Animating Water Using Profile Buffer

Haozhe Su, Wei Li, Zherong Pan,
Xifeng Gao, Zhenyu Mao, and Kui Wu

21.1 Introduction

Water plays a crucial role in video games, contributing to both the visual and interactive aspects of the gaming experience. One technique to create a realistic water surface animation is via a flowmap, a 2D velocity field texture, to advect the water displacement and normal textures at runtime. More details about flowmaps can be found in [Vlachos 10]. While the flowmap technique stands out for its computational efficiency in animating water within video games, it is not without limitations. One notable drawback is the potential for large distortion and stretching artifacts, particularly in regions where the velocity magnitude is substantial. This article introduces a novel approach to real-time water animation, leveraging a wavelet-based methodology enhanced by a precomputed profile buffer technique for optimal performance. The proposed technique aims to create water waves dynamically based on factors such as world position, time, and velocity present in the flowmap.

21.2 Background

In this section, we delve into the nuances of spectrum-based methods for simulating water waves, initially introduced in [Jeschke et al. 18]. Unlike partial differential equation (PDE)–based methods that rely on precise spatial discretization, linear wave theory [Johnson 97] describes the wave height dynamics $\eta(\mathbf{x}, t)$ at the position $\mathbf{x}$ in the 2D simulation domain with time t that can be articulated in terms of frequencies as

$$\eta(\mathbf{x}, t) = \int_{\mathrm{R}^2} \mathcal{A}(\mathbf{x}, \mathbf{k}, t) \cos\left(\mathbf{k} \cdot \mathbf{x} - \omega(k)t\right) d\mathbf{k}, \tag{21.1}$$

where $\mathcal{A}$ is the amplitude function that depends on $\mathbf{k}$, $\mathbf{x}$, and t and the term $\cos\left(\mathbf{k} \cdot \mathbf{x} - \omega(k)t\right)$ represents a traveling wave. The wave vector $\mathbf{k}$ is

a two-dimensional frequency function such that $k = |\mathbf{k}|$ is the wave number and $\hat{\mathbf{k}} = \mathbf{k}/k$ is the wave direction. The angular frequency $\omega(k) = \sqrt{gk}$ represents the dispersion relation for deep water waves by encoding the speed of each wave based on its wave number k and gravity g. To further elaborate on the formula by employing the polar representation of the wave vector $\mathbf{k}$, we have

$$\eta(\mathbf{x}, t) = \int_0^{2\pi} \int_0^{\infty} \mathcal{A}(\mathbf{x}, \mathbf{k}, t) \cos\left(\mathbf{k} \cdot \mathbf{x} - \omega(k)t\right) k dk d\theta. \tag{21.2}$$

For more information, please refer to [Jeschke et al. 18].

21.3 Our Method

In this section, we first introduce our decomposition of the amplitude field $\mathcal{A}$ and then describe the discretization of each decomposed component.

21.3.1 Amplitude Field Decomposition

Unlike the previous work [Jeschke et al. 18] that evolves the amplitude field $\mathcal{A}(\mathbf{x}, \mathbf{k}, t)$ in each time step to accurately depict wave movements, our input water flow remains static and is specified by a precomputed flowmap, such that the amplitude field $\mathcal{A}$ is only affected by the position and wave vector. Furthermore, we introduce the following novel decomposition of the amplitude field:

$$\mathcal{A}(\mathbf{x}, \mathbf{k}) = A \; W(\mathbf{x}, \theta) \; \psi(k), \tag{21.3}$$

where A serves as a constant factor for adjusting the wave strength, $W(\mathbf{x}, \theta)$ represents the weight associated with the wave direction θ at spatial position $\mathbf{x}$, and $\psi(k)$ denotes the basis function of the wave. In the remainder of this section, we aim to provide insight into this simplified wave model by offering detailed explanations of each component.

Direction Weight $W(\mathbf{x}, \theta)$ We denote the time-independent direction at the location $\mathbf{x}$ from the flowmap as the primary wave direction θ_p:

$$\theta_p(\mathbf{x}) = \arctan\left(\frac{v_x}{v_z}\right), \tag{21.4}$$

where v_x and v_z are the velocity components along the x- and z-axis, respectively. By assuming that the wave direction is in the vicinity of the primary

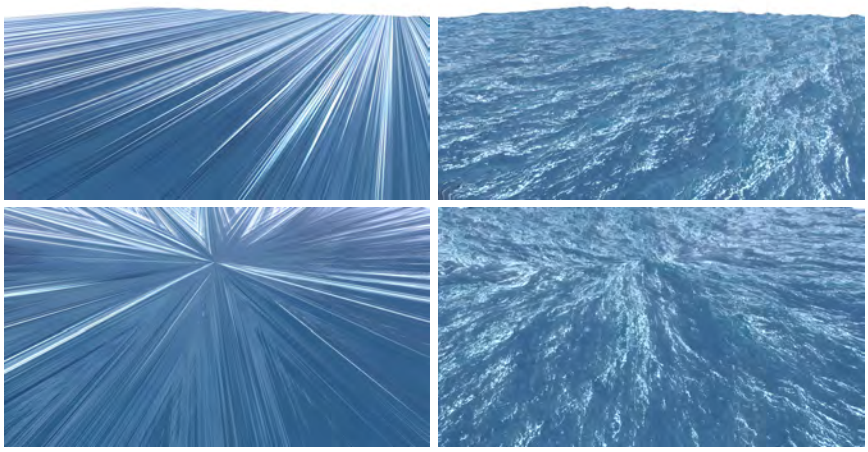

Figure 21.1. Top: Uniform water flow from left to right with consistent strength. Bottom: Circular water flow. For both scenarios, the left result uses a small relative angle range $[-\frac{\pi}{8}, \frac{\pi}{8}]$, whereas the right uses a large relative angle range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

wave direction, the corresponding weight $W(\mathbf{x}, \theta)$ can be computed as

$$W(\mathbf{x}, \theta) = \begin{cases} 1 - \frac{|\theta - \theta_p(\mathbf{x})|}{C}, & \text{if } |\theta - \theta_p(\mathbf{x})| \leq C; \\ 0, & \text{otherwise;} \end{cases} \tag{21.5}$$

where C is the half-width of the angle range with nonzero weight, which will be addressed in Section 21.4.3. The farther the angle deviates from the primary angle, the smaller the weight becomes. Figure 21.1 illustrates comparisons between utilizing smaller and larger angle ranges with nonzero weights.

Wave Basis $\psi(k)$ In the frequency space, $\psi(k)$ symbolizes the wave shape with a representative wave number k. Although any wave spectrum can be used as the wave basis, we choose to adopt the Phillips spectrum (Listing 21.1) as the basis for our waves.

Substituting Equation (21.3) into Equation (21.2) yields

$$\begin{aligned} \eta(\mathbf{x}, t) &= \int_0^{2\pi} \int_0^{\infty} AW(\mathbf{x}, \theta)\psi(k) \cos\left(\mathbf{k} \cdot \mathbf{x} - \omega(k)t\right) k dk d\theta \\ &= \int_0^{2\pi} AW(\mathbf{x}, \theta) \left(\int_0^{\infty} \psi(k) \cos\left(\mathbf{k} \cdot \mathbf{x} - \omega(k)t\right) k dk \right) d\theta. \end{aligned} \tag{21.6}$$

```
// Wave basis
float phillips_spectrum(float waveNumber)
{
    float windSpeed = 10.f;
    float A = 2.f * PI / waveNumber;
    float B = expf(-1.8038f * A * A / powf(windSpeed, 4));
    return 0.1391f * sqrtf(A * B);
}
```

Listing 21.1. The Phillips spectrum.

Profile Buffer $\bar{\Psi}(p,t)$ Performing a double integral to calculate $\eta(\mathbf{x},t)$ via Equation (21.6) for each pixel in the domain can be time-consuming, especially for high-resolution animations. Following the idea proposed by [Jeschke et al. 18], we define the profile buffer $\bar{\Psi}(p,t)$:

$$\bar{\Psi}(p,t) = \int_0^\infty \psi(k)\cos\left(kp - \omega(k)t\right)kdk, \tag{21.7}$$

where $p = \hat{\mathbf{k}} \cdot \mathbf{x}$ is the projected position. By precomputing the integral of $\bar{\Psi}(p,t)$, the original double integral in Equation (21.6) can be converted into a single integral:

$$\eta(\mathbf{x},t) = \int_0^{2\pi} AW(\mathbf{x},\theta)\bar{\Psi}(p,t)d\theta. \tag{21.8}$$

It is important to note that the profile buffer $\bar{\Psi}(p,t)$ is a function of both p and t, requiring updates in every time step.

21.3.2 Discretization

Discretization of $\bar{\Psi}(p,t)$ In theory, the integral of the profile buffer involves an infinite number of calculations when converted to the summation form. In practice, we approximate Equation (21.8) with a finite number of summations as

$$\bar{\Psi}(p,t) = \sum_{i=0}^{N_k-1} \psi(k_i)\cos\left(k_i p - \omega(k_i)t\right)k_i \Delta k, \tag{21.9}$$

where N_k represents the total number of sampled wave number values taken into account.

The profile buffer is designed to be periodic, ensuring seamless tiling across all of space. We introduce a spatial periodicity parameter L and define the normalized projected position $\bar{p} \in [0,1)$ as

$$\bar{p} = \frac{p - \lfloor p/L \rfloor L}{L}. \tag{21.10}$$

With this normalization, we can express the profile buffer in a reformulated manner:

$$\bar{\Psi}(p,t) = \bar{\Psi}(\bar{p},t) = \sum_{i=0}^{N_k-1} \psi(k_i)\cos\left(k_i L\bar{p} - \omega(k_i)t\right)k_i\Delta k. \tag{21.11}$$

We can store the profile buffer result in this 1D array by uniformly sampling N_k points in the interval $[0,1]$ as $\bar{p}_i$. The computation of the jth entry can be achieved via

$$\bar{\Psi}(\bar{p}_j,t) = \sum_{i=0}^{N_k-1} \psi(k_i)\cos\left(k_i L\bar{p}_j - \omega(k_i)t\right)k_i\Delta k, \tag{21.12}$$

where $\bar{p}_j = (j+0.5)/N_k$ and $j = 0,1,2,\ldots,N_k-1$. With lower and upper limits of the wave number, $k_{\min}$ and $k_{\max}$, we have $\Delta k = (k_{\max}-k_{\min})/N_k$ and $k_i = k_{\min} + (i+0.5)\Delta k$.

Discretization of $\eta(\mathbf{x},t)$ Similarly, we discretize the wave direction θ into N_θ samples and rewrite Equation (21.8) as

$$\eta(\mathbf{x},t) = \frac{2\pi A}{N_\theta}\sum_{i=0}^{N_\theta-1} W(\mathbf{x},\theta_i)\bar{\Psi}(\hat{\mathbf{k}}_i\cdot\mathbf{x},t), \tag{21.13}$$

where $\theta_i = 2\pi i/N_\theta$, $\mathbf{x} = (z,x)$, and $\hat{\mathbf{k}}_i = (\cos\theta_i, \sin\theta_i)$.

21.4 Implementation Details

In this section, we delineate the numerical algorithm for solving Equation (21.8), along with specific code implementations. At each time step, two primary tasks must be executed: computing the profile buffer and performing height integration. Refer to Listing 21.2 for an overview of the algorithm pipeline. In our implementation, we use a CUDA kernel to compute the profile buffer at the beginning of each frame and send it to OpenGL as a 1D texture via CUDA OpenGL `interop`.

```
t ← 0
Initialization                                    // Section 21.4.1
while t < T
{
    Precompute profile buffers                    // Section 21.4.2
    Integrate water height in vertex shader       // Section 21.4.3
    Integrate water normal in fragment shader     // Section 21.4.4
    t ← t + Δt
}
```

Listing 21.2. Pseudocode of our water wave animation.

```
// Physical constants
#define PI                          3.14159265359f
#define TAU                         6.28318530718f
#define GRAVITY                     9.81f

// Profile buffer constants
#define DIR_NUM                     32
#define SEG_PER_DIR                 8
#define PB_RESOLUTION               4096
#define WAVE_DIM                    4
#define FINE_DIR_NUM                (DIR_NUM * SEG_PER_DIR)
```

Listing 21.3. Constants.

21.4.1 Initialization

We initialize the simulation through the following steps. Firstly, as both the terrain and water are represented by height maps, we import these height maps to construct the scene. Secondly, a flowmap, which records the fluid velocity field at a stable state, is imported to indicate the wave propagation direction and strength within the simulation domain. Lastly, we define several macros to control the simulation details, as depicted in Listing 21.3.

21.4.2 Precomputing the Profile Buffer

Profile buffer computation stands as a crucial element in achieving real-time wave animation. Rather than performing intensive summations at all pixels/vertices in the domain at every time step, we calculate the profile buffer at the onset of each time step. We use the Phillips spectrum as the wave basis $\psi(k)$ mentioned in Section 21.1.

A 1×4096 texture with four channels (`WAVE_DIM=4`) is used as the profile buffer, in which displacement and spatial derivatives along vertical and horizontal directions are stored at four channels, respectively. Note that the cosine term in Equation (21.9) indicates the vertical displacement of the Gerstner wave, while displacement and spatial derivatives can be computed as demonstrated in Listing 21.4.

```
void gerstner_wave(float* gw, float phase, float k)
{
  float s = sin(phase);
  float c = cos(phase);

  gw[0] = -s;       // Horizontal displacement
  gw[1] =  c;       // Vertical displacement
  gw[2] = -k * c;   // Derivative of gw[0]
  gw[3] = -k * s;   // Derivative of gw[1]
```

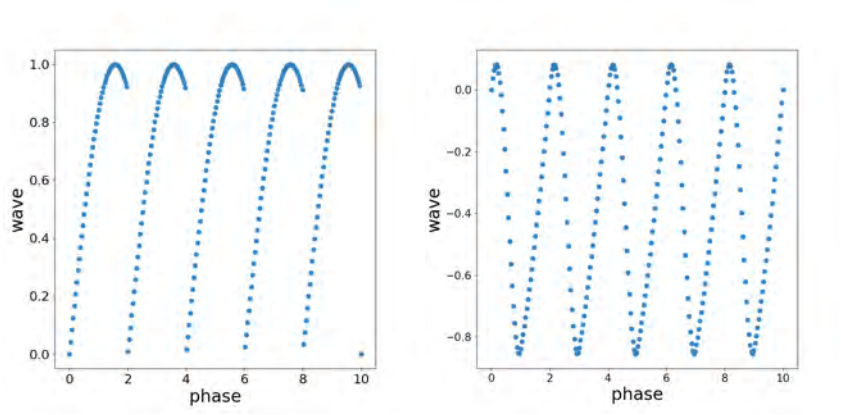

Figure 21.2. Left: Simplified wave without interpolation. Discontinuities arise at the boundary between adjacent profile buffers. Right: Interpolated wave.

```
}
```

Listing 21.4. The Gerstner wave.

To simulate water waves across large open worlds, we construct the profile buffers to be periodic for seamless extension in the world space. This configuration could lead to potential discontinuities near the borders of neighboring profile buffers, as illustrated in Figure 21.2. To resolve this, we introduce cubic bumps:

$$\texttt{cubic\_bump}\,(x) = \begin{cases} x^2(2|x| - 3) + 1, & \text{if } \ |x| < 1; \\ 0, & \text{otherwise;} \end{cases} \tag{21.14}$$

with which we can interpolate the Gerstner wave without concern for discontinuity, as shown in Listing 21.5.

Equation (21.9) can now be computed straightforwardly, as shown in Listing 21.6.

Fast/Slow Profile Tuning To gain finer control over wave behavior, we introduce the profile scale s to adjust the wave propagation speed. The modified dispersion relation $\omega(k)$ can then be expressed as

$$\omega(k) = s\sqrt{gk}. \tag{21.15}$$

A larger $\omega(k)$ corresponds to faster wave propagation. In practice, we utilize two distinct values, `s_fast` and `s_slow`, to represent fast and slow wave propagation, respectively. The profile buffers are then stored in two separate textures: `d_PBf_field_` and `d_PBs_field_`.

```
void interpolate_gerstner_wave(
    real* gw, real k, real p, real L, real scale, real t)
{
    real phase1 = k * p - disper_relation(k, scale) * t;
    real phase2 = k * (p - L) - disper_relation(k, scale) * t;

    real gw1[WAVE_DIM];
    real gw2[WAVE_DIM];

    gerstner_wave(gw1, phase1, k);
    gerstner_wave(gw2, phase2, k);

    real weight1 = p / L;
    real weight2 = 1 - weight1;

    real cb1 = cubic_bump(weight1);
    real cb2 = cubic_bump(weight2);

    for (int i = 0; i < WAVE_DIM; ++i)
    {
        gw[i] = cb1 * gw1[i] + cb2 * gw2[i];
    }
}
```

Listing 21.5. Gerstner wave interpolation.

```
void compute_profile_buffer(
    real* result, real k, real p, real L, real scale, real t)
{
    real waveLength = TAU / k;

    real gw[WAVE_DIM];
    interpolate_gerstner_wave(gw, k, p, L, scale, t);

    real ps = phillips_spectrum(k);
    for (int i = 0; i < WAVE_DIM; ++i)
    {
        result[i] += waveLength * ps * gw[i];
    }
}
```

Listing 21.6. Profile buffer computation.

21.4.3 Height Integration

Given the computed profile buffer, we integrate the displacement of the water surface vertex via Gaussian quadrature on the fly in the vertex shader. In particular, given a vertex position `inPosition` at its rest state, we integrate its world position based on the precomputed profile buffers

`pbOFieldTex` and flowmap `layerDataTex`.

Integration with Gaussian Quadrature Equation (21.13) represents the classical Riemann sum method designed for numerical integration. However, it necessitates a greater number of summations to attain high precision.

We introduce Gaussian quadrature to provide an exact result for polynomials of degree $2N_{\text{int}}-1$ or lower through a suitable selection of nodes P_i and weights α_i for $i = 0, \ldots, N_{\text{int}}-1$. It's important to note that Gaussian quadrature can be applied without modification when the original limits of integration are $[-1, 1]$. In this scenario, the positions of integration points $P_0, P_1, \ldots, P_{N_{\text{int}}-1}$ as well as their associated weights $\alpha_0, \alpha_1, \ldots, \alpha_{N_{\text{int}}-1}$ can be precomputed, without requiring knowledge of the specific expression of the integrand. With this, we can transform the original Riemann sum into a new form:

$$\eta(\mathbf{x}, t) = s_G A \sum_{i=0}^{N_{\text{int}}-1} \alpha_i W(\mathbf{x}, \theta_i) \bar{\Psi}(z \cos\theta_i + x \sin\theta_i, t), \tag{21.16}$$

where $\theta_i = \pi P_i + \pi$ and $s_G = \frac{2\pi}{2} = \pi$ is a scaling factor. When the limits of integration change to a custom range, the integration requires scaling, and the scaling factor is simply the ratio of the actual limit length to the original limit length, which is 2 for the range $[-1, 1]$. In the experiment, we utilize 32 points of integration. The precomputed Gaussian quadrature point positions and their associated weights are provided in Listing 21.7.

```
int nnodes = 32;
float nodes[32] = {
    -0.9972638618494816f,   -0.9856115115452684f,
    -0.9647622555875064f,   -0.9349060759377397f,
    -0.8963211557660522f,   -0.84936761373257f,
    -0.7944837959679424f,   -0.7321821187402897f,
    -0.6630442669302152f,   -0.5877157572407623f,
    -0.5068999089322294f,   -0.42135127613063533f,
    -0.33186860228212767f,  -0.23928736225213706f,
    -0.1444719615827965f,   -0.04830766568773831f,
    0.04830766568773831f,   0.1444719615827965f,
    0.23928736225213706f,   0.33186860228212767f,
    0.42135127613063533f,   0.5068999089322294f,
    0.5877157572407623f,    0.6630442669302152f,
    0.7321821187402897f,    0.7944837959679424f,
    0.84936761373257f,      0.8963211557660522f,
    0.9349060759377397f,    0.9647622555875064f,
    0.9856115115452684f,    0.9972638618494816f};
float weights[32] = {
    0.007018610009469298f,  0.016274394730905965f,
    0.025392065309262427f,  0.03427386291302626f,
    0.042835898022226426f,  0.050998059262376244f,
    0.05868409347853504f,   0.06582222277636175f,
    0.07234579410884845f,   0.07819389578707031f,
    0.08331192422694685f,   0.08765209300440391f,
```

```
    0.09117387869576386f,   0.09384439908080457f,
    0.09563872007927483f,   0.09654008851472781f,
    0.09654008851472781f,   0.09563872007927483f,
    0.09384439908080457f,   0.09117387869576386f,
    0.08765209300440391f,   0.08331192422694685f,
    0.07819389578707031f,   0.07234579410884845f,
    0.06582222277636175f,   0.05868409347853570 4f,
    0.05099805926237624 4f, 0.04283589802222642 6f,
    0.03427386291302162 6f, 0.02539206530926242 7f,
    0.01627439473090596 5f, 0.00701861000946929 8f};
```

Listing 21.7. Gaussian quadrature points and weights.

Next, we provide a detailed implementation for computing each entry in the summation in Equation (21.16) at the vertex position `inPosition`. Assuming we have computed the primary angle $\theta_p = \arctan(v_x/v_z)$, we can calculate the weight $W(\mathbf{x}, \theta_i)$ as follows:

```
float W = 1 - min(C, min(diffa, TAU-diffa))/C;
```

In our experiments, blending two profiles enhances the portrayal of wave propagation details. Depending on the strength of velocity, we define a weight function $w = \min(1, |\mathbf{v}|)$, ensuring that the fast profile buffer occupies a larger proportion as the magnitude of the current velocity increases. Readers may have observed that the retrieved profile buffer comprises four components, which are arranged in their original order as the horizontal and vertical displacements of both the slow and fast profile buffers. The blended profile buffer can be represented as:

```
vec2 pbb = (1.f - scale) * pb.xy + scale * pb.wz;
```

Here, `pb.xy` and `pb.wz` represent the displacement parts of the slow and fast profile buffer, respectively.

Adaptive Limits of Integration We assume that the wave only propagates within a range near the primary angle, as defined in Equation (21.4). Expanding this range results in more waves in different directions blending together, whereas narrowing it makes the wave appear to move predominantly in one direction. We have demonstrated in Figure 21.1 that a narrow range can lead to artificial constructive patterns. Conversely, an excessively wide range can obscure the recognition of wave direction, resulting in smoothed-out wave propagation. To dynamically adjust the angle range of wave propagation, we calculate the maximum difference in velocity direction at each cell center with its neighboring cells, which serves as an indicator of local velocity variation.

```
for each cell i in simulation domain
{
    M_i ← Neighboring cells of cell i
```

Figure 21.3. Adaptive limits of integration: a flowmap characterized by uniform strength and a radial outward direction from a central point is evaluated. From left to right, we employ a fixed small integration range, an adaptive integration range, and a fixed large integration range, respectively. Prominent artificial patterns are discernible on the left, while the wave propagation appears excessively smooth, hindering clear recognition on the right. Our model depicted in the middle exemplifies the outward movement of waves with realistic turbulence.

```
    θ_i ← arctan(v_i/u_i)
    Δθ_max ← 0
    for each cell j ∈ M_i
    {
        θ_j ← arctan(v_j/u_j)
        Δθ ← ||θ_j − θ_i||
        Δθ ← min(Δθ, 2π − Δθ)
        if Δθ > Δθ_max
        {
            Δθ_max ← Δθ
        }
    }
}
```

Listing 21.8. Pseudocode for local velocity variation computation.

With primary angles calculated and the maximum difference in velocity direction precomputed, the subsequent step involves determining the limits of integration associated with nonzero weights. We operate under the assumption that within a given area, if the local velocity variation is significant, we require a larger range of integration angles to accurately represent the varying wave propagation directions. As previously mentioned, we partition all possible propagation directions within the range $[0, 2\pi]$ into `DIR_NUM` segments. Each segment is further subdivided into `SEG_PER_DIR` pieces, resulting in a total of `DIR_NUM * SEG_PER_DIR` subdivisions across the full 2π range. Subsequently, we determine which subdivisions correspond to nonzero weights. We calculate the half-width of the nonzero weight area, denoted as `C`:

```
int INT_NUM = min(16,4+int(500*delta_mag/TAU));
int SEG_HW = SEG_PER_DIR*INT_NUM;
float C = float(SEG_HW)/float(FINE_DIR_NUM)*TAU;
```

Listing 21.9. Compute adaptive limits of integration.

21.4.4 Normal Integration

Surface normal calculation is indispensable for visually representing the surface shape of waves in rendering. Theoretically, the surface normal of a height map can be computed via $\mathbf{n} = \frac{\mathbf{T}_z \times \mathbf{T}_x}{\|\mathbf{T}_z \times \mathbf{T}_x\|}$, where $\mathbf{T}_z$ and $\mathbf{T}_x$ are the gradient vectors in the z/x-direction (in other words, the horizontal directions in OpenGL), respectively. Given the integration process for wave height from Equation (21.8), we can calculate the gradient of the height map concerning the horizontal position:

$$\nabla_{\mathbf{x}}\eta(\mathbf{x},t) = \nabla_{\mathbf{x}} \int_0^{2\pi} AW(\mathbf{x},\theta)\bar{\Psi}(p,t)d\theta, \tag{21.17}$$

$$= \int_0^{2\pi} \nabla_{\mathbf{x}} \left(AW(\mathbf{x},\theta)\right) \bar{\Psi}(p,t)d\theta + \int_0^{2\pi} AW(\mathbf{x},\theta)\nabla_{\mathbf{x}} \left(\bar{\Psi}(p,t)\right) d\theta, \tag{21.18}$$

We adopt the approach suggested in [Jeschke et al. 18], wherein we omit the first term. This decision is supported by the observation that the flowmap maintains continuity in space, indicating that the derivative $\nabla_{\mathbf{x}} \left(A\omega(\mathbf{x},\theta)\right)$ tends to be close to 0. We can now finalize the formula for calculating height gradients as follows:

$$\nabla_{\mathbf{x}}\eta(\mathbf{x},t) = \int_0^{2\pi} AW(\mathbf{x},\theta)\nabla_{\mathbf{x}}\bar{\Psi}(p,t)d\theta. \tag{21.19}$$

The calculation of the gradient vectors $\mathbf{T}_z$ and $\mathbf{T}_x$ can be executed as:

$$\mathbf{T}_z = (1, 0, \nabla_{\mathbf{x}}\eta_z) \Rightarrow \mathbf{T}_z = \mathbf{T}_z/||\mathbf{T}_z||, \tag{21.20}$$

$$\mathbf{T}_x = (0, 1, \nabla_{\mathbf{x}}\eta_x) \Rightarrow \mathbf{T}_x = \mathbf{T}_x/||\mathbf{T}_x||. \tag{21.21}$$

Note that we have not taken into account the horizontal displacement, but only the vertical displacement. Normal integration procedure is executed within the fragment shader, and the pipeline closely resembles that of height integration, albeit with the additional step of integrating $\mathbf{T}_z$ and $\mathbf{T}_x$ and subsequently employing the cross-product operation to derive the normal vector.

```
... // Initial setup
vec3 tz = vec3(0.f, 0.f, 1.f);
vec3 tx = vec3(1.f, 0.f, 0.f);
for(int sid = 0; sid < nnodes; ++sid)
{
    ... // Normal integration
}
```

```
vec3 newNormal = cross(tz, tx);
newNormal = normalize(newNormal);
```

Listing 21.10. Normal integration.

21.5 Results

In this section, we demonstrate the effectiveness of our method. In particular, we showcase how the profiling time is significantly reduced through the implementation of Gaussian quadrature. Additionally, we elucidate how adaptive limits of integration contribute to overall quality improvement and highlight how blending two profile buffers enhances the depiction of wave movements. We utilize a custom-designed map featuring a river scene for all demonstrations and comparisons, as shown in Figure 21.4. All tests have been conducted using a 4.0 GHz AMD Ryzen Threadripper PRO 5955WX 16-Cores processor paired with an NVIDIA GeForce RTX 4080 GPU with 16 GB of memory.

Figure 21.4. Left: terrain height map. The whiter areas indicate higher elevations. **Middle: flowmap.** The dark purple shades represent lower velocity magnitudes, while the bright yellow shades represent higher velocity magnitudes. **Right: top view of the rendered scene.**

21.5.1 Riemann Sum vs. Gaussian Quadrature

We observe that Gaussian quadrature with 32 integration points achieves visual results comparable to those of Riemann sum with 192 integration points. Notably, this achievement is accompanied by a significant reduction in profiling time by 63%, highlighting the efficiency of our proposed method. Moreover, to achieve a similar profiling time, Riemann sum is constrained to using only 32 points, resulting in a compromised visual appearance. This trade-off underscores the superiority of Gaussian quadrature in balancing

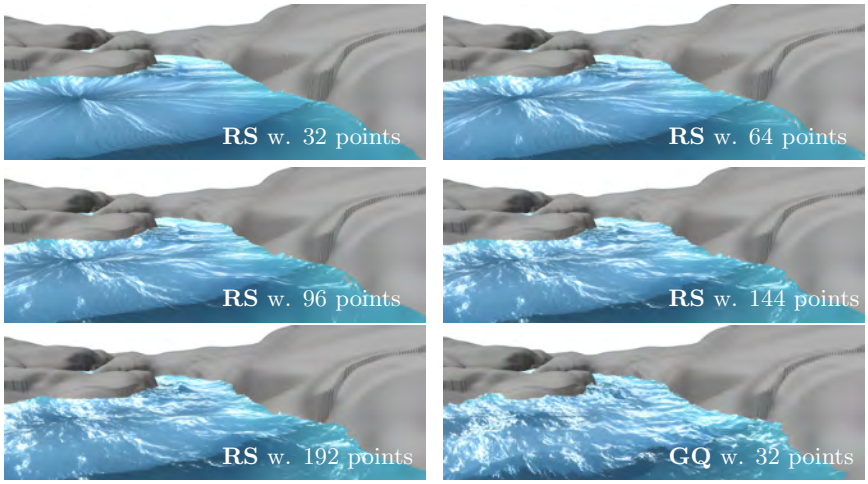

Figure 21.5. Comparison between Riemann sum (**RS**) with different numbers of integration points (32, 64, 96, 144, 192) and Gaussian quadrature (**GQ**). The Riemann sum exhibits noticeable unnatural patterns with an insufficient number of integration points (32). Furthermore, the Riemann sum method requires approximately six times the number of integration points to attain comparable results in visual fidelity and detail.

Integration Method	# integration points	Runtime (ms)
Riemann sum	32	3.01
	64	4.65
	96	4.91
	144	6.46
	192	8.26
Gaussian quadrature	32	3.03

Table 21.1. Performance comparison between the Riemann sum with different numbers of integration points and the Gaussian quadrature.

computational efficiency with visual quality. Please refer to the results depicted in Figure 21.5.

21.5.2 Adaptive Limits of Integration

Next, we delve into the adaptive limits of integration, showcasing its impact on visual appearance. During per-pixel normal integration and per-vertex height integration, the flowmap indicates the local flow orientation, and we set the maximum amplitude strength to appear along this direction. The

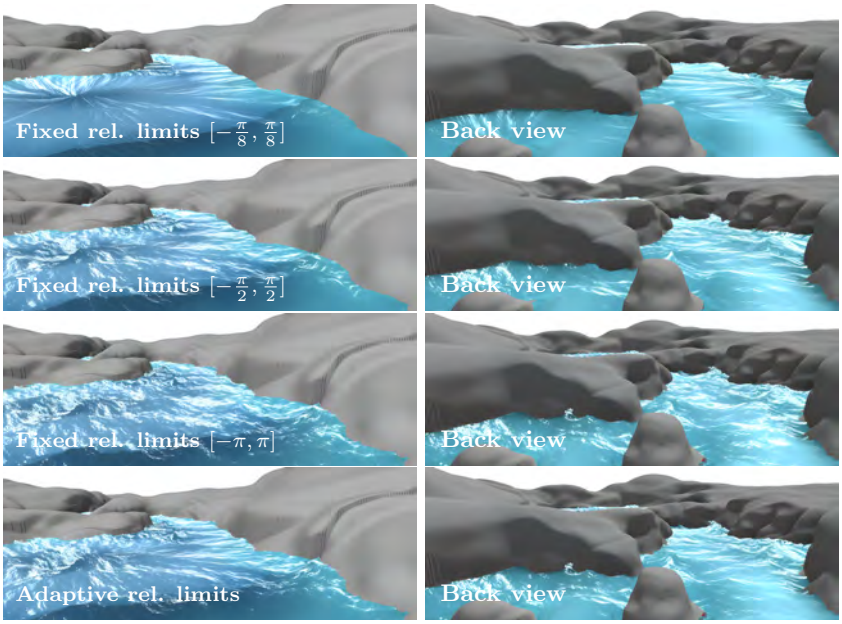

Figure 21.6. Adaptive integration limits are examined in comparison to fixed limits of integration (relative ranges include $[-\frac{\pi}{8}, \frac{\pi}{8}], [-\frac{\pi}{2}, \frac{\pi}{2}], [-\pi, \pi]$). With a small integration range, anomalous constructive patterns emerge, while a large range results in overly smoothed wave movement and vague wave propagation direction. Adaptive integration limits effectively mitigate observable constructive structures and enhance the clarity of wave propagation direction.

amplitude strength linearly diminishes as the direction deviates further from the central orientation. The proposed adaptive strategy decides the maximum nonzero amplitude range, based on variation in local flowmap. Please refer to the results illustrated in Figure 21.6.

21.5.3 Fast/Slow Wave Propagation Blending

Next, we compare wave behaviors characterized by fast propagation speed, slow propagation speed, and the blending of the two, illustrated in Figure 21.7.

21.6 Conclusion

We have introduced a real-time framework for simulating waves in open-world scenarios. Central to our approach is the profile buffer, which effi-

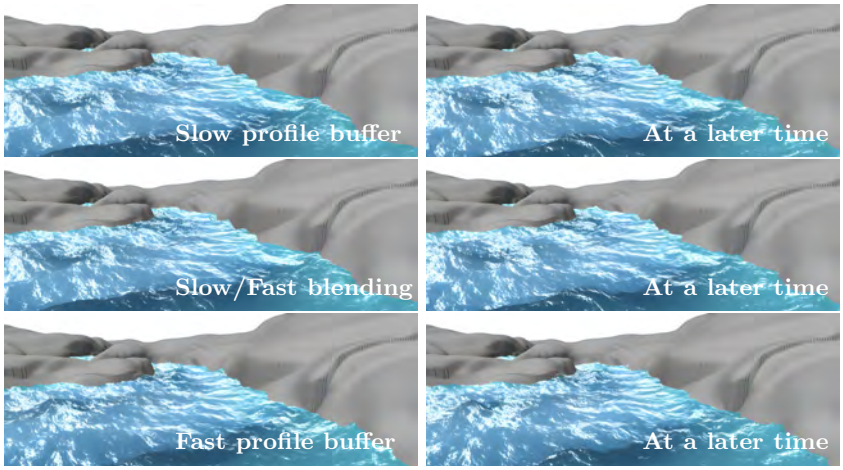

Figure 21.7. We compare the visual appearance of using (**Top**) slow profile buffer, (**Bottom**) fast profile buffer as well as (**Middle**) blending of slow/fast profile buffers.

ciently transforms computationally-intensive integration tasks into reusable texture fetching operations. By incorporating Gaussian quadrature, we achieve significant speed improvements while maintaining visual quality. Furthermore, the adaptive limits of integration enable us to achieve diverse visual effects, while the blending of profile buffers enriches wave propagation details.

21.7 Acknowledgement

The authors would like to specially thank Stefan Jeschke and Chris Wojtan for their support and guidance.

Bibliography

[Jeschke et al. 18] Stefan Jeschke, Tomáš Skřivan, Matthias Müller-Fischer, Nuttapong Chentanez, Miles Macklin, and Chris Wojtan. "Water surface wavelets." *ACM Trans. Graph.* 37:4.

[Johnson 97] R. S. Johnson. *A Modern Introduction to the Mathematical Theory of Water Waves*. Cambridge Texts in Applied Mathematics, Cambridge University Press, 1997.

[Vlachos 10] Alex Vlachos. "Water Flow in Portal 2.", 2010. Advances in Real-Time Rendering in 3D Graphics and Games in Siggraph.