Simulating Water Caustics and Refraction in Three JS

Michael Suehle

Honors Project for CMSC427

Dr. Roger Eastman

December 16, 2022

Contents

Introduction

The purpose of this report is to explore how refracting surfaces and caustics are simulated in computer graphics and to explain to anyone interested in computer graphics, whether they be students taking a course in computer graphics or videogame developers, how to simulate refracting surfaces and caustics in real time. This report discusses what caustics and refracting surfaces are, common methods for simulating them, and how they can be quickly rendered in real time using Three JS and WebGL. To go along with this report, I created a scene in WebGL using the methods I describe, available here.

Why Three JS and WebGL?

WebGL with Three JS is a great rendering engine for anyone interested in learning computer graphics because there are many resources for learning it and it ma easy to share and showcase work. WebGL is a rendering engine that runs on browsers. Since it runs on browsers, it is simple to run and display creations, whether they be renders, animations, or even games, on multiple platforms. It is also easy to find and learn from examples since it is possible to inspect the source code of nearly every script that uses WebGL on the web. Three JS is a well-documented JavaScript library that works with WebGL and provides functions for common rendering procedures such as moving the camera, creating meshes, and lathing points to create geometries.

Refracting and Reflecting Surfaces

Image credit: Tran, G. (2009). *An image created by using POV-Ray 3.6.* Wikipedia, https://en.wikipedia.org/wiki/Rendering_%28computer_graphics%29#/media/File:Glasses_800_edit.png

What are they?

Refracting surfaces are surfaces that light refracts through. They are typically made of transparent or partly transparent materials such as water, glass, or clear plastic. When light travels from one material to another, it generally refracts at an angle determined by Snell's law. Snell's law defines the following equation:

$$\eta_1 sin(\theta_1) = \eta_2 sin(\theta_2)$$

Where η_1 and η_2 are the indices of refraction for the material light is traveling from and to respectively, and θ_1 and θ_2 are the angles that the light rays hit and refract through the surface of the new material respectively.

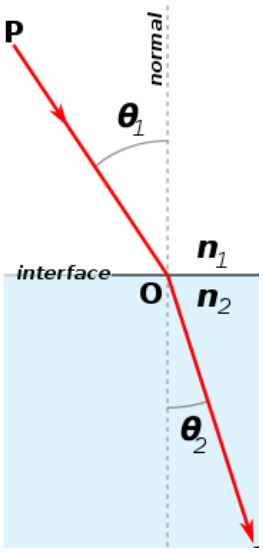


Image credit: Alexandrov, O. (2007). *Illustration of Snell's law*. Wikipedia, https://commons.wikimedia.org/wiki/File:Snells_law2.svg

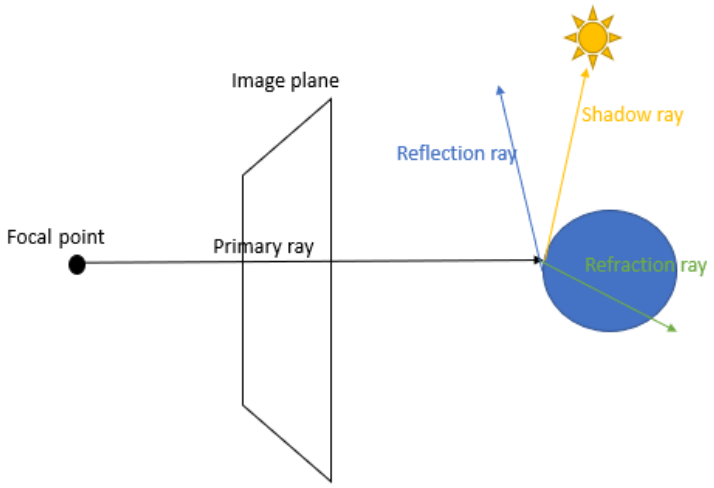
In the figure above, the light ray traveling through the first material, P, hits the surface of the second material at an angle of θ_1 relative to the normal to the second surface. Then it refracts through the surface at an angle of θ_2 relative to the same normal.

Not all the light gets refracted through surfaces. Some reflects off surfaces too. Fresnel equations are used to determine the percentage of light that gets refracted and the percentage of light that gets reflected.

How are they simulated?

Raytracing is one of most common and straightforward methods for simulating reflection and refraction. Raytracing is a method used to render sharp and hyper realistic images. It involves sending out rays from each pixel on the screen and tracing how they interact with the scene. Each time a ray intersects an object in the scene, it sends out secondary rays consisting of a ray to each light source, called shadow rays, a ray that reflects off the object, called the

reflection ray, and a ray that refracts through the object, called the refraction ray. The color of each pixel is determined by summing over the color information each ray obtains when it intersects objects or the background.



Ray tracing can be a fairly computationally intensive rendering method. Three JS and WebGL do support raytracing, but real time raytracing may not work on very many types of hardware. Some examples of raytracing using Three JS and WebGL can be found here. WebGL developers found a less computationally intensive method to simulate refraction and reflection without raytracing..

Three JS implementation

In his water caustics example, Martin Renou uses a temporary render target to simulate the effect of refraction without using ray tracing. A render target is used to create a texture from a render.

Renou uses the following code to instantiate one:

```
const width = canvas.width;
const height = canvas.height;
// Target for computing the water refraction
const temporaryRenderTarget = new THREE.WebGLRenderTarget(width, height);
```

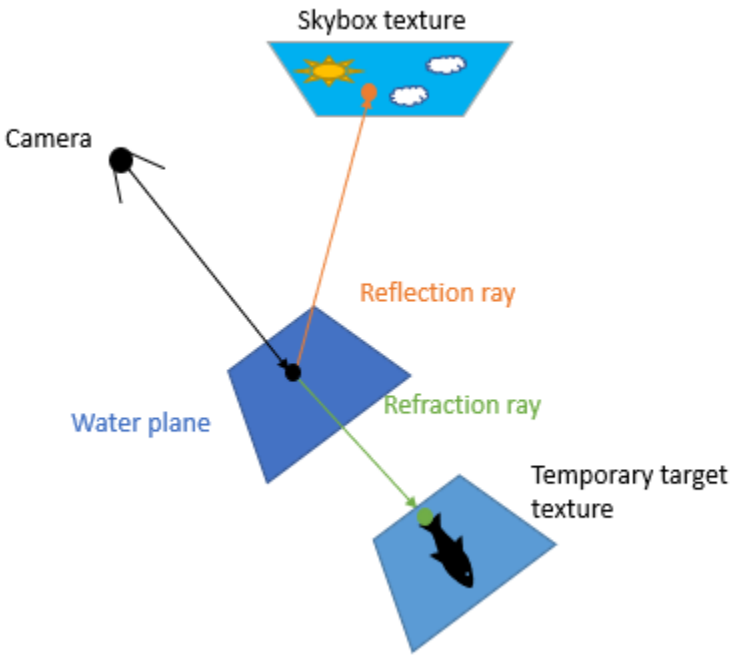
He sets the render target and renders to the target the scene with anything that could be seen through the water, which is everything except for the water in this case. Then he gives the water the texture generated by the render target. By doing this, the water effectively becomes ‘see through’ since it has the texture of everything that the camera can see behind it.

```
// Render everything but the refractive water
```

```
renderer.setRenderTarget(temporaryRenderTarget);
water.mesh.visible = false;
renderer.render(scene, camera);
water.setEnvMapTexture(temporaryRenderTarget.texture);
```

In the code snippet above, 'water' is an object that contains the mesh information for the water plane in the scene and 'camera' is an instance of a Three JS Perspective Camera.

Setting the water texture is not enough to fully simulate refraction however. In the vertex shader for the water, found here, Renou calculates the position on the water texture that a refracted ray hits as well as the percentage of light that gets reflected and refracted, and passes that information to the fragment shader which blends the color of the texture at that position with the color of the part of the texture on the skybox that the reflected ray hits.



Renou uses the following approximation of the Fresnel equation to determine the percentage of light that gets reflected.

$$reflection\ factor = b + fs * (1 + eye \cdot n)^{fp}$$

Where b is a hard-coded Fresnel bias, fs is a hard-coded Fresnel scaling factor, fp is a hard-coded Fresnel power, eye is the vector from the camera to the water vertex, and n is the normal of the water surface. The refraction factor can then be calculated as 1 – reflection factor, but Renou hardcodes in a reflection factor of 1, likely to maximize visibility of the objects under the water.

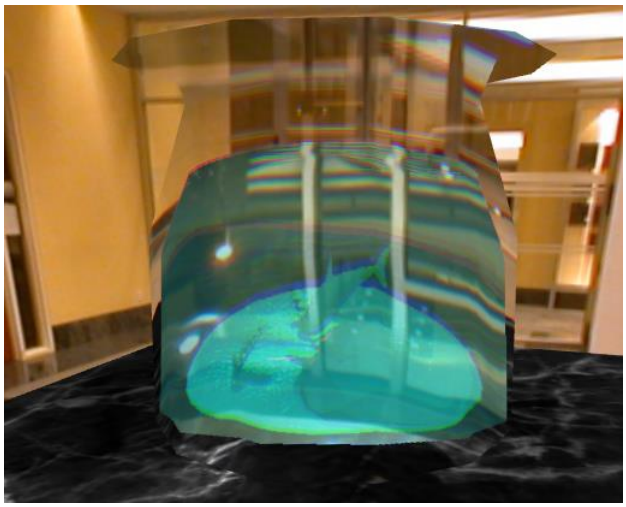
The refracted position on the texture is then calculated as the following:

$$refracted\ position = pos + refractionFactor * refractedRay$$

Where pos is the vertex on the water plane that *eye* is passing through and refractedRay is ray that refracted through the water plane. refractedRay is calculated using the glsl refract function.

The refracted color is then calculated using the glsl texture2D function. The reflected color is calculated using the reflectedRay, which is calculated using the glsl reflect function, and the glsl textureCube function. These two colors are mixed based on the reflection factor to produce the color that pixels associated with the water plane are assigned.

I adapted this code for water shading into code for glass shading to create a glass vase. This code can be found here. The main change I had to make was to change the *eta* variable to reflect the ratio between the index of refraction of air and the index of refraction of glass. The vase was created using the Three JS constructor, LatheGeometry.



Caustics

Image credit: *Underwater caustics*. alanzucconi. https://www.alanzucconi.com/2019/09/13/believable-caustics-reflections/underwater-caustics/

What are they?

Caustics are light patterns that are created by regions where multiple refracted light rays intersect. These occur when light passes through disturbed, refracting surfaces such as water with waves and curved glass.

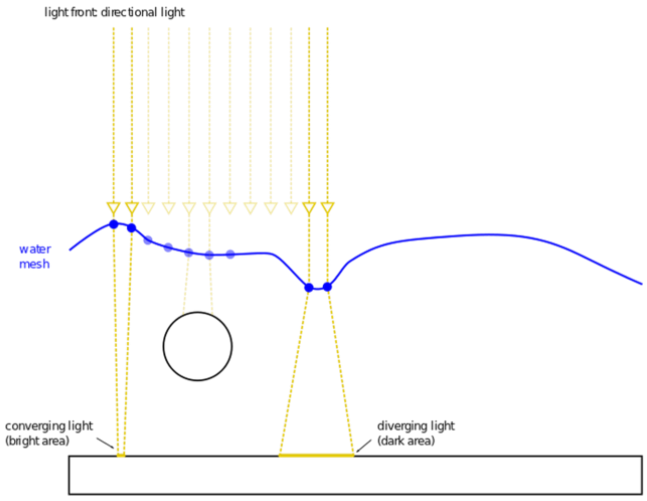


Image credit: Renou, M. (2020). *Real-time rendering of water caustics*. Medium. https://medium.com/@martinRenou/real-time-rendering-of-water-caustics-59cda1d74aa

How are they simulated?

Caustics are generally simulated with some form of tracing rays to a light source. Path tracing is one technique that is used. Path tracing is essentially ray tracing except, when each ray hits an object, it sends out additional random rays. The final color of each pixel is then determined from a sample of the random rays (Evanson).

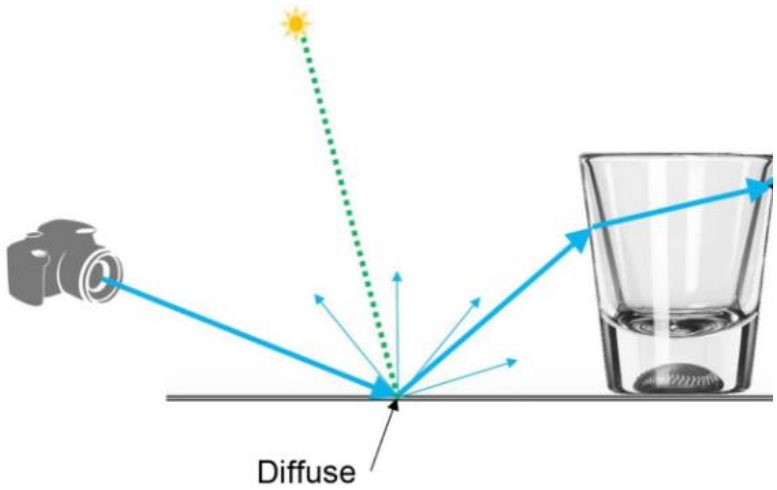
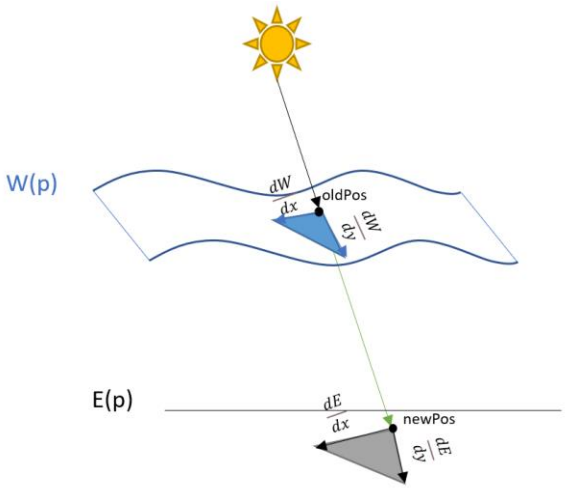


Image credit: Nichols, C. (2019). *What are caustics and how to render them the right way*. Chaos. https://www.chaos.com/blog/what-are-caustics-and-how-to-render-them-the-right-way

Developers often simulate caustics ahead of time to produce a texture animation that gets projected onto surfaces. This is cheap to do in real time, but there is no shadowing or dynamic lighting (NVIDIA Game Developer). Alternatively, caustics could be simulated in real-time, but this is often more computationally intense. To simulate caustics in Three JS, Even Wallace, the author who wrote the article that motivated Renou to create his example, combines the ideas of using a texture and tracing rays.

Three JS implementation

Wallace generates a caustics texture to map on the part of the scene that has caustics projected on it, the environment. He does this using an approximated "caustic intensity". The approximated intensity is calculated using the proportion of the area of the triangle in water mesh that is hit by the light ray divided by the area of the triangle in the environment that is hit by the refracted light ray.



If we let *W(p)* and *E(p)* be equations representing the water mesh and the surface of the environment respectively, we get the following equation for the intensity.

$$intensity = \frac{||\frac{dW}{dx}(oldPos)||||\frac{dW}{dy}(oldPos)||}{||\frac{dE}{dx}(newPos)||||\frac{dE}{dy}(newPos)||}$$

Note that since the water mesh and the environment are made up of triangles, the partial derivatives at a given point are the rays that make up the base and height of the triangle the point is on. Also note that the $\frac{1}{2}$ term in the triangle areas cancels out, so it is not in the equation.

In the shaders for the caustics, Wallace computes oldPos and newPos in the vertex shader and passes their values to the fragment shader. This shader is then used to generate a texture, like the one below, that gets overlayed on the environment.

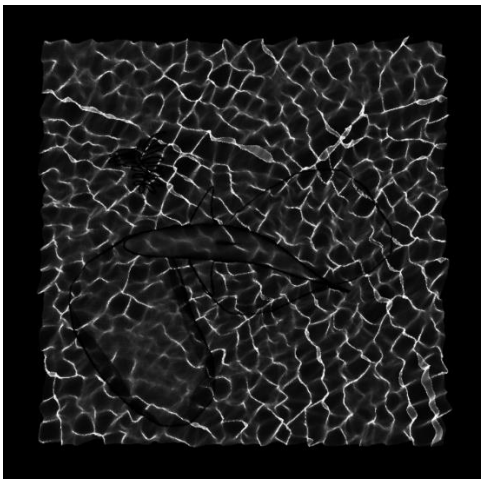
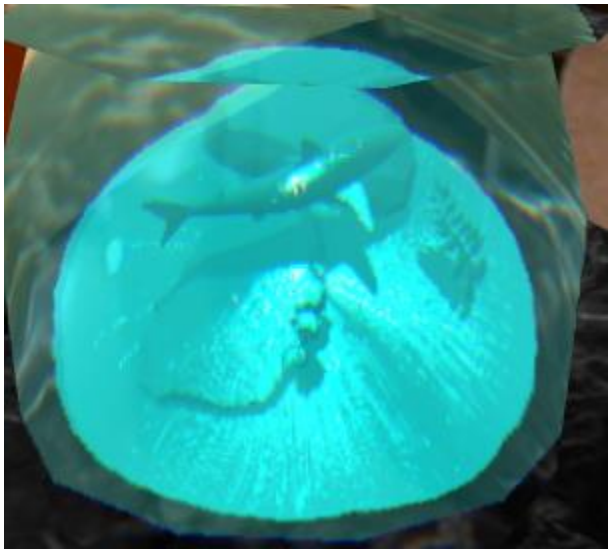


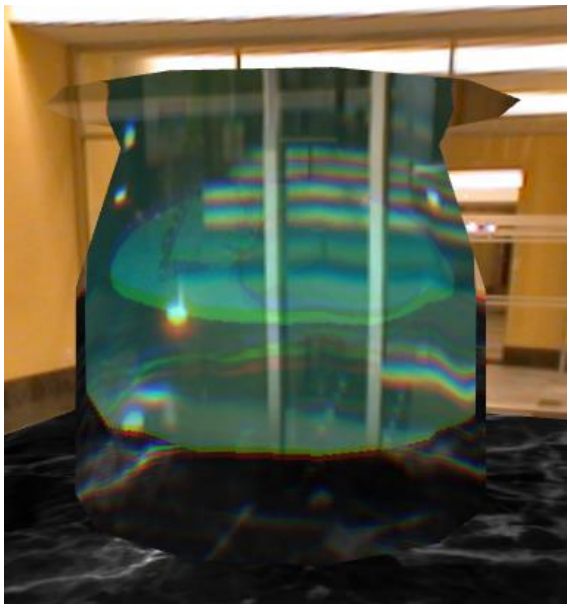
Image credit: Renou, M. (2020). *Real-time rendering of water caustics*. Medium. https://medium.com/@martinRenou/real-time-rendering-of-water-caustics-59cda1d74aa

For my scene, I changed the environment to be circular so that it would fit in the vase. This resulted in some high intensity, jagged caustics around the center of the environment. This is likely due to the circular geometry defining the floor of the environment having much smaller triangles than a rectangular plane.



Limitations

While working with these methods for simulating refraction and caustics, I found some limitations. One limitation I found with the refraction simulation is that it centers the environment at the origin when it gets rendered to a temporary render target. When the temporary render target's texture gets mapped to the refracting surface, it is also centered. This works fine for the plane of water centered above the environment, but when I created the vase, it made the environment appear floating in the center of the vase (see image below).



I found a work-around by moving the camera up before creating the temporary render target and moving it back down after. This looks fine for most camera angles, but it looks a little off after zooming in a certain amount.

Another limitation is that the water simulation is dependent on the vertices in the water's geometry. The wave simulation moves vertices up and down, so it works well with a flat geometry that has many vertices scattered throughout. Since the Three JS circleGeometry I used only has vertices in the center and on the edges, the wave simulation does not work as well. A work-around for my vase would be to create a custom geometry that is still disc-shaped, but has many vertices distributed throughout.

Conclusion

Rendering techniques such as ray tracing and path tracing are commonly used for rendering refracting surfaces and caustics, but they can be too computationally intensive to run on common hardware. Luckily, there are less intensive methods that can be used. Although the methods explained in this paper are not perfect, they still simulate the effects of refraction and caustics fairly quickly and inexpensively. These methods may not be the best for rendering hyper realistic images, but they would work well for games that are meant to run in browsers. There are many additional methods not mentioned in this paper for rendering refracting surfaces and caustics. Some methods are less realistic and less computationally intensive while others are more realistic and more computationally intensive. Ultimately, it is important to find a balance between quality and intensity that fits your needs.

References

Alexandrov, O. (2007, December 25). *Illustration of Snell's law*. Wikipedia. Retrieved from https://commons.wikimedia.org/wiki/File:Snells_law2.svg.

Evanson, N. (2022, July 11). *Path tracing vs. Ray Tracing, explained*. TechSpot. Retrieved December 14, 2022, from https://www.techspot.com/article/2485-path-tracing-vs-ray-tracing/

Nichols, C. (2019, November 7). *What are caustics and how to render them the right way*. Chaos. Retrieved December 14, 2022, from https://www.chaos.com/blog/what-are-caustics-and-how-to-render-them-the-right-way

NVIDIA Game Developer. (2019). *Working With Ray Traced Water Caustics in Dxr*. *YouTube*. Retrieved December 14, 2022, from https://www.youtube.com/watch?v=l-wTZLjhZ5Y.

Renou, M. (2020, October 1). *Real-time rendering of water caustics*. Medium. Retrieved December 12, 2022, from https://medium.com/@martinRenou/real-time-rendering-of-water-caustics-59cda1d74aa

Tran, G. (2009). *An image created by using Pov-Ray 3.6*. Wikipedia. Retrieved December 16, 2022, from https://en.wikipedia.org/wiki/Rendering_%28computer_graphics%29#/media/File:Glasses_800_edit.png.

Underwater caustics. (n.d.). alanzucconi. Retrieved from https://www.alanzucconi.com/2019/09/13/believable-caustics-reflections/underwater-caustics/.

Wallace, E. (2016, January 7). Rendering realtime caustics in webgl. Medium. Retrieved December 14, 2022, from https://medium.com/@evanwallace/rendering-realtime-caustics-in-webgl-2a99a29a0b2c

Wikimedia Foundation. (2022, November 9). *Snell's Law*. Wikipedia. Retrieved December 12, 2022, from https://en.wikipedia.org/wiki/Snell%27s_law