REAL TIME RENDERING OPTICAL EFFECTS OF WATER

Spring 2014

Abstract

This project attempts to render real-time the optical effects of the surface of water; such as reflection, refraction, and caustics. This is not a physical model, but only tries to render believable effects of them. The project was implemented using C++, OpenGL 4.4, and GLSL.

Aharon Turpie

Advised by Dr. David Mount

Phong Lighting Model

The Phong lighting model is a simple yet very convincing description of how lights illuminate objects. The Phong model says there are four types of lighting for every object.

1) Specular Lighting: Light rays from the light source hit object surfaces and reflect off. Because objects are not completely smooth, the specular reflections will vary a little, making the reflection on the object blurred and smoothed out. Specular lighting gives objects a shiny look.
2) Diffuse Lighting: Some of the light will pass under the surface of the material, bounce around underneath, and eventually emit out back out of the material in a fairly random direction. This means that diffuse lighting does not depend on the location of the viewer. Diffuse lighting gives objects a matte appearance. A portion of the lighting will never exit the object after going subsurface. This proportion is called the albedo.
3) Ambient Lighting: Light will reflect off objects and hit another object and reflect again and so forth. This lighting is very complicated and hard to model, but the result is that parts of the scene will be illuminated, even on parts occluded by the sun. To give this effect, a constant amount of illumination is added to everything as a "fudge factor". If it were not for ambient lighting, objects would look pitch black on their backs, which is not a very convincing effect.
4) Emissive Lighting: Some objects glow on their own. These objects have additional lighting added to them to make them brighter. Note that they do not illuminate any other objects nearby, as they are not a true light source. There is an effect which will make the glow effect nicer by blurring the glowing lights with post processing, but that technique will not be discussed here.

Every object material will have some balance of the different types of lighting. Some are shiny, some matte, and some are something in between. It is important to understand the physics behind these effects. Water will have plenty of specular reflections as water is a very shiny substance. Light that travels sub-surface will very rarely reflect back out because water is a very transparent liquid. The only light that will be scattered or absorbed is from light being scattered by particles in the water. The murkier the water, the more the scattering. This effect is not handled by the Phong model because the model does not take into account the depth of water. This is a problem that will be covered later in this paper.

Another major drawback of the Phong model is the lack of global lighting effects. It only renders a single object at a time, so interactions between objects do not occur. There are four major global lighting effects that the Phong model does not handle that are the focus of this paper; reflections, refractions, shadows, and caustics. The advantage of the Phong model is that it is a very light algorithm and requires very little work to calculate.

Model the water surface with a Sum of Polysines

The focus of this project was not to provide a realistic physical model of fluid dynamics, but it still be able to give the appearance of water movement. The moving waves was thus described with a sum of sines. The water had some number of circular waves, and each wave had an associated amplitude, speed, and length. The following is an equation of each wave function i:

$$W_i(x,y,t) = A_i \cdot v \sin\left(\left(D_i \cdot (x,y)\right) \cdot\ L_i + t \cdot S_i\right)$$

t = time
A_i = Amplitude
D_i = Direction in a 2D vector
L_i = Wavelength
S_i = Wave speed

Since the waves are circular, the direction is the direction vector from the wave's center to the position (x, y). Circular waves are generally good at modeling small bodies of water, such as a pond, where there is a source of the ripples, such as a waterfall. In ocean water, the main driving force is the wind, which produces line waves. For these ocean waves, just make the wave direction constant for the wave, and not depend on a center.

As an added level, we want to phase the different waves in and out. This makes some wave more and less apparent over time. The purpose is to try to reduce the amount of regularity of the wave patterns. The updated equation is below.

$$W_i(x,y,t) = A_i \cdot \sin\left(\left(D_i \cdot (x,y)\right) \cdot\ L_i + t \cdot S_i\right) \cdot \frac{\cos\left(\psi_i \cdot t + \phi_i\right) + 3}{4}$$

ψ_i=Phase speed
ϕ_i = Phase offset

Water in reality does not follow perfect sine patterns. There are many interactions with the water and with the air that cause the waves to become a bit more peaked. To achieve this effect, raising the sine expression to some power will make it more peaked and the wave thinner. The higher the power, the thinner the peak.

$$W_i(x,y,t) = 2 \cdot A_i \cdot \left(\left(\frac{sin\left(\left(D_i \cdot (x,y)\right) \cdot\ L_i + t \cdot S_i\right) + 1}{2}\right)^k - \frac{1}{2}\right) \cdot \frac{\cos\left(\psi_i \cdot t + \phi_i\right) + 3}{4}$$

k = Peak Power Constant

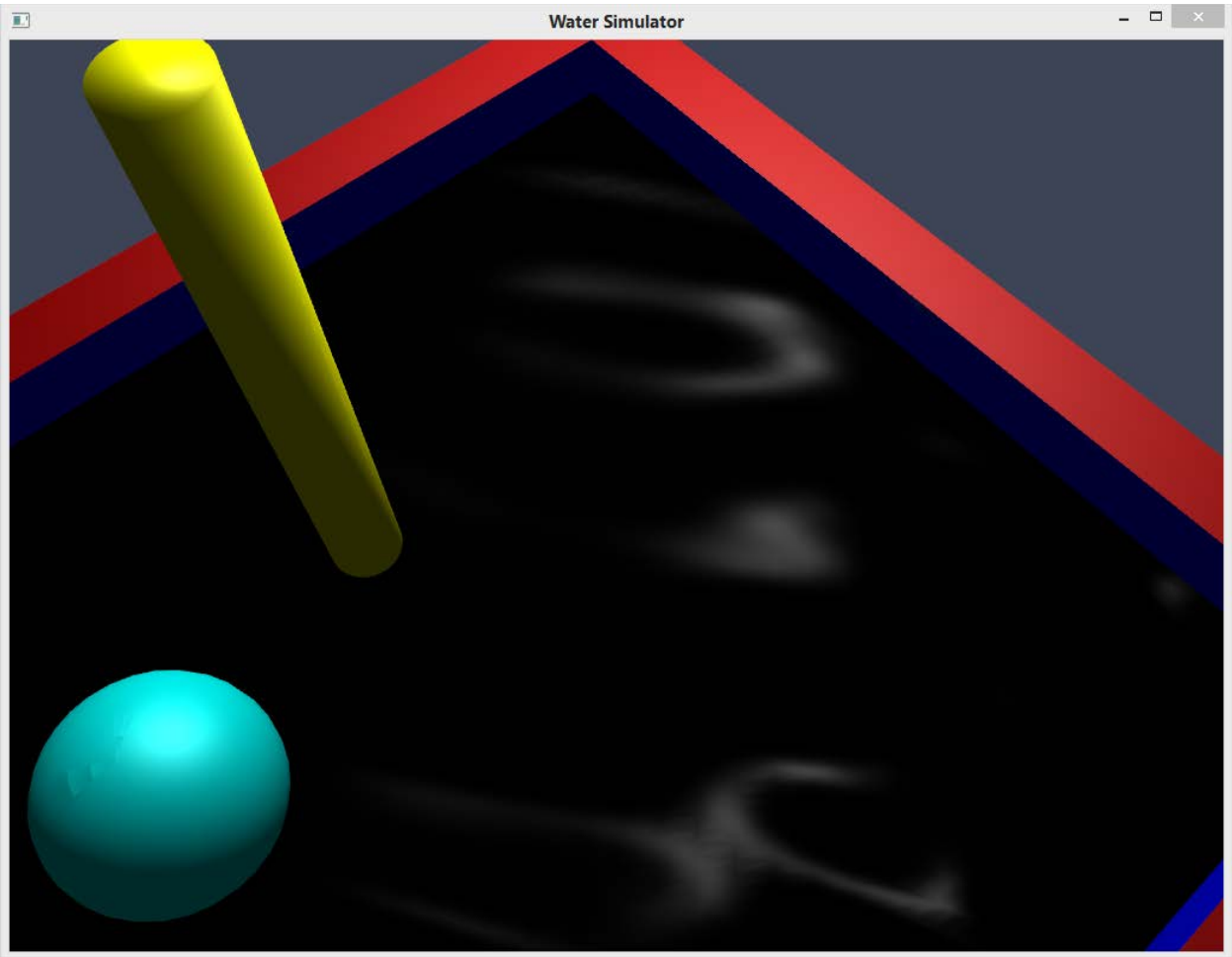
The derivative gradient of the wave is important for calculating the normal of the wave surface, and is as follows.

$$\nabla W_i(x,y,t) = A_i \cdot k \cdot D_i \cdot L_i \cdot \left(\frac{sin\left(\left(D_i \cdot (x,y)\right) \cdot\ L_i + t \cdot S_i\right) + 1}{2}\right)^{k-1} \cdot \cos\left(\left(D_i \cdot (x,y)\right) \cdot L_i + t \cdot S_i\right) \cdot \frac{\cos\left(\psi_i \cdot t + \phi_i\right) + 3}{4}$$

Displacement Mapping using the Tessellation Shader

To model a detailed scene, there would need to be millions of vertices for every object to accurately detail every bump and crevice. This is a huge expense in memory on the GPU, and is thoroughly avoided. To simplify the process, techniques such as normal mapping or bump mapping are commonly implemented to give the illusion that a low vertex count object is actually high resolution model. This is done by perturbing the normal on the surface to affect the lighting. It's an extremely simple approach and give surprisingly good effects.

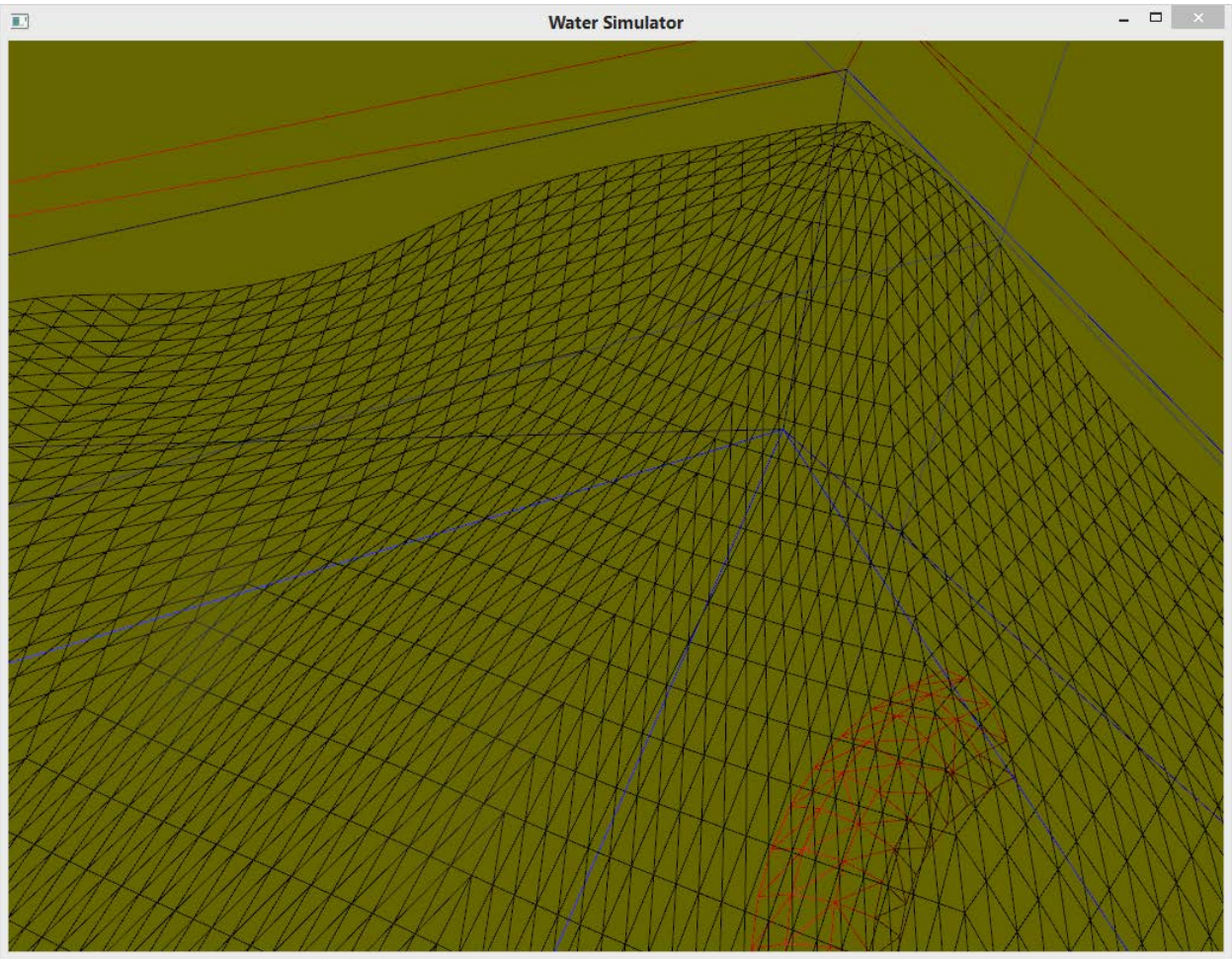
There are two major drawbacks of using a normal or bump mapping. First, the geometry of the object is still not actually a high resolution image and will appear flat around the edges. Second, the object does not occlude itself if parts of the bump should cover up a part of the object behind it.



This is the effect of normal mapping without displacement mapping. Notice the edges of the water are flat, as the water is still a quad.

To solve this issue, a process called displacement mapping is used instead. The idea is to actually change the geometry so that it is indeed a high resolution object, but without storing all the vertex positions in memory. The displacement map stores all the heights of the surface, similar to a bump map. The geometry of the object is then moved up or down along the normals of the surface by the amount on the displacement map. Storing and accessing the heights from a texture is significantly faster than creating an enormous vertex buffer.

The fragment shader cannot change the geometry and changing the geometry in the vertex shader would not add any resolution to the model. That is why the tessellation shader is used. The tessellation shader takes geometry primitives as input and breaks them up into much smaller primitives. So in this case, the two large water triangles are broken up into hundreds of very small triangles. The new triangles' geometry are changed following the water model in the tessellation evaluation shader.



The tessellation shader breaks the water quad into many triangles, which are then moved up or down to fit the water surface model.

The main disadvantage of displacement mapping is that it uses significantly more processing on the GPU than the normal or bump mapping alternatives, but gives a much nicer appearance. Displacement mapping is very commonly used for terrain as normal and bump mapping do not handle large changes of height well.

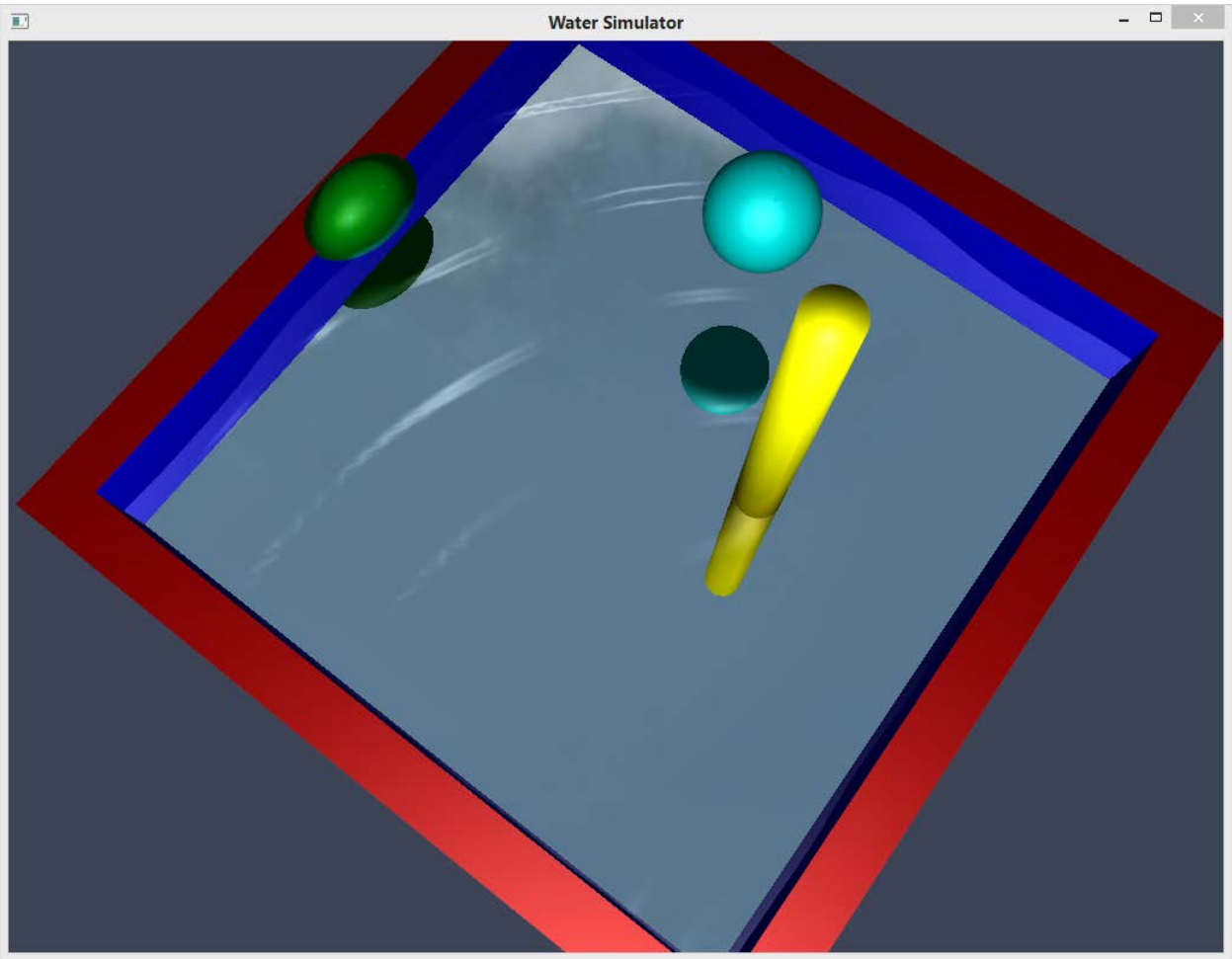
There is an alternate version of displacement mapping that uses the fragment shader instead of the tessellation shader. The idea is to perturb UV coordinates of a texture of the surface to give the illusion that the viewer is looking at a non-flat geometry. The key is to use ray tracing to identify where on the texture each fragment will appear, which in turn will have parts of the texture skipped over giving the effect of self-occlusion. The problem is that ray tracing is generally a difficult and process intensive task. There is a method to greatly speed up the ray tracing using a distance map, but that kind of technique only works if the geometry is pre-computed, as the distance map takes a long time to generate.

The advantage of using the fragment shader is that the GPU does not need to process a large amount of vertices. The disadvantage is that the method requires ray tracing, which is not very real time feasible unless the geometries are simple or static. Also, while it will have self-occlusion, the surface shape is still bounded by the original geometry.

Rendering Reflections and Refractions using Frame Buffers

It turns out that rendering reflections is fairly simple. Before rending the final scene, multiply the model matrix by a reflection matrix that will mirror the world about z = 0. This will move everything as if looking through a mirror except the ground and all the bottom of containers and such will occlude the view. To fix this, a clip plane is added to remove everything behind the “mirror”, or above z = 0. It is important to not draw the mirror in the reflection scene, or it will occlude the scene, and the actual mirror surface does not have any kind of color or texture before the reflection scene is generated.

To use this reflected rendered scene, a frame buffer is used to store the image. In OpenGL, it is possible to create multiple frame buffers, and change the rendering to draw to any of the frame buffers instead of main render frame buffer that outputs to the screen. The scene is recovered through a texture generated from the frame buffer. Frame buffers also store the depth and stencil buffers, which can also be obtained through textures if desired. When a mirror is rendered onto the final scene, the reflection texture generated from the scene rendered in the reflection frame buffer is mapped onto the mirror surface.



The reflection texture is generated and rendered onto the water. The water appears as a perfect mirror with no distortions with the exception of the shininess of the waves from specular lighting.

To get the refractions, render a second time with no reflection matrix on a second frame buffer. If the refraction texture is mapped onto the mirror, then exactly what is behind the mirror will appear on it, making the mirror act more akin to glass (or invisible if there is no specular lighting on the mirror surface).

There are two major draw backs to this method. First, to get the reflection and refraction textures, two extra passes over the scene is required, which may be costly. Second the water surface is not flat like a mirror. This means that parts of the water will be above or below the reflection and refraction plane. Because of this, the reflections and refractions are only an approximation, not an accurate description. The larger the difference between the water and the reflection plane, the larger the error. Luckily, reflections and refraction are complicated, so people will not be able to notice this discrepancy that well, and the effect is still quite good.

Using Du/Dv Maps for Refractive effects

When light travels from one substance to another, the light becomes bent. This is called refraction. The different frequencies of light will all be refracted by different amounts, separating the light colors, and creating a rainbow effect. This project does not consider the color separation aspect of refraction as it is not very prevalent from water surface refraction.

The amount by which light will bend as it goes from air to water depends on the change in the index of refraction of the two substances. Snell's Law could be used to determine the refracted direction.

$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \frac{n_2}{n_1}$$

Unfortunately, determining the color at a pixel through the use of the refracted direction from the water surface is a ray tracing problem. As mentioned earlier, ray tracing is generally a difficult problem to handle in real time rendering unless the geometry is simple or static.

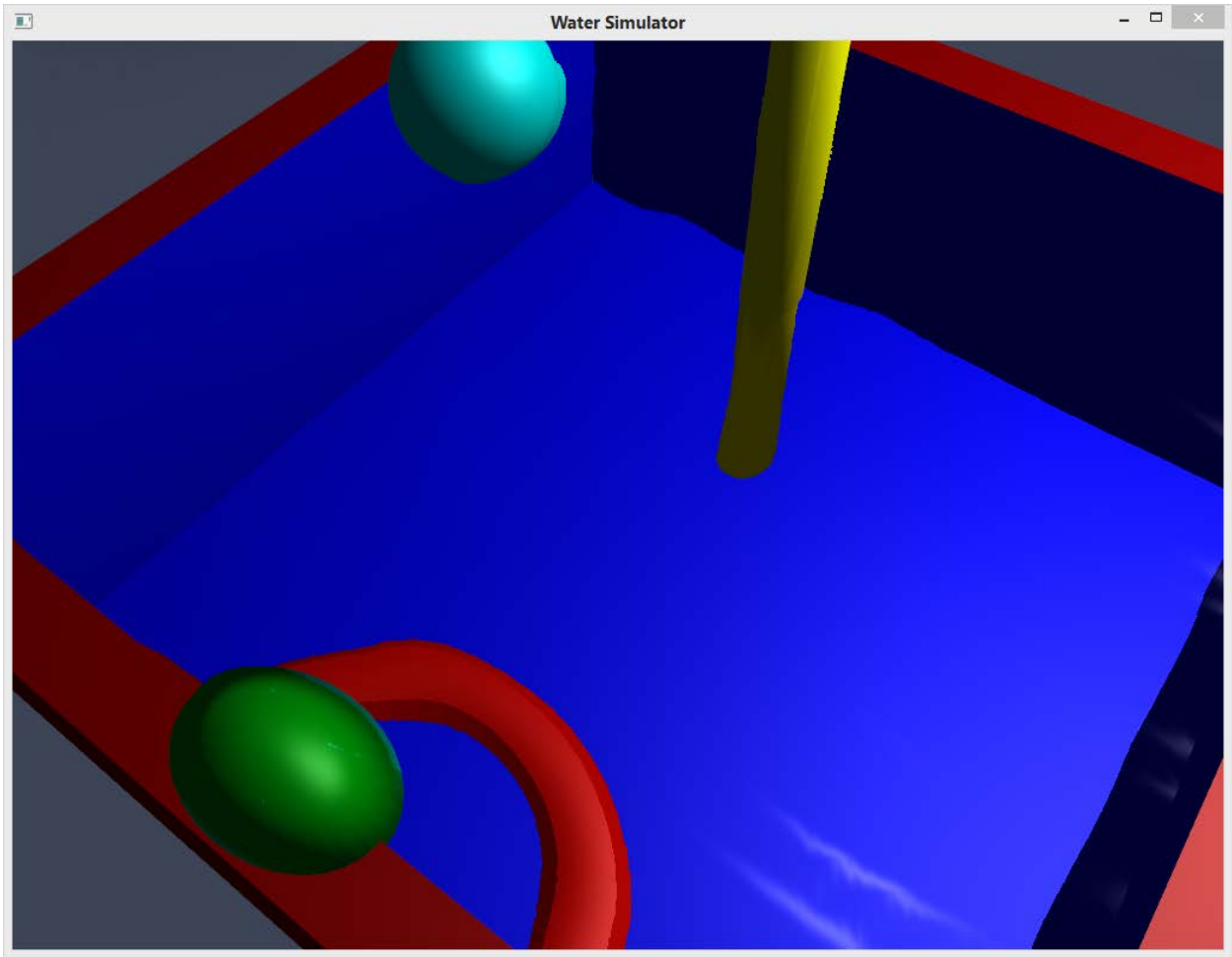
Instead, a more simple approach to give the appearance of something similar to refraction is used. The idea is to draw the color of pixel on the refraction texture (generated above), that is in the direction of the derivative of the normal of the surface. This does not take into account the index of refractions or Snell's Law at all. All the pixels on the surface of the water are perturbed, which will make objects below the water wobble in a way one would expect from refraction.

To determine the derivative of the normals, a Du/Dv map could be used. In cases where the water is modeled by a pre-generated texture, a texture with the derivatives could also be produced offline in a Du/Dv map. In this project, the water was modeled real-time using an analytical description, so the Du/Dv map can easily be determined using the gradient ∇W.

There are a few major draw backs. Calculating the Du/Dv map isn't that slow, but it does take up additional time. Also as mentioned with the reflections, generating the refraction texture requires an extra rendering of the scene. This is clearly not a physical model, and thus might not give an accurate look, but it will likely look good enough to give the impression of refractive water.

Another major issue is that objects occluding other objects in or above the water will likely "bleed" their colors in the refraction nearby. When the texture is perturbed near an occluding object, it will likely draw pixels from the object in front, instead of the object behind in a physical model. This is because the refraction texture only stores what is in front, and the information of what was behind is lost. A possible solution is to render the refraction texture and

water refractions one object at a time, and try to composite the images. This would require a lot more additional work, and might not give good results for the additional performance.



Refraction texture is perturbed giving a wobbling effect of refraction from water. Notice the color bleeding around the sphere, even though it is completely above the water.

As the camera gets further away from the water, the amount of space each pixel takes in the scene become larger. This means that the refractive "power" becomes stronger as the camera moves away. In order to alleviate this problem, the refractive perturbation was scaled by a factor of the depth. This became a major issue to try to handle, as the depths stored are not stored linearly. Depths are stored from 0 to 1, but almost all the values will be nearly 1, as there is much more depth resolution very close to the camera. This means that is very hard to distinguish depth as they get farther away. Raising the depths to some power to try to redistribute the values help deal with this, but it is no perfect solution.

The refractions also do not take into account how deep the water is. This is mainly taken into account by the nature of perspective geometry, but near the surface where the water depth is very shallow, the refraction is stronger than it should be. It might be possible to reduce this by scaling the amount of the perturbed texture by some factor of the depth (which is talked about below with water murkiness), but that was not covered in this project.

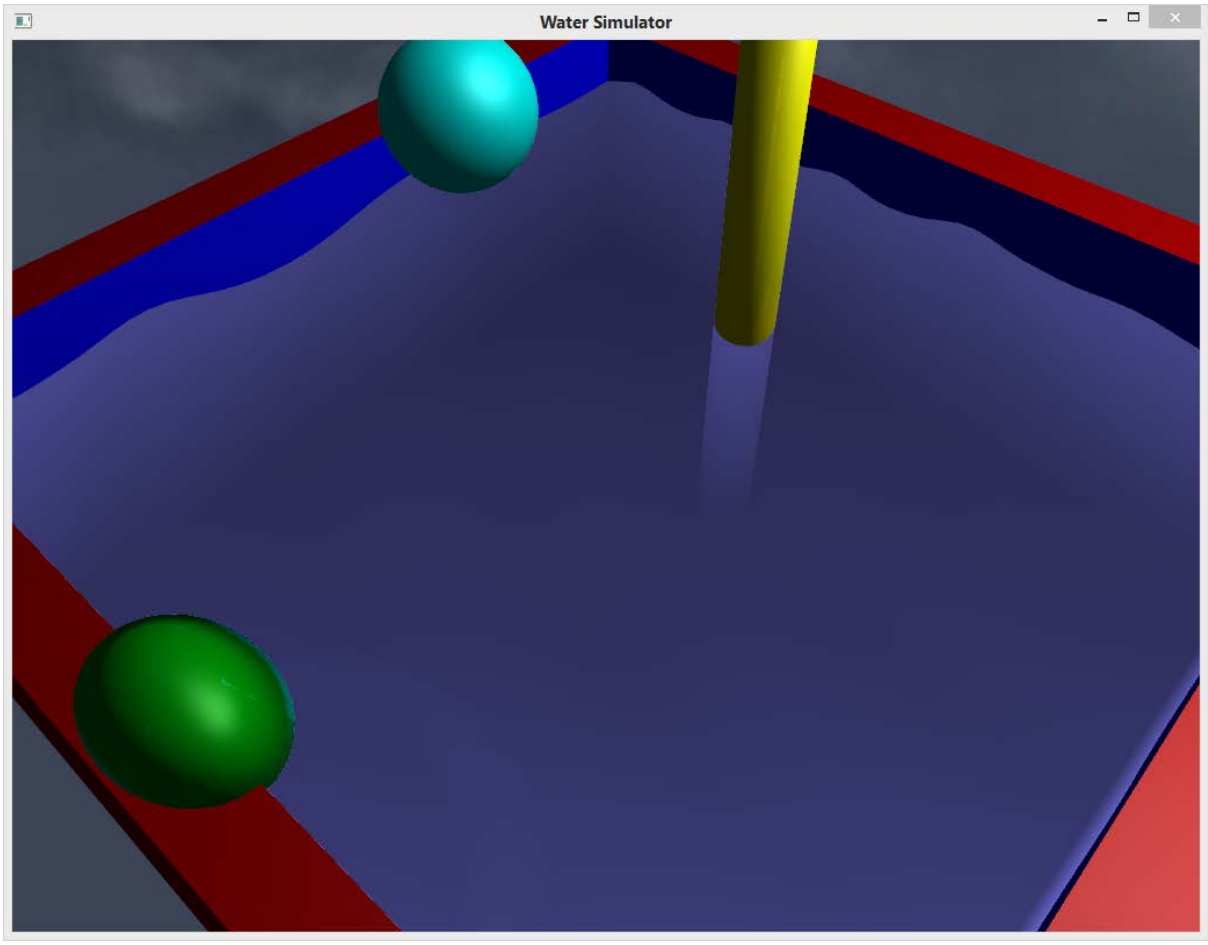
Fresnel Factor

When light goes from one substance to another, a portion of the light is reflected and some portion passes through refracted. Fresnel factor, also known as Fresnel equations, can be used to determine this. The equations are quite complicated and take into account polarized lights, unpolarized lights, index of refraction, and light wavelengths.

A much more simplified approximation for the Fresnel factor is to take the dot product of the incident direction and the normal. This will have the Fresnel equal to 1 when looking perpendicular to the water, and equal to 0 when looking parallel to the water. The Fresnel factor will determine the amount of light will be refracted. The inverse Fresnel is 1 minus the Fresnel, and is used for the amount of reflection. Using the Fresnel factor, the reflection and refraction can be combined in appropriate proportions.

Using the Depth Buffer to simulate water murkiness and color

As light travels through water, a small fraction of it is absorbed and reflected by particles in the water. The more distance the light travels, the more of it is absorbed. This makes deep water much darker than shallow water. Also, dirtier water will have more absorption than clean water. The light that bounces off of the particles take in its color, which gives water color. Again, the more distance the light travels, the more color will be prominent, and shallow water will be nearly clear.



The effect of using depth buffers to give darker color to deep water.

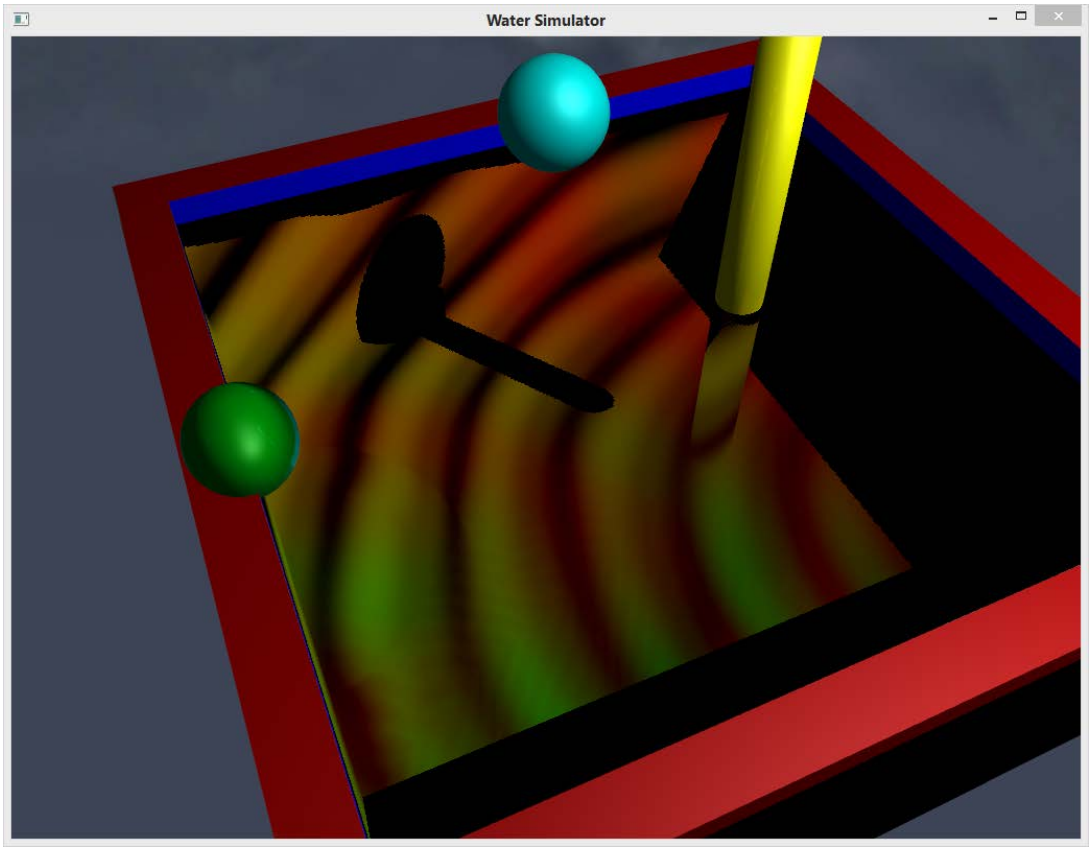
To give this effect, the refractions are multiplied by an inverse depth factor. Then the depth color is added to refractions. The depth color is the color of the water (which is normally some blue or green color) multiplied by the depth factor. This means that deeper water will have less refractions and more water color.

As with the depth correction with the refraction power mentioned earlier, calculating the meaningful depths is difficult. The idea here was to use two separate depth buffers to determine the water depth, one of which was the scene with water and the other was the scene without water. The difference between these two depths should be the depth of the water.

Unfortunately because of the way depths are stored, a simple subtraction is not possible. As before, raising the depths to some power to attempt to distribute the values more gave reasonable results. As the camera gets far away from the scene though, the difference between the two depth buffers is so small, since they store little resolution for objects far away. So the depth of the water becomes very uniform. This is not too noticeable, since it is hard for people to distinguish depths when the water is far away anyways.

Caustics Map

An interesting consequence of refraction is that as light is bent, light beams will often focus. When focused light hits an object under the water, it become brighter. When less light hits an object because all the lights are refracted away from it, it will becomes darker. This effect is called caustics. Caustics will can have many complicated patterns but they often have banding or circle patterns depending on the water surface.

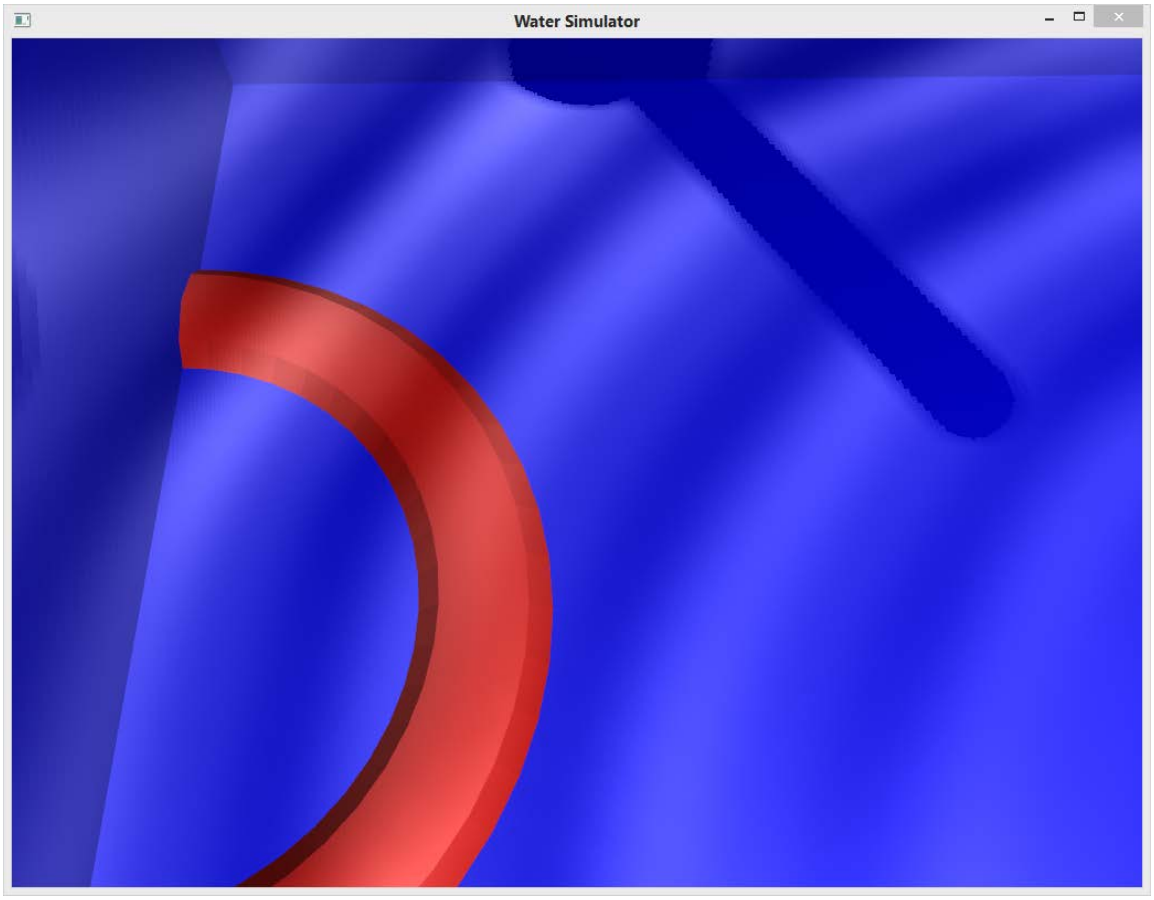


Generated caustic map. Colors are only in red and green, similar to a Du/Dv map. Also note a part of the map is black because the water is occluded by the sphere and part of the pole.

Unfortunately, similar to refraction, this is a problem that is natively solved using ray tracing. There are not many ways to create caustics without ray tracing. The best looking method is probably applying a pre-generated texture of what the caustics will appear with the given water surface texture and light position.

To avoid using a static environment, a different approach would have to be made. Using a frame buffer, the water is rendered again, but this time from the light's perspective. Instead of the normal colors being outputted, the refracted direction is used for the red and green channels. This is the caustic map. Along with the caustic map, the stencil buffer is also stored so that colors will only be read from places that have water. *(Image above)*

During the rendering of the scene, points are projected with the light's perspective, and that location is used to access the caustic map. This is idea was inspired by shadow mapping, and is essentially a way to do ray tracing without any kind of real ray tracing. Applying a projection matrix to the light's view on points will determine where points will appear on the caustic map. If the refracted direction in the caustic map is the same as the direction to the point, then the point will be illuminated. To give a smooth transition, this is applied with a Gaussian.

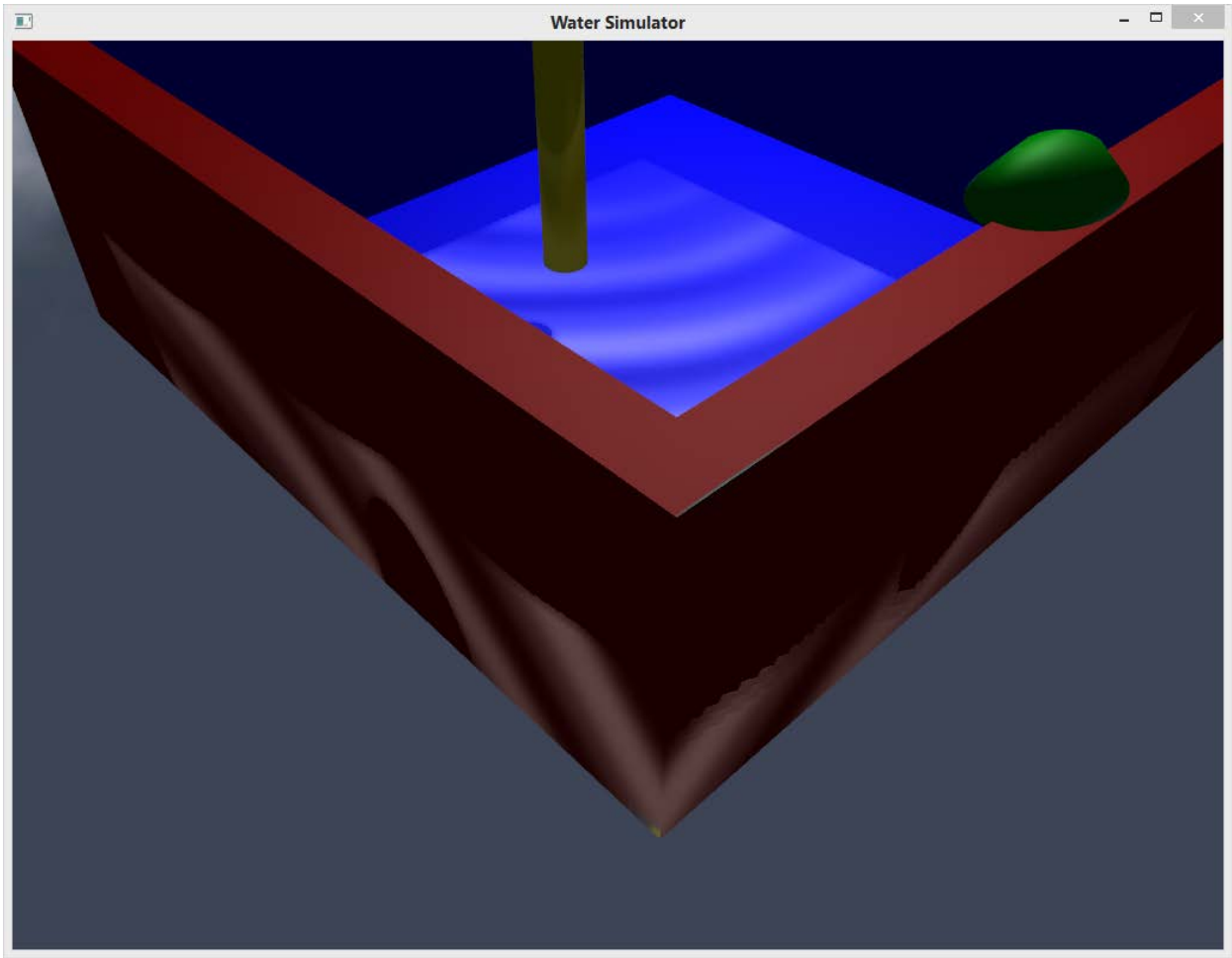


Caustics with sampling from the caustic map.

While the caustics do seem to have some of the effects expected, it is extremely pixelated. Since caustics are not just affected by the one beam going directly through the water, nearby locations on the caustic map are also sampled. This should now be able to produce caustics where light rays are refracted and lights up a spot close to incident rays. Using the sampling, the caustics now have a much nicer smooth appearance.

There are a couple major issues. The first is that near the edge of the water, and along edges of objects occluding the water from the light's perspective, there are a lot of blocky artifacts. This is due to the resolution of the caustic map. A higher resolution caustic map will reduce the amount of artifacts at the cost of more memory.

The second major issue is that the projection onto the caustic map does not take into account occlusion. This has the effect that the back side of the water floor and objects completely occluded will still have the caustic lighting on them. To fix this, a shadow map is applied so that the caustics are only used in places where they are not occluded from the light.

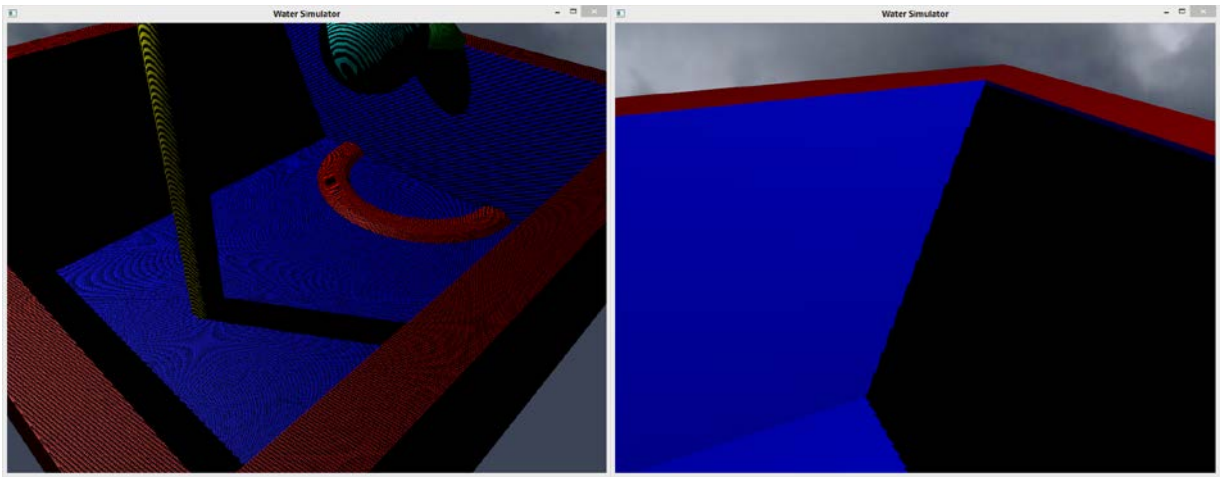


Problems of using a caustic map without detecting occlusions with a shadow map

Shadow Map

The concept of shadow maps is similar to caustic maps. The scene is once again rendered, this time without the water. The key here is the depth buffer, which will be the depth map. When the scene is rendered, the points are projected to the shadow map (the same way as the projection to the caustic map). If the depth of the point is greater than the depth stored in the shadow map, then that point must be occluded. Once again, the issue with resolution of the depth buffer arises, so instead of using a projective projection, an orthonormal projection is used instead. This way it is easier to compare depth values because they are more evenly distributed.

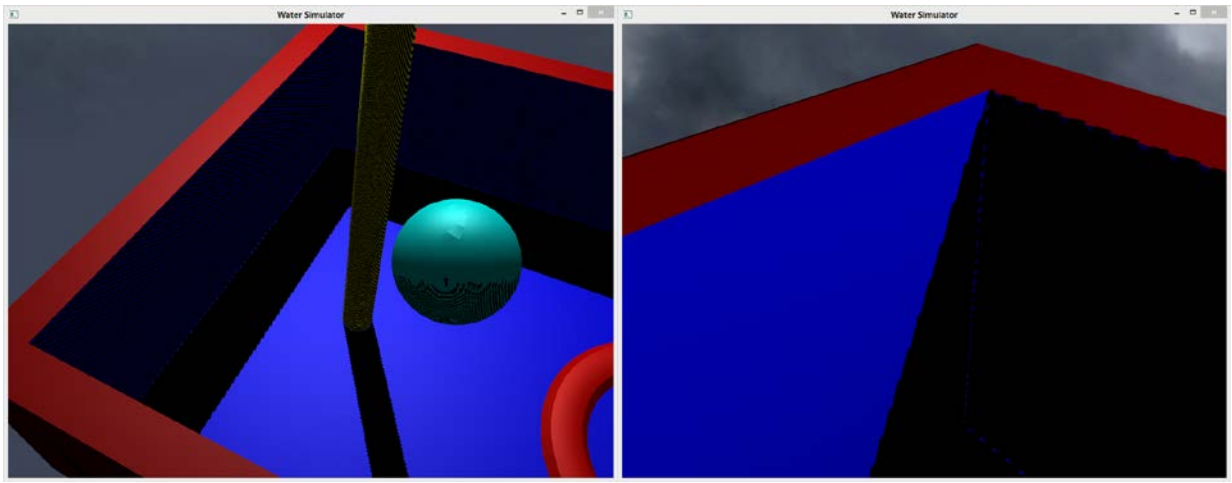
Doing this will result in a lot of artifacts in the illuminated areas, called acne. This is because the shadow map has low resolution, and parts of the depths in the shadow map will be above or below the actual depths. To alleviate this, a small error bias is added to the depths, to remove this.



Shadow mapping's acne effect (left) can be removed using a bias (right), but there is a gap in the shadow at the top, making it look like it is floating.

Unfortunately this adds an effect called "peter panning". The corners will be illuminated, even when they are completely in shadow. To help with this, if the face-culling process is changed to render the back faces instead of the front faces, then there is no resolution issue for the illuminated sections, removing the need for a bias and removing the peter panning effect.

Instead, this results in acne on the backs of objects. Once again, a bias is added to remove the acne. The peter panning effect is revered, so that shadows will bleed around corners a little. This is still a better look than when there was peter panning.



Using the back faces for the shadow map moves the acne to the shadow (left), which can again be removed with a bias (right), but the shadow bleeds around the lip at the top.

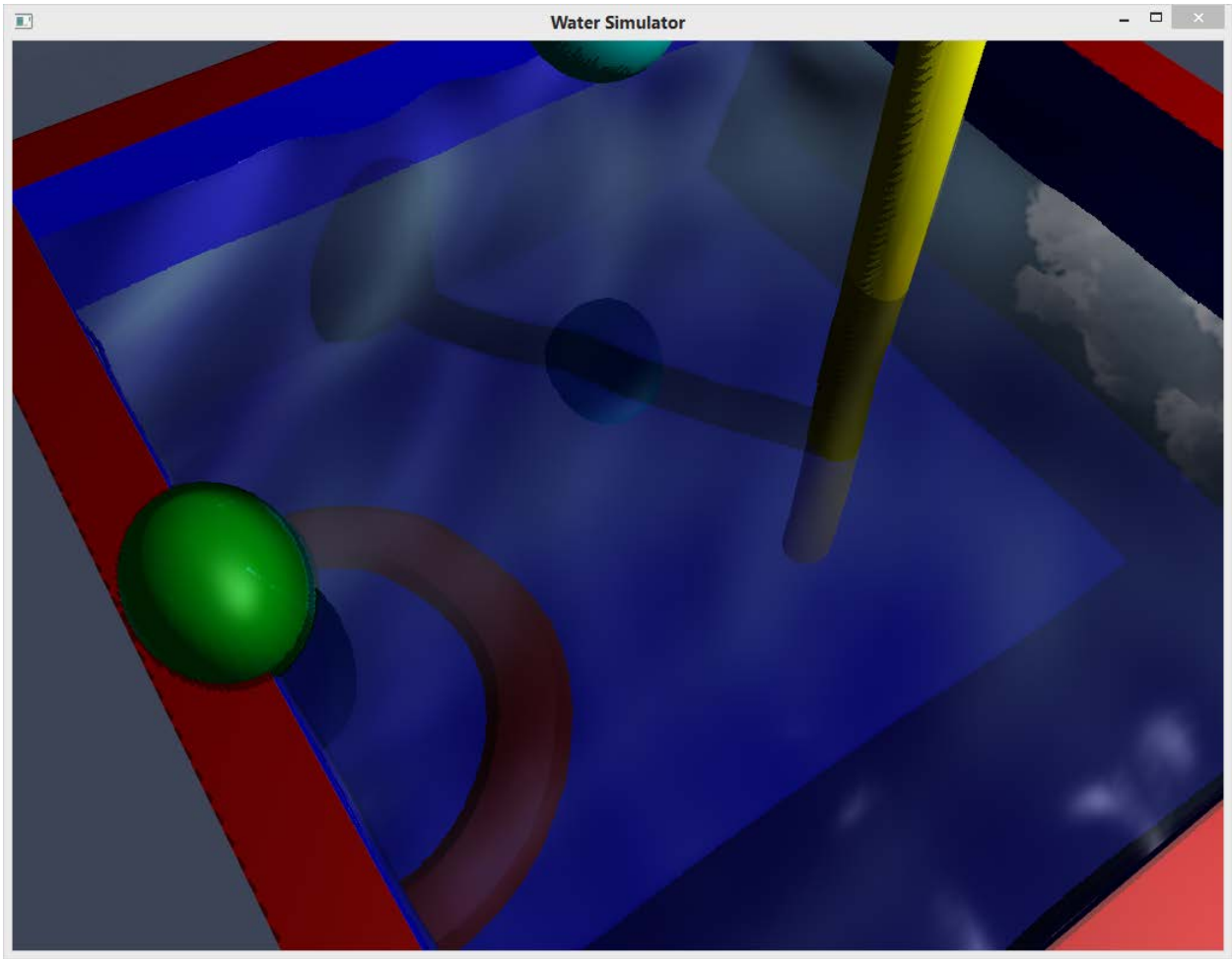
It is not very clear if the back face method of the shadow map turns out much better or worse. Both versions have a lot of artifacts which cannot be easily solved without overcoming

problems with the resolution. This resolution issue is present in both the shadow map and the caustic map. A simple fix would be to take a higher resolution texture, but as before, this is at the cost of additional memory required. Alternatively, the shadow map can be multi-sampled to try to blur the shadow edges a bit, making the edges of shadows a little less blocky.

Demonstration Video

A demonstration video is provided to see the rendering in action. It walks through the process, showing the different effects of the various techniques used in the project.

http://youtu.be/Ay2_YuPufmw



Final result of the techniques used.

Sources

Graham Sellers, Richard S. Wright, Jr, Nicholas Haemel, OpenGL SuperBible Sixth Edition, Addison Wesley (2014) http://www.openglsuperbible.com/

Sam Hocevar, "Tutorial for Modern OpenGL (3.3+)" (2012) http://www.opengl-tutorial.org/

"OpenGL Water Tutorial", Bonzai Software (2005) http://blog.bonzaisoftware.com/tnp/gl-water-tutorial/

Mark Finch, *Cyan Worlds*, GPU Gems, NVidia Corporation (2004) http://http.developer.nvidia.com/GPUGems/gpugems_ch01.html

Ben "DigiBen" Humphrey, "Realistic Water Using Bump Mapping and Refraction", Game Tutorials (2005) http://hydrogen2014imac.files.wordpress.com/2013/02/realisticwater.pdf