Realistic Water Simulation

Moussab Ibrahim*
mtibrahi@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Lingheng Tony Tao
linghent@andrew.cmu.edu
Entertainment Technology Center
Pittsburgh, Pennsylvania, USA

Weier Flora Xiao
weierx@andrew.cmu.edu
Information Networking Institute
Pittsburgh, Pennsylvania, USA

Figure 1: Realistic Water Simulation

Abstract

Simulating and rendering realistic water is a compelling yet challenging topic in computer graphics, especially in game development, where it plays a crucial role in enhancing immersion and visual appeal. High-quality water simulations can captivate players, creating memorable experiences, while poorly rendered water can detract from realism and player engagement. Achieving realistic water effects requires addressing various aspects, including dynamic wave interactions, foam shading, and the increasing demand for interactivity. This paper explores key techniques and methodologies for water simulation and rendering, highlighting the interplay between physical dynamics and visual fidelity to achieve realistic and interactive water in modern graphics applications.

Keywords

Water, Simulation, Fluid, Rendering, FFT, SPH, PBF

ACM Reference Format:
Moussab Ibrahim, Lingheng Tony Tao, and Weier Flora Xiao. 2024. Realistic Water Simulation. In *Proceedings of Carnegie Mellon University*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Both authors contributed equally to this research.

Carnegie Mellon University, 15673 - Visual Computing Systems

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Rendering and simulating realistic water is not only a popular topic in computer graphics but also a challenging one. Especially in the field of game development, realistic water can be breathtaking and become a highlight in the game, while poorly rendered water can significantly reduce player immersion or even drive players away. Realistic water involves a wide range of knowledge, from the dynamic changes of waves to the shading of foam, as well as today's higher demands for interactivity. Water simulation is an important

and cutting-edge topic in computer graphics. In this paper, we will introduce some of the techniques used in water rendering.

2 Linear Wave Superposition Method

In water rendering, vertex animation of the water surface is one of the key components for achieving realistic effects. The term *wave* commonly refers to the undulating effect on the water surface. In reality, the undulations of the water surface usually exhibit a subtle periodicity—it appears rhythmically up and down. However, it is challenging to discern this periodicity with the naked eye, and it might even seem fairly random. To achieve such undulations, we need a *mechanism*—or an algorithm—that can generate periodic waveforms. These mechanisms, which can generate periodic waveforms controllable through parameters such as *frequency*, *period*, and *amplitude*, among other concepts we will introduce shortly, are called **oscillators**. Below, we will introduce the concept starting with the simplest oscillators.

2.1 Sine Oscillator

The **sine wave** is the simplest and most fundamental waveform we have learned. Nelson L. Max[5] was the first to propose using a sequence of sine wave curves with varying amplitudes to simulate the undulations of the water surface.

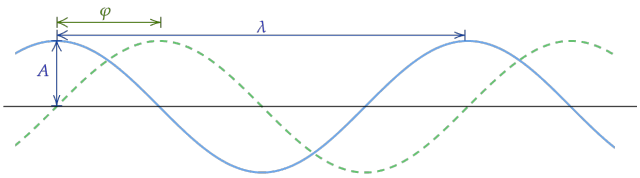

Figure 2: Parameters of a sine wave.

For the horizontal position (x, z) at time t, the formula for a **sine wave oscillator** is given by

$$y(x, z, t) = A \sin(((x, z) \cdot \mathbf{k}) - \omega t + \varphi) \tag{1}$$

where:

- A is called the **amplitude**. It refers to the height difference between the peak (*crest*) of the waveform and the equilibrium position, which is also half the difference between the peak and the lowest point (*trough*).
- λ is called the **wavelength**. It represents the distance between two adjacent wave crests. Using the wavelength, we can compute the **frequency** ω in the formula, which is defined as
$$\omega = \frac{2}{\lambda}. \tag{2}$$
- φ is called the **phase constant**. By adding different phase constants, the entire waveform can be shifted. Multiplying it by time t enables the waveform to progress over time.
- $\mathbf{k}$ is the **direction**, indicating the direction of wave propagation. This vector is also referred to as the **wave vector** or **wave number**.

For a single sine wave oscillator, the resulting waveform is regular and visually appears highly repetitive, as shown in Figure 3.

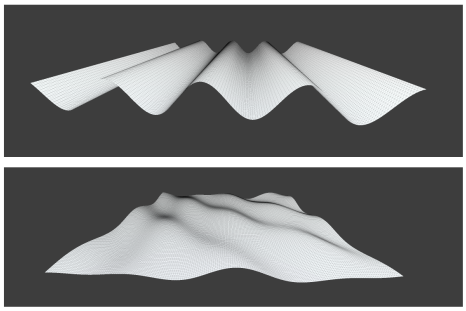

Figure 3: Oscillating wave effects generated by a single sine oscillator (top) and a sum-of-sine oscillator (bottom). It can be observed that a single sine oscillator produces periodic waveforms but appears rather dull and clearly unrealistic compared to water waves. In contrast, the sum-of-sine oscillator produces waveforms that still retain some periodicity but are more realistic and resemble the water surface.

However, as also shown in Figure 3, when we superimpose multiple sine waves with different frequencies, speeds, and amplitudes, the result becomes more interesting. Such an oscillator is called a **sum-of-sine oscillator**. As the name suggests, the sum-of-sine oscillator is the result of adding multiple sine oscillators. For a point $\mathbf{x} = (x, z)$ at time t, its height $y = h(\mathbf{x}, t)$ can be expressed as:

$$h(\mathbf{x}, t) = \sum A_i \sin((\mathbf{k}_i \cdot \mathbf{x}) - \omega_i t + \varphi_i). \tag{3}$$

2.2 Gerstner Oscillator

A characteristic of waveforms generated by sine waves is their smooth crests. This is why waveforms produced even by sum-of-sine oscillators often appear insufficiently realistic—real-world water surfaces, particularly oceans with rough waves, tend to have sharper crests and broader troughs. Next, we introduce the **Gerstner oscillator**. This oscillator generates Gerstner waves, which are characterized by sharp crests and wide troughs.

The mathematical expression for a Gerstner wave, where a point $\mathbf{x} = (x_0, z_0)$ in the xz-plane oscillates in the y direction, is given by:

$$\begin{aligned} x &= x_0 - \sum_{i=1}^{N_w} \hat{\mathbf{k}}_i A_i \sin(\mathbf{k}_i \cdot \mathbf{x}_0 - \omega_i t + \phi_i), \\ y &= \sum_{i=1}^{N_w} A_i \cos(\mathbf{k}_i \cdot \mathbf{x}_0 - \omega_i t + \phi_i), \\ z &= z_0 - \sum_{i=1}^{N_w} \hat{\mathbf{k}}_i A_i \sin(\mathbf{k}_i \cdot \mathbf{x}_0 - \omega_i t + \phi_i). \end{aligned} \tag{4}$$

where N_w represents the total number of waves, $\mathbf{k}_i$ is the wave vector of the ith wave with magnitude $k_i = |\mathbf{k}_i|$, and the unit direction vector is $\hat{\mathbf{k}_i} = \frac{\mathbf{k}_i}{k_i}$.

3 A Simple Discussion on Complex Numbers

Before diving into the statistical modeling methods below, we need to have a basic understanding of complex numbers; otherwise, it will be difficult to comprehend many mathematical foundations.

3.1 Complex Numbers

A **complex number**is an extension of real numbers. A complex number $\mathbf{z}$ is typically written as

$$\mathbf{z} = a + bi \tag{5}$$

where a and b are real numbers, and $i^2 = -1$. Since no real scalar i can satisfy $i^2 = -1$, i is referred to as the **imaginary unit**, describing an *imaginary* number. Here, the real number a is called the **real part** of the complex number $\mathbf{z}$, and b is called the **imaginary part**. The magnitude of the complex number is

$$|\mathbf{z}| = \sqrt{a^2 + b^2}, \tag{6}$$

and the **conjugate** of the complex number $\mathbf{z}$ is a complex number with the same real part but opposite imaginary part, denoted as $\mathbf{z}^*$:

$$\mathbf{z}^* = a - bi. \tag{7}$$

Complex numbers can undergo basic addition, subtraction, and multiplication by computing the real and imaginary parts separately and substituting $i^2 = -1$ when it appears. This provides the computation formulas for complex numbers:

$$\begin{aligned}(a+bi)+(c+di) &= (a+c)+(b+d)i,\\ (a+bi)-(c+di) &= (a-c)+(b-d)i,\\ (a+bi)(c+di) &= ac+adi+bci+bdi^2\\ &= (ac-bd)+(ad+bc)i.\end{aligned} \tag{8}$$

3.2 Matrix Representation of Complex Numbers

We can equivalently represent a complex number $\mathbf{z} = a + bi$ as a matrix:

$$a + bi \equiv \begin{bmatrix} a & -b \\ b & a \end{bmatrix}. \tag{9}$$

It is straightforward to verify that this matrix representation is equivalent for addition, subtraction, and multiplication. For instance,

$$(a+bi)+(c+di) \equiv \begin{bmatrix} a & -b \\ b & a \end{bmatrix} + \begin{bmatrix} c & -d \\ d & c \end{bmatrix} = \begin{bmatrix} a+c & -(b+d) \\ b+d & a+c \end{bmatrix} \equiv (a+c)+(b+d)i. \tag{10}$$

3.3 Rotation Representation with Complex Numbers

If we represent a 2D vector $\mathbf{v} = \begin{bmatrix} x \\ y \end{bmatrix}$ as a complex number $x + yi$, then applying a matrix transformation $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$,

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ y\sin\theta + y\cos\theta \end{bmatrix}, \tag{11}$$

is equivalent to the rotation transformation learned earlier. According to Section 3.2, the rotation matrix is also equivalent to $\cos\theta + i\sin\theta$, meaning we can interpret the complex number $\mathbf{r} = \cos\theta + i\sin\theta$ as a rotation. Here, we observe for the first time a fascinating connection between complex numbers and trigonometric functions.

3.4 Geometric Interpretation of Complex Numbers

A complex number $\mathbf{z} = a + bi$ consists of a real part a and an imaginary part b. It can also represent a point (a, b) in the 2D plane or a vector from the origin to (a, b), where the horizontal axis is called the **real axis**, and its unit is 1, while the vertical axis is called the **imaginary axis**, and its unit is i. This 2D plane is called the **complex plane**.

Note that the magnitude of a vector (a, b) in the complex plane is $\sqrt{a^2 + b^2}$, the same as the magnitude of the corresponding complex number. A point in the complex plane can also be represented in polar coordinates as

$$\mathbf{z} = r(\cos\theta + i\sin\theta), \tag{12}$$

where r is the magnitude of the vector (or complex number) $\mathbf{z}$, and θ is the **argument** of the vector (or complex number) $\mathbf{z}$, defined as

$$\theta = \arg(\mathbf{z}) = \tan^{-1}(b/a). \tag{13}$$

Thus, complex numbers are naturally suited for discussing 2D periodic rotation problems.

3.5 Euler's Formula

Finally, we introduce the most important formula in this chapter—the **Euler's Formula**, which connects trigonometric functions with exponential functions:

$$e^{ix} = \cos x + i\sin x, \tag{14}$$

where e is the base of the natural logarithm, i is the imaginary unit, and the trigonometric functions are in radians. A special case of Euler's Formula when $x = \pi$ is

$$e^{i\pi} + 1 = 0. \tag{15}$$

This formula simultaneously features five of the most significant constants in mathematics: $e, i, \pi, 1, 0$, making it often celebrated by mathematicians as the most beautiful and extraordinary formula.

To understand this formula, we start with the **Taylor Expansion**. In mathematics, for a function $f(x)$ that is infinitely differentiable with real or complex variables, its Taylor series is expressed as:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n, \tag{16}$$

where $n!$ is the factorial of n, and $f^{(n)}(a)$ is the n-th derivative of f at a. By expanding e^x, $\sin x$, and $\cos x$ based on Equation (16), we get:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots,\\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots,\\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots. \end{aligned} \tag{17}$$

Substituting $x = iz$ into e^x, we obtain:

$$e^{iz} = \cos z + i \sin z. \tag{18}$$

Thus, we verify Euler's Formula[1].

Euler's Formula also has a geometric interpretation: in the complex plane, e^{ix} describes a rotation on the unit circle. Here, x is the angle of rotation (in radians), representing the position of a point after rotating counterclockwise by x radians from $(1, 0)$. Using Euler's Formula, we can further express a complex number in the form:

$$\mathbf{z} = re^{i\theta}, \tag{19}$$

where r is the magnitude of the complex number, θ is the argument, and i is the imaginary unit.

If you have more sections or need adjustments, feel free to let me know!

4 Statistical Modeling Methods

The methods introduced in the previous section allow us to generate relatively realistic water surfaces. However, a critical issue arises when we raise the camera to a certain height—the repetitive texture patterns become easily noticeable. This occurs because no matter what type of wave superposition we use, the resulting function is ultimately periodic, leading to observable repetitions at certain scales. While this issue might be less noticeable from a grazing angle, it becomes evident when viewed from a top-down perspective.

We can mitigate this visual repetitiveness by carefully adjusting parameters or overlaying enough sine waves with different frequencies, wave numbers, and amplitudes. However, this requires tools to help generate such data. Meanwhile, research from other disciplines—fluid mechanics and oceanography—shows that even Gerstner waves struggle to accurately describe real water surfaces. Instead, ocean waves are often highly random [6]. If we denote the wave height at position $\mathbf{x}$ and time t as $h(\mathbf{x}, t)$, then h should be treated as a random variable. To compute wave heights h that follow realistic probability distributions as closely as possible, we need to understand the most critical mathematical tool in this process—the Fourier Transform.

4.1 Fourier Transform

The **Fourier Transform** is a mathematical tool that converts signals from the time domain (or spatial domain) into the frequency domain. Intuitively, it decomposes a complex waveform into a series of simple sine and cosine waves, providing information about their frequencies, amplitudes, and phases through a certain transformation.

For water rendering, the goal is to simulate realistic wave patterns. The main challenges in simulating realistic water waves are as follows:

(1) *They exhibit a certain periodicity.* Since water waves follow a periodic undulating pattern, we can assume they consist of waves of different wavelengths, whose propagation, interference, and attenuation exhibit periodic characteristics.
(2) *They also exhibit significant randomness.* As mentioned earlier, oceanographic studies have shown that wave undulations are typically random. However, these undulations exhibit statistical regularities, meaning wave heights and wavelengths can be described using statistical methods.
(3) *They have global effects.* The propagation of water waves creates continuous effects across the entire water surface.

The Fourier Transform allows us to decompose a waveform that appears to have no discernible computational pattern into a combination of simple sine waves. This enables us to reconstruct the water surface using this set of waveforms.

4.2 Discrete Fourier Transform (DFT)

While the ideal wave height is a continuous signal, it is impossible to handle every point on the water surface continuously during rendering. In practice, model files cannot represent a surface with an infinite number of vertices. Therefore, we can consider the wave height as a discrete signal. If our water surface is represented by a 512×512 grid, we can treat the height of each vertex as a discrete two-dimensional signal.

Definition (Discrete Fourier Transform) For a discrete signal f with N sampled points, the **Discrete Fourier Transform** (DFT) of f is:

$$f = (f[0], f[1], \ldots, f[N-1]), \tag{20}$$

producing an N-dimensional array

$$\mathbf{F} = (\mathbf{F}[0], \mathbf{F}[1], \ldots, \mathbf{F}[N-1]), \tag{21}$$

where

$$\mathbf{F}[k] = \sum_{n=0}^{N-1} f[n] e^{\frac{-2\pi ikn}{N}} = \sum_{n=0}^{N-1} f[n] \exp\left(\frac{-2\pi ikn}{N}\right). \tag{22}$$

Using a brute-force method to compute the DFT of a discrete signal f with N sampled points requires N complex multiplications and $N-1$ complex additions for each $\mathbf{F}[k]$. This results in an $O(N^2)$ algorithm, which is computationally expensive. The **Fast Fourier Transform** (FFT) is an algorithm designed to compute the DFT efficiently, reducing the complexity to $O(N \log N)$. Since N is typically a large number, this reduction in Big-O complexity has significant practical performance implications.

4.3 Four-Point DFT

The core idea of FFT is to exploit the periodicity and symmetry properties of Equation (22). To better understand FFT, we start with a simple example of a DFT with $N = 4$.

Using Euler's formula,

$$e^{ix} = \cos x + i \sin x,$$

we have

$$e^{-i\pi/2} = -i. \tag{23}$$

[1] Although widely introduced, this method relies on Taylor series and differentiation in the complex domain, which themselves often require Euler's Formula, leading to a circular argument. This verification is intended for understanding and testing purposes rather than a rigorous proof.

Substituting into Equation (22), the four-point DFT formula simplifies to:

$$\mathbf{F}[k] = \sum_{n=0}^{3} (-i)^{kn} f[n], \tag{24}$$

which, when expanded, becomes:

$$\mathbf{F}[k] = f[0] + (-i)^k f[1] + (-1)^k f[2] + i^k f[3]. \tag{25}$$

Thus, $\mathbf{F}^\top$ can be written as:

$$\mathbf{F}^\top = \begin{bmatrix} f[0]+f[1]+f[2]+f[3] \\ f[0]-if[1]-f[2]+if[3] \\ f[0]-f[1]+f[2]-f[3] \\ f[0]+if[1]-f[2]-if[3] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} f^\top \tag{26}$$

Here comes an optimization step. Recalling the properties of matrix multiplication, observe:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} f[0] \\ f[1] \\ f[2] \\ f[3] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} f[0] \\ f[2] \\ f[1] \\ f[3] \end{bmatrix}. \tag{27}$$

Additionally, note:

$$\begin{bmatrix} f[0]+f[2]+f[1]+f[3] \\ f[0]-f[2]-if[1]+if[3] \\ f[0]+f[2]-f[1]-f[3] \\ f[0]-f[2]+if[1]-if[3] \end{bmatrix} = \begin{bmatrix} (f[0]+f[2])+(f[1]+f[3]) \\ (f[0]-f[2])-i(f[1]+f[3]) \\ (f[0]+f[2])-(f[1]+f[3]) \\ (f[0]-f[2])+i(f[1]-f[3]) \end{bmatrix}. \tag{28}$$

Thus, we can precompute $f[0]+f[2]$, $f[0]-f[2]$, $f[1]+f[3]$, and $f[1]-f[3]$ to accelerate computation. Note that for four-point DFT, we can divide it into even and odd terms: even terms ($f[0]$, $f[2]$) and odd terms ($f[1]$, $f[3]$). This division facilitates recursive divide-and-conquer processing, breaking a DFT problem into two subproblems (even and odd terms). Specifically, the four-point DFT can be expressed as:

$$\mathbf{F}[k] = \sum_{n=0}^{3} f[n] \cdot e^{-i\frac{2\pi}{4}kn} = \sum_{n=0,2} f[n] \cdot e^{-i\frac{2\pi}{4}kn} + \sum_{n=1,3} f[n] \cdot e^{-i\frac{2\pi}{4}kn}. \tag{29}$$

These even and odd terms represent two independent subproblems.

4.4 Four-Point FFT

For two complex numbers **a** and **b** and a given complex number α, we compute two output complex numbers **A** and **B** using the following formula:

$$\begin{aligned} \mathbf{A} &= \mathbf{a} + \alpha\mathbf{b}, \\ \mathbf{B} &= \mathbf{a} - \alpha\mathbf{b}. \end{aligned} \tag{30}$$

Next, let's illustrate this with a diagram. In Equation (30), the output **A** depends on both input **a** and input **b**, as shown in Figure 4.

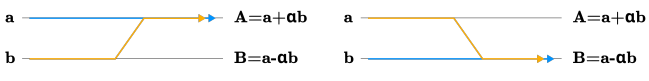

Figure 4: Connecting inputs to outputs. Note that regardless of how a, b, A, or B are positioned, each output must connect to both inputs.

When two operations are depicted together in one diagram, an intersecting pattern inevitably emerges, as shown in Figure 5. This operational pattern is referred to as the **butterfly operation**.

Figure 5: Diagram of butterfly operations for complex multiplication and addition. This diagram clearly indicates the source of data for each output.

In the four-point DFT, we can quickly draw the butterfly operation diagram, as shown in Figure 6.

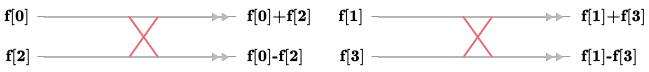

Figure 6: Butterfly operation pattern in four-point DFT. Here, $\alpha = 1$.

Based on the optimization strategy proposed in the previous section, it becomes clear that the butterfly operation is the smallest execution unit of the divide-and-conquer approach in FFT. The butterfly operation represents the concrete implementation of dividing and merging.

Additionally, note that the outputs of a butterfly operation can be reused in subsequent butterfly operations, ultimately yielding the desired DFT result.

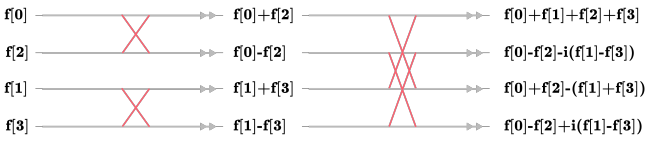

Figure 7: Computation diagram for a four-point FFT. Note that the diagram contains four butterfly operations. Each butterfly operation involves two complex calculations, resulting in a total of eight calculations, compared to the 16 calculations required previously.

4.5 Cooley-Tukey FFT Algorithm

Building on the idea of the four-point FFT, we observe that efficiently solving the DFT requires leveraging its symmetry properties. In the general case of N points, the simplifications made in Section 4.3, such as $e^{-i\pi/2} = -i$, no longer hold significant value. Therefore, we introduce a new **twiddle factor**,

$$W_N = e^{-i\frac{2\pi}{N}}. \tag{31}$$

Noting that

$$e^{i\frac{2\pi k}{N}} = e^{i\frac{2\pi(k+mN)}{N}}, \tag{32}$$

we derive an important property:

$$W_N^k = W_N^{k+mN}, \forall m \in \mathbb{Z}. \tag{33}$$

Here, the exponent k takes the form:

$$k = n \cdot \frac{N}{2^{\text{stage}}}, \tag{34}$$

where n is the vertical index. Additionally, observe that there are a total of $\log_2 N$ stages, and at each stage, the range covered by butterfly operations doubles, reaching at most $N/2$. Therefore, the complexity of computation at each stage is $O(N)$, and the total complexity is bounded by $O(N \log N)$.

The core idea of the divide-and-conquer approach in FFT is to decompose the N-point DFT into:

(1) *Subproblem of even points.* $E[k] = \sum_{m=0}^{N/2-1} f[2m] \cdot W_{N/2}^{km}$.
(2) *Subproblem of odd points.* $O[k] = \sum_{m=0}^{N/2-1} f[2m+1] \cdot W_{N/2}^{km}$.

Finally, the even and odd parts are combined to obtain the result of the original DFT:

$$\begin{aligned} F[k] &= E[k] + W_N^k \cdot O[k], \\ F[k+N/2] &= E[k] - W_N^k \cdot O[k]. \end{aligned} \tag{35}$$

4.6 Statistical Oceanographic Model

According to oceanographic research, the wave height $h(\mathbf{x}, t)$ at a horizontal position $\mathbf{x} = (x, z)$ and time t can indeed be described as a sum of sine waves [6], as shown previously:

$$h(\mathbf{x}, t) = \sum A_i \sin((\mathbf{k}_i \cdot \mathbf{x}) - \omega_i t + \varphi_i).$$

Since sine and cosine functions can be converted into each other via a phase shift, the wave height can also be described as a sum of cosine waves:

$$h(\mathbf{x}, t) = \sum A_k \cos((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k). \tag{36}$$

Notably, the term $\cos((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k)$ on the right-hand side is the real part of $e^{i((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k)}$, since:

$$e^{i((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k)} = \cos((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k) + i \sin((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k). \tag{37}$$

Thus, Equation (36) can be rewritten as:

$$h(\mathbf{x}, t) = \text{Re}\left[\sum A_k e^{i((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k)}\right], \tag{38}$$

where Re represents taking the real part of a complex number. Simplifying the expression:

$$\begin{aligned} h(\mathbf{x}, t) &= \text{Re}\left[\sum A_k e^{i((\mathbf{k}_k \cdot \mathbf{x}) - \omega_k t + \phi_k)}\right] \\ &= \text{Re}\left[\sum A_i e^{i\mathbf{k}_k \cdot \mathbf{x}} e^{-i\omega_k t} e^{\phi_k}\right] \\ &= \text{Re}\left[\sum (A_i e^{-i\omega_k t} e^{\phi_k}) e^{i\mathbf{k}_k \cdot \mathbf{x}}\right]. \end{aligned} \tag{39}$$

Since wave height is a physical quantity, the imaginary part has no physical meaning. Therefore, in subsequent descriptions, we omit the Re notation[2]. The term $(A_k e^{-i\omega_k t} e^{\phi_k})$ is a function of wave amplitude A_k (associated with wave $\mathbf{k}$) and time t. We can use an envelope function $\tilde{h}(\mathbf{k}, t)$ to represent this part. This allows us to simplify the wave height function into its complex form:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t) e^{i\mathbf{k} \cdot \mathbf{x}}. \tag{40}$$

Here, $\mathbf{k} = (k_x, k_z)$ is called the **wave vector**, a vector in the 2D xz plane pointing in the direction of wave propagation. Specifically:

$$\begin{aligned} k_x &= 2\pi \frac{n}{L}, \\ k_z &= 2\pi \frac{m}{L}. \end{aligned} \tag{41}$$

where n and m satisfy:

$$-\frac{N}{2} \le n, m \le \frac{N}{2}. \tag{42}$$

L represents the horizontal domain size considered in ocean wave simulation, effectively the side length of the wave grid area, determining the actual physical size of the 2D simulation grid. N is the number of discrete grid points, typically required to be an even number. For example, $N = 256$ means the area is simulated as an $N \times N$ grid. A larger N increases simulation accuracy but also computational cost. A larger L allows for simulating larger-scale waves.

The wave vector has physical significance:

(1) *Magnitude.* The magnitude of the wave vector $|\mathbf{k}| = \sqrt{k_x^2 + k_z^2} = \frac{2\pi}{\lambda}$, where λ is the wavelength. Thus, the magnitude of the wave vector represents the spatial frequency of the wave, which is the reciprocal of the wavelength.
(2) *Direction.* As discussed earlier, the wave vector indicates the direction in which the wave propagates.

In Equation (40), the final component, $\tilde{h}(\mathbf{k}, t)$, is referred to as the **amplitude Fourier component** of the wave height field. It is generated by a combination of a spatial spectrum and a Gaussian random function, and it is the key factor influencing the time-varying height of the water surface. Specifically, $h(\mathbf{x}, t)$ is a physical space representation (i.e., in the *time domain* or *spatial domain*) of the wave height field, describing the height of the water surface at a specific location and time. On the other hand, $\tilde{h}(\mathbf{k}, t)$ is the Fourier space representation of the wave height field, describing the complex amplitude corresponding to a specific wave vector $\mathbf{k}$ and time t.

$\tilde{h}(\mathbf{k}, t)$ is typically a complex number. According to Equation (19), its complex form is:

$$\tilde{h}(\mathbf{k}, t) = |\tilde{h}(\mathbf{k}, t)| \cdot e^{i\phi(\mathbf{k}, t)}, \tag{43}$$

which contains important physical information:

(1) *Amplitude.* The modulus of the complex number $|\tilde{h}(\mathbf{k}, t)|$ represents the strength of the wave.
(2) *Phase.* The argument of the complex number $\phi(\mathbf{k}, t)$ indicates the relative position of the wave in space.

Clearly, the complex amplitude $|\tilde{h}(\mathbf{k}, t)|$ is a function of time t, reflecting the dynamic process of wave evolution.

[2] In physics and mathematical analysis, omitting Re is a common convention. When no ambiguity is introduced, only the real part is considered to have physical significance, so it is assumed by default.

According to oceanographic studies, $|\tilde{h}(\mathbf{k}, t)|$ can be estimated using the **Phillips Spectrum**, which takes the form:

$$P_n(\mathbf{k}) = \left\langle |\tilde{h}(\mathbf{k}, t)|^2 \right\rangle = A \frac{e^{-\frac{1}{(kL)^2}}}{k^4} |\hat{\mathbf{k}} \cdot \hat{\mathbf{w}}|^2, \tag{44}$$

where

$$L = \frac{V^2}{g}. \tag{45}$$

Here, V is the wind speed, $\hat{\mathbf{w}}$ is the unit vector indicating the wind direction, and g is the gravitational acceleration constant. The final term in Equation (44), $|\hat{\mathbf{k}} \cdot \hat{\mathbf{w}}|^2$, removes waves that are completely perpendicular to the wind direction; this term equals 0 when the wave propagation direction is perpendicular to the wind. It is a function of the wave's direction and magnitude.

Although this model is relatively simple, when the wave number $k = |\mathbf{k}|$ becomes large (i.e., when the wavelength becomes very small), the mathematical behavior of the Phillips Spectrum causes poor numerical convergence [6]. Specifically, this poor convergence leads to unstable or unrealistic behavior in the simulation for high wave number components (i.e., short-wavelength waves). To address this convergence issue, for short waves where $l \ll L$, J. Tenssendorf proposed in [6] introducing an additional Gaussian factor into Equation (44):

$$\exp(-k^2 l^2). \tag{46}$$

4.7 Temporal Evolution of Wave Height Field

In Equation (44), although we estimate a time-dependent function $\tilde{h}(\mathbf{k}, t)$, the formula itself does not explicitly account for time. To introduce temporal evolution, we generate an initial spectrum using the Phillips spectrum and apply additional operations to simulate time variance. First, we compute an initial height field in Fourier space using two random samples from a Gaussian distribution:

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P_n(\mathbf{k})}. \tag{47}$$

Here, ξ_i and ξ_r are two independent random variables sampled from a Gaussian distribution with mean 0 and standard deviation 1. For each wave vector $\mathbf{k}$, we now have an initial amplitude $\tilde{h}_0(\mathbf{k})$ that follows the $P_n(\mathbf{k})$ distribution.

Next, we incorporate temporal variation through the **dispersion relation** of water waves, expressed as:

$$\omega^2(k) = gk, \tag{48}$$

where $\omega(k)$ is the angular frequency of the wave, indicating its temporal frequency; g is the gravitational acceleration constant; and k is the magnitude of the wave vector. Dispersion refers to the phenomenon where wave components with different wave vectors propagate at different speeds. Incorporating the dispersion relation, the final amplitude as a function of wave vector $\mathbf{k}$ and time t is:

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k}) e^{i\omega(k)t} + \tilde{h}_0^*(-\mathbf{k}) e^{-i\omega(k)t}. \tag{49}$$

Here, $\tilde{h}_0(\mathbf{k}) e^{i\omega(k)t}$ represents the forward-propagating wave component, and $\tilde{h}_0^*(-\mathbf{k}) e^{-i\omega(k)t}$ represents the backward-propagating wave component.

4.8 Inverse Fourier Transform for Generating Wave Height Field

Since Fast Fourier Transform (FFT) operates on discrete signals, the wave height field must be rewritten as a discrete function. To achieve this, we first impose the following restriction:

$$0 < k, l < N - 1. \tag{50}$$

Here, k and l are the discrete indices of the wave vector in the x and z directions in 2D Fourier space. Using these indices, the wave vector can be defined as:

$$\mathbf{k} = \left(\frac{2\pi k - \pi N}{L}, \frac{2\pi l - \pi N}{L} \right). \tag{51}$$

This redefinition introduces a shift of $-\frac{\pi N}{L}$ to center the range of the wave vector $\mathbf{k}$ symmetrically around the origin in the frequency domain.

$\tilde{h}(\mathbf{k}, t)$ remains the frequency domain representation of the height field $h(\mathbf{x}, t)$, so we need to transform it back to the time domain using the **Inverse Fast Fourier Transform** (IFFT). Without delving into the derivation of IFFT due to space constraints, the formula for the height at a grid point (n, m) at time t is:

$$\begin{aligned} h(n, m, t) = \\ \frac{1}{N^2} (-1)^n \sum_{k=0}^{N-1} \left[(-1)^m \sum_{l=0}^{N-1} \tilde{h}(k, l, t) \exp\left(i \frac{2\pi m l}{N} \right) \right] \\ \cdot \exp\left(i \frac{2\pi n k}{N} \right). \end{aligned} \tag{52}$$

Here, N is the resolution of the grid.

4.9 Steps of the Inverse Fourier Transform Algorithm

The IFFT consists of five main steps:

(1) *Generate the initial spectrum.* Create two spectral textures containing $\tilde{h}_0(\mathbf{k})$ and $\tilde{h}_0^*(-\mathbf{k})$.
(2) *Generate the Fourier amplitude map.* Use the dispersion relation to generate a map containing $\tilde{h}(\mathbf{k}, t)$.
(3) *Perform N horizontal 1D FFTs.* Use the Fourier amplitude map from the previous step as input.
(4) *Perform N vertical 1D FFTs.* Each row of the Fourier amplitude map from the previous step serves as input.
(5) *Multiply the amplitude by* $(-1)^m$ *and* $(-1)^n$*, then scale by* $\frac{1}{N^2}$.

5 Position-Based Fluids (PBF) Algorithm

Position-Based Fluids (PBF) is a particle-based method designed for real-time fluid simulations. It achieves stability and efficiency by enforcing constraints on particle positions, bypassing the instability issues present in force-based approaches like Smoothed Particle Hydrodynamics (SPH). In this section, we describe the algorithm's steps and associated mathematical formulations.

5.1 Predict Position

The predicted position of each particle is computed using the particle's velocity and external forces, such as gravity. The velocity update is given by:

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \cdot \frac{\mathbf{f}_{\text{ext}}}{m_i}, \tag{53}$$

where $\mathbf{v}_i$ is the velocity, Δt is the time step, $\mathbf{f}_{\text{ext}}$ is the external force, and m_i is the particle mass.

The predicted position $\mathbf{p}_i$ is then:

$$\mathbf{p}_i = \mathbf{x}_i + \Delta t \cdot \mathbf{v}_i, \tag{54}$$

where $\mathbf{x}_i$ is the current position.

This predicts where the particle will move under the influence of its current velocity and applied forces. It is the first step in simulating fluid dynamics over time.

5.2 Density Constraint

The density ρ_i for particle i is calculated using its neighbors within a radius h, applying a smoothing kernel W:

$$\rho_i = \sum_j m_j \cdot W(\mathbf{p}_i - \mathbf{p}_j, h). \tag{55}$$

One commonly used kernel is the poly6 kernel:

$$W(r,h) = \frac{315}{64\pi h^9}(h^2 - r^2)^3, \quad \text{for } r < h. \tag{56}$$

The density constraint is defined as below, where ρ_0 is the rest density:

$$C_i = \frac{\rho_i}{\rho_0} - 1, \tag{57}$$

Then enforces incompressibility by ensuring the density of each particle stays close to the target rest density ρ_0.

5.3 Position Correction

To satisfy the density constraint, a position correction $\Delta\mathbf{p}_i$ is applied iteratively. The correction is computed using a constraint solver:

$$\Delta\mathbf{p}_i = \frac{1}{\rho_0} \sum_j \lambda_j \cdot \nabla_{\mathbf{p}_i} W(\mathbf{p}_i - \mathbf{p}_j, h), \tag{58}$$

where λ_j is a Lagrange multiplier determined for each particle.

This correction ensures that particles maintain proper spacing, enforcing the density constraint iteratively until convergence.

5.4 Velocity Update and Integration

Once the position corrections are applied during constraint solving, the velocities of particles are updated to reflect the positional adjustments. The updated velocity for particle i is calculated as:

$$\mathbf{v}_i = \frac{\mathbf{p}_i - \mathbf{x}_i}{\Delta t}, \tag{59}$$

where $\mathbf{p}_i$ is the corrected position, $\mathbf{x}_i$ is the previous position and Δt is the simulation time step.

After updating the velocity, the final position of each particle is integrated for the next simulation step:

$$\mathbf{x}_i = \mathbf{p}_i. \tag{60}$$

These ensure that the particle states are consistent between time steps. The velocity update incorporates positional corrections into the particle's dynamic behavior, while the integration step progresses the simulation forward in time.

6 Neighbor Search Optimization

The neighbor search algorithm is a critical component of PBF, as it identifies particles within the interaction radius h of each particle. A naive implementation requires that every particle compares with all other particles, resulting in a computational complexity of $O(n^2)$, which is prohibitive for simulations involving large particle counts. To overcome this limitation, an optimized neighbor search algorithm is implemented, dividing the simulation space into grid cells and reducing the number of potential comparisons.

6.1 Grid-Based Partitioning

Grid Division: The simulation space is partitioned into a uniform grid where each cubic cell has a side length equal to h, the interaction radius. This ensures that all potential neighbors of a particle are contained within its own cell or the 26 adjacent cells. Particles are assigned to cells based on their spatial coordinates, calculated as:

$$\text{cell}_{x,y,z} = \left\lfloor \frac{\mathbf{x}_i}{h} \right\rfloor, \tag{61}$$

where $\mathbf{x}_i$ is the position of particle i, and h is the cell size.

Localized Search: To find neighbors for a given particle, only the particles in its grid cell and the 26 neighboring cells are checked. This reduces the number of comparisons from $O(n^2)$ to approximately $O(n)$ for large n, as the number of particles per cell is bounded by the fluid density and interaction radius.

Dynamic Updates: Since particles move between cells during the simulation, the grid is updated at each time step. This ensures that particles are correctly reassigned to their respective cells, maintaining the integrity of the neighbor search structure.

6.2 Neighbor Data Structure

For each particle, a neighbor structure is maintained to facilitate efficient access during constraint solving:

Number of Neighbors: Tracks the count of particles within the interaction radius h.

Neighbor IDs: Stores the IDs of neighboring particles for direct access during the simulation steps.

This structure not only speeds up the constraint-solving phase but also minimizes memory overhead by avoiding redundant computations.

6.3 Benefits of Grid-Based Neighbor Search

The grid-based neighbor search provides several advantages: **Scalability:** By limiting comparisons to a small subset of particles, the method scales effectively with increasing particle counts. **Efficiency:** The reduction in computational complexity makes it feasible to simulate large fluid systems in real time.

Flexibility: The grid structure can accommodate dynamic particle distributions, such as those found in non-uniform or turbulent flows.

This optimization ensures that neighbor determination remains computationally efficient and scalable, allowing real-time simulation of large particle systems without compromising accuracy.

6.4 Advantages of GPU Acceleration

Leveraging Compute Shaders in Unity enables the efficient simulation of Position-Based Fluids by utilizing the parallel processing capabilities of modern GPUs. This approach provides several key advantages:

(1) **Parallelism:** Each particle's computation, such as position prediction, constraint solving, and neighbor search, is executed concurrently across thousands of GPU threads. This significantly reduces the computational time compared to CPU-based implementations.
(2) **Scalability:** The grid-based neighbor search ensures that the simulation can handle large particle systems efficiently, as the computational cost grows linearly with the number of particles.
(3) **Real-Time Performance:** GPU acceleration enables interactive fluid simulations, making it ideal for applications in games, virtual reality, and interactive graphics, where high frame rates are critical.
(4) **Dynamic Behavior:** The high computational throughput of GPUs allows for the simulation of complex fluid phenomena, such as turbulence, splashing, and mixing, in real time.

GPU acceleration transforms PBF simulations from computationally intensive tasks into practical solutions for real-time applications. By leveraging parallel processing and optimized algorithms, the simulation achieves a balance between physical accuracy and computational efficiency.

7 Fluid Rendering Based On PBF

After the simulation is done using position-based particles, we can continue with the fluid rendering. The technique we are using is Screen-Space Fluid Rendering. This technique is used to simulate and render fluid effects in real-time on a two-dimensional screen space. Its main idea is to generate only the closest surface to the camera, as shown in the diagram in Figure 8. Figure 9 is an overview of the fluid rendering technique in the screen space. It includes a few main steps including generating the depth of particles, smoothing the depth, calculating the normals, calculating the thickness of particles, and shading the surface.

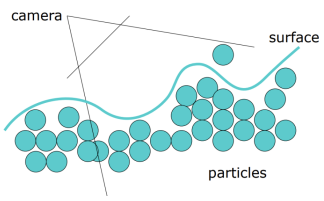

Figure 8: Screen Space Fluid Rendering

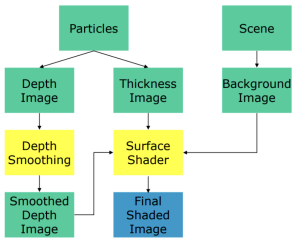

Figure 9: Overview of Screen Space Fluid Rendering

Rendering Particle Spheres. First, we need to create a quad for each particle in the geometry shader. Then in fragment shader, discard the pixels outside the circle and calculate the normal, diffuse and specular. Finally, render it on the screen using Graphics.DrawProcedureNow. The Graphics. The DrawProcedureNow function is used to draw geometry on the screen using GPU-generated vertices. The result is shown in Figure 10.

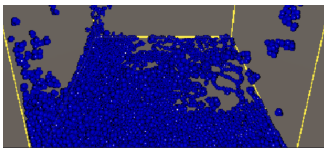

Figure 10: Particle shading effect

Generating the depth. We need to generate a depth buffer that stores the distance from the camera to each pixel in the scene. In the depth pass, calculate the depth and encode the depth into EncodeFloatRGBA to ensure proper interpretation and compatibility in order to use it as a color output or pass it to other stages of the rendering pipeline. Returning the depth directly without encoding will be fine if we only need the depth value within the shader itself or for internal calculations. The result is as shown in Figure 11.

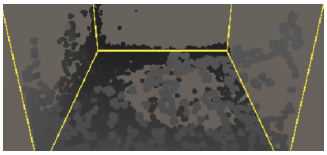

Figure 11: Unblurred depth effect

Smoothing the depth. Smoothing or blurring is a necessary step in fluid rendering. Applying a smoothing or blurring filter such as a Gaussian smoothing or bilateral filter to the intermediate buffers (for example, depth and thickness) can help to reduce noise or pixelation. After getting the depth texture, we need to blur the depth using the bilateral filter. The result is as shown in Figure 12.

Calculating the normal. After having the blurred depth, the normals of the fluid surface can be generated. In the fragment shader, calculate the eye space position from UV coordinates and depth. Then, calculate differences that represent the change in eye-space

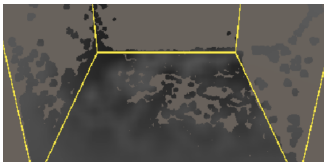

Figure 12: Blurred depth effect

position along the x and y axes. Next, calculate the cross products and return the normal. The normals will determine the orientation of the fluid surface and are essential for lighting and shading calculations. The result is as shown in Figure 13.

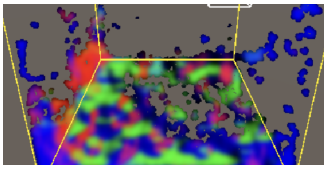

Figure 13: Normal effect

Calculating the thickness. Fluids are often transparent. Screen-space surface rendering only generates surface nearest to the camera, making it looks strange with transparency because we could not see surfaces behind front. Hence, to solve this problem we can shade fluid as semi-opaque using thickness through volume to attenuate color. To generate the thick- ness, we should render particles using additive blending and without depth test. To do this, use geometry shader to generate the quads for each particle again, then calculate the thickness and apply bilateral blurring on the result. The result is as shown in Figure 14.

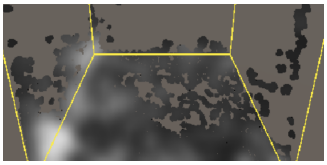

Figure 14: Thickness effect

Shading the surface. Finally, shade the surface with different effects in 'surfaceShading.shader'.

Adding Basic Characteristics. Calculating the diffuse, ambient and specular in fragment shader.

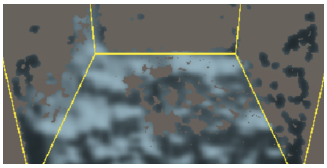

Figure 15: Diffuse Effect

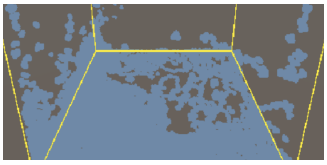

Figure 16: Ambient Effect

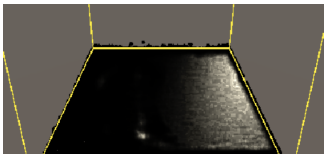

Figure 17: Specular Effect

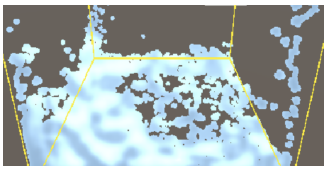

Figure 18: Diffuse + Ambient + Specular Effect

Calculating Beer's Lambert Law. Light decays exponentially with distance. Hence, we can use Beer Lambert's law to calculate light absorption and calculate the transparency as Figure 19. The formula of Beer's Lambert law is I = exp(-kd), where d = distance. The result is as shown in Figure 20.

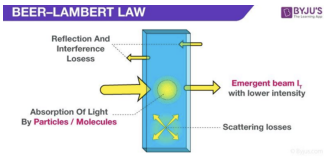

Figure 19: Beer's Lambert Law

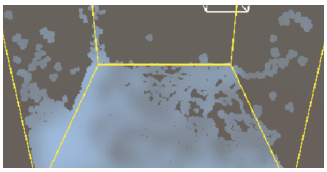

Figure 20: Beer's Lambert Law effect

Calculating Fresnel. The Fresnel effect is a phenomenon in physics that describes how light reflects and refracts when it encounters a surface. The simplified mathematical approximation of Fresnel's equations is known as the Schlick approximation. It provides an efficient and reasonably accurate way to compute the Fresnel reflection. The result is as shown in Figure 21.

Figure 21: Fresnel effect

Final Result. The final shading result with all the effects mentioned before is as shown in Figure 22.

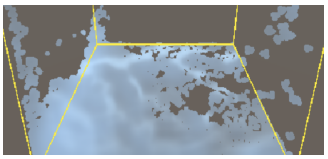

Figure 22: Final Fluid Effect

8 Smoothed Particle Hydrodynamics

8.1 Navier-Stokes equations

Smoothed particle hydrodynamics is a popular method for fluid simulation. It relies on the Navier-Stokes to simulate the velocity evolution across the fluid. The intuition behind it is quite simple, SPH approximate a fluid as group of particles that interact with each other. These particles have their own intrinsic mass, position, velocity and forces applied to them. Navier-Stokes equation tells us that the evolution of the velocity in the fluid is given by three elements.

$$\rho(\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v}) = -\nabla p + \rho\nu\nabla^2\mathbf{v} + \rho\mathbf{f},$$

where:

- $\mathbf{v}$ is the velocity field,
- t is time,
- ρ is the density,
- p is the pressure,
- ν is the kinematic viscosity,
- $\mathbf{f}$ is the body force per unit volume.

The left hand side of the equation is derived from

$$\frac{D\mathbf{v}}{Dt} = \frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v},$$

, so the acceleration of a particle depends only on the pressure gradient, the Laplacian of the speed (viscosity) and the external forces. We are making two strong assumptions about the simulated fluid which are constant viscosity factor and incompressibility. The first one is directly guaranteed with this version of Navier-Stokes equation since the viscosity term is a field and not a tensor. We need to verify the mass conservation equation

$$\nabla \cdot \mathbf{v} = 0$$

. This equation is directly verified because the moving particles carry their mass around.

8.2 Kernels

The principle characteristic of SPH is to use kernels to compute quantities for a particle. Indeed interactions between particles can be written as [3]

$$f(\mathbf{r}) \approx \sum_j m_j \frac{f_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h),$$

where:

- m_j is the mass of particle j,
- ρ_j is the density at particle j,
- f_j is the value of the scalar field at particle j,
- $W(\mathbf{r} - \mathbf{r}_j, h)$ is the smoothing kernel with support radius h.

Smoothing kernels are functions that say how much a particle influences another one with respect to their distance. We use finite support function that integrate to 1. It means that the function is 0 for all distances exceeding h.

In our case we have the following quantities to compute:
1. **Pressure Force[3]:**

$$\mathbf{F}_{\text{pressure}} = -\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\right) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h),$$

where:

- P_i and P_j are the pressures at particles i and j,
- ρ_i and ρ_j are the densities,
- m_j is the mass of particle j,
- ∇W is the gradient of the smoothing kernel.

2. **Viscosity Force[3]:**

$$\mathbf{F}_{\text{viscosity}} = \sum_j m_j \frac{\mu}{\rho_j} \frac{\mathbf{v}_j - \mathbf{v}_i}{\epsilon^2} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h),$$

where:

- μ is the dynamic viscosity coefficient,
- $\mathbf{v}_i$ and $\mathbf{v}_j$ are the velocities of particles i and j,
- ϵ is a small value to avoid division by zero.

3. **Density Computation[3]:**

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h),$$

where:

- m_j is the mass of particle j,
- $W(\mathbf{r}_i - \mathbf{r}_j, h)$ is the smoothing kernel.

From density we can easily get pressure which is simply: **Equation of State**

$$P = k(\rho - \rho_0),$$

where:

- k is the stiffness constant,
- ρ is the current density,
- ρ_0 is the reference (rest) density.

We found different kind of kernels in the literature and we decided to use these three following kernels:

1. **Density Kernel[3]:**

$$W_{\text{density}}(\mathbf{r}) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - |\mathbf{r}|^2)^3, & \text{if } |\mathbf{r}| \leq h, \\ 0, & \text{otherwise,} \end{cases}$$

where:

- $|\mathbf{r}|$ is the magnitude of the vector $\mathbf{r}$,
- h is the influence radius.

2. **Pressure Kernel Gradient[3]:**

$$\nabla W_{\text{pressure}}(\mathbf{r}) = \begin{cases} -\frac{45}{\pi h^6}(|\mathbf{r}| - h)^2 \frac{\mathbf{r}}{|\mathbf{r}|+10^{-5}}, & \text{if } |\mathbf{r}| \leq h, \\ 0, & \text{otherwise.} \end{cases}$$

3. **Viscosity Kernel Laplacian[3]:**

$$\nabla^2 W_{\text{viscosity}}(|\mathbf{r}|) = \begin{cases} \frac{45}{\pi h^6}(h - |\mathbf{r}|), & \text{if } |\mathbf{r}| \leq h, \\ 0, & \text{otherwise.} \end{cases}$$

8.3 Implementation

Having defined the theoretical basis of SPH, we now describe its computational implementation. We can quickly see that this simulation can benefit from parallelization. Indeed every particles performs only read operations to other particles' data when it computes its density, pressure etc. Thus we can separate each of these computation in functions that execute in parallel for particles. This can be done in Unity with the use of compute shaders.

The main bottleneck in this simulation is similar to that in Position-Based Fluids (PBF). Indeed we also perform O(n^2) particles look up operations every time step. This can greatly reduce the number of particles we may be able to simulate. The best method to address this issue is as PBF to split the space into a grid mapping every particles to its cell. Then look ups will only be performed with particles within the same cell. This method take advantage of the particles distribution in the space. Indeed it is very unlikely that all particles are within a single cell. Thus if we have k cells with a uniform distribution we can reach O($\frac{n^2}{k}$). However too many cells would reduce realism. This method is in fact very similar to sort-middle tiled approach we used to build a parallel renderer without lock.

8.4 Results

We had some good results with SPH reaching around 10'000 particles without the grid partition. We could reach that many particles because we used indirect instancing in Unity allowing each particle's data to be initialized and computed exclusively on the GPU. We could also avoid the use of a mesh and simply represent the particle as a point and a texture but the mesh approach opens the way to more advanced rendering method such as Raymarching/Raytracing. On our side we implemented the optimization however Unity kept crashing. Moreover as a sorting operation is required every update we need to transfer all data to CPU every cycle since sorting directly on GPU would lead to severe race conditions.

Finally, the hyper parameters are crucial to obtain a good simulation. The time step has to be carefully chosen because a small time step slow down the simulation and a large one make it unstable. Additionally, the frame rate i.e. the number of update every second is determined by GPU capabilities so these hyper parameter wouldn't produce the same result on different machines.

9 Eulerian Grid

Opposed to Smoothed particle hydrodynamics, this method simulate the space as vector field rather than discrete particles. The space is divided into a grid of cells in which each cell has its own quantities such as velocity or density. The evolution of the cell's values is determined by the Navier-Stokes equations.While the grid itself does not represent a moving fluid, introducing density or velocity into specific cells initiates a propagation of these values through the grid. This behavior can simulate phenomena such as smoke spreading or dye diffusing in water, emulating fluid motion. There are many different way to represent this fluid movement such as pressure/density color gradient or arrows for the velocities field.

9.1 Navier-Stokes

In order to have a simulation we only need to repeat three steps every time step. Compute advection, enforce incompressibility and add external forces [4].

Indeed the left hand side of the Navier-Stokes equation only contains the velocity evolution over time and the advection term that we have to compute ourselves since the grid is fixed the velocity is not moving with the particle as in SPH (this term cause a viscosity effect since we interpolate value in cells making small value to vanish)[4]. Additionally, we need to enforce incompressibility to satisfy the mass conservation equation, as a cell does not carry any mass. Finally, we include external forces, which correspond to the first term of the Navier-Stokes equation, representing the evolution of velocity in a cell over time. We present an implementation for a 2D simulation, However, it is fairly easy to transition to a 3D simulation in Unity.

9.2 Implementation

1. **Avdvection[4]:** Advection is calculated using the semi-Lagrangian method, which tracks the movement of fluid quantities (e.g., velocity or density) by tracing them backward in time. This method is suitable in our case because it provides a great stability even with large time step.

(1) Compute the source position for each grid cell:

$$\mathbf{x}_{\text{source}} = \begin{bmatrix} x_{\text{current}} \\ y_{\text{current}} \end{bmatrix} - \Delta t \begin{bmatrix} u(x_{\text{current}}, y_{\text{current}}) \\ v(x_{\text{current}}, y_{\text{current}}) \end{bmatrix},$$

where Δt is the time step, u and v are the horizontal and vertical components of the velocity field, and $\mathbf{x}_{\text{source}}$ is the position from which the quantity is advected.

(2) Perform bilinear interpolation to obtain the quantity q at $\mathbf{x}_{\text{source}}$:

$$q_{\text{new}}(x_{\text{current}}, y_{\text{current}}) = \sum_{i=0}^{1} \sum_{j=0}^{1} w_{ij}\, q(x_i, y_j),$$

where (x_i, y_j) are the grid points surrounding $\mathbf{x}_{\text{source}}$, and w_{ij} are the bilinear interpolation weights.

(3) Update the grid value:

$$q(x_{\text{current}}, y_{\text{current}}) \leftarrow q_{\text{new}}(x_{\text{current}}, y_{\text{current}}).$$

2. **Incompressibility[4]:**

To maintain incompressibility, we solve the pressure projection step using the Gauss-Seidel method. Although this iterative solver is quite slow we decided to stick with it because of its simplicity. We could parallelize the algorithm with a compute shader with minimal overhead. The pressure projection step ensures that the velocity field $\mathbf{v}$ is divergence-free ($\nabla \cdot \mathbf{v} = 0$) by solving the Poisson equation for pressure p:

$$\nabla^2 p = \nabla \cdot \mathbf{v},$$

where $\nabla^2 p$ is the Laplacian of the pressure field, and $\nabla \cdot \mathbf{v}$ is the divergence of the velocity field(In our case we could directly use the known velocities).

(1) Calculate Divergence: For each cell (i, j), compute the divergence based on the velocity field:

$$\text{divergence}(i, j) = v(i, j) - v(i, j+1) + u(i, j) - u(i+1, j),$$

where $u(i, j)$ and $v(i, j)$ are the horizontal and vertical velocity components, respectively.

(2) Distribute Divergence: Distribute the divergence correction among neighboring cells:

$$\text{divergence} \leftarrow \text{divergence} \times \text{overRelaxation},$$

$$v(i, j) \leftarrow v(i, j) - \frac{\text{divergence}}{\text{tot}},$$

$$v(i, j+1) \leftarrow v(i, j+1) + \frac{\text{divergence}}{\text{tot}},$$

$$u(i, j) \leftarrow u(i, j) - \frac{\text{divergence}}{\text{tot}},$$

$$u(i+1, j) \leftarrow u(i+1, j) + \frac{\text{divergence}}{\text{tot}},$$

where tot is the number of valid neighbors (accounting for boundary cells).

(3) Iterate for Convergence: Repeat the above steps for a fixed number of iterations or until the residual divergence is sufficiently small

$$\text{Divergence} =< \epsilon.$$

(4) Boundary Conditions: Adjust velocity values at boundaries $(u_{\text{boundary}}, v_{\text{boundary}})$.

9.3 Results

We got good result with this simulation that is more stable than SPH as it avoids the numerical instabilities caused by very close particles. Initially, we implemented the simulation as a script, as debugging is easier compared to using shaders. This approach saved us significant time. We learned this lesson during the SPH implementation, where we spent several days debugging in shaders, only to rewrite everything back as a script and start over. We could simulate around 2000 cells with the script only and the compute shader increased the performance dramatically to reach more than 10'000 cells.

Opposed to SPH the parallelization of this simulation is not trivial since cells are not independent entities compared to particles. We had to find some techniques to avoid race condition in the shader. To parallelize the Gauss-Seidel solver, we partitioned the grid into independent groups of cells, allowing updates to be performed concurrently within each group without race conditions in a compute shader. Since cells share between 2 to 4 edges, we had to use a chessboard partition that splits the computation into two disjoint groups of cells. So only half of the grid is processed in parallel.

Advection does not directly modify the velocity buffer. Since each computation must be independent of the previous one and cells share edges, we use the original array for computations and write the results to an output array. This approach simplifies parallelization, as there is not any concurrent read and write operation. However, further optimization is possible because all edges, except those on the border, are computed twice. To address this, we perform the computation for only one of the two neighboring cells, except for border cells where all edges are processed. Finally the external forces are simply added to the process without any modification.

10 State of the art

Smoothed particle hydrodynamics (SPH) and the grid-based approach are both commonly used in fluid simulations today. SPH, with its particle-based nature, excels at capturing small details such as droplets and splashes, producing highly realistic results. In the other hand, the grid-based approach offers a more robust and computationally efficient solution, particularly for simulating large fluid zones. While the computational cost of SPH increases with area to maintain realism, the grid-based method can be easily extended to simulate larger fluid volumes without significant performance degradation. We also tried to explore two advanced methods we came across during our research: the material point method (MPM)[1] and convolutional neural networks (CNN)[2], but unfortunately we ran out of time to be able to implement them successfully.

References

[1] A. Imam Kistidjantoro Afwarman Manaf Dody Dharma, Cliff Jonathan. 2017. Material point method based fluid simulation on GPU using compute shader. (2017). https://ieeexplore.ieee.org/abstract/document/8090962

[2] Pablo Sprechmann Ken Perlin Jonathan Tompson, Kristofer Schlachter. 2017. Accelerating Eulerian Fluid Simulation With Convolutional Network. (2017). https://proceedings.mlr.press/v70/tompson17a

[3] David Charypar Matthias Müller. 2003. Particle-Based Fluid Simulation for Interactive Applications. *SIGGRAPH Symposium on Computer Animation* (2003). https://matthias-research.github.io/pages/publications/sca03.pdf

[4] Nuttapong Chentanez Matthias Müller. 2011. Real-Time Eulerian Water Simulation Using a Restricted Tall Cell Grid. *ACM SIGGRAPH 2011* (07 2011). https://dl.acm.org/doi/pdf/10.1145/1964921.1964977

[5] Nelson L. Max. 1981. Vectorized procedural models for natural terrain: Waves and islands in the sunset. *SIGGRAPH Comput. Graph.* 15, 3 (aug 1981), 317–324. https://doi.org/10.1145/965161.806820

[6] Jerry Tessendorf. 2001. Simulating Ocean Water. *SIG-GRAPH'99 Course Note* (01 2001).