Phong technique checks if the reflected light points at the camera, while Blinn-Phong uses the half-vector to check if the surface normal is aligned to reflect light toward the camera. This is an important distinction in the implementation of lighting. The half vector is acquired by adding the sun direction and the view position of the camera, and then the dot product is taken and a power is applied, which will determine the sharpness of the highlights as shown in Figure 41.

```wgsl
struct CameraUniform {
    view_proj: mat4x4<f32>,
    view_proj_sky: mat4x4<f32>,
    camera_pos: vec3<f32>,
};

@group(0) @binding(0) var<uniform> camera: CameraUniform;

// ... other bind group definitions and functions
// ... calculations for the surface normal and d_L as shown in the previous section.

let view = normalize(camera.camera_pos - in.world_pos);
let half_vector = normalize(d_L + view);

// Higher power for sharp water highlights
let specular_intensity = pow(max(dot(normal_geometry, half_vector), 0.0), 256.0);
// Combine the color of incoming light with intensity
let specular_component = ocean_settings.sun_color.rgb * specular_intensity;
```

Figure 41: *An example code of the specular highlights present in Blinn Phong*

This concludes the Blinn-Phong section, where the resulting mesh with a standard sinusoidal pattern would look as shown in Figure 42.

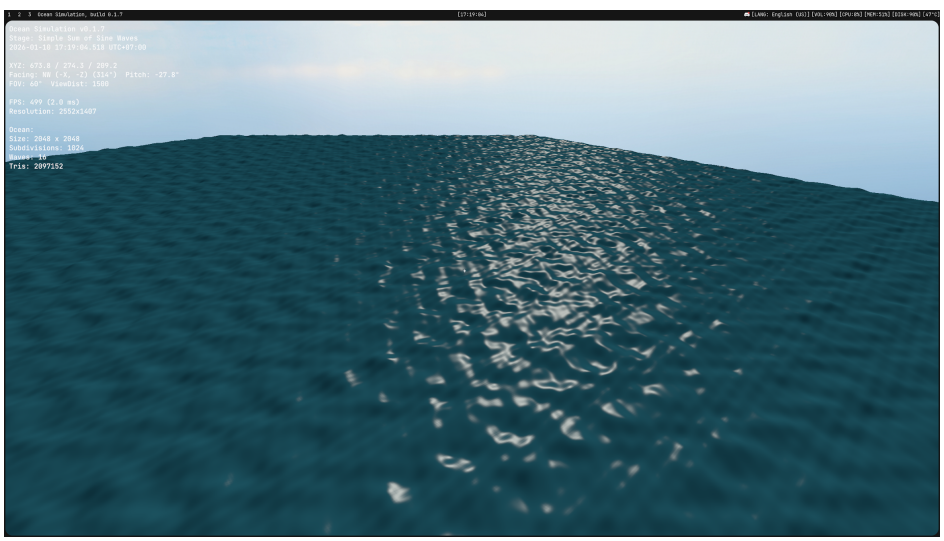


Figure 42: *Image of an older build (v0.1.7) of the ocean with a randomised ocean pattern and simple Blinn-Phong lighting*

3.7 Tessendorf's Model Implementation

Tessendorf's model is relatively straightforward for the result that it produces. The simulation following the Tessendorfs model can be broken down into multiple stages, initiation, propagation and rendering. This section will now build upon the knowledge discussed in Section 2.4, where the Phillips Spectrum will be applied to its full potential. The whole pipeline can be represented in a single diagram, as shown in Figure 43.

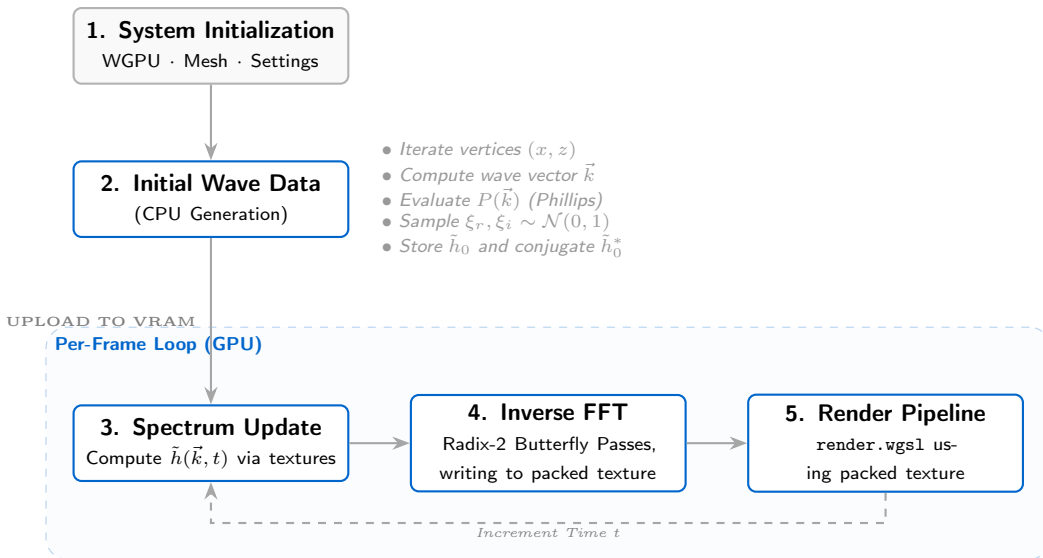


Figure 43: *The Tessendorf Ocean Simulation Pipeline, detailing the CPU initialization and GPU Fourier computations.*

3.7.1 Initial Data Generation

First, the initial data has to be generated once at startup, this data will then later be continuously updated in the compute shader. For this purpose a new struct, `InitialData`, will be created to house all the needed methods. The `State` struct 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.

Note: Vectors vs Arrays in Rust

Rust primarily has 2 main methods of storing data, to be indexed in a semi-organised way. This includes arrays and vectors (not to be confused with vectors in mathematics). Both objects contain slots for data to go into and can be indexed. The definitions for each can be seen in Figure 44.

```rs
let array: [u32; 10] = [0,1,2,3,4,5,6,7,8,9];
let item_array = array[0];

let vector: Vec<u32> = vec![0,1,2,3,4,5,6,7,8,9];
let item_vector = vector[0];
vector.pop();
vector.push(10);
```

Figure 44: *Vector vs Array definition*

However, arrays are only limited to a predefined size, while vectors can dynamically grow and shrink using `.push()` and `.pop()` respectivly.

The `InitialData` struct will now contain a `::new()` method, which upon being called will first construct the vector $\vec{k}$. The vector $\vec{k} = \langle k_x, k_y \rangle$ is directly related to the position in the grid, and can be acquired as shown in Figure 45. This is the discretisation of the continuous wave vector for a grid of `subdivisions` × `subdivisions` cells covering a physical tile size of `fft_size`

```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 45: *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. Afterwards, using this value of $\vec{k}$, the Phillips Spectrum Value is to be generated. For that matter, the `::get_phillips_spectrum_value()` is created which will encapsulate all the logic related to those calculations. As previously stated in Section 2.4, the full equation for the Phillips Spectrum is as shown in Equation (82).

$$P(\vec{k}) = A \cdot (\hat{k} \cdot \hat{W})^2 \cdot \exp\left(\frac{-1}{||\vec{k}||^2 L^2}\right) \cdot \exp\left(-||\vec{k}||^2 l^2\right) \cdot ||\vec{k}||^{-4} \tag{82}$$

The directional weighting is calculated using the dot product from the `cgmath` package, as shown in Figure 46.

```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 46: *Modified section of the* `::generate_phillips_spectrum_value()` *method inside of* `InitialData` *struct.*

After acquiring the spectrum values, two random values, ξ_r, ξ_i, have to be sampled from the Gaussian field. For this, a random field is first generated based on the `ocean_seed`, which is important to keep the ocean waves consistent upon rebuilding initial data as shown in Figure 47.

```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 47: *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 48.

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

Figure 48: *Box-Muller transformation to form a gaussian distribution*

It is important to note that the distribution here samples from a standard normal distribution. This is a common practical shortcut, which will still result in a random distribution. Next, frequency domain is constructed by first computing `let sqrt_phk = (phk / 2.0).sqrt()`, where `phk` is the Phillips Spectrum value and then multiplying it by the two random numbers to result in a real component and an imaginary component. The conjugate value for the initial spectrum will not be stored here, as it is more straightforward for the GPU to look-up the mirrored index, than waste space in the shader.

Note: Bit Shift

Bit shifting is a low level operation performed in computing, involving moving (literally shifting) the bits in a binary number a specified number of times in a particular direction. This operation is a highly efficient way of performing multiplication and division by a factor of 2, as shown in Figure 49.

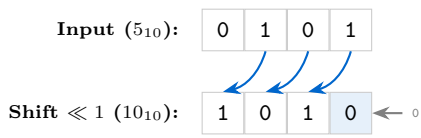


Figure 49: *A logical left bit shift ($\ll 1$). Each position move left doubles the value.*

Now, although the FFT itself is not completed yet, the butterfly passes will require twiddle factors. Recall that the twiddle factor is $\exp\left(\frac{-2\pi ik}{N}\right)$, which is a complex number sitting on the unit circle. By Euler's formula, every complex number can be represented in terms of sin and cos, since $e^{i\theta} = \cos\theta + i\sin\theta$. Now since the factors consist of sinusoidal functions, they 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 in a texture and pass it into the GPU. The twiddle factors are calculated as shown in Figure 50.

```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 50: *Twiddle code generation computed prematurely on the CPU*

The angle calculation can be rearranged as shown in Figure 83, where $N_{\text{local}} = 2s$ is the size of the sub-problem at that stage. This is precisely the twiddle angle formula $e^{-2\pi ik/N}$ but applied locally to each sub-FFT size rather than the global N.

$$\theta = \frac{2\pi \cdot \text{offset}}{2s} = \frac{2\pi \cdot \text{offset}}{N_{\text{local}}} \tag{83}$$

In the end, this will leave the program with `InitialData` struct, containing `k_vec`, `h_0` and `w`. As well as the twiddle factors vector.

3.7.2 Time Evolution

Now, after the data has been successfully generated, it has to be packaged to be sent to the GPU's compute shader. This is done using bind groups, storage textures and compute shaders. First of all, a storage buffer is created using `create_buffer_init`, which means the GPU will be able to index into it as an array, since it's read-only. This buffer will not change between frames, only the time evolution will change.

Now, this data will be passed onto the Compute shader, inside the GPU. Compute shaders work slightly differently to normal fragment, or vertex shaders. First of all, they do not produce an output to the screen, they work outside of the render group. They are launched in a grid of workgroups, each containing threads, working in parallel, which allows for fast and efficient calculation, such as the IFFT needed to convert the frequency spectrum to the time domain. The entry point of a compute shader is defined by `@compute @workgroup_size(16, 16)`, and it takes in a global parameter `@builtin(global_invocation_id) id: vec3<u32>`. The workgroup size statement means that the workgroup has 256 threads arranged in a 16×16 block. This means that when the CPU dispatches the call using `pass.dispatch_workgroups(N / 16, N / 16, 1)`, there will be enough workspaces to cover the entire $N \times N$ texture, allowing every pixel to get its own thread. The FFT is notoriously parallel at each butterfly stage, as each output element can be computed independently from a fixed pair of inputs. The GPU can do this for all N^2 elements simultaneously, while the CPU would struggle doing them serially. This is best demonstrated in Figure 51.

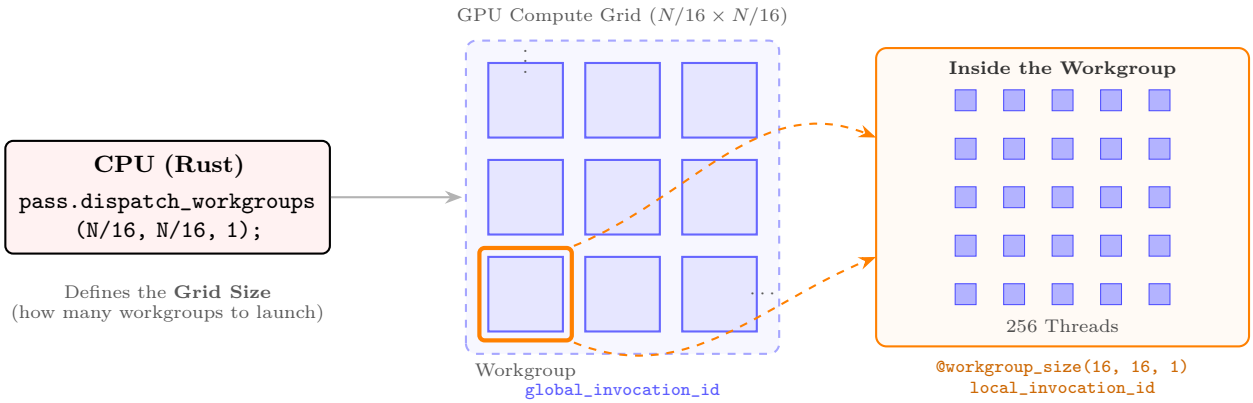


Figure 51: *The Compute Dispatch hierarchy. The CPU defines the total number of workgroups to be processed (the Grid), while the Shader code defines the internal density of threads per workgroup. This 2D mapping allows the GPU to process an $N \times N$ texture by assigning exactly one thread to every pixel.*

Before the butterfly passes can be done however, the spectrum has to be evolved in time. This is run once per frame, where the code takes a static `h_0` passed in at initialisation and evolves, following the Equation (47) discussed in previous sections. The shader code will be as shown in Figure 52.

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

Now, it is really interesting how to handle complex numbers inside of computing. First of all, complex numbers are used in mathematics to solve two dimensional problems, where $\sqrt{-1} = i$. However, in Rust, there are no imaginary numbers, therefore every complex number has to be treated as a vector, with the x component containing the real value and the y component containing the imaginary value. Through experience and deep mathematical understanding, a connection can be made that complex numbers are highly related to standard mathematical vectors, with a trick up their sleeve. When a complex number is multiplied by i, at a deeper level, the whole vector is being rotated $\pi/2$ in the counter-clockwise direction as shown in Figure 53. This is fundamental for the `complex_multiplication` function discussed in the compute shader, as it functions by rotating the coordinates in the two dimensional plane to match the transformation done by the rotation. The full function can be derived, as shown in Equation (84).

$$\begin{aligned} f(z_1, z_2) &= z_1 \cdot z_2 = (a_1 + ib_1)(a_2 + ib_2) \\ &= a_1 \cdot a_2 + a_1 \cdot ib_2 + ib_1 \cdot a_2 + ib_1 \cdot ib_2 \\ &= a_1\, a_2 + ia_1\, b_2 + ia_2\, b_1 - b_1\, b_2 \\ &= (a_1\, a_2 - b_1\, b_2) + i(a_1\, b_2 + a_2\, b_1) \end{aligned} \tag{84}$$

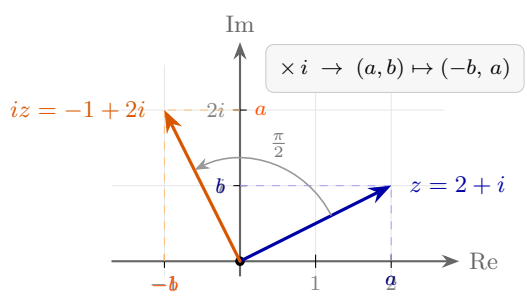


Figure 53: *Here $z = 2 + i$ maps to $iz = -1 + 2i$, demonstrating the general transformation $(a, b) \mapsto (-b,\, a)$ used in* `complex_multiplication`.

After calculating `h_tilda`, the horizontal displacements dx and dz have to be calculated. The calculation involved multiplication by i, but as previously mentioned, it can be modeled as a rotation of the vector by $\pi/2$. Figure 54 contains the code responsible for the displacement calculation.

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

Note: Textures

Throughout this discussion, textures have been mentioned, but a proper definition and introduction is required. A texture is fundamentally a two dimensional array of data living on the GPU. Each element inside the texture is referred to as a texel. These textures can store colors, frequency domains, or any other piece of data that satisfies the `rgba16float` format. What makes a texture so different and special from a buffer is that most modern GPUs have dedicated hardware for accessing the textures, specifically the sampler. This sampler is able to access mipmaps, clamp coordinates and much more. These textures can be defined in `wgsl` as `var src: texture_2d<f32>`. And they can be read by utilising `textureLoad(src, vec2<i32>(x, y), 0)`, where `0` is the mip level. You can read from these in a compute shader, but not write.
Storage textures however, although working very similarly, allow for writing. They can be defined using `var dst: texture_storage_2d<rgba16float, write>`. Hence, many methods like the ping-pong model, which will be covered shortly, require both types of textures.
Hence, the simulation contains a `TextureInstance` struct which is responsible for the construction of these textures and storage textures with appropriate flags and settings.
To address any texel in the texture, the UV coordinates (`vec2<i32>`) have to be known. The UV coordinates are normalised floats, which only exist in the range $[-1.0, 1.0]$ regardless of texture size and resolution, where (0.0, 0.0) is one of the corners and (1.0, 1.0) is the opposite, as shown in Figure 55.

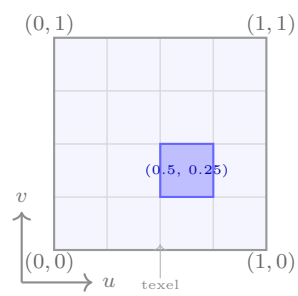


Figure 55: *UV coordinate space mapped over a* 4×4 *texture. Each cell is a texel, coordinates are always normalised to* $[0, 1]$.

Hence, the UV coordinates are often scaled based on the resolution of the texture to address the specific pixel. Then, these UV coordinates are used along with a sampler: `textureSampleLevel(texture, sampler, scaled_uv, 0.0)` to extract data from that texel.

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 with the second one having an empty value which could be later filled in with additional data. Figure 56 shows this in action.

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

3.7.3 IFFT Implementation

The Cooley-Tukey IFFT theory has been covered in Section 2.5.1, 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 (which was accounted for in the `InitialData`) and the normalisation $1/N$. In practice, this means the four steps discussed earlier can be reduced as following.

1. Conjugation of input is now handled implicitly, since the Hermitian symmetry guarantee establishes during the spectrum evolution.
2. Forward butterfly passes are still present for all $\log_2(N) \times 2$ passes, with positive twiddle values.
3. Another conjugation of output which is also skipped, since the imaginary parts are discarded in the final output due to Hermitian Symmetry. This also allows you to pack the real values onto a single texture.
4. Scaling by $1/N$ is applied on the final vertical pass.

However, there is a fundamental constraint of GPU compute shaders. A texture cannot be read from and written to within the same call, as this would bring up race conditions and lots of bugs, where data can be overwritten. 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 57 represents this process.

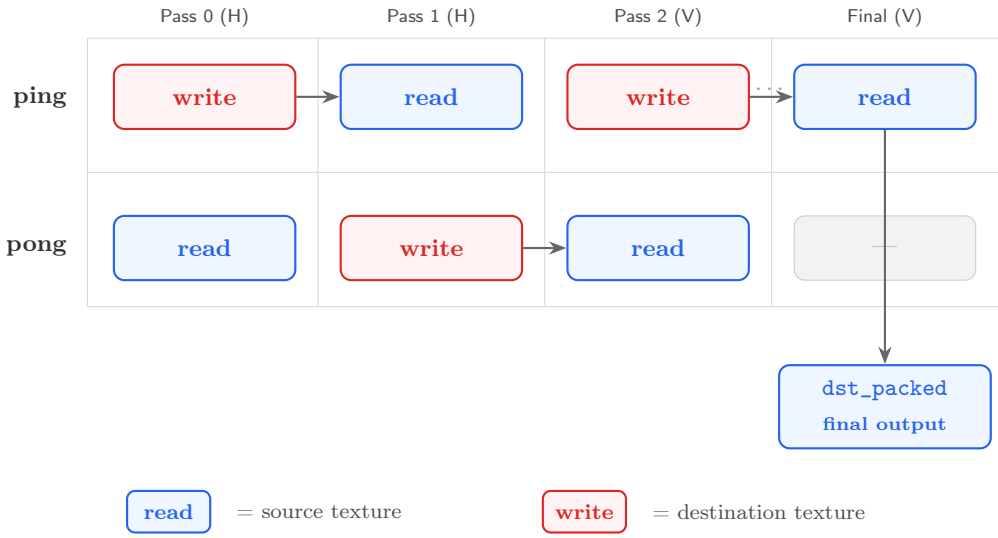


Figure 57: *Ping-pong texture swapping across butterfly passes.*

On the CPU side, a boolean flag tracks which texture is currently active, and is toggled after each dispatch loop pass. 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 58.

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

Every pass dispatches the same compute shader, which handles both horizontal and vertical directions using a single flag, `is_vertical`, swapping the required axis as shown in Figure 59.

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

Figure 59: *A single shader handles both horizontal and vertical passes via the* `is_vertical` *flag.*

Each thread then calculates its own position inside the butterfly structure for the current stage using left logical bit shift, similar to the one present in Twiddle Factors generation mentioned previously. From the size $s = 2^{\text{stage}}$, the thread is able to determine which pair of elements it is responsible for combining, as shown in Figure 60.

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

Figure 61 illustrates the process in which the two outputs can be generated depending on the `pair` and how

the twiddle factor comes into play.

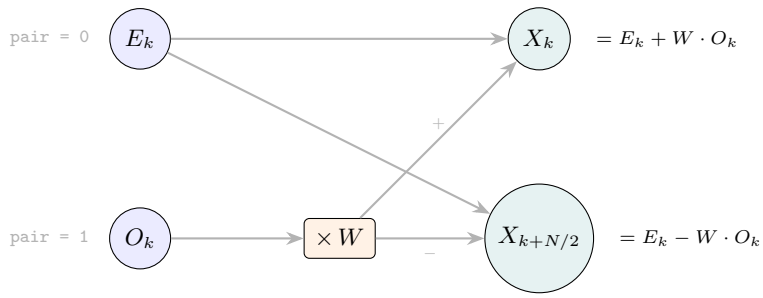


Figure 61: *A single radix-2 DIT butterfly, where the twiddle factor W rotates the lower element in the complex plane before the addition and subtraction that produces the two outputs that can be seen*

This diagram can be compared with the `wgsl` code as shown in Figure 62.

```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 62: *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`*.*

Now, recall the twiddle factors that have been pre-generated on the CPU and uploaded as a storage buffer. Each stage s contributes 2^s entries in the buffer, so the starting index for s in the array would correspond to $2^s - 1$. The twiddle factor look-up code can be seen in Figure 63.

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

On the last vertical pass, the shader will detect its ending completion by checking the stage and direction of the incoming pass. Then, it would apply the $1/N$ normalisation and discard the imaginary components. This allows the shader to write the real components directly into a single `packed_texture`, saving extra VRAM, as shown in Figure 64.

```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 64: *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 65.

R	G	B	A
$\eta(x, z, t)$	$D_x(x, z, t)$	$D_z(x, z, t)$	1.0
height	x-displacement	z-displacement	unused

`rgba16float` — one texel per grid vertex

Figure 65: *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.*

Congratulations! The IFFT is complete, however although the data is now correctly calculated, no visible changes will appear unless the data is then passed into the vertex shader. On the CPU side, this `packed_texture` is now passed to the main render pipeline, where the vertex shader can apply it to the vertices themselves. Figure 66 shows the vertex shader responsible for that.

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

3.8 Settings and Presets

3.8.1 Preset, Builder and Uniform

Note: GPU Memory Alignment

In `wgsl`, data stored in storage buffers must follow specific alignment rules (`std430`). The GPU expects data to be aligned to 16-byte boundaries, meaning if a field is taking up 4 bytes, explicit padding (`_pad0, _pad1, _pad2`) is added to the shader struct. This padding ensures that each struct instance has a size that is a multiple of 16 bytes, matching the expected memory layout on the GPU and maintaining compatibility with the corresponding Rust-side struct.

This simulation is really complex and requires fine tuning of every variable and every possible scale to achieve a good result. To manage that, a system for tracking and updating ocean settings have to be implemented. For that purpose, the `OceanSettingsUniform` is created, which will act as the centralised struct containing all the fields that could possibly be needed in any shader code. This struct is used throughout all the shaders and Rust code, which allows the simulation to be consistent throughout. While the struct is relatively large for a uniform buffer, approaching the 130 field counts and mounting to approximately 1-2kB, whose impact on GPU bandwidth is negligible compared to the benefit of having synchronised state between CPU and GPU. It is unusual for a buffer to have such a high number of fields, this is a known issue in the simulation, since most commonly the settings are spread out across multiple buffers or constants.

The `OceanSettingsUniform` struct was designed to not be hard-coded, meaning values are set in the code and can't be changed without editing it, but rather be dynamic and be able to adjust to various conditions. Therefore, a `OceanSettingsPreset` struct was implemented. This struct contains the needed methods, to load from and to `.json` files, converting the preset stored there to an accessible format for `wgpu`.

This process is straightforward as it involves using the standard libraries for reading files and folders, however an issue arises at the conversion step. A builder has to be implemented to handle the difference between the two structs, it will be acting as an intermediate, into which a preset can be loaded and a uniform struct will be outputted as shown in Figure 67.

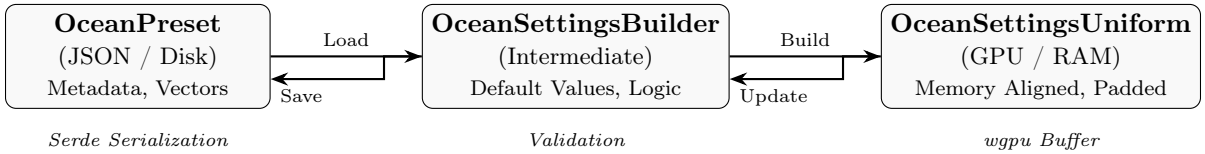


Figure 67: *The transformation pipeline for ocean configuration: moving from persistent storage to the GPU-ready uniform via the Builder pattern.*

3.8.2 User Interface

Note: Macros in Rust

Macros like `println!` are used all the time in Rust, but the understanding of what they do and how they work is often left out. A macro in rust, is really similar to a function, where a certain block of code will be executed. However, a macro doesn't call the code, a macro is a way of writing code that writes code, often known as meta-programming. Basically, the macros expand and insert the macro definition code into the source code itself. The benefit of macros is that they are expanded at compile time, which allows for certain features, like implementing a trait on a type, which wouldn't work with functions, since they are primarily executed during the run-time.

This architecture, where the user can create, share and edit presets allows for creativity and encourages collaboration. Each preset drives the simulation forward to its completion. However, changing a `json` file and re-starting the program may indeed be tiring. A solution to this problem includes creating a user interface,

which can be toggled by a press of a button (F4), displaying a menu with all the variables laid out in neat groups, being able to be changed on the fly. This UI is achieved using the `egui` library, it provides a very intuitive and robust approach towards initialising and generating the base layer for the user interface.

First and foremost, the `State` struct is adjusted to include the `egui_state` and `egui_renderer`, where afterwards in the `::new()` method, the context and renderer are initialised as shown in Figure 68.

```rs
// Create default context, which will be used to define the layout of the interface
let egui_context = egui::Context::default();
// State that handles the interaction between egui and winit
let egui_state = egui_winit::State::new(
    egui_context,
    egui::ViewportId::ROOT,
    &window,
    None,
    None,
    None,
);
// Renderer for the egui based UI
let egui_renderer = egui_wgpu::Renderer::new(
    &device,
    surface_config.format,
    egui_wgpu::RendererOptions {
        depth_stencil_format: Some(DEPTH_FORMAT),
        msaa_samples: 1,
        dithering: true,
        predictable_texture_filtering: true,
    },
);
```

Figure 68: *Initialisation logic for* `egui`

Afterwards, the code is slightly adjusted to listen for user inputs and when `winit::keyboard::KeyCode::F4` is pressed, toggle the flag to display the UI. This flag triggers the `::render_settings_ui()` method call, whose main responsibility is actively displaying the user interface. This method works by first defining custom macros, to simplify the repeating logic in creating the `vec3` and `vec4` slides, the text boxes and any toggles. The method then continues to include all the variables from the preset with the appropriate macro. Each macro contains the following parameters: the UI context, the label, value to change and the range for the input verification, as shown in Figure 69.

```rs
s!(ui, "FOV", fovy, 10.0..=160.0);
```

Figure 69: *A macro which will expand to generate a slider for the FOV, which will trigger the change in* `fovy` *variable, with the allowed range from* 10° *to* 160° *inclusive.*

However, since the UI is re-rendered each frame, the updated value that the user has changed would be immediately gone. Therefore the `State` struct will contain a `draft_settings` field, meaning all the changes in the UI will correlate directly to this `draft_settings`, allowing to persist data across re-renders. After all the sections, with all the sliders and fields laid out, the UI will contain an "Apply Preset" button, allowing the user to apply the current draft settings to the main simulation, as well as extra options to update the current preset with new values or create a new preset.

When a preset gets applied, the current ocean settings have to be swapped out with the draft settings, which will change what data is coming into the shaders. However, if the data changed has any relation to the waves themselves, like amplitude, FFT size, then nothing would happen. This is because the data has already been

generated and updating the settings, won't change that. Therefore, after the "Apply Settings" button is pressed, all the data regarding textures, FFTs, Initial Data will have to be re-generated, while using the new values. During the initial data generation, the gaussian distribution was generated using an `ocean_seed`. This is to ensure that after this re-generation step, the wave pattern will not be different from what is already present, allowing for consistent re-generation each time.

3.9 PBR Implementation

The vertex displacement calculations are over, meaning the hardest part of simulating realistic waves is done. However, the output produced will not be as visually appealing due to the simple Blinn-Phong model implemented earlier. As discussed earlier in Section 2.6.3, this model is based on the statistical approach that the surface of any object is made up of many microfacets, which create much more realistic effects seen in real life lighting.

3.9.1 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 70.

```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 70: *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 71, where first the base value of `roughness` is acquired from the `ocean_settings`, and is scaled by multiple parameters, such as the distance and foam coverage.

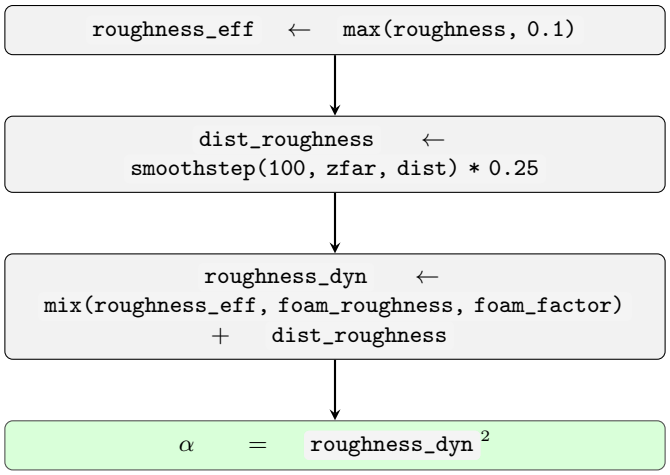


Figure 71: *Dynamic calculation of* `alpha`.

As previously shown in Equation 64, these functions combined make up the Cook-Torrence BRDF, and can be represented in code as shown in Figure 72.

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

This function is then to be used inside of the fragment shader, to acquire the specular intensity value. This value is then used in combination with the light colour and other various scale factors, which limits the amount of reflection passing through. These scalar values include `specular_scale` from ocean settings, `reflection_dampener` and `1.0 - foam_factor` which will be discussion in Section 3.10.2.The `reflection_dampener` terms are required to limit the amount of reflections occurring when a wave is near the point of collapse or creases. This term is calculated by interpolating the Jacobian between the minimum and maximum reflection value set in the settings, where the Jacobian is a measure of creases in a wave, meaning at low values the wave is likely to fold over itself. The Jacobian will be discussed more in Section 3.10.1.

3.9.2 Sky Reflections and Tone Mapping

Next, it is important to discuss the extended use of the Fresnel term. This term quantifies the physical value of reflectance, and is used in two distinct cases in the fragment shader. One of them is as previously discussed passed into the BRDF model which governs specular highlights, while the second value governs how much of the procedural sky will get blended over the water colour. Both terms share the same base reflectivity f_0, they take different dot product inputs. The specular highlight function call uses the `view_dot_half`, the angle between the camera view vector and the half-way vector of geometry discussed earlier. The sky reflection component however, uses `n_dot_view`, which is the angle between the normal vector and camera view, which measures the grazing angle of the microscopic surface. This is what controls the mirror-like reflections seen in the water, since at near normal vector angles the water seems transparent, while at higher grazing angles the water becomes highly reflective despite the structure of microfacets. This sky reflection component is then clamped and adjusted by scaling factors based on the distance.

Afterwards, the ambient light is applied which is sampled from the sky's current sky colour, which dynamically responds to the current time of day. All of the factors such as ambient light, specular highlights, sky reflections will cause the brightness to often exceed 1.0. Clamping the final brightness would cause an unrealistic and flat image, losing all the detailed highlights. Therefore, tone mapping is used to apply a curve, compressing high values into a valid range while preserving detail. Therefore, Academy Colour Encoding System (ACES) [1] filmic tone mapping curve will be used as seen in Figure 73. Afterwards, a gamma correction is applied by raising color 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 73: *The* `wgsl` *implementation of the Narkowicz 2015 ACES approximation.*

The complete flow of the new PBR system is summarised in Figure 74.

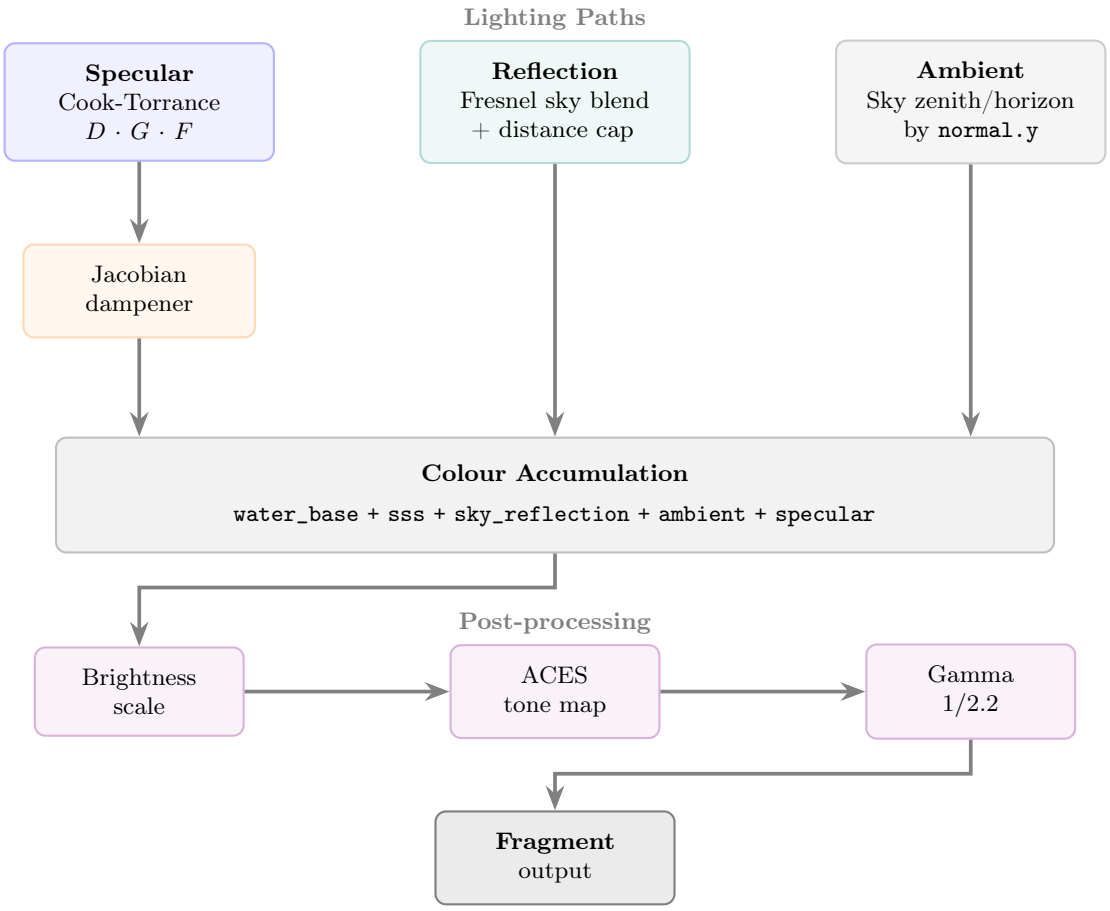


Figure 74: *Three light paths add up onto the water base colour and then get post-processed.*

3.10 Jacobian Foam

In a real ocean, the waves constantly battle against one another. They often clash and crease in various ways. The Tessendorf's implementation does its best to account for this, creating various wave patterns. However, the visual standpoint could be improved. The Ocean usually generates foam and spray particles when sharp peaks occur, and this can be modelled into our simulation by first identifying these positions where the foam would occur, then generating the foam itself and advecting it through time. The spray particles are beyond the scope of this project, as they require a completely new model of particle generation, which would extend the project's development time even further.

3.10.1 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. In this simulation use case, this is perfect, since there are multiple variables such as η, dx and dz to account for when calculating the derivative. The Jacobian matrix definition is as shown in Equation (85).

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{85}$$

By taking the determinant of the Jacobian matrix, which measures how much a small patch of surface is being stretched or compressed, we can identify the disturbed areas of the surface and generate the foam on those positions. Recall that earlier the position of a vertex was defined as shown in Equation (86).

$$v = \langle\, u + D_x(u,v),\ \eta(u,v),\ v + D_z(u,v) \,\rangle \tag{86}$$

This means that the Jacobian can be applied onto the position of the vertex and its determinant can be calculated as shown in Equation (87).

$$\begin{aligned} \mathbf{J} &= \begin{bmatrix} \frac{\partial}{\partial u}\left[u + D_x(u,v)\right] & \frac{\partial}{\partial v}\left[u + D_x(u,v)\right] \\ \frac{\partial}{\partial u}\left[v + D_z(u,v)\right] & \frac{\partial}{\partial v}\left[v + D_z(u,v)\right] \end{bmatrix} \\ &= \begin{bmatrix} 1 + \frac{\partial D_x(u,v)}{\partial u} & \frac{\partial D_x(u,v)}{\partial v} \\ \frac{\partial D_z(u,v)}{\partial u} & 1 + \frac{\partial D_z(u,v)}{\partial v} \end{bmatrix} \\ J = \det \mathbf{J} &= \left(1 + \frac{\partial D_x}{\partial u}\right)\left(1 + \frac{\partial D_z}{\partial v}\right) - \frac{\partial D_x}{\partial v}\,\frac{\partial D_z}{\partial u} \end{aligned} \tag{87}$$

Where when $J = 1$ the surface is undisturbed, when $J > 1$ the surface is being stretched creating flat troughs, while when $J \to 0$ the vertices are converging to a single point. However, when $J < 1$ the ocean surface has been flipped inside out, which should not happen with the current implementation. Therefore a range for the Jacobian values has to be taken into account when generating the foam. J is calculated similarly to the surface normal, using the central differences method as shown in Figure 75.

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

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. The result is shown in Figure 76 where the ocean surface has the full spectrum of colours.

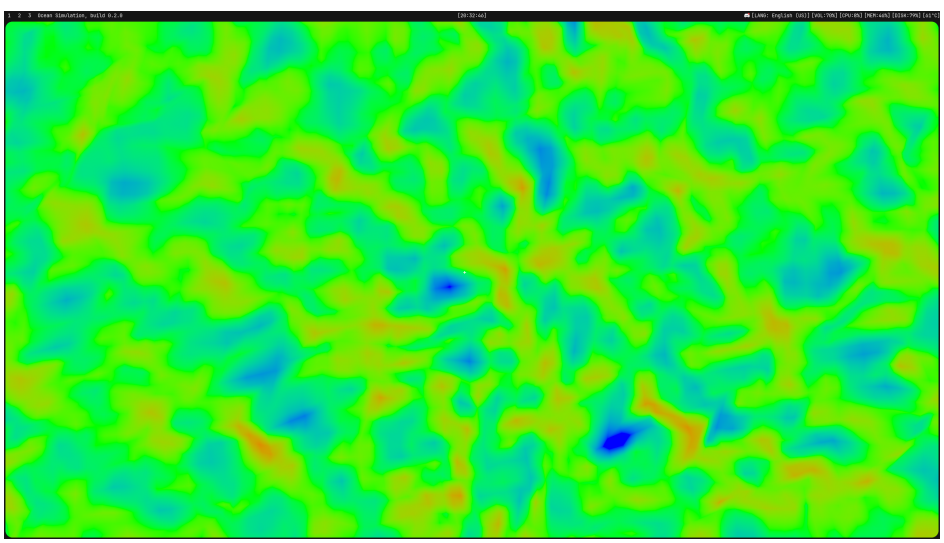


Figure 76: *Jacobian mapped to the colour spectrum*

3.10.2 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 75 can be translated to the compute shader code where the `uv` and `uv_offset` will have to be calculated using the provided `global_id`. Afterwards, the compute shader will calculate the `breaking` and `generated_value`, as well as decay the previous foam as shown in Figure 77.

```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 77: *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 (88), where the `foam_speed` is taken from ocean settings and $\frac{N}{\text{size}}$ is a scaling factor to adjust to the world size.

$$s_{\text{px}} = v \cdot \text{foam_speed} \cdot \Delta t \cdot \frac{N}{\text{size}} \tag{88}$$

Afterwards, using the displacement value the previous position of the foam is calculated. The previous position 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. The position is then tiled properly by using the dimensions of the texture and then bilinear interpolation is used to acquire a smooth and accurate position of the foam without making the appearance blocky. 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 78

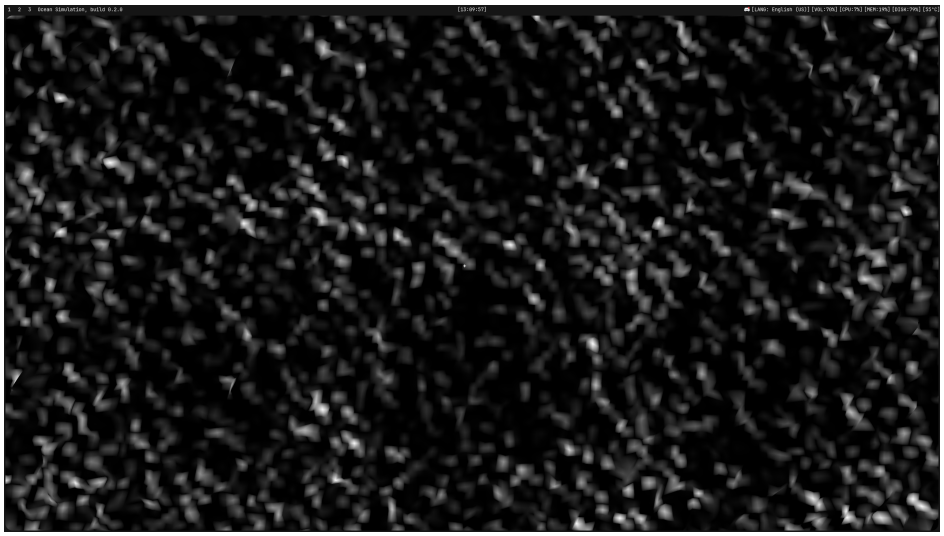


Figure 78: *Fragment shader output when* `vec3<f32>(foam_strength)` *is the output, acknowledging the fact the output is not perfect and could be improved, however this satisfies the simulation requirements.*

3.10.3 Foam Rendering

The foam is now present inside of the texture and can be used inside the fragment shader. First, the texture is sampled at the appropriate `uv` coordinates and data is extracted from the red channel. As previously seen in Figure 78 the output is still blocky and missing sharper detail. Therefore the fragment shader is also responsible for amplifying the detail on the texture. This is done by first 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 79.

```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 79: *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 80, by mapping `foam_shaped` onto the output colour.

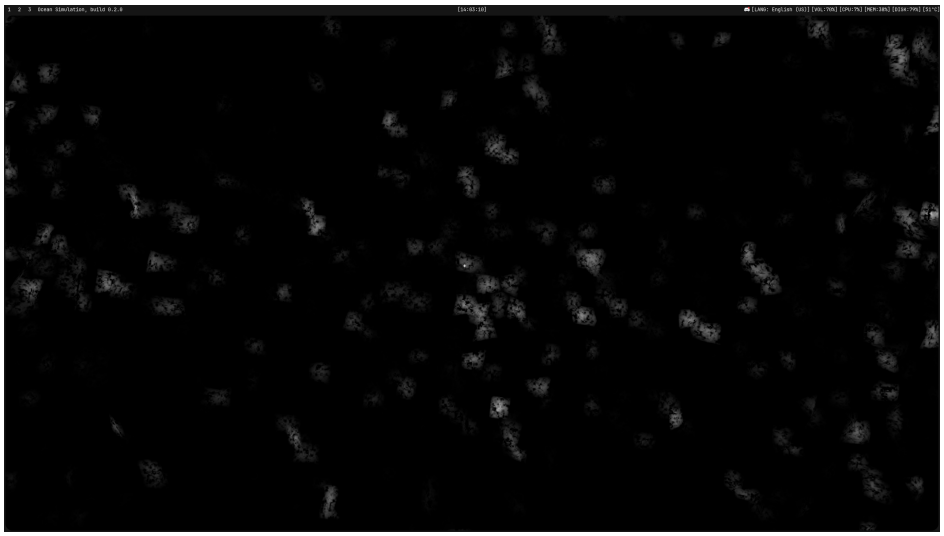


Figure 80: *Image of* `foam_shaped` *mapped to the output colour, where bright spots represent foam present in that area.*

Before this more detailed foam can be applied, so called wind streaks can be generated. In real life, wind usually pulls foam in long, thin stripes, often referred to as Langmuir circulation. The calculation works by applying anisotropic transformations to the noise coordinates, scaling the frequency differently along the wind vector and its perpendicular tangent as shown in Figure 81

```wgsl
// Anisotropic Streak Generation
let streak_along = dot(in.world_pos.xz, wind_norm) * 0.15 + camera.time * wind_mag *
    0.0002;
let streak_across = dot(in.world_pos.xz, vec2(-wind_norm.y, wind_norm.x)) * 2.5;

let streak_val = pow(noise(vec2(streak_along, streak_across)), 6.0);
let total_foam = clamp(foam_shaped + streak_val * foam_shaped * 0.6, 0.0, 0.9);
```

Figure 81: *Wind streak calculations*

However, the foam is not ready yet to be blended into the ocean colour. Foam is a physical layer of bubbles, meaning it will block out reflections and mirror-like specular highlights commonly present. Therefore, the `roughness_roughness` is calculated as previously shown in Figure 71. With that accounted for, the rendering of the foam is to be computed, using the half-Lambert model. The foam is given an off-white albedo color, instead of pure white to be closer to the physical reality. 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 82.

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

It is important to note that the direct light term would make foam appear too dark when the sun is near the horizon. Therefore, `sky_fill` term is calculated using the current `horizon` and `zenith` sky colour,

which will respond dynamically to the time of day. 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. However, it is scaled back to 0.08 and clamped using `intensity`, making it disappear at night. 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 83.

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

3.11 Subsurface Scattering

Although the rendering pipeline is nearly done, there is one last thing that has to be implemented. 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. Without it, the water would look opaque, dark and solid. Figure 84 shows how the light undergoes SSS.

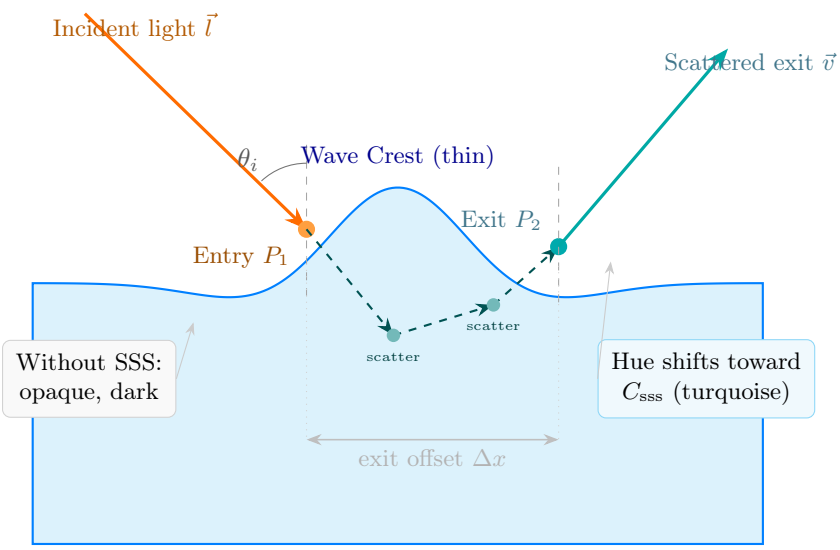


Figure 84: *Subsurface scattering on a wave crest. The thinner the crest, the shorter the internal path and the more visible the turquoise tint*

The ocean water is not perfectly clear, it contains salt, microbes, air bubbles and much more, and whenever a wave crests, the volume of water becomes thin enough that light can pass through it, bounce around and exit. For that, 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 [3]. 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

```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 87: *The final SSS calculations*

towards the viewer. Lastly, the power function creates a bloom effect, which changes with the angle of viewer and wave crest. Figure 85 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 85: *Code responsible for generating the light effect generated based off the* `sss_crest_mask`

This section of the code uses `sss_crest_mask` to identify where the water is thin, which is at the top of a wave. 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 86.

```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 86: *The SSS crest mask calculation involving height of wave and the jacobian*

Afterwards, all the variables can be combined to form a strength factor, which decreases with camera distance. 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 87.

3.12 Cascading multiple FFT's

Although the Tessendorfs model is set in place and the waves are generated based on a statistical model, the ocean may still look repetitive and bland. This is a common issue, which involves a genius fix.

Instead of increasing the complexity of the spectra itself to generate a more precise and random ocean, a more efficient approach can be taken. By overlaying multiple FFT cascades running in real-time in parallel to each other, a realistic model of the ocean can be created, where big swells and small ripples can exist co-united. Each cascade will have its own FFT Size S_c and Amplitude A_c, by adjusting these parameters, any desired outlook of the ocean can be achieved as shown in Figure 88.

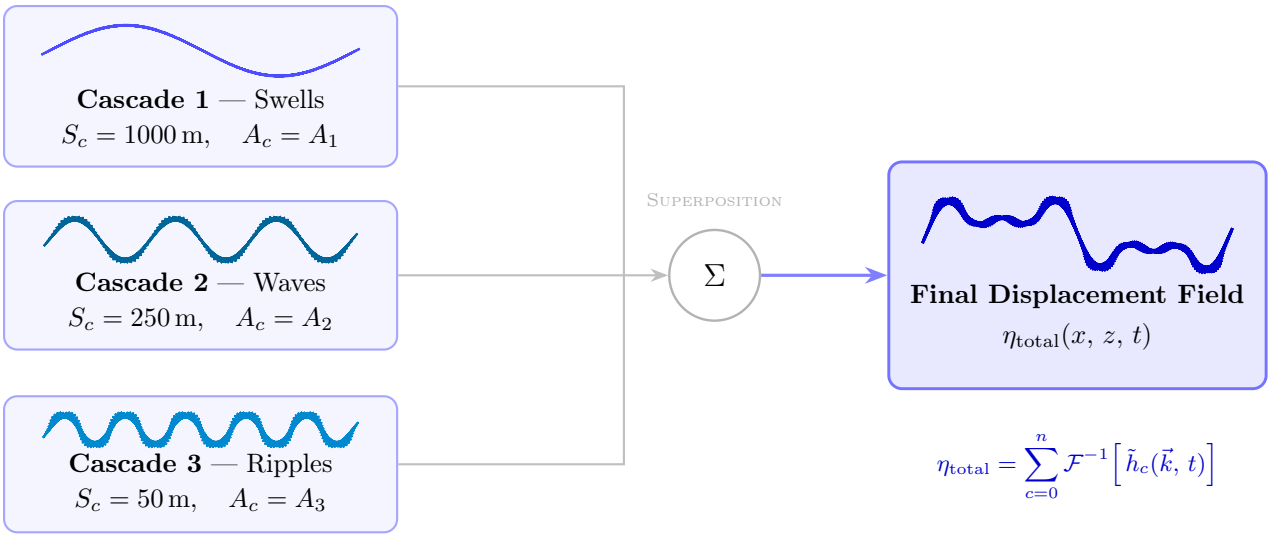


Figure 88: *Multiple independent IFFT simulations run in parallel, each with its own tile size S_c and amplitude A_c. Their displacement fields are summed via superposition to produce a single combined displacement field.*

To allow the user to freely adjust the cascades, the `OceanPreset` struct has to be adjusted. Define a new field, `cascade_data`, which will hold a vector of `CascadePreset` struct, where each struct holds the S_c and A_c. This vector is read on initialisation and converted to a 2D array, with max size `MAX_CASCADES` (by default 6), where at index 0 the S_c is stored and index 1 stores the A_c, which is stored in the `OceanSettingsUniform` as shown in Figure 89.

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

Figure 89: *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 hold a vector of `CascadeResources`. This `CascadeResources` struct will now be responsible for holding the necessary textures and bind groups as shown in Figure 90.

```rs
pub struct CascadeResources {
    // Size of the waves
    pub size: f32,
    // Textures needed for the ping-pong technique for height map, displacement in x and
    displacement in z
    pub texture_ping_h_dx: TextureInstance,
    pub texture_pong_h_dx: TextureInstance,
    pub texture_ping_dz: TextureInstance,
    pub texture_pong_dz: TextureInstance,
    // Single texture with only the real values to store height, dx and dz
    pub texture_packed: TextureInstance,
    pub bind_groups_ping: Vec<wgpu::BindGroup>,
    pub bind_groups_pong: Vec<wgpu::BindGroup>,
    // Intial Phillips spectrum data passed onto the GPU
    pub initial_data_group: wgpu::BindGroup,
    pub initial_data_buffer: wgpu::Buffer,
    // Config buffer holding the vertex data required for butterfly passes
    pub config_buffer: wgpu::Buffer,
    pub output_is_ping: bool,
    pub combine_uniform_buffer: wgpu::Buffer,
    // ...Height fields bind groups for both render and compute pipelines
    // ...other fields
}
```

Figure 90: *Modified snippet representing the fields in a* `CascadeResources` *struct*

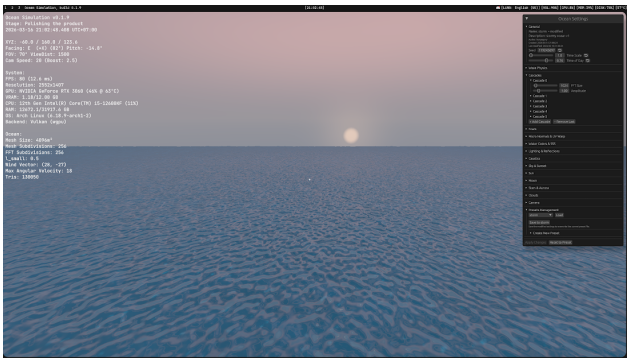
Now, instead of passing out one of the ping or pong textures to the main `render.wgsl` shader, all the cascades have to be added up. Therefore, a new compute pipeline `cascade.wgsl` (creativity at its peak) has to be created. Every frame, each cascade is handled individually first, where the spectrum is updated and butterfly passes are applied, and the `texture_packed` is generated. 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 process is really simple, where the data is extracted from the read texture and incremented onto the write texture as shown in Figure 91.

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

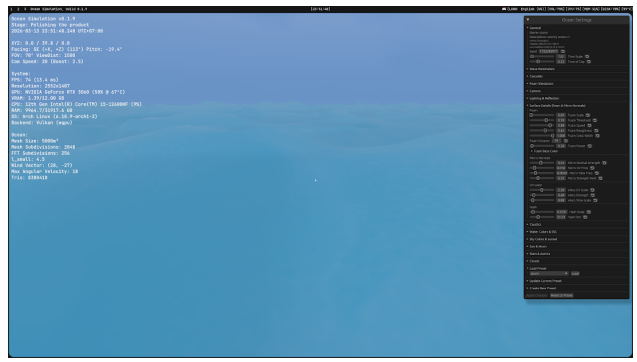
Afterwards this combined cascade texture will be used throughout the simulation, in the rendering pipeline, foam pipeline and so on. This new cascade system now allows for endless possible combinations of the ocean, where some examples can be seen in Figure 92.

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

3.13 Limitations and issues

Any simulation will have its own set of issues and challenges that it faces, and this one is no different. Since this discussion values the open-source model and honesty, all the issues are to be publicly shared.

3.13.1 The Blob Object

First and foremost, 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, but a fix that requires time, time that this project is not able to afford any more. The fix is to split up the whole struct into sub-modules, such as `Engine`, `FFT`, `Settings`, where each module is able to create itself, manage its own fields and the `State` struct will be the intermediate, managing the communication as shown in Figure 93.

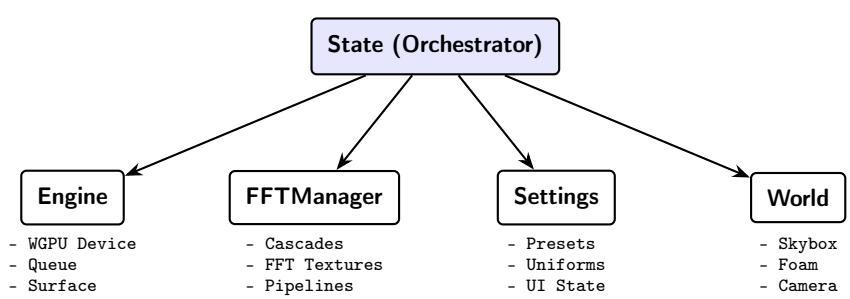


Figure 93: *The modular hierarchy*

3.13.2 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.

3.13.3 Visual Fidelity

Second, the simulation ultimately falls slightly behind the set expectations, with the foam not being as realistic as expected, general lighting effects being slightly off, as well as noticeable tiling. The noticeable tiling appears due to the fact that all the cascades have to be compressed into a single packed texture in the `combine_cascades` compute shader. This makes big FFT size cascades leave a noticeable seam at the edges. The proposed solution was to discard the combined texture, and just have an array of textures which will gold each cascades individual data, and then in the vertex and fragment shader a simple `for` loop would accumulate the displacements. This turned out to be more complex and introduced new issues, which the project can't afford due to time constraints. However, 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, a preset is dynamically extracted and parsed to fit the Uniform Struct expected for `wgpu`. This preset governs the settings for the whole simulation, including mesh generation, cascades and rendering. 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.

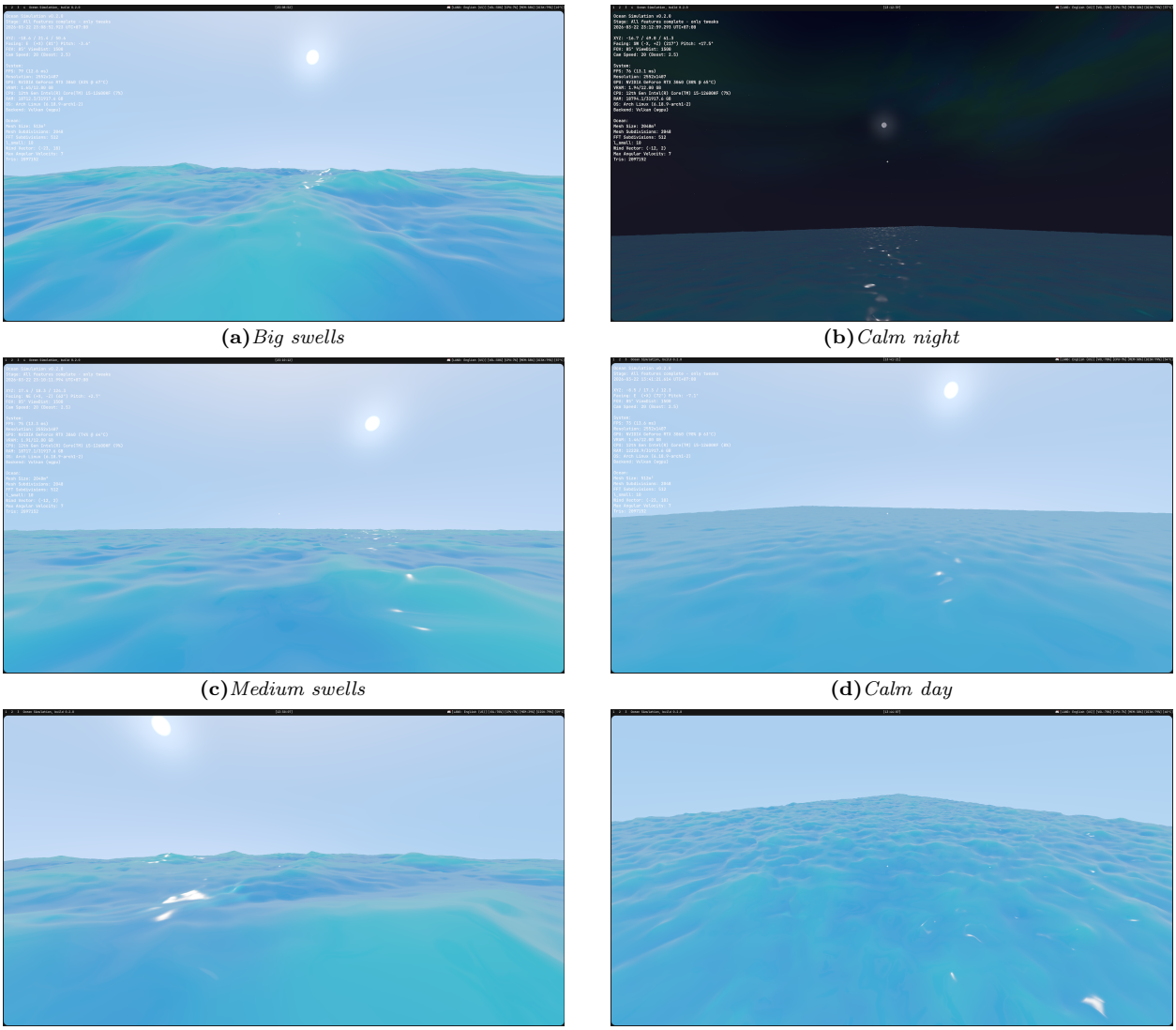
This data along with pre-calculated twiddle values are passed into the GPU compute shader, which uses a ping-pong texture model to compute an Inverse Fast Fourier Transform. This IFFT uses a DIT Tukey-Cooley Radix-2 butterfly passes to transform the spectrum from frequency domain into a time domain, 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 then 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, such as improved foam and 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 are implemented, which was not mentioned in the discussion, as the effect was not pronounced or realistic when taking into consideration 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 94, which does intact resemble an ocean surface.

In the end, the ocean scales nicely on a large scale with a high subdivision size. The simulation runs at approximately 70FPS, correlating to an average 14.5ms frame time, while using an *NVIDIA RTX 3060 12GB* [20] graphics card 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 94: *The resultant images with all effects combined, some containing debug information*

References

[1] 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).

[2] 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).

[3] 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).

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

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

[6] 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).

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

[8] Epic Games. *Introduction to Visual Scripting with Blueprints*. URL: https://www.unrealengine.com/fr/blog/introduction-to-blueprints (visited on 01/06/2026).

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

[10] 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).

[11] gfx-rs. *wgpu GitHub Repository*. URL: https://github.com/gfx-rs/wgpu (visited on 01/20/2026).

[12] 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).

[13] David Henry. *On Gerstner's Water Wave*. 2008. URL: https://link.springer.com/article/10.2991/jnmp.2008.15.s2.7 (visited on 01/20/2026).

[14] ISO/IEC JTC1/SC22/WG14. *The C Programming Language Standardization*. URL: https://www.open-std.org/JTC1/SC22/WG14/ (visited on 12/09/2025).

[15] 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).

[16] 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).

[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] 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).

[19] Mozilla Foundation. *Mozilla: Internet for People*. URL: https://www.mozilla.org/en-US/ (visited on 01/20/2026).

[20] 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).

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

[22] Emil Persson. *Sorsele Cubemap*. Polycount Wiki. 2014. URL: http://wiki.polycount.com/w/images/7/73/Cubemap_Sorsele_humus.jpg (visited on 03/17/2026).

[23] 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).

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

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

[26] 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).

[27] Christophe Schlick. "An Inexpensive BRDF Model for Physically-based Rendering". In: *Computer Graphics Forum* 13.3 (1994), pp. 233–246. (Visited on 03/14/2026).

[28] Brad Smith. *Blinn-Phong Reflection Model Components*. Wikimedia Commons. CC BY-SA 3.0. 2010. URL: https://commons.wikimedia.org/w/index.php?curid=1030364 (visited on 03/13/2026).

[29] Standard C++ Foundation. *The C++ Programming Language*. URL: https://isocpp.org/ (visited on 01/06/2026).

[30] George Gabriel Stokes. "On the Theories of the Internal Friction of Fluids in Motion". In: *Transactions of the Cambridge Philosophical Society* 8 (1845), pp. 287–341. (Visited on 03/22/2026).

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

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

[33] The Rust Team. *Defining and Instantiating Structs*. URL: https://doc.rust-lang.org/book/ch05-01-defining-structs.html (visited on 03/19/2026).

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

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

[36] 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).

[37] wgpu Project. *Supported wgpu Backends*. URL: https://wgpu.rs/doc/wgpu/enum.Backend.html (visited on 01/20/2026).

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