Rohith Vishwajith
Dr. Abraham
Computer Graphics
1 May 2025

Final Project: FFT Ocean Simulations in Unity

1. PROJECT OVERVIEW

1.1. Ocean Simulation using Compute Shaders

The goal of this project is to create a real-time ocean wave simulation fast enough to run on any modern computer with support for standard graphics features such as compute shaders (which process data using textures on the GPU in parallel as opposed to treating pixels as colors, like normal shaders). The wave simulation should have some form of basic environment and shading, such as a skybox, colors for the water, and potentially a curtain below the water to display the waterline. Of these features, I was able to successfully add in all of the water simulation steps and shading using an arbitrary skybox (this one is re-used from a past project, and was composed together from a series of open-source stylized images).

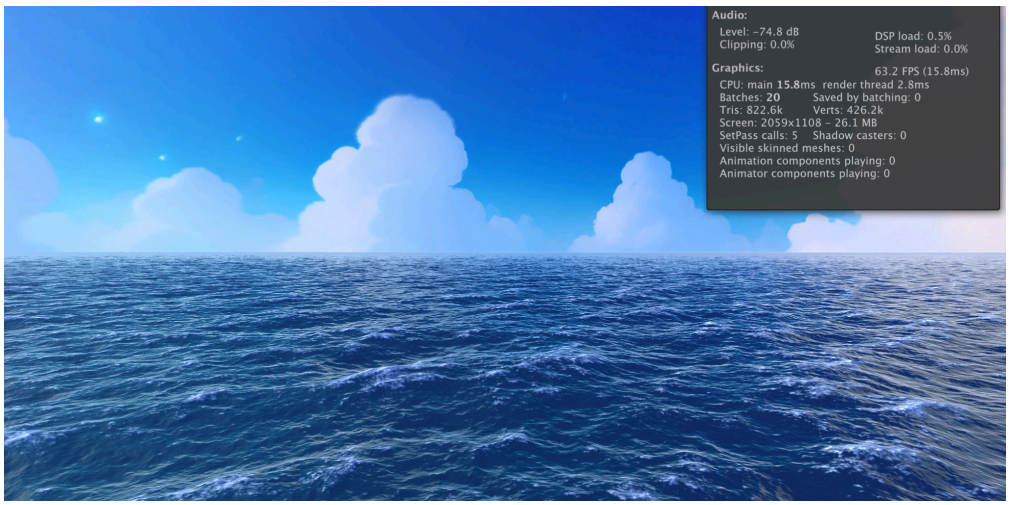

Figure 2.1.1 (Above). Screenshots of the real-time ocean simulation from this project, running in the Unity editor on an M1 Pro MacBook. Runs at 60 frames-per-second using a 512x512 grid (first image) and a 256x256 (second image) grid with an approximate rendering resolution of 2Kx1K. In the first image, the simulation runs at about 60 FPS, while the simulation averages about 240 FPS in the second image.

The industry standard approach to simulating a high-fidelity oceanscape, both in complex scenes for movies and more simple game environments, is using the FFT wave-spectrum technique outlined by Jerry Tessendorf in *Simulating Ocean Water* (2004). In the past, this technique has been used in large-scale simulations for movies such as *Titanic*, is prevalent in 3-D modelling and graphics software including Blender and Houdini, and is available in the two largest game engines, Unreal Engine and Unity. The medium for implementing this project will be in Unity, however this solution is built from scratch besides the use of Unity's shading language and compute shader APIs for implementing lighting and interfacing with the GPU.

1.2. Performance Optimization using Mesh LOD

In order to optimize the performance of the compute shader, as it will likely be necessary even with parallelization on the GPU, some form of mesh level-of-detail selections (LODs) or tessellation is needed for large scale simulations. Mesh tessellation is a form of optimization where the amount of detail, represented in a 3-dimensional mesh as vertices, is reduced based on the distance from the camera.

The image below showcases a use-case in the project of mesh level-of-details. A high-fidelity plane representing an ocean surface from close-up (relative to the active camera and its field-of-view) requires a large number of polygons. In a realistic use-case for real-time simulation, a grid of 1,024 x 1,024 vertices forms a 20x20 meter^2 plane with 2,000,000 triangles. Processing this amount of data for multiple meshes each frame is impossible on the CPU, and very difficult for a GPU since the data needs to be read back to the CPU in the following frame. The level-of-detail reduces the amount of data to process and apply to the mesh and scaling by camera distance minimizes any loss in quality and allows the oceanscape to scale until the horizon.

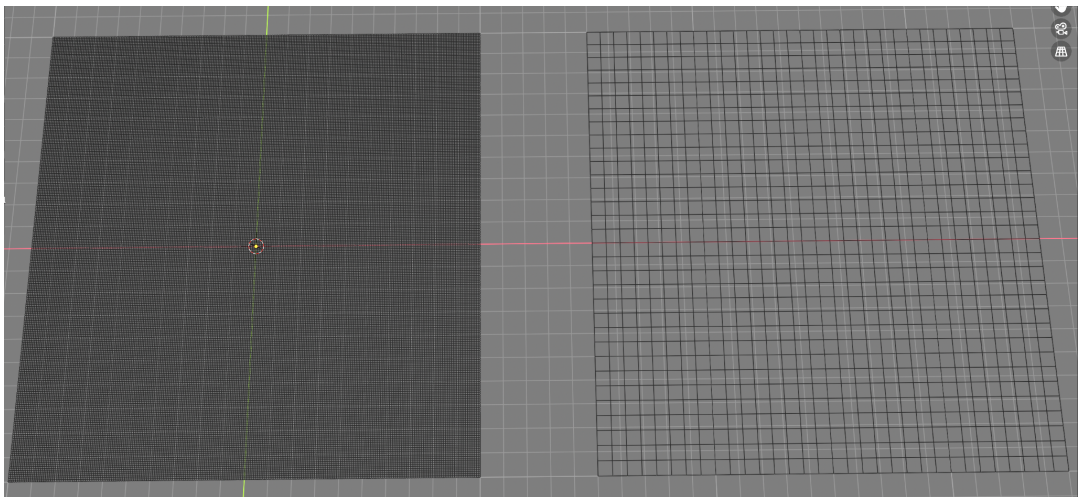

Figure 2.2.1 (Above). High-resolution meshes (left) are scaled down relative to camera distance to optimize lower quality meshes after a certain distance.

Some helper methods have been added to create skirt meshes which consist of sets of planes forming a border with a hollow square in the center.

1.3. Initial Planned Features

I had initially planned to add additional features to turn this project into a more lively oceanscape using various algorithms I've implemented in the past, such as kelp simulations and boids. However, due to the limited time frame of this project, I elected not to.

2. WAVE SURFACE IMPLEMENTATION - INITIAL SPECTRUM

2.1. Overview

Tessendorf's style of simulation is meant to replicate a large fluid simulation without following all the steps of smaller scale fluid simulations with N-bodies. In order to do this accurately, using real oceanography data is required. First, a spectrum of initial wave data needs to be generated.

In order to evolve (i.e. animate) this spectrum over time to represent a real moving ocean, theoretical data such as wind speed and direction, water depth (for gravitational influences and turbulence factor), and choppy wave enhancement are used. All of these inputs are not meant to realistically simulate real-world physics data in the engine (as one could do on a small scale using a Navier-Stokes fluid simulation), as this is impossible without infinite compute power, but rather approximate it.

Before generating a wave spectrum, it's important to understand that despite real-world oceans being impossible to simulate physically, they can ultimately be represented as a complex set of accumulated sinusoids, and the oscillating motion of the waves can be recreated using a strategy based on these sinusoids.

3.2. Domain and Initial Wave Spectrum Generation - Wave Cascades

The first step in simulating the ocean is creating a starting simulation grid. Note that all simulation grids are represented as a 2D texture for usage in the compute shader, which, in application, is adjacent to a 2D array. In the 2D texture, data is stored as high-precision floats and passed to the GPU in the form of an uncompressed image.

In order to generate a proper photorealistic result, we need to generate spectrums for multiple wave cascades, which means a compute shader needs to be dispatched and the first step of the simulation needs to run on a seperate texture for each cascade. A wave cascade generates a spectrum and evolves it using the standard FFT compute shader based on a scale, in meters. Larger cascades help add a primary level of detail, with large, slow moving waves, and they can be compounded with smaller, higher resolution simulations to approximate a more complex surface with a linear performance increase. As the level-of-detail for each mesh decreases based on the camera distance, high-resolution wave cascades can be neglected for better performance without losing much visual detail.

One downside of reducing the number of cascade simulations as the camera distances itself is that from a high point-of-view, like an aerial shot, the wave simulation can look choppier and potentially have seams without an algorithm to blend the textures seamlessly.

The WavesCascade class contains all of the texture data for each compute shader, as well as a "size" attribute to control the scale of the simulation. The "lambda" attribute in the class influences the choppiness of this cascade's simulation. Note that the size value cannot be changed in real-time, as the compute shader is created at a set resolution and changing it while

running would severely affect performance. However, the “lambda” value can be changed with no hit to performance for a more or less intense wave cascade. The following code segment outlines all of the textures and other variables that go into generating and updating the wave cascade.

WavesCascade.cs

```
public class WavesCascade
{
    // Displacement, Derivatics, and Turbulence render textures.
    public RenderTexture Displacement => displacement;
    public RenderTexture Derivatives => derivatives;
    public RenderTexture Turbulence => turbulence;

    public Texture2D GaussianNoise => gaussianNoise;

    // Precomputed JONSWAP spectrum.
    public RenderTexture PrecomputedData => precomputedData;
    public RenderTexture InitialSpectrum => initialSpectrum;

    // Scale. Higher scale = larger, lwoer-resolution simulation.
    readonly int size;
    // Choppiness.
    float lambda;

    // Initial spectrum data.
    readonly ComputeShader initialSpectrumShader;
    readonly ComputeShader timeDependentSpectrumShader;

    readonly ComputeShader texturesMergerShader;
    readonly FastFourierTransform fft;
    readonly Texture2D gaussianNoise;
    readonly ComputeBuffer paramsBuffer;
    readonly RenderTexture initialSpectrum;
    readonly RenderTexture precomputedData;
```

```
    readonly RenderTexture buffer;
    readonly RenderTexture DxDz;
    readonly RenderTexture DyDxz;
    readonly RenderTexture DyxDyz;
    readonly RenderTexture DxxDzz;

    readonly RenderTexture displacement;
    readonly RenderTexture derivatives;
    readonly RenderTexture turbulence;
```

Figure 3.2.1 (Above). All fields of the WaveCascade class. The "RenderTexture" object is a built-in Unity type that refers to a generic 2D texture that the compute shader can read and write from quickly with high-precision floats.

2.3. Domain and Initial Wave Spectrum Generation - JONSWAP Spectrum Generation

Once the wave cascades are initialized, the simulation needs an initial wave spectrum to describe the shape and energy of the ocean surface at time zero. This spectrum acts as the foundation from which the ocean waves are evolved forward in time using Fourier analysis. The initial wave spectrum is calculated inside the *InitialSpectrum.compute* compute shader, particularly the *CalculateInitialSpectrum* kernel. Each texel in the WavesData texture represents the properties of a wave vector at a specific frequency. The parameters influencing the calculation are:

1. Wind Speed and Direction
2. Gravity
3. Water Depth
4. Turbulence / Choppiness (the "lambda" field from WavesCascade)

The core formula used for the wave energy spectrum is the JONSWAP spectrum (Joint North Sea Wave Project), a more realistic and physically based spectrum compared to simpler models like the Phillips spectrum, which Tessendorf's paper uses.

2.3.1. Why JONSWAP Spectrum?

The JONSWAP spectrum better models real-world ocean waves because it enhances the peak frequency energy (higher, sharper peaks), models swell and fetch conditions more accurately. It also provides more tunable parameters than the Phillips spectrum like peak enhancement (gamma) and swell(s), and provides visually richer, non-uniform wave

structures as well. Since the release of Tessendorf's paper in 2004, JONSWAP has become the standard spectrum, used in Houdini, Blender, and ocean-based games such as *Sea of Thieves*.

2.4. JONSWAP Spectrum Generation in Code

The wave energy per frequency bin is calculated with this (simplified) formula inside *InitialSpectrum.compute*:

```
float spectrum = JONSWAP(omega, GravityAcceleration, Depth, Spectrums[0])
                 * DirectionSpectrum(kAngle, omega, Spectrums[0])
                 * ShortWavesFade(kLength, Spectrums[0]);
```

This does the following:

- Applies the base JONSWAP energy at frequency omega. In my case, this energy is low, to represent a calm ocean. In order to change the energy, I can modify it via the "lambda" variable in each wave cascade.
- The angular spreading (spread of wave directions).
- Suppression of very short waves (short waves fade out).

Then, random Gaussian noise (I just use a pre-generated texture, but a parametrically generated one for large tiles may work better) is multiplied by a value proportional to the spectrum energy (which can be scaled from 0 - 1), leading to an initial displacement texture.

3. WAVE SURFACE IMPLEMENTATION - UPDATING THE SPECTRUM

3.1. Frame-by-frame Update Loop

Each frame, the ocean simulation undergoes several sequential coordinated stages for each Wave Cascade, and the results of the simulation are merged after this step. The core simulation loop looks like this inside WavesGenerator.cs:

WavesGenerator.cs

```
private void Update()
{
    if (alwaysRecalculateInitials)
        InitialiseCascades();

    // Update each cascade.
    cascade0.CalculateWavesAtTime(Time.time);
    cascade1.CalculateWavesAtTime(Time.time);
    cascade2.CalculateWavesAtTime(Time.time);

    RequestReadbacks();
}
```

Figure 4.1.1 (Above): The update loop for the cascade. While the paper references updating the spectrum "per frame", in application, this is untrue. This is because readback from the GPU is asynchronous and may take more than one frame. Asynchronous GPU readback is an optimization that prevents bottlenecks for the rest of the game in case the hardware struggles to run this simulation in particular, due to a device having, for example, an integrated GPU or old dedicated GPU.

3.2. Step 1: Recalculate the Initial Spectrum after Spectrum Evolution

If the simulation settings (e.g., wind speed, direction, or gravity) have to be changed dynamically, we can optionally rebuild the initial wave spectrum textures based on the settings. This will negatively impact performance, however, because 3 additional compute shaders now have to be run.

3.3. Step 2: Update Time-dependent Fourier Amplitudes

Advance the simulation forward in time by evolving the complex amplitudes of each Fourier mode based on elapsed time t. The 2 code segments below show how the compute shader is updated before dispatching, and the compute shader handles the calculations after the textures are dispatched.

WavesCascade.cs

```
cascade0.CalculateWavesAtTime(Time.time);
cascade1.CalculateWavesAtTime(Time.time);
cascade2.CalculateWavesAtTime(Time.time);
```

WavesCascade.cs

```
timeDependentSpectrumShader.SetTexture(KERNEL_TIME_DEPENDENT_SPECTRUMS,
Dx_Dz_PROP, DxDz);
timeDependentSpectrumShader.SetTexture(KERNEL_TIME_DEPENDENT_SPECTRUMS,
Dy_Dxz_PROP, DyDxz);
timeDependentSpectrumShader.SetTexture(KERNEL_TIME_DEPENDENT_SPECTRUMS,
Dyx_Dyz_PROP, DyxDyz);
timeDependentSpectrumShader.SetTexture(KERNEL_TIME_DEPENDENT_SPECTRUMS,
Dxx_Dzz_PROP, DxxDzz);
timeDependentSpectrumShader.SetTexture(KERNEL_TIME_DEPENDENT_SPECTRUMS,
H0_PROP, initialSpectrum);
timeDependentSpectrumShader.SetTexture(KERNEL_TIME_DEPENDENT_SPECTRUMS,
PRECOMPUTED_DATA_PROP, precomputedData);
timeDependentSpectrumShader.SetFloat(TIME_PROP, time);
timeDependentSpectrumShader.Dispatch(KERNEL_TIME_DEPENDENT_SPECTRUMS, size /
8, size / 8, 1);
```

TimeDependentSpectrum.compute

```
float2 h = ComplexMult(H0[id.xy].xy, e^(i * omega * time))
         + ComplexMult(H0[id.xy].zw, e^(-i * omega * time));
```

Figure 4.3.1 (Above): An overview of the compute shader dispatch and update process from WavesCascade.

This results in new time-dependent complex wave amplitudes at (x, y), and displacements and curvature derivatives, which are stored in the compute shader as Dx_Dz (horizontal displacement), Dy_Dxz (vertical displacement), Dyz_Dyz (slope displacement), and Dxx_Dxz (the curvature derivative).

3.4. Step 3: Run an Inverse Fast-Fourier Transform in 2-D

Now, we need to take the real-space displacement and derivative maps, and combine them into usable render textures (height, normals, and turbulence) for shading the water surface. The texture merger compute shader taxes the resulting textures from regenerating (optionally) and evolving the spectrum:

```
texturesMergerShader.SetFloat("DeltaTime", Time.deltaTime);
texturesMergerShader.SetTexture(KERNEL_RESULT_TEXTURES, Dx_Dz_PROP, DxDz);
texturesMergerShader.SetTexture(KERNEL_RESULT_TEXTURES, Dy_Dxz_PROP, DyDxz);
texturesMergerShader.SetTexture(KERNEL_RESULT_TEXTURES, Dyx_Dyz_PROP,
DyxDyz);
texturesMergerShader.SetTexture(KERNEL_RESULT_TEXTURES, Dxx_Dzz_PROP,
DxxDzz);
texturesMergerShader.SetTexture(KERNEL_RESULT_TEXTURES, DISPLACEMENT_PROP,
displacement);
texturesMergerShader.SetTexture(KERNEL_RESULT_TEXTURES, DERIVATIVES_PROP,
derivatives);
texturesMergerShader.SetTexture(KERNEL_RESULT_TEXTURES, TURBULENCE_PROP,
turbulence);
texturesMergerShader.SetFloat(LAMBDA_PROP, lambda);
texturesMergerShader.Dispatch(KERNEL_RESULT_TEXTURES, size / 8, size / 8,
1);
```

Figure 4.3.1 (Above): Dispatching derivative/curvature data to texture merger shader. Time.deltaTime needs to be passed manually due to asynchronous readback.

Inside the texture merger compute shader, an inverse FFT is run on the 2D grid which writes to a final texture containing displacement, derivatives, and turbulence data.

```
TextureMergerShader.compute
```

```
Displacement[id.xy] = float3(Lambda * DxDz.x, DyDxz.x, Lambda * DxDz.y);
Derivatives[id.xy] = float4(DyxDyz, DxxDzz * Lambda);
float jacobian = (1 + Lambda * DxxDzz.x) * (1 + Lambda * DxxDzz.y) - Lambda
* Lambda * DyDxz.y * DyDxz.y;
Turbulence[id.xy] = Turbulence[id.xy].r + DeltaTime * 0.5 / max(jacobian,
0.5);
```

Figure 4.4.1 (Above): Writing displacement, derivative, and turbulence data to individual textures. Some of these, such as turbulence, can potentially be combined for improved performance in the future.

This is the final step for the parallel parts of the simulation. Once the textures are updated, the built-in Unity materials read from them using the Ocean shader and normals data to handle displacement, lighting, and foam (based on the turbulence texture) data.

4. CODE STRUCTURE

4.1. Class Overview

The following C# classes are used to handle the wave simulation, as well as update the mesh that represents the ocean (OceanGeometryController.cs). All scripts can be found in the “Scripts” folder in the Unity project.

4.1.1. WavesGenerator.cs

The main controller, which is attached to the GameObject. Initializes the FFT system, Gaussian noise textures, and three separate WavesCascade instances for multiple wave detail levels. It also ipdates all cascades every frame based on the elapsed simulation time and interfaces with GPU readback for height sampling. Note that the GPU readback is asynchronous, so it doesn;t necessarily have to occur within 1 frame. This helps avoid limitations on the entire scene in a game as a result of the simulation needing to complete each frame.

4.1.2. WavesCascade.cs

Manages the lifecycle of one wave cascade and creates and stores simulation textures (displacement, derivatives, turbulence). As mentioned in the prior spectrum, each wave cascade handles initial spectrum creation (static) and time evolution of waves (dynamic) for a given “lambda” (choppiness factor) and SIZExSIZE (scale of the simulation) grid. During each update, each WavesCascade object dispatches compute shaders for both spectrum evolution and IFFT.

4.1.3. FastFourierTransform.cs

Implements a GPU-based 2D Fast Fourier Transform (FFT) and Inverse FFT (IFFT) using compute shaders and optimizes memory access using a ping-pong buffer and precomputed twiddle factors (the twiddling is standard). There are also provides utility methods for scaling and permuting FFT results, but these should be made into static standalone methods in the future.

4.1.4. OceanGeometryController.cs

Handles the ocean surface mesh rendering and dynamically instantiates different LOD meshes based on the camera distance. Also assigns displacement and normal maps to the ocean material for proper shading.

4.1.5. WavesSettings.cs

Stores tunable physical parameters (gravity, water depth, wind speed, wave height scaling, etc.), and provides methods to populate compute shader buffers with properly converted spectrum settings.

4.2. Compute Shaders

All compute shaders can be found in the “ComputeShaders” folder in the Unity project.

4.2.1 FastFourierTransform.compute

Precomputes the FFT twiddle factors (standard) and Applies horizontal and vertical FFT passes to textures. Performs optional scaling and data permutation after IFFT.

4.2.2. InitialSpectrum.compute

Computes the initial wave energy spectrum based on the JONSWAP model, and stores the base wave vectors and initial amplitudes for each simulation grid point.

4.2.3. TimeDependentSpectrum.compute

Evolves the wave spectrum over time by applying a time-based phase shift to each complex amplitude, and outputs displacement and derivative quantities in Fourier space.

4.2.4. WavesTextureMerger.compute

Combines the displacement and derivative maps into final height and normal maps, and computes turbulence using a Jacobian determinant for foam rendering.

5. IMPROVEMENTS AND OPTIMIZATIONS

5.1. Future Optimizations

Several performance optimizations can be made, which are outlined below.

5.1.1. JOBS System for Multi-threaded GPU Dispatch

Using Unity's built-in JOBS system, which can help dispatch graphics compute jobs from multiple threads, could greatly improve performance by getting each cascade to dispatch on a separate processes CPU-side. This will only improve performance, however, if the GPU has the capability to handle multiple large-resolution high-precision textures at once. However, most modern discrete GPUs and even integrated ARM SoC's are more than capable of this, including the device all testing was done on (M1 Pro Mabook Pro, 32 GB RAM)..

5.1.2. FFT Butterfly Structure

Using an FFT butterfly structure could potentially allow running the entire FFT as a single dispatch rather than multiple horizontal/vertical passes, reducing memory barriers. This has not been explored thoroughly, as this approach seems exceedingly complex and may ultimately not improve performance on a capable GPU.

5.2. Additional Features

5.2.1. Buoyancy Physics Simulations

One of the upsides of using FFT over simpler simulations is re-using the displacement data to make buoyancy calculations for floating objects such as boats and buoys is much easier, simply requiring reading the computed texture asynchronously.

5.2.2. Aforementioned Planned Improvements

Adding a lively environment using verlet-rope kelp, procedurally animated creatures, and schools of fish would be useful if this project was used in an actual game. If this was the case, an underwater side of the surface shader would be necessary as well, with transparency and Snell's window.

5.2.3. Moving to the Universal Render Pipeline

The Universal Render Pipeline offers better general performance and more advanced lighting features. However, migrating proved to be difficult due to limitations on custom written shaders in URP, and for a full conversion re-implementation in Shader Graph would be required.

6. WORKS CITED

Epic Dragonfly. *How Sea of Thieves Water Is Made*. YouTube, premiered 4 June 2022, www.youtube.com/watch?v=FCDpIMPvx-k. Accessed 23 Apr. 2025.

Flügge, F.-J. *Realtime GPGPU FFT Ocean Water Simulation*. Institute of Embedded Systems, 2017, tore.tuhh.de/bitstream/11420/1439/1/GPGPU_FFT_Ocean_Simulation.pdf. Accessed 25 Apr. 2025.

Gamper, Tobias. *Ocean Surface Generation and Rendering*. Vienna University of Technology, Faculty of Informatics, 2018, www.cg.tuwien.ac.at/research/publications/2018/GAMPER-2018-OSG/GAMPER-2018-OSG-thesis.pdf. Accessed 21 Apr. 2025.

Horvath, Christopher J. "Empirical Directional Wave Spectra for Computer Graphics." *Proceedings of the 2015 Symposium on Digital Production (DigiPro '15)*, Association for Computing Machinery, 2015, pp. 29–39. https://doi.org/10.1145/2791261.2791267. Accessed 22 Apr. 2025.

Jourverse. *How Ocean Waves Work in Unreal Engine: FFT & Wave Simulation*. YouTube, premiered 19 Dec. 2023, www.youtube.com/watch?v=OWiyIc2bVwM. Accessed 24 Apr. 2025.

Matusiak, R. *Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS 320 DSP Platform*. Texas Instruments, 2002, www.ti.com/lit/an/spra291/spra291.pdf. Accessed 22 Apr. 2025.

Tcheblokov, T. *Ocean Simulation and Rendering in War Thunder*. NVIDIA, CGDC 2015, developer.download.nvidia.com/assets/gameworks/downloads/regular/events/cgdc15/CGDC2015_ocean_simulation_en.pdf. Accessed 25 Apr. 2025.

Tessendorf, Jerry. *Simulating Ocean Water*. SIGGRAPH 1999 Course Notes, 2001, www.researchgate.net/publication/264839743_Simulating_Ocean_Water. Accessed 23 Apr. 2025.