Real-Time Water Caustics

Sebastian Collard
University of Calgary
Calgary, Canada
sebastian.collard1@ucalgary.ca

Kamalpreet Mundi
University of Calgary
Calgary, Canada
kamalpreet.mundi@ucalgary.ca

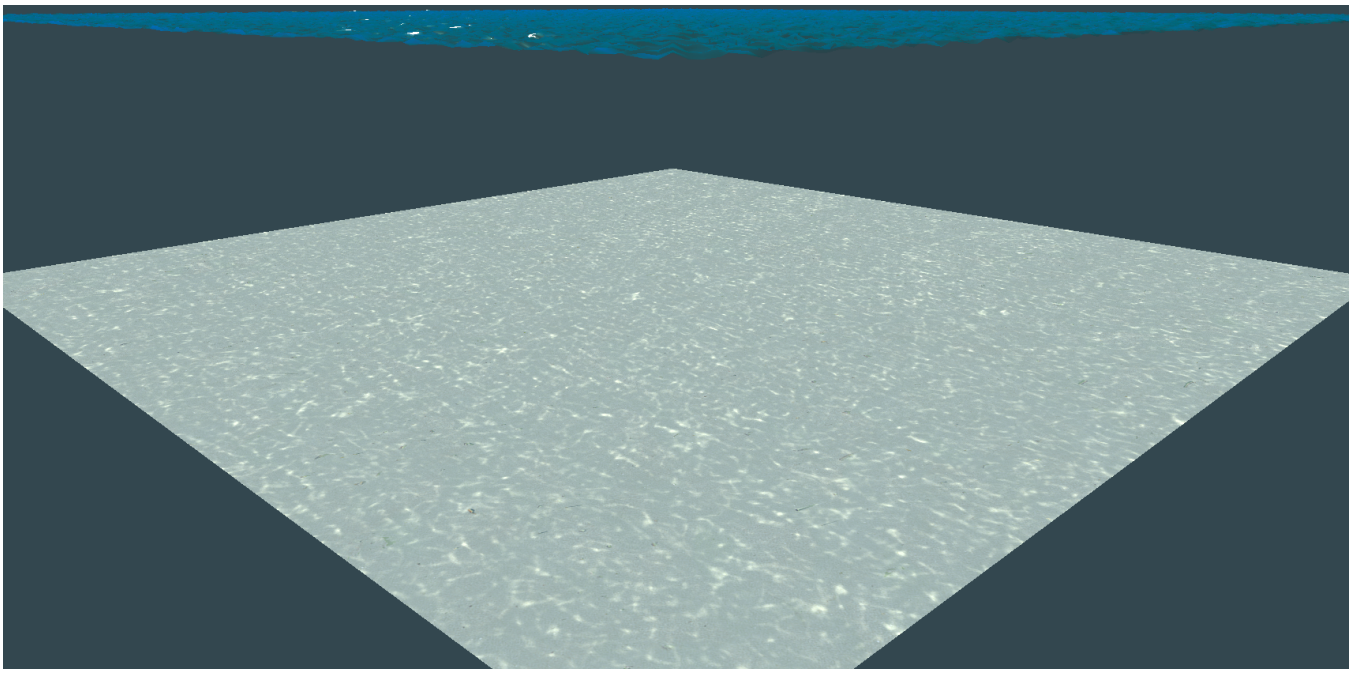

Figure 1: Still image of our caustic implementation displaying wave preset 1

ABSTRACT

Real-time rendering of water caustics has been an ongoing topic of research for quite some time. With the rise of gaming as a medium of entertainment, video game graphics are constantly improving year after year in search for the greatest results through the quickest means. As a result, the generation of playable environments have increased in realism while also having to abide by real-time constraints. For water related environments, many fine details are required to come together in order to generate believable and realistic looking water behaviour. Caustics are one of such details. In this paper we will be proposing an implementation strategy that generates acceptable water caustics in a 3D environment that responds directly to the water surface deformation and the depth of the ground receiving the caustics.

KEYWORDS

Caustic, Texture, G-buffer, Environment map, Refraction

1 INTRODUCTION

In a lit environment, water caustics can be observed as unusual yet fascinating light patterns that appear on underwater surfaces. Such environments may include your bathroom, specifically when taking a bath in a bathtub, or coastal vacation hot spots, where the water is so clear that the grainy sandy floor can easily be seen and touched. The peculiar manner in which light interacts with water results in certain regions of the ocean floor receiving a greater focus of light than others, creating the light patterns. This occurs because water on the surface is both a refractive and reflective surface, sending rays in different directions than their incident direction. In this paper, a baseline algorithm is followed and then improved to generate better looking caustics. The revised algorithm we propose for generating caustics can be broken into 4 general steps:

(1) Wave function update
  (a) Update deformation applied to wave surface as time passes

(2) Caustic Evaluation
  (a) Deferred rendering pass to extract relevant geometric data of water surface
  (b) Caustic Map calculation
  (c) Applying blur effect to caustic map

(3) Ground Floor Rendering
  (a) Blending ground floor with caustic map

(4) Water Surface Rendering
  (a) Applying water shading to deformed surface

2 RELATED WORK

Real time water caustics was first explored by Jos Stam [4]. The method proposed uses wave theory and a pre-computed texture to generate light patterns onto the ground floor. A blend pass is applied to blend in the light pattern with the ground floor. The texture is then animated using a random phase function which is correlated over time to map a texture to coordinates. While the approach does produce a caustic result, it highly favors performance over accurate and detailed looking caustics.

Another proposed method involves using a single ray intersection with the water to define caustic behavior per-vertex on the ground [5] [1]. This approach uses three different surfaces, a water surface plane, a ground plane, and a fine mesh plane which has the added constraint of having the same geometry as the water surface. For each vertex in the fine mesh plane, the idea is to project a single ray vertically, collide it with the water surface plane and then use Snell's law to refract the ray. Once this ray is refracted, the ray is mapped to set of texture coordinates. The texture coordinates are used to read from an environment map, which assigns an intensity value to each vertex on the fine mesh. Then, by using a blend function between the ground floor and fine mesh, caustic patterns can appear on top of the ground floor. This does produce a caustic result, but the single ray is not very accurate since it only takes into account light contributions from one direction, causing total light distribution per-vertex to depend on that single ray only.

3 APPROACH BREAKDOWN

The latter approach discussed[5] [1]is considered as a starting point for the final algorithm we propose. Due to limitations present in the caustic patterns from the reliance on one raycast, our method expands the algorithm in this area with several additions, allowing for more variation and detail to be incorporated in the final result. Such improvements include:

(1) Caustic results per-vertex depend on multiple light rays and directions
(2) Blurring to caustic result to get rid of traces of checkerboarding or pixelated artifacts
(3) Combining the fine-mesh and ground floor together to perform a better composite on the ground using the caustic
(4) Improving visual appearance of water surface

The following is a high level overview of our rendering pipeline:

(1) Water surface update
    (a) Starting with a flat plane containing sufficient detail for deformation, calculate the new height of each vertex on the water plane using a wave function.
    (b) Update the vertex normals after the wave function has been applied.
    (c) Buffer the water mesh's data using newly calculated positions and normals.
(2) Render wave surface normals to a texture
    (a) Resize and reorient the water plane such that it can be captured in OpenGL's (x,y) coordinate space for 2D rendering.
    (b) Output the 3D components of the normals into the RGB channels of the texture.
(3) Render caustic map to a single-channel texture
    (a) Using the aforementioned normal texture and an environment map, calculate accumulated light contribution from the number of sample refractions specified.
    (b) For each refraction, take the dot product of the refracted direction and the direction of the sun to determine how much sunlight the fragment should receive from the single refraction.
    (c) Fit the ground mesh to screen coordinates and output the accumulated light intensities to the red channel of the texture.
(4) Render post processed caustic map that applies optional smoothing to the result
    (a) To alleviate possible artifacts in the caustic texture, we introduce an intermediate step where we apply a blur to the caustic map, before using it for ground rendering.
(5) Ground Rendering
    (a) Apply sand texture as a base color.
    (b) Additive blend with the caustic texture and clamp values where necessary.
    (c) Apply attenuation coefficient based on depth of ground plane from the water surface.
    (d) Add influence water color.
(6) Water Rendering
    (a) Use phong shading as a baseline.
    (b) While the water is rendered with a blue tint in mind, we introduce more green and transparency based on the steepness of the normal with respect to the sun direction.
    (c) Final specular result is exaggerated from phong's specular to evoke more of a glistening effect on light reflections.

3.1 Water Surface Update

We start with a square plane centered at the origin point containing a uniform distribution of vertices. The water geometry is updated first because all subsequent steps depend on the state of the water surface in some manner. Every time we reach the beginning of the render loop, a chosen wave function dictates what the new vertex positions will be. Positional updates are only applied to the height component of the water surface, and so both the X and Z components remain constant throughout runtime. Sine functions are primarily used to morph the starting plane geometry into various wave patterns, where common variables such as amplitude, frequency and time are used as global modifiers to control the overall

size and speed of the wave[2]. To propagate noticeable variation across the surface, values that are vertex-dependant are utilized such as x and z components or distance from origin.

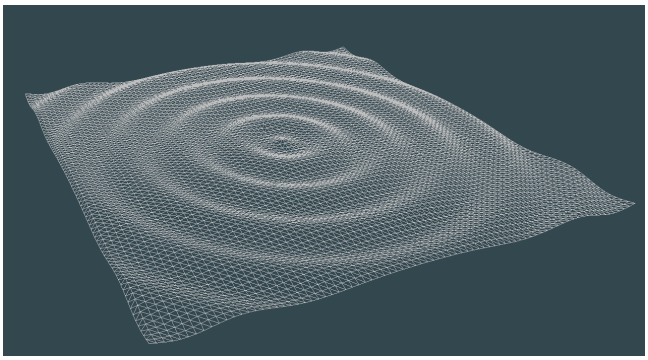

Figure 2: Plane mesh with height component driven by the function $Amplitude * sin(-\pi * Distance * Frequency + Time)$

To achieve wave behaviour that can imitate more realistic scenarios such as the ocean surface, a more random approach is needed that eliminates easy-to-spot patterns. To do this, we use a non-periodic wave function which, in the same vein as 1D perlin noise, can be used to introduce randomness that preserves a sense of continuity across timesteps.

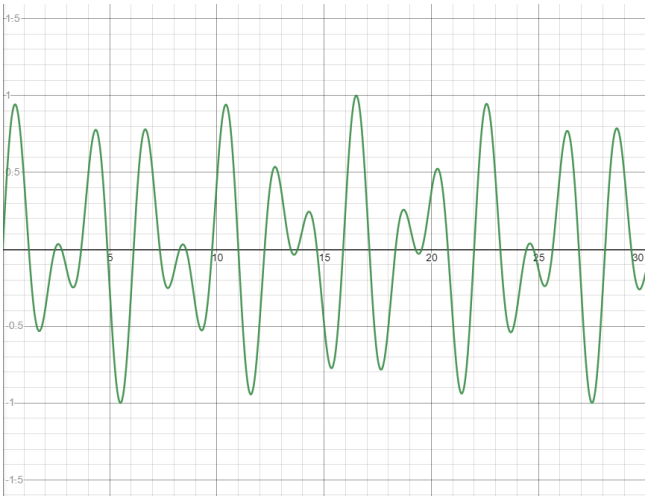

Figure 3: Plot of non-periodic function: $\frac{sin(2x)+sin(\pi x)}{2}$

With the wave function exclusively varying the Y-component, the non-periodic function is applied to each vertex (making use of the same collection of inputs previously discussed). While this grants the continuity needed to avoid jittery behaviour in a per-vertex setting, vertices looking discontinuous relative to one another remains a possibility. Thus, the non-periodic function is strictly used in our model for subtle variations such as ripples. Additional sine functions are summed on top of this result to introduce greater wave patterns across the surface.

Once all vertices have had their positions updated, they are ready to be buffered along with the normals after they have also been addressed.

3.2 Water Surface Normals

With updated positions for all vertices on the water mesh, each vertex normal needs to be updated as well. This is done by iterating over the list of vertex positions in increments of 3. For every 3 vertices, we determine their new normals by calculating the face normal of the triangle formed by the vertices. The three positions can form two direction vectors denoting segments of the triangle, which can used to perform the cross product for the normal direction:

$$
\begin{aligned}
&for(int\ i = 0; i < indices.size(); i+ = 3)\\
&\quad p_1 = vertex(indices[i])\\
&\quad p_2 = vertex(indices[i+1])\\
&\quad p_3 = vertex(indives[i+2])\\
&\quad faceNormal = normalize(cross(p_2 - p_1, p_3 - p_1))\\
&\quad normals[3*i] = faceNormal.x\\
&\quad normals[3*i+1] = faceNormal.y\\
&\quad normals[3*i+2] = faceNormal.z\\
&endfor
\end{aligned}
$$

In order to pass in large amounts of data into the vertex or fragment shader for calculations, a texture's RGB channels can be used to store information about vector positions or normals. Uniforms can be used as well, though shaders have a limit to how many uniform values can be passed to them. In order to avoid this limit while also keeping things organized in the shader, textures are used as the storage mechanism to pass the data over. The wave normals are crucial for our implementation, as they are required farther into the render process for the caustic map generation. Therefore, a geometric rendering pass will be done on the water surface in order to extract the normals.

To store this information, a G-buffer is first created which holds an RGB color buffer for the normal texture. Once the G-buffer is initialized, the water surface plane has to be resized and reoriented in the main loop to prepare for 2D rendering. Since the default render plane is $x : [-1, 1], y : [-1, 1]$, depending how the original water surface is set up a custom rotation, scaling, and translation has to be applied. In our case the 3D water surface plane was centered at origin along the x and z coordinate plane, spanning across [-2,-2] and [2,2]. The plane is then rotated 90 degrees along the x axis, and scaled down by 0.5 to prepare for rendering on the XY plane. In the fragment shader the normals are set as the output color, storing them into the texture's RGB channels.

3.3 Caustic Map

From Nvidia's GPU Gems approach[5], the topology of the fine mesh has to match the water mesh from a top-down perspective. This satisfies the assumption that at each vertex position on the fine mesh, there is a vertex belonging to the water surface that is always directly above it. This works well in their approach as they only concern themselves with sampling once, using the normal

of the wave directly above. Since the additional fine mesh on top of the ground plane is not used for our method, the caustic map is instead rendered with the same mesh used for the final ground composite. Additionally, our strategy builds upon the idea of adding finer details to the caustics by increasing the sampling count to light each fragment. Instead of strictly looking up from the current position, a rectangular area on the wave is defined such that the water directly above lies at the center of it:

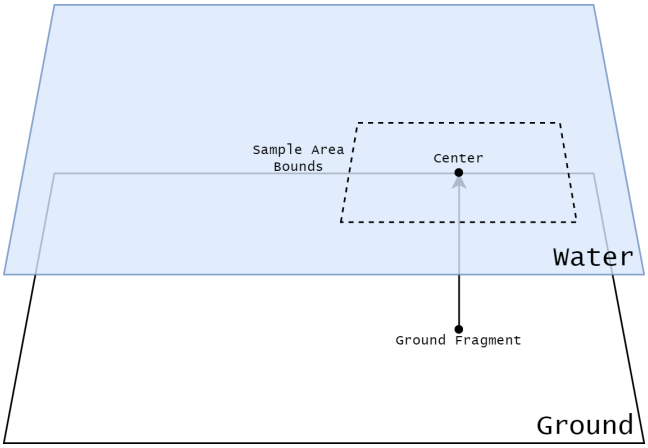

Figure 4: Area on the water from which multiple samples can be taken in order to add more caustic detail.

The sampling area remains centered above the fragment position as a simplification of our model, which assumes that the shortest path from the current ground vertex to the water surface is up. In the event where the wave surface is of a more chaotic nature, waves might dip lower to the ground in other areas, resulting in the shortest path being elsewhere than up. When dealing with cases like these, we chose to expand the area of the square, in an attempt to include that section of the wave in the sample square. In order to accumulate light contributions from the additional samples lying within the square, the positions and normals of the relevant water vertices must be known when working in the shaders. Passing all positions and normals belonging to water vertices as uniforms can exceed the capable limit rather quickly when working with semi-detailed surfaces. What is instead employed for the sampling is the texture-based approach discussed in the previous section. All the wave surface normals are stored in a texture from the previous render step, bypassing the uniform limit of shaders.

Rather than working with surface information in 3D space, all water normals needed for caustic evaluation can be directly accessed in texture coordinate space by reading from the texture. With the normal texture capturing the entire wave surface from a top-down view, the center of the sampling square requires converting the ground fragment position to texture coordinate space. Since the position of the current ground fragment is known in the shader and both surfaces cover the same area in XZ space, the texture coordinate mapping of the ground fragment is the same as the location on the water directly upwards. With that in mind, the ground surface is a 4 unit-wide square centered at the origin, the XZ positions are converted to texture coordinates by first offsetting both XZ components by 2, moving the ground coordinates from $xz : [-2, 2]$ to $xz : [0, 4]$. After scaling the ground by $1/4$ the ground lies within $xz : [0, 1]$, resulting in a one-to-one mapping to $xy : [0, 1]$ for texture coordinates. The bounds of the sampling square are now defined using bounds on the texture coordinates, and the water surface sampling is now defined by reading pixels on the normal texture located within these bounds.

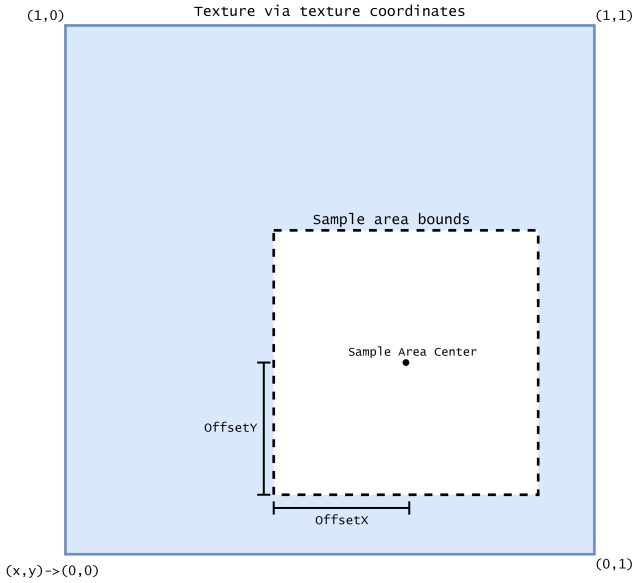

Figure 5: Sample area defined in terms of the texture being used. Values are read by pixels chosen within bounds specified through texture coordinates.

With dimensions of the sample area conveyed through component-wise offsets, a double for loop is used to step through the sample area using evenly spaced increments based on a predefined *sampleSteps* uniform. It is worth noting that in edge cases where a section of the sample area falls out of range of both the ground and water surface, GL_MIRRORED_REPEAT is used for texture wrapping of the normal texture. While this can introduce patterns of overlap around the borders, retaining even sampling rates by relying on approximations near the borders rather than reading out-of-bound coordinates as black pixels ended up producing more favorable results. For each sample taken in the iterative process, the following steps are carried out to calculate light contribution from one refraction:

(1) Construct a 3D vector approximating the direction from ground to water
(2) Calculate the dot product of the approximation and the water normal
(3) Construct a new refracted direction using the dot product as a weight on the normal
(4) Calculate the dot product of the refraction and the sun direction
(5) Convert the second dot product into a texture coordinate mapping for the environment map

(6) Multiply intensity of environment map value by a distance falloff
(7) Add result to output variable

Firstly, a 3D vector roughly approximating the direction from the ground fragment to the current sample location on the water is constructed. With the omission of generating a position texture to calculate caustics, positions along the water are instead approximated using a $groundOffset$ uniform, which indicates the height difference between the ground plane and the water plane *without the wave function*. To get the XZ components of the 3D vector, the texture coordinates of the sample area's center is subtracted from the current texture coordinates. This gives the relative direction of the current texture coordinates from the center. To complete the 3D vector, the $groundOffset$ variable is used as an approximated height component.

After normalizing this direction, the normal vector is fetched from the texture using the current texture coordinates. The first of two dot products are carried out using these vectors, which will be used as a weighting term.

For our approach, a heavier reliance on the incident vector (from ground to water) to determine the refracted direction ended up requiring more samples to get sufficient results. Performing a standard refraction using an IOR of 1.333 also produced similar behaviour, deviating rays too far from the sun to capture enough light. Instead, the refracted direction used is primarily weighed towards the normal along the wave and only takes on small directional variation based on the incident vector. As this sends refracted rays further up towards the overhead sun's direction when the normal is closely pointed toward the sun, it allows our chosen environment map's falloff to be used more effectively. This ends up producing caustics that require less samples to start showing interesting emerging patterns. Thus, the refracted direction is initialized using the normal as a baseline and the previously calculated weighting term is used to introduce refractive variation.

After having calculated the refracted direction, another dot product is used. This time between the refraction and the up vector. For our model, the sun position remained fixed directly overhead. As the sun is very far away, we also assume that this direction applies to any point on the water surface. Given light intensity is queried through the environment map, the idea is to convert this dot product into a sensible texture coordinate mapping for it.

Figure 6 shows the grey-scale environment map that acts as the sun intensity. With the brightest pixel located at the center $(0.5, 0.5)$, distancing yourself from this point introduces falloff until hitting the black borders of the image, indicating no light intensity at all. With the sun direction and refraction direction's dot product in mind, an ideal mapping of the dot product to the environment map's texture coordinates is:

$$dot : [0, 1] \rightarrow texCoord : \{[0, \frac{1}{2}], [0, \frac{1}{2}]\}$$

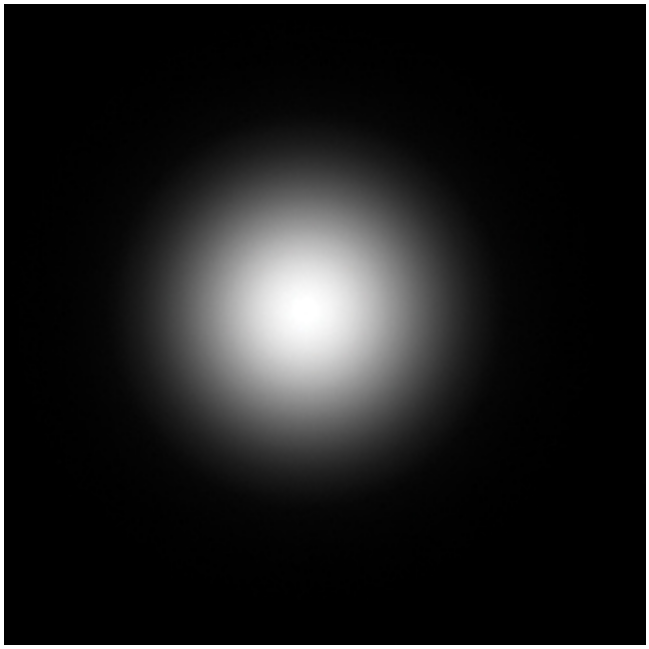

Figure 6: Environment map used in our implementation to capture sunlight intensity and falloff

Consequently, a dot product equal to 1 maps to $(0.5, 0.5)$ in texture coordinates and as the dot product decreases to zero, so do the texture coordinates. The texture coordinate derivation for the environment map then proceeds as follows:

$$sunDir = vec3(0, 1, 0)$$
$$float\ mapping = max(\frac{dot(refracDir,\ sunDir)}{2}, 0)$$
$$vec2\ texCoords = vec2(mapping)$$

Before adding the intensity value to the $FragColor$ output variable, we first multiply it by a couple remaining variables: $baseIntensity$ and $distanceIntensity$. $BaseIntensity$ is a user-controlled variable brought in as a uniform to adjust the overall brightness of incoming light refractions (for cases with too much exposure or a lack thereof). $DistanceIntensity$ uses the XZ components of the approximation vector and computes the length of it. Since this vector resides in texture coordinate space $[0, 1]$ we define $distanceIntensity$ by subtracting the length of the vector from 1. In our implementation, raising this value to multiple powers also serves as a way to control the falloff amount this variable applies to incoming light from a single refraction.

To render the caustics map as a texture, the ground plane is reoriented and scaled accordingly for 2D rendering. Once all the sample contributions have been accumulated for each pixel using the above steps, the caustic map is output as a single-channel image using the red channel and stored in a newly created G-buffer. The redness of a pixel contained between [0,1] dictates the intensity of the caustic.

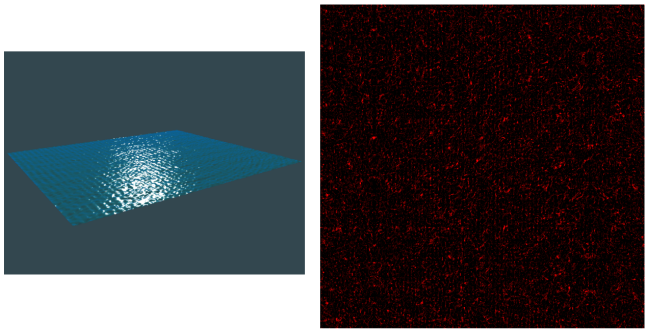

Figure 7: Example water surface and the resulting caustic map generated and stored in the red channel

3.4 Caustic post-processing

After having generated the caustic map, an intermediate step serving as a solution for artifact removal is to introduce some blurring to the caustic texture. For our implementation, a simple box blur is used to generate the blur effects. In order to prepare for this step, another G-buffer has to be created that holds a 2D texture with a single channel (GL_RED) containing the light intensity values. Just as before, a rotation, translation, and rotation has to be applied to reorient the plane for 2D rendering in the x and y plane. The caustic map is then passed in as a texture to the fragment shader and depending on the desired blur intensity, the surrounding texels of the current texel are obtained using an offset value. These surrounding texels' red channel values are added together to get a cumulative result. To get the blur result the cumulative result is then divided by the amount of texels traversed to get an average result, thus producing a blur effect at the current texel/texture coordinate. The approach itself is pretty simple, but still produces a decent blurring effect[3].

3.5 Ground Rendering

Once the caustic map has been calculated, with the post processing blur applied if needed, the next step is to finally apply this texture map to the actual ground floor along with the default texture for the floor. This involves sending in a default ground texture into the fragment shader, as well as the processed caustic texture. Now the final step is to generate a color value to output from the fragment shader:

$$float\ caustic = texture(gCausticBlurred, texCoord).r$$
$$vec3\ waterColor = vec3(0., 112.5/255., 1.)$$
$$float\ attenuation = 2$$
$$attenuation /= 0.1 * groundOffset^2 + 0.1 * groundOffset + 1$$
$$vec3\ Col = texture(groundTex, texCoord).rgb;$$
$$Col += Col * vec3(caustic)$$
$$Col* = min(attenuation, 1)$$
$$Col += waterColor * 0.25$$

Depending on how the user wants to control the final look of the caustics and ground floor, this can be modified in the ground floor fragment shader. A custom attenuation value is calculated to control how fast the distance between the water surface and ground floor affects the lighting of the ground floor instead of using the physically accurate $\frac{1}{x^2}$ for distance falloff calculations. Our custom attenuation function slows down the effect of the distance fallout drastically, as seen in Figure(8).

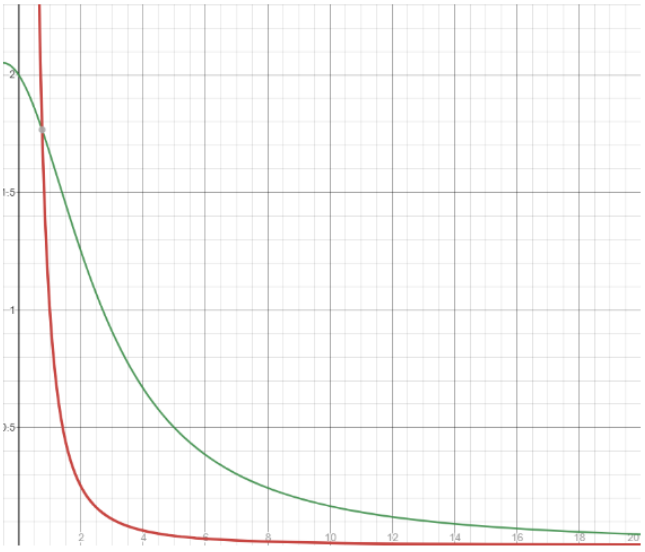

Figure 8: Graph of custom attenuation used in implementation (green) along with standard inverse squared falloff (red)

By combining the ground floor RGB value with the ground floor multiplied by the caustic value, a blended caustic result can be produced. After combining the RGB values, the attenuation function can be used to introduce some lighting falloff. Adjustments can be made here to increase or decrease the clamping applied to how bright the ground floor can get. The last step is to add water color contributions to the final RGB color, as the water surface above will affect the base color of the ground floor below. Values for this total RGB calculation can be modified and played around with to generate different lighting results.

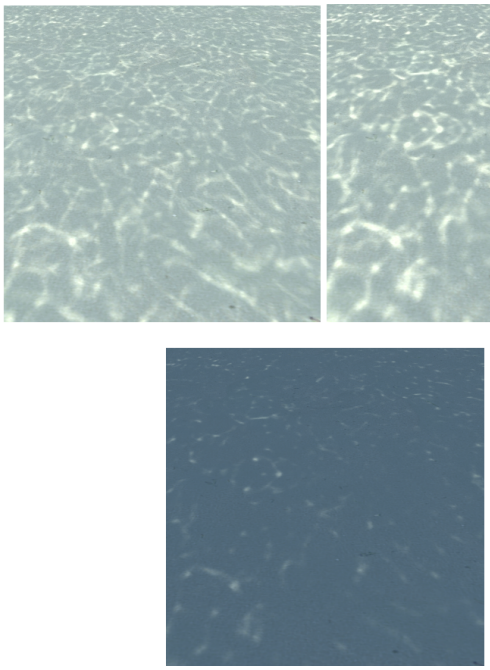

Figure 9: Caustic patterns rendered on the ground plane with a sand texture applied. The ground plane is 0.4, 1 and 5 units below the water for the top left, top right and bottom images respectively. Uses wave preset 1.

3.6 Water Rendering

While rendering real-time caustics is the prime goal behind our approach, some thought was still put in to how the deformed surface can look more like proper ocean water rather than a solid blue color. Phong shading is used as a baseline for our implementation and additional tweaks are added afterward to arrive at output that better resembles water. The main components that are considered when improving the water shading include color variation, transparency and high specularity. For the diffuse component of the water, we wanted our implementation to introduce color variation between blue and green color profiles based on how the light hits the surface. In a similar vein, transparency of the water should also change based on how the light interacts with the water. In order to add variation to both, we use a dot product with the normal and light direction, $NdotL$. $NdotL$ is added to the final alpha result and instead of directly changing the color values along the surface, the incoming light color's green channel is weighted by $NdotL$. Additionally, a better color palette on the surface was achieved by changing the weighting from the direct dot product value to the following quadratic function:

$$2NdotL - NdotL^2$$

Specular through phong shading sets most of the groundwork ahead of time, though an issue arises where shiny areas in the render bleed into the diffuse color too much to look like water. Since we want very bright specularity to achieve a glistening effect, we add an extra step to the specular calculation. After calculating

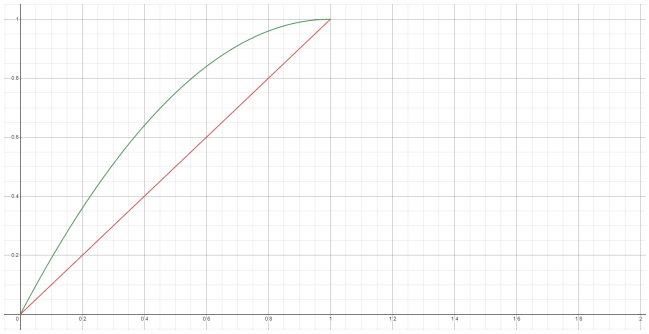

Figure 10: Weighting *added* to the alpha (red) and the weighting *multiplied* with the light color's green channel (green).

the specular through traditional phong shading, we check if the value of the specular is above a specified threshold. If so, then we replace the specular value with full specularity.

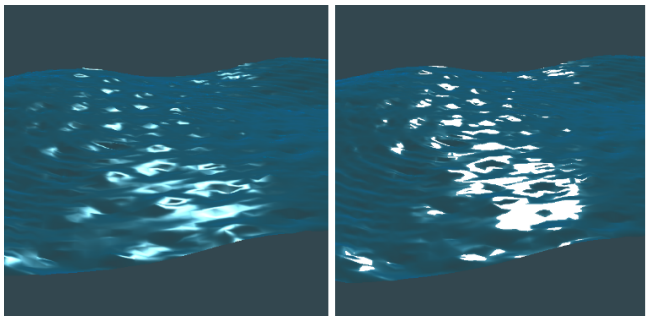

Figure 11: Phong specular (left) and updated specular (right).

Taking all of these modifications into account, below is the process used in order to tweak the phong shading into results that resemble water:

$vec3\ lightPos = vec3(2.5, 5, -10)$
$//\ diffuse$
$NdotL = dot(normal, lightDir)$
$r = NdotL * lightColor.r$
$g = clamp(2NdotL - NdotL^2, 0, 1) * lightColor.g$
$b = NdotL * lightColor.b$
$vec3\ diffuse = vec3(r, g, b)$

$//\ specular$
$...$
$if\ (average(specular.rgb) > threshold)$
$specular = vec3(1.0)$
$endif$

$//\ finalize$
$vec3\ color = (ambient + diffuse) * waterColor + specular$
$float\ alpha = NdotL + specular$
$FragColor = ve4(color, alpha)$

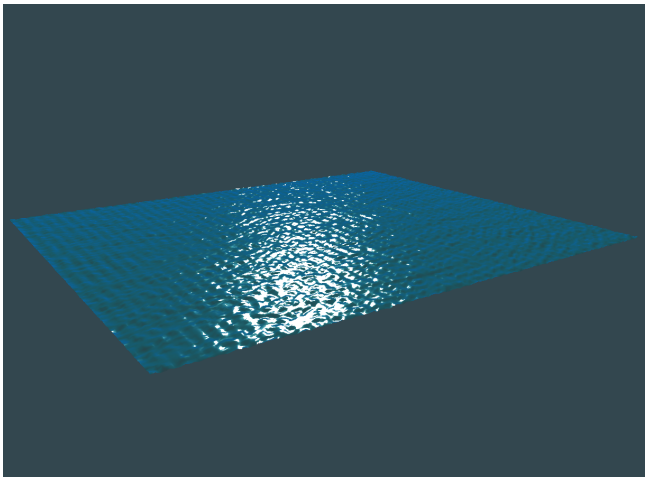

Figure 12: Water shading (using revised Phong shading) applied to surface with small ripples

4 IMPLEMENTATION

We implemented the caustics algorithm and overall environment using OpenGL. Three G-buffers are used to store information for caustic calculations:

(1) Normal texture with 3 color channels used, 16-bit float per component (GL_RGBA16F)
(2) Caustic texture with 1 color channel (GL_RED)
(3) Blurred Caustic texture with 1 color channel (GL_RED)

There are many tools available when it comes to texture configuration. To avoid harsh edges and pixelation of stored textures, smoother texture filtering methods are used. In the texture configuration, GL_LINEAR is used over GL_NEAREST for both magnifying and minifying filtering options. The resolution of these textures are currently configured to 1024x1024, which strikes a good balance between favourable results and performance. With a final render resolution of 1920x1080, an AMD Ryzen 7 2700 8 core processor running at 3.2GHz, and a RX 5700 XT GPU can achieve uncapped framerates of around 240 FPS on the 160x160 vertex plane. Using the same screen resolution, an Intel i7-10700k running at 3.80GHz with an RTX 3070 GPU can achieve uncapped framerates as high as 460 FPS using the 160x160 vertex plane.

5 LIMITATIONS AND FUTURE WORK

The current implementation of our algorithm generates decent caustic results, but the current mapping of this result doesn't take into account other objects that may be present in the scene such as rocks, seaweed or other underwater objects. In order to take in account such objects/meshes in the scene, these objects would also have to be included in the textures. A depth value would also have to be stored depending on how far away the object is from the surface of the water plane. For future work this is definitely something worth implementing and testing with the current algorithm.

Another limitation of our algorithm is that it depends a lot on the texture resolutions. If the resolution is too small, performance greatly increases, though the scene may appear to be randomly fluctuating blurry light intensities across the plane. With a resolution that is too big, the caustic result will look sharper and clearer but performance will be affected due to more pixels being traversed every iteration in the sampling algorithm. Getting the right number in between is important, in our implementation textures of resolution 1024 x 1024 were used, favoring performance while producing decent results. We were able to increase the texture size but upon increasing the resolution to around 4K, FPS considerably dropped on one system from 200+ FPS to about 50 FPS while using the 160x160 vertex plane.

As for the water shader, it provides satisfactory results from a demonstrative point of view, though choosing light sources higher up in the sky will begin showing the water shader's limitations. The blend between blue and green hues as well as the transparency of the water will not look as one would expect from water. From a prioritization standpoint, the water shader is implemented to look good enough when shown alongside the caustics. That is, it currently relies on hard-coded light positions in order to properly demonstrate colors, transparency and glistening specular highlights when viewing the water from an angle.

Another addition to the overall project would be to create a better enclosed underwater environment so that god rays can be added on already existing caustics. The addition of god rays would allow beams of light to pierce the water surface, providing the underwater section with more interaction with incoming rays of light. Adding this with the caustics would definitely provide a better look for a full-featured underwater environment.

6 CONCLUSION

In this paper we proposed an improved caustic algorithm that can produce convincing looking caustics while not being overly performance heavy. Using the wave normals, the algorithm reads an environment map to generate a cumulative amount of light intensities per-vertex and directly maps the resulting caustic texture to the underwater surface. The algorithm with the current 1024x1024 texture resolution achieves relatively high frame rates, but as the texture resolutions are increased to resolutions past 4k, performance will be affected significantly.

REFERENCES

[1] Daniel Sánchez-Crespo Dalmau. 2001. Inexpensive Underwater Caustics Using Cg. https://www.gamasutra.com/view/feature/2811/inexpensive_underwater_caustics_.php?print=1 Last accessed 20 April 2022.
[2] Mark Finch. 2004. Effective Water Simulation from Physical Models. In *GPU Gems*. Nvidia.
[3] OpenGl. 2002. SSAO. https://learnopengl.com/Advanced-Lighting/SSAO Last accessed 20 April 2022.
[4] Jos Stam. 1996. Random causitcs: natural textures and wave theory revisited. In *SIGGRAPH '96: ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96*. 150.
[5] Daniel Sánchez-Crespo. 2004. Rendering Water Caustics. In *GPU Gems*, Juan Guardado (Ed.). Nvidia.

7 RESULTS

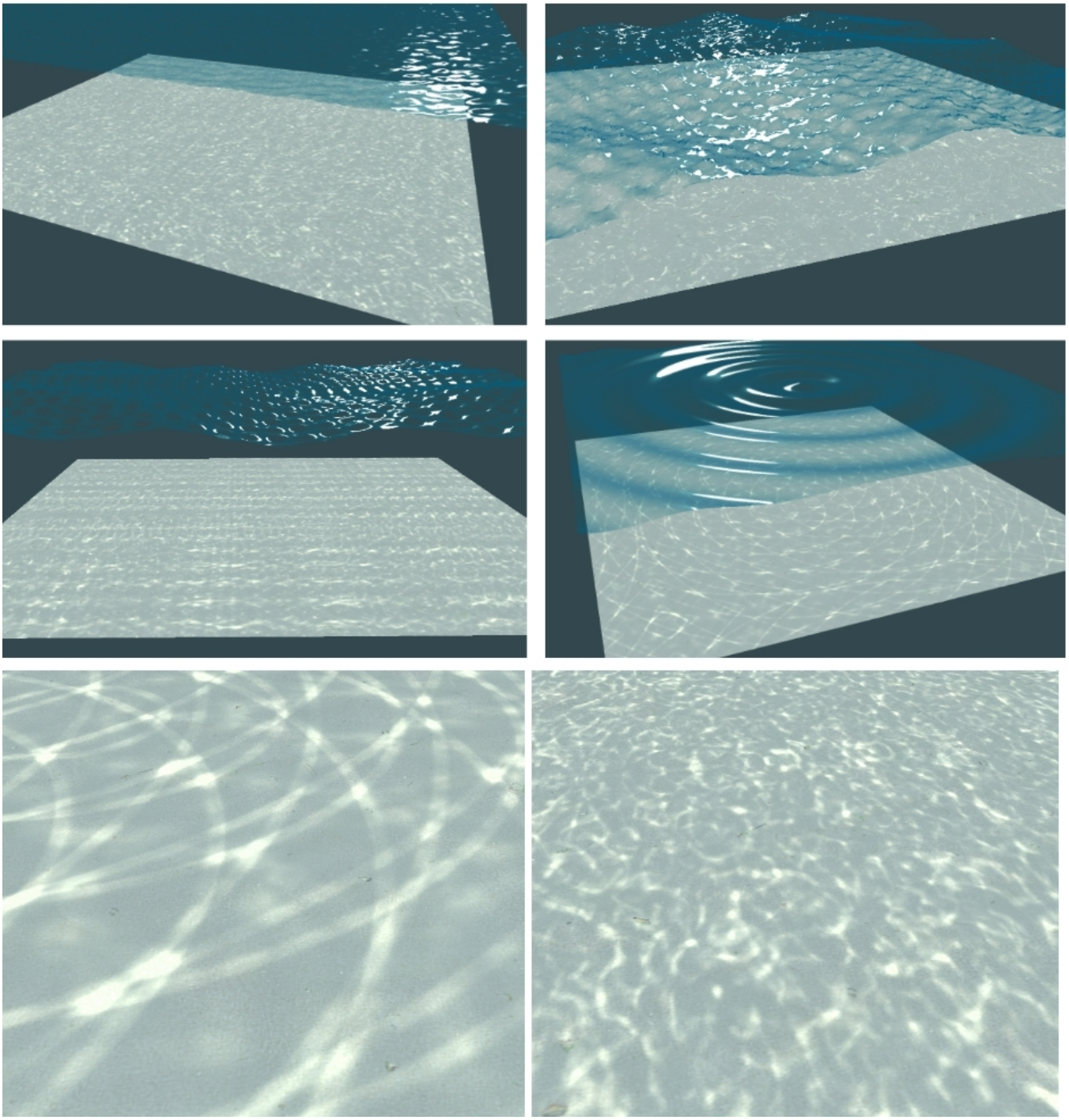

Figure 13: Caustic and wave results for each Wave Preset

Source code and the standalone version of the program can be found on GitHub: https://github.com/sebastiancollard/water-caustics