VR Ocean

Real-Time Ocean Surface Synthesis via Inverse FFT with Vectorized Job Parallelism and Adaptive Quadtree Level-of-Detail for Virtual Reality

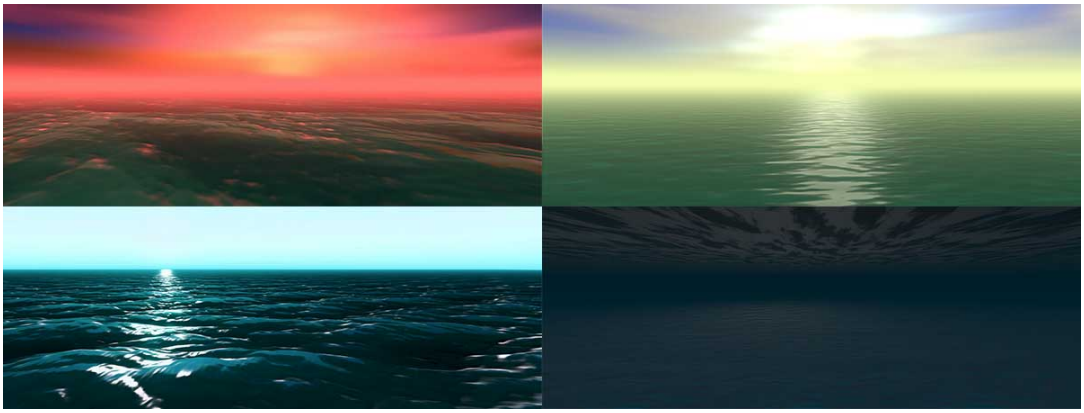
Roman Aebi

September 5, 2026

Contents

Abstract

This work presents a real-time ocean simulation system optimized for virtual reality applications. The implementation combines Fast Fourier Transform (FFT) based wave synthesis using the Phillips spectrum with an adaptive quadtree level-of-detail rendering system. Through extensive use of Unity's Job System and Burst compilation, the simulation achieves performance suitable for VR's demanding frame rate requirements. The system features a six-stage parallel simulation pipeline replicated across up to three independent wave cascades, each covering a disjoint band of the wave spectrum at its own patch size. The pipeline combines a shared precomputed twiddle table, in-place stack-allocated FFT buffers, Hermitian conjugate packing for SIMD-optimized temporal evolution, and an adaptive quadtree with 16 pre-computed edge-stitching configurations. This document focuses exclusively on the core mathematical foundations, algorithmic details, and parallel architecture that constitute the simulation engine.



Link: VR Ocean Tool Documentation Link: Unity Asset Store

1 Introduction

Real-time ocean simulation remains a challenging problem in computer graphics, particularly for virtual reality where maintaining high frame rates with stereoscopic rendering is crucial for user comfort. This work presents a solution that balances visual quality with computational efficiency through parallel processing and adaptive rendering techniques. The simulation builds upon Tessendorf's seminal work on ocean surface modeling (Tessendorf, 1999), implementing the Phillips spectrum for initial wave amplitude distribution and utilizing inverse Fast Fourier Transform (iFFT) for efficient wave synthesis. The core contributions of this implementation include:

- A six-stage parallelized simulation pipeline using Unity's Job System with Burst compilation, featuring a shared precomputed twiddle table and in-place stack-allocated FFT buffers for optimal cache utilization
- A multi-cascade spectrum decomposition that layers up to three independent FFT simulations over disjoint wave number bands, removing the visible tiling of a single patch without raising the FFT resolution
- Hermitian conjugate packing with algebraically derived SIMD pre-transformations that reduce the temporal evolution to a single vectorized multiply-add per wave vector
- An adaptive quadtree-based level-of-detail system with iterative stack-based traversal, 16 precomputed edge-stitching index buffer configurations, and frustum-culled GPU instanced rendering

- A complete surface gradient pipeline computing normals via central differences, Jacobian-based foam detection, polynomial roughness-to-smoothness conversion, and RGBA8 channel packing with mipmap generation

2 Testing Environment and Hardware Requirements

2.1 Development Environment

The ocean simulation system was developed and optimized using Unity 6000.3.18f1, leveraging the latest performance improvements and rendering features of Unity 6. The implementation utilizes the Universal Render Pipeline (URP) 17.3.0 for optimal VR rendering performance, together with the XR Interaction Toolkit 3.3.2 and XR Hands 1.7.3 for interaction and hand tracking.

2.2 Target Hardware Platform

The primary target platform is the Meta Quest 3, offering a Qualcomm Snapdragon XR2 Gen 2 processor, 8GB RAM, native resolution of 2064×2208 per eye, and 90Hz/120Hz refresh rate capability. The Quest 3's enhanced processing power enables real-time FFT calculations while maintaining comfortable VR frame rates.

2.3 Minimum System Requirements

For development and PC VR deployment: 8-core CPU with AVX2 support for Burst compilation, RTX 3070 or equivalent GPU, Unity 6000.3.18f1 or later, and OpenXR 1.16.1 with Meta Quest support. Memory pressure from the simulation itself is modest: at the maximum 512×512 resolution with three cascades, the persistent native arrays and GPU textures together occupy roughly 90 MB, so the practical memory requirement is set by the host project rather than by the ocean system.

3 Mathematical Foundation

3.1 Ocean Wave Theory

The ocean surface simulation is based on the linear superposition of sinusoidal waves with different frequencies and directions. The height field $h(\mathbf{x}, t)$ at position $\mathbf{x} = (x, z)$ and time t is computed as the sum of complex amplitudes:

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

where $\mathbf{k}$ represents the wave vector and $\tilde{h}(\mathbf{k}, t)$ is the time-dependent complex amplitude in frequency domain. The wave vectors are discretized on a regular grid of resolution $N \times N$:

$$\mathbf{k}_{m,n} = \frac{2\pi}{L_{\text{patch}}} \left(m - \frac{N}{2},\ n - \frac{N}{2} \right), \quad m, n \in \{0, \ldots, N-1\} \tag{2}$$

where L_{patch} is the physical extent of one simulated patch and $N \in \{16, 32, 64, 128, 256, 512\}$. When several cascades are active (Section 3.10), each runs this same discretization at its own L_{patch}, so L_{patch} should be read throughout this section as the extent of the cascade under discussion rather than of the ocean as a whole.

3.2 Phillips Spectrum

The initial wave spectrum follows the Phillips spectrum (Tessendorf, 1999), which describes the statistical distribution of ocean waves based on wind conditions.

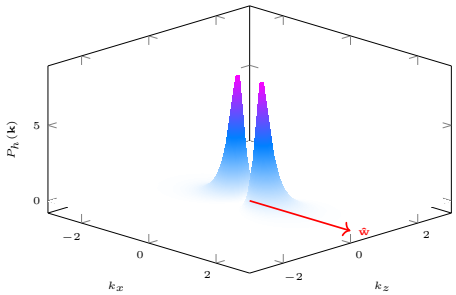


Figure 1: Phillips spectrum $P_h(\mathbf{k})$ of equation (3) with the directional factor of equation (7), evaluated for $L = 2$ and $s = 3$ with the wind along $+k_x$. The k^{-4} pole at the origin is clamped for display; in the implementation it is removed by the DC guard of Section 3.2.7. Energy concentrates at long wavelengths near the origin and falls off away from the wind axis.

The theoretical Phillips spectrum is given by:

$$P_h(\mathbf{k}) = A\frac{e^{-1/(kL)^2}}{k^4} \cdot D(\mathbf{k}, \hat{\mathbf{w}}) \tag{3}$$

where $k = |\mathbf{k}|$ is the wave number magnitude, $L = V^2/g$ is the largest wave scale (peak wavelength), V is the wind speed, g is gravitational acceleration, and $\hat{\mathbf{w}}$ is the normalized wind direction.

3.2.1 Amplitude Normalization

The practical implementation requires normalization factors to ensure energy conservation and FFT compatibility. The amplitude scale factor decomposes into two independent terms, an FFT resolution normalization and a spatial density normalization:

$$A = A_{\text{FFT}}^2 \cdot A_{\text{spatial}}^2 \tag{4}$$

where the individual factors are:

$$A_{\text{FFT}} = N^{-1/4}, \qquad A_{\text{spatial}} = \frac{e}{L_{\text{patch}}} \tag{5}$$

Both factors are empirical normalizers rather than closed-form results, chosen so that wave height stays visually stable when the grid resolution and the patch size are varied independently. The butterfly stages apply no $1/N$ scaling of their own (Section 3.6.3), so the entire transform gain is absorbed here. The factor $N^{-1/4}$ enters the power spectrum squared and therefore contributes $N^{-1/2}$ to the power and $N^{-1/4}$ to the amplitude, which was found to hold apparent wave height roughly constant across the supported resolutions. The spatial factor e/L_{patch} makes the amplitude inversely proportional to patch size, so that enlarging a patch does not simply scale the same surface up; the constant $e \approx 2.71828$ is a fixed dimensionless gain rather than a derived quantity, and any tuned constant of similar magnitude would serve.

3.2.2 Directional Spreading

The directional energy distribution uses a power-cosine spreading model. The spreading exponent is a quadratic function of the alignment parameter $\alpha \in [0, 1]$:

$$s(\alpha) = 1 + 5\alpha^2 \tag{6}$$

This maps $\alpha = 0$ (omnidirectional) to $s = 1$, and $\alpha = 1$ (fully directional) to $s = 6$. The directional factor is applied to the *power* spectrum at twice this exponent:

$$D(\mathbf{k}, \hat{\mathbf{w}}) = |\hat{\mathbf{k}} \cdot \hat{\mathbf{w}}|^{2s} \tag{7}$$

where $\hat{\mathbf{k}} = \mathbf{k}/k$ is the normalized wave direction. Because the amplitude is obtained as the square root of the power spectrum (Section 3.2.6), the exponent seen in the amplitude domain is s itself, ranging from 1 to 6. The doubling exists so that s can be reasoned about directly as an amplitude lobe exponent while the spectrum is still assembled in the power domain.

3.2.3 Counter-Wave Suppression

For waves traveling against the wind direction ($\hat{\mathbf{k}} \cdot \hat{\mathbf{w}} < 0$), an energy suppression factor models the physical dissipation of counter-propagating waves:

$$\sigma(\alpha) = 0.01 + 0.49(1 - \alpha)^2 \tag{8}$$

At full alignment ($\alpha = 1$), opposing waves retain only 1% of their energy; at no alignment ($\alpha = 0$), they retain 50%. The suppression enters the amplitude as a square root factor to operate in the amplitude domain rather than the power domain:

$$h_0^{\text{opposing}} = h_0 \cdot \sqrt{\sigma(\alpha)} \tag{9}$$

3.2.4 Small-Wave Suppression

To suppress small wavelengths that cause aliasing artifacts, an exponential Gaussian cutoff is applied in the frequency domain:

$$P_h'(\mathbf{k}) = P_h(\mathbf{k}) \cdot e^{-k^2 \, l_{\text{cutoff}}^2} \tag{10}$$

The cutoff wavelength l_{cutoff} is interpolated exponentially between bounds, providing logarithmically uniform control across the suppression range:

$$l_{\text{cutoff}} = l_{\text{min}} \cdot \left(\frac{L_{\text{patch}} \cdot 0.25}{l_{\text{min}}} \right)^{\beta} \tag{11}$$

with $l_{\text{min}} = 0.001$, the upper bound at 25% of patch size, and $\beta \in [0, 1]$ controlling suppression strength. This exponential interpolation ensures that the parameter β produces perceptually uniform transitions: $\beta = 0$ applies virtually no suppression (cutoff at l_{min}) while $\beta = 1$ aggressively filters all wavelengths below one quarter of the patch.

3.2.5 Stochastic Amplitude Generation

The initial complex amplitudes require Gaussian-distributed random variables. The Box-Muller transform converts uniform random samples $u_1, u_2 \sim \mathcal{U}(0, 1)$ into independent standard normal variates:

$$
\begin{aligned}
r &= \sqrt{-2 \ln u_1} \\
\xi_{\text{Re}} &= r \cdot \sin(2\pi u_2) \\
\xi_{\text{Im}} &= r \cdot \cos(2\pi u_2)
\end{aligned}
\tag{12}
$$

The implementation generates four Gaussian values from four uniform inputs simultaneously, producing the two complex Gaussian variables ξ_+ and ξ_- needed for both the positive and negative frequency components in a single pass. Per-index deterministic seeding ensures reproducible spectra across frames and platforms.

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static float4 GenerateGaussianPair(float4 uniform) {
    var radius = sqrt(BOX_MULLER_RADIUS_SCALE * log(uniform.xz));
    var angle = TWO_PI * uniform.yw;

    return float4(
        radius.x * sin(angle.x),
        radius.x * cos(angle.x),
        radius.y * sin(angle.y),
        radius.y * cos(angle.y));
}
```

Listing 1: Box-Muller transform producing four Gaussian variates from four uniform samples

3.2.6 Complete Amplitude Computation

The directional factor of equation (7) and the small-wave cutoff of equation (10) are both already contained in $P'_h(\mathbf{k})$, so the final amplitude is obtained by a single square root of the assembled power spectrum combined with the stochastic component:

$$
h_0(\mathbf{k}) = \frac{1}{\sqrt{2}} \xi \cdot \sqrt{P'_h(\mathbf{k})}
\tag{13}
$$

where the $1/\sqrt{2}$ factor accounts for the Hermitian conjugate pairing: the total energy is distributed equally between $\tilde{h}_0(\mathbf{k})$ and $\tilde{h}_0^*(-\mathbf{k})$. The counter-wave suppression of equation (9) is the only term applied after this square root, because it is defined directly in the amplitude domain.

3.2.7 DC Component Guard

The spectrum has a singularity at $\mathbf{k} = (0, 0)$ where the wave number $k = 0$ causes division by zero in the k^{-4} term. This is handled by explicitly zeroing both the amplitude and angular frequency at the DC component before any computation:

$$
h_0(\mathbf{0}) = 0, \qquad \omega(\mathbf{0}) = 0
\tag{14}
$$

The same early exit also handles the cascade band split: a wave vector whose magnitude reaches the upper bound owned by the current cascade (Section 3.10) has its amplitude and angular frequency zeroed, so that band is left entirely to the next, smaller cascade. With both guards in place, the implementation uses fast floating-point mode with reduced precision across all jobs for maximum SIMD utilization.

3.3 Dispersion Relation

Ocean waves follow the deep-water dispersion relation linking angular frequency to wave number:

$$\omega(k) = \sqrt{g\,k} \tag{15}$$

For seamless tiling of the ocean animation, the continuous frequency is quantized to integer multiples of a fundamental frequency ω_0:

$$\omega'(k) = \left\lfloor \frac{\omega(k)}{\omega_0} \right\rfloor \omega_0, \qquad \omega_0 = \frac{2\pi}{T} \tag{16}$$

where T is the repeat period (configurable, default 200 seconds). This quantization ensures that all frequency components return to their initial phase after exactly T seconds, producing a seamlessly looping animation. The floor operation snaps each component to the nearest lower harmonic, introducing negligible visual error while guaranteeing periodicity.
Because the spectrum is exactly periodic over T, the simulation clock is wrapped into that window rather than allowed to grow without bound:

$$t = (t_{\text{engine}} \cdot \kappa) \bmod T \tag{17}$$

where κ is a user-facing time scale (default 1.0, range $[0, 2]$) that speeds up or slows down the whole surface without altering the spectrum. Wrapping is not merely cosmetic: the phase argument $\omega' t$ is evaluated in single precision, so an unwrapped clock would steadily lose angular resolution over a long session. Since the surface is identical at t and $t+T$, the wrap is free of any visual discontinuity.

3.4 Time Evolution and Hermitian Conjugate Packing

3.4.1 Theoretical Time Evolution

The time-dependent complex amplitudes evolve according to the dispersion relation. The implementation adopts the sign convention

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k})\, e^{-i\omega' t} + \tilde{h}_0^*(-\mathbf{k})\, e^{i\omega' t} \tag{18}$$

which is the complex conjugate of the more commonly quoted form. The choice is deliberate and is dictated by the transform kernel: equation (1) synthesizes the surface with $e^{+i\mathbf{k}\cdot\mathbf{x}}$, whereas the butterfly stages are implemented with the forward DFT kernel $W_N = e^{-2\pi i/N}$ (Section 3.6.1). Running an inverse synthesis through a forward kernel flips the sign of the exponent, and the temporal convention above cancels that flip exactly. Adopting the opposite convention in either place alone would reverse the direction in which waves travel relative to the wind.
Either form guarantees a real-valued height field, as required physically. A naïve implementation would require accessing both index $\mathbf{k}$ and its conjugate $-\mathbf{k}$ for each evaluation, causing scattered memory access patterns that defeat cache prefetching.

3.4.2 Packed Representation

The implementation eliminates the conjugate index lookup by pre-computing and packing both components into a single four-component vector during spectrum generation. Define:

$$\mathbf{S}(\mathbf{k}) = \begin{pmatrix} \mathrm{Re}(\tilde{h}_0(\mathbf{k})) \\ \mathrm{Im}(\tilde{h}_0(\mathbf{k})) \\ \mathrm{Re}(\tilde{h}_0^*(-\mathbf{k})) \\ \mathrm{Im}(\tilde{h}_0^*(-\mathbf{k})) \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \tag{19}$$

This packing reduces memory bandwidth by 50% and enables purely sequential access patterns during time evolution.

3.4.3 SIMD Pre-Transformation Derivation

To reduce the time evolution to a single vectorized operation, the packed spectrum undergoes an algebraic pre-transformation during spectrum generation. Expanding the theoretical evolution:

$$\begin{aligned} \tilde{h}(\mathbf{k}, t) &= (a + bi)(\cos\phi + i\sin\phi) + (c + di)(\cos\phi - i\sin\phi) \\ &= \big[(a + c)\cos\phi + (b - d)\sin\phi\big] \\ &\quad + i\big[(b + d)\cos\phi + (c - a)\sin\phi\big] \\ &\text{where } \phi = -\omega' t \end{aligned} \tag{20}$$

Here ϕ absorbs the negative exponent of equation (18), so that $\cos\phi = \cos(\omega' t)$ and $\sin\phi = -\sin(\omega' t)$. Working in ϕ keeps every coefficient below free of sign corrections.
Defining the rotation vector $\mathbf{r} = (\cos\phi, \sin\phi)$, the goal is to express this as $\mathbf{S}'_{xy} \cdot r_x + \mathbf{S}'_{zw} \cdot r_y$ for some pre-transformed vector $\mathbf{S}'$. Collecting coefficients of $\cos\phi$ and $\sin\phi$:

$$\tilde{h} = \underbrace{\begin{pmatrix} a + c \\ b + d \end{pmatrix}}_{\mathbf{S}'_{xy}} \cos\phi + \underbrace{\begin{pmatrix} b - d \\ c - a \end{pmatrix}}_{\mathbf{S}'_{zw}} \sin\phi \tag{21}$$

The pre-transformation that produces these components from the original packing $\mathbf{S} = (a, b, c, d)$ is performed in two steps. First, construct the intermediate vector:

$$\mathbf{S}^{(1)} = \underbrace{(a, b, a, b)}_{\mathbf{S}_{xyxy}} + \underbrace{(c, d, -c, -d)}_{(\mathbf{S}_{zw}, -\mathbf{S}_{zw})} = (a{+}c,\ b{+}d,\ a{-}c,\ b{-}d) \tag{22}$$

Then swap and negate the zw components:

$$\mathbf{S}' = \big(S_x^{(1)},\ S_y^{(1)},\ S_w^{(1)},\ -S_z^{(1)}\big) = (a{+}c,\ b{+}d,\ b{-}d,\ c{-}a) \tag{23}$$

Verification. Substituting into $\mathbf{S}'_{xy} \cdot r_x + \mathbf{S}'_{zw} \cdot r_y$:

$$\begin{aligned} \mathrm{Re}(\tilde{h}) &= (a{+}c)\cos\phi + (b{-}d)\sin\phi \\ \mathrm{Im}(\tilde{h}) &= (b{+}d)\cos\phi + (c{-}a)\sin\phi \end{aligned} \tag{24}$$

These are exactly the $\cos\phi$ and $\sin\phi$ coefficients of equation (20), so the pre-transformation is verified with no residual sign correction. The single negation introduced by the swap in equation (23) is precisely what carries the $-i$ of the adopted convention into the packed layout. The temporal evolution thus reduces to:

$$\boxed{\tilde{h}(\mathbf{k},t) = \mathbf{S}'_{xy}\cos\phi + \mathbf{S}'_{zw}\sin\phi, \qquad \phi = -\omega' t} \tag{25}$$

In the implementation the negation of ϕ is folded into the packed spectrum rather than evaluated at runtime, so the job computes sincos($\omega' t$) directly. Both trigonometric values therefore come from a single intrinsic call, and the entire operation maps to one fused multiply-add on SIMD hardware.

```
public void Execute(int index) {
    var omega = _angularFrequencyTable[index];
    var phase = omega * _time;

    float2 rotation;
    sincos(phase, out rotation.y, out rotation.x);

    var h0 = _initialAmplitudes[index];
    var heightSample = h0.xy * rotation.x + h0.zw * rotation.y;
    _heightSpectrum[index] = heightSample;

    var gridX = index & _indexMask;
    var gridY = index >> _logResolution;
    var waveVector = float2(gridX, gridY) - _halfResolution;

    var lengthSquared = dot(waveVector, waveVector);
    waveVector *= rsqrt(max(1.0f, lengthSquared));

    var heightConjugate = Conjugate(float2(heightSample.y, heightSample.x));
    var displacementX = heightConjugate * waveVector.x;
    var displacementZ = heightConjugate * waveVector.y;

    _displacementSpectrum[index] = float4(displacementX, displacementZ);
}
```

Listing 2: Temporal evolution: packed SIMD height evaluation and displacement spectrum in one pass

3.5 Displacement Calculation

3.5.1 Displacement Theory

The horizontal displacement for choppy waves is computed from the height spectrum using the normalized wave vector as a directional weight:

$$\mathbf{D}(\mathbf{x},t) = -\lambda \sum_{\mathbf{k}} i\frac{\mathbf{k}}{k}\tilde{h}(\mathbf{k},t)\, e^{i\mathbf{k}\cdot\mathbf{x}} \tag{26}$$

where $\lambda \in [0,1]$ is the choppiness parameter controlling wave sharpness. The factor $i \cdot \mathbf{k}/k$ applies a 90° phase rotation scaled by the wave direction, concentrating surface area at wave crests and expanding it at troughs.

3.5.2 Bitwise Wave Vector Extraction

For a flattened linear index into the $N \times N$ grid, the two-dimensional grid coordinates are recovered without division or modulo using bitwise operations that exploit the power-of-two grid size:

$$g_x = \texttt{index} \,\&\, (N-1)$$
$$g_y = \texttt{index} \gg \log_2 N \tag{27}$$

where & denotes bitwise AND and $\gg$ denotes right shift. The mask $(N-1)$ isolates the lower $\log_2 N$ bits (equivalent to $\bmod N$), while the shift extracts the upper bits (equivalent to $\lfloor \text{index}/N \rfloor$). The centered wave vector is then:

$$\mathbf{k}_{\text{grid}} = (g_x, g_y) - N/2 \tag{28}$$

3.5.3 Wave Vector Normalization

The wave direction $\hat{\mathbf{k}} = \mathbf{k}/|\mathbf{k}|$ uses reciprocal square root with a DC-safe guard:

$$\hat{\mathbf{k}} = \mathbf{k}_{\text{grid}} \cdot \text{rsqrt}\big(\max(1,\ |\mathbf{k}_{\text{grid}}|^2)\big) \tag{29}$$

The $\max(1, \cdot)$ clamp prevents division by zero at the origin while leaving all non-DC components unaffected (since $|\mathbf{k}|^2 \geq 1$ for all non-zero integer grid coordinates).

3.5.4 Vectorized Complex Displacement

The multiplication by the imaginary unit i performs a 90° phase rotation, which algebraically amounts to swapping and negating components of the complex number:

$$i \cdot (h_x + i h_y) = -h_y + i h_x \tag{30}$$

In the implementation, this is expressed as a conjugation with swapped input ordering: $\text{conj}(h_y, h_x) = (h_y, -h_x)$. The complete displacement computation for both horizontal axes is then vectorized into a single four-component operation:

$$\mathbf{D}_{\text{spec}} = \text{conj}(h_y, h_x) \otimes (\hat{k}_x, \hat{k}_x, \hat{k}_y, \hat{k}_y) \tag{31}$$

where $\otimes$ denotes element-wise multiplication and the subscript patterns indicate component replication. This produces both the x- and z-displacement spectra in a single SIMD instruction.

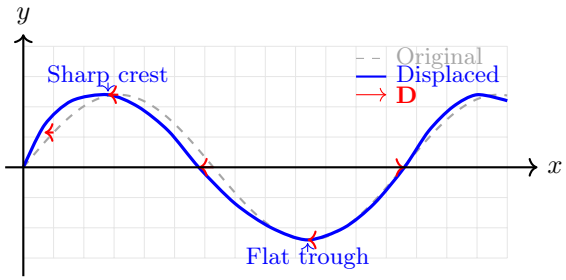


Figure 2: Horizontal displacement creating characteristic ocean wave profile with sharp crests and flat troughs

3.6 FFT Implementation

3.6.1 Discrete Fourier Transform

The summations in equation (1) are efficiently computed using a 2D FFT. The one-dimensional DFT of N elements is:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot W_N^{kn}, \qquad W_N = e^{-2\pi i/N} \tag{32}$$

where W_N is the primitive N-th root of unity. The 2D transform is computed as separable row-wise and column-wise 1D transforms.

Note that the implementation uses the *forward* kernel $e^{-2\pi i/N}$ throughout, even though equation (1) describes an inverse synthesis with $e^{+i\mathbf{k}\cdot\mathbf{x}}$. Two consequences follow. First, the sign of the exponent is compensated in the temporal evolution rather than in the butterfly, as described in Section 3.4. Second, no $1/N$ or $1/N^2$ scaling is applied at any stage; the full transform gain is folded into the amplitude normalization of Section 3.2.6 instead, which saves a pass over the data.

3.6.2 Cooley-Tukey Butterfly Structure

The radix-2 decimation-in-time Cooley-Tukey algorithm (Cooley and Tukey, 1965) decomposes the DFT into $\log_2 N$ stages of $N/2$ butterfly operations each. At stage $p \in \{0, \ldots, \log_2 N - 1\}$, each butterfly operates on a pair of elements separated by stride 2^p:

$$\begin{aligned} X_{\text{top}} &= x_a + W_N^{k_t} \cdot x_b \\ X_{\text{bot}} &= x_a - W_N^{k_t} \cdot x_b \end{aligned} \tag{33}$$

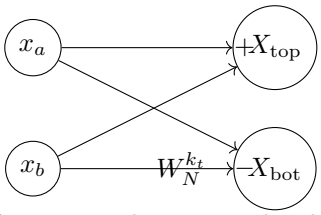


Figure 3: FFT butterfly operation combining two complex values with twiddle factor

3.6.3 Precomputed Twiddle Table

The twiddle factors are evaluated once at allocation time into a table of $N/2$ complex entries, using Euler's formula:

$$W_N^{k_t} = e^{-2\pi i\, k_t/N} = \cos\left(\frac{-2\pi k_t}{N}\right) + i \sin\left(\frac{-2\pi k_t}{N}\right) \tag{34}$$

The twiddle exponent k_t for butterfly index j at stage p is derived from the butterfly's position within its group:

$$k_t = \underbrace{\big(j \;\&\; (2^p - 1)\big)}_{\text{position in group}} \cdot \underbrace{\frac{N}{2^{p+1}}}_{\text{table stride}} \tag{35}$$

The element indices for the butterfly pair are computed as:

$$\begin{aligned} g &= j \gg p && \text{(group index)} \\ \text{idx}_a &= (g \ll (p+1)) + (j \;\&\; (2^p - 1)) && \text{(top element)} \\ \text{idx}_b &= \text{idx}_a + 2^p && \text{(bottom element)} \end{aligned} \tag{36}$$

Entry k of the table stores W_N^k, and equation (35) is used directly as the table index. Because the exponent is always a multiple of $N/2^{p+1}$ and never reaches $N/2$, a table of $N/2$ entries covers

every stage of every pass with no bounds arithmetic.

The table is built once per resolution change and is shared by every FFT work item and, when several cascades are active, by every cascade as well. Its footprint is $4N$ bytes, 2 KB at the maximum supported resolution, so it stays resident alongside the stack buffers of Section 3.6.5 for the duration of a pass. Compared with evaluating the trigonometric pair inline, this removes $\frac{N}{2}\log_2 N$ transcendental calls per row or column while adding only a read from a small, sequentially reused table. Both the group split and the position within the group are derived with a shift and a mask rather than a division and a modulo, exploiting the same power-of-two structure used in Section 3.5.2.

```
var group = butterfly >> pass;
var posInGroup = butterfly & strideMask;

var idxA = (group << (pass + 1)) + posInGroup;
var idxB = idxA + stride;

var twiddle = twiddleTable[posInGroup * twiddleStep];

// Both operands are read before either is written, so the
// butterfly is safe in place and needs no scratch buffers.
var hA = height[idxA];
var hB = height[idxB];
var hBTwiddle = Multiply(hB, twiddle);
height[idxA] = hA + hBTwiddle;
height[idxB] = hA - hBTwiddle;
```

Listing 3: One butterfly: shift and mask index derivation, table lookup, and in-place update

3.6.4 Bit-Reversal Permutation

The decimation-in-time Cooley-Tukey algorithm requires input data in bit-reversed order. For an index i with $b = \log_2 N$ significant bits, the bit-reversed index is:

$$\text{BitReverse}(i, b) = \text{reversebits}(i) \gg (32 - b) \tag{37}$$

where reversebits is a hardware intrinsic that reverses all 32 bits of an unsigned integer, and the right shift discards the $(32 - b)$ least significant bits of the reversed result. For example, with $N = 8$ ($b = 3$):

$$\begin{aligned} \text{BitReverse}(1, 3): &\quad 001_2 \xrightarrow{\text{rev32}} \ldots 100_2 \xrightarrow{\gg 29} 100_2 = 4 \\ \text{BitReverse}(3, 3): &\quad 011_2 \xrightarrow{\text{rev32}} \ldots 110_2 \xrightarrow{\gg 29} 110_2 = 6 \end{aligned} \tag{38}$$

This produces the standard input ordering: $\{x_0, x_4, x_2, x_6, x_1, x_5, x_3, x_7\}$ for $N = 8$.

3.6.5 Stack-Allocated Buffer Strategy

The FFT implementation avoids persistent heap-allocated ping-pong buffers by using stack-allocated memory for each row or column. Each parallel work item allocates exactly two local buffers:

$$\text{Buffers} = \left\{\texttt{local}_h[N],\ \texttt{local}_d[N]\right\} \tag{39}$$

where subscripts h and d denote height (2-component) and displacement (4-component) data respectively. Both spectra are processed in the same job to maximize data reuse. No scratch or swap buffers are required: within a butterfly, both operands are read before either is written, so each stage transforms the line in place. The data flow for a single FFT work item proceeds in three phases:

1. **Load**: Input data is gathered from global arrays with bit-reversed indexing into the local buffers
2. **Compute**: All $\log_2 N$ butterfly stages are processed entirely in local memory, each stage overwriting the line in place with no copy and no buffer swap
3. **Store**: The final result is written back to global output arrays in a single sequential pass

This strategy provides three advantages. First, stack memory resides in L1 cache throughout the computation, eliminating cache misses during the iterative butterfly stages. Second, the in-place formulation halves the stack footprint that a ping-pong scheme would require, to $24N$ bytes per work item, being one N-element buffer of 8-byte height samples and one of 16-byte displacement samples. Third, no persistent buffer management is needed between frames, since the allocations are reclaimed automatically when the work item completes.

3.6.6 Dual-Spectrum Processing

The horizontal and vertical FFT passes simultaneously transform both the height spectrum (stored as 2-component complex values) and the displacement spectrum (stored as 4-component values encoding both x- and z-displacement as complex pairs). Each butterfly operation is applied to both spectra using the same twiddle factor:

$$\begin{aligned} h_{\text{top}} &= h_a + \text{Mul}(h_b, W), \qquad & d_{\text{top}} &= d_a + \big(\text{Mul}(d_b^{xy}, W),\ \text{Mul}(d_b^{zw}, W)\big) \\ h_{\text{bot}} &= h_a - \text{Mul}(h_b, W), \qquad & d_{\text{bot}} &= d_a - \big(\text{Mul}(d_b^{xy}, W),\ \text{Mul}(d_b^{zw}, W)\big) \end{aligned} \tag{40}$$

where $\text{Mul}(a, b) = (a_x b_x - a_y b_y,\ a_x b_y + a_y b_x)$ is the complex multiplication. This co-processing eliminates redundant twiddle factor computation and exploits instruction-level parallelism.

3.6.7 Vertical Pass Memory Access Pattern

The horizontal pass operates on contiguous rows, achieving optimal sequential access. The vertical pass must gather column data from strided memory (stride = N elements). The implementation loads strided input into contiguous stack-allocated buffers before processing:

$$\texttt{local}[r] = \texttt{source}[\text{col} + \text{BitReverse}(r, b) \cdot N], \quad r \in \{0, \ldots, N{-}1\} \tag{41}$$

After the FFT, results are scattered back with the transpose:

$$\texttt{result}[\text{col} + r \cdot N] = \texttt{local}[r] \tag{42}$$

This gather-process-scatter pattern converts the unfavorable column access into sequential local processing, allowing the L1 cache to serve the butterfly stages without cache line thrashing. The strided access is thereby confined to two passes over the data, one at load and one at store, instead of recurring in each of the $\log_2 N$ butterfly stages.

3.7 Displacement Finalization

After the 2D FFT, the raw frequency-domain output requires two corrections to produce the final spatial-domain displacement field.

3.7.1 Sign Correction

The standard DFT convention places the DC component at index $(0, 0)$, but the centered spectrum representation (with $\mathbf{k}$ offset by $N/2$) introduces an alternating sign pattern that must be corrected:

$$s(x, y) = (-1)^{x+y} \tag{43}$$

This is computed branch-free using a bitwise test:

$$s = \begin{cases} +1 & \text{if } (x + y) \,\&\, 1 = 0 \\ -1 & \text{otherwise} \end{cases} \tag{44}$$

3.7.2 Final Displacement Assembly

The sign correction and choppiness scaling are combined in a single pass:

$$\mathbf{D}_{\text{final}}(x, y) = \big(-D_x \cdot \lambda \cdot s, \;\; h \cdot s, \;\; -D_z \cdot \lambda \cdot s\big) \tag{45}$$

where h is the height from the FFT output, D_x, D_z are the horizontal displacement components, λ is the choppiness parameter, and s is the sign correction. The output is dual-written to both a three-component floating-point array (for subsequent gradient computation) and a four-component half-precision texture array (for GPU rendering), avoiding a separate copy pass.

3.8 Surface Gradient Computation

3.8.1 Normal Calculation via Central Differences

Surface normals are computed from the displacement field using central finite differences with toroidal (wrap-around) boundary conditions:

$$\left.\frac{\partial \mathbf{D}}{\partial x}\right|_{c,r} \approx \frac{\mathbf{D}(c{+}1, r) - \mathbf{D}(c{-}1, r)}{2} \cdot \frac{N}{L_{\text{patch}}} \tag{46}$$

The factor N/L_{patch} converts from texel-space differences to world-space derivatives. Indices wrap toroidally:

$$\text{wrap}(i) = (i + N) \bmod N \tag{47}$$

The surface normal is constructed from the height derivatives:

$$\mathbf{n} = \text{normalize}\left(-\frac{\partial h}{\partial x}, \; 1, \; -\frac{\partial h}{\partial z}\right) \tag{48}$$

3.8.2 Jacobian Determinant for Foam Detection

Wave breaking is detected through the Jacobian determinant of the displaced surface mapping. When the horizontal displacement causes the surface to fold (multiple surface points mapping to the same horizontal position), the Jacobian becomes negative:

$$J = \underbrace{\left(1 + \frac{\partial D_x}{\partial x}\right)}_{J_{xx}} \underbrace{\left(1 + \frac{\partial D_z}{\partial z}\right)}_{J_{zz}} - \underbrace{\frac{\partial D_x}{\partial z}}_{J_{xz}} \underbrace{\frac{\partial D_z}{\partial x}}_{J_{zx}} \tag{49}$$

When $J < 0$, the surface has folded and foam should appear. When $J > 0$, the surface is smooth. The threshold $J = 0$ marks the exact breaking point. This determinant is stored in the blue channel of the output texture for use by the rendering shader.

3.8.3 Roughness-to-Normal-Length Polynomial

To encode surface roughness information into the normal map for physically correct mipmap filtering, the implementation maps a roughness value $r \in [0, 1]$ to an expected normal vector length using a polynomial approximation of the microfacet distribution integral:

$$|\mathbf{n}|_{\text{scaled}} = 1 - r\big(0.306 + r(0.113 + r \cdot 0.081)\big) \tag{50}$$

This Horner-form polynomial requires exactly 3 fused multiply-add (FMA) operations with no branches or transcendental functions. The approximation maps $r = 0$ (perfectly smooth) to $|\mathbf{n}| = 1$ and $r = 1$ (maximum roughness) to $|\mathbf{n}| = 0.5$. The coefficients were fitted to replace the expensive atanh-based conversion from Beckmann microfacet theory.
The argument passed to the polynomial is not the perceptual roughness directly but its square, following the standard convention that maps artist-facing roughness to the microfacet distribution parameter:

$$r = (1 - \text{smoothness})^2 \tag{51}$$

where the smoothness is read from the surface material. Two remarks on range are in order. The polynomial spans $[0.5, 1]$, whereas the inverse mapping used during mipmap filtering (Section 3.9.2) is defined on $[2/3, 1]$ and saturates below it. The lower portion of the polynomial's range is therefore not recoverable through the round trip, which is harmless in practice because the ocean material is configured at high smoothness and equation (51) keeps r small, but the two mappings are not exact inverses of one another and should not be treated as such.

3.8.4 RGBA8 Channel Packing

The surface gradient data is packed into a 32-bit integer for direct texture upload. The normal vector components are mapped from the signed range $[-1, 1]$ to unsigned $[0, 1]$, then quantized to 8-bit integers:

$$\begin{aligned} R &= \lfloor (n_x \cdot 0.5 + 0.5) \cdot 255 + 0.5 \rfloor \\ G &= \lfloor (n_z \cdot 0.5 + 0.5) \cdot 255 + 0.5 \rfloor \\ B &= \lfloor \text{saturate}(J \cdot 0.5 + 0.5) \cdot 255 + 0.5 \rfloor \\ A &= \lfloor \text{smoothness} \cdot 255 + 0.5 \rfloor \end{aligned} \tag{52}$$

The four bytes are combined into a single 32-bit integer using bit shifts:

$$\text{packed} = R \mid (G \ll 8) \mid (B \ll 16) \mid (A \ll 24) \tag{53}$$

where $\ll$ denotes left shift and $\mid$ denotes bitwise OR. This packing is performed identically in both the base-level gradient job and the mipmap filter jobs, ensuring consistent data layout across all

mip levels.

3.9 Mipmap Generation

3.9.1 Box Filter Averaging

A chain of mipmap filter jobs generates the mipmap pyramid from the base-level normal and foam data. Each texel of the working chain is a four-component value carrying the roughness-scaled normal in xyz and the Jacobian determinant in w, and the box filter is applied to all four components at once. For each mip level m from 1 to $\log_2 N$, the four source texels are averaged:

$$\bar{\mathbf{n}}_m(x,y) = \frac{1}{4}\sum_{i=0}^{1}\sum_{j=0}^{1}\mathbf{n}_{m-1}(2x+i, 2y+j) \tag{54}$$

Critically, the averaged result preserves the *unnormalized* normal vectors. The length of the averaged normal encodes the variance of the source normals: when all four source normals point in the same direction, $|\bar{\mathbf{n}}| \approx 1$; when they diverge significantly, $|\bar{\mathbf{n}}| \ll 1$. The averaged Jacobian in the fourth component is written to the blue channel of the corresponding mip level, so foam coverage is box-filtered down the chain rather than recomputed per level.

3.9.2 Normal-Length to Smoothness Conversion

The averaged normal length is converted to an equivalent smoothness value suitable for the rendering shader's roughness model. The conversion maps the domain $[2/3, 1]$ to $[0, 1]$ using a two-stage polynomial:

$$\begin{aligned} t &= \text{saturate}\left(\frac{|\bar{\mathbf{n}}| - 2/3}{1/3}\right) \\ r &= 1 - t(0.85 + 0.15t) \\ \text{smoothness} &= 1 - \sqrt{r} \end{aligned} \tag{55}$$

The first line normalizes the input to $[0, 1]$. The second line converts to roughness using a quadratic that transitions from $r = 1$ (maximum roughness) at $t = 0$ to $r = 0$ (perfectly smooth) at $t = 1$, with the coefficients 0.85 and 0.15 providing a slightly concave profile that matches perceptual roughness better than a linear ramp. The final square root converts from roughness to smoothness in the standard PBR convention where smoothness $= 1 - \sqrt{\text{roughness}}$.

After conversion, the averaged normal is re-normalized and packed into RGBA8 format using the same bit-packing scheme described in Section 3.8.4, with the smoothness value replacing the base-level smoothness in the alpha channel.

3.10 Multi-Cascade Spectrum Decomposition

A single FFT patch has one fundamental wavelength, equal to its extent L_{patch}, and one shortest resolvable wavelength, set by the grid resolution. Widening that range by enlarging the patch coarsens the smallest waves; narrowing it to keep fine detail makes the tiling of the patch obvious at distance. The implementation resolves this by running several independent simulations, called cascades, each covering a disjoint band of the wave spectrum at its own patch size, and summing their contributions.

3.10.1 Patch Size Ladder

Up to $C_{\max} = 3$ cascades are supported, ordered from the largest patch to the smallest. The smallest cascade is pinned to the user-facing detail size, and each larger cascade is derived from its successor by a fixed ratio ρ:

$$L_{C-1} = L_{\text{detail}}, \qquad L_i = \frac{L_{i+1}}{\rho}, \quad i = C-2, \ldots, 0 \tag{56}$$

where $\rho \in [0.05, 0.5]$ after clamping, with a default of 0.25. Anchoring the ladder at the smallest cascade rather than the largest means that adding a cascade preserves the fine detail already on screen and extends the simulation upward into longer waves, instead of resampling the existing surface. With the default settings of $L_{\text{detail}} = 64\,\text{m}$, $\rho = 0.25$ and $C = 2$, the patch sizes are 256 m and 64 m; a third cascade adds a 1024 m patch.

3.10.2 Band Assignment

Each cascade is restricted to the wave numbers it alone represents. Cascade i discards every wave vector at or above the fundamental of the next smaller cascade:

$$k_{\max}^{(i)} = \begin{cases} \dfrac{2\pi}{L_{i+1}} & i < C-1 \\ \infty & i = C-1 \end{cases} \tag{57}$$

and any wave vector failing this test has both its amplitude and its angular frequency zeroed by the same early exit that handles the DC component (Section 3.2.7). The construction is exactly seamless: the upper bound of cascade i equals the lower bound of cascade $i+1$, because a patch of extent L_{i+1} has fundamental wave number $2\pi/L_{i+1}$. Every wave number in the combined range is therefore simulated by exactly one cascade, with neither a gap nor a double count of energy.

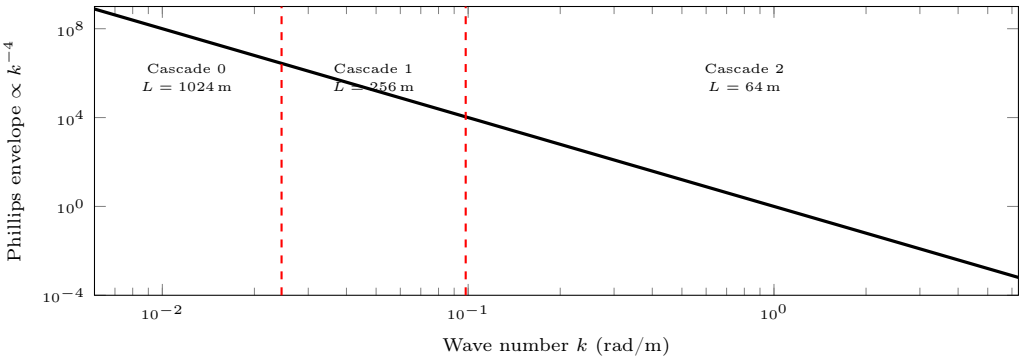


Figure 4: Band assignment for three cascades at $\rho = 0.25$ and $L_{\text{detail}} = 64\,\text{m}$. Dashed lines mark the cutoffs $2\pi/L_{i+1}$; each cutoff coincides with the fundamental of the next cascade, so the bands tile the spectrum without gaps or overlap.

3.10.3 Phase Decorrelation

Every cascade seeds its Box-Muller sampler (Section 3.2.5) from the texel index, which would give all cascades identical random phases and cause their crests to align into a visible grid. A per-cascade offset is therefore added to the seed:

$$\text{seed}_i = \text{index} + i \cdot \Phi_{32}, \qquad \Phi_{32} = \texttt{0x9E3779B9} \tag{58}$$

where Φ_{32} is the 32-bit golden ratio constant, chosen because successive multiples of it are maximally spread over the integer range. The offset is zero for cascade 0, so a single-cascade configu-

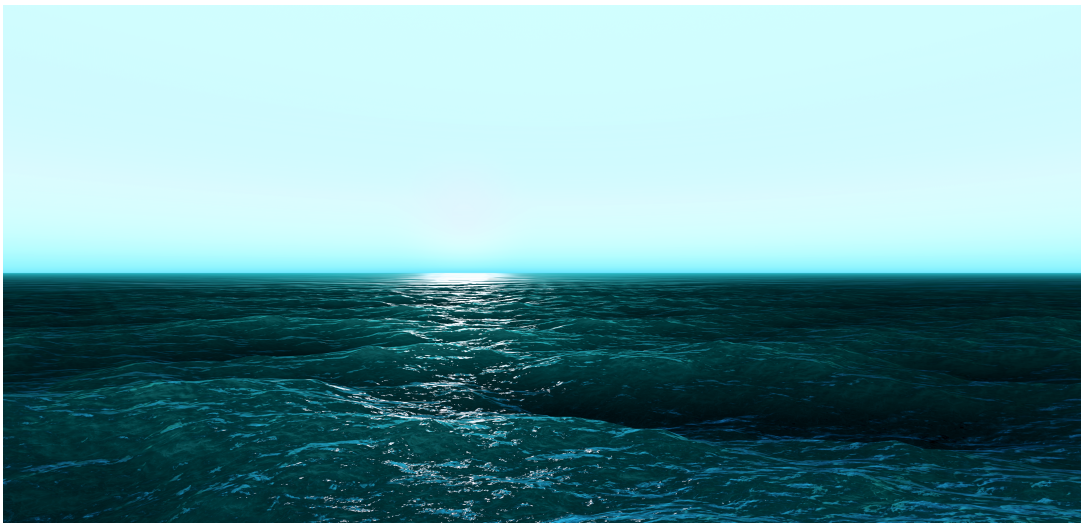
ration produces bit-identical output to one generated without the cascade mechanism at all.

3.10.4 Surface Reconstruction

Because the bands are disjoint, the cascades combine by simple summation. The displacement at a world position $\mathbf{x}$ is

$$\mathbf{D}_{\text{total}}(\mathbf{x}) = \sum_{i=0}^{C-1} \mathbf{D}_i\left(\frac{\mathbf{x}}{L_i}\right) \tag{59}$$

with each cascade sampled at its own reciprocal patch scale. The same sum is evaluated in two places: in the surface shader, which samples C displacement textures in the vertex stage and accumulates C normal and folding contributions in the fragment stage; and on the CPU, where height queries for buoyancy sum the same cascades from the displacement snapshots. Both paths are driven by a single cascade count uniform, so a configuration change needs no shader variant.



3.10.5 Cost

Each cascade owns a complete and private set of simulation buffers and runs the full six-stage chain of Section 4.2 over them, so the CPU cost of the simulation is linear in C. The only shared resource is the twiddle table of Section 3.6.3, which depends on resolution alone. On the GPU the cost is C additional texture samples in the vertex stage and C in the fragment stage. The default of two cascades was selected as the point at which the visible repeat of a single patch disappears at typical viewing distances, with the third reserved for scenes where long swell is prominent.

4 Parallel Implementation

4.1 Job System Architecture

The ocean simulation leverages Unity's Job System with Burst compilation to achieve parallelization. The per-frame computation consists of a six-stage chain of specialized jobs, each compiled with reduced floating-point precision and fast math mode for maximum SIMD throughput, and each with container safety checks disabled to eliminate bounds checking overhead. A seventh job, spectrum generation, runs outside the per-frame chain and is scheduled only on demand. The

quadtree traversal job of Section 5.2 is the one exception to the safety-check policy: it retains its checks, because it writes to a list whose length is data dependent rather than to fixed-size arrays. When several cascades are active (Section 3.10), the entire six-stage chain is instantiated once per cascade over that cascade's private buffers. The chains share no data and are therefore scheduled as independent dependency chains, combined into a single handle so that the caller waits on all of them at once.

4.2 Pipeline Stages

4.2.1 Stage 0: Spectrum Generation (On Demand)

Computes the Phillips spectrum with directional spreading for all N^2 wave vectors as an embarrassingly parallel operation (one work item per wave vector, batch size 64). This stage simultaneously generates both the positive and negative frequency amplitudes, applies the SIMD pre-transformation derived in Section 3.4.3, and stores the result in the packed four-component representation (equation 19). This job runs **on demand**, only when wind parameters or profile change, not every frame.

4.2.2 Stage 1: Temporal Evolution

Advances all wave amplitudes forward in time using the packed SIMD formulation (equation 25). Each work item computes the trigonometric pair via a single intrinsic, performs the vectorized height evaluation, extracts the wave vector via bitwise operations (Section 3.5.2), and produces the displacement spectrum (equation 31). The operation is embarrassingly parallel with zero data dependencies between elements.

4.2.3 Stage 2–3: Separable 2D FFT

The horizontal pass (Stage 2) processes each of the N rows independently, and the vertical pass (Stage 3) processes each of the N columns independently. Both passes use the stack-allocated buffer strategy (Section 3.6.5) with bit-reversed input loading, $\log_2 N$ butterfly stages driven by the shared twiddle table (Section 3.6.3), and dual-spectrum processing (equation 40). The pipeline uses a double-buffering scheme: Stage 2 reads from buffer A and writes to buffer B; Stage 3 reads from buffer B and writes back to buffer A.

4.2.4 Stage 4: Displacement Finalization

Applies sign correction and choppiness scaling (equation 45), dual-writing the result to both the CPU-side displacement array and the GPU-side half-precision texture data.

4.2.5 Stage 5–6: Surface Gradients and Mipmaps

Computes normals via central differences, the Jacobian determinant, roughness-scaled normal length, and RGBA8 packing (Section 3.8.3 through Section 3.8.4). Subsequently, a chain of $\log_2 N$ mipmap filter jobs (Stage 6) generates the complete mip pyramid with per-level smoothness encoding. These jobs are serialized against one another, since each level consumes the level above it.

4.3 Pipeline Scheduling and Double Buffering

All stages are scheduled as a dependency chain within a single frame:

$$\text{Handle}_n = \text{Job}_n.\text{Schedule}(\ldots,\ \text{Handle}_{n-1}) \tag{60}$$

The chain is scheduled at the end of a frame, after rendering, and completed at the start of the following frame, so the jobs have a full frame of worker-thread time and the main thread normally waits for nothing. Texture data is applied to GPU textures at that completion point.
Completion also publishes a stable snapshot of the displacement data. Physics and gameplay code may query wave height at any point in the frame, including while the next step is already in flight, so those queries read the snapshot rather than the live simulation buffer.

Within one cascade, the buffer flow through the FFT stages follows an A→B→A pattern:

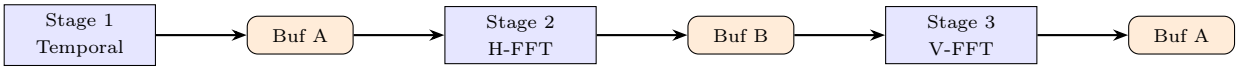


Figure 5: Double-buffering flow through the FFT stages: temporal evolution writes to A, the horizontal pass reads A and writes B, and the vertical pass reads B and writes back to A. Each cascade owns a private pair of buffers and repeats this flow; only the twiddle table is shared.

Stage	Job	Parallel Strategy	Batch
0	WaveSpectrumInitializerJob	Per wave vector (on demand)	64
1	TemporalEvolutionJob	Per wave vector	64
2	HorizontalSpectralPassJob	Per row	1
3	VerticalSpectralPassJob	Per column	1
4	DisplacementFinalizationJob	Per texel	64
5	SurfaceGradientJob	Per texel	64
6	MipmapFilterJob ($\times \log_2 N$)	Per mip texel	64

Table 1: Per-frame simulation pipeline with parallelization strategy and batch size. Stage 0 runs only when wind or profile parameters change. Stages 1 to 6 are instantiated once per active cascade.

The per-texel jobs (spectrum generation, temporal evolution, finalization, gradients and mipmaps) use a batch size of 64 elements, balancing scheduling overhead against work distribution granularity. The two FFT passes instead use a batch size of 1, because a single work item already represents a complete $N \log_2 N$ transform of one row or column; grouping several of them per batch would only coarsen load balancing without reducing scheduling overhead meaningfully.

4.4 Mipmap Offset Computation

The mipmap chain is stored in a single contiguous pixel buffer. The write offset for mip level m within this buffer is computed from the geometric series of mip sizes:

$$\text{offset}(m) = \frac{4N^2 - 1}{3} - \frac{4^{(\log_2 N + 1 - m)} - 1}{3} \tag{61}$$

The first term is the total pixel count across all mip levels (the sum $N^2 + (N/2)^2 + \cdots + 1 = (4N^2 - 1)/3$). The second term subtracts the remaining pixels from level m onward, yielding the correct texel offset for contiguous storage. The value indexes the packed 32-bit output array directly, so it is an element index rather than a byte offset.

5 Adaptive Quadtree Rendering

5.1 Hierarchical Level-of-Detail System

The ocean surface utilizes an adaptive quadtree structure to efficiently render vast water areas while maintaining geometric detail near the viewer. The system supports configurable quality presets:

Preset	**Vertices** V	**LOD Scale** τ	**Cull Scale** β
Low	16	2.0	1.2
Medium	32	1.5	1.5
High	64	1.2	1.8
Ultra	64	1.0	2.0

Table 2: Quality presets controlling mesh density and LOD transitions

The maximum number of subdivision levels is derived automatically from the ocean size $S \in [256, 8192]$ by rounding to the nearest power-of-two multiple of a 64 m base cell:

$$L_{\max} = \mathrm{clamp}\left(\mathrm{round}\left(\log_2 \frac{S}{64}\right),\ 2,\ 8\right) \tag{62}$$

where round denotes round-half-to-even, the default rounding mode of the underlying platform call. This produces a subdivision map of width $W = 2^{L_{\max}}$ cells at the finest level, holding W^2 entries.

An explicit override is also exposed, accepting values from 1 to 12. When set, it replaces equation (62) in full, including the $[2, 8]$ clamp, so depths beyond the automatic range are reachable at the cost of a subdivision map that grows as $4^{L_{\max}}$.

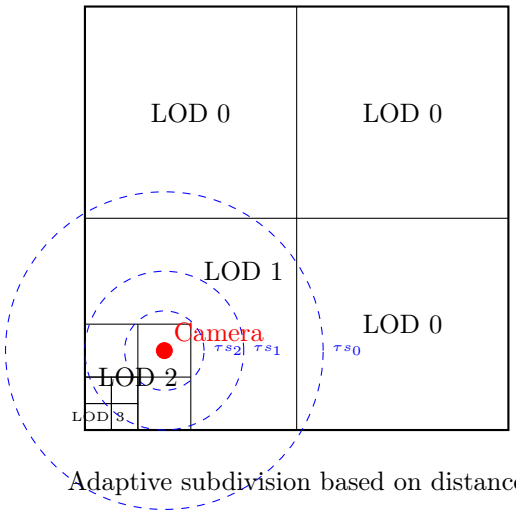


Figure 6: Quadtree subdivision showing LOD levels relative to camera position. Each dashed circle is the subdivision threshold τs_l of equation (63) for one level; since the node size $s_l = S/2^l$ halves with depth, the thresholds form a geometric rather than a uniform sequence.

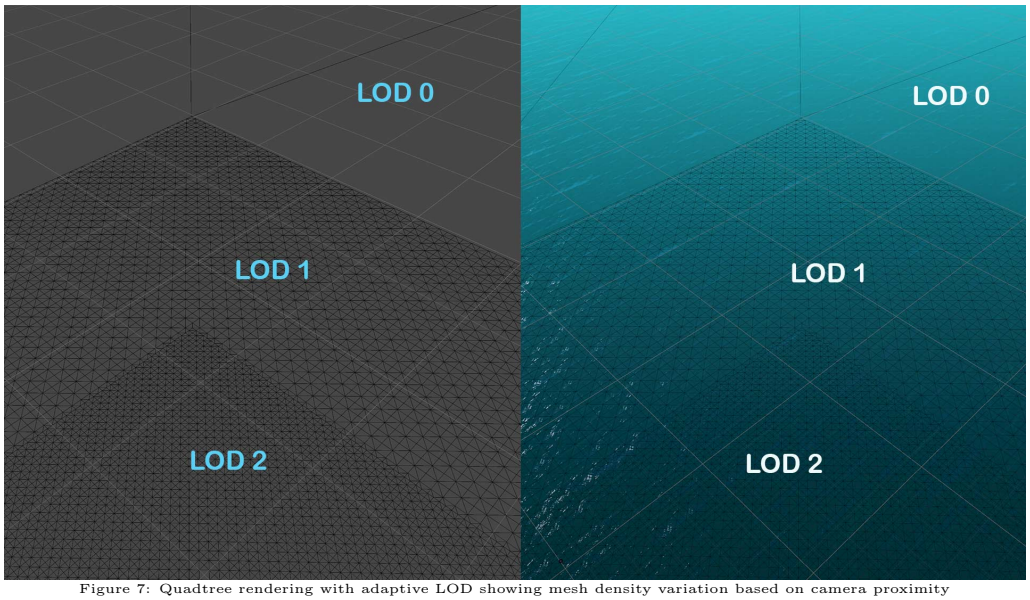


Figure 7: Quadtree rendering with adaptive LOD showing mesh density variation based on camera proximity

5.2 Iterative Quadtree Traversal

The quadtree traversal is implemented as a single-threaded Burst-compiled job using an explicit stack rather than recursion, ensuring deterministic performance. The traversal maintains a stack of node descriptors, each storing grid coordinates (x, y), subdivision level, LOD level, and a rendered flag (Pajarola, 2002). The stack is allocated from the temporary allocator with capacity $4(L_{\max}+1)$, which bounds the working set of a depth-first descent through the tree.

Unlike the simulation jobs, this job is executed synchronously on the calling thread rather than dispatched to workers. It runs once per rendering camera, inside the render callback, so its cost falls on the main thread. This is a deliberate trade: the result is needed immediately to build the instance matrices for that same camera, and a round trip through the job scheduler would cost more than the traversal itself at the supported tree depths. It is nonetheless the only part of the system that is not off the main thread, and it is the term that grows with $L_{\max}$ and with camera count.

5.2.1 Traversal Algorithm

The algorithm begins with the root node at level 0 covering the entire ocean. For each node popped from the stack, the subdivision decision is based on a squared-distance comparison:

$$\text{subdivide} = |\mathbf{c}_{\text{node}} - \mathbf{p}_{\text{viewer}}|^2 < (s_{\text{node}} \cdot \tau)^2 \tag{63}$$

where $\mathbf{c}_{\text{node}}$ is the node center in world space, $s_{\text{node}} = S/2^l$ is the node size at level l, and τ is the LOD distance scale from the quality preset. Both sides use squared quantities to avoid a square root.

The node center is reconstructed from grid coordinates:

$$\mathbf{c}_{\text{node}} = \mathbf{c}_{\text{grid}} - \frac{S}{2}\hat{\mathbf{1}} + (x + 0.5,\ 0,\ y + 0.5) \cdot s_{\text{node}} \tag{64}$$

The processing logic follows three rules:

1. If the node should *not* subdivide and has not been rendered by an ancestor: attempt frustum-culled rendering, and mark the node as rendered

2. If the node is not at leaf level: push four children onto the stack (in reverse order to process lower-left first), each inheriting the parent LOD incremented by one if and only if the parent actually subdivided and had not already been rendered

3. At leaf level: write the LOD value to the subdivision map and render if not yet handled

The distinction between subdivision level and LOD level in rule 2 matters. The level counts depth in the tree and always increments; the LOD counts how many times the surface has genuinely been refined, and stops incrementing once an ancestor has emitted the patch. Descent nevertheless continues to the leaves in that case, because the subdivision map must be filled at the finest resolution for the neighbour queries of Section 5.3.
The `HasRendered` flag propagates from parent to children, preventing double-rendering when a coarser ancestor has already emitted the patch. Because rule 1 marks every non-subdividing node as rendered regardless of the outcome of its frustum test, the inherited state reduces to

$$\texttt{childRendered} = \texttt{parentRendered} \vee \neg\texttt{subdivide} \tag{65}$$

A node that is culled therefore still suppresses rendering in its entire subtree, which is the intended behaviour: an invisible region should emit no geometry at any level.

5.2.2 Frustum Culling

Each node is tested against the six camera frustum planes using an AABB (Axis-Aligned Bounding Box) test. For each plane with normal $\mathbf{n}$ and distance d, the *positive vertex* (the corner farthest in the direction of the normal) is tested:

$$\mathbf{p}_{\text{test}} = \mathbf{c} + \text{select}(-\mathbf{e}, \mathbf{e}, \mathbf{n} \geq 0) \tag{66}$$

where $\mathbf{c}$ is the box center, $\mathbf{e}$ the half-extents, and select is a component-wise conditional. The node is visible only if:

$$\mathbf{n} \cdot \mathbf{p}_{\text{test}} + d \geq 0 \quad \text{for all 6 planes} \tag{67}$$

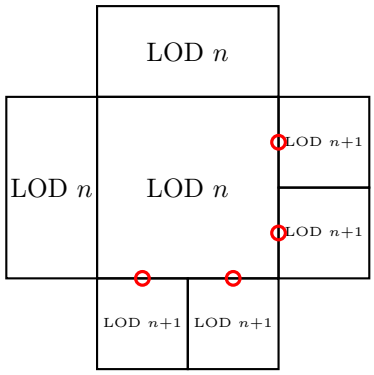
The bounding box extents account for maximum possible wave displacement:

$$\mathbf{e} = \left(\frac{s \cdot \beta}{2}, \quad \frac{h_{\max}}{2}, \quad \frac{s \cdot \beta}{2} \right) \tag{68}$$

where β is the cull scale preset and $h_{\max}$ is the maximum expected displacement height (computed as $S \cdot 0.03$ by default or user-overridden).

5.3 Edge Stitching for Crack Prevention

Adjacent quadtree nodes at different LOD levels create T-junctions where the finer mesh has vertices along an edge that do not exist in the coarser neighbor. These T-junctions manifest as visible cracks in the rendered surface.



T-junctions requiring edge stitching

Figure 8: T-junction formation at LOD boundaries

5.3.1 Neighbor Detection and Stride Computation

For each visible patch at grid position (x, y) and level l, the neighbor LOD levels are read from the subdivision map. The stride at the current level determines where to sample neighbors:

$$\Delta = 2^{(L_{\max} - l)} \tag{69}$$

Neighbor positions in the finest-level map are:

$$\begin{aligned} \text{East}: &\quad (g_x + \Delta,\ g_y) \\ \text{West}: &\quad (g_x - 1,\ g_y) \\ \text{North}: &\quad (g_x,\ g_y + \Delta) \\ \text{South}: &\quad (g_x,\ g_y - 1) \end{aligned} \tag{70}$$

where $g_x = x \cdot \Delta$ and $g_y = y \cdot \Delta$ are the map-space coordinates. Neighbors outside the map bounds are treated as having the same LOD (no stitching needed).

5.3.2 Edge Configuration Flags

Each edge direction is encoded as a single bit in a 4-bit flag word:

$$\text{flags} = \sum_{d \in \{\text{E,N,W,S}\}} 2^{f(d)} \cdot [\text{LOD}_d < l] \tag{71}$$

where $f(\text{E}) = 0$, $f(\text{N}) = 1$, $f(\text{W}) = 2$, $f(\text{S}) = 3$, and $[\cdot]$ is the Iverson bracket evaluating to 1 when the neighbor has a coarser (numerically lower) LOD level. This produces a value in $\{0, \ldots, 15\}$ that uniquely identifies one of the 16 possible edge-stitching configurations.

```
var stride = 1 << (LodLevels - lod);
var gx = patchX * stride;
var gy = patchY * stride;

var width = MapWidth;

var flags = AdjacentLodFlags.None;
flags |= GetNeighborFlag(gx + stride, gy, width, lod, AdjacentLodFlags.East);
flags |= GetNeighborFlag(gx - 1, gy, width, lod, AdjacentLodFlags.West);
flags |= GetNeighborFlag(gx, gy + stride, width, lod, AdjacentLodFlags.North);
flags |= GetNeighborFlag(gx, gy - 1, width, lod, AdjacentLodFlags.South);
```

Listing 4: Neighbour LOD queries producing the 4-bit edge configuration for one patch

5.3.3 Triangulation Patterns

Each patch is divided into a grid of $V/2 \times V/2$ cells, where each cell corresponds to a 3×3 vertex sub-grid.

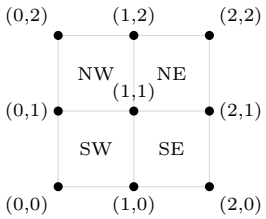


Figure 9: Cell vertex layout for edge-stitching triangulation patterns

Only cells along patch boundaries receive edge-specific flags; interior cells always use the full 8-triangle pattern (flags = 0). The flag for a specific cell is determined by its position within the patch:

$$\text{cellFlags} = \begin{cases} \texttt{West} & \text{if } x = 0 \text{ and patchFlags has West} \\ \texttt{East} & \text{if } x = V/2 - 1 \text{ and patchFlags has East} \\ \vdots & \text{(analogous for North, South)} \end{cases} \tag{72}$$

The 16 triangulation patterns range from the full 8-triangle subdivision (no coarser neighbors) to the minimal 4-triangle fan (all four neighbors coarser). The complete set is listed below.

Flags	Coarser edges	Triangles	Flags	Coarser edges	Triangles
0000_2	none	8	1000_2	S	7
0001_2	E	7	1001_2	S,E	6
0010_2	N	7	1010_2	N,S	6
0011_2	N,E	6	1011_2	N,S,E	6
0100_2	W	7	1100_2	S,W	6
0101_2	W,E	6	1101_2	S,W,E	7
0110_2	N,W	6	1110_2	N,S,W	6
0111_2	N,W,E	7	1111_2	all	4

Table 3: Triangle count for each of the 16 edge configurations. Bit order is E, N, W, S from the least significant bit. The count is not monotone in the number of coarser neighbours: the two configurations with three coarser edges that share an axis (N,W,E and S,W,E) yield 7 triangles, more than several two-edge configurations, because the surviving fine edge still requires its own midpoint.

When an edge is marked for stitching, the midpoint vertex along that edge is omitted from the triangulation, and the adjacent triangles are replaced with a fan that connects directly to the corner vertices. This ensures that the shared edge between adjacent patches at different LOD levels has identical vertex positions, eliminating T-junction cracks.

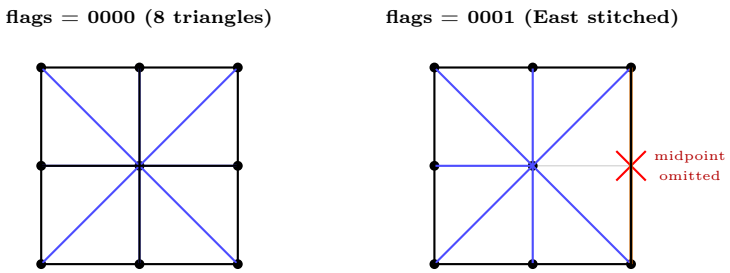


Figure 10: Edge-stitching triangulation comparison: full 8-triangle pattern (left, 8 triangles) versus East-stitched pattern (right, 7 triangles). The east midpoint vertex is omitted and the two triangles that used it are replaced by a fan of three, whose outer member spans the full edge from corner to corner.

5.4 Mesh Generation

Each LOD level uses a regular grid mesh with $(V+1) \times (V+1)$ vertices in unit space $[-0.5, 0.5]^2$:

$$\mathbf{v}_{i,j} = \left(\frac{i}{V} - 0.5,\ \ 0,\ \ \frac{j}{V} - 0.5 \right), \quad i, j \in \{0, \ldots, V\} \tag{73}$$

The mesh is created with 16 submeshes (one per edge configuration). Each submesh's index buffer is generated by iterating over the $V/2 \times V/2$ cell grid, determining the per-cell flags (equation 72), and emitting the corresponding triangle pattern. The vertex index for triangle vertex (v_x, v_y) in cell (c_x, c_y) within the $(V+1) \times (V+1)$ grid is:

$$\text{idx} = (2c_x + v_x) + (2c_y + v_y) \cdot (V + 1) \tag{74}$$

5.5 Grid Snapping

To prevent swimming artifacts during camera movement, the grid origin snaps to discrete cell positions:

$$\mathbf{c}_{\text{snap}} = \left\lfloor \frac{\mathbf{c}_{\text{camera}}}{\delta} \right\rfloor \cdot \delta, \qquad \delta = \frac{S}{2V} \tag{75}$$

The snap quantum $\delta = S/(2V)$ is the vertex spacing of a level-1 patch, not that of the finest level, which is $S/(2^{L_{\max}}V)$. Snapping to this coarser quantum is sufficient in practice because the whole grid translates rigidly: every patch, at every level, shifts by the same amount, so relative vertex alignment between neighbouring patches is preserved regardless of δ. What δ actually controls is how far the grid may drift from the camera before it jumps, and therefore how much the sampled wave texture appears to swim during motion. A smaller quantum would reduce that residual further at the cost of more frequent re-evaluation of the quadtree. The snap operates independently on the x and z axes.

5.6 Instance Batching

Patches sharing the same edge configuration are rendered using GPU instancing to minimize draw calls. The renderer maintains 16 transformation matrix arrays, one per submesh configuration:

$$\mathbf{M}_{\text{instance}} = \text{TRS}(\mathbf{c}_{\text{patch}},\ \mathbf{I},\ s_{\text{patch}}) \tag{76}$$

where $\mathbf{c}_{\text{patch}}$ is the world-space patch center (computed from grid position, level, and snapped center), $\mathbf{I}$ is the identity rotation, and $s_{\text{patch}} = S/2^l$ is the uniform scale for level l.
Exactly one batch is maintained per configuration, holding at most 128 instances. The total cost is therefore bounded at 16 instanced draw calls per camera, one per non-empty configuration, in place of potentially hundreds of individual draws. The bound is enforced rather than grown: a patch arriving at a configuration whose batch is already full is discarded with a warning instead of opening a second batch. At the supported ocean sizes and quality presets the visible patch count stays well inside this limit, but the behaviour is a hard cap rather than a soft one, and raising $L_{\max}$ through the override of Section 5.1 can reach it.

5.7 Skirting Mesh

For horizon rendering beyond the main quadtree grid, a strip mesh extends from the ocean boundary to a configurable distance (default $10 \times S$). The mesh consists of 3 vertex pairs forming two quads, spanning from inner extent $S/2$ to the outer extent. Four draw calls are issued per camera, one per cardinal direction, each with a rotation matrix that orients the same strip outward; the strip is drawn directly rather than instanced, since four draws do not justify an instance buffer. The result is a continuous horizon without additional LOD overhead. The skirt is skipped entirely when the configured outer extent does not exceed the ocean size.

References

Cooley, James W. and John W. Tukey (1965). "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19.90, pp. 297–301. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1965-0178586-1. URL: https://www.ams.org/mcom/1965-19-090/S0025-5718-1965-0178586-1/ (visited on 05/07/2025).

Pajarola, Renato (2002). "Overview of quadtree-based terrain triangulation and visualization". In: URL: https://escholarship.org/uc/item/24h2g848 (visited on 03/05/2025).

Tessendorf, Jerry (1999). *Simulating Ocean Water*. URL: https://evasion.inrialpes.fr/Membres/Fabrice.Neyret/NaturalScenes/fluids/water/waves/fluids-nuages/waves/Jonathan/articlesCG/simulating-ocean-water-01.pdf (visited on 05/18/2025).