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

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

1 Functionality
  1.1 “what specific ocean phenomena need to be simulated? (e.g. waves, foam, spray, currents)”
  1.2 “what parameters of the simulation need to be configurable?”
  1.3 “does there need to be an accompanying GUI?”
2 Visuals
  2.1 “do i need to implement an atmosphere / skybox?”
  2.2 “do i need to implement a pbr water shader?”
  2.3 “do i need to implement caustics, reflections, or other light-related phenomena?”
3 Technologies
  3.1 “are there any limitations due to existing technology?”
  3.2 “does this need to interop with existing code?”
4 Scope
  4.1 “are there limitations due to the target device(s)?”
  4.2 “are there other performance intesive systems in place?”
  4.3 “is the product targeted to low / mid / high end systems?”

1.3. Interview Notes

1 Functionality
  1.1 it should simulate waves in all real world conditions and be able to generate foam, if possible simulating other phenomena would be nice.
  1.2 all necessary parameters in order to simulate real world conditions, ability to control tile size / individual wave quantity
  1.3 accompanying GUI to control parameters and tile size. GUI should output frametime, and the current state of every parameter.
2 Visuals
  2.1 a basic skybox would be nice, if possible include an atmosphere shader
  2.2 implement a PBR water shader, include a microfacet BRDF
  2.3 caustics are out of scope, implement approximate subsurface scattering, use GGX distribution in combination with brdf to simulate reflections
3 Technologies
  3.1 client has not started technical implementation of project, so is not beholden to an existing technical stack
  3.2 see response 3.1
4 Scope
  4.1 the simulation is intended to run on both x86 and arm64 devices
  4.2 see response 3.1
  4.3 the simulation is targeted towards mid to high end systems, ideally the solution would also be performant on lower end hardware

1.4. Technologies

- Rust:
  - Fast, memory efficient programming language
- wgpu:
  - Graphics library
- Rust GPU:
  - (Rust as a) shader language
- Winit:
  - cross platform window creation and event loop management library
- Dear ImGui
  - Bloat-free GUI library with minimal dependencies
- GLAM:
  - used for various linear algebra operations
- Pollster / Env Logger:
  - used to read errors, as wgpu does not log as would be standard
- Image:
  - used to read HDRIs from a file into memory for processing
- Nix:
  - Used to create a declarative, reproducible development environment

1.5. Algorithm Overview (Unfinished) [1], [2]

Below is a high-level explanation of the algorithm used in this project, primarily for providing context for the upcoming theory. It is explained in more detail in the documented design.

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

1.6. Spectrum Generation

1.6.1. 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
$\vec{x} = [x_x, x_z]$	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
λ	Choppiness factor
κ	Foam bias
μ	Foam injection threshold
ζ	Foam decay rate
I	Foam value

Table 1: Simulation Variable Definitions

1.6.2. Dispersion Relation [1], [2]

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.[2]

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

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

1.6.3. Non-Directional Spectrum (JONSWAP) [2], [3], [4], [5]

The JONSWAP energy spectrum is a more parameterised version of the Pierson-Moskowitz spectrum, and an improvement over the Philips Spectrum used in [1], 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.

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

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

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

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

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

1.6.4. Depth Attenuation Function (Approximation of Kitaiigorodskii) [2]

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 [2].

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

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

1.6.5. Directional Spread Function (Donelan-Banner) [2]

The directional spread models how waves react to wind direction [4]. This function is multiplied with the non-directional spectrum in order to produce a direction dependent spectrum [2].

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

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

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

1.6.6. Swell [2]

Swell refers to the waves which have travelled out of their generating area [2]. 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 [2], approximation taken from [4]

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

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

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

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

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

1.6.7. Directional Spectrum Function [2]

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 [1] - in order to do this we apply the following transformation [2]. Similarly, to make the function directional, we also need to multiply it by the directional spread function [2].

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

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

1.7. Ocean Geometry & Foam

All of the following is repeated for every lengthscale, with the final output values being the sum of the individual displacement, normal, and foam maps.

1.7.1. Displacements [1], [4], [5], [6], [7]

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. [1]. 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.

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

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

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

1.7.2. Derivatives [1], [4]

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.

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

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

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

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

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

1.7.3. Frequency Spectrum Function [1], [4], [5]

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.

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

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

1.7.4. Box-Muller Transform [8]

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. [1]. Derivation is from polar coordinates, by treating x and y as cartesian coordinates, more details at [8]

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

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

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

1.7.5. Foam, The Jacobian, and Decay [1], [2], [5], [6]

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 [1], we compute the determinant of the jacobian for the horizontal displacement, $D(\vec{x}, t)$.

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

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

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

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

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

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

1.7.6. Cascades (Unfinished) [1], [2], [4]

1.8. The Fourier Transform (Unfinished)

1.8.1. 2D GPGPU Cooley-Tukey Radix-2 Inverse Fast Fourier Transform (Unfinished) [1], [4], [6]

The Cooley-Tukey FFT is a common implementation of the FFT algorithm 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 summations in the order of $10^4 - 10^6$ multiple times per frame. The FFT exploits the redundancy in DFT computation in order to increase performance, but is only applicable when the following conditions are met:

- $L_x = L_z = 2^x, x \in \mathbb{Z}$
- the coordinates and wave vectors lie on a regular grid

The overarching FFT algorithm is as follows:

- kA

1.8.2. Butterfly Texture (Unfinished)

1.8.3. Butterfly Operations (Unfinished)

1.8.4. Permutation (Unfinished)

1.9. Post Processing

1.9.1. 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

1.9.2. Rendering Equation [5], [9], [10]

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 [9] 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 [9]. We also Increase the roughness in areas covered with foam for L_{specular} [9].

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

1.9.3. Normalisation & Surface Normals [1], [4], [11]

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.

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

1.9.4. Subsurface Scattering [5], [9]

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. [9]

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

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

1.9.5. Fresnel Reflectance (Schlick's Approximation) [10], [11], [12], [13]

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.

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

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

1.9.6. Blinn-Phong Specular Reflection [11], [13]

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.

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

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

1.9.7. Environment Reflections [10], [11]

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.

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

1.9.8. GGX Distribution [13], [14]

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.

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

1.9.9. Geometric Attenuation [14]

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.

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

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

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

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

1.9.10. Microfacet BRDF [5], [9], [13], [14]

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.

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

1.9.11. Reinhard Tonemapping [15]

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)

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

1.9.12. Distance Fog & Sun [10]

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.

1.10. 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 [16], and then did some trigonometry to create a basic camera controller using Winit's event loop.

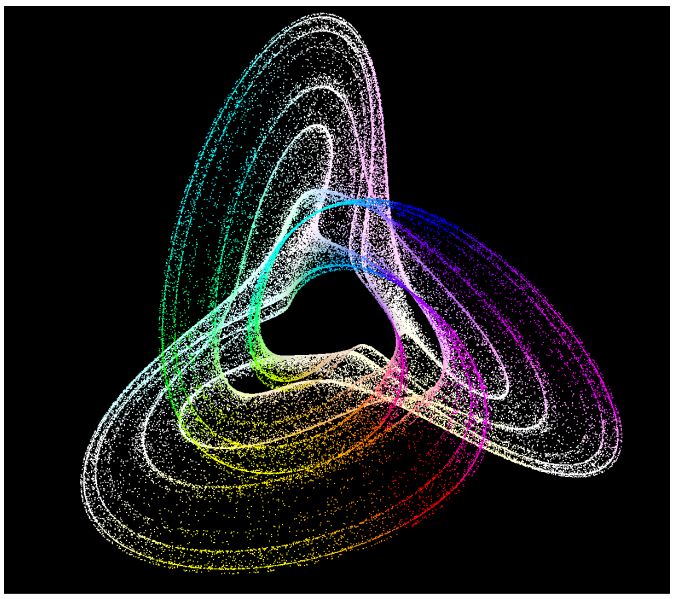

Figure 1: Found at https://github.com/CmrCrabs/chaotic-attractors

1.11. Project Considerations

The project will be split into 4 major stages - the simulation, implementing the IFFT, non PBR lighting, and PBR lighting. The simulation will most likely take the bulk of the project duration as implementing the spectrums, DFT and a GUI with just a graphics library is already a major undertaking. I will then implement the Blinn-Phong lighting model [11] in conjunction with the subsurface scattering seen in Atlas [9]. Beyond this I will implement full PBR lighting using a microfacet BRDF and statistical distribution functions in order to simulate surface microfacets.

1.12. Additional Features

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

- Swell, the waves which have travelled out of their generating area [2].
- 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)

1.13. Project Objectives (Unfinished)

1 Scene
- 1.1 Language & Environment Setup
  - 1.1.1 setup all dependencies
  - 1.1.2 have development shell to ensure correct execution
  - 1.1.3 ensure compatability for all engines
- 1.2 Window & Compatability
  - 1.2.1 ensure compatability with windows, macos & wayland (& X11?) linux
  - 1.2.2 title & respects client side rendering of respective os
- 1.3 Data Structure
  - 1.3.1 talk abt shared data structures
  - 1.3.2 create struct for all variables
  - 1.3.3 camera struct etc
- 1.4 Render Pipeline
  - 1.4.1 list steps and that it works
- 1.5 Event Loop
  - 1.5.1 able to detect mouse movement for camera inputs
  - 1.5.2 able to detect mouse down for camera inputs
  - 1.5.3 escape to close
  - 1.5.4 resize
  - 1.5.5 redraw requested

2 Simulation
- 2.1 Startup
- 2.2 On Parameter Change
- 2.3 Every Frame
- 2.4 Optimisations
  - 2.4.1 dynamic render scaling stuff

3 Rendering
- 3.1 Lighting
  - 3.1.1 calculate light / view / halfway / normal vectors
  - 3.1.2 normalise all vectors
  - 3.1.3 fresnel
  - 3.1.4 subsurface scattering
  - 3.1.5 specular reflections
    - 3.1.5.1 blinn-phong
    - 3.1.5.2 pbr
      - 3.1.5.2.1 microfacet brdf
      - 3.1.5.2.2 distribution function
      - 3.1.5.2.3 geometric attenuation
  - 3.1.6 env reflections
    - 3.1.6.1 acerola
    - 3.1.6.2 LEADR
  - 3.1.7 lerp between this and foam

2. Bibliography

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

[2] 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

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

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

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

[6] "An introduction to Realistic Ocean Rendering through FFT - Fabio Suriano - Codemotion Rome 2017." Accessed: Sep. 25, 2024. [Online]. Available: https://www.slideshare.net/slideshow/an-introduction-to-realistic-ocean-rendering-through-fft-fabio-suriano-codemotion-rome-2017/74458025

[7] "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/

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

[9] 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

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

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

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

[13] Acerola, *The Secret Behind Photorealistic And Stylized Graphics*. Accessed: Sep. 17, 2024. [OnlineVideo]. Available: https://youtu.be/KkOkx0FiHDA

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

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

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

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

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