NEA
Real-Time, Physically Based Ocean Simulation & Renderer

Zayaan Azam

Contents

1. Analysis

1.1. Abstract

// Fill Later

1.2. Client

1.2.1. 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.

1.2.2. 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 shader 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?”

1.2.3. 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 also output debug information and performance statistics
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 beckmann 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, however it would be ideal for the solution to be performant on lower end hardware

1.3. Research

1.3.1. 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
- Naga:
  - Shader translation library
- GLAM:
  - Linear algebra library
- Nix:
  - Declarative, reproducible development environment

1.3.2. Simulation Concepts, Formulae, & Algorithms

Note that throughout this project we are defining the y axis as "up".

1.3.2.1. the Statistical Wave Summation [2]–[6], [1]

For a height field of dimensions L_x and L_z, we calculate the height (h) at a position $\vec{x}$ by summating multiple sinusoids with complex, time dependant amplitudes. [5]. The frequency domain representation of the waves are converted to the spatial domain using an inverse discrete fourier transform. This is split into 2 components, with the derivatives computed seperately to find exact normals:

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

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

$$\text{Height Derivative} : \nabla h(\vec{x}, t) = \sum_{\vec{k}} i\vec{k}\hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

$$\text{Displacement Derivative} : \lambda \nabla D(\vec{x}, t) = \sum_{\vec{k}} \vec{k}\frac{\vec{k}}{\left|\vec{k}\right|} \hat{h}\Big(\vec{k}, t\Big) e^{i\vec{k}\cdot\vec{x}}$$

where

- t is the time
- $\vec{k} = [k_x, k_z]$, the wave vector, direction vector of the spectrum's texture
- $\omega\Big(\vec{k}\Big) = \sqrt{\left|\vec{k}\right| g}$ is the dispersion relation, a multiplier that determines the speed of the ocean
- $\vec{x} = [x_x, x_z]$, the direction vector for the height map for which we are summing
- $\hat{h}\Big(\vec{k}, t\Big)$ is the frequency spectrum function

- $h(\vec{x}, t)$ is the wave height at horizontal position $\vec{x}$
- $h(\vec{x}, t)$ gives the vertical displacement vector at the point x at time t
- $\vec{D}(\vec{x}, t)$ gives the horizontal displacement at $\vec{x}$ at time t, used to simulate "choppy waves". We would split the normalised vector $\vec{k}$ into its components and compute them seperately [5]
- λ is a convenient scale factor "choppiness" in order to create sharper wave peaks [5]
- $\nabla h(\vec{x}, t)$ gives the rate of change of the height, used to calculate the normal vector
- $\nabla \vec{D}(\vec{x}, t)$ gives the rate of change of the displacement, used to calculate the normal vector

1.3.2.2. The Inverse Discrete Fourier Transform (IDFT) [2], [6], [5], [1]

The IDFT can be computed using the fast fourier transform if the following conditions are met:
- $N = M = L_x = L_z$
- the coordinates & wavenumbers lie on regular grids
- $N, M, L_x, L_z = 2^x$, for any positive integer x

For implementation, the statistical wave summation is represented in terms of the indices n and m as below

where
- N, M are the number of points & waves respectively
- L_x, L_z are the horizontal dimensions of a wave tile
- n, m are the simulation domain resolutions, integers with bounds $-\frac{N}{2} \leq n < \frac{N}{2}, -\frac{M}{2} \leq m < \frac{M}{2}$
- $\vec{k} = \left[\frac{2\pi n}{L_x}, \frac{2\pi m}{L_z}\right]$
- $\vec{x} = \left[\frac{nL_x}{N}, \frac{mL_z}{M}\right]$

1.3.2.3. Frequency Spectrum Function (Unfinished) [5], [2], [4]

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.

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

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

where
- $\hat{h}$ evolves $\hat{h}_0$ through time using eulers formula. by combining a positive and negative version of the wave number you ensure the functions output is real [5]
- $\hat{h}_0$ is the initial wave state as determined by the energy spectra & gaussian distribution. This is only computed on parameter change / startup and then stored into a texture

1.3.2.4. Cooley-Tukey Fast Fourier Transform (FFT) (Unfinished) [7]

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 M (and N) in the millions.

complex, will write up after learning roots of unity & partial derivatives

1.3.2.5. JONSWAP (Joint North Sea Wave Observation Project) Spectrum [8], [2], [4]

This energy spectrum determines is where the final height is ultimately derived from. The JONSWAP energy spectrum is a more parameterised version of the Philips Spectrum used in [5], 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.

$$S(\omega) = \frac{\alpha g^2}{\omega^5} \exp\left[-\beta\left(\frac{\omega_p}{\omega}\right)^4\right]\gamma^r$$

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

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

where

- α is the intensity of the spectra
- $\beta = \frac{5}{4}$, a "shape factor", rarely changed [8]
- $\gamma = 3.3$
- $\sigma = \begin{cases} 0.07 \text{ if } \omega \leq \omega_p \\ 0.09 \text{ if } \omega > \omega_p \end{cases}$ [8]
- $\omega = 2\pi f$ where f is the wave frequency in Hz [8]
- $\omega_p = 22\left(\frac{g^2}{U_{10}F}\right)^{\frac{1}{3}}$, the peak wave frequency
- U_{10} is the wind speed at $10m$ above the sea surface [8]
- F is the distance from a lee shore (a fetch) - distance over which wind blows with constant velocity [8]
- g is gravity

1.3.2.6. Gaussian Random Numbers

The ocean exhibits gaussian variance in the possible waves. Due to this the frequency spectrum function is varied by gaussian random numbers with mean ($\tilde{x}$) 0 and standard deviation (σ) 1. These are generated in pairs and then stored into the red and green channels of a texture to be accessed.

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(x-\tilde{x}^2\right)}{2\sigma^2}}$$

where

- σ is the standard deviation
- $\tilde{x}$ is the mean
- x is a random number, $-1..1$

1.3.2.7. Surface Normals (Unfinished) [5], [2]

In order to compute the surface normals we need the derivatives of the displacement(s), the values for which are obtained from the fourier transform above.

$$\vec{N}(\vec{x},t) = \begin{bmatrix} -\nabla h_{x(\vec{x},t)} \\ 1 \\ -\nabla h_{z(\vec{x},t)} \end{bmatrix}$$

1.3.2.8. The Jacobian (Unfinished) [5], [4], [9], [1]

The jacobian describes the "uniqueness" of a transformation. This is useful as where the waves would crash, the jacobian of the displacements goes negative. We can then offset this to bias the results and generate more foam.

Will write up once I better understand

1.3.2.9. Exponential Decay (Unfinished) [9], [4], [5]

In order to dissipate the stored foam over time instead of instantaneously, we apply an exponential decay function to each pixel in the texture. This may potentially be replaced by a gaussian blur and fade pass depending on results produced.

$$N(t) = N_0 e^{-\lambda t}$$

where
- N_0 is the initial quantity
- λ is the rate constant

1.3.2.10. The Ocean Simulation Algorithm (Unfinished) [5], [2], [4], [9], [10], [1]

// like 20x more complex than this
- generate gaussian noise (4 bands)
- jonswap it based on params
- fft to frequency domain (for all 4 bands)
- evolve frequencies with time
- inverse fft to spatial domain
- postprocess results for brdf / pbr
- recombine 4 bands in vertex shader?

startup:
- generate gaussian random number pairs and store into texture on cpu

param change:
- generate energy spectrum for every wave and store into texture
- generate dispersion relation for every wave and store into texture

every frame:
- evolve spectrum
- inverse fft for all 3 axes
- inverse fft for all 5 derivatives
- store results into textures
- displace vertices per textures
- compute jacobian of textures
- inject foam into foam texture
- lighting

- color pixels for foam
- exponential decay function on foam

1.3.3. Level of Detail (LOD) Optimisations (Unfinished) [1]

1.3.4. Lighting Algorithms & Formulae

1.3.4.1. Rendering Equation [9], [4], [3]

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), and subsurface scattering [9] allowing us to replace the L_{diffuse} and L_{ambient} terms, we are able to take an analytical approach to solving this.

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].

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

where

- F is the fresnel reflectance term
- L_{scatter} is the light emitted due to subsurface scattering
- L_{specular} is the reflected light from the source
- $L_{\text{env\_reflected}}$ is the reflectioned light from the environemnt

1.3.4.2. Normalisation [11], [12]

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. Henceforth, a vector $\vec{A}$ when normalised is represented with $\hat{A}$.

$$\vec{A} \cdot \vec{B} = |A||B| \cos\theta$$

$$\hat{A} = \frac{\vec{A}}{|A|} \Rightarrow \left|\hat{A}\right| = 1$$

$$\therefore \hat{A} \cdot \hat{B} = \cos\theta$$

where

- θ is the angle between vectors A and B

1.3.4.3. Blinn-Phong Specular Reflection & Vector Definitions [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.

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

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

where

- $\widehat{H}$ is the halfway vector
- $\widehat{N}$ is the surface normal
- $\hat{V}$ is the camera view vector
- $\hat{L}$ is the light source vector
- S is the shininess of the material

1.3.4.4. Subsurface Scattering (Unfinished, rewrite vectors) [9], [4]

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 current computing power. 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]

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

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

where
- $\hat{L}, \hat{V}, \widehat{H}, \widehat{N}$ are the sun / eye / half / normal vectors respectively
- $W_{\max}$ is the $\max(0, \text{wave height})$
- k_1, k_2, k_3, k_4 are artistic parameters
- C_{ss} is the water scatter color
- C_f is the air bubbles color
- P_f is the density of air bubbles spread in water
- $\langle \omega_a, \omega_b\rangle$ is the $\max(0, (\omega_a \cdot \omega_b))$
- λ_{GGX} is the masking function defined under Smith's G_1

1.3.4.5. Fresnel Reflectance (Schlick's Approximation) [3], [11], [14], [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.

$$F\big(\widehat{N}, \hat{V}\big) = F_0 + (1 - F_0)\big(1 - \widehat{N}\cdot\hat{V}\big)^5$$

where
- $F_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$
- n_1 & n_2 are the refractive indices of the two media [14]
- $\widehat{N}$ is the normal vector
- $\hat{V}$ is the view vector (in microfacet models can also be the light vector) [14]
- if using a microfacet model, replace $\widehat{N}$ with the Halfway vector, $\widehat{H}$) [14]

1.3.4.6. Environment Reflections [3], [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's cubemap and use that color as the reflected color. This method is somewhat simplistic, and not physically based.

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

where

- $\widehat{N}$ is the normal vector for the point
- $\hat{V}$ is the camera view vector
- $\hat{R}$ is the vector that points to the point on the cubemap which we sample

1.3.4.7. Post Processing (Unfinished) [3]

To hide the imperfect horizon line we use a distance fog. This is then attenuated based oon height. In order to do this we use the depth buffer to determine the depth of each pixel and then based on that scale the light color to be closer to a defined fog color. Finally we blend a sun into the skybox based on the light position.

1.3.4.8. Color Grading (Unfinished) [3]

in order to really sell the sun being as bright as it would be on an open ocean, we apply a bloom pass to the whole image. In order to prevent it from being completely blown out we then apply a tone mapping to rebalance the colors.

1.3.5. PBR-Specific Algorithms / Formulae

1.3.5.1. Microfacet BRDF [9], [4], [15], [13]

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. $D(h)$ can be any distribution function. The geometric attenuation is a function that models how some reflections are masked / shadowed by the microfacets "geometry" and serves to counteract the fresnel.

$$L_{\text{specular}} = f_{\text{microfacet}}\left(\widehat{N}, \widehat{H}, \hat{L}, \hat{V}\right) = \frac{F\left(\widehat{N}, \widehat{H}\right) G\left(\hat{L}, \widehat{H}\right) D\left(\widehat{N}, \widehat{H}\right)}{4\left(\widehat{N} \cdot \hat{L}\right)\left(\widehat{N} \cdot \hat{V}\right)}$$

where

- $F\left(\widehat{N}, \widehat{H}\right)$ is the Fresnel Reflectance
- $D\left(\widehat{N}, \widehat{H}\right)$ is the Distribution Function
- $G\left(\hat{L}, \hat{V}, \widehat{H}\right)$ is the Geometric Attenuation Function

1.3.5.2. GGX Distribution [15], [13]

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.

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

where
- $\alpha = \text{roughness}^2$
- $\widehat{N}$ and $\widehat{H}$ are the surface normal and halfway vector respectively

1.3.5.3. Geometric Attenuation Function (Smith's G_1 Function) [15]

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.

$$G_1 = \frac{1}{1 + \lambda_{\text{GGX}}(a)}$$

$$a = \frac{\widehat{H} \cdot \hat{L}}{\alpha\sqrt{1 - \Big(\widehat{H} \cdot \hat{L}\Big)^2}}$$

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

where
- $\alpha = \text{roughness}^2$
- $\widehat{H}$ is a microfacet normal
- $\hat{L}$ is the light vector (can also be the view vector)

1.3.6. 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.

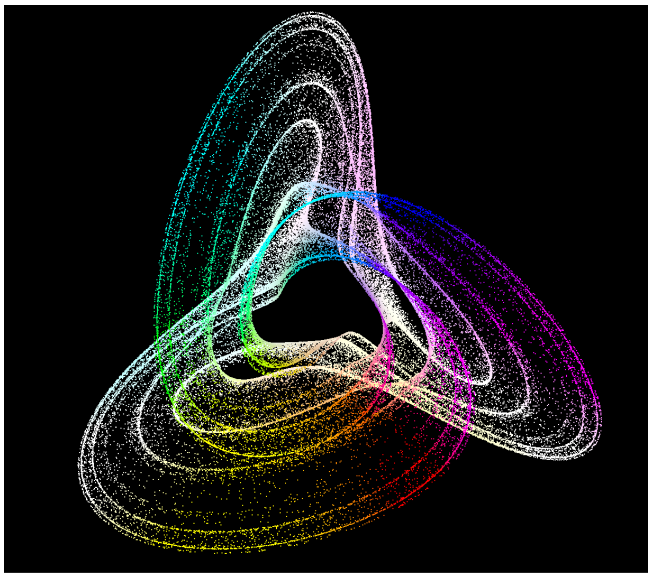

Figure 1: Found at https://github.com/CmrCrabs/chaotic-attractors

1.3.7. Project Considerations

The project will be split into 3 major stages - the simulation, non PBR lighting, and PBR lighting. The simulation will most likely take the bulk of the project duration as implementing the FFT 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.

finally, I would also like to look into implementing a sky color simulation based on sun position - as this would allow the complete simulation of a realistic day night cycle of any real world ocean conditions.

1.4. 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
  - 3.1.8 adjust roughness of areas with foam
- 3.2 Post Processing / Scene
  - 3.2.1 HDRI

2. Bibliography

[1] "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

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

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

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

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

[6] "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/

[7] Wikipedia, "Fast Fourier Transform." Accessed: Sep. 08, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Fast_fourier_transform

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

[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] siggraph, "The Technical Art of Sea of Thieves." Accessed: Sep. 14, 2024. [Online]. Available: https://history.siggraph.org/learning/the-technical-art-of-sea-of-thieves/

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

[12] Wikipedia, "Specular Highlight." Accessed: Sep. 11, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Specular_highlight

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

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

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

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

[17] Wikipedia, "Fourier Transform." Accessed: Sep. 08, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Fourier_transform