Real-Time, Empirical, Ocean Simulation & Physically Based Renderer

A-Level Computer Science NEA

Zayaan Azam

Contents

0.1. Abstract

// synopsis

1. Analysis

1.1. Client Introduction

The client is Jahleel Abraham. They are a game developer who require a physically based, performant, configurable simulation of an ocean for use in their game. They also require a physically based lighting model derived from microfacet theory, including PBR specular, and empirical subsurface scattering. Also expected is a fully featured GUI allowing direct control over every input parameter, and a functioning camera controller.

1.2. Interview Questions

Interview notes are paraphrased.

Q: What specific ocean phenomena need to be simulated?
A: The simulation should include waves in all real-world conditions and generate foam. If possible, it should also simulate other ocean phenomena.
Q: What parameters of the simulation need to be configurable?
A: All necessary parameters for simulating real-world conditions, including tile size and individual wave quantity.
Q: Does there need to be an accompanying GUI?
A: Yes, the GUI should allow control over parameters and tile size, display frametime, and show the current state of all parameters.
Q: Do I need to implement an atmosphere/skybox?
A: A basic skybox would be nice, and an atmosphere shader should be included if possible.
Q: Do I need to implement a PBR water shader?
A: Yes, the simulation should use a physically based rendering (PBR) water shader with a microfacet BRDF.
Q: Do I need to implement caustics, reflections, or other light-related phenomena?
A: Caustics are out of scope. Implement approximate subsurface scattering and use GGX distribution with the BRDF to simulate reflections.
Q: Are there any limitations due to existing technology?
A: The client has not started technical implementation, so they are not limited by an existing technical stack.
Q: Does this need to interoperate with existing code?
A: See the response to technical limitations (no existing stack constraints).
Q: Are there limitations due to the target device(s)?
A: The simulation should run on both x86 and ARM64 devices.
Q: Are there other performance-intensive systems in place?
A: See the response to technical limitations.
Q: Is the product targeted to low/mid/high-end systems?
A: The simulation is primarily targeted at mid-to-high-end systems, but ideally, it should also be performant on lower-end hardware.

1.3. Similar Solutions

As this is a fairly niche simulation that is generally used in closed source applications, there were only a few sources I could find that are relevant and have available information in relation to their implementation. Understanding existing implementations, particularly in regard to their spectrum synthesis, has allowed me to develop the correct balance between fidelity and performance.

1.3.1. Sea of Thieves

Sea of thieves is by far the highest profile game utilising an FFT ocean. It focuses on developing a stylised ocean scene over a realistic one so differs in the following ways:

- Does not have exact PBR lighting, but still derives its lighting from PBR theory
- Is intended to have minimal frametime impact while operating on low end hardware, so has a lower resolution, using a less complicated spectrum
- Has a more detailed foam simulation, where they convolve the foam map with a stylised texture, and use Unreal Engine's particle system to simulate sea spray

The implementation is closed source, so I do not have access to any detailed implementation information. Anything I could find was primarily from a conference they held on its development.

1.3.2. Acerola

Acerola's ocean simulation was made as part of a pseudo-educational youtube series on simulating water in games. It has a few notable similarities, and key benefits over others:

- It is a realistic simulation, using a spectrum that is (As far as I am aware) fully in-line with the most recent oceanographic literature
- Uses a highly performant FFT implementation, based on Nvidia WaveWorks
- Is built upon unity and utilises full PBR lighting, as derived from microfacet theory

1.3.3. Jump Trajectory

Jump trajectory created a very detailed video explaining the process of creating an FFT-based ocean simulation, also sharing the source code publically. It is similar to acerolas in most ways, but has a few notable differences:

- uses a "simpler" FFT implementation, that is much easier to understand
- Abstracts his data and algorithms into different compute passes
- Decays foam non-exponentially, using a flat base color for foam

1.4. Success Criteria

1. The application opens with a titled, resizable, movable, and closable OS window that follows OS styling.
2. A single skybox texture can be loaded from a specified filepath.
3. The camera can be controlled via mouse input, with left-click activation, full panning, scroll-based zoom, and stable field of view across window resizes.
4. A user interface (UI) is present, featuring resizable, movable, and collapsable panels, color pickers, sliders, an FPS display, and simulation details.
5. The ocean simulation exhibits Gaussian wave variance with three controllable length scales and frequency cutoffs.
6. The simulation runs at a minimum of 60 FPS on mid-range gaming hardware (GTX 1060 and above).
7. The simulation has sensible default parameters
8. The ocean has adjustable size, including tile size, instance count, and simulation resolution.
9. Foam effects are implemented with adjustable color, decay rate, visibility, and wave-breaking injection settings.
10. The ocean conditions are user-adjustable, including depth, gravity, wind parameters, fetch, choppiness, and swell properties.
11. A skybox with a correctly transformed sun is rendered, responding correctly to camera view and zoom
12. The ocean surface has no obvious pixellation beyond that as a result of the simulation resolution
13. The ocean has approximate subsurface scattering, with adjustable scatter color, bubble effects, and lighting properties.
14. Real-time reflections from the environment are visible, influenced by user-controllable fresnel effects and refractive indices.
15. Specular reflections are implemented with toggleable PBR/non-PBR modes, adjustable shininess, and roughness settings.
16. Distance fog is implemented with user-adjustable density, offset, falloff, and height settings.

1.5. Spectrum Synthesis

1.5.1. Nomenclature

To compute the inital spectra, we rely on various input parameters and definitions shared between various functions and parts of the program. Table 1 lists the relevant symbols, meanings and relationships where appropriate. Note that throughout this project we are defining the positive y direction as “up”.

Variable	Definition
$\vec{k}$	2D Wave Vector
k_x	X Component of wave vector
k_z	Z Component of wave vector
k	Magnitude of the wave vector
t	time
m, n	Indices of summation
M, N	Dimensions of summation
$\vec{x} = [x_x, x_z]$	Position vector in world space
$\omega = \varphi(k)$	Dispersion relation
ω_p	Peak Frequency
g	Gravitational Constant = 9.81
h	Ocean Depth
U_{10}	Wind speed 10m above surface
F	Fetch, distance over which wind blows
θ	Wind direction = $\arctan2(k_z, k_x) - \theta_0$
θ_0	Wind direction offset
L	Lengthscale
$L_x = L_z$	Simulation Dimensions, power of two
ξ	Swell amount
ξ_r, ξ_i	Gaussian Random Numbers
λ	Choppiness factor
κ	Foam bias
μ	Foam injection threshold
ζ	Foam decay rate
I	Foam value

Table 1: Simulation Variable Definitions

1.5.2. Dispersion Relation

Citations: [4], [5]

The relation between the travel speed of the waves and their wavelength, written as a function relating angular frequency ω to wave number $\vec{k}$. This simulation involves finite depth, and so we will be using a dispersion relation that considers it.[5]

$$\omega = \varphi(k) = \sqrt{gk\tanh(kh)}$$

$$\frac{d\varphi(k)}{dk} = \frac{g(\tanh(hk) + hk\,\mathrm{sech}^2(hk))}{2\sqrt{gk\tanh(hk)}}$$

1.5.3. Non-Directional Spectrum (JONSWAP)

Citations: [5], [6], [7], [8]

The JONSWAP energy spectrum is a more parameterised version of the Pierson-Moskowitz spectrum, and an improvement over the Philips Spectrum used in [4], simulating an ocean that is not fully developed (as recent oceanographic literature has determined this does not happen). The increase in parameters allows simulating a wider breadth of real world conditions.

$$S_{\text{JONSWAP}}(\omega) = \frac{\alpha g^2}{\omega^5}\ \exp\ \left[-\frac{5}{4}\left(\frac{\omega_p}{\omega}\right)^4\right] 3.3^r$$

$$r = \exp\left[-\frac{\left(\omega-\omega_p\right)^2}{2\omega_p^2\sigma^2}\right]$$

$$\alpha = 0.076\left(\frac{U_{10}^2}{Fg}\right)^{0.22}$$

$$\omega_p = 22\left(\frac{g^2}{U_{10}F}\right)^{\frac{1}{3}}$$

$$\sigma = \begin{cases} 0.07 \text{ if } \omega \leq \omega_p \\ 0.09 \text{ if } \omega > \omega_p \end{cases}$$

1.5.4. Depth Attenuation Function (Approximation of Kitaiigorodskii)

Citations: [5]

JONSWAP was fit to observations of waves in deep water. This function adapts the JONSWAP spectrum to consider ocean depth, allowing a realistic look based on distance to shore. The actual function is quite complex for a relatively simple graph, so can be well approximated as below [5].

$$\Phi(\omega, h) = \begin{cases} \frac{1}{2}\omega_h^2 \text{ if } \omega_h \leq 1 \\ 1 - \frac{1}{2}(2-\omega_h)^2 \text{ if } \omega_h > 1 \end{cases}$$

$$\omega_h = \omega\sqrt{\frac{h}{g}}$$

1.5.5. Directional Spread Function (Donelan-Banner)

Citations: [5]

The directional spread models how waves react to wind direction [7]. This function is multiplied with the non-directional spectrum in order to produce a direction dependent spectrum [5].

$$D(\omega,\theta) = \frac{\beta_s}{2\tanh(\beta_s\pi)}\operatorname{sech}^2(\beta_s\theta)$$

$$\beta_s = \begin{cases} 2.61\left(\frac{\omega}{\omega_p}\right)^{1.3} \text{ if } \frac{\omega}{\omega_p} < 0.95 \\ 2.28\left(\frac{\omega}{\omega_p}\right)^{-1.3} \text{ if } 0.95 \leq \frac{\omega}{\omega_p} < 1.6 \\ 10^{\varepsilon} \text{ if } \frac{\omega}{\omega_p} \geq 1.6 \end{cases}$$

$$\varepsilon = -0.4 + 0.8393\exp\left[-0.567\ln\left(\left(\frac{\omega}{\omega_p}\right)^2\right)\right]$$

1.5.6. Swell

Citations: [5]

Swell refers to the waves which have travelled out of their generating area [5]. In practice, these would be the larger waves seen over a greater area. the directional spread function including swell is based on combining donelan-banner with a swell function as below. The integral seen in Q_{final} is computed numerically using the rectangle rule. Q_ξ is a normalisation factor to satisfy the condition specified in equation (31) in [5], approximation taken from [7]

$$D_{\text{final}}(\omega,\theta) = Q_{\text{final}}(\omega)D_{\text{base}}(\omega,\theta)D_\varepsilon(\omega,\theta)$$

$$Q_{\text{final}}(\omega) = \left(\int_{-\pi}^{\pi} D_{\text{base}}(\omega,\theta)D_\xi(\omega,\theta)d\theta\right)^{-1}$$

$$D_\xi = Q_\xi(s_\xi)\ |\cos\left(\frac{\theta}{2}\right)|^{2s_\xi}$$

$$s_\xi = 16\tanh\left(\frac{\omega_p}{\omega}\right)\xi^2$$

$$Q_\xi(s_\xi) = \begin{cases} -0.000564s_\xi^4 + 0.00776s_\xi^3 - 0.044s_\xi^2 + 0.192s_\xi + 0.163 \text{ if } s_\xi < 5 \\ -4.80\times10^{-}8s_\xi^4 + 1.07\times10^{-5}s_\xi^3 - 9.53\times10^{-4}s_\xi^2 + 5.90\times10^{-2}s_\xi + 3.93e-01 \text{ otherwise} \end{cases}$$

1.5.7. Directional Spectrum Function

Citations: [5]

The TMA spectrum below is an undirectional spectrum that considers depth.

$$S_{\mathrm{TMA}}(\omega, h) = S_{\mathrm{JONSWAP}}(\omega)\Phi(\omega, h)$$

This takes inputs ω, h, whilst we need it to take input $\vec{k}$ per Tessendorf [4] - in order to do this we apply the following transformation [5]. Similarly, to make the function directional, we also need to multiply it by the directional spread function [5].

$$S_{\mathrm{TMA}}\left(\vec{k}\right) = 2S_{\mathrm{TMA}}(\omega, h)\frac{d\omega}{dk}\frac{1}{k}\Delta k_x \Delta k_z D(\omega, \theta)$$

$$\Delta k_x = \Delta k_z = \frac{2\pi}{L}$$

1.6. Ocean Geometry & Foam

1.6.1. Displacements

Citations: [4], [7], [8], [9]

For a field of dimensions L_x and L_z, we calculate the displacements at positions $\vec{x}$ by summating multiple sinusoids with complex, time dependant amplitudes. [4]. By arranging the equations into a specific format, we can convert the frequency domain representation of the wave into the spatial domain using the inverse discrete fourier transform.

$$\text{Vertical Displacement (y)} : h(\vec{x}, t) = \sum_{\vec{k}} \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

$$\text{Horizontal Displacement (x)}: \lambda D_x(\vec{x}, t) = \sum_{\vec{k}} -i\frac{\vec{k}_x}{k} \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

$$\text{Horizontal Displacement (z)}: \lambda D_z(\vec{x}, t) = \sum_{\vec{k}} -i\frac{\vec{k}_z}{k} \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

1.6.2. Derivatives

Citations: [4], [7]

For lighting calculations and the computation of the jacobian, we require the derivatives of the above displacements. As the derivative of a sum is equal to the sum of the derivatives, we compute exact derivatives using the following summations.

$$\text{X Component of Normal} : \varepsilon_x(\vec{x}, t) = \sum_{\vec{k}} ik_x \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

$$\text{Z Component of Normal} : \varepsilon_z(\vec{x}, t) = \sum_{\vec{k}} ik_z \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

$$\text{Jacobian xx} : \frac{dD_x}{dx} = \sum_{\vec{k}} -\frac{k_x^2}{k} \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

$$\text{Jacobian zz} : \frac{dD_x}{dx} = \sum_{\vec{k}} -\frac{k_z^2}{k} \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

$$\text{Jacobian xz} : \frac{dD_x}{dz} = \sum_{\vec{k}} -\frac{k_x k_z}{k} \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

1.6.3. Frequency Spectrum Function

Citations: [4], [7], [8]

This function defines the amplitude of the wave at a given point in space at a given time depending on it's frequency. The frequency is generated via the combination of 2 gaussian random numbers and a energy spectrum in order to simulate real world ocean variance and energies. Note that the

time dependant component of the exponent is pushed into this amplitude, in order to simplify the summation.

$$\hat{h}_0\left(\vec{k}\right) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{S_{\text{TMA}}\left(\vec{k}\right)}$$

$$\hat{h}\left(\vec{k}, t\right) = \hat{h}_0\left(\vec{k}\right)e^{i\varphi(k)t} + h_0(-k)e^{-i\varphi(k)t}$$

1.6.4. Box-Muller Transform

Citations: [10]

The ocean exhibits gaussian variance in the possible waves. Due to this the frequency spectrum function is varied by gaussian random numbers with mean 0 and standard deviation 1, which we generate using the box-muller transform converting from uniform variates from 0..1. [4]. Derivation is from polar coordinates, by treating x and y as cartesian coordinates, more details at [10]

$$\xi_r, \xi_i \sim N(0, 1)$$

$$\xi_r = \sqrt{-2.0\ln(u_1)}\cos(2\pi u_2)$$

$$\xi_i = \sqrt{-2.0\ln(u_1)}\sin(2\pi u_2)$$

1.6.5. Foam, The Jacobian, and Decay

Citations: [4], [5], [8]

The jacobian describes the "uniqueness" of a transformation. This is useful as where the waves would crash, the jacobian determinant of the displacements goes negative. Per Tessendorf [4], we compute the determinant of the jacobian for the horizontal displacement, $D(\vec{x}, t)$.

$$J(\vec{x}) = J_{\text{xx}}J_{\text{zz}} - J_{\text{xz}}J_{\text{zx}}$$

$$J_{\text{xx}} = 1 + \lambda\frac{dD_x}{dx}, J_{\text{zz}} = 1 + \lambda\frac{dD_z}{dz}$$

$$J_{\text{xz}} = J_{\text{zx}} = 1 + \lambda\frac{dD_x}{dz}$$

we then threshold the value such that $J(\vec{x}) < \mu$, wherein if true we inject foam into the simulation at the given point. This value should accumulate (and decay) over time to mimic actual ocean foam, which is achieved by modulating the previous foam value (I_0) by an exponential decay function:

$$J_{\text{biased}} = \kappa - J(\vec{x})$$

$$I = \begin{cases} I_0 e^{-\zeta} + J_{\text{biased}} \text{ if } J_{\text{biased}} > \mu \\ I_0 e^{-\zeta} \text{ if } J_{\text{biased}} < \mu \end{cases}$$

1.6.6. Cascades

Citations: [4], [5], [7]

The periodicity of sinusoidal waves leads to visible tiling, even with an amount of waves of order 10^6. It is possible to increase the simulation resolution to counteract this, but even the FFT becomes computationally impractical at large enough scales. To counteract this, we instead compute the entire simulation multiples times with different lengthscales at a lower resolution - combining the results such that there is no longer any visual tiling. This results in an output with comparable quality to an increase in resolution, while requiring less overall calculations, e.g $3(256^2) < 512^2$. To do this, we have to select low and high cutoffs for each lengthscale such that the waves do not overlap and superposition.

1.6.7. 2D GPGPU Cooley-Tukey Radix-2 Inverse Fast Fourier Transform

Citations: [4], [7]

The Cooley-Tukey FFT is a common implementation of the FFT, used for fast calculation of the DFT. The direct DFT is computed in $O(n^2)$ time whilst the FFT is computed in $O(n \log n)$. This is a significant improvement as we are dealing with n in the order of $10^4 - 10^6$, computed multiple times per frame. The FFT exploits the redundancy in DFT computation in order to increase performance, being able to do so only when $L_x = L_z = M = N = 2^x, x \in \mathbb{Z}$, the coordinates and wave vectors lie on a regular grid, and the summation is in the following format, which are all true save some differences in summation limits.

$$\text{Inverse DFT: } F_n = \sum_{m=0}^{N-1} f_m e^{2\pi i \frac{m}{N} n}$$

$$\text{Wave Summation: } h(\vec{x}, t) = \sum_{m=-\frac{N}{2}}^{\frac{N}{2}} h(t) e^{2\pi i \frac{m}{N} n}$$

1.7. Post Processing

1.7.1. Nomenclature

Definitions of the initial variables and parameters for post processing. Note that there are alot of omitted scaling variables and similar that are applied throughout the shader, these are all of the format `consts.sim/shader.variable_name`

Variable	Definition
L_{eye}	final light value for a fragment
L_{scatter}	light re-emitted due to subsurface scattering
L_{specular}	light reflected from the sun
$L_{\text{env\_reflected}}$	light reflected from the environment
F	Fresnel Reflectance
$\hat{H}$	Halfway vector
$\hat{N}$	Surface normal
$\hat{V}$	Camera view vector
$\hat{L}$	Light source vector
k_1	Subsurface Height Attenuation
k_2	Subsurface Reflection Scale Factor
k_3	Subsurface Diffuse Scale Factor
k_4	Subsurface Ambient Scale Factor
C_{ss}	Water Scatter Color
C_f	Air Bubbles Color
L_{sun}	Sun Color
P_f	Air Bubbles Density
W_{max}	Max between 0 and wave height
$\langle \omega_a, \omega_b \rangle$	$\max(0, (\omega_a \cdot \omega_b))$
λ_{GGX}	Smith's G_1 masking function
n_1	Refractive index of water
n_1	Refractive index of air
S	The Shininess of the material for fresnel
B	The Shininess of the material for blinn phong
G_2	Smith's G_2 geometric attenuation function
D_{GGX}	GGX Distribution of microfacet normals
α	Roughness of the surface

Table 2: Post Processing Variable Definitions

1.7.2. Rendering Equation

Citations: [1], [8], [11]

This abstract equation models how a light ray incoming to a viewer is "formed" (in the context of this simulation). Due to there only being a single light source (the sun), subsurface scattering [11] can be used to replace the standard L_{diffuse} and L_{ambient} terms.

To include surface foam, we *lerp* between the foam color and L_{eye} based on foam density [11]. We also Increase the roughness in areas covered with foam for L_{specular} [11].

$$L_{\text{eye}} = (1 - F)L_{\text{scatter}} + L_{\text{specular}} + FL_{\text{env\_reflected}}$$

1.7.3. Normalisation & Surface Normals

Citations: [2], [4], [7]

When computing lighting using vectors, we are only concerned with the direction of a given vector not the magnitude. In order to ensure the dot product of 2 vectors is equal to the cosine of their angle we normalise the vectors.

In order to compute the surface normals we sum, for each lengthscale, the value of $\varepsilon(\vec{x}, t)$ such that the following is true, which is then normalised.

$$\vec{N} = \begin{bmatrix} -\varepsilon_x(\vec{x}, t) \\ 1 \\ -\varepsilon_z(\vec{x}, t) \end{bmatrix}$$

1.7.4. Subsurface Scattering

Citations: [8], [11]

This is the phenomenon where some light absorbed by a material eventually re-exits and reaches the viewer. Modelling this realistically is impossible in a real time context (with my hardware). Specifically within the context of the ocean, we can approximate it particularly well as the majority of light is absorbed. An approximate formula taking into account geometric attenuation, a crude fresnel factor, lamberts cosine law, and an ambient light is used, alongside various artistic parameters to allow for adjustments. [11]

$$L_{\text{scatter}} = \frac{\Big(k_1 W_{\max}\langle \hat{L}, -\hat{V}\rangle^4 \Big(0.5 - 0.5\Big(\hat{L} \cdot \hat{N}\Big)\Big)^3 + k_2 \langle \hat{V}, \hat{N}\rangle^2\Big) C_{\text{ss}} L_{\text{sun}}}{1 + \lambda_{\text{GGX}}}$$

$$L_{\text{scatter}} \mathrel{+}= k_3 \langle \hat{L}, \hat{N}\rangle C_{\text{ss}} L_{\text{sun}} + k_4 P_f C_f L_{\text{sun}}$$

1.7.5. Fresnel Reflectance (Schlick's Approximation) [1], [2], [3]

The fresnel factor is a multiplier that scales the amount of reflected light based on the viewing angle. The more grazing the angle the more light is refleceted.

$$F\Big(\hat{N}, \hat{V}\Big) = F_0 + (1 - F_0)\Big(1 - \hat{N} \cdot \hat{V}\Big)^S$$

$$F_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

1.7.6. Blinn-Phong Specular Reflection

Citations: [2]

This is a simplistic, empirical model to determine the specular reflections of a material. It allows you to simulate isotropic surfaces with varying roughnesses whilst remaining very computationally efficient. The model uses "shininess" as an input parameter, whilst the standard to use roughness (due to how PBR models work). In order to account for this when wishing to increase roughness we decrease shininess.

$$L_{\text{specular}} = \left(\hat{N} \cdot \hat{H}\right)^B$$

$$\hat{H} = \hat{L} + \hat{V}$$

1.7.7. Environment Reflections

Citations: [1], [2]

In order to get the color of the reflection for a given pixel, we compute the reflected vector from the normal and view vector. We then sample the corresponding point on the skybox and use that color as the reflected color.

$$\hat{R} = 2\hat{N}\left(\hat{N} \cdot \hat{V}\right) - \hat{V}$$

1.7.8. GGX Distribution

Citations: [12]

The distribution function used in the BRDF to model the proportion of microfacet normals aligned with the halfway vector. This is an improvement over the beckmann distribution due to the graph never reaching 0 and only tapering off at the extremes.

$$D_{\text{GGX}} = \frac{\alpha^2}{\pi\left((\alpha^2 - 1)\left(\hat{N} \cdot \hat{H}\right)^2 + 1\right)^2}$$

1.7.9. Geometric Attenuation

Citations: [12]

Used to counteract the fresnel term, mimics the phenomena of masking & shadowing presented by the microfactets. The λ_{GGX} term changes depending on the distribution function used (GGX). $\hat{S}$ is either the light or view vector.

$$G_2 = G_1\left(\hat{H}, \hat{L}\right)G_1\left(\hat{H}, \hat{V}\right)$$

$$G_1\left(\hat{H}, \hat{S}\right) = \frac{1}{1 + a\lambda_{\text{GGX}}}$$

$$a = \frac{\hat{H} \cdot \hat{S}}{\alpha\sqrt{1 - \left(\hat{H} \cdot \hat{S}\right)^2}}$$

$$\lambda_{\text{GGX}} = \frac{-1 + \sqrt{1 + a^{-2}}}{2}$$

1.7.10. Microfacet BRDF

Citations: [8], [11], [12]

This BRDF (Bidirectional Reflectance Distribution Function) is used to determine the specular reflectance of a sample. There are many methods of doing this - the one used here is derived from microfacet theory. D can be any distribution function - the geometric attenuation function G changing accordingly.

$$L_{\text{specular}} = \frac{L_{\text{sun}} F G_2 D_{\text{GGX}}}{4\left(\hat{N} \cdot \hat{L}\right)\left(\hat{N} \cdot \hat{V}\right)}$$

1.7.11. Reinhard Tonemapping

Citations: [13]

To account for the HDR output values of some lighting functions, we tonemap the final output to be within a 0..1 range by dividing a color c as follows (given that c is effectively a 3D Vector)

$$c_{\text{final}} = \frac{c}{c + [1, 1, 1]}$$

1.7.12. Distance Fog & Sun

Citations: [1]

To hide the imperfect horizon line we use a distance fog attenuated based on height and distance. This is generated by exponentially decaying a fog factor based on the relative height of the fragment compared to the ocean, then *lerping* between the fog color and sky_color based on this. For the ocean surface we instead *lerp* based on the distance from camera compared to a max distance, and then decaying and offsetting based on input parameters.

To render the sun, I compare the dot product of the ray and sun directions to the cosine of the maximum sky angle the sun can occupy and then *lerp* between the sun and sky color based on a linear falloff factor.

1.8. Prototyping

A prototype was made in order to test the technical stack and gain experience with graphics programming and managing shaders. I created a Halvorsen strange attractor [14], using differential equations, and then did some trigonometry to create a basic camera controller using Winit's event loop.

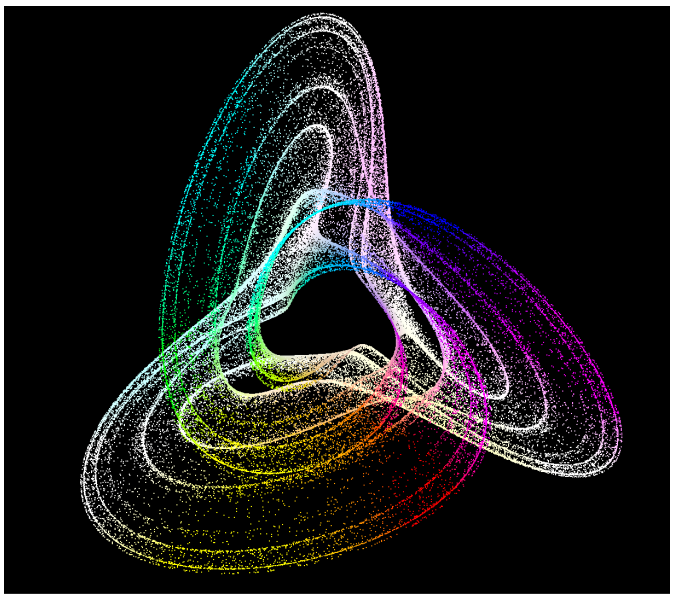

Figure 1: Shader Output, Found at https://github.com/CmrCrabs/chaotic-attractors

1.9. Additional Features

If given enough time I would like to implement the following:

- Swell, the waves which have travelled out of their generating area [5].
- Further post processing effects, such as varying tonemapping options and a toggleable bloom pass
- A sky color simulation, as this would allow the complete simulation of a realistic day night cycle for any real world ocean condition.
- LEADR environment reflections, based on the paper by the same name (Linear Efficient Antialiased Displacement and Reflectance Mapping)

1.10. Project Objectives

1.10.1. Engine

1 there is an os window created on startup
   1.1 the window has a appropriate title
   1.2 the window follows the OS's conventions and compositor styling
   1.3 the window can be resized without breaking the simulation
   1.4 the window can be moved
   1.5 the window can move between monitors without breaking
   1.6 the window can be closed by pressing the escape key
2 the scene has a single mesh stored on the cpu
   2.1 the user can resize the mesh
   2.2 there are multiple instances of the mesh visible
   2.3 the user can control how many instances there are
3 a filepath can be specified such that a single skybox .exr texture is read into GPU memory
4 the user can control the camera
   4.1 the camera can only be controlled when left-click is held down
   4.2 the camera's pitch can be controlled by moving the mouse
   4.3 the camera's yaw can be controlled by moving the mouse
   4.4 the camera's zoom can be controlled by using the scroll wheel
   4.5 the camera's render distance is large enough to see the entire ocean
   4.6 the cameras field of view does not change upon window resize
   4.7 the cameras view direction does not change upon window resize
5 the user can only control the camera if the UI is not selected
6 the user can access a user interface
   6.1 the user can resize the UI
   6.2 the user can move the UI
   6.3 the user can collapse the UI
   6.4 the user can edit colors using the UI
   6.5 the user can control sliders using the UI
   6.6 the user can read the fps from the UI
   6.7 the user can get the simulation resolution from the UI
   6.8 the user can see the time elapsed from the UI

1.10.2. Simulation

1 The simulation exhibits gaussian variance in the possible waves
2 The simulation is performant, being able to run at above 60fps on mid-range gaming hardware (gtx 1060+)
3 the simulation should have sensible default parameters
4 the simulations input parameters should be clamped such that changing parameters cannot permanently break the simulation
5 the simulation should not have visible tiling
6 the simulation should have 3 lengthscales, with user control over
   6.1 all 3 lengthscales

6.2 all 3 low frequency cutoffs
6.3 all 3 high frequency cutoffs
7 the simulation should have a controllable size
7.1 user adjustable tile size
7.2 user adjustable instance quantity
7.3 user adjustable simulation resolution
8 the simulation should support foam, with the following controllable conditions
8.1 the foam color
8.2 how fast the foam decays
8.3 how much foam is visible
8.4 when foam is injected on breaking waves
8.5 how much foam is injected on breaking waves
9 the user should be able to control the ocean conditions
9.1 user adjustable ocean depth
9.2 user adjustable gravitational field strength
9.3 user adjustable wind speed
9.4 user adjustable wind angle
9.5 user adjustable fetch
9.6 user adjustable choppiness
9.7 user adjustable amount of swell
9.8 user adjustable integration step for swell computation
9.9 user adjustable height offset for the ocean surface

1.10.3. Renderer

1 the scene has a skybox
1.1 the skybox is correctly transformed when the camera view direction is changed
1.2 the skybox is correctly transformed when the camera zoom is changed
1.3 the skybox is correctly transformed when the window is resized
1.4 there is a sun interpolated into the skybox
1.4.1 the sun is in the correct position in the sky relative to the light vector
1.4.2 the sun is correctly transformed when the camera view direction is changed
1.4.3 the sun is correctly transformed when the camera zoom is changed
1.4.4 the sun is correctly transformed when the window is resized
1.4.5 the sun has user controllable parameters
1.4.5.1 user adjustable sun color
1.4.5.2 user adjustable sun x direction
1.4.5.3 user adjustable sun y direction
1.4.5.4 user adjustable sun z direction
1.4.5.5 user adjustable sun angle
1.4.5.6 user adjustable sun distance
1.4.5.7 user adjustable sun size
1.4.5.8 user adjustable sun falloff factor
2 displacement map sampled such that there is no visible pixelation in output
3 normal map sampled such that there is no visible pixelation in output

4 foam map sampled such that there is no visible pixelation in output
5 there is visible, user controllable, subsurface scattering for ocean base color
  5.1 user adjustable scatter color
  5.2 user adjustable bubble color
  5.3 user adjustable bubble density
  5.4 user adjustable height attenuation factor
  5.5 user adjustable reflection strength
  5.6 user adjustable diffuse lighting strength
  5.7 user adjustable ambient lighting strength
6 there is visible reflections from the environment
  6.1 user adjustable reflection strength
  6.2 there is visible fresnel reflectance affecting reflections
    6.2.1 user adjustable water refractive index
    6.2.2 user adjustable air refractive index
    6.2.3 user adjustable fresnel shine
    6.2.4 user adjustable fresnel effect scale factor
    6.2.5 user adjustable fresnel normals scale factor
7 there is non physically based specular reflections
  7.1 user toggleable pbr / non pbr specular
  7.2 user adjustable shininess of surface
8 there is user controllable physically based specular reflections
  8.1 user adjustable specular scale factor
  8.2 user adjustable pbr fresnel scale factor
  8.3 user adjustable pbr cutoff low
  8.4 user adjustable water roughness
  8.5 user adjustable foam roughness factor
9 there is user controllable distance fog
  9.1 user adjustable fog density
  9.2 user adjustable fog distance offset
  9.3 user adjustable fog falloff factor
  9.4 user adjustable fog height offset

2. Documented Design

2.1. Technologies

Library	Version	Purpose	Link
Rust	-	Fast, memory-efficient programming language	https://www.rust-lang.org/
wgpu	23.0.1	Graphics library	https://github.com/gfx-rs/wgpu
Rust GPU	git	(Rust as a) shader language	https://github.com/Rust-GPU/rust-gpu
winit	0.29	Cross-platform window creation and event loop management	https://github.com/rust-windowing/winit
Dear ImGui	0.12.0	Bloat-free GUI library with minimal dependencies	https://github.com/imgui-rs/imgui-rs
imgui-wgpu-rs	0.24.0	only rendering code used, snippets taken directly from source instead of library being imported	https://github.com/yatekii/imgui-wgpu-rs
imgui-winit-support	0.13.0	only datatype translation code used, snippet taken directly from source instead of library being imported	https://github.com/imgui-rs/imgui-winit-support
GLAM	0.29	Linear algebra operations	https://github.com/bitshifter/glam-rs
Pollster	0.3	Used to read errors (async executor for wgpu)	https://github.com/zesterer/pollster
Env Logger	0.10	Logging for debugging and error tracing	https://github.com/env-logger-rs/env_logger
Image	0.24	Used to read a HDRI from a file into memory	https://github.com/image-rs/image
Nix	-	Creating a declarative, reproducible development environment	https://nixos.org/

2.2. Core Algorithm [4], [5]

Below is a high-level explanation of the core algorithm for the simulation. A visual representation can be seen on the next page.

On Startup:

- Generate gaussian random number pairs, and store them into a texture, on the CPU
- Compute the butterfly operation's twiddle factors, and indices, and store them into a texture

On Parameter Change (For Every Lengthscale):

- Compute the initial wave vectors and dispersion relation, and store into a texture
- Compute the initial frequency spectrum for each wave vector, and store into a texture
- Compute the conjugate of each wave vector, and store into a texture

Frame-by-Frame:

- For Every Lengthscale:
  - Evolve initial frequency spectrum through time
  - Pre-compute & store amplitudes for FFT into textures
  - perform IFFT for height displacement
  - perform IFFT for normal calculation
  - perform IFFT(s) for jacobian
  - Process & merge IFFT results into displacement, normal, and foam maps
- Create a fullscreen quad and render skybox, sun & fog to framebuffer
- Offset tiles based on instance index & centering
- Combine values for all 3 lengthscales & offset vertices based on displacement map
- Compute lighting value and render result to framebuffer

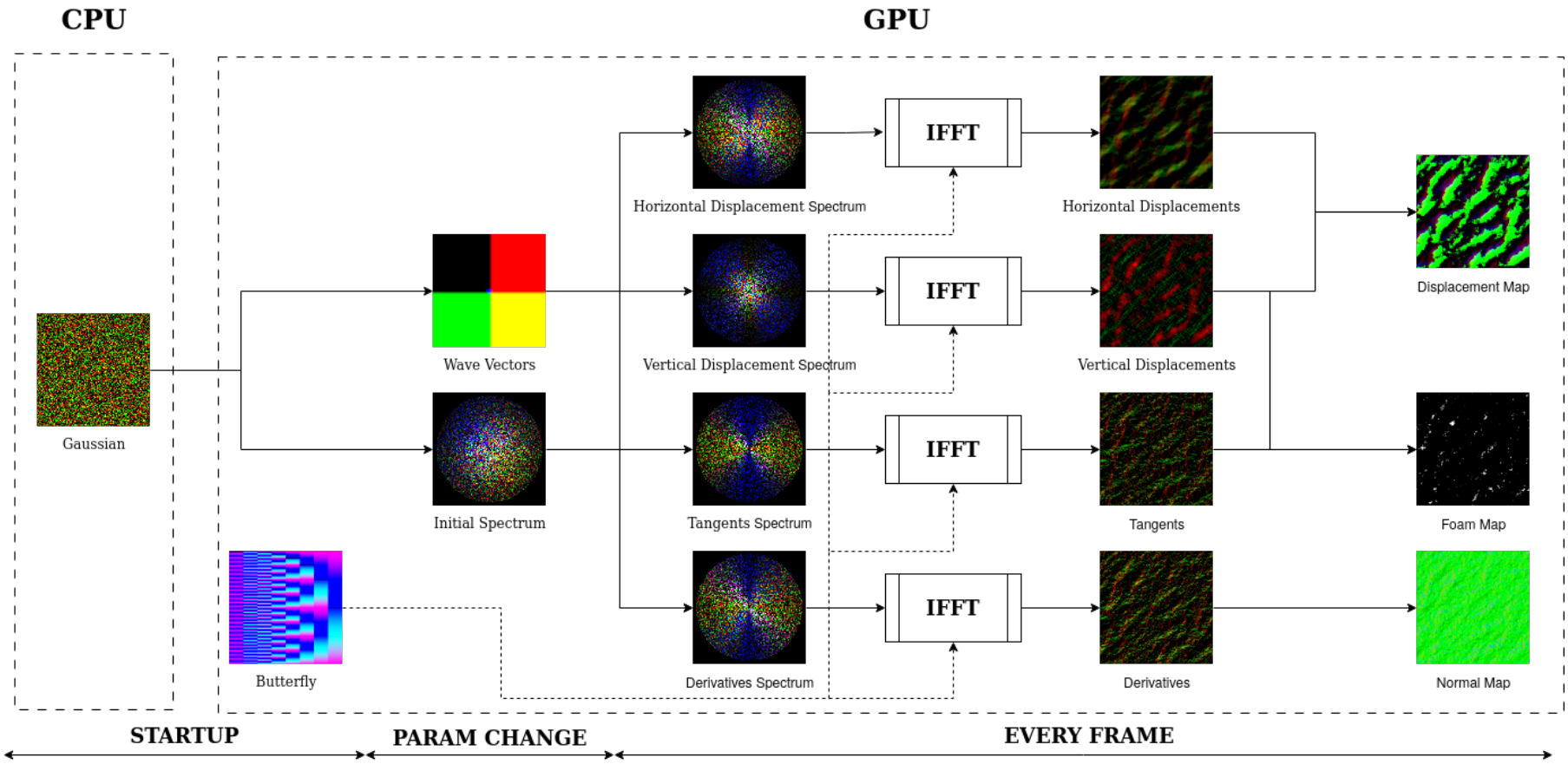

Figure 2: Visualisation of the core algorithm. Texture white / black points are not consistent for ease of demonstration

2.3. The Fourier Transform

Citations: [7], [8], [15], [16]

The IFFT algorithm I am using consists of 3 phases. The first phase involves precomputing the butterfly texture that holds the twiddle factors and input indices on the GPU. The second phase executes horizontal IFFT operations on the GPU, alternating between reading and writing to/from the input texture and a pingpong texture based on the pingpong push constant. The Third phase performs vertical IFFT operations on the same 2 textures.

Note that the "id" referenced in any pseudocode corresponds to the workgroup global invocation id, which is analogous in this context to the pixel coordinate of the texture we are operating over.

```
let pingpong = 0;
for i in 0..log_2(size) {
  dispatch_horizontal_IFFT_shader(i);
  pingpong = (pingpong + 1) % 2;
}
for i in 0..log_2(size) {
  dispatch_vertical_IFFT_shader(i);
  pingpong = (pingpong + 1) % 2;
}
dispatch_permute_shader();
```

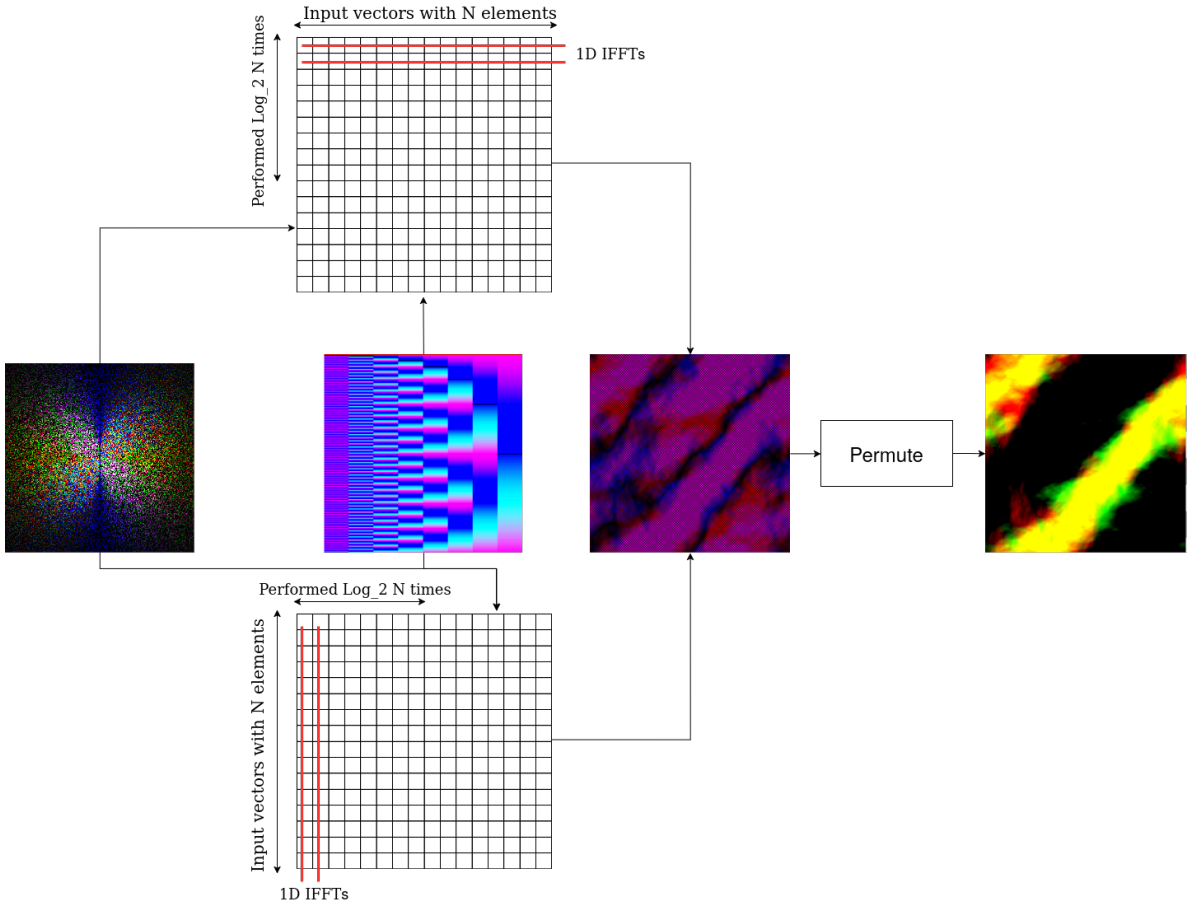

Figure 3: Visualised IFFT Algorithm showing the horizontal and vertical operations on an input texture

2.3.1. Butterfly Texture

Citations: [15]

For the following butterfly operations, input twiddle factors and indices are required. Following [15], we generate a texture with width $\log_2 N$ and height N, storing the real and complex parts of the twiddle factor into the r and g channels of the texture. We then store the input x/y indices into the b and a channels.

the twiddle factor at n is a complex number W_n^k such that

$$k = y_{\text{current}} \frac{n}{2^{\text{stage}+1}} \mod n$$

$$W_n^k = \exp\left(\frac{-2\pi i k}{n}\right)$$

which we represent in code using eulers formula as

$$W_n^k = \cos\left(\frac{-2\pi i k}{n}\right) + i \sin\left(\frac{-2\pi i k}{n}\right)$$

for the input indices, we compute a y_t and y_b, being the indices of the top and bottom wing respectively. for the first stage we sort our data in bit reversed order, meaning if we are in the first column of the texture, we should perform a bit reverse on the resulting indices.

We determine whether we are operating in the upper or lower "wing" based on the following equation, where we map 1 to be true and 0 to be false.

$$\text{wing} = y_{\text{current}} \bmod 2^{\text{stage}+1}$$

```
let yt = id.y;
let yb = id.y;
if id.x == 0 {
  if wing {
      yb += 1;
  } else {
      yt -= 1;
  }
  yt.bit_reverse();
  yb.bit_reverse();
} else {
  if wing {
      yb += step as u32;
  } else {
      yt -= step as u32;
  }
}
```

2.3.2. Butterfly Operations

Citations: [15], [16]

The FFT algorithm is designed to operate over 1D data, however our input is a 2D texture. To amend this, we perform a 1D IFFT on each row horizontally, and each column vertically, as shown in Figure 3. As our textures have 4 channels that can store 2 complex numbers each, we perform the butterfly operation twice per invocation, effectively cutting the needed IFFTs in half.

```
        let twiddle_factor = butterfly_texture.read(stage, id.x).rg();
        let indices = butterfly_texture.read(stage, id.x).ba();
        let top_signal = pingpong0.read(indices.x, id.y);
        let bottom_signal = pingpong0.read(indices.y, id.y);
        output = top_signal + complex_mult(twiddle_factor, bottom_signal);
```

Listing 1: Horizontal Butterfly Operation, over two sets of inputs

```
        let twiddle_factor = butterfly_texture.read(stage, id.y).xy();
        let indices = butterfly_texture.read(stage, id.y).zw();
        let top_signal = pingpong0.read(id.x, indices.x);
        let bottom_signal = pingpong0.read(id.x, indices.y);
        output = top_signal + complex_mult(twiddle_factor, bottom_signal);
```

Listing 2: Vertical Butterfly Operation, over two sets of inputs

2.3.3. Permutation

Citations: [15]

Our data needs to be permuted, as per Section 1.6.7 our summation limits are offset compared to those used by the IFFT algorithm. The offset causes our data to flip signs in a gridlike pattern, as is visible in Figure 3.

```
if (id.x+ id.y) % 2 == 0.0 {
  sign = 1.0;
} else {
  sign = -1.0;
}
output = sign * pingpong0.read(id.x, id.y);
```

2.3.4. FFT Stage Visualisation

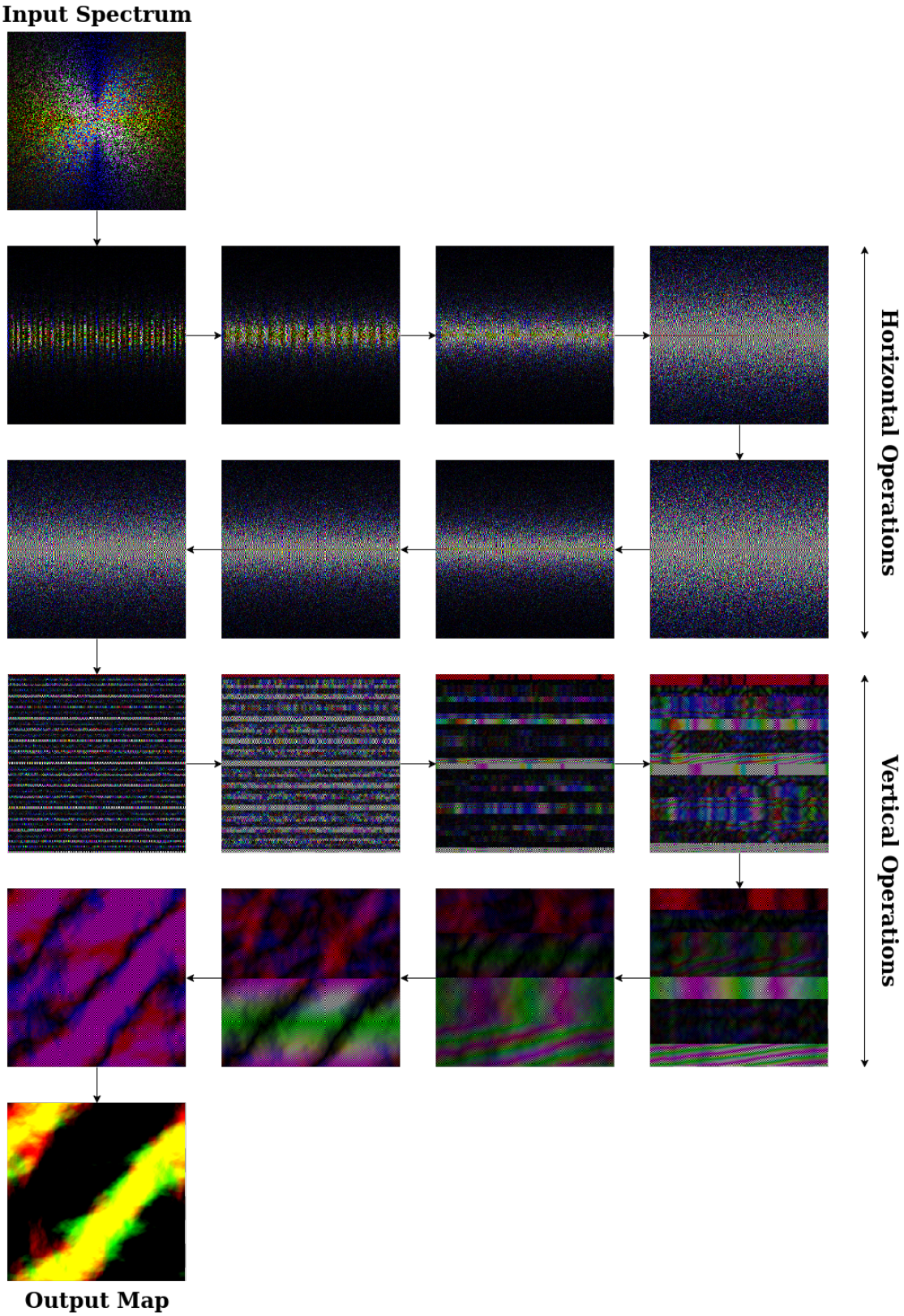

Figure 4: Visualised FFT Steps, black / white points are not consistent for demonstrative purposes.

2.4. Event Loop Control Flow

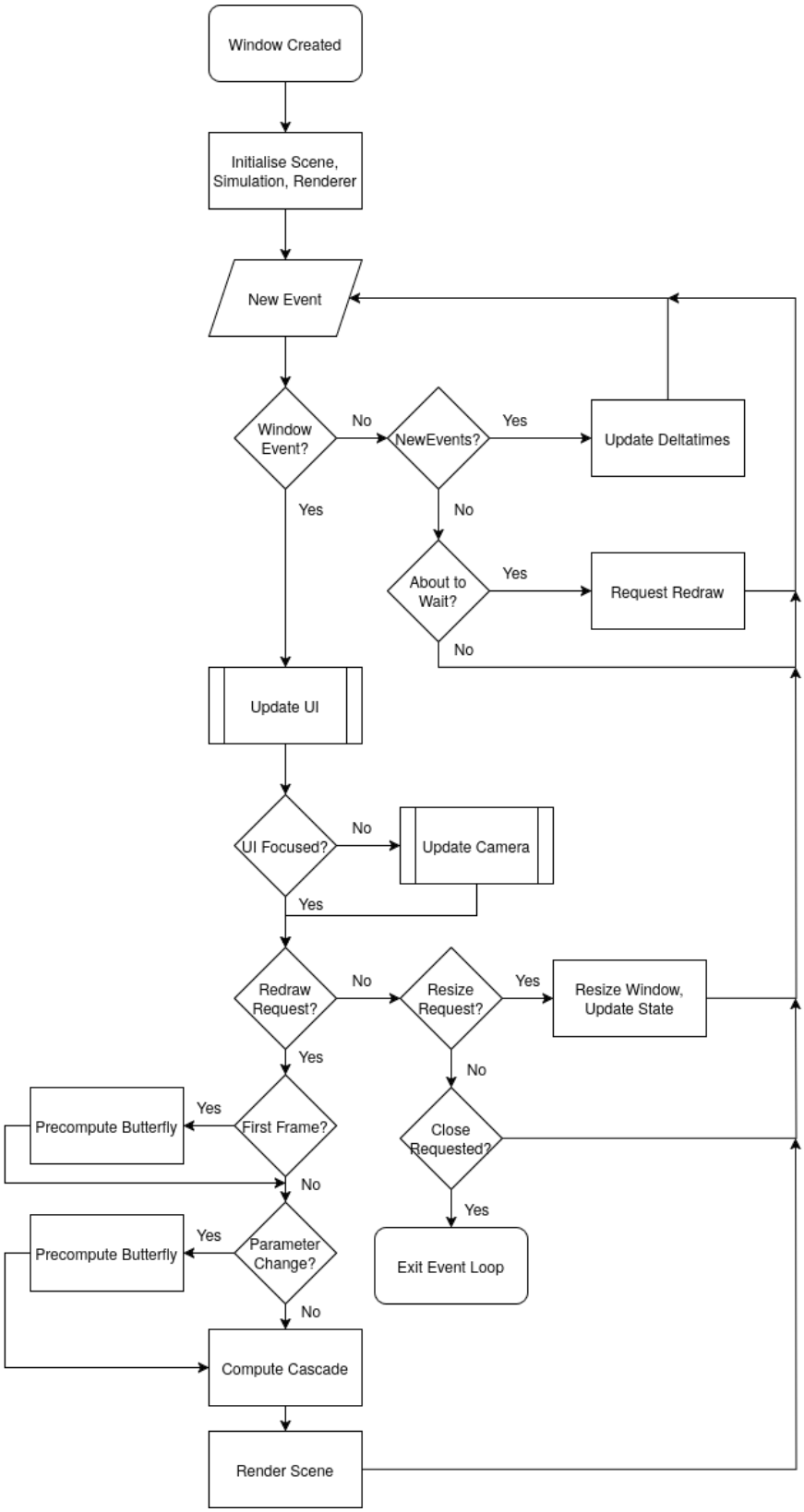

Figure 5: Abstracted event loop control flow flowchart

2.4.1. UI Event Flow

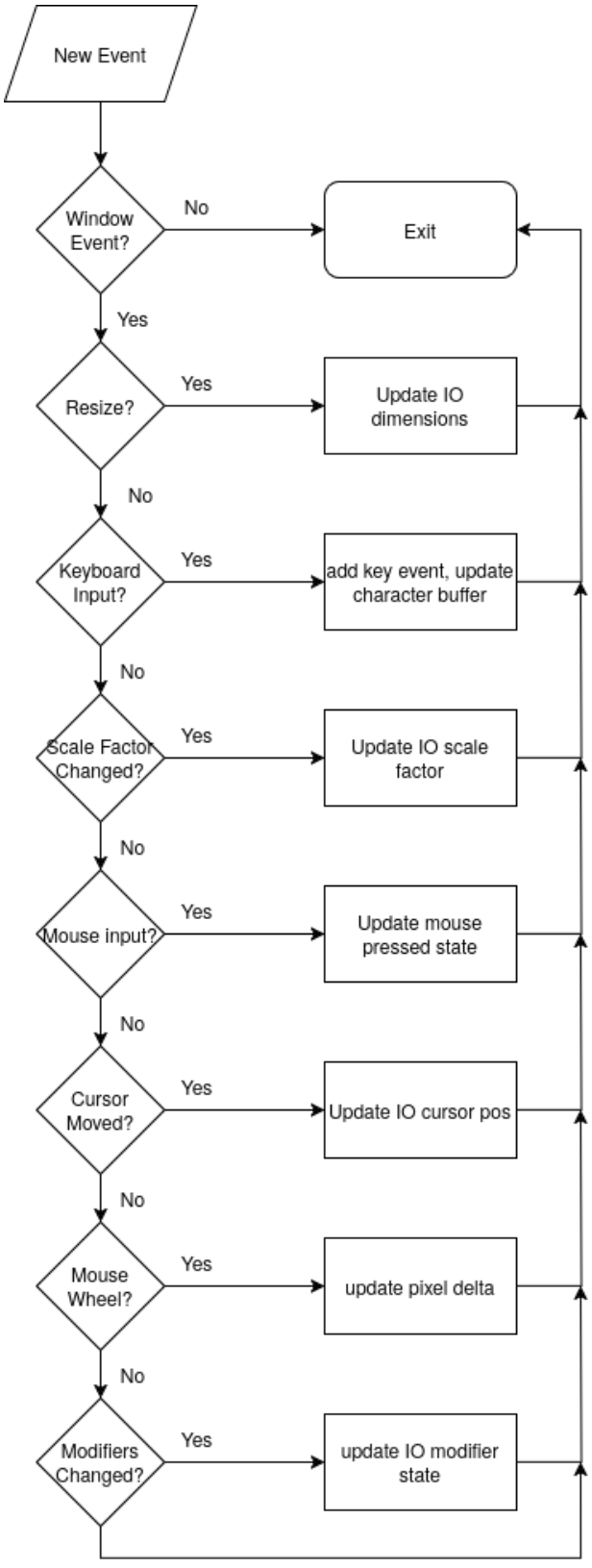

Figure 6: UI Event Handling Flowchart

2.4.2. Camera Controller Flow

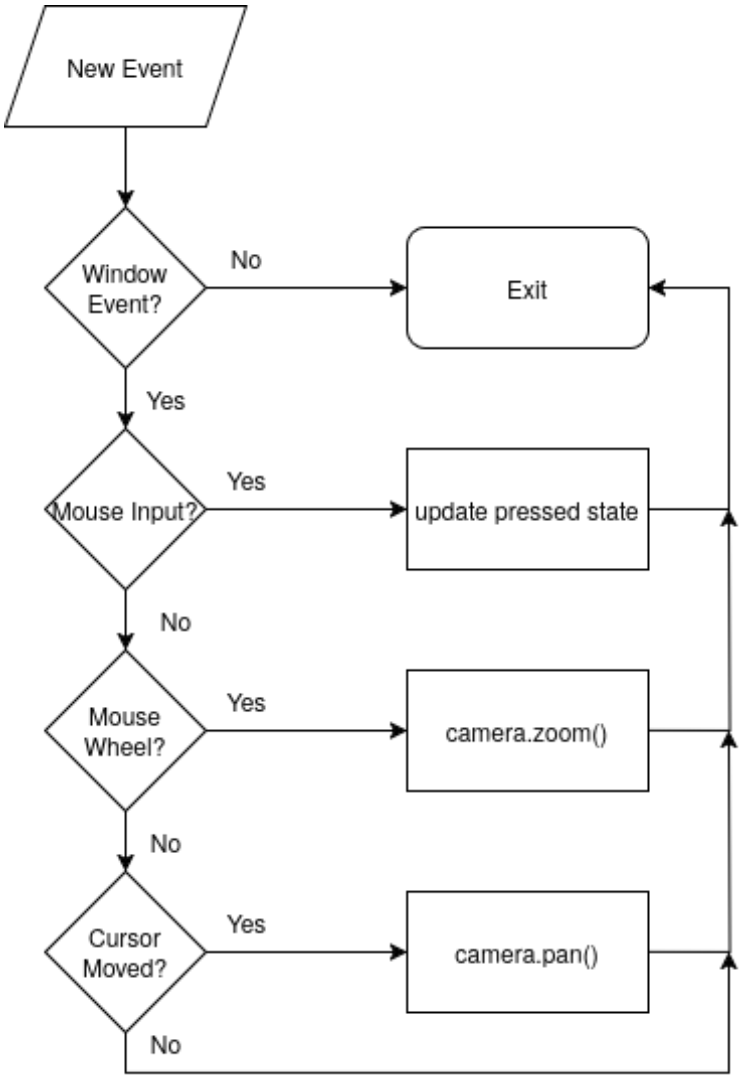

Figure 7: UI Event Handling Flowchart

2.5. Other Algorithms

2.5.1. Cascades

Citations: [7], [16]

To prevent any overlap and wasted computation, we choose lengthscales and cutoffs such that we only have to sample the spectrum at a given lengthscale only once. We select a larger lengthscale for bigger waves, and smaller lengthscale for smaller waves. For a visually pleasing ocean, we follow 3 key principles:

1 Avoid smaller waves in larger cascades as they have a course spatial resolution
2 Avoid larger waves in small cascades, as they will cause visible tiling
3 Cutoffs should be chosen such that there are no large gaps in spectrum coverage

Given this, we arrive at the following base cascade setup, altho there is room to change any of the parameters depending on the desired outcome.

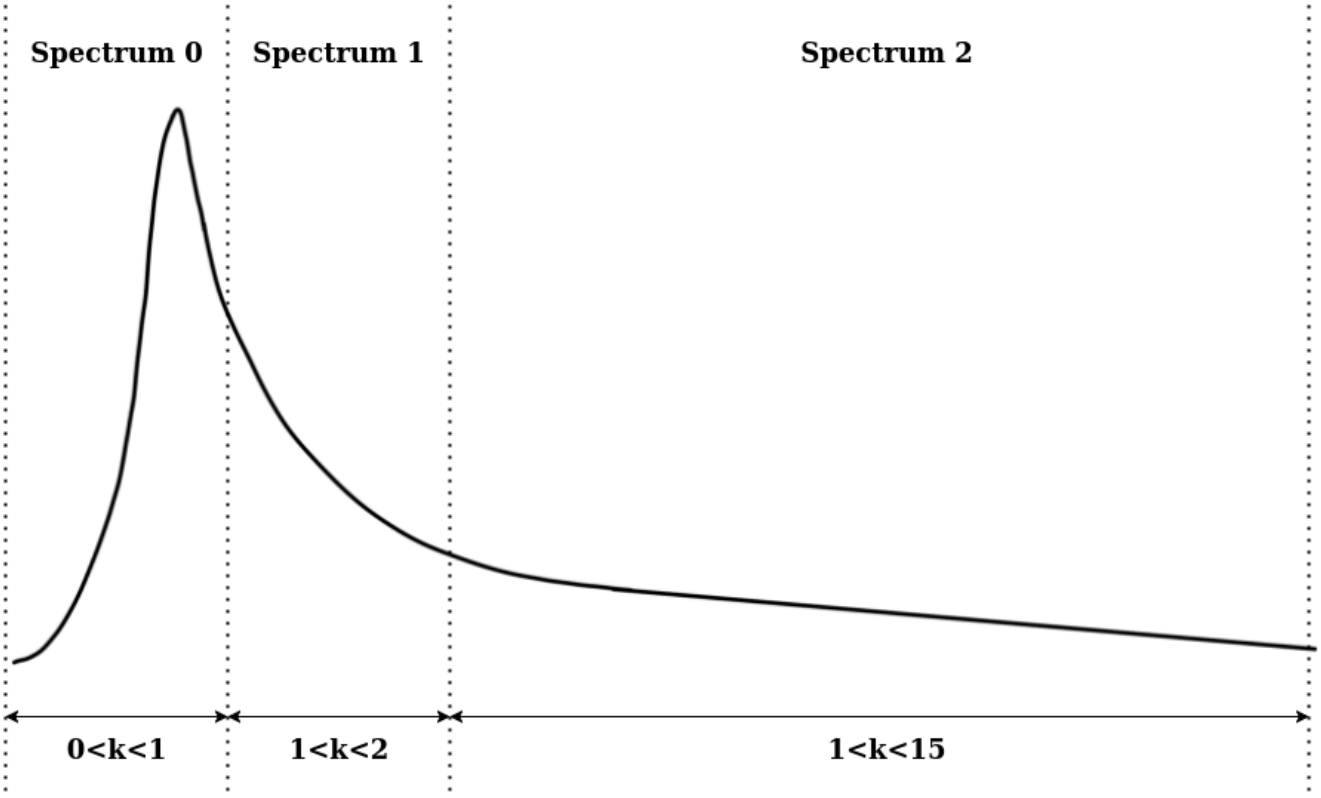

Figure 8: Graphical approximation of the spectrum for demonstrative purposes

2.5.2. Index Buffer

To enable proper backface culling, the vertices of the mesh are connected in a specific (counterclockwise) ordering such that the GPU can discern whether the face is pointed towards the camera. The GPU decides the ordering based on an index buffer, which specifies the index of which vertex is to be connected. The algorithm below only works for a square mesh, and was designed by me so is more than likely innefficient.

```
fn square_mesh_indices(size: u32) -> Vec<u32> {
  let mut indices: Vec<u32> = vec![];
  for y in 0..size - 1 {
      for x in 0..size - 1 {
          indices.push(x + y * size);
          indices.push((x + 1) + (y + 1) * size);
          indices.push(x + (y + 1) * size);
          indices.push(x + y * size);
          indices.push((x + 1) + y * size);
          indices.push((x + 1) + (y + 1) * size);
      }
  }
  indices
}
```

2.6. The Skybox & Equirectangular Projection

Citations: [17]

A standard skybox is rendered using a cubemap, which consists of 6 square textures that are projected onto a cube that is placed surrounding the scene. An alternative method is to use an equirectangular skybox, which is where a single 2D texture is sampled using a 3D vector which is transformed using polar coordinates. Ordinarily, cubemaps are chosen because they can be sampled directly, saving computation, however given that I would have to load the cubemap, pass it to the GPU and manually render the actual sky cube, the performance overhead is worth the significant ease of implementation.

Instead, we send a draw call of only 3 vertices to the GPU, which we then map to create a triangle large enough to cover the screen [17], which all skybox details are rendererd to. Below we offset the vertices based on their index to form the triangle in the $[0, 1]$ space, and then convert it to clip space $[-1, 1]$.

```
// Vertex Shader
let out_uv1 = Vec2::new(
    ((vertex_index << 1) & 2) as f32,
    (vertex_index & 2) as f32,
);
*out_pos = Vec4::new(out_uv1.x * 2.0 - 1.0,out_uv1.y * 2.0 - 1.0, 0.0, 1.0);
```

in the fragment shader, we take the 2D fragment coord and convert it to the same $[-1, 1]$ clip space, such that we can then inverse the camera view and projection affine transforms, converting the screen space 2 dimensional coordinate to a 3 dimensional world space coordinate.

```
// Fragment Shader
let uv = Vec2::new(
    frag_coord.x / consts.width * 2.0 - 1.0,
    1.0 - frag_coord.y / consts.height * 2.0,
);

let target = proj_inverse * Vec4::new(uv.x, uv.y, 1.0, 1.0);
let view_pos = (target.truncate() / target.w).extend(1.0);
let world_pos = view_inverse * view_pos;
let ray_dir = world_pos.truncate().normalize();
```

The 3D normalised direction vector is then used to sample the equirectangular HDRI texture, with the resulting value used to color the skybox.

```
// 3D direction vector -> 2D vector for texture reading
fn equirectangular_to_uv(v: Vec3) -> Vec2 {
    Vec2::new(
        (v.z.atan2(v.x) + consts::PI) / consts::TAU,
        v.y.acos() / consts::PI,
    )
}
```

2.7. Spectrum Conjugates

Citations: [4], [7], [16]

From Tessendorf [4], we need to compute the spectrum and its conjugate in order to ensure a real output per Section 1.6.3. “The symmetry of the fourier series allows us to mirror the amplitudes,eliminiating the need for complex conjugate recalculation” [16]. Below is pseudocode of the concept from [16] / [7], where N corresponds to the simulation resolution.

```
// The Computed Spectrum
let h0 = spectrum_tex.read(id.x, id.y);
// The Conjugate Spectrum
let h0c = spectrum_tex.read(
  (N - id.x) % N,
  (N - id.y) % N,
);
```

3. Technical Solution

```rust
let example = skibidi;
```

4. Testing

4.1. Testing Strategy

4.2. Testing Table

5. Evaluation

5.1. Evaluation Against Criteria

5.2. Client Feedback

5.3. Evaluation of Feedback

6. Bibliography

[1] Acerola, *How Games Fake Water*. Accessed: Sep. 13, 2024. [OnlineVideo]. Available: https://youtu.be/PH9q0HNBjT4

[2] Wikipedia, "Blinn–Phong reflection model." Accessed: Sep. 11, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Blinn-Phong_reflection_model

[3] Wikipedia, "Schlick's Approximation." Accessed: Sep. 10, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Schlick's_approximation

[4] Jerry Tessendorf, "Simulating Ocean Water." Accessed: Sep. 08, 2024. [Online]. Available: https://people.computing.clemson.edu/~jtessen/reports/papers_files/coursenotes2004.pdf

[5] Christopher J. Horvath, "Empirical directional wave spectra for computer graphics." Accessed: Oct. 20, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/2791261.2791267

[6] wikiwaves, "Ocean-wave Spectra." Accessed: Sep. 08, 2024. [Online]. Available: https://wikiwaves.org/Ocean-Wave_Spectra

[7] Jump Trajectory, *Ocean waves simulation with Fast Fourier transform*. Accessed: Sep. 13, 2024. [OnlineVideo]. Available: https://youtu.be/kGEqaX4Y4bQ

[8] Acerola, *I Tried Simulating The Entire Ocean*. Accessed: Sep. 08, 2024. [OnlineVideo]. Available: https://www.youtube.com/watch?v=yPfagLeUa7k

[9] "Ocean simulation part one:using the discrete fourier Fourier transform." Accessed: Sep. 27, 2024. [Online]. Available: https://www.keithlantz.net/2011/10/ocean-simulation-part-one-using-the-discrete-fourier-transform/

[10] math et al, *Box-Muller Transform + R Demo*. Accessed: Jan. 04, 2025. [OnlineVideo]. Available: https://www.youtube.com/watch?v=T6Oay7g_Ik8

[11] Mark Mihelich and Tim Tcheblokov, "Wakes, Explosions and Lighting:Interactive Water Simulation in Atlas." Accessed: Sep. 13, 2024. [Online]. Available: https://www.youtube.com/watch?v=Dqld965-Vv0

[12] Jakub Bokansky, "Crash Course in BRDF Implementation." Accessed: Sep. 17, 2024. [Online]. Available: https://boksajak.github.io/files/CrashCourseBRDF.pdf

[13] Learn OpenGL, "HDR." Accessed: Mar. 02, 2025. [Online]. Available: https://learnopengl.com/Advanced-Lighting/HDR

[14] Dynamic Mathematics, "Strange Attractors." Accessed: Jun. 14, 2024. [Online]. Available: https://www.dynamicmath.xyz/strange-attractors/

[15] F.-J. Flügge, "Realtime GPGPU FFT ocean water simulation." Accessed: Jan. 14, 2025. [Online]. Available: http://tubdok.tub.tuhh.de/handle/11420/1439

[16] Saulius Vincevičius, "Realistic Ocean Simulation using Fourier Transform." Accessed: Oct. 20, 2024. [Online]. Available: https://github.com/Biebras/Ocean-Simulation-Unity

[17] Chris k. Wallis, “Optimizing Triangles for a Full-screen Pass.” Accessed: Mar. 02, 2025. [Online]. Available: https://wallisc.github.io/rendering/2021/04/18/Fullscreen-Pass.html

[18] Polyhaven, “Kloofendal 43d Clear (Pure Sky).” Accessed: Mar. 12, 2025. [Online]. Available: https://polyhaven.com/a/kloofendal_43d_clear_puresky

[19] rebelot, “kanagawa.nvim.” Accessed: Mar. 12, 2025. [Online]. Available: https://github.com/rebelot/kanagawa.nvim/blob/master/extras/tmTheme/kanagawa.tmTheme

[20] sotrh, “A Perspective Camera.” Accessed: Oct. 20, 2024. [Online]. Available: https://sotrh.github.io/learn-wgpu/beginner/tutorial6-uniforms/