CAP 5705: Final Project Report

Davis Arthur, davisarthur@ufl.edu

December 11, 2021

1 Goals

In my original proposal, I listed a variety of goals for my project. First, I hoped to use Tessendorf's [1] FFT-based wave generation technique to generate and animate a height map and normal map representing the surface of realistic wind-driven waves. Next, I hoped to shade this surface using Snell's law and the Fresnel equations. Additionally, I planned to enhance the realism of the scene by using environment mapping and simulating water caustics and light absorption in the water volume. Finally, I intended to add keyboard and mouse controls for the user to fly around the scene. In the preceding section, I discuss my progress towards each of these goals.

2 Overview

2.1 Wave Generation and Animation

The first goal of my project was to generate a 2D height map that could be used to render the surface of wind driven waves. I chose to follow an approach proposed by Tessendorf [1] that models the surface of ocean water as a height map $y = h(\boldsymbol{x}, t)$ on top of a planar grid [1]. Tessendorf suggests generating the height map in Fourier space first using an analytical model of an ocean wave spectrum. For this project I used the Phillip's spectrum which is given by the following expression:

$$P_h(\boldsymbol{k}) = A\frac{e^{-1/(kL)^2}}{k^4}|\hat{\boldsymbol{k}} \cdot \hat{\boldsymbol{w}}|^2$$

where A is the amplitude of the spectrum, $\boldsymbol{w}$ is the wind velocity, $\boldsymbol{k}$ is the wave vector, $k = |\boldsymbol{k}|$, and $L = \boldsymbol{w} \cdot \boldsymbol{w}/g$. First, I wrote a function that computes $P_h(\boldsymbol{k})$ over a uniform grid in Fourier space. In Figure 1, I plotted the Phillip's spectrum over a 64×64 grid as a function of the spatial frequencies k_x and k_z. To generate the Phillip's spectrum shown in the figure, I used a wind vector of $\boldsymbol{w} = (2.0, 0.0, 0.0)$ and an amplitude of $A = 0.001$.

Next, I wrote a function that generates a wave spectrum using the Phillip's spectrum. Specifically, at time $t = 0$, the generated wave spectrum is given by:

$$\tilde{h}(\boldsymbol{k}, t = 0) = \frac{1}{\sqrt{2}}(\epsilon_r + i\epsilon_i)\sqrt{P_h(\boldsymbol{k})}$$

where ϵ_r and ϵ_i are independent normally distributed random variables. In Figure 2, I plotted the real and imaginary components of a wave spectrum computed using the Phillip's spectrum shown in Figure 1.

After writing the code to generate a wave spectrum, my next step was to use the Fourier transform to calculate the height map in world space. Mathematically, this transformation is described by the following equation:

$$h(\boldsymbol{x}, t) = \sum_{\boldsymbol{k}} \tilde{h}(\boldsymbol{k}, t)e^{i\boldsymbol{k}\cdot\boldsymbol{x}}$$

Evaluating this sum directly can be very computationally expensive, so computing the height map in real time requires the Fast Fourier Transform (FFT). While there are a number of existing implementations of

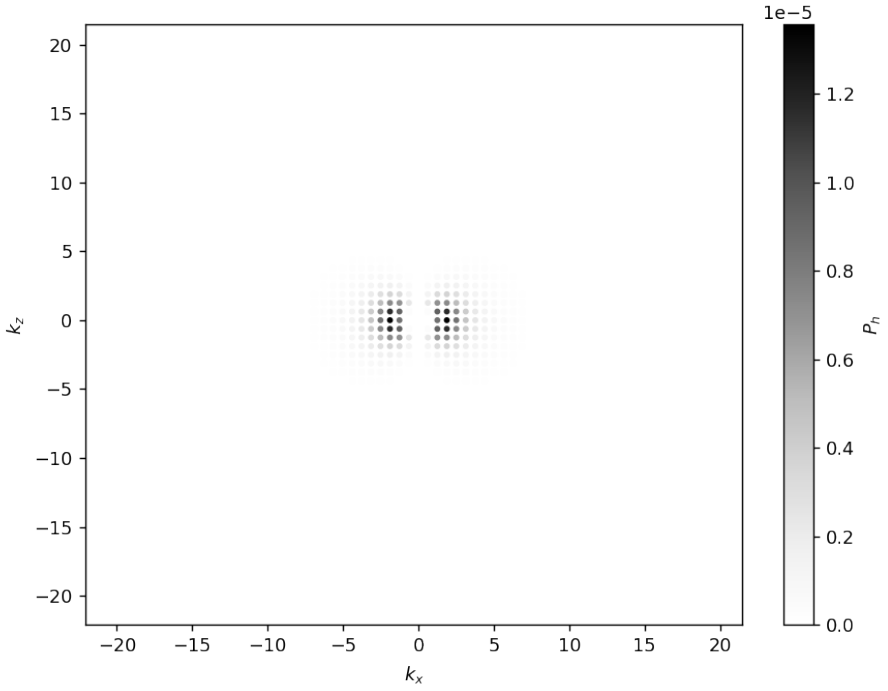

Figure 1: Phillip's spectrum P_h as a function of $\boldsymbol{k} = (k_x, k_z)$.

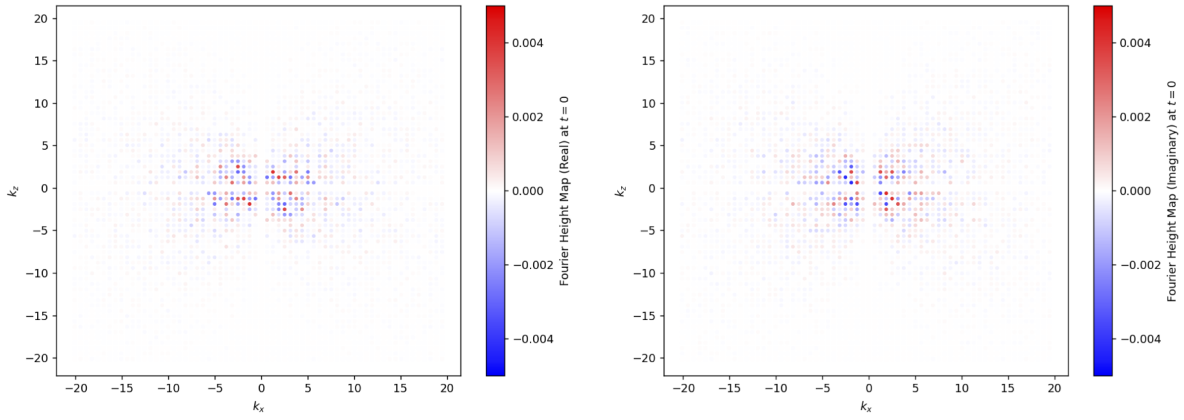

Figure 2: Real and imaginary components of a wave spectrum generated using the Phillip's spectrum illustrated in Figure 1.

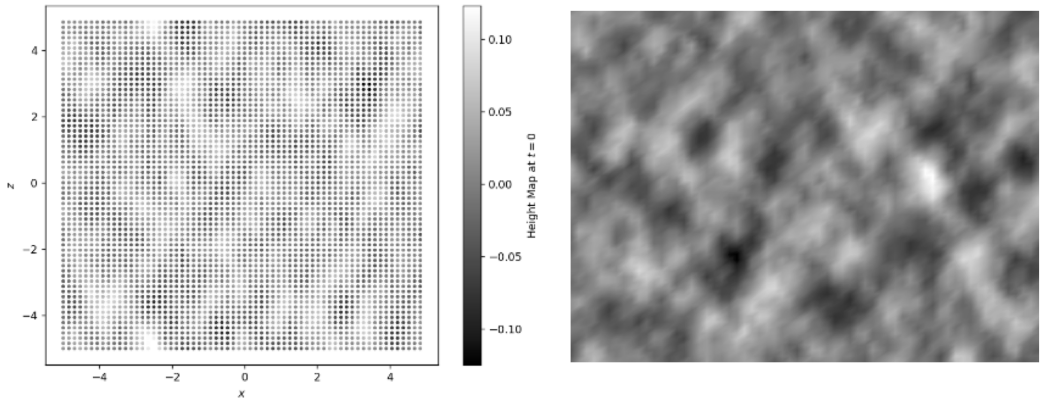

Figure 3: World space height map computed using the wave spectrum shown in Figure 2. The left image shows the depth of each vertex. The right image is rendered in OpenGL using the normalized height as the RGB color.

the FFT, I chose to implement a recursive version of the Cooley-Tukey FFT algorithm on my own as a learning exercise. My implementation is based closely on the psuedocode from the algorithm's Wikipedia page [1]. Currently, my FFT algorithm is implemented on the CPU, which unfortunately limits the size of the height map that I can use while still achieving frame rates suitable for real time applications. In practice, I found that any height map with 64×64 or fewer points worked well. In the future, I hope to move the FFT algorithm to a compute shader in order to improve the performance of my application.

It is worth noting that the Cooley-Tukey algorithm that I implemented is typically only expressed for the case of a 1D FFT. However, my wave spectrum and height map are defined over a 2D grid. Fortunately, the 2D Fourier transform can be computed efficiently by performing the 1D FFT of each row of the grid, and then performing the 1D FFT of each column of the modified grid. It is also worth mentioning that I implemented a radix 2 version of the FFT algorithm. This means that the length of the input to my FFT function must be a power of 2. In Figure 3, I used my implementation of the 2D Fast Fourier Transform to convert the wave spectrum shown in Figure 2 to a height map in world space.

Finally, I added functionality required to animate the height map as a function of time. According to Tessendorf, once the wave spectrum at $t = 0$ is known, the wave spectrum at any time t can be generated using the following expression:

$$\tilde{h}(\boldsymbol{k}, t) = \tilde{h}(\boldsymbol{k}, t = 0)e^{i\omega(k)t} + \tilde{h}^*(-\boldsymbol{k}, t = 0)e^{-i\omega(k)t}$$

where $\omega(k)$ is the frequency associated with the wave vector magnitude k by the dispersion relation:

$$\omega(k) = \sqrt{gk}$$

Using this equation, each frame I compute the wave spectrum at the new time t and then evaluate its FFT to update the height map. In my video report, I included a clip that shows how the height map evolves in time according to Tessendorf's equation. After including this functionality, I had all of the necessary components to generate a realistic wave height map and animate the height map as a function of time. In Figure 4 I provided an image of the height map rendered in 3D using wireframe mode.

2.2 Shading

My next goal was to shade the vertices of the water's surface. In order to do this, I needed to compute the normal of the surface at each vertex. Recall that the normal of a surface is equal to the gradient of the surface's implicit equation. I found a tutorial online [2] that shows the implicit equation of a surface created

[1] https://en.wikipedia.org/wiki/Cooley–Tukey_FFT_algorithm
[2] https://zero-radiance.github.io/post/surface-gradient/

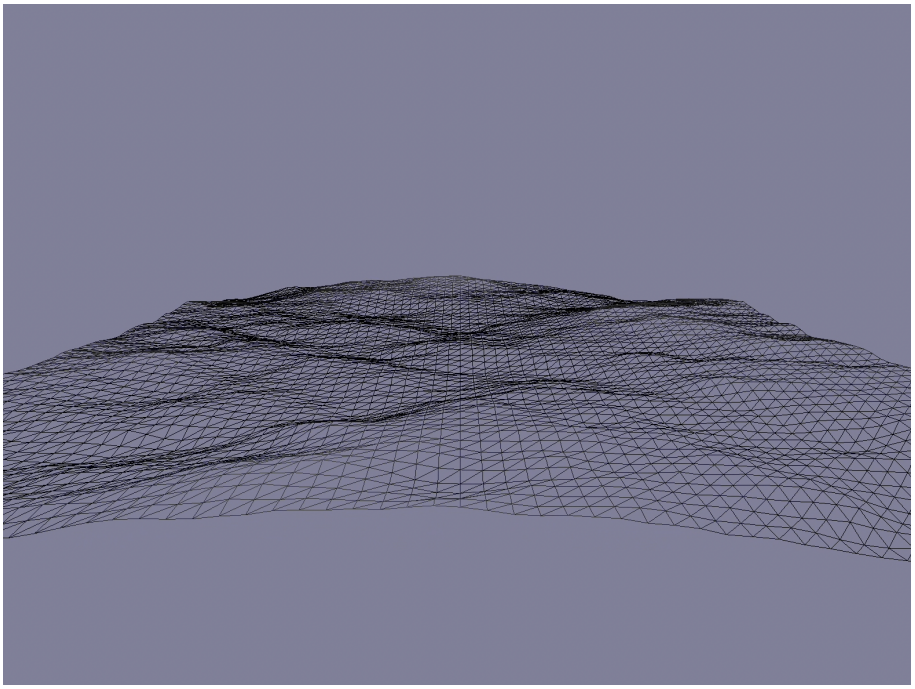

Figure 4: Height map rendered in 3D in wireframe mode.

from a height map is:

$$f(x, y, z) = y - h(x, z)$$

Using this definition, the normal of the wave surface is:

$$\boldsymbol{n} = \boldsymbol{\nabla} f = (-\frac{\partial h}{\partial x}, 1.0, -\frac{\partial h}{\partial z})$$

Therefore, by computing the gradient of the height map, the normal of each vertex can be reconstructed. According to Tessendorf, the gradient of the height map in position space is given by:

$$\boldsymbol{\epsilon}(\boldsymbol{x}, t) = \boldsymbol{\nabla} h(\boldsymbol{x}, t) = \sum_{\boldsymbol{k}} i\boldsymbol{k}\tilde{h}(\boldsymbol{k}, t)e^{i\boldsymbol{k}\cdot\boldsymbol{x}} \tag{1}$$

Using this equation, I evaluate the gradient using the 2D FFT algorithm. Both the x and z coordinates of the $\boldsymbol{\epsilon}$ vector are then sent to the shader program which subsequently reconstructs the normal of each vertex. In Figure 5, I have plotted the x and z components of the gradient of the height map rendered in Figure 3.

Once the vertex positions and normals have been computed, the water can be rendered and shaded. As a first approximation, the color c of a position $\boldsymbol{p}$ on the water's surface is a superposition of the color of the ray reflected off the water ($c_{\text{reflected}}$) and the color of the ray that refracts into the water ($c_{\text{refracted}}$).

$$c = Rc_{\text{reflected}} + Tc_{\text{refracted}}$$

The coefficient of the reflected color R is known as the reflectivity, and the coefficient of the refracted color T is know as the transmissivity. These coefficients can be evaluated at each vertex using the Fresnel equations. As illustrated in Figure 6, the incidence angle θ_i is related to the refracted angle θ_t by Snell's law:

$$n_a \sin\theta_i = n_w \sin\theta_t$$

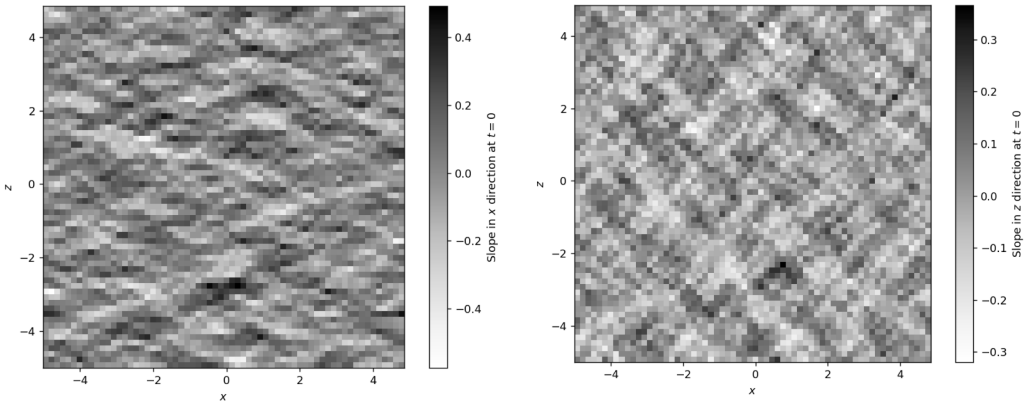

Figure 5: Slope of the height map shown in Figure 3 in the x direction (left) and z direction (right).

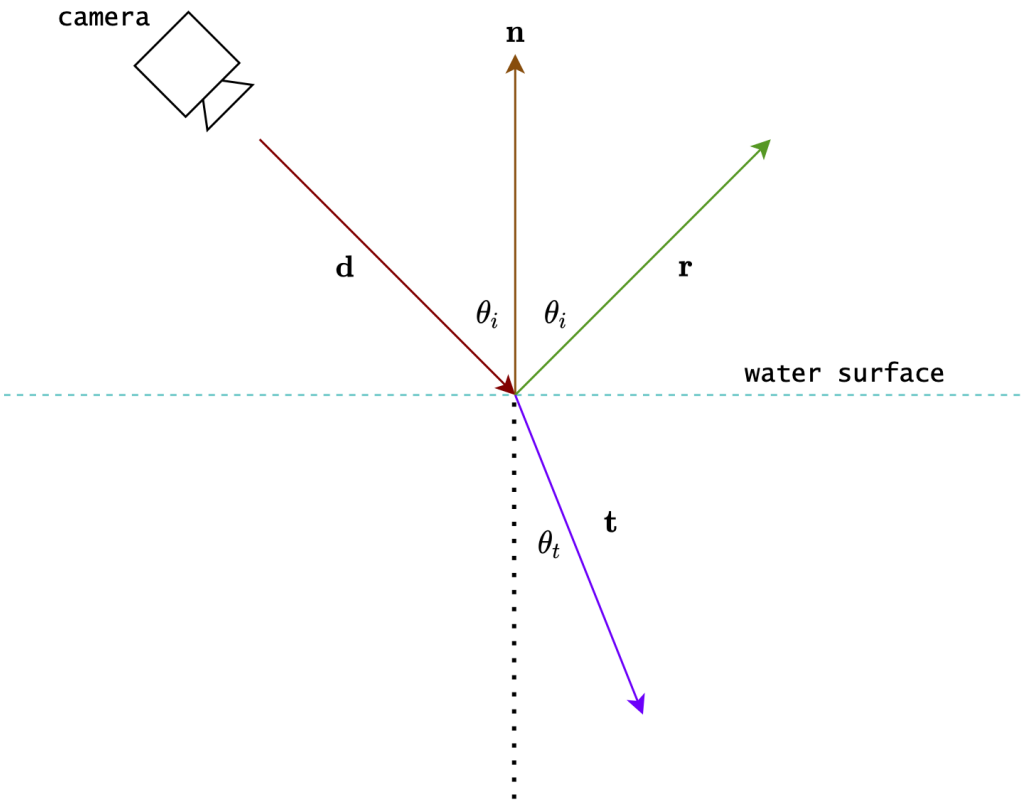

Figure 6: The incidence ray $\boldsymbol{d}$ splits into a refracted ray $\boldsymbol{t}$ and a reflected ray $\boldsymbol{r}$ upon interacting with the water surface. The surface normal is given by $\boldsymbol{n}$.

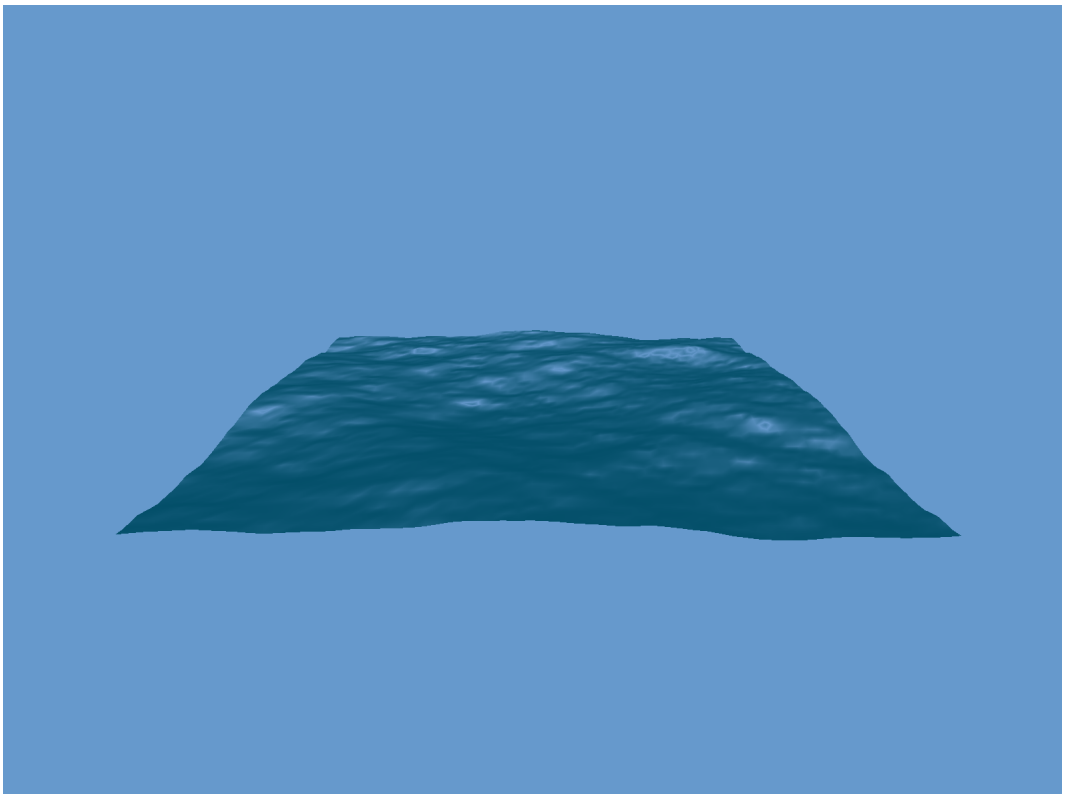

Figure 7: Rendering of an ocean wave using the Fresnel equations to determine the reflectivity and transmissivity of each vertex. Reflected and refracted colors are assumed to be constant for all fragments.

where $n_a \approx 1$ is the index of refraction of air and $n_w \approx 4/3$ is the index of refraction of water. Once the two angles have been computed, the reflectivity is given by:

$$R = \frac{1}{2}\left(\frac{\sin^2(\theta_t - \theta_i)}{\sin^2(\theta_t + \theta_i)} + \frac{\tan^2(\theta_t - \theta_i)}{\tan^2(\theta_t + \theta_i)}\right)$$

The transmissivity and the reflectivity sum to 1, so the transmissivity can be computed as

$$T = 1 - R$$

To test that the reflectivity and transmissivity were computed correctly, I wrote a simple shader to compute the reflectivity and transmissivity at each fragment. For each fragment, I set the reflected color to $c_{\text{reflected}} = (0.64, 0.84, 1.0)$ and the refracted color to $c_{\text{refracted}} = (0.0, 0.3, 0.4)$. A screenshot from this rendering is shown in Figure 7. I also included a short animation of this rendering in my video report.

Once the reflectivity and transmissivity are known, all that is left to compute is the reflected and refracted colors. To evaluate these colors, I first compute the direction of the reflected and refracted ray. Using the notation from Figure 6, the direction of the ray reflected off the surface is given by:

$$\boldsymbol{r} = \boldsymbol{d} - 2(\boldsymbol{d} \cdot \boldsymbol{n})\boldsymbol{n}$$

The direction of the refracted ray is slightly more complicated as it also depends on the indices refraction. According to Tessendorf, the direction of the refracted ray is given by:

$$\boldsymbol{t} = \frac{n_a}{n_w}\boldsymbol{d} + \Gamma\boldsymbol{n}$$

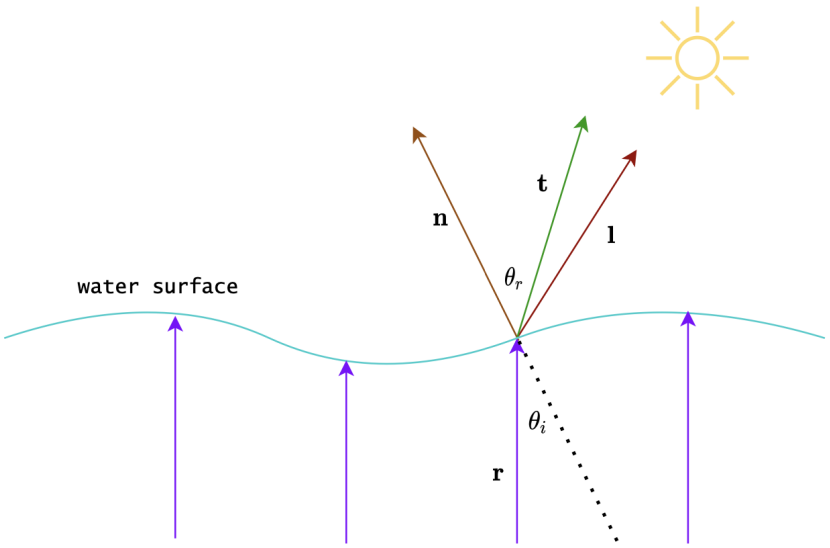

Figure 8: Schematic diagram used to evaluate caustic brightness at each vertex on the water's surface.

where Γ is

$$\Gamma = \frac{n_a}{n_w}\boldsymbol{d}\cdot\boldsymbol{n} \pm \left(1 - \left(\frac{n_a}{n_w}\right)^2 |\boldsymbol{d}\times\boldsymbol{n}|^2\right)^{1/2}$$

In the equation above, the plus sign is used when $\boldsymbol{d}\cdot\boldsymbol{n} < 0$ and the minus sign is used when $\boldsymbol{d}\cdot\boldsymbol{n} > 0$. In my implementation, I compute the direction of the reflected and refracted rays in a single line of code by calling the `reflect` and `refract` GLSL commands. Once $\boldsymbol{r}$ and $\boldsymbol{t}$ are known, $c_{\text{reflected}}$ and $c_{\text{refracted}}$ can be determined by sampling an environment map texture (more in Section 2.4).

2.3 Caustics

One of the most visually striking properties of water that is not captured by the simple model discussed above is water caustics. Water caustics are light patterns that emerge on non-uniform water surfaces due to refracted light focusing and defocusing at certain points underwater. To capture the effect of water caustics with physical accuracy, ray tracing is required. However, there are some heuristic approaches that achieve similar effects without compromising efficiency.

For caustic simulation, I chose to follow an approach proposed by Guardado in the book *GPU Gems* [2]. The basic idea of Guardado's approach is to simulate backwards ray tracing by casting an upwards vertical ray through each vertex. The refraction of this ray at the surface can be computed using the surface normal. Once the direction of the refracted ray is determined, the brightness of the caustic pattern at this vertex is computed by taking the inner product between the refracted ray and the ray pointing towards the light source. To sharpen the caustic patterns, this inner product can be raised to a higher power p. Using the notation shown in Figure 8, the caustic brightness is given by:

$$\text{Caustic brightness} = K = (\boldsymbol{t}\cdot\boldsymbol{l})^p$$

After including water caustics, the complete form of my water shading model is

$$c = Rc_{\text{reflected}} + Tc_{\text{refracted}} + Kc_{\text{caustic}}$$

where c_{caustic} is the caustic color which is typically chosen to be white.

In Figure 9, I have shaded a wave twice using caustic shading. In the left image, only caustic shading is used and in the right image, caustic shading is combined with the prior shading model.

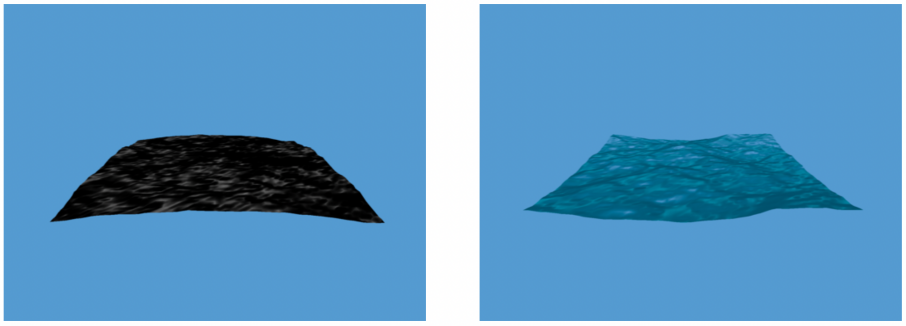

Figure 9: Wave rendered with caustic shading only (left) and complete shading model (right).

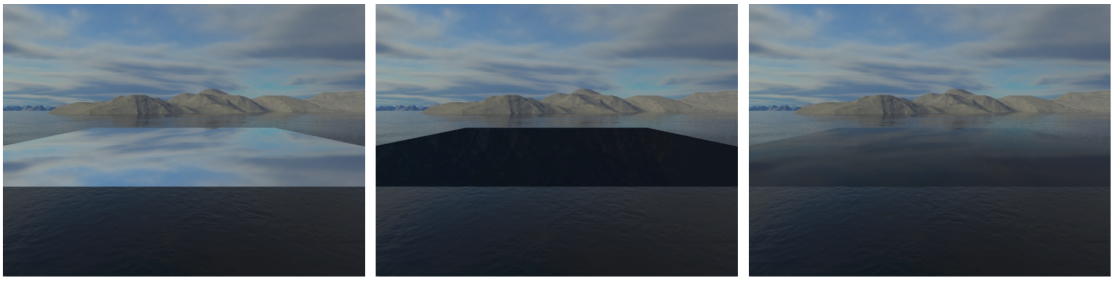

Figure 10: Shading of a flat water surface using only the reflected color (left), only the refracted color (middle), and a combination of both according to the Fresnel equations (right).

2.4 Skybox Environment Mapping

Up to this point, my shading had assumed all reflected rays were a particular color and all refracted rays were a particular color. However, in a realistic scene, the color of the sky and the color of the water volume likely varies in space. Therefore, my next goal was to use a skybox as an environment map to determine the color of the reflected and refracted rays for each fragment. I followed an online tutorial [3] to generate a skybox around my water surface grid. A skybox is essentially 6 textures wrapped in a cube around the origin of the scene. Each texture is intended to represent an image of the surrounding scene in the distance. In OpenGL, a skybox can be loaded and rendered as a GL_TEXTURE_CUBE_MAP object. Once the cube map is loaded, the texture at any point can be sampled using a 3D vector. In my shader program, I determine the refracted color by sampling the skybox with the refracted ray, and I determine the reflected color by sampling the skybox with the reflected ray.

To test that the environment mapping was performed correctly, In Figure 10, I rendered the water without waves three times. The first time, I used only the reflected color to shade the water. The second time, I used only the refracted color to shade the water. The third time, I used a superposition of both colors based on the reflectivity and transmissivity determined by the Fresnel equations. It should be noted that the middle image appears very dark because the ocean depth is modelled as a very dark blue. In Figure 11, I rendered the scene once more with animated waves. A recording of this scene is shown in my video report. In each of the renderings, it is fascinating to see the water take on the color of its surrounding scene.

2.5 Tiling the Water Surface using a Frame Buffer Object

So far, in each of my demos I have only rendered a small water mesh. However, in a realistic scene, the water should appear to reach the horizon. To achieve this effect, I first generate my typical square water mesh. Next, I generate a planar grid that appears to span to the horizon after projection. Each frame, I then perform the following procedure:

[3] https://learnopengl.com/Advanced-OpenGL/Cubemaps

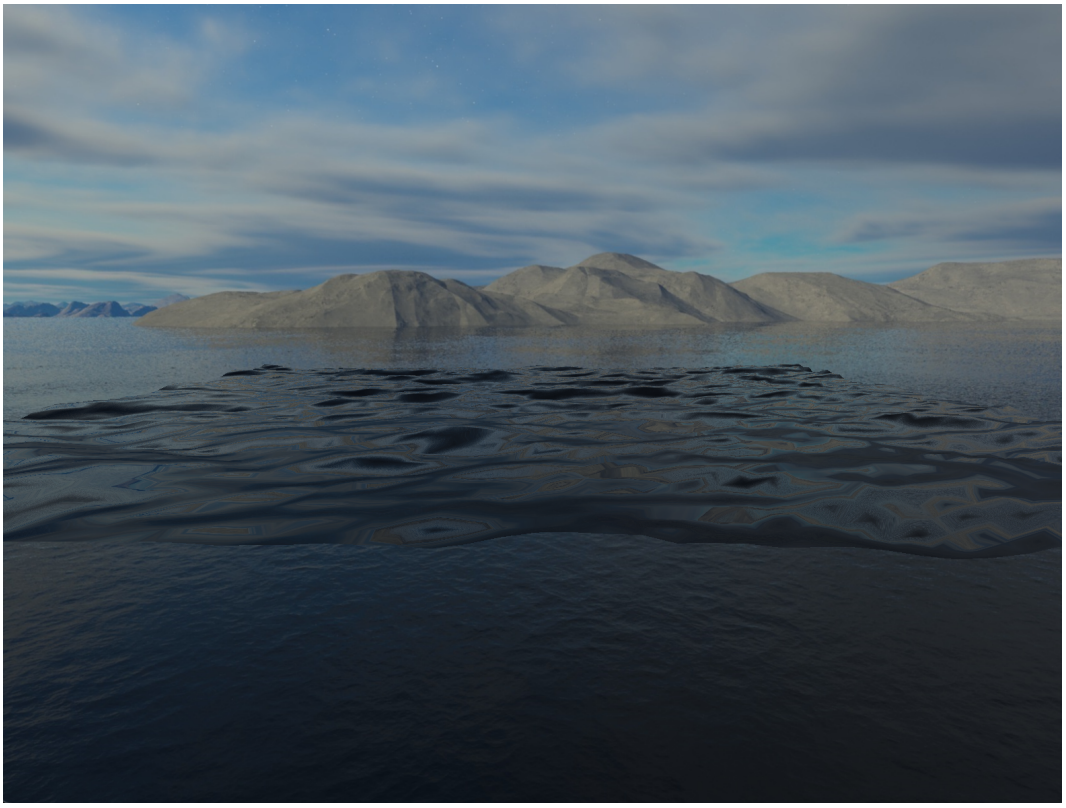

Figure 11: Shading of a wave surface using caustic shading and the Fresnel equations.

1. Compute the height map and the corresponding slope maps of the water mesh.
2. Write the water mesh's height, slope in the x direction, and slope in the z direction to the red, green, and blue channels of an RGB texture stored as a frame buffer object (FBO).
3. In the vertex shader of the projected grid, sample the FBO texture using GL_REPEAT wrapping mode. Use the sampled data to determine the height of each vertex and its corresponding normal.
4. Pass the vertex and its normal to the fragment shader to perform shading based on the skybox using the techniques discussed in the previous section.
5. Render the vertices of the projected grid to the screen.

At first glance this approach produced a fairly convincing ocean scene. In Figure 12, I have shared two screenshots of my results, and an extended clip of this scene is presented in my video report. It is worth noting, however, that I believe there are some errors in the final geometry of the rendered projected grid. I address these concerns in greater detail in Section 4.

2.6 Interactivity

Finally, in order to create my demo video, I also added keyboard and mouse controls to move the camera around the scene. Translational motion is performed using the `W, A, S, D` keys, and rotational motion is performed using the mouse input. Implementing translation was fairly straightforward. Each frame I keep track of the camera's viewing direction $\boldsymbol{v}$ (i.e. the vector from the camera $\boldsymbol{e}$ to its target $\boldsymbol{t}$), and I also keep track of the vector pointing to the left of the camera's viewing direction $\boldsymbol{l}$. The leftwards vector is computed

Figure 12: In the left image, when zoomed in close to the surface, the water appears to stretch to the horizon. In the right image, when zoomed out, the edges of the projected grid can be seen, and the periodicity of the water is more noticeable.

by taking the cross product between the upwards direction $\boldsymbol{u}$ and camera's viewing direction $\boldsymbol{v}$.

$$\boldsymbol{v} = \text{normalize}(\boldsymbol{t} - \boldsymbol{e})$$
$$\boldsymbol{l} = \boldsymbol{u} \times \boldsymbol{v}$$

Once these two vectors are known, the position of the camera, and the position of the camera's target can be translated forwards, backwards, left, or right relative to its coordinate system.

I followed an online tutorial [4] to implement camera rotations. Based on this tutorial, to rotate the camera, I relate the x and y mouse position to the yaw ϕ and pitch θ angles of the camera respectively. Once these two angles are known, the camera's viewing direction can be updated using the following equation:

$$v_x = \cos(\phi)cos(\theta)$$
$$v_y = \sin(\theta)$$
$$v_z = \sin(\phi)cos(\theta)$$

These computations are performed on the CPU each frame.

3 Learning Outcomes

In the preceding section, I discussed my process towards achieving each of my project goals. Each task required learning new skills and overcoming a variety of technical hurdles. In this section, I summarize some of the most important skills that I learned from the project.

3.1 Fast Fourier Transform

Correctly implementing the Cooley-Tukey FFT algorithm was one of the most time consuming and challenging parts of my project. While implementing the algorithm, I learned that some FFT algorithms (including the Cooley-Tukey algorithm) assume that an input spectrum including negative frequencies is ordered such that negative frequencies are listed after the positive frequencies. For example if the spectrum's frequencies in increasing order are $-f_n, -f_{n-1}, \ldots, -f_1, 0, f_1, f_2, \ldots, f_{n-1}$, the input to the FFT algorithm should be formatted as $0, f_1, f_2, \ldots, f_{n-1}, -f_n, -f_{n-1}, \ldots, -f_1$. This was one of the most challenging bugs to detect in my program, and I only learned the cause after reading into the NumPy implementation of the FFT

[4] https://learnopengl.com/Getting-started/Camera

algorithm in Python [5]. It was also useful to learn that N-dimensional FFT algorithms can be computed efficiently by repeatedly calling the 1D FFT algorithm. This drastically expands the utility of the 1D FFT algorithm. In a future project, I would likely opt to use an open source implementation of the FFT algorithm to improve performance. However, I am glad I chose to implement it myself for this project as it is an algorithm that had always fascinated me in the past.

3.2 Snell's Law and the Fresnel Equations

In previous assignments, we never attempted to account for the refraction of light in a medium. However, in order to accurately shade a water surface, refraction cannot be ignored. In this project, I learned that the refraction of light can be modelled using Snell's law and the Fresnel Equations. I familiarized myself with each of these equations and learned that refraction can implemented in GLSL using the `refract` command. After learning about refraction, I am interested to revisit the ray tracer assignment and add functionality to account for the refraction of translucent objects.

3.3 Environment Mapping

This project also was my first chance to explore texture mapping. Specifically, I used textures to generate a skybox, and then I treated the skybox as an environment map when performing shading. Rendering the skybox required mapping the skybox texture to vertices in the scene, and sampling the skybox was performed on the GPU in the shader program. While the skybox was relatively simple to implement, it made the scene significantly more immersive. I look forward to using similar effects in future graphics projects on my own time.

4 Future Work

4.1 Compute Shaders

Currently, the performance of my application is limited by the time required to compute the FFT each frame. In fact, if I use more than 64×64 vertices in the water mesh, my application begins to fall well below 30 frames per second. In the future, I hope to move the FFT to a compute shader that can be executed on the GPU. By making this change, many of the calls to the 1D FFT algorithm could be performed in parallel which would greatly improve performance. With the improved performance, I could also use higher fidelity meshes which would in turn improve the realism of the scene.

4.2 Tiling the Ocean

In my final demo scene, I show how my water mesh can be tiled to create a much larger body of water. However, I believe there are still some bugs in the shader programs that perform the tiling process. I have noticed that when I render the scene depicted in Figure 12 using wireframe mode, many vertices remain stationary throughout the animation, and other vertices propagate up and down wildly (see Figure 13). I believe the FBO water data texture is constructed correctly, so I assume the source of my error comes from either improperly sampling the texture, or improperly constructing the projected grid vertices. Unfortunately, despite several days of debugging, I have not been able to find or fix the source of my issue. I hope to keep working on this project in the coming weeks to find and fix this error.

4.3 Water Perturbations

There are a variety of extensions to Tessendorf's work that interest me, but I am most curious about a method proposed by Kryachko [3] that allows for local perturbations in the wave height map. Using Kryachko's method, it is possible to simulate the perturbations created by bouyant objects in the water, such as the wake formed by a moving ship or the ripples caused by a surfacing whale or dolphin. After

[5] https://numpy.org/devdocs/reference/routines.fft.html

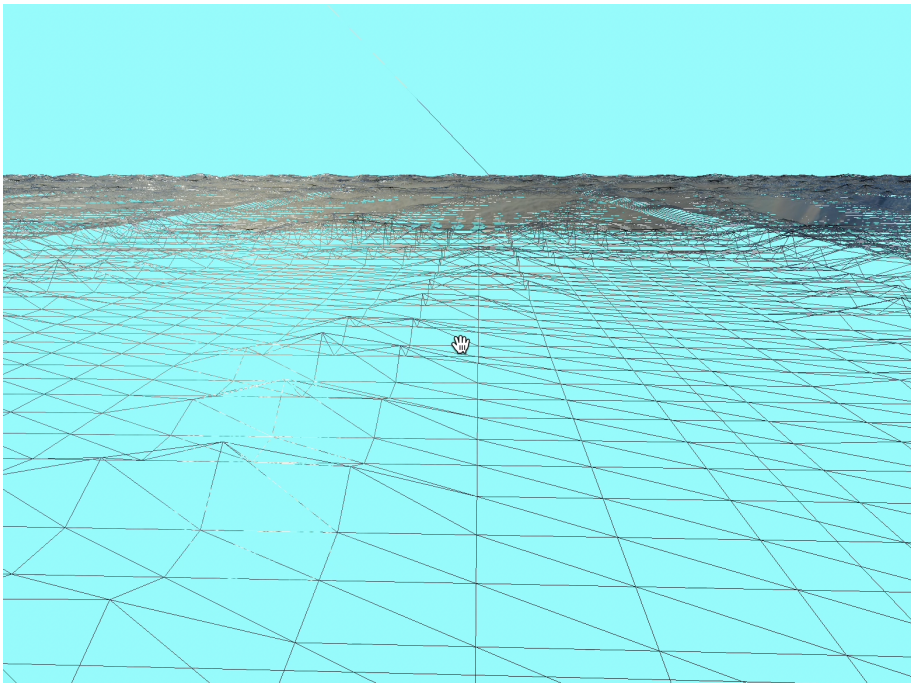

Figure 13: When the scene depicted in Figure 12 is rendered in wireframe mode, it is clear the vertex geometry is not behaving like a true wave.

improving my applications performance and fixing the ocean tiling process, this is the most interesting extension of the project to me.

5 Notes

In my project folder I have included my video report and my source code. Instructions for compiling and running the demos shown in my video report are given in the README file in my source code folder. I also included a make file that will automatically compile each demo. The main demo for my project is the `tiling.cpp` program, and this is the only demo that supports the keyboard and mouse controls discussed in Section 2.6.

References

[1] J. Tessendorf *et al.*, "Simulating ocean water," *Simulating nature: realistic and interactive techniques. SIGGRAPH*, vol. 1, no. 2, p. 5, 2001.

[2] J. Guardado and D. Sánchez-Crespo, "Rendering water caustics," *GPU Gems*, pp. 31–44, 2004.

[3] Y. Kryachko, "Using vertex texture displacement for realistic water rendering," *GPU gems*, vol. 2, pp. 283–294, 2005.

[4] C. Johanson and C. Lejdfors, "Real-time water rendering," *Lund University*, 2004.

[5] Y.-F. Chiu and C.-F. Chang, "Gpu-based ocean rendering," in *2006 IEEE international conference on multimedia and expo*, pp. 2125–2128, IEEE, 2006.