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:

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

$$I = \begin{cases} I_0e^{-\zeta} + J_{\text{biased}} \text{ if } J_{\text{biased}} > \mu \\ I_0e^{-\zeta} \text{ if } J_{\text{biased}} < \mu \end{cases}$$

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.

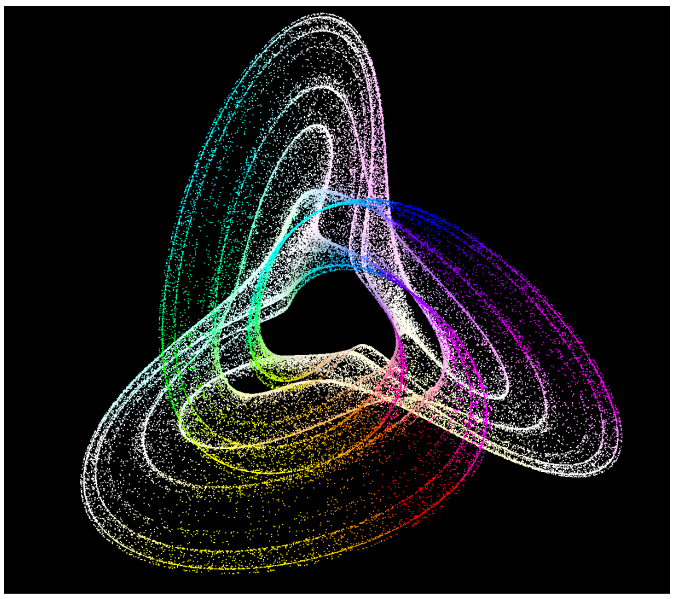


Figure 1: Shader Output, Found at https://github.com/CmrCrabs/chaotic-attractors

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

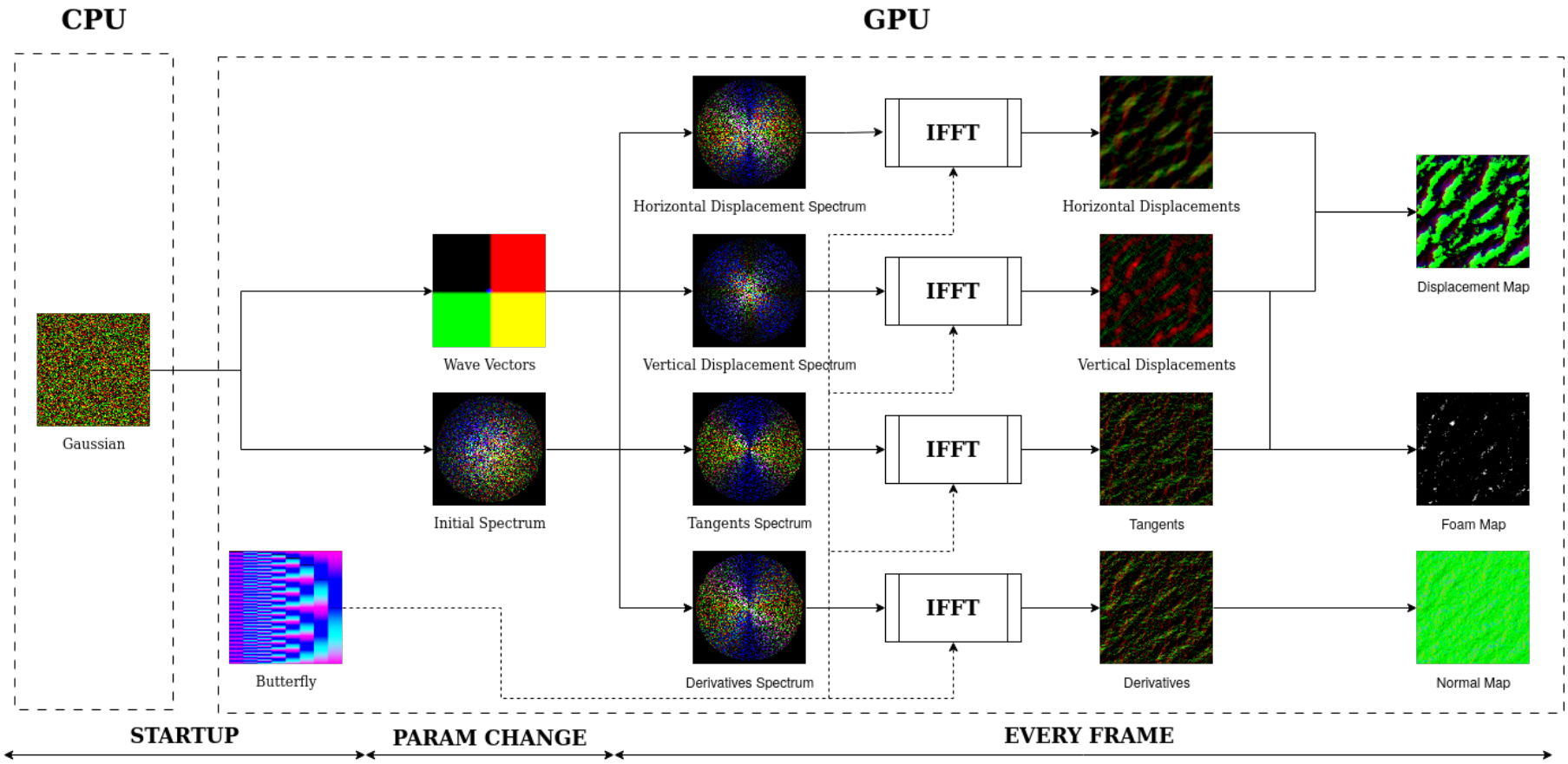


Figure 2: Visualisation of the core algorithm. Texture white / black points are not consistent for ease of demonstration

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();
```

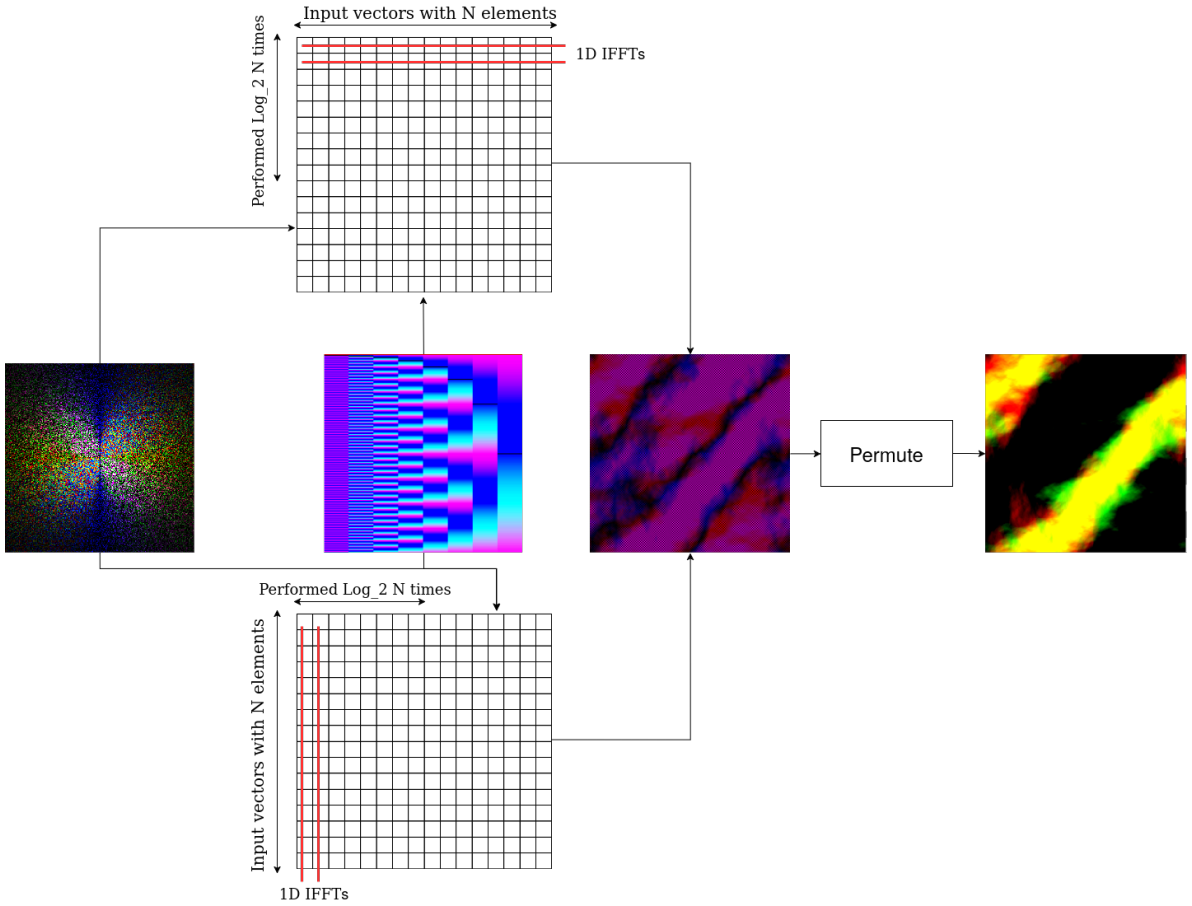


Figure 3: Visualised IFFT Algorithm showing the horizontal and vertical operations on an input texture

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

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

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

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

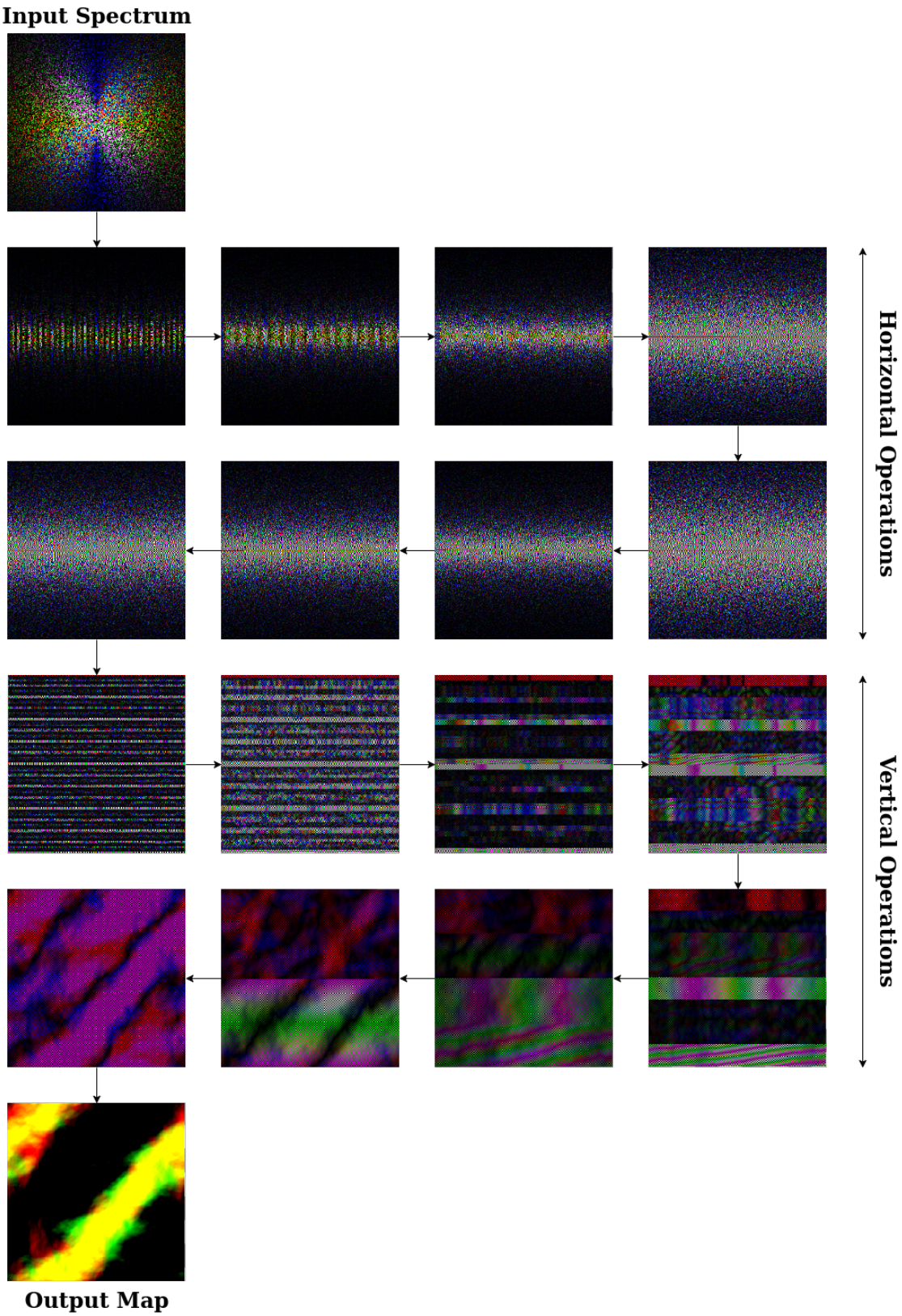


Figure 4: Visualised FFT Steps, black / white points are not consistent for demonstrative purposes.

2.4. Event Loop Control Flow

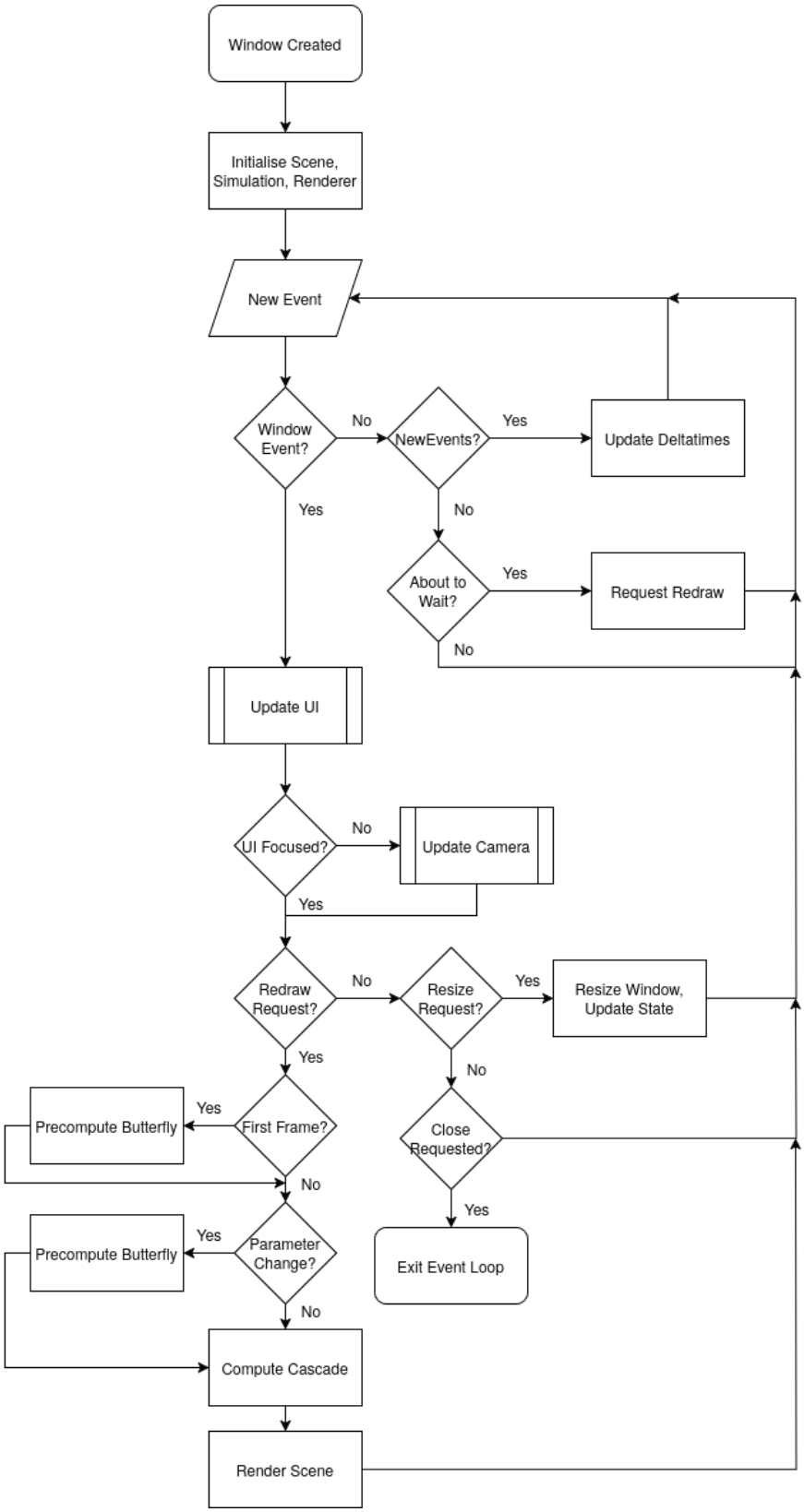


Figure 5: Abstracted event loop control flow flowchart

2.4.1. UI Event Flow

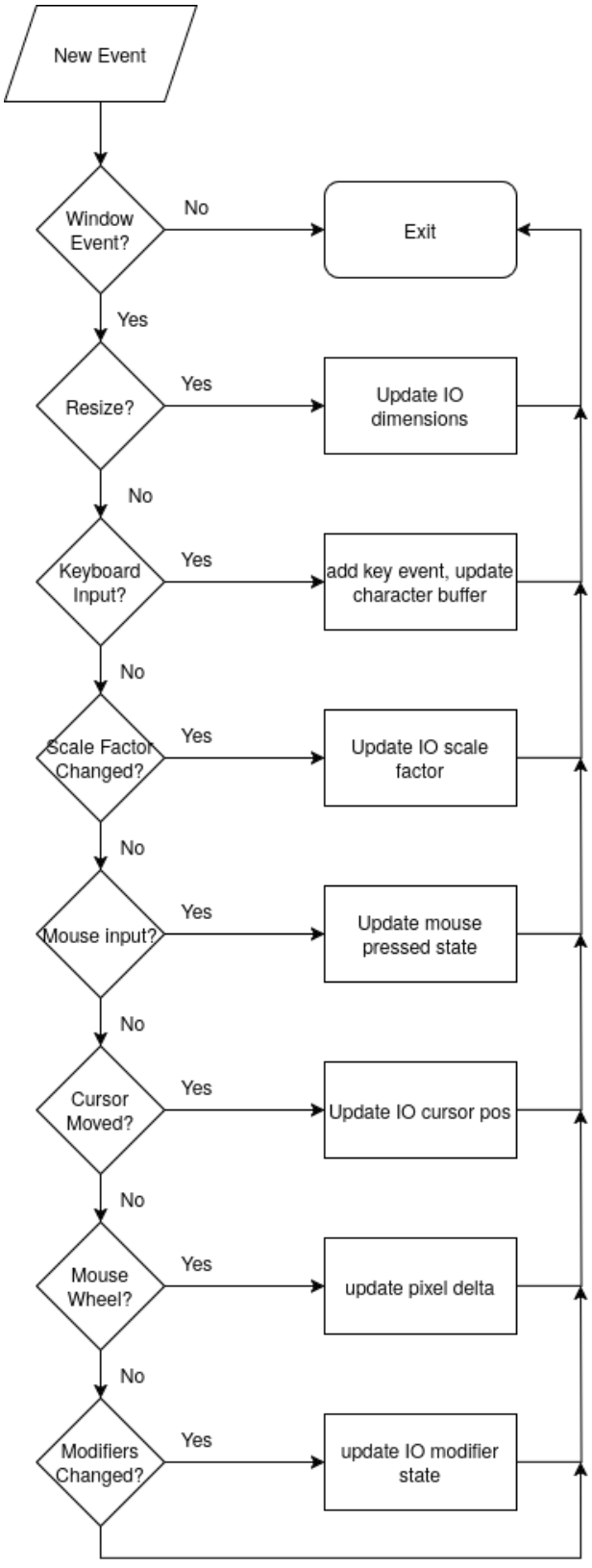


Figure 6: UI Event Handling Flowchart

2.4.2. Camera Controller Flow

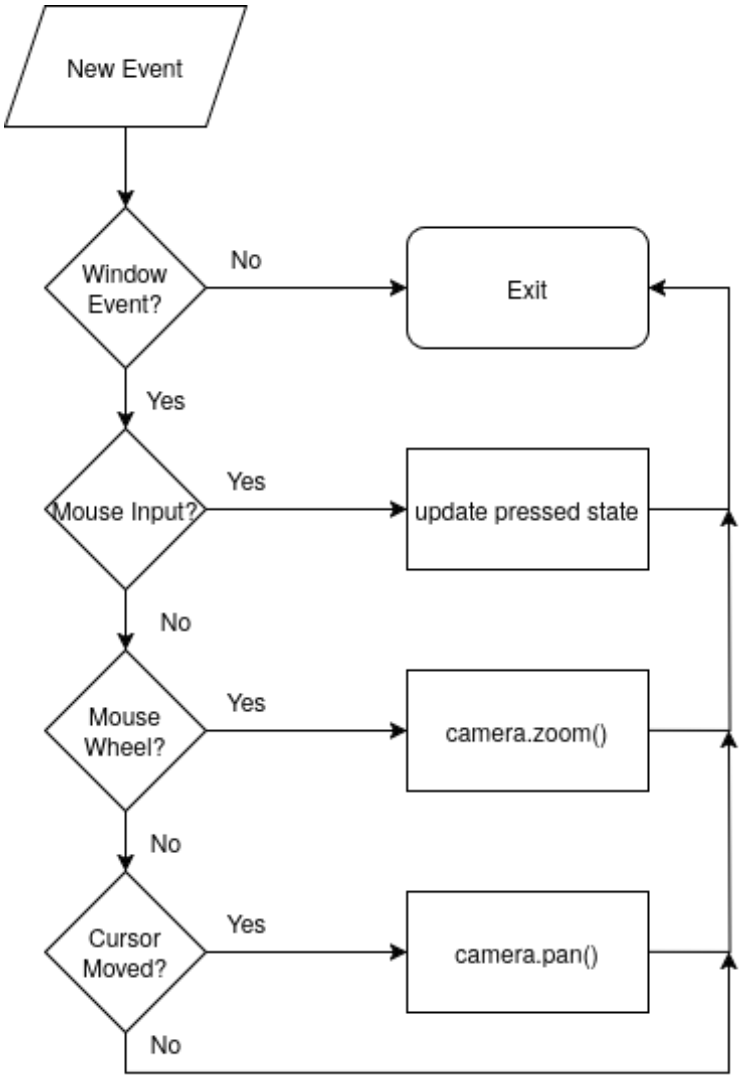


Figure 7: UI Event Handling Flowchart

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.

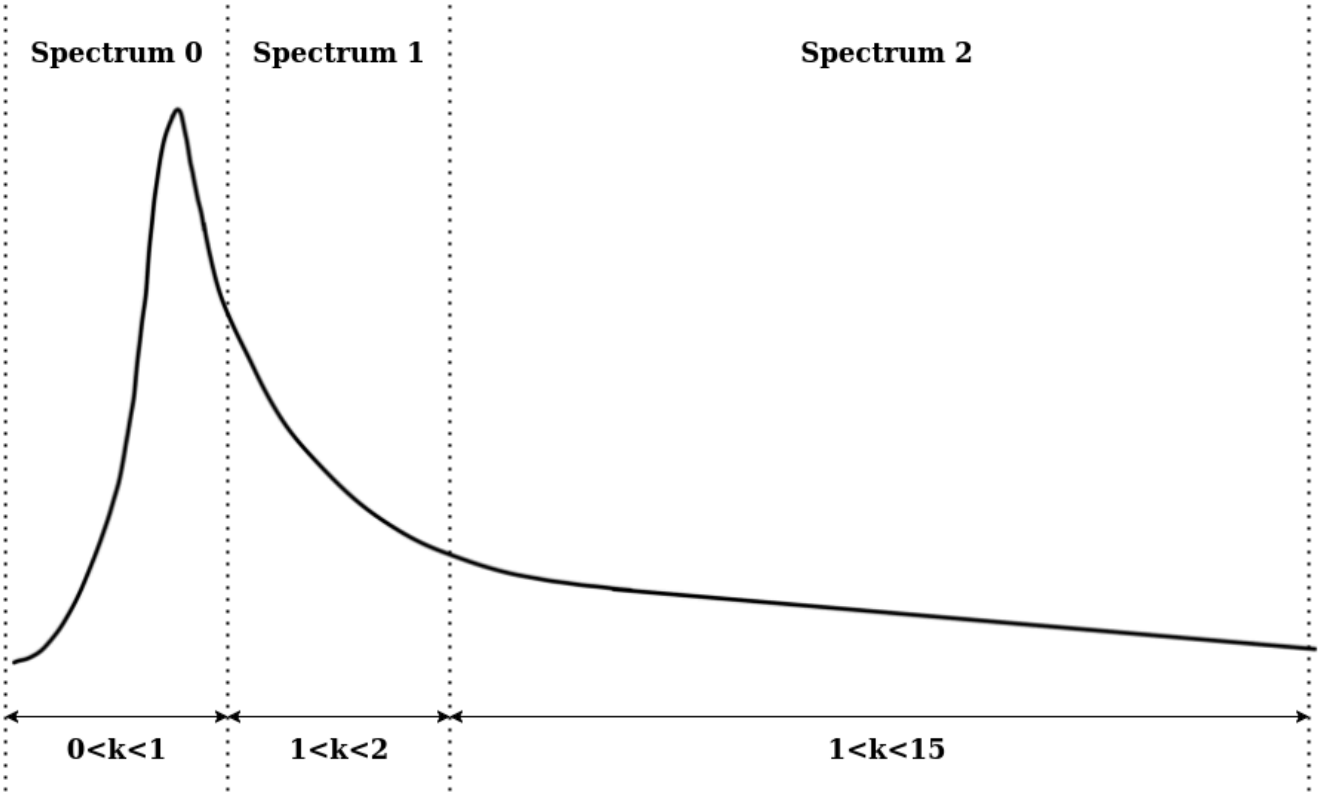


Figure 8: Graphical approximation of the spectrum for demonstrative purposes

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,
);
```

2.8. Xoshiro256plus Pseudorandom Number Generator

Citations: [18]

In order to have consistency between different instances of the simulation, we seed the gaussian numbers such that there is reproducability. To do so, I have chosen the Xoshiro256plus PRNG as it operates in O(1) in both space & time complexity, runs in nanoseconds and is relatively simple to implement. Below is pseudocode showcasing the method.

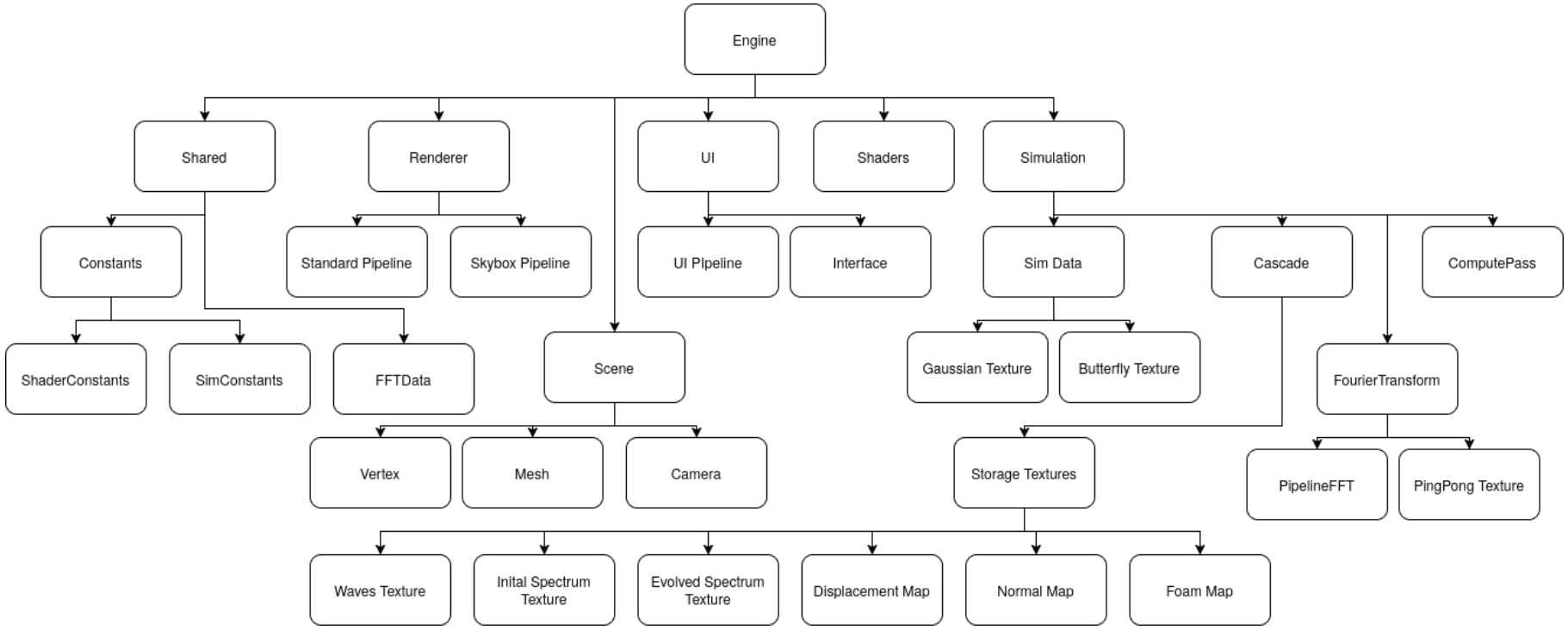
```
fn next(&mut self) -> u64 {
    let result = self.seed[0].wrapping_add(self.seed[3]);
    let t = self.s[1] << 17;

    self.seed[2] ^= self.seed[0];
    self.seed[3] ^= self.seed[1];
    self.seed[1] ^= self.seed[2];
    self.seed[0] ^= self.seed[3];

    self.seed[2] ^= t;
    self.seed[3] = rol64(self.seed[3], 45);

    result
}
```

2.9. Code Structure



,

Figure 9: Projects Data structure, in terms of “key” data types - structs, pipelines & textures

3. Technical Solution

3.1. Skills Demonstrated

I believe the GPU / CPU can be seen as an equivalent model to Client / Server. This is because they follow similar key concepts, being:

- Task Delegation, the CPU sends tasks to the GPU in a similar way to server processing requests from clients
- Concurrency / Parallellism, executing and managing data from many GPU threads in parallel is similar to a server handling multiple clients, as both must handle resource allocation and data synchronization
- Communication Overhead, there are similar data transfer bottlenecks for both client/server & gpu/cpu, as I have had to consider data alignment, padding and typing when passing data to the gpu, as well as synchronizing the updating of data between passes and individual invocations
- Synchronization, I have to manage updating data throughout an individual invocation through push constants in a similar manner to how a server would have to update data on a client

Group	Skill	Description / Reasoning	Evidence
A	Complex Scientific Model	Entire Spectrum Synthesis	`shaders/sim/initial_spectra.rs` entire file
A	Complex Mathematical Model	Implementation of a PBR Lighting Model	`shaders/lib.rs` entire file
A	Complex Mathematical Model	The computing, storing and processing of a large amount of complex numbers	`shaders/evolve_spectra.rs`, `shaders/process_deltas.rs` entire file(s)
A	Complex Control Model	The entire applications event loop and resultitng control flow	`engine/mod.rs` line 92 onwards
A	List Operations	Generation of gaussian texture data, generation of index buffer, manipulation of indices for use in fft	`shaders/fft.rs`, `sim/simdata.rs`, `engine/scene.rs`
A	Hashing	Xoshiro256++ PRNG Implementation	`sim/simdata.rs` line 83 onwards

A	Advanced Matrix Operations	Various screen / world space transformations, reversing affine transformations	`shaders/skybox.rs` fragment shader, `shaders/lib.rs` vertex shader
A	Recursive Algorithms	the 2D GPGPU IFFT I have manually recurses, due to it being on the GPU	`sim/fft.rs` `66-120`, should be clearer after seeing `shaders/fft.rs`
A	Complex User Defined Algorithms	Computation of the index buffer for the mesh drawing	`engine/scene.rs` lines `181-191`
A	Complex OOP model	Program is based around different objects and classes, objects are generated / regenerated based on user input, designed with composition and inheritence in mind	most demonstrated in `engine/mod.rs` entire file
A	Complex client-server model	explained in preamble	`engine/mod.rs` `run()` function, `sim/compute.rs`, `engine/renderer.rs`, `sim/fft.rs`, `sim/mod.rs` whole file(s)
B	Multi-Dimensional Arrays	Usage of textures throughout entire project (analagous to 2D arrays), managing data between FFT passes	`shaders/*`
B	Simple Mathematical Model	Box-Muller Transform, gaussian random number generation	`sim/simdata.rs` line `62` function(s)

3.2. Completeness of Solution

Everything specified in objectives has been completed, the performance has not been tested on a gtx 1060 level GPU, but given that it is performant at 40-50fps on an Intel Iris Xe integrated graphics card at 256x256x3, it will be easily performant on a discrete GPU. At a more reasonable (given the system) resolution of 128x128x3, the simulation easily reaches 90+fps, while still looking reasonably good. From the additional features, swell has been included in the spectrum synthesis.

3.3. Coding Style

I have followed the rust programming conventions and principles for this project, using the result type for error handling where possible. As the vast majority of possible errors come from wgpu/rust-gpu, errors are in majority handled via the pollster and env_logger crates, where any errors comign from wgpu / rust-gpu are passed up the chain and outputted from the standard log using env_logger. Shader code is written following “standard” shader conventions, with notable stylings being an emphasis on defining variables for every interim step, and swizzling vectors / avoiding unncessary abstractions where convenient. Smaller things like manually computing the power where convenient and “pre”computing certain values into variables and reusing them where possible are also done.

3.4. Source Code

3.4.1. main.rs

rust

```rust
use engine::Engine;
use log::LevelFilter;
use std::{mem, slice};
use winit::{event_loop::EventLoop, window::WindowBuilder};

pub mod engine;
pub mod sim;

pub type Result<T = (), E = Box<dyn std::error::Error>> = std::result::Result<T, E>;

pub const FORMAT: wgpu::TextureFormat = wgpu::TextureFormat::Bgra8UnormSrgb;
pub const DEPTH_FORMAT: wgpu::TextureFormat = wgpu::TextureFormat::Depth32Float;
pub const WG_SIZE: u32 = 8;

fn main() -> Result {
    env_logger::builder().filter_level(LevelFilter::Info).init();
    // Forces application to run in XWayland on Linux, allows graphics debugger to run
    std::env::remove_var("WAYLAND_DISPLAY");
    let event_loop = EventLoop::new()?;
    let window = WindowBuilder::new().with_title("NEA").build(&event_loop)?;

    let mut engine = Engine::new(&window);

    engine.run(event_loop)?;
    Ok(())
}

pub fn cast_slice<T>(x: &[T]) -> &[u8] {
    unsafe { slice::from_raw_parts(x.as_ptr() as _, mem::size_of_val(x)) }
}

```

3.4.2. engine

3.4.2.1. mod.rs

```rust
use super::{FORMAT, WG_SIZE};
use crate::{cast_slice, Result};
use {
    renderer::Renderer,
    scene::{Mesh, Scene},
    crate::sim::Simulation,
    ui::UI,
};
use winit::event::{Event, WindowEvent};
use winit::event_loop::EventLoop;
use winit::keyboard::{KeyCode, PhysicalKey};

pub mod renderer;
pub mod scene;
pub mod ui;
pub mod util;

pub struct Engine<'a> {
    pub device: wgpu::Device,
    pub queue: wgpu::Queue,
    pub config: wgpu::SurfaceConfiguration,
    pub window: &'a winit::window::Window,
    pub surface: wgpu::Surface<'a>,
    pub simulation: Simulation,
    pub renderer: Renderer,
    pub scene: Scene,
    pub ui: UI,
}

impl<'a> Engine<'a> {
    pub fn new(window: &'a winit::window::Window) -> Self {
        let instance = wgpu::Instance::new(wgpu::InstanceDescriptor::default());
        let surface = instance.create_surface(window).unwrap();
        let adapter = pollster::block_on(instance.request_adapter(&wgpu::RequestAdapterOptions {
            power_preference: wgpu::PowerPreference::default(),
            force_fallback_adapter: false,
            compatible_surface: Some(&surface),
        }))
        .expect("failed to create adapter");

        let required_limits = wgpu::Limits {
            max_storage_textures_per_shader_stage: 6,
            max_bind_groups: 6,
            max_push_constant_size: 8,
            ..Default::default()
        };
        let (device, queue) = pollster::block_on(adapter.request_device(
            &wgpu::DeviceDescriptor {
                required_features: wgpu::Features::TEXTURE_ADAPTER_SPECIFIC_FORMAT_FEATURES
                    | wgpu::Features::VERTEX_WRITABLE_STORAGE
                    | wgpu::Features::PUSH_CONSTANTS,
                required_limits,
                memory_hints: wgpu::MemoryHints::Performance,
                label: None,
```

```
55              },
56              None,
57          ))
58          .expect("failed to create device & queue");
59
60          let shader = device.create_shader_module(wgpu::include_spirv!(env!("shaders.spv")));
61
62          let config = wgpu::SurfaceConfiguration {
63              usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
64              format: FORMAT,
65              width: window.inner_size().width,
66              height: window.inner_size().height,
67              present_mode: wgpu::PresentMode::AutoNoVsync,
68              alpha_mode: wgpu::CompositeAlphaMode::Opaque,
69              view_formats: vec![],
70              desired_maximum_frame_latency: 2,
71          };
72          surface.configure(&device, &config);
73
74          let scene = Scene::new(&device, window);
75          let simulation = Simulation::new(&device, &queue, &shader, &scene);
76          let renderer = Renderer::new(&device, &queue, &shader, window, &simulation, &scene);
77          let ui = UI::new(&device, &queue, window, &shader, &renderer, &scene);
78
79          Self {
80              config,
81              device,
82              queue,
83              surface,
84              window,
85              simulation,
86              scene,
87              renderer,
88              ui,
89          }
90      }
91
92      pub fn run(&mut self, event_loop: EventLoop<()>) -> Result {
93          let mut last_frame = std::time::Instant::now();
94          let mut first_frame = true;
95          let workgroup_size = self.scene.consts.sim.size / WG_SIZE;
96
97          event_loop.run(move |event, elwt| match event {
98              Event::AboutToWait => self.window.request_redraw(),
99              Event::NewEvents(_) => {
100                 let now = std::time::Instant::now();
101                 self.ui.context.io_mut().update_delta_time(now - last_frame);
102                 last_frame = now;
103             }
104             Event::WindowEvent { event, .. } => {
105                 self.ui.handle_events(&event);
106                 if !self.ui.focused {
107                     self.scene.update_camera(&event, self.window);
108                 }
109                 match event {
110                     WindowEvent::RedrawRequested => {
111                         self.scene.update_redraw(self.window);
112
```

```
                        let surface = self
                            .surface
                            .get_current_texture()
                            .expect("failed to get surface");
                        let surface_view = surface
                            .texture
                            .create_view(&wgpu::TextureViewDescriptor::default());
                        let mut encoder = self
                            .device
                            .create_command_encoder(&wgpu::CommandEncoderDescriptor::default());

                        if first_frame {
                            self.simulation.butterfly_precompute_pass.compute(
                                &mut encoder,
                                "Precompute Butterfly",
                                &[
                                    &self.scene.consts_bind_group,
                                    &self.simulation.simdata.bind_group,
                                ],
                                self.scene.consts.sim.size.ilog2(),
                                self.scene.consts.sim.size / WG_SIZE,
                            );
                        }

                        // Compute Initial spectrum on param change
                        if self.scene.consts_changed {
                            self.scene.write(&self.queue);
                            self.simulation.compute_initial(
                                &mut encoder,
                                &[
                                    &self.scene.consts_bind_group,
                                    &self.simulation.simdata.bind_group,
                                    &self.simulation.cascade0.bind_group,
                                ],
                                0,
                                workgroup_size,
                                workgroup_size,
                            );
                            self.simulation.compute_initial(
                                &mut encoder,
                                &[
                                    &self.scene.consts_bind_group,
                                    &self.simulation.simdata.bind_group,
                                    &self.simulation.cascade1.bind_group,
                                ],
                                1,
                                workgroup_size,
                                workgroup_size,
                            );
                            self.simulation.compute_initial(
                                &mut encoder,
                                &[
                                    &self.scene.consts_bind_group,
                                    &self.simulation.simdata.bind_group,
                                    &self.simulation.cascade2.bind_group,
                                ],
                                2,
                                workgroup_size,
```

```
171                                 workgroup_size,
172                             );
173                             // updates mesh based on mesh_step input, technically redundant to do
174                             // on every param change but not an issue in any practical sense
175                             self.scene.mesh = Mesh::new(&self.device, &self.scene.consts);
176                         }
177 
178                         // per frame computation
179                         self.simulation.compute_cascade(&mut encoder, &self.simulation.cascade0,
    &mut self.scene, workgroup_size, 0);
180                         self.simulation.compute_cascade(&mut encoder, &self.simulation.cascade1,
    &mut self.scene, workgroup_size, 1);
181                         self.simulation.compute_cascade(&mut encoder, &self.simulation.cascade2,
    &mut self.scene, workgroup_size, 2);
182 
183                         // Render Skybox
184                         self.renderer
185                             .render_skybox(&mut encoder, &surface_view, &self.scene);
186 
187                         // Standard Render Pass
188                         self.queue.write_buffer(
189                             &self.scene.consts_buf,
190                             0,
191                             cast_slice(&[self.scene.consts]),
192                         );
193                         self.renderer.render_standard(
194                             &mut encoder,
195                             &self.renderer.std_pipeline,
196                             &[
197                                 &self.scene.consts_bind_group,
198                                 &self.renderer.sampler_bind_group,
199                                 &self.renderer.hdri.bind_group,
200                                 &self.simulation.cascade0.bind_group,
201                                 &self.simulation.cascade1.bind_group,
202                                 &self.simulation.cascade2.bind_group,
203                             ],
204                             &surface_view,
205                             &self.scene.mesh,
206                             self.scene.consts.sim.instances,
207                         );
208 
209                         // UI Pass
210                         let consts_copy = self.scene.consts;
211                         self.ui.update_cursor(self.window);
212                         let ui_frame = self.ui.context.frame();
213                         self.ui.focused = ui::build(ui_frame, &mut self.scene.consts);
214                         self.ui.render(
215                             &self.device,
216                             &self.queue,
217                             &mut encoder,
218                             &surface_view,
219                             &self.renderer.sampler_bind_group,
220                             &self.scene,
221                         );
222 
223                         // updating some rendering logic
224                         self.scene.consts_changed = consts_copy != self.scene.consts;
225                         first_frame = false;
```

```
226
227                         // Submitting queue to be computed
228                         self.queue.submit([encoder.finish()]);
229                         surface.present();
230                     }
231                     WindowEvent::Resized(size) => {
232                         self.config = wgpu::SurfaceConfiguration {
233                             usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
234                             format: FORMAT,
235                             width: size.width,
236                             height: size.height,
237                             present_mode: wgpu::PresentMode::AutoNoVsync,
238                             alpha_mode: wgpu::CompositeAlphaMode::Opaque,
239                             view_formats: vec![],
240                             desired_maximum_frame_latency: 2,
241                         };
242                         self.surface.configure(&self.device, &self.config);
243                         self.renderer.new_depth_view(&self.device, self.window);
244
245                         self.scene.camera.update_fov(self.window);
246                         self.scene.consts.camera_viewproj =
247                             self.scene.camera.proj * self.scene.camera.view;
248                     }
249                     WindowEvent::CloseRequested => elwt.exit(),
250                     WindowEvent::KeyboardInput { event, .. } => if let
    PhysicalKey::Code(KeyCode::Escape) = event.physical_key {
251                         elwt.exit()
252                     },
253                     _ => {}
254                 }
255             }
256             _ => {}
257         })?;
258         Ok(())
259     }
260 }
261
```

3.4.2.2. renderer.rs

```rust
use super::util::Texture;
use crate::{DEPTH_FORMAT, FORMAT};
use super::scene::{Mesh, Scene};
use super::Simulation;

pub struct Renderer {
    pub sampler_bind_group: wgpu::BindGroup,
    pub sampler_layout: wgpu::BindGroupLayout,
    pub depth_view: wgpu::TextureView,
    pub std_pipeline: wgpu::RenderPipeline,
    pub hdri: Texture,
    pub skybox_pipeline: wgpu::RenderPipeline,
}

impl Renderer {
    pub fn new(
        device: &wgpu::Device,
        queue: &wgpu::Queue,
        shader: &wgpu::ShaderModule,
        window: &winit::window::Window,
        sim: &Simulation,
        scene: &super::scene::Scene,
    ) -> Self {
        let sampler = device.create_sampler(&wgpu::SamplerDescriptor::default());
        let sampler_layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
            entries: &[wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX_FRAGMENT,
                ty: wgpu::BindingType::Sampler(wgpu::SamplerBindingType::NonFiltering),
                count: None,
            }],
            label: None,
        });
        let sampler_bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
            layout: &sampler_layout,
            entries: &[wgpu::BindGroupEntry {
                binding: 0,
                resource: wgpu::BindingResource::Sampler(&sampler),
            }],
            label: None,
        });

        let hdri = Texture::from_file(device, queue, "HDRI", "./assets/industrial.exr");

        let depth_texture = device.create_texture(&wgpu::TextureDescriptor {
            label: None,
            size: wgpu::Extent3d {
                width: window.inner_size().width,
                height: window.inner_size().height,
                depth_or_array_layers: 1,
            },
            mip_level_count: 1,
            sample_count: 1,
            dimension: wgpu::TextureDimension::D2,
            format: DEPTH_FORMAT,
            usage: wgpu::TextureUsages::RENDER_ATTACHMENT | wgpu::TextureUsages::TEXTURE_BINDING,
```

```
            view_formats: &[],
        });
        let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

        let skybox_pipeline_layout =
            device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
                bind_group_layouts: &[&scene.consts_layout, &hdri.layout, &sampler_layout],
                push_constant_ranges: &[],
                label: None,
            });
        let skybox_pipeline = device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
            layout: Some(&skybox_pipeline_layout),
            vertex: wgpu::VertexState {
                module: shader,
                entry_point: Some("skybox::skybox_vs"),
                buffers: &[],
                compilation_options: Default::default(),
            },
            fragment: Some(wgpu::FragmentState {
                module: shader,
                entry_point: Some("skybox::skybox_fs"),
                targets: &[Some(wgpu::ColorTargetState {
                    format: FORMAT,
                    blend: Some(wgpu::BlendState::ALPHA_BLENDING),
                    write_mask: wgpu::ColorWrites::ALL,
                })],
                compilation_options: Default::default(),
            }),
            primitive: wgpu::PrimitiveState::default(),
            depth_stencil: Some(wgpu::DepthStencilState {
                format: DEPTH_FORMAT,
                depth_write_enabled: true,
                depth_compare: wgpu::CompareFunction::Less,
                stencil: wgpu::StencilState::default(),
                bias: wgpu::DepthBiasState::default(),
            }),
            multisample: wgpu::MultisampleState::default(),
            multiview: None,
            label: None,
            cache: None,
        });

        let std_pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
            bind_group_layouts: &[
                &scene.consts_layout,
                &sampler_layout,
                &hdri.layout,
                &sim.cascade0.layout,
                &sim.cascade1.layout,
                &sim.cascade2.layout,
            ],
            push_constant_ranges: &[],
            label: None,
        });
        let std_pipeline = device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
            layout: Some(&std_pipeline_layout),
            vertex: wgpu::VertexState {
                module: shader,
```

```
115                 entry_point: Some("main_vs"),
116                 buffers: &[wgpu::VertexBufferLayout {
117                     array_stride: std::mem::size_of::<super::scene::Vertex>() as _,
118                     step_mode: wgpu::VertexStepMode::Vertex,
119                     attributes: &wgpu::vertex_attr_array![0 => Float32x4, 1=> Uint32x2],
120                 }],
121                 compilation_options: Default::default(),
122             },
123             fragment: Some(wgpu::FragmentState {
124                 module: shader,
125                 entry_point: Some("main_fs"),
126                 targets: &[Some(wgpu::ColorTargetState {
127                     format: FORMAT,
128                     blend: Some(wgpu::BlendState::ALPHA_BLENDING),
129                     write_mask: wgpu::ColorWrites::ALL,
130                 })],
131                 compilation_options: Default::default(),
132             }),
133             primitive: wgpu::PrimitiveState::default(),
134             depth_stencil: Some(wgpu::DepthStencilState {
135                 format: DEPTH_FORMAT,
136                 depth_write_enabled: true,
137                 depth_compare: wgpu::CompareFunction::Less,
138                 stencil: wgpu::StencilState::default(),
139                 bias: wgpu::DepthBiasState::default(),
140             }),
141             multisample: wgpu::MultisampleState::default(),
142             multiview: None,
143             label: None,
144             cache: None,
145         });
146 
147         Self {
148             sampler_layout,
149             sampler_bind_group,
150             depth_view,
151             std_pipeline,
152             hdri,
153             skybox_pipeline,
154         }
155     }
156 
157     pub fn new_depth_view(&mut self, device: &wgpu::Device, window: &winit::window::Window) {
158         let depth_texture = device.create_texture(&wgpu::TextureDescriptor {
159             label: None,
160             size: wgpu::Extent3d {
161                 width: window.inner_size().width,
162                 height: window.inner_size().height,
163                 depth_or_array_layers: 1,
164             },
165             mip_level_count: 1,
166             sample_count: 1,
167             dimension: wgpu::TextureDimension::D2,
168             format: DEPTH_FORMAT,
169             usage: wgpu::TextureUsages::RENDER_ATTACHMENT | wgpu::TextureUsages::TEXTURE_BINDING,
170             view_formats: &[],
171         });
172         self.depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default())
```

```
173     }
174
175     pub fn render_skybox<'a>(
176         &'a self,
177         encoder: &'a mut wgpu::CommandEncoder,
178         surface_view: &wgpu::TextureView,
179         scene: &Scene,
180     ) {
181         let mut pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
182             color_attachments: &[Some(wgpu::RenderPassColorAttachment {
183                 view: surface_view,
184                 resolve_target: None,
185                 ops: wgpu::Operations {
186                     load: wgpu::LoadOp::Clear(wgpu::Color::BLACK),
187                     store: wgpu::StoreOp::Store,
188                 },
189             })],
190             depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
191                 view: &self.depth_view,
192                 depth_ops: Some(wgpu::Operations {
193                     load: wgpu::LoadOp::Clear(1.0),
194                     store: wgpu::StoreOp::Store,
195                 }),
196                 stencil_ops: None,
197             }),
198             timestamp_writes: None,
199             occlusion_query_set: None,
200             label: None,
201         });
202         pass.set_pipeline(&self.skybox_pipeline);
203         pass.set_bind_group(0, &scene.consts_bind_group, &[]);
204         pass.set_bind_group(1, &self.hdri.bind_group, &[]);
205         pass.set_bind_group(2, &self.sampler_bind_group, &[]);
206         // Draw fullscreen triangle
207         pass.draw(0..3, 0..1);
208     }
209
210     pub fn render_standard<'a>(
211         &'a self,
212         encoder: &'a mut wgpu::CommandEncoder,
213         pipeline: &wgpu::RenderPipeline,
214         bind_groups: &[&wgpu::BindGroup],
215         surface_view: &wgpu::TextureView,
216         mesh: &Mesh,
217         instances: u32,
218     ) {
219         let mut pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
220             color_attachments: &[Some(wgpu::RenderPassColorAttachment {
221                 view: surface_view,
222                 resolve_target: None,
223                 ops: wgpu::Operations {
224                     load: wgpu::LoadOp::Load,
225                     store: wgpu::StoreOp::Store,
226                 },
227             })],
228             depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
229                 view: &self.depth_view,
230                 depth_ops: Some(wgpu::Operations {
```

```
231                     load: wgpu::LoadOp::Clear(1.0),
232                     store: wgpu::StoreOp::Store,
233                 }),
234                 stencil_ops: None,
235             }),
236             timestamp_writes: None,
237             occlusion_query_set: None,
238             label: None,
239         });
240         pass.set_pipeline(pipeline);
241         for (i, bind_group) in bind_groups.iter().enumerate() {
242             pass.set_bind_group(i as _, *bind_group, &[]);
243         }
244         pass.set_vertex_buffer(0, mesh.vtx_buf.slice(..));
245         pass.set_index_buffer(mesh.idx_buf.slice(..), wgpu::IndexFormat::Uint32);
246         // Draw multiple mesh instances for tiling
247         pass.draw_indexed(0..(mesh.length as _), 0, 0..(instances * instances));
248     }
249 }
250
```

3.4.2.3. scene.rs

```rust
use crate::cast_slice;
use glam::{Mat4, Vec3, Vec4};
use shared::{Constants, ShaderConstants, SimConstants};
use std::{f32::consts::PI, mem, time::Instant};
use wgpu::{util::DeviceExt, Buffer};
use winit::event::WindowEvent;
use winit::{dpi::PhysicalPosition, event::MouseScrollDelta, window::Window};

pub struct Scene {
    start_time: Instant,
    cursor_down: bool,
    pub camera: Camera,
    pub mesh: Mesh,
    pub consts: Constants,
    pub consts_layout: wgpu::BindGroupLayout,
    pub consts_buf: wgpu::Buffer,
    pub consts_bind_group: wgpu::BindGroup,
    pub consts_changed: bool,
}

pub struct Camera {
    pub proj: Mat4,
    pub view: Mat4,
    pitch: f32,
    yaw: f32,
    zoom: f32,
    eye: Vec3,
    target: Vec3,
    up: Vec3,
    aspect: f32,
    fovy: f32,
    znear: f32,
    zfar: f32,
}

pub struct Mesh {
    pub _vertices: Vec<Vertex>,
    pub idx_buf: Buffer,
    pub vtx_buf: Buffer,
    pub length: usize,
}

#[repr(C, align(16))]
pub struct Vertex {
    pos: Vec4,
    uv: glam::UVec2,
}

impl Scene {
    pub fn new(device: &wgpu::Device, window: &Window) -> Self {
        let cursor_down = false;
        let camera = Camera::new(window);
        let consts = Constants {
            time: 0.0,
            deltatime: 0.0,
            width: 0.0,
```

```
57              height: 0.0,
58              camera_viewproj: camera.proj * camera.view,
59              eye: camera.eye.extend(1.0),
60              shader: ShaderConstants::default(),
61              sim: SimConstants::default(),
62          };
63          let mesh = Mesh::new(device, &consts);
64          let start_time = Instant::now();
65
66          let consts_layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
67              entries: &[wgpu::BindGroupLayoutEntry {
68                  binding: 0,
69                  visibility: wgpu::ShaderStages::VERTEX_FRAGMENT | wgpu::ShaderStages::COMPUTE,
70                  ty: wgpu::BindingType::Buffer {
71                      ty: wgpu::BufferBindingType::Uniform,
72                      has_dynamic_offset: false,
73                      min_binding_size: None,
74                  },
75                  count: None,
76              }],
77              label: None,
78          });
79          let mem_size = (mem::size_of::<Constants>()
80              + mem::size_of::<SimConstants>()
81              + mem::size_of::<ShaderConstants>()) as u64;
82          let consts_buf = device.create_buffer(&wgpu::BufferDescriptor {
83              size: mem_size as u64,
84              mapped_at_creation: false,
85              usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
86              label: Some("Consts Buffer"),
87          });
88          let consts_bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
89              layout: &consts_layout,
90              entries: &[wgpu::BindGroupEntry {
91                  binding: 0,
92                  resource: consts_buf.as_entire_binding(),
93              }],
94              label: Some("Consts Bind Group"),
95          });
96
97          // set to true so the initial spectrum is computed for frame 1
98          let consts_changed = true;
99
100         Self {
101             cursor_down,
102             start_time,
103             consts,
104             camera,
105             mesh,
106             consts_layout,
107             consts_buf,
108             consts_bind_group,
109             consts_changed,
110         }
111     }
112
113     pub fn update_redraw(&mut self, window: &Window) {
114         // Update the constants for use in shaders
```

```
115         let duration = self.start_time.elapsed().as_secs_f32();
116         self.consts.deltatime = duration - self.consts.time;
117         self.consts.time = duration;
118 
119         self.consts.eye = self.camera.eye.extend(1.0);
120         self.consts.shader.proj_mat = self.camera.proj;
121         self.consts.shader.view_mat = self.camera.view;
122 
123         self.consts.shader.light = (Vec3::new(
124             self.consts.shader.sun_x,
125             self.consts.shader.sun_y,
126             self.consts.shader.sun_z,
127         ) * self.consts.shader.sun_distance).extend(1.0);
128         self.consts.shader.light =
129             Mat4::from_rotation_y(self.consts.shader.sun_angle) * self.consts.shader.light;
130         self.consts.sim.logsize = self.consts.sim.size.ilog2();
131 
132         // update incase resized
133         let dimensions = window.inner_size();
134         self.consts.width = dimensions.width as f32;
135         self.consts.height = dimensions.height as f32;
136     }
137 
138     pub fn write(&self, queue: &wgpu::Queue) {
139         queue.write_buffer(&self.consts_buf, 0, cast_slice(&[self.consts]));
140     }
141 
142     pub fn update_camera(&mut self, event: &WindowEvent, window: &Window) {
143         match event {
144             WindowEvent::MouseInput { state, button, .. } => if button ==
    &winit::event::MouseButton::Left {
145                 self.cursor_down = state.is_pressed()
146             },
147             WindowEvent::MouseWheel { delta, .. } => {
148                 self.camera.zoom(*delta);
149                 self.consts.camera_viewproj = self.camera.proj * self.camera.view;
150             }
151             WindowEvent::CursorMoved { position, .. } => {
152                 if self.cursor_down {
153                     self.camera.pan(*position, window);
154                     self.consts.camera_viewproj = self.camera.proj * self.camera.view;
155                 }
156             }
157             _ => {}
158         }
159     }
160 }
161 
162 impl Mesh {
163     pub fn new(device: &wgpu::Device, consts: &Constants) -> Self {
164         let scale = consts.sim.size;
165         let step = consts.sim.mesh_step;
166         // create vertices, stepping from 0,0 towards the top right, offset in shader for centring
167         let mut vertices: Vec<Vertex> = vec![];
168         for z in 0..scale {
169             for x in 0..scale {
170                 let pos = Vec4::new(x as f32 * step, 0.0, z as f32 * step, 1.0);
171                 let uv = glam::UVec2::new(x, z);
```

```
172                 vertices.push(Vertex { pos, uv });
173             }
174         }
175         let vtx_buf = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
176             contents: cast_slice(&vertices),
177             usage: wgpu::BufferUsages::VERTEX,
178             label: None,
179         });
180 
181         let mut indices: Vec<u32> = vec![];
182         for y in 0..scale - 1 {
183             for x in 0..scale - 1 {
184                 indices.push(x + y * scale);
185                 indices.push((x + 1) + (y + 1) * scale);
186                 indices.push(x + (y + 1) * scale);
187                 indices.push(x + y * scale);
188                 indices.push((x + 1) + y * scale);
189                 indices.push((x + 1) + (y + 1) * scale);
190             }
191         }
192 
193         let idx_buf = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
194             contents: cast_slice(&indices),
195             usage: wgpu::BufferUsages::INDEX,
196             label: None,
197         });
198         let length = indices.len();
199 
200         Self {
201             _vertices: vertices,
202             vtx_buf,
203             idx_buf,
204             length,
205         }
206     }
207 }
208 
209 impl Camera {
210     pub fn new(window: &Window) -> Camera {
211         let pitch: f32 = -PI / 2.0;
212         let yaw: f32 = PI / 12.0;
213         let zoom: f32 = 50.0;
214 
215         let mut camera = Camera {
216             pitch,
217             yaw,
218             zoom,
219             eye: Vec3::new(
220                 zoom * yaw.cos() * pitch.sin(),
221                 zoom * yaw.sin(),
222                 zoom * yaw.cos() * pitch.cos(),
223             ),
224             target: Vec3::new(0.0, 0.0, 0.0),
225             // defined y axis as up
226             up: Vec3::new(0.0, 1.0, 0.0),
227             aspect: window.inner_size().width as f32 / window.inner_size().height as f32,
228             fovy: PI / 4.0,
229             znear: 0.1,
```

```
230             // set high enough to not be an issue
231             zfar: 100000.0,
232             proj: Mat4::ZERO,
233             view: Mat4::ZERO,
234         };
235 
236         camera.proj = Mat4::perspective_rh(camera.fovy, camera.aspect, camera.znear, camera.zfar);
237         camera.view = Mat4::look_at_rh(camera.eye, camera.target, camera.up);
238         camera
239     }
240 
241     pub fn zoom(&mut self, delta: MouseScrollDelta) {
242         match delta {
243             MouseScrollDelta::LineDelta(_, y) => {
244                 self.zoom -= y;
245             }
246             MouseScrollDelta::PixelDelta(winit::dpi::PhysicalPosition { y, .. }) => {
247                 self.zoom -= y as f32;
248             }
249         }
250         self.eye = Vec3::new(
251             self.zoom * self.yaw.cos() * self.pitch.sin(),
252             self.zoom * self.yaw.sin(),
253             self.zoom * self.yaw.cos() * self.pitch.cos(),
254         );
255         self.view = Mat4::look_at_rh(self.eye, self.target, self.up);
256     }
257 
258     pub fn pan(&mut self, position: PhysicalPosition<f64>, window: &Window) {
259         let PhysicalPosition { x, y } = position;
260         // +0.01 to prevent edge case at screen borders
261         self.yaw = (PI / window.inner_size().height as f32)
262             * (y as f32 - (window.inner_size().height as f32 / 2.0) + 0.01);
263         self.pitch = ((2.0 * PI) / window.inner_size().width as f32)
264             * (x as f32 - (window.inner_size().width as f32 / 2.0));
265         self.eye = Vec3::new(
266             self.zoom * -self.yaw.cos() * self.pitch.sin(),
267             self.zoom * self.yaw.sin(),
268             self.zoom * self.yaw.cos() * self.pitch.cos(),
269         );
270         self.view = Mat4::look_at_rh(self.eye, self.target, self.up);
271     }
272 
273     pub fn update_fov(&mut self, window: &Window) {
274         self.aspect = window.inner_size().width as f32 / window.inner_size().height as f32;
275         self.proj = Mat4::perspective_rh(self.fovy, self.aspect, self.znear, self.zfar);
276     }
277 }
278 
```

3.4.2.4. ui.rs

```rust
use {crate::{cast_slice, FORMAT}, super::renderer::Renderer, super::scene::Scene,
super::util::Texture};
use imgui::{BackendFlags, DrawVert, FontSource, Key, MouseCursor, TreeNodeFlags, Ui};
use shared::Constants;
use std::{f32::consts::PI, mem};
use wgpu::{util::DeviceExt, Buffer, Device, Queue, RenderPipeline};
use winit::{
    event::{MouseButton, MouseScrollDelta, WindowEvent},
    keyboard::{KeyCode, PhysicalKey},
    window::{CursorIcon, Window},
};

pub struct UI {
    pub pipeline: RenderPipeline,
    pub vtx_buf: Buffer,
    pub idx_buf: Buffer,
    pub context: imgui::Context,
    pub focused: bool,
    texture: Texture,
}

impl UI {
    pub fn new(
        device: &Device,
        queue: &Queue,
        window: &Window,
        shader: &wgpu::ShaderModule,
        renderer: &Renderer,
        scene: &Scene,
    ) -> Self {
        let mut context = imgui::Context::create();
        context.set_ini_filename(None);

        let hidpi_factor = window.scale_factor();
        let dimensions = window.inner_size();

        let io = context.io_mut();
        io.backend_flags.insert(BackendFlags::HAS_MOUSE_CURSORS);
        io.backend_flags.insert(BackendFlags::HAS_SET_MOUSE_POS);
        io.backend_flags
            .insert(BackendFlags::RENDERER_HAS_VTX_OFFSET);
        io.display_size = [dimensions.width as _, dimensions.height as _];
        io.display_framebuffer_scale = [hidpi_factor as f32, hidpi_factor as f32];

        let style = context.style_mut();
        style.window_rounding = 4.0;
        style.popup_rounding = 4.0;
        style.frame_rounding = 2.0;
        style.scale_all_sizes(hidpi_factor as _);

        let fonts = context.fonts();
        fonts.add_font(&[FontSource::DefaultFontData { config: None }]);
        let font_texture = fonts.build_rgba32_texture();
        let texture = Texture::new_sampled(
            font_texture.width,
            font_texture.height,
```

```
56              wgpu::TextureFormat::Rgba8UnormSrgb,
57              device,
58              "Font Atlas",
59          );
60          texture.write(queue, font_texture.data, 4);
61
62          let pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
63              label: None,
64              bind_group_layouts: &[
65                  &scene.consts_layout,
66                  &texture.layout,
67                  &renderer.sampler_layout,
68              ],
69              push_constant_ranges: &[],
70          });
71          let pipeline = device
72              .create_render_pipeline(&wgpu::RenderPipelineDescriptor {
73                  layout: Some(&pipeline_layout),
74                  vertex: wgpu::VertexState {
75                      module: shader,
76                      entry_point: Some("ui::ui_vs"),
77                      buffers: &[wgpu::VertexBufferLayout {
78                          array_stride: mem::size_of::<DrawVert>() as _,
79                          step_mode: wgpu::VertexStepMode::Vertex,
80                          attributes: &wgpu::vertex_attr_array![0 => Float32x2, 1 => Float32x2, 2 =>
    Unorm8x4],
81                      }],
82                      compilation_options: Default::default(),
83                  },
84                  fragment: Some(wgpu::FragmentState {
85                      module: shader,
86                      entry_point: Some("ui::ui_fs"),
87                      targets: &[Some(wgpu::ColorTargetState {
88                          format: FORMAT,
89                          blend: Some(wgpu::BlendState::ALPHA_BLENDING),
90                          write_mask: wgpu::ColorWrites::ALL,
91                      })],
92                      compilation_options: Default::default(),
93                  }),
94                  primitive: wgpu::PrimitiveState::default(),
95                  depth_stencil: None,
96                  multisample: wgpu::MultisampleState::default(),
97                  multiview: None,
98                  cache: None,
99                  label: None,
100             });
101         let vtx_buf = device.create_buffer(&wgpu::BufferDescriptor {
102             size: 0,
103             usage: wgpu::BufferUsages::VERTEX,
104             mapped_at_creation: false,
105             label: None,
106         });
107         let idx_buf = device.create_buffer(&wgpu::BufferDescriptor {
108             size: 0,
109             usage: wgpu::BufferUsages::INDEX,
110             mapped_at_creation: false,
111             label: None,
112         });
```

```
113
114         let focused = true;
115
116         Self {
117             pipeline,
118             vtx_buf,
119             idx_buf,
120             context,
121             texture,
122             focused,
123         }
124     }
125
126     pub fn render<'a>(
127         &'a mut self,
128         device: &Device,
129         queue: &Queue,
130         encoder: &'a mut wgpu::CommandEncoder,
131         surface_view: &'a wgpu::TextureView,
132         sampler_bind_group: &wgpu::BindGroup,
133         scene: &Scene,
134     ) {
135         let mut renderpass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
136             color_attachments: &[Some(wgpu::RenderPassColorAttachment {
137                 view: surface_view,
138                 resolve_target: None,
139                 ops: wgpu::Operations {
140                     load: wgpu::LoadOp::Load,
141                     store: wgpu::StoreOp::Store,
142                 },
143             })],
144             depth_stencil_attachment: None,
145             timestamp_writes: None,
146             occlusion_query_set: None,
147             label: None,
148         });
149
150         let draw_data = self.context.render();
151         if draw_data.total_idx_count == 0 {
152             return;
153         }
154         let mut vertices = Vec::with_capacity(draw_data.total_vtx_count as _);
155         let mut indices = Vec::with_capacity(draw_data.total_idx_count as _);
156         for draw_list in draw_data.draw_lists() {
157             vertices.extend_from_slice(draw_list.vtx_buffer());
158             indices.extend_from_slice(draw_list.idx_buffer());
159         }
160
161         indices.resize(
162             indices.len() + wgpu::COPY_BUFFER_ALIGNMENT as usize
163                 - indices.len() % wgpu::COPY_BUFFER_ALIGNMENT as usize,
164             0,
165         );
166
167         // Logic taken from https://github.com/Yatekii/imgui-wgpu-rs/blob/master/src/lib.rs
168         if (self.idx_buf.size() as usize) < indices.len() * mem::size_of::<u16>() {
169             self.idx_buf = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
170                 contents: cast_slice(&indices),
```

```
                usage: wgpu::BufferUsages::INDEX | wgpu::BufferUsages::COPY_DST,
                label: None,
            });
        } else {
            queue.write_buffer(&self.idx_buf, 0, cast_slice(&indices));
        }

        if (self.vtx_buf.size() as usize) < vertices.len() * mem::size_of::<DrawVert>() {
            self.vtx_buf = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
                contents: cast_slice(&vertices),
                usage: wgpu::BufferUsages::VERTEX | wgpu::BufferUsages::COPY_DST,
                label: None,
            });
        } else {
            queue.write_buffer(&self.vtx_buf, 0, cast_slice(&vertices));
        }

        queue.write_buffer(&scene.consts_buf, 0, cast_slice(&[scene.consts]));
        renderpass.set_pipeline(&self.pipeline);
        renderpass.set_bind_group(0, &scene.consts_bind_group, &[]);
        renderpass.set_bind_group(1, &self.texture.bind_group, &[]);
        renderpass.set_bind_group(2, sampler_bind_group, &[]);
        renderpass.set_vertex_buffer(0, self.vtx_buf.slice(..));
        renderpass.set_index_buffer(self.idx_buf.slice(..), wgpu::IndexFormat::Uint16);

        // Logic taken from https://github.com/Yatekii/imgui-wgpu-rs/blob/master/src/lib.rs
        let mut vtx_offset = 0;
        let mut idx_offset = 0;
        for draw_list in draw_data.draw_lists() {
            for cmd in draw_list.commands() {
                if let imgui::DrawCmd::Elements { count, cmd_params } = cmd {
                    renderpass.set_scissor_rect(
                        cmd_params.clip_rect[0].floor() as _,
                        cmd_params.clip_rect[1].floor() as _,
                        (cmd_params.clip_rect[2] - cmd_params.clip_rect[0].ceil()) as _,
                        (cmd_params.clip_rect[3] - cmd_params.clip_rect[1].ceil()) as _,
                    );
                    let start = idx_offset as u32 + cmd_params.idx_offset as u32;
                    renderpass.draw_indexed(
                        start..(start + count as u32),
                        vtx_offset as i32 + cmd_params.vtx_offset as i32,
                        0..1,
                    );
                }
            }
            vtx_offset += draw_list.vtx_buffer().len();
            idx_offset += draw_list.idx_buffer().len();
        }
    }

    pub fn update_cursor(&mut self, window: &Window) {
        if let Some(cursor) = self.context.mouse_cursor() {
            window.set_cursor_visible(true);
            window.set_cursor_icon(to_winit_cursor(cursor));
        } else {
            window.set_cursor_visible(false);
        }
    }
```

```
229
230     pub fn handle_events(&mut self, event: &WindowEvent) {
231         let io = self.context.io_mut();
232         match event {
233             WindowEvent::Resized(size) => {
234                 io.display_size = [size.width as _, size.height as _];
235             }
236             WindowEvent::KeyboardInput { event, .. } => {
237                 if let PhysicalKey::Code(key) = event.physical_key {
238                     for k in to_imgui_keys(key) {
239                         io.add_key_event(*k, event.state.is_pressed());
240                     }
241                 }
242                 if event.state.is_pressed() {
243                     if let Some(txt) = &event.text {
244                         for ch in txt.chars() {
245                             io.add_input_character(ch);
246                         }
247                     }
248                 }
249             }
250             WindowEvent::ScaleFactorChanged { scale_factor, .. } => {
251                 io.display_framebuffer_scale = [*scale_factor as _; 2];
252             }
253             WindowEvent::MouseInput { state, button, .. } => match button {
254                 MouseButton::Left => io.mouse_down[0] = state.is_pressed(),
255                 MouseButton::Right => io.mouse_down[1] = state.is_pressed(),
256                 MouseButton::Middle => io.mouse_down[2] = state.is_pressed(),
257                 _ => {}
258             },
259             WindowEvent::CursorMoved { position, .. } => {
260                 io.mouse_pos = [position.x as _, position.y as _];
261             }
262             WindowEvent::MouseWheel { delta, .. } => {
263                 // Adjusting scroll speed
264                 let sf = 0.01;
265                 let (h, v) = match delta {
266                     MouseScrollDelta::LineDelta(h, v) => (*h, *v),
267                     MouseScrollDelta::PixelDelta(pos) => (sf * pos.x as f32, sf * pos.y as f32),
268                 };
269                 io.mouse_wheel_h = h;
270                 io.mouse_wheel = v;
271             }
272             WindowEvent::ModifiersChanged(modifiers) => {
273                 io.key_shift = modifiers.state().shift_key();
274                 io.key_alt = modifiers.state().alt_key();
275                 io.key_ctrl = modifiers.state().control_key();
276                 io.key_super = modifiers.state().super_key();
277             }
278             _ => {}
279         }
280     }
281 }
282
283 pub fn build(ui: &Ui, consts: &mut Constants) -> bool {
284     let mut focused = false;
285     let mut pbr_bool = consts.shader.pbr != 0;
286     ui.window("NEA Ocean Simulation")
```

```
287         .always_auto_resize(true)
288         .build(|| {
289             ui.text("Parameters marked with (*) generally should not be changed");
290             ui.text("Info:");
291             ui.text(format!("{:.1$} Elapsed", consts.time, 2));
292             ui.text(format!(
293                 "A {}x{} simulation, running at {} fps",
294                 consts.sim.size,
295                 consts.sim.size,
296                 ui.io().framerate
297             ));
298             if ui.collapsing_header("Simulation Parameters", TreeNodeFlags::DEFAULT_OPEN) {
299                 ui.text("Waves");
300                 ui.slider("Depth", 1.0, 50.0, &mut consts.sim.depth);
301                 ui.slider("Gravity", 0.1, 100.0, &mut consts.sim.gravity);
302                 ui.slider("Wind Speed", 0.1, 100.0, &mut consts.sim.wind_speed);
303                 ui.slider("Wind Offset", -PI, PI, &mut consts.sim.wind_offset);
304                 ui.slider("Fetch", 1000.0, 10000.0, &mut consts.sim.fetch);
305                 ui.slider("Choppiness", 0.0, 1.0, &mut consts.sim.choppiness);
306                 ui.slider("Swell", 0.001, 1.0, &mut consts.sim.swell);
307 
308                 ui.text("Lengthscales");
309                 ui.slider("Lengthscale 0", 1, consts.sim.size, &mut consts.sim.lengthscale0);
310                 ui.slider("Cutoff Low 0*", 0.00000, 0.00001, &mut consts.sim.cutoff_low0);
311                 ui.slider("Cutoff High 0", 0.01, 15.0, &mut consts.sim.cutoff_high0);
312                 ui.slider("Lengthscale 0 Scale Factor", 0.0, 1.0, &mut
    consts.sim.lengthscale0_sf);
313 
314                 ui.slider("Lengthscale 1", 1, consts.sim.size, &mut consts.sim.lengthscale1);
315                 ui.slider("Cutoff Low 1", 0.00000, 15.0, &mut consts.sim.cutoff_low1);
316                 ui.slider("Cutoff High 1", 0.01, 15.0, &mut consts.sim.cutoff_high1);
317                 ui.slider("Lengthscale 1 Scale Factor", 0.0, 1.0, &mut
    consts.sim.lengthscale1_sf);
318 
319                 ui.slider("Lengthscale 2", 1, consts.sim.size, &mut consts.sim.lengthscale2);
320                 ui.slider("Cutoff Low 2", 0.00000, 15.0, &mut consts.sim.cutoff_low2);
321                 ui.slider("Cutoff High 2", 0.0, 15.0, &mut consts.sim.cutoff_high2);
322                 ui.slider("Lengthscale 2 Scale Factor", 0.0, 1.0, &mut
    consts.sim.lengthscale2_sf);
323 
324                 ui.text("Foam");
325                 ui.color_edit4("Foam Color", consts.shader.foam_color.as_mut());
326                 ui.slider("Decay", 0.0, 0.3, &mut consts.sim.foam_decay);
327                 ui.slider("Bias", 0.00, 2.0, &mut consts.sim.foam_bias);
328                 ui.slider("Injection Threshold", -1.00, 1.0, &mut consts.sim.injection_threshold);
329                 ui.slider("Injection Amount", 0.00, 2.0, &mut consts.sim.injection_amount);
330                 ui.text("Misc");
331                 ui.slider("Instances per Axis",1, 10, &mut consts.sim.instances);
332                 ui.slider("Instance micro Offset",0.9, 1.0, &mut
    consts.sim.instance_micro_offset);
333                 ui.slider("Mesh Step", 0.0, 1.0, &mut consts.sim.mesh_step);
334                 ui.slider("Integration Step*", 0.001, 0.02, &mut consts.sim.integration_step);
335             }
336             ui.separator();
337             if ui.collapsing_header("Shader Parameters", TreeNodeFlags::DEFAULT_OPEN) {
338                 ui.text("PBR");
339                 ui.checkbox("PBR", &mut pbr_bool);
340                 ui.slider("PBR Specular Scale Factor", 0.0, 10.0, &mut consts.shader.pbr_sf);
```

```
341                 ui.slider("PBR Fresnel Effect Scale Factor", 0.0, 1.0, &mut
    consts.shader.fresnel_pbr_sf);
342                 ui.slider("PBR Cutoff Low*", 0.0, 0.2, &mut consts.shader.pbr_cutoff);
343                 ui.slider("Water Roughness", 0.0, 0.5, &mut consts.shader.roughness);
344                 ui.slider("Foam Roughness Modifier", 0.0, 2.0, &mut consts.shader.foam_roughness);
345                 ui.text("Fresnel");
346                 ui.slider("Water Refractive Index", 0.0, 2.0, &mut consts.shader.water_ri);
347                 ui.slider("Air Refractive Index*", 0.0, 2.0, &mut consts.shader.air_ri);
348                 ui.slider("Fresnel Shine", 0.0, 10.0, &mut consts.shader.fresnel_shine);
349                 ui.slider("Fresnel Effect Scale Factor", 0.0, 1.0, &mut consts.shader.fresnel_sf);
350                 ui.slider("Fresnel Normal Scale Factor", 0.0, 1.0, &mut
    consts.shader.fresnel_normal_sf);
351                 ui.text("Subsurface Scattering");
352                 ui.color_edit4("Scatter Color", consts.shader.scatter_color.as_mut());
353                 ui.color_edit4("Bubble Color", consts.shader.bubble_color.as_mut());
354                 ui.slider("Height Attenuation", 0.0, 1.0, &mut consts.shader.ss_height);
355                 ui.slider("Reflection Strength", 0.0, 1.0, &mut consts.shader.ss_reflected);
356                 ui.slider("Diffuse Strength", 0.0, 1.0, &mut consts.shader.ss_lambert);
357                 ui.slider("Ambient Light Strength", 0.0, 1.0, &mut consts.shader.ss_ambient);
358                 ui.slider("Air Bubble Density", 0.0, 1.0, &mut consts.shader.bubble_density);
359                 ui.text("Fog");
360                 ui.color_edit4("Fog Color", consts.shader.fog_color.as_mut());
361                 ui.slider("Fog Density", 0.0, 10.0, &mut consts.shader.fog_density);
362                 ui.slider("Fog Offset", 0.0, 500.0, &mut consts.shader.fog_offset);
363                 ui.slider("Fog Falloff", 0.0, 10.0, &mut consts.shader.fog_falloff);
364                 ui.slider("Fog Height", 0.0, 100.0, &mut consts.shader.fog_height);
365                 ui.text("Misc");
366                 ui.slider("Blinn Phong Shininess", 0.0, 50.0, &mut consts.shader.shininess);
367                 ui.slider("Reflections Strength", 0.0, 10.0, &mut consts.shader.reflection_sf);
368             }
369             ui.separator();
370             if ui.collapsing_header("World Parameters", TreeNodeFlags::DEFAULT_OPEN) {
371                 ui.text("Sun");
372                 ui.color_edit4("Sun Color", consts.shader.sun_color.as_mut());
373                 ui.slider("Sun X", -1.0, 1.0, &mut consts.shader.sun_x);
374                 ui.slider("Sun Y", -1.0, 1.0, &mut consts.shader.sun_y);
375                 ui.slider("Sun Z", -1.0, 1.0, &mut consts.shader.sun_z);
376                 ui.slider("Sun Angle", -PI, PI, &mut consts.shader.sun_angle);
377                 ui.slider("Sun Distance", 0.0, 500.0, &mut consts.shader.sun_distance);
378                 ui.slider("Sun Size", 0.0, PI / 18.0, &mut consts.shader.sun_size);
379                 ui.slider("Sun Falloff", 0.0, 10000.0, &mut consts.shader.sun_falloff);
380                 ui.text("Water");
381                 ui.slider("Height Offset", 0.0, 50.0, &mut consts.sim.height_offset);
382             }
383             focused = ui.is_window_focused();
384             consts.shader.pbr = pbr_bool as u32;
385         });
386     focused
387 }
388
389 // code adapted from https://github.com/imgui-rs/imgui-winit-support
390 fn to_winit_cursor(cursor: MouseCursor) -> CursorIcon {
391     match cursor {
392         MouseCursor::Arrow => CursorIcon::Default,
393         MouseCursor::TextInput => CursorIcon::Text,
394         MouseCursor::ResizeAll => CursorIcon::Move,
395         MouseCursor::ResizeNS => CursorIcon::NsResize,
396         MouseCursor::ResizeEW => CursorIcon::EwResize,
```

```
397         MouseCursor::ResizeNESW => CursorIcon::NeswResize,
398         MouseCursor::ResizeNWSE => CursorIcon::NwseResize,
399         MouseCursor::Hand => CursorIcon::Grab,
400         MouseCursor::NotAllowed => CursorIcon::NotAllowed,
401     }
402 }
403
404 // code adapted from https://github.com/imgui-rs/imgui-winit-support
405 fn to_imgui_keys(keycode: KeyCode) -> &'static [Key] {
406     match keycode {
407         KeyCode::Tab => &[Key::Tab],
408         KeyCode::ArrowLeft => &[Key::LeftArrow],
409         KeyCode::ArrowRight => &[Key::RightArrow],
410         KeyCode::ArrowUp => &[Key::UpArrow],
411         KeyCode::ArrowDown => &[Key::DownArrow],
412         KeyCode::PageUp => &[Key::PageUp],
413         KeyCode::PageDown => &[Key::PageDown],
414         KeyCode::Home => &[Key::Home],
415         KeyCode::End => &[Key::End],
416         KeyCode::Insert => &[Key::Insert],
417         KeyCode::Delete => &[Key::Delete],
418         KeyCode::Backspace => &[Key::Backspace],
419         KeyCode::Space => &[Key::Space],
420         KeyCode::Enter => &[Key::Enter],
421         KeyCode::Escape => &[Key::Escape],
422         KeyCode::ControlLeft => &[Key::LeftCtrl, Key::ModCtrl],
423         KeyCode::ShiftLeft => &[Key::LeftShift, Key::ModShift],
424         KeyCode::AltLeft => &[Key::LeftAlt, Key::ModAlt],
425         KeyCode::SuperLeft => &[Key::LeftSuper, Key::ModSuper],
426         KeyCode::ControlRight => &[Key::RightCtrl, Key::ModCtrl],
427         KeyCode::ShiftRight => &[Key::RightShift, Key::ModShift],
428         KeyCode::AltRight => &[Key::RightAlt, Key::ModAlt],
429         KeyCode::SuperRight => &[Key::RightSuper, Key::ModSuper],
430         KeyCode::ContextMenu => &[Key::Menu],
431         KeyCode::Digit0 => &[Key::Alpha0],
432         KeyCode::Digit1 => &[Key::Alpha1],
433         KeyCode::Digit2 => &[Key::Alpha2],
434         KeyCode::Digit3 => &[Key::Alpha3],
435         KeyCode::Digit4 => &[Key::Alpha4],
436         KeyCode::Digit5 => &[Key::Alpha5],
437         KeyCode::Digit6 => &[Key::Alpha6],
438         KeyCode::Digit7 => &[Key::Alpha7],
439         KeyCode::Digit8 => &[Key::Alpha8],
440         KeyCode::Digit9 => &[Key::Alpha9],
441         KeyCode::KeyA => &[Key::A],
442         KeyCode::KeyB => &[Key::B],
443         KeyCode::KeyC => &[Key::C],
444         KeyCode::KeyD => &[Key::D],
445         KeyCode::KeyE => &[Key::E],
446         KeyCode::KeyF => &[Key::F],
447         KeyCode::KeyG => &[Key::G],
448         KeyCode::KeyH => &[Key::H],
449         KeyCode::KeyI => &[Key::I],
450         KeyCode::KeyJ => &[Key::J],
451         KeyCode::KeyK => &[Key::K],
452         KeyCode::KeyL => &[Key::L],
453         KeyCode::KeyM => &[Key::M],
454         KeyCode::KeyN => &[Key::N],
```

```
455         KeyCode::KeyO => &[Key::O],
456         KeyCode::KeyP => &[Key::P],
457         KeyCode::KeyQ => &[Key::Q],
458         KeyCode::KeyR => &[Key::R],
459         KeyCode::KeyS => &[Key::S],
460         KeyCode::KeyT => &[Key::T],
461         KeyCode::KeyU => &[Key::U],
462         KeyCode::KeyV => &[Key::V],
463         KeyCode::KeyW => &[Key::W],
464         KeyCode::KeyX => &[Key::X],
465         KeyCode::KeyY => &[Key::Y],
466         KeyCode::KeyZ => &[Key::Z],
467         KeyCode::F1 => &[Key::F1],
468         KeyCode::F2 => &[Key::F2],
469         KeyCode::F3 => &[Key::F3],
470         KeyCode::F4 => &[Key::F4],
471         KeyCode::F5 => &[Key::F5],
472         KeyCode::F6 => &[Key::F6],
473         KeyCode::F7 => &[Key::F7],
474         KeyCode::F8 => &[Key::F8],
475         KeyCode::F9 => &[Key::F9],
476         KeyCode::F10 => &[Key::F10],
477         KeyCode::F11 => &[Key::F11],
478         KeyCode::F12 => &[Key::F12],
479         KeyCode::Quote => &[Key::Apostrophe],
480         KeyCode::Comma => &[Key::Comma],
481         KeyCode::Minus => &[Key::Minus],
482         KeyCode::Period => &[Key::Period],
483         KeyCode::Slash => &[Key::Slash],
484         KeyCode::Semicolon => &[Key::Semicolon],
485         KeyCode::Equal => &[Key::Equal],
486         KeyCode::BracketLeft => &[Key::LeftBracket],
487         KeyCode::Backslash => &[Key::Backslash],
488         KeyCode::BracketRight => &[Key::RightBracket],
489         KeyCode::Backquote => &[Key::GraveAccent],
490         KeyCode::CapsLock => &[Key::CapsLock],
491         KeyCode::ScrollLock => &[Key::ScrollLock],
492         KeyCode::NumLock => &[Key::NumLock],
493         KeyCode::PrintScreen => &[Key::PrintScreen],
494         KeyCode::Pause => &[Key::Pause],
495         KeyCode::Numpad0 => &[Key::Keypad0],
496         KeyCode::Numpad1 => &[Key::Keypad1],
497         KeyCode::Numpad2 => &[Key::Keypad2],
498         KeyCode::Numpad3 => &[Key::Keypad3],
499         KeyCode::Numpad4 => &[Key::Keypad4],
500         KeyCode::Numpad5 => &[Key::Keypad5],
501         KeyCode::Numpad6 => &[Key::Keypad6],
502         KeyCode::Numpad7 => &[Key::Keypad7],
503         KeyCode::Numpad8 => &[Key::Keypad8],
504         KeyCode::Numpad9 => &[Key::Keypad9],
505         KeyCode::NumpadDecimal => &[Key::KeypadDecimal],
506         KeyCode::NumpadDivide => &[Key::KeypadDivide],
507         KeyCode::NumpadMultiply => &[Key::KeypadMultiply],
508         KeyCode::NumpadSubtract => &[Key::KeypadSubtract],
509         KeyCode::NumpadAdd => &[Key::KeypadAdd],
510         KeyCode::NumpadEnter => &[Key::KeypadEnter],
511         KeyCode::NumpadEqual => &[Key::KeypadEqual],
512         _ => &[],
```

```
513     }
514 }
515
```

3.4.2.5. util.rs

```rust
use image::io::Reader;
use image::GenericImageView;
use std::fs::File;
use std::io::Cursor;
use std::io::Read;
use wgpu::util::DeviceExt;
use wgpu::Queue;

pub struct Texture {
    pub texture: wgpu::Texture,
    pub view: wgpu::TextureView,
    pub bind_group: wgpu::BindGroup,
    pub layout: wgpu::BindGroupLayout,
}

impl Texture {
    pub fn new_sampled(
        width: u32,
        height: u32,
        format: wgpu::TextureFormat,
        device: &wgpu::Device,
        label: &str,
    ) -> Self {
        let texture = device.create_texture(&wgpu::TextureDescriptor {
            size: wgpu::Extent3d {
                width,
                height,
                depth_or_array_layers: 1,
            },
            mip_level_count: 1,
            sample_count: 1,
            dimension: wgpu::TextureDimension::D2,
            format,
            usage: wgpu::TextureUsages::TEXTURE_BINDING | wgpu::TextureUsages::COPY_DST,
            view_formats: &[],
            label: Some(label),
        });
        let view = texture.create_view(&wgpu::TextureViewDescriptor::default());
        let layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
            entries: &[sampled_bind_group_descriptor(0)],
            label: Some(label),
        });
        let bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
            layout: &layout,
            entries: &[wgpu::BindGroupEntry {
                binding: 0,
                resource: wgpu::BindingResource::TextureView(&view),
            }],
            label: Some(label),
        });

        Self {
            texture,
            view,
            bind_group,
            layout,
```

```
        }
    }

    pub fn new_storage(
        width: u32,
        height: u32,
        format: wgpu::TextureFormat,
        device: &wgpu::Device,
        label: &str,
    ) -> Self {
        let texture = device.create_texture(&wgpu::TextureDescriptor {
            size: wgpu::Extent3d {
                width,
                height,
                depth_or_array_layers: 1,
            },
            mip_level_count: 1,
            sample_count: 1,
            dimension: wgpu::TextureDimension::D2,
            format,
            usage: wgpu::TextureUsages::STORAGE_BINDING
                | wgpu::TextureUsages::COPY_DST
                | wgpu::TextureUsages::TEXTURE_BINDING,
            view_formats: &[],
            label: Some(label),
        });
        let view = texture.create_view(&wgpu::TextureViewDescriptor::default());
        let layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
            entries: &[bind_group_descriptor(0, format)],
            label: Some(label),
        });
        let bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
            layout: &layout,
            entries: &[wgpu::BindGroupEntry {
                binding: 0,
                resource: wgpu::BindingResource::TextureView(&view),
            }],
            label: Some(label),
        });

        Self {
            texture,
            view,
            layout,
            bind_group,
        }
    }

    pub fn write(&self, queue: &Queue, data: &[u8], size: u32) {
        queue.write_texture(
            wgpu::ImageCopyTexture {
                texture: &self.texture,
                mip_level: 0,
                origin: wgpu::Origin3d::ZERO,
                aspect: wgpu::TextureAspect::All,
            },
            data,
            wgpu::ImageDataLayout {
```

```
115                 offset: 0,
116                 bytes_per_row: Some(size * self.texture.width()),
117                 rows_per_image: Some(self.texture.height()),
118             },
119             self.texture.size(),
120         );
121     }
122 
123     pub fn from_file(device: &wgpu::Device, queue: &Queue, label: &str, file: &str) -> Self {
124         // cant include_bytes! as filepath is non static
125         let mut file_data = Vec::new();
126         File::open(file)
127             .expect("failed to open file")
128             .read_to_end(&mut file_data)
129             .expect("failed to read file");
130 
131         let diffuse_image = Reader::new(Cursor::new(file_data))
132             .with_guessed_format()
133             .expect("failed to guess format")
134             .decode()
135             .expect("failed to decode image");
136         let diffuse_rgba = diffuse_image.to_rgba32f();
137         let dimensions = diffuse_image.dimensions();
138         let texture_size = wgpu::Extent3d {
139             width: dimensions.0,
140             height: dimensions.1,
141             depth_or_array_layers: 1,
142         };
143 
144         let texture = device.create_texture_with_data(
145             queue,
146             &wgpu::TextureDescriptor {
147                 size: texture_size,
148                 mip_level_count: 1,
149                 sample_count: 1,
150                 dimension: wgpu::TextureDimension::D2,
151                 format: wgpu::TextureFormat::Rgba32Float,
152                 usage: wgpu::TextureUsages::TEXTURE_BINDING | wgpu::TextureUsages::COPY_DST,
153                 label: None,
154                 view_formats: &[],
155             },
156             wgpu::util::TextureDataOrder::default(),
157             super::cast_slice(diffuse_rgba.as_raw()),
158         );
159         let view = texture.create_view(&wgpu::TextureViewDescriptor::default());
160         let layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
161             entries: &[sampled_bind_group_descriptor(0)],
162             label: Some(label),
163         });
164         let bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
165             layout: &layout,
166             entries: &[wgpu::BindGroupEntry {
167                 binding: 0,
168                 resource: wgpu::BindingResource::TextureView(&view),
169             }],
170             label: Some(label),
171         });
172         Self {
```

```
            texture,
            view,
            layout,
            bind_group,
        }
    }
}


// debatably redundant as not app agnositc, but reduces LOC significantly
pub fn bind_group_descriptor(
    binding: u32,
    format: wgpu::TextureFormat,
) -> wgpu::BindGroupLayoutEntry {
    wgpu::BindGroupLayoutEntry {
        binding,
        visibility: wgpu::ShaderStages::COMPUTE | wgpu::ShaderStages::VERTEX_FRAGMENT,
        ty: wgpu::BindingType::StorageTexture {
            access: wgpu::StorageTextureAccess::ReadWrite,
            format,
            view_dimension: wgpu::TextureViewDimension::D2,
        },
        count: None,
    }
}

pub fn sampled_bind_group_descriptor(binding: u32) -> wgpu::BindGroupLayoutEntry {
    wgpu::BindGroupLayoutEntry {
        binding,
        visibility: wgpu::ShaderStages::VERTEX_FRAGMENT,
        ty: wgpu::BindingType::Texture {
            sample_type: wgpu::TextureSampleType::Float { filterable: false },
            view_dimension: wgpu::TextureViewDimension::D2,
            multisampled: false,
        },
        count: None,
    }
}

```

3.4.3. simulation

3.4.3.1. mod.rs

```rust
use super::sim::compute::ComputePass;
use super::sim::fft::FourierTransform;
use simdata::SimData;
use crate::cast_slice;
use crate::engine::scene::Scene;
use cascade::Cascade;

pub mod compute;
pub mod fft;
pub mod cascade;
pub mod simdata;


pub struct Simulation {
    pub simdata: SimData,
    pub cascade0: Cascade,
    pub cascade1: Cascade,
    pub cascade2: Cascade,
    pub butterfly_precompute_pass: ComputePass,
    pub initial_spectra_pass: ComputePass,
    pub conjugates_pass: ComputePass,
    pub evolve_spectra_pass: ComputePass,
    pub process_deltas_pass: ComputePass,
    pub fft: FourierTransform,
}

impl Simulation {
    pub fn new(
        device: &wgpu::Device,
        queue: &wgpu::Queue,
        shader: &wgpu::ShaderModule,
        scene: &Scene,
    ) -> Self {
        let simdata = SimData::new(device, &scene.consts);

        let cascade0 = Cascade::new(device, &scene.consts, "0");
        let cascade1 = Cascade::new(device, &scene.consts, "1");
        let cascade2 = Cascade::new(device, &scene.consts, "2");

        let push_constant_ranges = &[wgpu::PushConstantRange {
            stages: wgpu::ShaderStages::COMPUTE,
            range: 0..std::mem::size_of::<u32>() as u32,
        }];
        let initial_spectra_pass = ComputePass::new(
            &[&scene.consts_layout, &simdata.layout, &cascade0.layout],
            push_constant_ranges,
            device,
            shader,
            "Initial Spectra",
            "sim::initial_spectra::main",
        );
        let butterfly_precompute_pass = ComputePass::new(
            &[&scene.consts_layout, &simdata.layout],
            &[],
```

```
            device,
            shader,
            "Precompute Butterfly",
            "sim::fft::precompute_butterfly",
        );
        let conjugates_pass = ComputePass::new(
            &[&scene.consts_layout, &simdata.layout, &cascade0.layout],
            &[],
            device,
            shader,
            "Pack Conjugates",
            "sim::initial_spectra::pack_conjugates",
        );
        let evolve_spectra_pass = ComputePass::new(
            &[
                &scene.consts_layout,
                &cascade0.layout,
                &cascade0.h_displacement.layout,
                &cascade0.v_displacement.layout,
                &cascade0.h_slope.layout,
                &cascade0.jacobian.layout,
            ],
            &[],
            device,
            shader,
            "Evolve Spectra",
            "sim::evolve_spectra::main",
        );
        let process_deltas_pass = ComputePass::new(
            &[
                &scene.consts_layout,
                &cascade0.h_displacement.layout,
                &cascade0.v_displacement.layout,
                &cascade0.h_slope.layout,
                &cascade0.jacobian.layout,
                &cascade0.layout,
            ],
            &[],
            device,
            shader,
            "Process Deltas",
            "sim::process_deltas::main",
        );
        let fft = FourierTransform::new(device, shader, scene, &simdata);

        simdata
            .gaussian_tex
            .write(queue, cast_slice(&simdata.gaussian_noise.clone()), 16);

        Self {
            cascade0,
            cascade1,
            cascade2,
            simdata,
            initial_spectra_pass,
            butterfly_precompute_pass,
            conjugates_pass,
            evolve_spectra_pass,
```

```
113             process_deltas_pass,
114             fft,
115         }
116     }
117     pub fn compute_cascade<'a>(
118         &'a self,
119         encoder: &'a mut wgpu::CommandEncoder,
120         cascade: &Cascade,
121         scene: &mut Scene,
122         workgroup_size: u32,
123         index: u32,
124     ) {
125         self.evolve_spectra_pass.compute(
126             encoder,
127             "Evolve Spectra",
128             &[
129                 &scene.consts_bind_group,
130                 &cascade.bind_group,
131                 &cascade.h_displacement.bind_group,
132                 &cascade.v_displacement.bind_group,
133                 &cascade.h_slope.bind_group,
134                 &cascade.jacobian.bind_group,
135             ],
136             workgroup_size,
137             workgroup_size,
138         );
139         self.fft.ifft2d(
140             encoder,
141             scene,
142             &self.simdata,
143             &cascade.h_displacement,
144             index,
145         );
146         self.fft.ifft2d(
147             encoder,
148             scene,
149             &self.simdata,
150             &cascade.v_displacement,
151             index,
152         );
153         self.fft.ifft2d(
154             encoder,
155             scene,
156             &self.simdata,
157             &cascade.h_slope,
158             index,
159         );
160         self.fft.ifft2d(
161             encoder,
162             scene,
163             &self.simdata,
164             &cascade.jacobian,
165             index,
166         );
167
168         self.process_deltas_pass.compute(
169             encoder,
170             "Process Deltas",
```

```
171             &[
172                 &scene.consts_bind_group,
173                 &cascade.h_displacement.bind_group,
174                 &cascade.v_displacement.bind_group,
175                 &cascade.h_slope.bind_group,
176                 &cascade.jacobian.bind_group,
177                 &cascade.bind_group,
178             ],
179             workgroup_size,
180             workgroup_size,
181         );
182     }
183     // creating a proper abstraction for a compute pass would just involve recreating a
184     // computepipeline struct from scratch, so instead as this isnt a true "engine" i have just
185     // special cased a computepass that requires push constants
186     pub fn compute_initial<'a>(
187         &'a self,
188         encoder: &'a mut wgpu::CommandEncoder,
189         bind_groups: &[&wgpu::BindGroup],
190         pc: u32,
191         x: u32,
192         y: u32,
193     ) {
194         let mut pass = encoder.begin_compute_pass(&wgpu::ComputePassDescriptor {
195             timestamp_writes: None,
196             label: Some("Initial Spectra"),
197         });
198 
199         pass.set_pipeline(&self.initial_spectra_pass.pipeline);
200         for (i, bind_group) in bind_groups.iter().enumerate() {
201             pass.set_bind_group(i as _, *bind_group, &[]);
202         }
203         pass.set_push_constants(0, cast_slice(&[pc]));
204         pass.dispatch_workgroups(x, y, 1);
205         drop(pass);
206 
207         let mut pass = encoder.begin_compute_pass(&wgpu::ComputePassDescriptor {
208             timestamp_writes: None,
209             label: Some("Conjugates"),
210         });
211 
212         pass.set_pipeline(&self.conjugates_pass.pipeline);
213         for (i, bind_group) in bind_groups.iter().enumerate() {
214             pass.set_bind_group(i as _, *bind_group, &[]);
215         }
216         pass.dispatch_workgroups(x, y, 1);
217     }
218 }
219 
```

3.4.3.2. simdata.rs

```rust
use glam::{Vec2, Vec4};
use crate::engine::util::{bind_group_descriptor, Texture};
use shared::Constants;

pub struct SimData {
    pub gaussian_noise: Vec<Vec4>,
    pub gaussian_tex: Texture,
    pub layout: wgpu::BindGroupLayout,
    pub bind_group: wgpu::BindGroup,
}

impl SimData {
    pub fn new(device: &wgpu::Device, consts: &Constants) -> Self {
        let gaussian_tex = Texture::new_storage(
            consts.sim.size,
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            "Gaussian",
        );
        let gaussian_noise = Self::guassian_noise(consts);

        let butterfly_tex = Texture::new_storage(
            consts.sim.size.ilog2(),
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            "Butterfly",
        );

        let layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
            entries: &[
                bind_group_descriptor(0, wgpu::TextureFormat::Rgba32Float),
                bind_group_descriptor(1, wgpu::TextureFormat::Rgba32Float),
            ],
            label: Some("Sim Data Layout"),
        });
        let bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
            layout: &layout,
            entries: &[
                wgpu::BindGroupEntry {
                    binding: 0,
                    resource: wgpu::BindingResource::TextureView(&gaussian_tex.view),
                },
                wgpu::BindGroupEntry {
                    binding: 1,
                    resource: wgpu::BindingResource::TextureView(&butterfly_tex.view),
                },
            ],
            label: Some("Sim Data Textures"),
        });

        Self {
            gaussian_tex,
            gaussian_noise,
            bind_group,
```

```
57              layout,
58          }
59      }
60
61      fn guassian_noise(consts: &Constants) -> Vec<Vec4> {
62          let mut rng = Xoshiro256plus::new(consts.sim.seed as _);
63          let mut data = vec![];
64          for _ in 0..(consts.sim.size * consts.sim.size) {
65              let gaussian_pair =
66                  Self::gaussian_number(
67                      rng.next() as _,
68                      rng.next() as _,
69              );
70              data.push(Vec4::new(gaussian_pair.x, gaussian_pair.y, 0.0, 1.0));
71          }
72          data
73      }
74      // box muller transform, gaussian pair technically not needed but is slightly cooler
75      fn gaussian_number(u1: f32, u2: f32) -> Vec2 {
76          Vec2::new(
77              (-2.0 * u1.ln()).sqrt() * (2.0 * std::f32::consts::PI * u2).cos(),
78              (-2.0 * u1.ln()).sqrt() * (2.0 * std::f32::consts::PI * u2).sin(),
79          )
80      }
81  }
82
83  struct Xoshiro256plus {
84      seed: [u64; 4]
85  }
86
87  impl Xoshiro256plus {
88      pub fn rol64(x: u64, k: i32) -> u64 {
89          (x << k) | (x >> (64 - k))
90      }
91      fn new(seed: u64) -> Self {
92          let mut rng = SplitMix::new(seed);
93          Xoshiro256plus {
94              seed: [rng.next(), rng.next(), rng.next(), rng.next()],
95          }
96      }
97      fn next(&mut self) -> f64 {
98          let result = self.seed[0].wrapping_add(self.seed[3]);
99          let t = self.seed[1] << 17;
100
101         self.seed[2] ^= self.seed[0];
102         self.seed[3] ^= self.seed[1];
103         self.seed[1] ^= self.seed[2];
104         self.seed[0] ^= self.seed[3];
105
106         self.seed[2] ^= t;
107         self.seed[3] = Xoshiro256plus::rol64(self.seed[3], 45);
108
109         (result >> 11) as f64 * (1.0 / (1u64 << 53) as f64)
110     }
111 }
112
113 pub struct SplitMix {
114     seed: u64,
```

```
}


impl SplitMix {
    fn new(seed: u64) -> Self {
        SplitMix { seed }
    }
    /// from https://xoshiro.di.unimi.it/splitmix64.c
    fn next(&mut self) -> u64 {
        self.seed = self.seed.wrapping_add(0x9e3779b97f4a7c15);
        let mut z: u64 = self.seed;
        z = (z ^ (z >> 30)).wrapping_mul(0xbf58476d1ce4e5b9);
        z = (z ^ (z >> 27)).wrapping_mul(0x94d049bb133111eb);
        z ^ (z >> 31)
    }
}

```

3.4.3.3. cascade.rs

```rust
use crate::engine::util::{bind_group_descriptor, Texture};
use shared::Constants;

pub struct Cascade {
    pub bind_group: wgpu::BindGroup,
    pub layout: wgpu::BindGroupLayout,
    pub h_displacement: Texture,
    pub h_slope: Texture,
    pub v_displacement: Texture,
    pub jacobian: Texture,
}

impl Cascade {
    pub fn new(device: &wgpu::Device, consts: &Constants, index: &str) -> Self {
        let wave_texture = Texture::new_storage(
            consts.sim.size,
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            &format!("Waves {}", index),
        );
        let initial_spectrum_texture = Texture::new_storage(
            consts.sim.size,
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            &format!("Initial Spectrum {}", index),
        );
        let evolved_spectrum_texture = Texture::new_storage(
            consts.sim.size,
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            &format!("Evolved Spectrum {}", index),
        );
        let displacement_map = Texture::new_storage(
            consts.sim.size,
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            &format!("Displacement Map {}", index),
        );
        let normal_map = Texture::new_storage(
            consts.sim.size,
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            &format!("Normal Map{}", index),
        );
        let foam_map = Texture::new_storage(
            consts.sim.size,
            consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            &format!("Foam {}", index),
        );
```

```
57
58          let layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
59              entries: &[
60                  bind_group_descriptor(0, wgpu::TextureFormat::Rgba32Float),
61                  bind_group_descriptor(1, wgpu::TextureFormat::Rgba32Float),
62                  bind_group_descriptor(2, wgpu::TextureFormat::Rgba32Float),
63                  bind_group_descriptor(3, wgpu::TextureFormat::Rgba32Float),
64                  bind_group_descriptor(4, wgpu::TextureFormat::Rgba32Float),
65                  bind_group_descriptor(5, wgpu::TextureFormat::Rgba32Float),
66              ],
67              label: Some("Storage Textures Layout"),
68          });
69          let bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor {
70              layout: &layout,
71              entries: &[
72                  wgpu::BindGroupEntry {
73                      binding: 0,
74                      resource: wgpu::BindingResource::TextureView(&wave_texture.view),
75                  },
76                  wgpu::BindGroupEntry {
77                      binding: 1,
78                      resource: wgpu::BindingResource::TextureView(&initial_spectrum_texture.view),
79                  },
80                  wgpu::BindGroupEntry {
81                      binding: 2,
82                      resource: wgpu::BindingResource::TextureView(&evolved_spectrum_texture.view),
83                  },
84                  wgpu::BindGroupEntry {
85                      binding: 3,
86                      resource: wgpu::BindingResource::TextureView(&displacement_map.view),
87                  },
88                  wgpu::BindGroupEntry {
89                      binding: 4,
90                      resource: wgpu::BindingResource::TextureView(&normal_map.view),
91                  },
92                  wgpu::BindGroupEntry {
93                      binding: 5,
94                      resource: wgpu::BindingResource::TextureView(&foam_map.view),
95                  },
96              ],
97              label: Some(&format!("Storage Textures {}", index)),
98          });
99
100         let h_displacement = Texture::new_storage(
101             consts.sim.size,
102             consts.sim.size,
103             wgpu::TextureFormat::Rgba32Float,
104             device,
105             &format!("h_displacement {}", index),
106         );
107         let h_slope = Texture::new_storage(
108             consts.sim.size,
109             consts.sim.size,
110             wgpu::TextureFormat::Rgba32Float,
111             device,
112             &format!("h_slope {}", index),
113         );
114         let jacobian = Texture::new_storage(
```

```
115             consts.sim.size,
116             consts.sim.size,
117             wgpu::TextureFormat::Rgba32Float,
118             device,
119             &format!("jacobian {}", index),
120         );
121         let v_displacement = Texture::new_storage(
122             consts.sim.size,
123             consts.sim.size,
124             wgpu::TextureFormat::Rgba32Float,
125             device,
126             &format!("v_displacment {}", index),
127         );
128 
129         Self {
130             layout,
131             bind_group,
132             h_slope,
133             h_displacement,
134             v_displacement,
135             jacobian,
136         }
137     }
138 }
139 
```

3.4.3.4. compute.rs

```rust
pub struct ComputePass {
    pub pipeline: wgpu::ComputePipeline,
}

impl ComputePass {
    pub fn new(
        bind_group_layouts: &[&wgpu::BindGroupLayout],
        push_constant_ranges: &[wgpu::PushConstantRange],
        device: &wgpu::Device,
        shader: &wgpu::ShaderModule,
        label: &str,
        entry_point: &str,
    ) -> Self {
        let pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
            bind_group_layouts,
            push_constant_ranges,
            label: Some(label),
        });
        let pipeline = device.create_compute_pipeline(&wgpu::ComputePipelineDescriptor {
            entry_point: Some(entry_point),
            layout: Some(&pipeline_layout),
            module: shader,
            compilation_options: Default::default(),
            cache: None,
            label: Some(label),
        });

        Self { pipeline }
    }
    pub fn compute<'a>(
        &'a self,
        encoder: &'a mut wgpu::CommandEncoder,
        label: &str,
        bind_groups: &[&wgpu::BindGroup],
        x: u32,
        y: u32,
    ) {
        let mut pass = encoder.begin_compute_pass(&wgpu::ComputePassDescriptor {
            timestamp_writes: None,
            label: Some(label),
        });

        pass.set_pipeline(&self.pipeline);
        for (i, bind_group) in bind_groups.iter().enumerate() {
            pass.set_bind_group(i as _, *bind_group, &[]);
        }
        pass.dispatch_workgroups(x, y, 1);
    }
}

```

3.4.3.5. fft.rs

```rust
use super::SimData;
use crate::{cast_slice, engine::{scene::Scene, util::Texture}, WG_SIZE};
use shared::FFTData;
use std::mem;

pub struct FourierTransform {
    h_ifft: PipelineFFT,
    v_ifft: PipelineFFT,
    permute: PipelineFFT,
    pingpong1: Texture,
}

impl FourierTransform {
    pub fn new(
        device: &wgpu::Device,
        shader: &wgpu::ShaderModule,
        scene: &Scene,
        simdata: &SimData,
    ) -> Self {
        let pingpong1 = Texture::new_storage(
            scene.consts.sim.size,
            scene.consts.sim.size,
            wgpu::TextureFormat::Rgba32Float,
            device,
            "PingPong 1",
        );
        let bind_group_layouts = &[&simdata.layout, &pingpong1.layout, &pingpong1.layout]; // layout is same for 0 and 1
        let push_constant_ranges = &[wgpu::PushConstantRange {
            stages: wgpu::ShaderStages::COMPUTE,
            range: 0..mem::size_of::<FFTData>() as u32,
        }];

        let h_ifft = PipelineFFT::new(
            bind_group_layouts,
            push_constant_ranges,
            device,
            shader,
            "H-Step IFFT",
            "sim::fft::hstep_ifft",
        );
        let v_ifft = PipelineFFT::new(
            bind_group_layouts,
            push_constant_ranges,
            device,
            shader,
            "V-Step IFFT",
            "sim::fft::vstep_ifft",
        );
        let permute = PipelineFFT::new(
            bind_group_layouts,
            push_constant_ranges,
            device,
            shader,
            "Permute",
            "sim::fft::permute",
```

```
56          );
57
58          Self {
59              h_ifft,
60              v_ifft,
61              permute,
62              pingpong1,
63          }
64      }
65
66      pub fn ifft2d<'a>(
67          &'a self,
68          encoder: &'a mut wgpu::CommandEncoder,
69          scene: &Scene,
70          simdata: &SimData,
71          pingpong0: &Texture,
72          index: u32,
73      ) {
74          let bind_groups = &[
75              &simdata.bind_group,
76              &pingpong0.bind_group,
77              &self.pingpong1.bind_group,
78          ];
79          let wg_size = scene.consts.sim.size / WG_SIZE;
80          let mut data = FFTData {
81              stage: 0,
82              pingpong: 0,
83          };
84
85          for stage in 0..scene.consts.sim.size.ilog2() {
86              data.stage = stage;
87              self.h_ifft.compute(
88                  encoder,
89                  bind_groups,
90                  cast_slice(&[data]),
91                  &format!("H-Step {}, {}", stage, index),
92                  wg_size,
93                  wg_size,
94              );
95              data.pingpong = (data.pingpong + 1) % 2;
96          }
97
98          for stage in 0..scene.consts.sim.size.ilog2() {
99              data.stage = stage;
100             self.v_ifft.compute(
101                 encoder,
102                 bind_groups,
103                 cast_slice(&[data]),
104                 &format!("V-Step {}, {}", stage, index),
105                 wg_size,
106                 wg_size,
107             );
108             data.pingpong = (data.pingpong + 1) % 2;
109         }
110
111         self.permute.compute(
112             encoder,
113             bind_groups,
```

```
            cast_slice(&[data]),
            &format!("Permute {}", index),
            wg_size,
            wg_size,
        );
    }
}

// abstraction of the pipeline is just to make code significantly nicer
pub struct PipelineFFT {
    pipeline: wgpu::ComputePipeline,
}

impl PipelineFFT {
    pub fn new(
        bind_group_layouts: &[&wgpu::BindGroupLayout],
        push_constant_ranges: &[wgpu::PushConstantRange],
        device: &wgpu::Device,
        shader: &wgpu::ShaderModule,
        label: &str,
        entry_point: &str,
    ) -> Self {
        let pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
            bind_group_layouts,
            push_constant_ranges,
            label: Some(label),
        });
        let pipeline = device.create_compute_pipeline(&wgpu::ComputePipelineDescriptor {
            entry_point: Some(entry_point),
            layout: Some(&pipeline_layout),
            module: shader,
            compilation_options: Default::default(),
            cache: None,
            label: Some(label),
        });
        Self { pipeline }
    }

    pub fn compute<'a>(
        &'a self,
        encoder: &'a mut wgpu::CommandEncoder,
        bind_groups: &[&wgpu::BindGroup],
        push_constants: &[u8],
        label: &str,
        x: u32,
        y: u32,
    ) {
        let mut pass = encoder.begin_compute_pass(&wgpu::ComputePassDescriptor {
            timestamp_writes: None,
            label: Some(label),
        });

        pass.set_pipeline(&self.pipeline);
        for (i, bind_group) in bind_groups.iter().enumerate() {
            pass.set_bind_group(i as _, *bind_group, &[]);
        }
        pass.set_push_constants(0, push_constants);
        pass.dispatch_workgroups(x, y, 1);
```

```
172     }
173 }
174
```

3.4.4. shaders

3.4.4.1. lib.rs

```rust
#![no_std]
pub mod sim;
pub mod ui;
pub mod skybox;

use core::f32::consts;
use core::ops::{Add, Mul};

use spirv_std::glam::{Vec4, Vec3, UVec2, Vec2};
use spirv_std::image::Image2d;
use spirv_std::Sampler;
use spirv_std::{spirv, image::Image};
use spirv_std::num_traits::Float;
use shared::Constants;

type StorageImage = Image!(2D, format = rgba32f, sampled = false);

#[spirv(vertex)]
pub fn main_vs(
    pos: Vec4,
    uv: UVec2,
    #[spirv(instance_index)] instance_index: u32,
    #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
    #[spirv(descriptor_set = 3, binding = 3)] displacement_map0: &StorageImage,
    #[spirv(descriptor_set = 3, binding = 4)] normal_map0: &StorageImage,
    #[spirv(descriptor_set = 3, binding = 5)] foam_map0: &StorageImage,
    #[spirv(descriptor_set = 4, binding = 3)] displacement_map1: &StorageImage,
    #[spirv(descriptor_set = 4, binding = 4)] normal_map1: &StorageImage,
    #[spirv(descriptor_set = 4, binding = 5)] foam_map1: &StorageImage,
    #[spirv(descriptor_set = 5, binding = 3)] displacement_map2: &StorageImage,
    #[spirv(descriptor_set = 5, binding = 4)] normal_map2: &StorageImage,
    #[spirv(descriptor_set = 5, binding = 5)] foam_map2: &StorageImage,
    #[spirv(position)] out_pos: &mut Vec4, out_normal: &mut Vec3,
    out_foam: &mut Vec3,
    out_world_pos: &mut Vec4,
) {
    let mut displacement = displacement_map0.read(uv) * consts.sim.lengthscale0_sf;
    displacement += displacement_map1.read(uv) * consts.sim.lengthscale1_sf;
    displacement += displacement_map2.read(uv) * consts.sim.lengthscale2_sf;
    let mut normal = normal_map0.read(uv) * consts.sim.lengthscale0_sf;
    normal += normal_map1.read(uv) * consts.sim.lengthscale1_sf;
    normal += normal_map2.read(uv) * consts.sim.lengthscale2_sf;
    normal.y = 1.0;
    let mut foam = foam_map0.read(uv) * consts.sim.lengthscale0_sf;
    foam += foam_map1.read(uv) * consts.sim.lengthscale1_sf;
    foam += foam_map2.read(uv) * consts.sim.lengthscale2_sf;

    let width = consts.sim.size as f32 * consts.sim.mesh_step;
    let x = instance_index % consts.sim.instances;
    let z  = instance_index / consts.sim.instances;
    let tiling_offset = Vec4::new(x as f32 * width, 0.0, z as f32 * width, 0.0) *
consts.sim.instance_micro_offset;
    let positive_offset = Vec4::new(width * 0.5, 0.0, width * 0.5, 0.0) * (consts.sim.instances as
f32 - 1.0);
```

```
53
54      let centring_offset = Vec4::new(0.5 * width, consts.sim.height_offset, 0.5 * width, 0.0);
55
56      let mut resultant_pos = pos + displacement - centring_offset + tiling_offset -
    positive_offset;
57      resultant_pos.w = 1.0;
58      *out_pos = consts.camera_viewproj * resultant_pos;
59      *out_normal = normal.truncate();
60      *out_foam = foam.truncate();
61      *out_world_pos = resultant_pos;
62  }
63
64  #[inline(never)]
65  #[spirv(fragment)]
66  pub fn main_fs(
67      normal: Vec3,
68      foam: Vec3,
69      world_pos: Vec4,
70      #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
71      #[spirv(descriptor_set = 1, binding = 0)] sampler: &Sampler,
72      #[spirv(descriptor_set = 2, binding = 0)] hdri: &Image2d,
73      output: &mut Vec4,
74      ) {
75      let pos = world_pos.truncate();
76      let n = normal.normalize();
77      let l = (consts.shader.light.truncate() - pos).normalize();
78      let v = (consts.eye.truncate() - pos).normalize();
79      let h = (l + v).normalize();
80
81      let dist = (consts.eye - world_pos).length();
82      let max_dist = (consts.eye - 0.5 * consts.sim.size as f32 * consts.sim.mesh_step *
    consts.sim.instances as f32).length();
83      let t = ((dist - consts.shader.fog_offset) / (max_dist - consts.shader.fog_offset)).clamp(0.0,
    1.0);
84      let fog = t.powf(consts.shader.fog_falloff) * consts.shader.fog_density;
85
86      let foam = foam.x.max(0.0).min(1.0);
87
88      let roughness = consts.shader.roughness + foam * consts.shader.foam_roughness;
89
90      let fresnel = fresnel(n, v, &consts) * consts.shader.fresnel_sf;
91      let l_scatter = subsurface_scattering(l, v, n, pos.y, roughness, consts);
92      let l_env_reflected = hdri.sample(*sampler, equirectangular_to_uv(reflect(n, v))).truncate() *
    consts.shader.reflection_sf;
93      let l_specular = match consts.shader.pbr {
94          1 => pbr_specular(l, h, n, v, consts, roughness) * consts.shader.pbr_sf,
95          _ => blinn_phong(n, h, consts) * fresnel,
96      };
97      let l_eye = lerp(
98          (1.0 - fresnel) * l_scatter + l_specular + fresnel * l_env_reflected,
99          consts.shader.foam_color.truncate(),
100         foam,
101     );
102     let l_eye = lerp(
103         l_eye,
104         consts.shader.fog_color.truncate(),
105         fog,
106     );
```

```
107
108     *output = reinhard_tonemap(l_eye).extend(1.0);
109 }
110
111 fn fresnel(n: Vec3, v: Vec3, consts: &Constants) -> f32 {
112     let fresnel_n = Vec3::new(n.x * consts.shader.fresnel_normal_sf, n.y, n.z *
    consts.shader.fresnel_normal_sf); let f0 = ((consts.shader.air_ri - consts.shader.water_ri)
113         / (consts.shader.air_ri + consts.shader.water_ri)).powf(2.0);
114     f0 + (1.0 - f0) * (1.0 - fresnel_n.dot(v)).powf(consts.shader.fresnel_shine)
115 }
116
117 fn subsurface_scattering(l: Vec3, v: Vec3, n: Vec3, height: f32, roughness: f32, consts:
    &Constants) -> Vec3 {
118     let height_factor = consts.shader.ss_height * height.max(0.0) * l.dot(-v).max(0.0).powf(4.0)
119         * (0.5 - 0.5 * l.dot(n)).powf(3.0);
120     let reflection_factor = consts.shader.ss_reflected * v.dot(n).max(0.0).powf(2.0);
121     let lambert_factor = consts.shader.ss_lambert * l.dot(n).max(0.0) *
    consts.shader.scatter_color.truncate() * consts.shader.sun_color.truncate();
122     let ambient_factor = consts.shader.ss_ambient * consts.shader.bubble_density *
    consts.shader.bubble_color.truncate() * consts.shader.sun_color.truncate();
123
124     ((height_factor + reflection_factor) * consts.shader.scatter_color.truncate() *
    consts.shader.sun_color.truncate()) / (1.0 + lambda_ggx(roughness))
125         + lambert_factor + ambient_factor
126 }
127
128 fn pbr_specular(l: Vec3, h: Vec3, n: Vec3, v: Vec3, consts: &Constants, roughness: f32) -> Vec3 {
129     consts.shader.sun_color.truncate() * microfacet_brdf(l, h, n, v, consts, roughness)
130 }
131
132 fn microfacet_brdf(l: Vec3, h: Vec3, n: Vec3, v: Vec3, consts: &Constants, roughness: f32) -> f32
    {
133     let f = fresnel(h, v, consts) * consts.shader.fresnel_pbr_sf;
134     let g = smith_g2(h, l, v, roughness);
135     let d = ggx(n, h, roughness);
136     let div = (4.0 * n.dot(l) * n.dot(v)).max(consts.shader.pbr_cutoff);
137     f * g * d / div
138 }
139
140 fn ggx(n: Vec3, h: Vec3, roughness: f32) -> f32 {
141     let nh = n.dot(h).max(0.0001).min(0.9999);
142     roughness * roughness / (consts::PI *
143     ((roughness * roughness - 1.0) * nh.powf(2.0) + 1.0).powf(2.0))
144 }
145
146 fn smith_g2(h: Vec3, l: Vec3, v: Vec3, roughness: f32) -> f32 {
147     1.0 / (1.0 + smith_g1(h, l, roughness) + smith_g1(h, v, roughness))
148 }
149
150 fn smith_g1(h: Vec3, s: Vec3, roughness: f32) -> f32 {
151     let alpha = roughness * roughness;
152     let hs = h.dot(s);
153     let a = hs / (alpha * (1.0 - hs * hs).sqrt());
154     1.0 / (1.0 + lambda_ggx(a))
155 }
156
157 fn lambda_ggx(a: f32) -> f32 {
158     ((1.0 + 1.0 / (a * a)).sqrt() - 1.0) * 0.5
```

```rust
}

fn lerp<T: Add<Output = T> + Mul<f32, Output = T>>(a: T, b: T, t: f32) -> T {
    a * (1.0 - t) + b * t
}

fn blinn_phong(n: Vec3, h: Vec3, consts: &Constants) -> Vec3 {
    n.dot(h).max(0.0).powf(consts.shader.shininess) * consts.shader.sun_color.truncate()
}

fn reflect(n: Vec3, v: Vec3) -> Vec3 {
    2.0 * (n * n.dot(v)) - v
}

fn equirectangular_to_uv(v: Vec3) -> Vec2 {
    Vec2::new(
        (v.z.atan2(v.x) + consts::PI) / consts::TAU,
        v.y.acos() / consts::PI,
    )
}

fn reinhard_tonemap(c: Vec3) -> Vec3 {
    c / (c + Vec3::splat(1.0))
}

```

3.4.4.2. skybox.rs

```rust
use spirv_std::glam::{Vec4,  Vec2, Vec3};
use spirv_std::{spirv,image::Image2d, Sampler};
use spirv_std::num_traits::Float;
use shared::Constants;
use crate::{equirectangular_to_uv, reinhard_tonemap, lerp};

#[inline(never)]
#[spirv(vertex)]
pub fn skybox_vs(
    #[spirv(vertex_index)] vertex_index: u32,
    #[spirv(position)] out_pos: &mut Vec4,
) {
    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);
}

#[inline(never)]
#[spirv(fragment)]
pub fn skybox_fs(
    #[spirv(frag_coord)] frag_coord: Vec4,  // Get screen-space fragment coordinates
    #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
    #[spirv(descriptor_set = 1, binding = 0)] hdri: &Image2d,
    #[spirv(descriptor_set = 2, binding = 0)] sampler: &Sampler,
    out_color: &mut Vec4,
) {
    let proj_inverse = consts.shader.proj_mat.inverse();
    let view_inverse = consts.shader.view_mat.inverse();

    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();

    let h = (ray_dir.y / consts.eye.normalize().y).clamp(0.0, 1.0);
    let fog = (-h * consts.shader.fog_density - consts.shader.fog_height).exp();

    let sky_col = reinhard_tonemap(hdri.sample(
        *sampler,
        equirectangular_to_uv(ray_dir),
    ).truncate()).extend(1.0);

    let sky_col = sky_col + dist_to_sun(ray_dir, &consts) *
consts.shader.sun_color.truncate().extend(1.0);
    *out_color = lerp(
        sky_col,
        consts.shader.fog_color,
        fog,
    );
```

```
56 }
57
58 fn dist_to_sun(ray: Vec3, consts: &Constants) -> f32 {
59     let dot = ray.dot(consts.shader.light.truncate().normalize());
60     let cos = consts.shader.sun_size.cos();
61     (dot - cos).max(0.0) * consts.shader.sun_falloff
62 }
63
```

3.4.4.3. ui.rs

```rust
use spirv_std::glam::{Vec4,  Vec2};
use spirv_std::{spirv,image::Image2d, Sampler};
use shared::Constants;

#[spirv(vertex)]
pub fn ui_vs(
    pos: Vec2,
    uv: Vec2,
    col: Vec4,
    #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
    #[spirv(position)] out_pos: &mut Vec4,
    out_uv: &mut Vec2,
    out_col: &mut Vec4,
) {
    *out_pos = Vec4::new(
        2.0 * pos.x / consts.width - 1.0,
        1.0 - 2.0 * pos.y / consts.height,
        0.0,
        1.0,
    );
    *out_uv = uv;
    *out_col = col;
}

#[spirv(fragment)]
pub fn ui_fs(
    uv: Vec2,
    col: Vec4,
    #[spirv(descriptor_set = 1, binding = 0)] tex: &Image2d,
    #[spirv(descriptor_set = 2, binding = 0)] sampler: &Sampler,
    out_col: &mut Vec4,

) {
    *out_col = tex.sample(*sampler, uv) * col.powf(1.2);
}

```

3.4.4.4. sim/mod.rs

```rust
pub mod initial_spectra;
pub mod evolve_spectra;
pub mod fft;
pub mod process_deltas;

```

3.4.4.4.1. evolve_spectra.rs

```rust
use spirv_std::{
    spirv,
    num_traits::Float,
};
use spirv_std::glam::{UVec3, Vec3Swizzles, Vec2, Vec4, Vec4Swizzles};
use shared::Constants;
use crate::StorageImage;

#[spirv(compute(threads(8,8)))]
pub fn main(
    #[spirv(global_invocation_id)] id: UVec3,
    #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
    #[spirv(descriptor_set = 1, binding = 0)] wave_tex: &StorageImage,
    #[spirv(descriptor_set = 1, binding = 1)] initial_spectrum_tex: &StorageImage,
    #[spirv(descriptor_set = 2, binding = 0)] h_displacement: &StorageImage,
    #[spirv(descriptor_set = 3, binding = 0)] v_displacement: &StorageImage,
    #[spirv(descriptor_set = 4, binding = 0)] h_slope: &StorageImage,
    #[spirv(descriptor_set = 5, binding = 0)] jacobian: &StorageImage,
) {
    // Evolving spectra
    let wave = wave_tex.read(id.xy());
    let spectrum = initial_spectrum_tex.read(id.xy());
    let h0 = spectrum.xy();
    let h0c = spectrum.zw();
    let phase = wave.w * consts.time;
    let exponent = euler(phase);
    let negative_exponent = Vec2::new(exponent.x, -exponent.y);

    // Precalculating Amplitudes
    let h = complex_mult(h0, exponent) + complex_mult(h0c, negative_exponent);
    let ih = Vec2::new(-h.y, h.x);

    // Displacement in x,y,z
    let dx = -ih * wave.x * wave.z;
    let dy = h;
    let dz = -ih * wave.y * wave.z;

    // normal
    let nx = ih * wave.x;
    let nz = ih * wave.y;

    // jacobian
    let j_xx = -h * wave.x * wave.x * wave.z;
    let j_zz = -h * wave.y * wave.y * wave.z;
    let j_xz = -h * wave.x * wave.y * wave.z;

    unsafe {
        h_displacement.write(id.xy(), Vec4::new(dx.x, dx.y, dz.x, dz.y));
        v_displacement.write(id.xy(), Vec4::new(dy.x, dy.y, j_xz.x, j_xz.y));
        h_slope.write(id.xy(), Vec4::new(nx.x, nx.y, nz.x, nz.y));
        jacobian.write(id.xy(), Vec4::new(j_xx.x, j_xx.y, j_zz.x, j_zz.y));
    }
}

pub fn complex_mult(a: Vec2, b: Vec2) -> Vec2 {
    Vec2::new(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x)
```

```rust
}

pub fn euler(exp: f32) -> Vec2 {
    Vec2::new(exp.cos(), exp.sin())
}

pub fn complex_exp(a: Vec2) -> Vec2 {
    Vec2::new(a.y.cos(), a.y.sin()) * a.x.exp()
}

```

3.4.4.4.2. fft.rs

```rust
use spirv_std::{
    spirv,
    num_traits::Float,
};
use core::f32::consts;
use crate::{sim::evolve_spectra::complex_mult, StorageImage};
use shared::{Constants, FFTData};
use spirv_std::glam::{UVec3, UVec2, Vec3Swizzles, Vec2, Vec4, Vec4Swizzles};


#[spirv(compute(threads(8,8)))]
pub fn hstep_ifft(
    #[spirv(global_invocation_id)] id: UVec3,
    #[spirv(push_constant)] data: &FFTData,
    #[spirv(descriptor_set = 0, binding = 1)] butterfly_tex: &StorageImage,
    #[spirv(descriptor_set = 1, binding = 0)] pingpong0: &StorageImage,
    #[spirv(descriptor_set = 2, binding = 0)] pingpong1: &StorageImage,
) {
    let butterfly_data: Vec4 = butterfly_tex.read(UVec2::new(data.stage, id.x));
    let twiddle: Vec2 = butterfly_data.xy();
    let indices: UVec2 = UVec2::new(butterfly_data.z as u32, butterfly_data.w as u32);

    if data.pingpong == 0 {
        let top_signal0 = pingpong0.read(UVec2::new(indices.x, id.y)).xy();
        let top_signal1 = pingpong0.read(UVec2::new(indices.x, id.y)).zw();
        let bottom_signal0 = pingpong0.read(UVec2::new(indices.y, id.y)).xy();
        let bottom_signal1 = pingpong0.read(UVec2::new(indices.y, id.y)).zw();

        let h0 = top_signal0 + complex_mult(twiddle, bottom_signal0);
        let h1 = top_signal1 + complex_mult(twiddle, bottom_signal1);

        unsafe {
            pingpong1.write(id.xy(), Vec4::new(h0.x, h0.y, h1.x, h1.y));
        }
    } else if data.pingpong == 1 {
        let top_signal0 = pingpong1.read(UVec2::new(indices.x, id.y)).xy();
        let top_signal1 = pingpong1.read(UVec2::new(indices.x, id.y)).zw();
        let bottom_signal0 = pingpong1.read(UVec2::new(indices.y, id.y)).xy();
        let bottom_signal1 = pingpong1.read(UVec2::new(indices.y, id.y)).zw();

        let h0 = top_signal0 + complex_mult(twiddle, bottom_signal0);
        let h1 = top_signal1 + complex_mult(twiddle, bottom_signal1);

        unsafe {
            pingpong0.write(id.xy(), Vec4::new(h0.x, h0.y, h1.x, h1.y));
        }
    }
}

#[spirv(compute(threads(8,8)))]
pub fn vstep_ifft(
    #[spirv(global_invocation_id)] id: UVec3,
    #[spirv(push_constant)] data: &FFTData,
    #[spirv(descriptor_set = 0, binding = 1)] butterfly_tex: &StorageImage,
    #[spirv(descriptor_set = 1, binding = 0)] pingpong0: &StorageImage,
    #[spirv(descriptor_set = 2, binding = 0)] pingpong1: &StorageImage,
```

```
57 ) {
58     let butterfly_data: Vec4 = butterfly_tex.read(UVec2::new(data.stage, id.y));
59     let twiddle: Vec2 = butterfly_data.xy();
60     let indices: UVec2 = UVec2::new(butterfly_data.z as u32, butterfly_data.w as u32);
61 
62     if data.pingpong == 0 {
63         let top_signal0 = pingpong0.read(UVec2::new(id.x, indices.x)).xy();
64         let top_signal1 = pingpong0.read(UVec2::new(id.x, indices.x)).zw();
65         let bottom_signal0 = pingpong0.read(UVec2::new(id.x, indices.y)).xy();
66         let bottom_signal1 = pingpong0.read(UVec2::new(id.x, indices.y)).zw();
67 
68         let h0 = top_signal0 + complex_mult(twiddle, bottom_signal0);
69         let h1 = top_signal1 + complex_mult(twiddle, bottom_signal1);
70 
71         unsafe {
72             pingpong1.write(id.xy(), Vec4::new(h0.x, h0.y, h1.x, h1.y));
73         }
74     } else if data.pingpong == 1 {
75         let top_signal0 = pingpong1.read(UVec2::new(id.x, indices.x)).xy();
76         let top_signal1 = pingpong1.read(UVec2::new(id.x, indices.x)).zw();
77         let bottom_signal0 = pingpong1.read(UVec2::new(id.x, indices.y)).xy();
78         let bottom_signal1 = pingpong1.read(UVec2::new(id.x, indices.y)).zw();
79 
80         let h0 = top_signal0 + complex_mult(twiddle, bottom_signal0);
81         let h1 = top_signal1 + complex_mult(twiddle, bottom_signal1);
82 
83         unsafe {
84             pingpong0.write(id.xy(), Vec4::new(h0.x, h0.y, h1.x, h1.y));
85         }
86     }
87 }
88 
89 #[spirv(compute(threads(8,8)))]
90 pub fn permute(
91     #[spirv(global_invocation_id)] id: UVec3,
92     #[spirv(descriptor_set = 1, binding = 0)] pingpong0: &StorageImage,
93 ) {
94     let sign = match ((id.x + id.y) % 2) as f32 {
95         0.0 => 1.0,
96         _ => -1.0,
97     };
98 
99     let h0 = sign * pingpong0.read(id.xy()).x;
100     let h1 = sign * pingpong0.read(id.xy()).z;
101     unsafe {
102         pingpong0.write(id.xy(), Vec4::new(h0, h1, 0.0, 1.0));
103     }
104 }
105 
106 #[spirv(compute(threads(1,8)))]
107 pub fn precompute_butterfly(
108     #[spirv(global_invocation_id)] id: UVec3,
109     #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
110     #[spirv(descriptor_set = 1, binding = 1)] butterfly_tex: &StorageImage,
111 ) {
112     let k = (id.y as f32 * consts.sim.size as f32 / 2.0_f32.powf(id.x as f32 + 1.0)) %
    consts.sim.size as f32;
113     let exp = -2.0 * consts::PI * k / consts.sim.size as f32;
```

```
114     let twiddle = Vec2::new(exp.cos(), exp.sin());
115 
116     let step = 2.0_f32.powf(id.x as f32);
117     let wing = id.y as f32 % 2.0_f32.powf(id.x as f32 + 1.0) < step;
118 
119     let mut yt: u32 = id.y;
120     let mut yb: u32 = id.y;
121 
122     if id.x == 0 {
123         if wing {
124             yb += 1;
125         } else {
126             yt -= 1;
127         }
128         yt = bit_reverse(yt, consts.sim.logsize);
129         yb = bit_reverse(yb, consts.sim.logsize);
130     } else {
131         if wing {
132             yb += step as u32;
133         } else {
134             yt -= step as u32;
135         }
136     }
137 
138     unsafe {
139         butterfly_tex.write(id.xy(), Vec4::new(twiddle.x, twiddle.y, yt as f32, yb as f32));
140     }
141 }
142 
143 // algorithm from https://stackoverflow.com/questions/746171/efficient-algorithm-for-bit-reversal-from-msb-lsb-to-lsb-msb-in-c
144 fn bit_reverse(mut x: u32, size: u32) -> u32 {
145     let mut n: u32 = 0;
146     let mask: u32 = 0x1;
147     for _ in 0..size {
148         n <<= 1;
149         n |= x & mask;
150         x >>= 1;
151     }
152     n
153 }
154 
```

3.4.4.4.3. initial_spectra.rs

rust

```rust
use spirv_std::{
    spirv,
    num_traits::Float,
};
use crate::StorageImage;
use core::f32::consts::{self, PI};
use spirv_std::glam::{UVec3, UVec2, Vec3Swizzles, Vec2, Vec4, Vec4Swizzles};
use shared::{Constants, SimConstants};

#[spirv(compute(threads(8,8)))]
pub fn main(
#[spirv(global_invocation_id)] id: UVec3,
#[spirv(push_constant)] cascade: &u32,
#[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
#[spirv(descriptor_set = 1, binding = 0)] gaussian_tex: &StorageImage,
#[spirv(descriptor_set = 2, binding = 0)] wave_tex: &StorageImage,
#[spirv(descriptor_set = 2, binding = 1)] spectrum_tex: &StorageImage
) {
    let lengthscale = match cascade {
        0 => consts.sim.lengthscale0,
        1 => consts.sim.lengthscale1,
        _ => consts.sim.lengthscale2,
    } as f32;
    let cutoff_high = match cascade {
        0 => consts.sim.cutoff_high0,
        1 => consts.sim.cutoff_high1,
        _ => consts.sim.cutoff_high2,
    };
    let cutoff_low = match cascade {
        0 => consts.sim.cutoff_low0,
        1 => consts.sim.cutoff_low1,
        _ => consts.sim.cutoff_low2,
    };

    let dk: f32 = 2.0 * consts::PI / lengthscale;
    let n = id.x as f32 - 0.5 *  consts.sim.size as f32;
    let m = id.y as f32 - 0.5 *  consts.sim.size as f32;
    let k: Vec2 = Vec2::new(n, m) * dk;
    let k_length = k.length();

    if k_length <= cutoff_high && k_length >= cutoff_low {
        let theta = angle(k, consts.sim.wind_offset);
        let omega = dispersion_relation(k_length, &consts.sim);
        let domega_dk = dispersion_derivative(k_length, &consts.sim); //Derivative
        let omega_peak = 22.0 * ((consts.sim.gravity * consts.sim.gravity) /
(consts.sim.wind_speed * consts.sim.fetch)).powf(1.0 / 3.0);
        let jonswap = jonswap(omega, omega_peak, &consts.sim);
        let depth_attenuation = depth_attenuation(omega, &consts.sim);
        let tma = jonswap * depth_attenuation;
        let spread = final_spread(omega, omega_peak, theta, &consts);
        let spectrum = 2.0 * tma * spread * domega_dk.abs() * dk * dk / k_length;
        let h0 = 1.0 / 2.0_f32.sqrt() * gaussian_tex.read(id.xy()).xy() * spectrum.sqrt();

        unsafe {
            wave_tex.write(id.xy(), Vec4::new(k.x, k.y, 1.0 / k_length, omega));
            spectrum_tex.write(id.xy(), Vec4::new(h0.x, h0.y, 0.0, 1.0));
```

```
        }
    } else {
        unsafe {
            wave_tex.write(id.xy(), Vec4::new(k.x, k.y, 0.0, 1.0));
            spectrum_tex.write(id.xy(), Vec4::ZERO);
        }
    }
}

fn dispersion_relation(k: f32, consts: &SimConstants) -> f32 {
    (consts.gravity * k * (k * consts.depth).min(20.0).tanh()).sqrt()
}

fn dispersion_derivative(k: f32, consts: &SimConstants) -> f32 {
    let tanh = (consts.depth * k).min(20.0).tanh();
    let sech = 1.0 / (consts.depth * k).cosh();
    (consts.gravity * (tanh + consts.depth * k * sech * sech)) / (2.0 * (consts.gravity * k *
tanh).sqrt())
}

// from biebras TODO: credit
fn angle(k: Vec2, offset: f32) -> f32 {
    let mut angle: f32 = (k.y).atan2(k.x) - offset;
    angle = fmod(angle + PI, 2.0 * PI);
    if angle < 0.0 {
        angle += 2.0 * PI;
    }
    angle - PI
}
fn fmod(a: f32, b: f32) -> f32 {
    a - b * (a / b).floor()
}

fn jonswap(omega: f32,omega_p: f32, consts: &SimConstants) -> f32 {
    let sigma: f32;
    if omega <= omega_p {
        sigma = 0.07;
    } else {
        sigma = 0.09;
    }
    let alpha = 0.076 * (
        (consts.wind_speed * consts.wind_speed)
        / (consts.fetch * consts.gravity)
    ).powf(0.22);
    let r = (
        -1.0 * (omega - omega_p) * (omega - omega_p)
        / (2.0 * omega_p * omega_p * sigma * sigma)
    ).exp();
    alpha * consts.gravity * consts.gravity /
    (omega * omega * omega * omega * omega) * (-consts.beta * (omega_p / omega).powf(4.0)).exp() *
consts.gamma.powf(r)
}

fn depth_attenuation(omega: f32, consts: &SimConstants) -> f32 {
    let omega_h = omega * (consts.depth / consts.gravity).sqrt();
    if omega_h <= 1.0 {
        0.5 * omega_h * omega_h
    } else if omega_h < 2.0 {
```

```
        1.0 - 0.5 * (2.0 - omega_h) * (2.0 - omega_h)
    } else {
        1.0
    }
}

fn donelan_banner(omega: f32,omega_p: f32, theta: f32) -> f32 {
    let k = omega / omega_p; // arbitrary shorthand
    let beta_s: f32;
    if k < 0.95 {
        beta_s = 2.61 * k.abs().powf(1.3);
    } else if k <= 1.6 && k >= 0.95 {
        beta_s = 2.28 * k.abs().powf(-1.3);
    } else {
        beta_s = 10.0_f32.powf(-0.4 + 0.8393 * (-0.567 * (k * k).ln()).exp())
    }
    let sech = 1.0 / (beta_s * theta).cosh();
    beta_s / (2.0 * (beta_s * PI).tanh()) * sech * sech
}

fn directional_spread(omega: f32, omega_p: f32, theta: f32, consts: &Constants) -> f32 {
    let base = donelan_banner(omega, omega_p, theta);
    let swell = d_epsilon(omega, omega_p, theta, consts);
    base * swell
}

fn final_spread(omega: f32, omega_p: f32, theta: f32, consts: &Constants) -> f32 {
    let spread = directional_spread(omega, omega_p, theta, consts);
    let integral = integral(omega_p, omega, consts);
    spread * integral
}

fn integral(omega_p: f32, omega: f32, consts: &Constants) -> f32 {
    let mut sum = 0.0;
    let steps = 2.0 * PI / consts.sim.integration_step;

    for i in 0..steps as usize {
        let angle = i as f32 * consts.sim.integration_step - PI;
        sum += directional_spread(omega, omega_p, angle, consts) * consts.sim.integration_step;
    }
    1.0 / sum
}

fn d_epsilon(omega: f32, omega_p: f32, theta: f32, consts: &Constants) -> f32 {
    let s = 16.0 * (omega_p / omega).tanh() * consts.sim.swell * consts.sim.swell;
    normalisation_factor(s) * (theta / 2.0).cos().abs().powf(2.0 * s)
}

fn normalisation_factor(s: f32) -> f32 {
    let s2 = s * s;
    let s3 = s2 * s;
    let s4 = s3 * s;

    if s < 5.0 {
        -0.000564 * s4 + 0.00776 * s3 - 0.044 * s2 + 0.192 * s + 0.163
    } else {
        -4.80e-08 * s4 + 1.07e-05 * s3 - 9.53e-04 * s2 + 5.90e-02 * s + 3.93e-01
    }
```

```rust
}

// TODO: explained in biebras -> explain
#[spirv(compute(threads(8,8)))]
pub fn pack_conjugates(
    #[spirv(global_invocation_id)] id: UVec3,
    #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
    #[spirv(descriptor_set = 2, binding = 1)] spectrum_tex: &StorageImage,
) {
    let h0 = spectrum_tex.read(id.xy());
    let h0c = spectrum_tex.read(UVec2::new(
        (consts.sim.size - id.x) % consts.sim.size,
        (consts.sim.size - id.y) % consts.sim.size
    )).xy();
    unsafe {
        spectrum_tex.write(id.xy(), Vec4::new(h0.x, h0.y, h0c.x, -h0c.y));
    }
}

```

3.4.4.4.4. process_deltas.rs

rust

```rust
use spirv_std::{
    spirv,
    num_traits::Float,
};
use spirv_std::glam::{UVec3, Vec3Swizzles, Vec3, Vec4};
use shared::Constants;
use crate::StorageImage;

#[spirv(compute(threads(8,8)))]
pub fn main(
    #[spirv(global_invocation_id)] id: UVec3,
    #[spirv(uniform, descriptor_set = 0, binding = 0)] consts: &Constants,
    #[spirv(descriptor_set = 1, binding = 0)] h_displacement: &StorageImage,
    #[spirv(descriptor_set = 2, binding = 0)] v_displacement: &StorageImage,
    #[spirv(descriptor_set = 3, binding = 0)] h_slope: &StorageImage,
    #[spirv(descriptor_set = 4, binding = 0)] jacobian: &StorageImage,
    #[spirv(descriptor_set = 5, binding = 3)] displacement_map: &StorageImage,
    #[spirv(descriptor_set = 5, binding = 4)] normal_map: &StorageImage,
    #[spirv(descriptor_set = 5, binding = 5)] foam_map: &StorageImage,
) {
    let dy = v_displacement.read(id.xy()).x;
    let dx = h_displacement.read(id.xy()).x;
    let dz = h_displacement.read(id.xy()).y;
    let displacement = Vec4::new(dx * consts.sim.choppiness, dy, dz * consts.sim.choppiness, 1.0);

    let nx = h_slope.read(id.xy()).x;
    let nz = h_slope.read(id.xy()).y;
    let normal = Vec3::new(-nx, 1.0, -nz).normalize().extend(1.0);

    let jxx = 1.0_f32 + consts.sim.choppiness * jacobian.read(id.xy()).x;
    let jzz = 1.0_f32 + consts.sim.choppiness * jacobian.read(id.xy()).y;
    let jxz = consts.sim.choppiness * v_displacement.read(id.xy()).y;

    let jacobian = consts.sim.foam_bias - (jxx * jzz - jxz * jxz);
    let mut accumulation = foam_map.read(id.xy()).x * (-consts.sim.foam_decay).exp();
    if jacobian >= consts.sim.injection_threshold {
        accumulation += jacobian * consts.sim.injection_amount;
    }
    let foam = Vec3::splat(accumulation).extend(1.0);

    unsafe {
        displacement_map.write(id.xy(), displacement);
        normal_map.write(id.xy(), normal);
        foam_map.write(id.xy(), foam);
    }
}

```

3.4.5. build.rs

```rust
use spirv_builder::{MetadataPrintout, SpirvBuilder};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    SpirvBuilder::new("shaders", "spirv-unknown-spv1.5")
        .capability(spirv_builder::Capability::StorageImageExtendedFormats)
        //.capability(spirv_builder::Capability::StoragePushConstant8)
        .print_metadata(MetadataPrintout::Full)
        .build()?;
    Ok(())
}

```

3.4.6. shared/lib.rs

```rust
#![no_std]

use core::f32;
use glam::{Vec4, Mat4};

#[repr(C)]
#[derive(Clone, Copy, PartialEq, Debug)]
pub struct Constants {
    pub time: f32,
    pub deltatime: f32,
    pub width: f32,
    pub height: f32,
    pub camera_viewproj: Mat4,
    pub eye: Vec4,
    pub shader: ShaderConstants,
    pub sim: SimConstants,
}

#[derive(Clone, Copy, PartialEq, Debug)]
pub struct ShaderConstants {
    pub light: Vec4,
    pub sun_x: f32,
    pub sun_y: f32,
    pub sun_z: f32,
    pub sun_angle: f32,
    pub sun_distance: f32,
    pub foam_color: Vec4,
    pub water_ri: f32,
    pub air_ri: f32,
    pub roughness: f32,
    pub foam_roughness: f32,
    pub ss_height: f32,
    pub ss_reflected: f32,
    pub ss_lambert: f32,
    pub ss_ambient: f32,
    pub bubble_density: f32,
    pub bubble_color: Vec4,
    pub scatter_color: Vec4,
    pub sun_color: Vec4,
    pub shininess: f32,
    pub pbr: u32,
    pub reflection_sf: f32,
    pub view_mat: Mat4,
    pub proj_mat: Mat4,
    pub fresnel_sf: f32,
    pub fresnel_pbr_sf: f32,
    pub pbr_sf: f32,
    pub fresnel_normal_sf: f32,
    pub fresnel_shine: f32,
    pub sun_size: f32,
    pub sun_falloff: f32,
    pub pbr_cutoff: f32,
    pub fog_density: f32,
    pub fog_color: Vec4,
    pub fog_offset: f32,
    pub fog_falloff: f32,
```

```
57      pub fog_height: f32,
58  }
59  impl Default for ShaderConstants {
60      fn default() -> Self {
61          Self {
62              light: Vec4::new(0.0,0.0,0.0,1.0),
63              sun_x: 1.0,
64              sun_y: 0.08,
65              sun_z: 0.034,
66              sun_angle: 0.0,
67              sun_distance: 100.0,
68              foam_color: Vec4::new(0.79,0.92,0.96, 1.0),
69              water_ri: 1.33,
70              air_ri: 1.003,
71              roughness: 0.05,
72              foam_roughness: 0.1,
73              ss_height: 0.76,
74              ss_reflected: 1.0,
75              ss_lambert: 1.0,
76              ss_ambient: 0.75,
77              bubble_density: 0.35,
78              bubble_color: Vec4::new(0.0, 0.15, 0.15, 1.0),
79              scatter_color: Vec4::new(0.04, 0.06, 0.14, 1.0),
80              sun_color: Vec4::new(1.0, 0.8, 0.55, 1.0),
81              shininess: 50.0,
82              pbr: 1,
83              reflection_sf: 1.0,
84              view_mat: Mat4::ZERO,
85              proj_mat: Mat4::ZERO,
86              fresnel_sf: 1.0,
87              fresnel_pbr_sf: 1.0,
88              pbr_sf: 1.0,
89              fresnel_normal_sf: 0.60,
90              fresnel_shine: 4.5,
91              sun_size: 0.02,
92              sun_falloff: 5000.0,
93              pbr_cutoff: 0.1,
94              fog_color: Vec4::new(0.9,0.9,0.9,1.0),
95              fog_density: 2.47,
96              fog_offset: 24.5,
97              fog_falloff: 3.54,
98              fog_height: 2.04,
99          }
100     }
101 }
102
103 #[derive(Clone, Copy, PartialEq, Debug)]
104 pub struct SimConstants {
105     pub size: u32,
106     pub lengthscale0: u32,
107     pub cutoff_low0: f32,
108     pub cutoff_high0: f32,
109     pub lengthscale1: u32,
110     pub cutoff_low1: f32,
111     pub cutoff_high1: f32,
112     pub lengthscale2: u32,
113     pub cutoff_low2: f32,
114     pub cutoff_high2: f32,
```

```
    pub lengthscale0_sf: f32,
    pub lengthscale1_sf: f32,
    pub lengthscale2_sf: f32,
    pub mesh_step: f32,
    pub standard_deviation: f32,
    pub mean: f32,
    pub depth: f32,
    pub gravity: f32,
    pub beta: f32,
    pub gamma: f32,
    pub wind_speed: f32,
    pub wind_offset: f32,
    pub fetch: f32,
    pub choppiness: f32,
    pub logsize: u32,
    pub swell: f32,
    pub integration_step: f32,
    pub foam_bias: f32,
    pub foam_decay: f32,
    pub injection_threshold: f32,
    pub injection_amount: f32,
    pub height_offset: f32,
    pub instances: u32,
    pub instance_micro_offset: f32,
    pub seed: u32,
}
impl Default for SimConstants {
    fn default() -> Self {
        // Defining simulation resolution, cannot be updated at runtime so defined here
        let size = 256;
        // Defining seed for gaussian number
        let seed = 1;
        Self {
            depth: 500.0,
            size,
            // Calm Ocean
            //
            //lengthscale0: 40,
            //cutoff_low0: 0.00000001,
            //cutoff_high0: 1.0,
            //lengthscale0_sf: 1.0,
            //lengthscale1: 106,
            //cutoff_low1: 1.0,
            //cutoff_high1: 2.0,
            //lengthscale1_sf: 1.0,
            //lengthscale2: 180,
            //cutoff_low2: 2.0,
            //cutoff_high2: 999.0,
            //lengthscale2_sf: 1.0,
            //wind_speed: 5.0,
            //fetch: 4000.0,
            //choppiness: 0.2,

            // Choppy Ocean
            //
            lengthscale0: 20,
            cutoff_low0: 0.00000001,
            cutoff_high0: 1.0,
```

```
173             lengthscale0_sf: 1.0,
174             lengthscale1: 124,
175             cutoff_low1: 1.0,
176             cutoff_high1: 2.0,
177             lengthscale1_sf: 1.0,
178             lengthscale2: 256,
179             cutoff_low2: 2.0,
180             cutoff_high2: 999.0,
181             lengthscale2_sf: 1.0,
182             wind_speed: 0.5,
183             fetch: 100000.0,
184             choppiness: 0.6,
185 
186             mesh_step: 0.4 * 128.0 / size as f32,
187             standard_deviation: 1.0,
188             mean: 0.0,
189             gravity: 9.81,
190             beta: 5.0 / 4.0,
191             gamma: 3.3,
192             wind_offset: f32::consts::FRAC_PI_4,
193             logsize: 0,
194             swell: 0.1,
195             integration_step: 0.01,
196             foam_bias: 0.92,
197             foam_decay: 0.3,
198             injection_threshold: -0.3,
199             injection_amount: 1.0,
200             height_offset: 4.5,
201             instances: 5,
202             instance_micro_offset: 0.99,
203             seed,
204         }
205     }
206 }
207 
208 #[derive(Clone, Copy, PartialEq, Debug)]
209 pub struct FFTData {
210     pub stage: u32,
211     pub pingpong: u32,
212 }
213 
```

3.4.7. cargo.toml

toml

```toml
[package]
name = "nea"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
winit = { version = "0.29", features = ["rwh_05"] }
wgpu = { version = "23.0.1", features = ["spirv"] }
pollster = "0.3"
log = "0.4"
glam = { version = "0.29", features = ["mint"]}
env_logger = "0.10"
image = { version = "0.24", default-features = false, features = ["png", "jpeg", "exr"]}
imgui = "0.12.0"

shared = { path = "./shared" }

[build-dependencies]
spirv-builder = { git="https://github.com/Rust-GPU/rust-gpu", default-features = false, features = ["use-installed-tools"]}

[profile.dev]
opt-level = 1

[profile.dev.package."*"]
opt-level = 3

[target.x86_64-pc-windows-msvc]
linker = "rust-lld.exe"
rustflags = ["-Zshare-generics=off"]

```

3.4.8. rust-toolchain.toml

toml

```toml
[toolchain]
channel = "nightly-2024-11-22"
components = ["rust-src", "rustc-dev", "llvm-tools"]

```

3.4.9. default.nix

nix

```nix
# from rust-gpu

let
  pkgs = import <nixpkgs> {};
in with pkgs; stdenv.mkDerivation rec {
  name = "rust-gpu";

  hardeningDisable = [ "fortify" ];

  SSL_CERT_FILE = "${cacert}/etc/ssl/certs/ca-bundle.crt";
```

```
11
12   nativeBuildInputs = [ rustup ];
13
14   LD_LIBRARY_PATH = with xorg; lib.makeLibraryPath [
15     vulkan-loader
16
17     wayland libxkbcommon
18
19     libX11 libXcursor libXi libXrandr
20   ];
21 }
22
```

4. Testing

4.1. Testing Strategy

4.2. Testing Table

5. Evaluation

5.1. Results

Figure 10: Calm Ocean. $L_0 = 40$, $L_1 = 106$, $L_2 = 180$, $U_{10} = 5$, $F = 4000$, $\lambda = 0.2$, $h = 500$

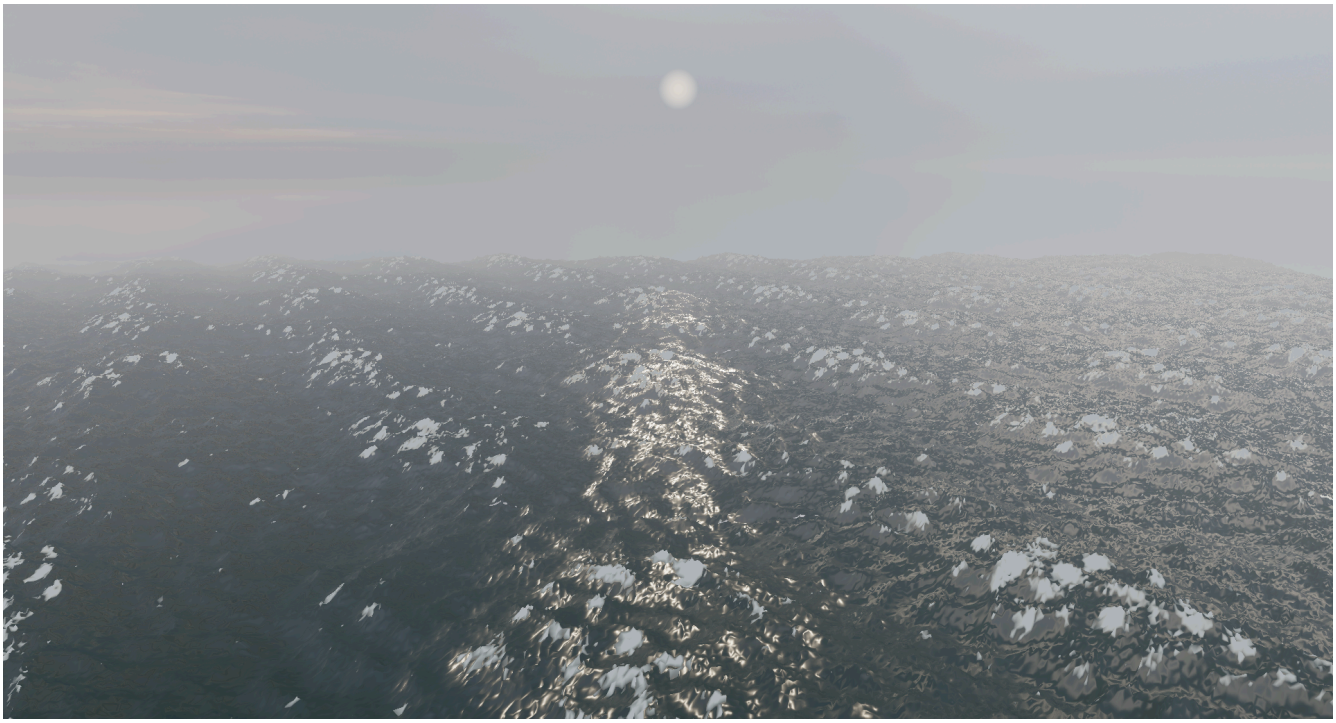


Figure 11: Choppy Ocean. $L_0 = 55$, $L_1 = 102$, $L_2 = 256$, $U_{10} = 36$, $F = 10000$, $\lambda = 0.2$, $h = 500$

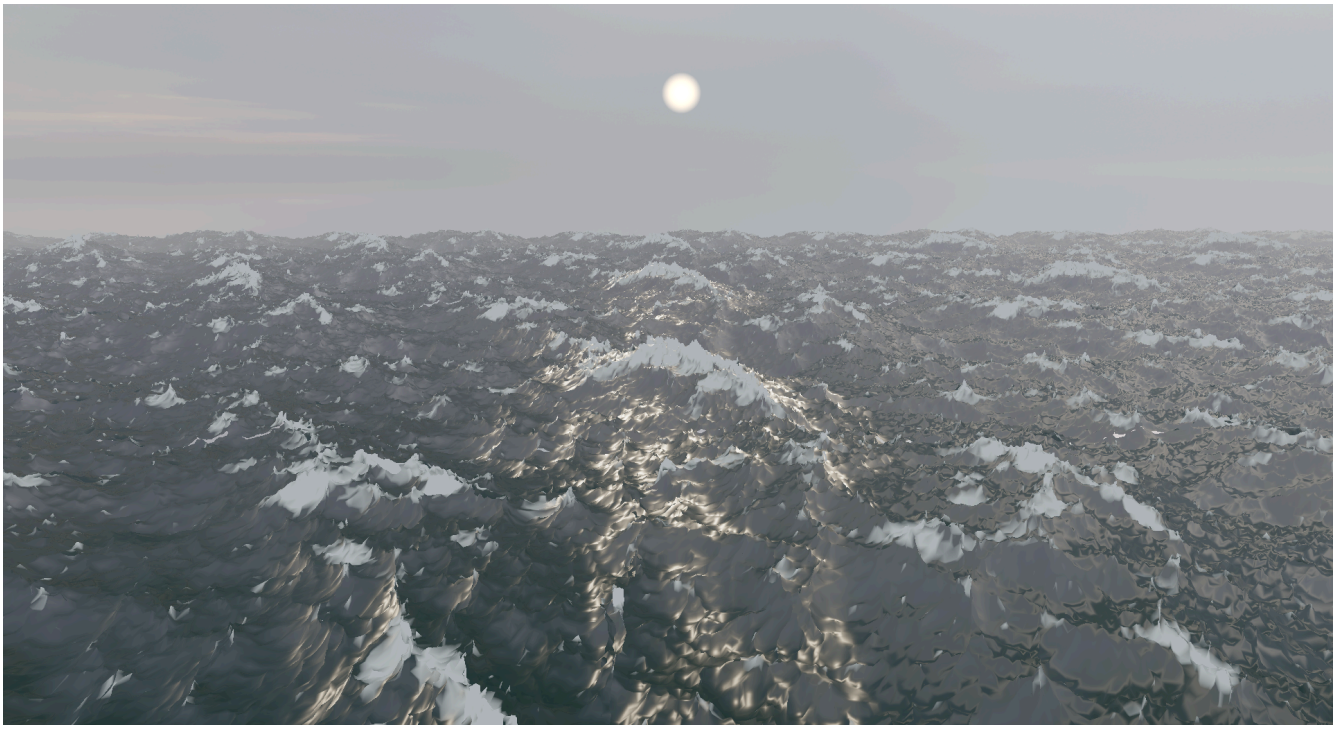


Figure 12: Stormy Ocean. $L_0 = 41$, $L_1 = 106$, $L_2 = 180$, $U_{10} = 62$, $F = 10000$, $\lambda = 0.8$, $h = 500$

Figure 13: Alternative Angle. $L_0 = 41$, $L_1 = 106$, $L_2 = 180$, $U_{10} = 62$, $F = 10000$, $\lambda = 0.8$, $h = 500$

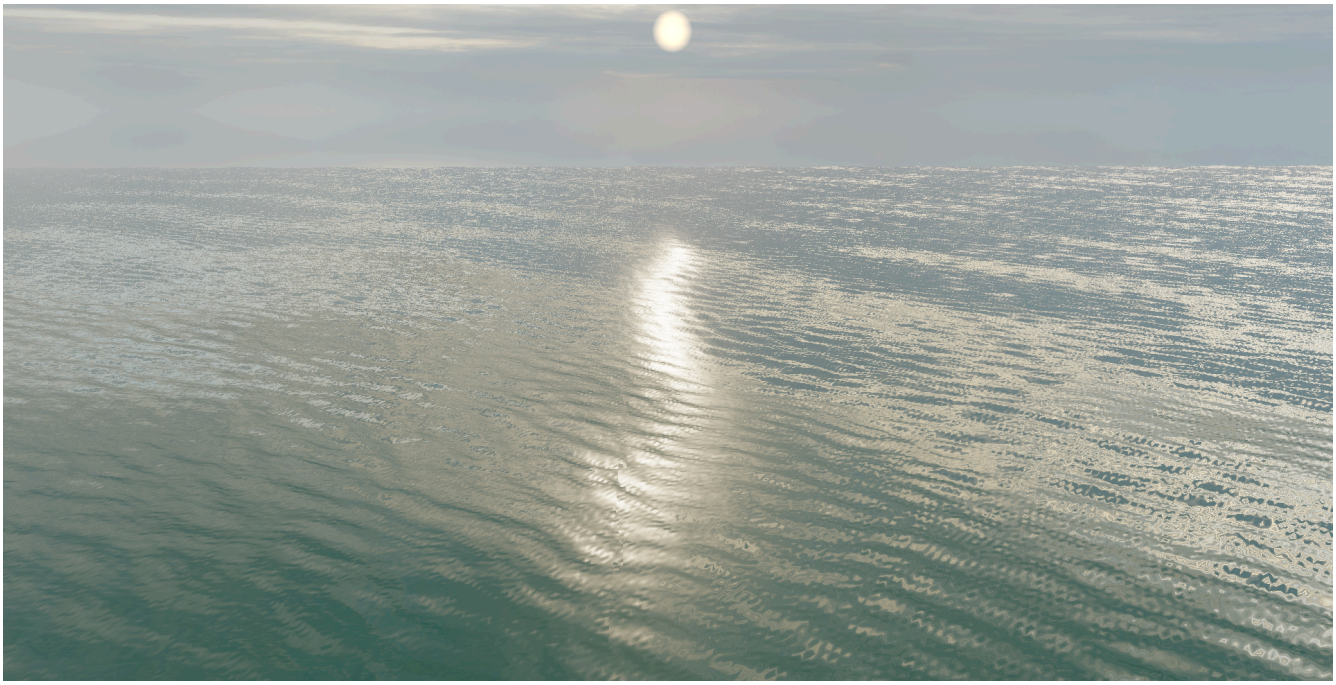


Figure 14: Early Morning Ocean. $L_0 = 20$, $L_1 = 124$, $L_2 = 256$, $U_{10} = 0.5$, $F = 100000$, $\lambda = 0.1$, $h = 30$

5.2. Evaluation Against Criteria

5.3. Client Feedback

5.4. 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] PRNG Shootout, “xoshiro / xoroshiro generators and the PRNG shootout.” Accessed: Mar. 13, 2025. [Online]. Available: https://prng.di.unimi.it/

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

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

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