Real-time Water Rendering Techniques

Francisco Macedo Ferreira
pg55942@alunos.uminho.pt

***Abstract*—Rendering realistic water surfaces in real-time graphics is a challenging task due to the complex nature of water's interaction with light and its inherently dynamic properties. Pure particle-based simulations are computationally prohibitive in real-time scenarios, compelling developers to employ clever approximations. This paper explores various rendering techniques, including Screen Space Reflections (SSR), planar reflections, cubemaps, and ray tracing, each providing a trade-off between realism and performance. Additionally, it examines wave simulation approaches, such as sum-of-sines, Gerstner waves, and FFT-based spectral methods, as well as specialized techniques like flow maps for rivers and a brief mention of buoyancy calculations for interactive realism.**

***Index Terms*—Water Rendering, Screen Space Reflections, Planar Reflections, Environment Mapping, Ray Tracing, Wave Simulation, Flow Maps, Buoyancy**

I. Introduction

Real-time simulation and rendering of water remain some of the most computationally intensive challenges in computer graphics due to water's complex and dynamic physical properties. Achieving a realistic portrayal traditionally involves particle-based simulations, yet this method is impractical for real-time applications due to computational costs. Consequently, developers must rely on creative approximations and strategic optimizations to simulate water convincingly within strict performance constraints. This paper reviews various sophisticated approaches to water rendering, balancing visual fidelity and computational efficiency, crucial for interactive applications such as video games.

Fig. 1: The Last of Us Part II main menu [1]

II. Water Reflection and Refraction

Rendering water involves simulating the interaction of light with the water surface. Water surfaces act like natural mirrors, reflecting the environment around them, and they also allow to see beneath the surface with distortion (refraction), especially at glancing angles (Fresnel effect). Achieving believable reflections in real-time graphics is a challenging task, as it requires simulating or faking the interaction of light with the water surface, and the environment around it.

Fig. 2: Screenshot of a river in Red Dead Redemption 2 [2]

From this point forward, it is assumed that access to the normals of the water surface is available, allowing their use to simulate the reflection and refraction of light. The method for computing these normals will be covered in Section III.

A. Screen Space Reflections (SSR)

Screen Space Reflection is a widely used real-time technique to approximate reflective surfaces, thus it is a good candidate for water rendering. SSR works by reusing information already on screen: after the scene is rendered, a post-processing pass traces rays in the screen-space depth buffer, trying to find a reflected hit for each pixel of a reflective surface. Basically, it marches a ray from the view, bouncing off the water surface, and intersects against the depth texture to find what on-screen geometry it would reflect.

Implementing SSR is a relatively simple task, as seen in Sugu Lee's blog post [3]. The SSR pass can be executed using a compute shader. Below is the pseudo code[1].

[1]You can find a Metal shader implementation in the Sugu Lee's blog post [3]

Algorithm 1: SSR Compute Shader Pseudocode

inputs:

color_buffer: The color of each pixel.

depth_buffer: The depth (in clip space) of each pixel.

reflection_mask: A mask indicating which pixels are reflective (=1) or not (=0).

normal_buffer: The normal of each pixel.

camera: The camera view size, view transform matrix, projection matrix and inverse projection matrix.

outputs:

out_color: The final color of each pixel.

algorithm:

```
Fetch normal from the normal buffer and the reflective mask from the reflection mask buffer.
if the reflection mask is not 0 then
    Compute the position, the reflection vector and the max distance for the current sample in texture space.
    if the reflection vector is moving away from the camera then
        Find the intersection between the reflection ray and the scene geometry by tracing the ray.
        if the intersection is found then
            Compute the reflection color by fetching the color from the color buffer at the intersection point.
        else
            Set the reflection to the background color.
        end
    end
end
Add the reflection color to the current sample to create the final color.
```

SSR became popular in games because, as shown in the pseudo code, it integrates well with deferred rendering (using the existing depth and normal buffers) and is easy to implement without requiring significant changes to the overall pipeline. The effect is also quite fast, as it is a post-processing effect, so its cost is independent of the number of objects in the scene and instead depends on the number of pixels that are reflective.

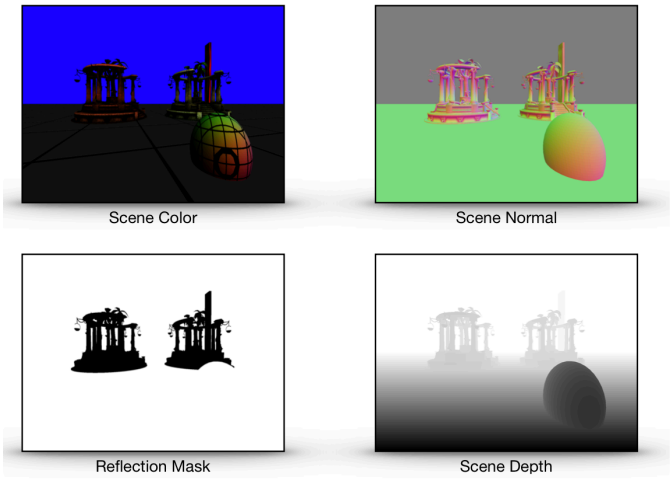


Fig. 3: SSR Buffers used in the compute shader (Sugu Lee's blog post [3])

However, SSR comes with notable **limitations** due to its screen-space nature. It can only reflect the objects that are visible on the screen. This leads to the common artifact of "missing" reflections near the edges of the screen or holes where an occluding object in front causes the reflected object to be absent in the depth buffer.

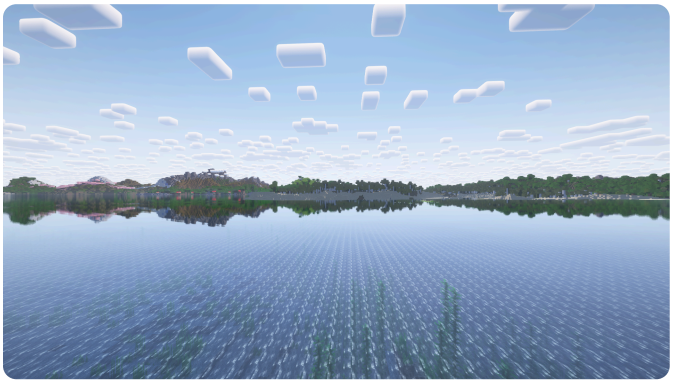


Fig. 4: SSR implementation in Minecraft Shader Mod Rethinking Voxels [4]

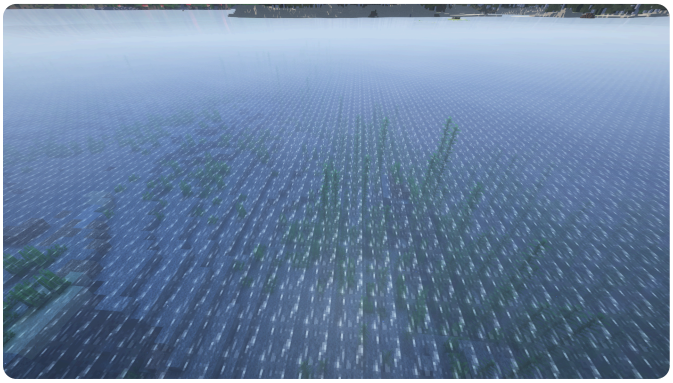


Fig. 5: Missing terrain reflections from the example above

Developers mitigate this by faking blur at the screen edges, fading out SSR, or blending with other reflection methods. Additionally, objects behind transparent objects (which won't appear in the depth buffer) may not be

reflected properly. This can be worked around by rendering the transparent objects in a separate pass, after the SSR pass.

Despite these issues, SSR is still a good candidate for water rendering, it remains one of the best compromises between **visual fidelity** and **performance** for planar-ish reflectors like water. Almost every game uses it for water reflections in combination with cubemaps to cover its weaknesses.

B. *Planar Reflections*

Although not used as much as SSR in water, being used more in mirrors, planar reflections are still a good choice for water rendering. This technique involves rendering a second view of the scene from the perspective of a "mirror" positioned below the water plane.

Fig. 6: Planar reflection in a mirror in Grand Theft Auto V [5]

Essentially, one renders the scene upside-down (flipped across the water surface) to a texture, and then during water rendering, samples that texture to get reflection color. Planar reflections produce **very high fidelity** results, since it's an actual render of the scene, all off-screen and occluded objects above the water can appear correctly in the reflection. This technique is especially straightforward for perfectly horizontal water planes (in oceans or lakes) because the mirror camera can be an exact reflection of the main camera's orientation, making the reflection very precise.

Fig. 7: Planar reflection[2] in the water level in Half-Life 2 [6]

The **downside** is performance cost. Planar reflections render the world twice, one for the main view and another for the mirrored view. If a game has a huge open world, doubling the draw calls and triangle processing for reflections is expensive. Some optimizations are possible, for example, rendering the reflection at a lower resolution or using stricter culling rules. Scalability is also a problem, if there are multiple water patches at different orientations, each would need a different world render.

A hybrid approach seen in some titles is to use planar reflections only for important things like the sky and distant scenery, while still using SSR for fine details. For instance, Unreal Engine's water can use a "scene capture" [7] for the sky reflection and combine it with SSR for local reflections, achieving a good balance. Planar reflections yield the best image quality short of true ray tracing, but developers must budget for their significant cost.

C. *Environment Maps and Probes (Cubemaps)*

Another lightweight approach is to use environment maps, also called cubemaps, to approximate the reflection of the environment. Environment maps resisted the test of time and are still a common technique in games.

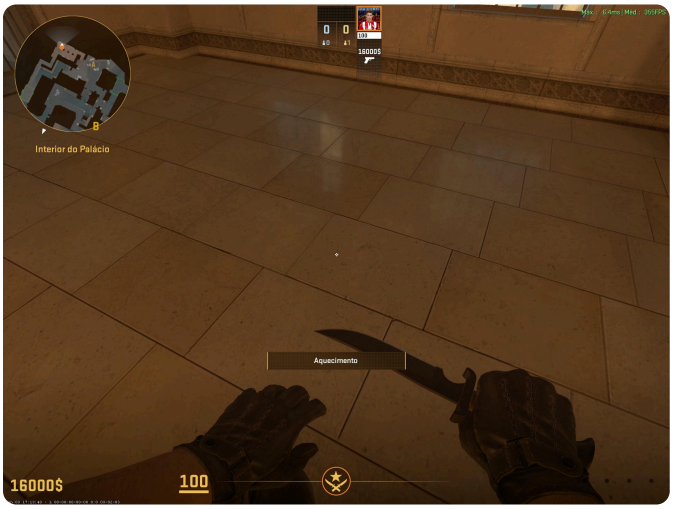


Fig. 8: Cubemap reflecting a window in Counter-Strike 2 [8]

For water, environment maps are often used to reflect the sky and distant environment. Many engines include reflection probes that capture the scene from certain points; these can be used for water as well, although for large water surfaces a single static cubemap for the sky is common. The visual realism of environment maps is limited compared to SSR or planar methods as they only capture the world from one point (often not the player's exact position) and cannot reflect dynamic objects unless the cubemap is frequently updated (which would also impact performance).

Fig. 9: Cubemap in the scope of the AWP weapon in Counter-Strike 2 [8]

[2]https://developer.valvesoftware.com/wiki/Water_%28shader%29

This means, for example, characters in Counter-Strike 2 cannot be reflected through the AWP scope (as seen in Fig. 9), and the same will happen with water. Either way, a blurry reflection via a cubemap still adds a lot of realism to the game at virtually no cost. The **performance** of environment maps is also very good, as they are a single texture it is just a texture lookup. The main **drawback** is the lack of accuracy and dynamism.

D. Ray Tracing

The advent of hardware-accelerated ray tracing (via APIs like DXR and hardware like NVIDIA's RTX or AMD's RDNA) has significantly advanced the realism of real-time water rendering. Unlike screen-space or planar techniques, ray tracing simulates the physical behavior of light, enabling reflections that are pixel-perfect, perspective-correct, and capable of including off-screen and dynamic elements.

For each pixel on a water surface, a ray-traced reflection computes the direction a reflected light ray would travel based on the surface normal—which itself may be animated or displaced by waves, ripples, or foam. The renderer traces this ray into the scene, testing for intersections with geometry or skydomes. If the ray hits an object, that object's shading result is used as the reflected color; if it misses, the renderer samples the environment map or skybox. This process is repeated per-pixel, producing highly accurate, temporally stable reflections that respect scene geometry and lighting.

Because it directly simulates light transport, ray tracing is conceptually clean and highly flexible. In addition to reflections, the same technique can be extended to handle refraction, caustics, and even light scattering effects, all within the same unified framework. The visual fidelity is generally unmatched: ray tracing eliminates the view-dependent artifacts of screen-space reflections and the geometric constraints of planar methods. Dynamic characters, off-screen geometry, and complex surfaces are all faithfully reflected, even across curved or undulating water.

Fig. 10: Ray tracing in Cyberpunk 2077 [9] from Digital Foundry's video [10]

The main limitation is performance. Tracing rays through a complex scene is computationally expensive, even with dedicated hardware acceleration. Real-time implementations typically reduce ray count, cap bounce depth, or trace at lower resolutions and then apply temporal and spatial denoising to produce a clean image. Many engines also incorporate upscaling technologies like DLSS or FSR to further amortize the cost.

Despite these optimizations, ray-traced reflections remain a premium feature, generally reserved for high-end hardware or enabled as an optional setting for quality presets. Developers must carefully budget for the added GPU cost and decide when to fall back to cheaper methods like SSR.

E. Refraction and Lighting (Transparency, Fresnel, Caustics)

In addition to reflections, water rendering must account for **refraction**, the bending of light that allows to see into the water.

Real water is typically transparent, but it is view-dependent: at shallow angles you mainly see reflection, while looking straight down you see through the underwater terrain. This behavior is governed by the **Fresnel effect**, which engines simulate by blending reflection vs. refraction based on the angle, often using Schlick's approximation. According to the Schlick's model, the specular reflection coefficient R can be approximated by the following equation:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{1}$$

where

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{2}$$

where θ is half the angle between the incoming and outgoing light direction, n_1 being the refractive index of the first medium (air, which can be approximated to 1) and n_2 the refractive index of the second medium (water), and R_0 is the reflection coefficient for light incoming parallel to the normal (when $\theta = 0$) [11].

So, in practice, the water shader will take the reflection color (from SSR/planar/etc reflections) and blend it with the refraction color (usually by sampling the scene underwater).

The refracted scene color can be obtained by rendering the world from the camera with only the underwater parts (or by a cheap method: copy the color buffer and offset it by the water normal to mimic bending). For instance, one common approach is: render the scene without water to get the "background" image, then when drawing the water, sample that background texture with UVs perturbed by the water surface normal (scaled by the water depth) [12]. This produces a distortion of the underwater view, approximating true refraction. If done well, you can see the lakebed or objects beneath the surface, distorted by ripples.

Fig. 11: Water refraction in Photon Minecraft Shaders [13]

The water also attenuates light, so shaders often fade the refracted color to a deep color (blue/green) with depth, to simulate that light absorption.

Another important light effect is **specular highlights**, the small bright spots that appear on the surface of the water. This can be implemented with a simple approach using the Blinn-Phong model, where the specular highlight is computed as:

$$S = \max\left(0, \vec{N} \cdot \vec{H}\right)^p \tag{3}$$

where $\vec{N}$ is the surface normal, $\vec{H}$ is the half vector between the view direction and the light direction, and p is the shininess exponent. This value is then multiplied with the light color to get the final specular color. Artists can modify the shininess exponent to control the size and shape of the specular highlight [14].

A more advanced method can use a BRDF (physically-based material) to calculate specular highlights, allowing for more physically accurate water surfaces [15].

Caustics also deserve a special mention. These are the focused light patterns on surfaces caused by the water's surface curvature. Caustics can greatly enhance the realism of water rendering. Simulating real caustics is very costly, requiring tracing a vast number of light rays through the water surface. Instead, games use clever approximations and tricks to simulate them.

A common approach is to simply project an animated texture onto the underwater surfaces that resembles caustic patterns [16]. Although not physically accurate, this can create surprisingly believable results as seen in Fig. 12.

Fig. 12: Underwater caustics in Red Dead Redemption 2 [2] using a projected texture

However, using projected textures for caustics often results in visible repeating patterns, which can break immersion if not handled carefully.

With modern ray-tracing hardware, caustics can be generated in real time: each frame, the water's animated surface is rendered from the light's perspective into a "caustics map", rays are traced through it to collect hit data on underwater surfaces, and those contributions accumulate over frames (using temporal blending) into a dynamic buffer that is sampled during the final render, producing accurate, evolving caustic illumination [17].

Fig. 13: Caustics from NVIDIA's raytracing technique [17]

III. Waves

With the water shader techniques covered, the discussion can now move on to water wave simulation techniques.

A central task in water rendering is simulating the **waves** that define the surface shape. Water waves can be modeled in various ways depending on the desired realism and performance.

A. Sum of Sines Waves

Realistic water waves are complex, but a simple model that approximates them is to use a sum of sines. This approach leverages the fact that many individual waves (a simple oscillatory function) can be added together to form the overall shape of the water surface.

The simplest approach to create waves is using sine or cosine functions. In a basic implementation, these functions can approximate a water surface, because the water height at any point oscillates up and down following a sinusoidal pattern. For example, a single wave can be described through time t by a height function:

$$W_i(x, z, t) = A_i \sin(D_i \cdot (x, z) \times \omega_i + t \times \varphi_i) \tag{4}$$

having as parameters:

- Wavelength (L): the distance between two consecutive peaks or troughs, with $\omega = \frac{2}{L}$;
- Amplitude (A): the maximum height of the wave;
- Speed (S): the distance the wave travels in one second, with $\varphi = S \times \frac{2}{L}$;
- Direction (D): the direction the wave travels, with $D = (D_x, D_z)$ and $A \cdot B$ being the dot product between A and B.

All the waves can then be summed to get the final height function:

$$H(x, z, t) = \sum_{i=1}^{N} W_i(x, z, t) \tag{5}$$

These parameters can be adjusted by the artist to create the desired wave shape. They can also be interpolated over time to create a more dynamic water surface.

This approach done in the GPU (in the vertex shader, for example) is extremely fast (just sine calculations per vertex) and was common in older games or simple water shaders. Even today, some games still use this approach for small bodies of water or sometimes as a fallback to lower quality settings.

Another consideration is computing the surface *normals* for the lighting. A naive approach is to sample two points on the water surface and compute the normal by the cross product. However, with access to the height function, the normal can be computed by using the partial derivatives with respect to the x and z axes in the height function and performing a cross product with them.

As each point in the water surface can be calculated as:

$$P(x, z, t) = (x, H(x, z, t), z) \tag{6}$$

For the x-axis (the binormal), consider:

$$\begin{aligned} B(x, z, t) &= \left(\frac{\partial x}{\partial x}, \frac{\partial}{\partial x} H(x, z, t), \frac{\partial z}{\partial x}\right) \\ &= \left(1, \frac{\partial}{\partial x} H(x, z, t), 0\right) \end{aligned} \tag{7}$$

Similarly, for the z-axis (the tangent):

$$\begin{aligned} T(x, z, t) &= \left(\frac{\partial x}{\partial z}, \frac{\partial}{\partial z} H(x, z, t), \frac{\partial z}{\partial z}\right) \\ &= \left(0, \frac{\partial}{\partial z} H(x, z, t), 1\right) \end{aligned} \tag{8}$$

The normal is then computed by the cross product of the binormal and the tangent:

$$N(x, z, t) = B(x, z, t) \times T(x, z, t) \tag{9}$$

This equation expansion can be found in the NVIDIA GPU Gems – Chapter 1 [18]. The normal is then normalized to have a length of 1 and used in the lighting calculations.

One problem that can arise from the formula above is that, as more sines are summed to increase detail, the wave starts to look more like a "spiky" surface instead of a wave.

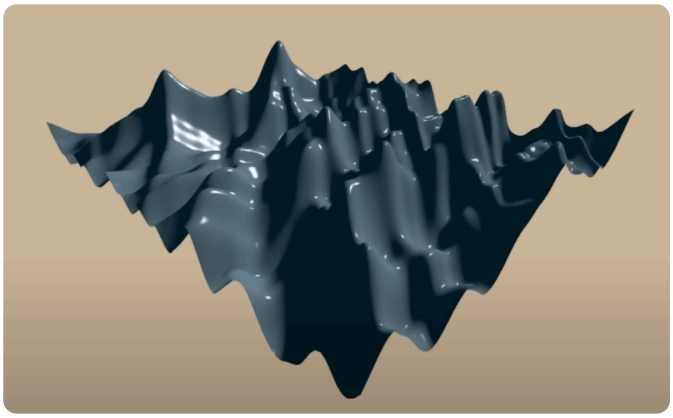


Fig. 14: The result of summing of a lot of sine waves from "How Games Fake Water" video [19]

To address this issue, the author [19] suggests using Fractal Brownian Motion techniques [20], as the formula effectively creates a type of white noise by summing sines with random parameters.

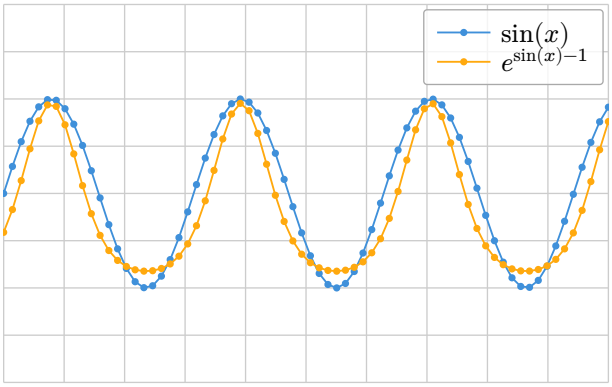
The process begins with amplitude and frequency values set to 1 and a random direction. With each iteration, the amplitude is multiplied by a value less than 1, and the frequency by a value greater than 1, using these updated values to compute the next wave. This approach causes the amplitude to decrease exponentially toward zero, allowing the summation to stop once the amplitude becomes negligible.

The method starts with a large wave and progressively adds smaller, higher-frequency waves with reduced amplitude. The higher frequencies contribute fine detail to the water surface, but their lower amplitude ensures they have less influence on the overall shape, thus avoiding the "spiky" appearance.

An additional enhancement involves applying domain warping [20] to the formula. During the summation of waves, the sampling position can be shifted by the derivative of the previous wave, creating the effect of waves interacting and pushing against each other.

$$H(x, z, t) = \sum_{i=1}^{N} W_i\left(x + \frac{\partial}{\partial x} W_{i-1}, z + \frac{\partial}{\partial z} W_{i-1}, t\right) \tag{10}$$

An additional refinement involves replacing $\sin(x)$ with $e^{\sin(x)-1}$ in the wave function. This transformation produces waves with sharper, more pronounced crests and broader, flatter troughs, resulting in a more visually striking and natural appearance. Adjusting the sharpness and width of these features is straightforward: simply scale a constant in the exponent to control the effect.



The calculation of the surface normal is slightly modified when using this function, but it remains feasible. For a detailed solution of the updated normal computation, refer to the video [19] or the implementation [21].

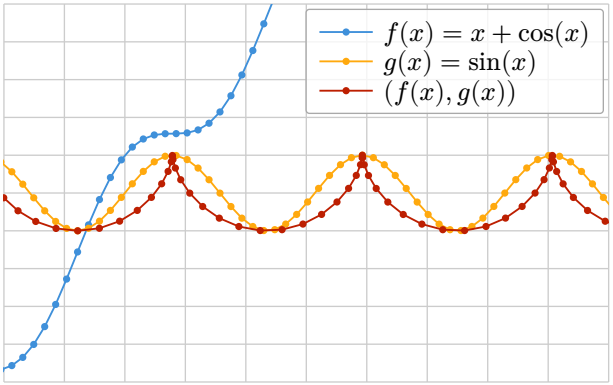
The result can be found in a Shadertoy view [21] which produces a really convincing water surface.

Fig. 15: The result of the sum of sines water shader toy [21]

B. Gerstner Waves

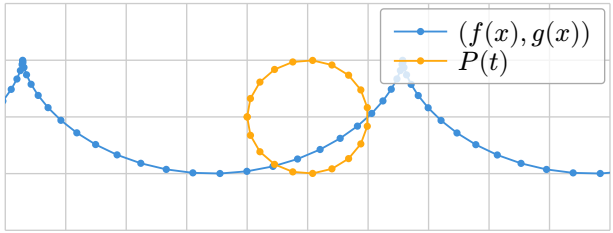
While the sum of sines approach provides a simple and efficient way to approximate water surfaces, it only allows for vertical (y-axis) displacement of the vertices. This limitation means that the resulting waves move points up and down, but do not capture the characteristic forward and backward (horizontal) motion of real water waves. As a result, the surface can look artificial, especially when viewed at glancing angles.

Gerstner waves, first described by Franz Josef Gerstner in 1802, offer a more physically accurate model for water waves by introducing horizontal displacement in addition to the vertical movement. In the Gerstner wave model, each point on the water surface is displaced not only in the y-direction (height), but also in the x and z directions, following the same underlying sine function. This creates the effect of water particles moving in circular or trochoidal paths, which closely resembles the motion of real ocean waves. [22]



If a point x_0 is fixed and a time component is added, the motion of a water particle under a Gerstner wave can be visualized as:

$$P(t) = (x_0 + \cos(x_0 + t), \sin(x_0 + t)) \tag{11}$$



This illustrates that as time progresses, the point x_0 moves along a circular path in the xy-plane, as described by $P(t)$. This circular motion is characteristic of Gerstner waves: it causes the wave crests to become sharper and the troughs to appear more rounded, closely resembling the shape of real ocean waves.

Just as multiple sine waves can be summed, Gerstner waves can also be generalized to three dimensions by combining all their components to compute the final vertex position, as described in [18]:

$$P = \begin{pmatrix} x + \sum\left(Q_i A_i D_{i_x} \cos(\omega_i D_i \cdot (x, z) + \varphi_i t)\right) \\ \sum(A_i \sin(\omega_i D_i \cdot (x, z) + \varphi_i t)) \\ z + \sum\left(Q_i A_i D_{i_z} \cos(\omega_i D_i \cdot (x, z) + \varphi_i t)\right) \end{pmatrix} \tag{12}$$

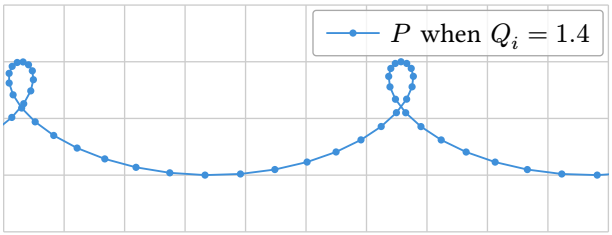
Here, Q_i controls the steepness of each wave, while the other terms are as defined in (4).

Calculating the surface for Gerstner waves is more complex than for simple sine waves, but the resulting expression is still differentiable. Computing the surface normal is crucial for realistic lighting and shading, as it determines how light interacts with the water. The process for finding the normal is similar to that used for a sum of sine waves, but now using the Gerstner formulation. Due to the length of the full formula, it is not reproduced here; however, a derivation and final result can be found in [18].

Gerstner wave displacement creates ocean surfaces with waves that move in a clear direction, conveying the sense of wind or water current. Unlike the sum of sine waves, which can appear as random noise, Gerstner waves produce more natural patterns [23].

Although Gerstner waves can produce more realistic wave shapes, they are less commonly used than the sum of sine waves. This is mainly because the sum of sines is simpler to implement and control (and using other functions can make the waves crests more pronounced). Additionally, FFT-based wave techniques offer even higher visual quality, making them a popular choice in modern applications.

Another challenge with Gerstner waves is that their parameters must be carefully tuned to avoid unnatural artifacts. Specifically, if the sum $Q_i \times \omega_i \times A_i$ exceeds 1, the y component of the surface normal can become negative at the wave crests [18]. This causes the wave to fold over itself, resulting in visually broken or looping artifacts.



Therefore, artists must be especially careful to not generate parameters that will cause the wave to break.

C. Spectral FFT-Based Waves

One limitation of the previous wave techniques is that they inherently produce repeating patterns. When the simulated water surface is large enough, these periodic patterns become noticeable and can break the illusion of realism.

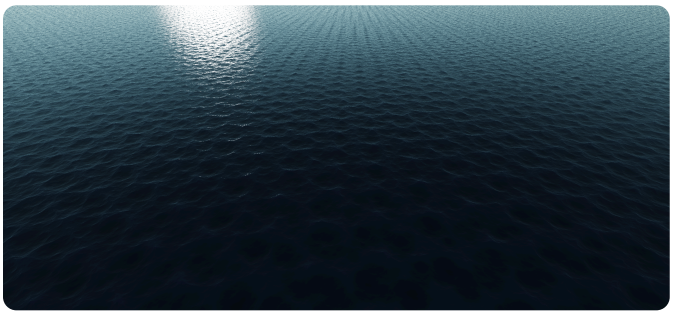


Fig. 16: The repeating pattern of a sum of sines wave in [21] when viewed from afar

There are a few ways to mitigate this: you can reduce the visible area of water (for example, by making the water body smaller or using fog to obscure distant regions), or you can try to hide the repetition by adding more sine waves, increasing the complexity and detail of the surface so that the periodicity is less apparent. Of course, doing the latter would be very expensive.

For expansive ocean surfaces, a common approach is to use spectral wave synthesis via the **Fast Fourier Transform (FFT)**, specifically, its inverse transform. This approach was popularized in Jerry Tessendorf's paper "Simulating Ocean Water" [24], and it has since been adopted in numerous films and games. The core idea is to treat the ocean surface as a superposition of thousands of waves with different frequencies, amplitudes, and directions, according to an oceanographic wave spectrum. Instead of manually summing many sine waves, an inverse FFT allows to get the result of summing them without having to compute each wave, which makes the process much faster.

The technique involves generating a wave spectrum using a statistical model and then computing the inverse FFT of the wave spectrum to get the height of the waves for each point in the water surface.

The wave spectrum describes how the amplitude and phase of waves are distributed across all possible directions and frequencies. In other words, it tells how much energy is present in waves of different sizes and directions. There are several physical models for generating a wave spectrum.

This paper will not delve into the specifics of generating the wave spectrum, the various models available for its creation or computing the wave normals. However, there are many excellent resources that cover these topics in depth, such as [12], [24], [25] and [26].

The wave spectrum can be stored in a texture, allowing multiple clients to generate an identical wave surface—this approach is used effectively in games like Sea of Thieves [27]. Once the wave spectrum is available, the inverse FFT can be used to obtain the height of the waves for each point on the water surface. The result is a heightmap of the water surface, capturing a wide range of wave sizes: from small ripples to large swells, all in one unified approach.

Modern GPUs are capable of computing the inverse FFT for grids of size 256×256 or 512×512 well within the time constraints of a single frame. This is typically done in a compute shader, with the resulting data stored in a texture.

For large ocean surfaces, this grid can be tiled to cover a greater area. However, doing this tiling can reintroduce the repeating patterns that the technique is meant to avoid. To overcome this limitation, multiple FFT cascades at different length scales can be combined—much like summing many sine waves together [25]. Since modern GPUs can efficiently compute several inverse FFTs per frame, layering these cascades creates an ocean surface with the complexity and detail of millions of summed sine waves, mitigating the repeating patterns.

By blending the cascades in different ways, artists can simulate a wide range of ocean conditions, from calm waters to stormy seas.

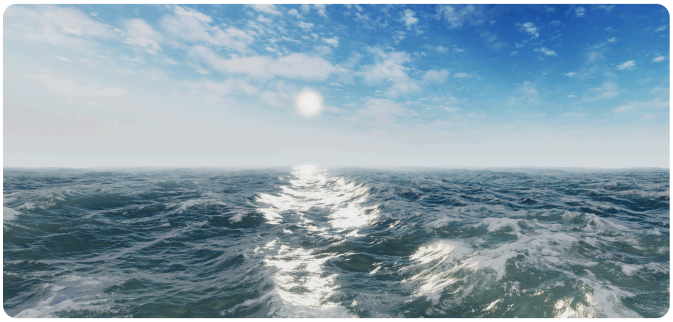


Fig. 17: The result of the inverse FFT of the wave spectrum in [25]

IV. Rendering Rivers (Flow Maps)

Flowing rivers present a different challenge from open oceans – the water is not just waving in place but moving persistently along a path. A straightforward way to animate flow is to scroll the water's texture or normal map uniformly along a path. However, real rivers twist and turn, so a single uniform scroll looks obviously fake when the river bends or splits.

Flow maps were introduced to address this, allowing spatially-varying flow direction and speed across a water surface [28]. A flow map is a 2D vector field texture mapped over the river; each pixel stores a flow direction (as a 2D vector) and often a magnitude (vector length) that represents the local water velocity.

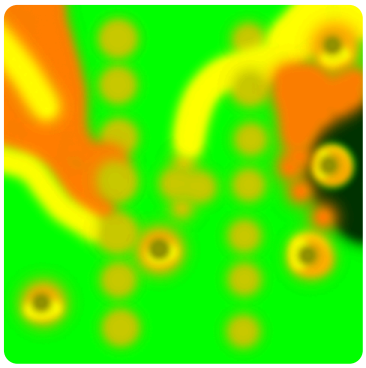


Fig. 18: A flow map texture from [28]

The water shader uses this vector field to **warp** its **UV Coordinates** over time, essentially creating a **flow** effect. This produces a convincingly realistic movement, water can accelerate down rapids, slow in pools and even curl around obstacles, all driven by a texture authored by an artist rather than a complex fluid simulation.

To implement flow mapping, the engine samples the flow map to find the direction for each water pixel, then offsets the lookup into the water's normal map (and other textures) accordingly. As time advances, the UVs slide along the flow vector, making the water appear to flow in that direction.

One practical challenge is that continuously offsetting the UVs will eventually stretch the texture too far (which will cause visible repeats or blurring). To overcome this, developers run the animation in repeating cycles: after the offset reaches a certain limit, it resets back, and by using

two layers of the animated texture out-of-phase, the shader can smoothly blend between a resetting layer and the next one without a visible jump.

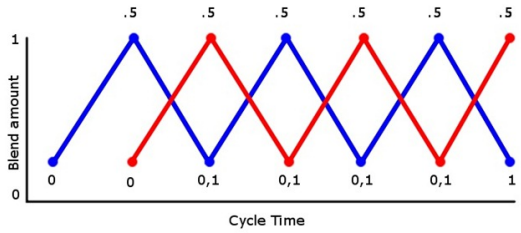


Fig. 19: The blending of two flow maps (from [28]) to avoid visible jumps

Valve employs this technique in their games, for instance, to animate the toxic pools in Portal 2 [29].

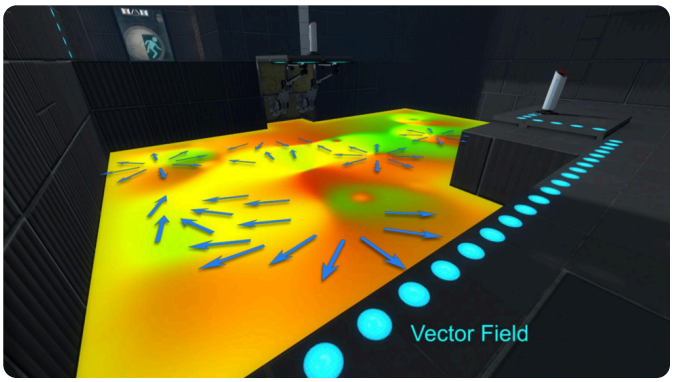


Fig. 20: The vector field used in Portal 2 from [29]

This technique is also used in Red Dead Redemption 2 [2], where rivers show calm, glassy water in pools and fast, detailed currents around rocks and rapids. The flow maps add subtle eddies and variations, making the water look highly realistic and dynamic.

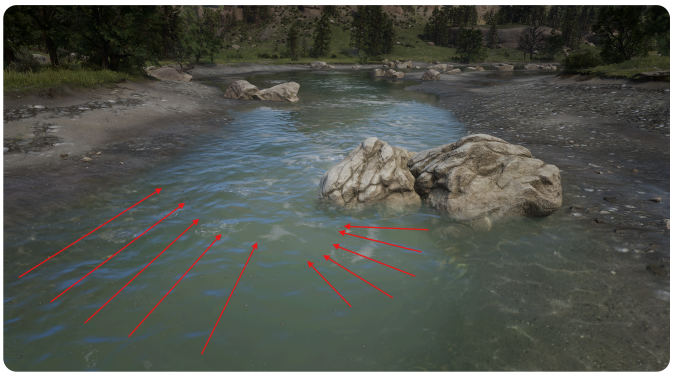


Fig. 21: Flow maps being used in Red Dead Redemption 2 [2] rivers

V. Buoyancy

Beyond visual fidelity, water in games must interact believably with objects through buoyancy. This refers to the upward force exerted by the water that makes objects float, as described by Archimedes' principle: the buoyant force equals the weight of the displaced fluid, which is formulated as:

$$F_{\text{buoyant}} = \rho_{\text{water}} \times g \times V_{\text{object underwater}} \tag{13}$$

where ρ_{water} is the density of the water (in real life it varies depending on temperature and salinity, but for simplicity it can be assumed as a constant of $1000\frac{kg}{m^3}$), g is the acceleration due to gravity, and $V_{\text{object underwater}}$ is the volume of the displaced fluid [30].

Implementing buoyancy involves estimating the V part of the equation, which involves determining the submerged volume of the object. A common technique is to subdivide the object into sample points or volumes (e.g. voxels) and test each one against the water level. Each submerged portion applies an upward force equal to the weight of the water it displaces, and the forces are summed to get the net buoyant force on the object. This method is both efficient and stable, and has been used for gameplay elements like floating crates and props in Half-Life 2 [6] that when placed underwater, it raises a heavy ramp which solves a gameplay puzzle.

More advanced buoyancy models are used for larger or irregular shapes like ships [31].

Ensuring consistency between the rendered waves (often generated on the GPU) and the physics simulation (on the CPU) is crucial; otherwise an object might visibly float above or below the water [30]. Techniques to handle this include duplicating the wave calculation on the CPU, or using asynchronous readback of the GPU height field.

Developers also dampen buoyant motion with drag forces to prevent objects from jittering or bouncing too violently on rough water. The end result, when tuned well, is a natural bobbing and rocking motion.

VI. Conclusion

This paper has discussed various techniques employed for real-time water rendering, including reflections, refractions, wave simulations, river flow using flow maps, and a brief mention of physical interactions via buoyancy calculations.

While these methods significantly enhance visual realism, the scope of this paper does not exhaustively cover all aspects of realistic water simulation. Missing components that further improve realism but remain topics for future exploration include:

- Underwater god rays, fog and volumetric lighting
- Dynamic water level-of-detail adjustments
- Water interactions with dynamic objects
- Techniques for preventing water rendering inside objects
- Simulation and rendering of water foam, especially in breaking waves and shorelines
- Generation and rendering of water spray

Each of these areas offers further avenues for enhancing visual realism and computational efficiency in real-time water rendering.

References

[1] *The Last of Us Part II*. Naughty Dog. [Online]. Available: https://www.playstation.com/en-us/games/the-last-of-us-part-ii/

[2] *Red Dead Redemption 2*. Rockstar Games. [Online]. Available: https://www.rockstargames.com/reddeadredemption2/~

[3] "Screen Space Reflections : Implementation and optimization - Part 1 : Linear Tracing Method," Jan. 16, 2021.

[4] *Rethinking Voxels (Minecraft Shader)*. [Online]. Available: https://modrinth.com/shader/rethinking-voxels

[5] *Grand Theft Auto V*. Rockstar Games. [Online]. Available: https://www.rockstargames.com/gta-v

[6] *Half-Life 2*. Valve. [Online]. Available: https://store.steampowered.com/app/220/HalfLife2/

[7] "Unreal Engine Scene Capture." [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/1.6---scene-capture-cube?application_version=4.27

[8] *Counter-Strike 2*. Valve. [Online]. Available: https://www.counter-strike.net/

[9] *Cyberpunk 2077*. CD Projekt Red. [Online]. Available: https://www.cyberpunk.net/us/en/

[10] *Cyberpunk 2077 PC: What Does Ray Tracing Deliver... And Is It Worth It?*. [Online Video]. Available: https://www.youtube.com/watch?v=6bqA8F6B6NQ

[11] "Schlick's approximation." [Online]. Available: https://en.wikipedia.org/wiki/Schlick%27s_approximation

[12] "Ocean Simulation with FFT and WebGPU," Mar. 20, 2024. [Online]. Available: https://barthpaleologue.github.io/Blog/posts/ocean-simulation-webgpu

[13] *Photon Minecraft Shaders*. [Online]. Available: https://github.com/sixthsurge/photon

[14] "Advanced Lighting - Blinn-Phong." [Online]. Available: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting

[15] "Three Things You Need to Know About WaveWorks 2.0," *NVIDIA*.

[16] "GPU Gems - Chapter 2. Rendering Water Caustics." NVIDIA. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-2-rendering-water-caustics

[17] "Generating Ray-Traced Caustic Effects in Unreal Engine 4, Part 2," *NVIDIA*. [Online]. Available: https://developer.nvidia.com/blog/generating-ray-traced-caustic-effects-in-unreal-engine-4-part-2/

[18] "GPU Gems - Chapter 1. Effective Water Simulation for Virtual Water." NVIDIA. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models

[19] *How Games Fake Water*. [Online Video]. Available: https://www.youtube.com/watch?v=PH9q0HNBjT4

[20] "The Book of Shaders - Chapter 13. Fractal Brownian Motion and Domain Warping." [Online]. Available: https://thebookofshaders.com/13/

[21] "Very fast procedural ocean - It's fast because it doesn't use noise but rather sin waves." [Online]. Available: https://www.shadertoy.com/view/MdXyzX

[22] "3D Ocean Shader Using Gerstner Waves." [Online]. Available: https://gameidea.org/2023/12/01/3d-ocean-shader-using-gerstner-waves/

[23] "Model-oriented trochoidal waves with vertex shaders." [Online]. Available: https://madblade.github.io/waves-gerstner/

[24] "Simulating Ocean Water." [Online]. Available: https://people.computing.clemson.edu/~jtessen/reports/papers_files/coursenotes2004.pdf

[25] *I Tried Simulating The Entire Ocean*. [Online Video]. Available: https://www.youtube.com/watch?v=yPfagLeUa7k

[26] *Ocean waves simulation with Fast Fourier transform*. [Online Video]. Available: https://www.youtube.com/watch?v=kGEqaX4Y4bQ

[27] *Sea of Thieves*. Rare. [Online]. Available: https://www.seaofthieves.com/

[28] "Animating Water Using Flow Maps." [Online]. Available: https://graphicsrunner.blogspot.com/2010/08/water-using-flow-maps.html

[29] "Manipulating UVs through Color Data in Portal 2," *Valve*. [Online]. Available: https://cdn.cloudflare.steamstatic.com/apps/valve/2011/gdc_2011_grimes_nonstandard_textures.pdf

[30] "Buoyancy for Dummies." [Online]. Available: https://www.vertexfragment.com/ramblings/buoyancy-for-dummies/

[31] "Water interaction model for boats in video games." [Online]. Available: https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games