State of the Art in Water Rendering

Francisco Macedo Ferreira
pg55942@alunos.uminho.pt

***Abstract*—TODO**

***Index Terms*—Water Rendering, Waves, Flow Maps, Reflection, Refraction, Ray Tracing, Buoyancy**

I. Introduction

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. 1: Screenshot of a river in Red Dead Redemption 2 [1]

From now on we will assume that we have access to the normals of the water surface, so we can use them to simulate the reflection and refraction of light. How to compute them 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 buffer 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 [2]. The SSR pass can be executed using a compute shader. Below is the pseudo code[1].

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:

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

SSR became popular in games, because as we can see in the pseudo code, it integrates well with deffered rendering (using the existing depth and normal buffers) and it is easy to implement without requiring a lot of changes to the whole pipeline. The effect is also quite fast, as it is a post-processing

[1]You can find a Metal shader implementation in the Sugu Lee's blog post [2]

effect, so its cost is independent of the number of objects in the scene, but rather depends on the number of pixels that are reflective.

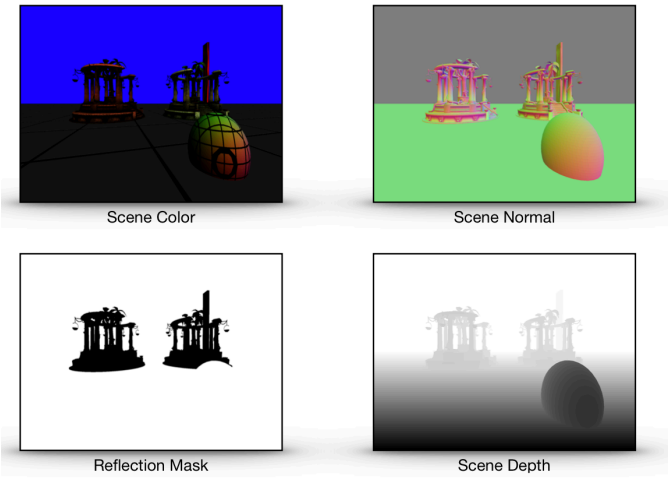

Fig. 2: SSR Buffers used in the compute shader (Sugu Lee's blog post [2])

However, SSR come 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 obect to be absent in the depth buffer.

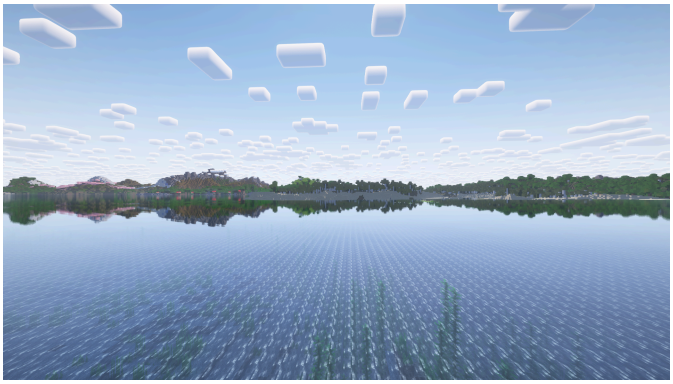

Fig. 3: SSR implementation in Minecraft Shader Mod Rethinking Voxels [3]

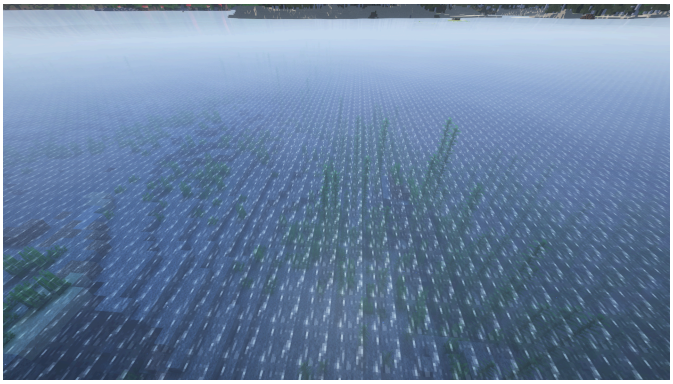

Fig. 4: Missing terrain reflections from the example above

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. 5: Planar reflection in a mirror in Grand Theft Auto V [4]

Essentially, one rendes 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 expact reflection of the main camera's orientation, making the reflection very precise.

Fig. 6: Planar reflection[2] in the water level in Half-Life 2 [5]

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" [6] 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 light-weight 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.

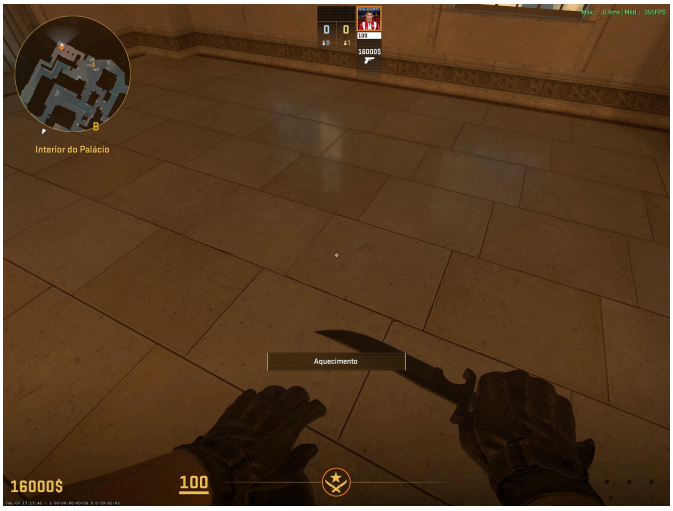

Fig. 7: Cubemap reflecting a window in Counter-Strike 2 [7]

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

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. 8: Cubemap in the scope of the AWP weapon in Counter-Strike 2 [7]

This means, for example, characters in Counter-Strike 2 cannot be reflected through the AWP scope (as seen in the image above), 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's just a texture lookup. The main **draw back** 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. 9: Ray tracing in Cyberpunk 2077 [8] from Digital Foundry's video [9]

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 includes **refraction**, the part of seeing into the water.

Real water is typically transparent, but it is view-dependant: 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 $\theta = 0$) [10].

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 perturbated by the water surface normal (scaled by the water depth) [11]. This produces a distorcion of the underwater view, approximating true refraction. If done well, you can see the lakebed or objects beneath the surface, distorced by ripples.

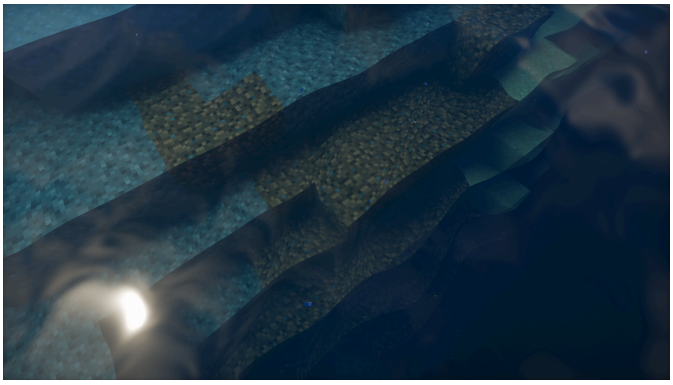

Fig. 10: Refraction in Photon Minecraft Shaders [12]

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 [13].

A more advanced method can use a BRDF (physically-based material) to calculate specular highlights, allowing for more physically accurate water surfaces [14].

Caustics also deserve a special mention. These are the focused light pattern os 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 giant amount 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 [15]. Although not physically accurate, this can create surprisingly believable results.

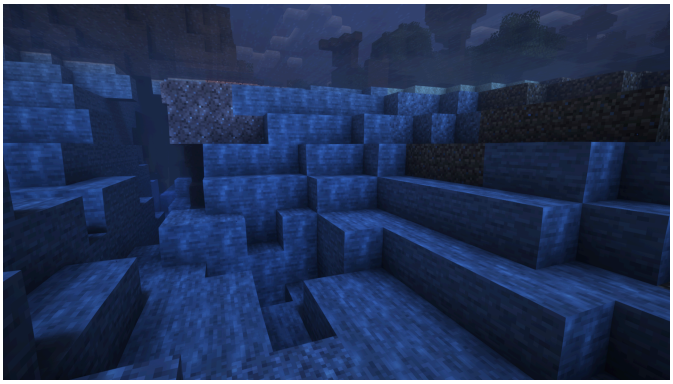

Fig. 11: Underwater caustics in Minecraft Rethinking Voxels Shader [3] using a projected texture

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 [16].

Fig. 12: Caustics from NVIDIA's raytracing technique [16]

III. Waves

A. *Sum of Sines Waves*

B. *Gerstner Waves*

C. *Spectral FFT-Based Waves*

IV. Rendering Rivers (Flow Maps)

V. Buoyancy

- https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games
- https://nejclesek.blogspot.com/2016/05/thousand-ships-with-real-time-buoyancy.html

VI. Conclusion

- god rays
- underwater fog
- volumetric lighting underwater
- water level of detail

References

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

[2] "Screen Space Reflections : Implementation and optimization - Part 1 : Linear Tracing Method."

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

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

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

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

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

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

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

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

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

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

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

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

[15] "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

[16] "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/