CS562 – Final Project: Rendering Water as a Post-Process Effect

PROBLEM DESCRIPTION:

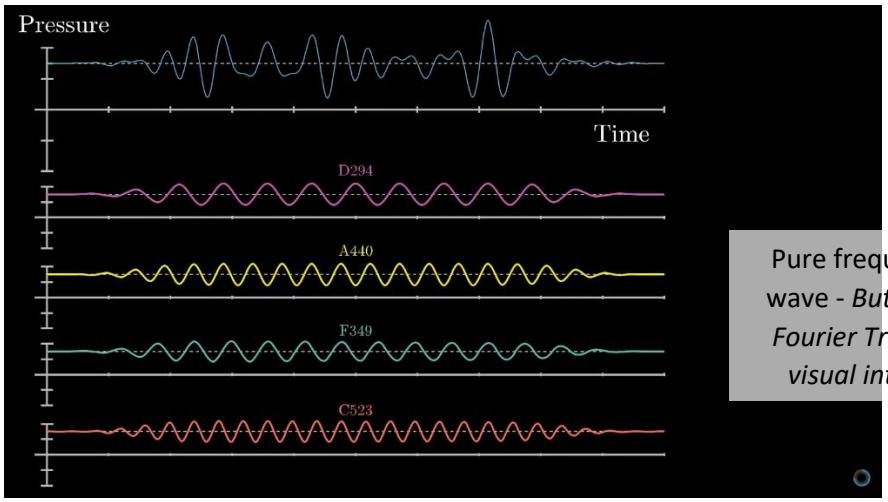
The objective of this project is to render realistic looking water. To do so, the paper suggests to render it as a post-processing effect and to make use of height maps to approximate the waves of the water.

PREVIOUS PAPER IMPLEMENTATION:

The first paper that I selected was 'Using Vertex Texture Displacement for Realistic Water Rendering' by Yuri Kryachko. In contrast with the actual implemented one, this paper suggests modifying geometry of a plane to mimic realistic waves.

Waves

To obtain realistic water, complex waves need to be mimicked. These waves can be decomposed onto waves of different frequencies. In fact, there are waves of pure frequencies (also called pure tones in psychoacoustics) which are sinusoidal waveforms. So, the idea is to define some waves and add them to obtain this complex waves. This could be made using Fast Fourier Transform (FFT), but as it could be heavy computationally, the paper suggests to approach it using height maps.



Pure frequencies of a wave - *But what is the Fourier Transform? A visual introduction*

Height Maps

Height maps are grayscale textures that in this case, store heights of the waves. So, to mimic the addition of waves of different frequencies, height maps of different frequencies are going to be added instead. These will not give results as realistic as using fluid dynamics or FFT, but for real time applications is more than enough. Height maps can be done by an artist or with a perlin noise. In the second option, textures can be accessed multiple times, each with different texture coordinate sizes to sample them in different frequencies.

The goal is to obtain animated water. That is why, three parameters are necessary to access height maps: position in the x and y axes and time. With the time, texture coordinates can be modified over time to get moving waves.

$$Height = \frac{1}{N}\sum_{i=1}^{N} h(x_i, y_{i,} t)$$

Modifying Geometry

Once the height map is computed the paper suggests to modify geometry in vertex shader to obtain waves. The mesh to use is a little special though. For the sake of having Level of Detail (LOD), the author explains how to use radial meshes, which store polar coordinates of each point and the radius of the circumferences. This radius gets higher while the vertices get farther from the camera. In this way, only by locating the mesh at the center of the camera, a simple LOD is obtained.

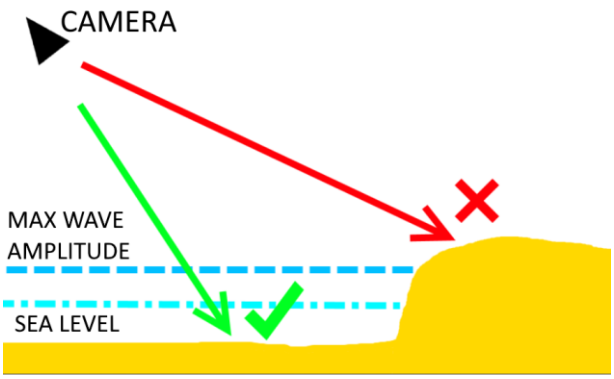
ACTUAL SELECTED PAPER IMPLEMENTATION:

While implementing the first paper, I come across with the actual paper I implemented 'Rendering Water as a Post-Process Effect' by Myopic Rhino. I found it more interesting and complete than the previous paper. So, although it does not modify the geometry itself to get actual waves, it explains how to implement optics, which is essential to get good looking water. So, I stick with this second approach, but maintaining the computation of the height map as a previous render pass. Note that all the computations will take place on the fragment shader and in world space.

Obtaining geometry

As it is a post processing effect, geometry needs to be constructed from scratch using the depth buffer from the GBuffer and the height map. The idea is to approximate the waves iteratively by tracing rays.

Water does not affect the fragments above the max wave amplitude, so they need to be discarded. The result will only be between the sea level and the given max amplitude.



Valid and invalid fragments

For getting the position out of the height map, first an intersection between the vector from view to the fragment and the sea level needs to be computed. Using the x and z component of the computed surface position, the height map can be sampled. Get the next surface level by adding the current one and the height given by the height map. Take into account that the height map is between 0 and 1, so the height value needs to be multiplied by the max wave amplitude before the addition. Then, repeat the process N times, each time intersecting the vector with the new surface level. As it will be repeated several times, it could happen that the wave gets above the wave amplitude. To prevent that, divide the height to be added by N.

Normal

Once that the surface position on the wave is computed, to obtain good results in the optics and on lighting, normal needs to be computed. To do so, the information of the neighborhood pixels is needed. Sample the points close to the actual texture coordinates in the x and in the y axes. These values describe how the normal changes in the x and z components, so by subtracting them, the actual components are obtained. For the y component, a scaling factor is used to obtain smoother or harder normal. It is a user defined parameter.

$$Normal = vec3(West - East, NormalScale, South - North)$$

Although this normal is more than enough to get good looking water, the paper suggests using normal maps to obtain even more realistic water, as the look with the previous normal could be a little like plastic. However, to extract normals from the normal map, tangent and bitangents are needed and therefore computed in the fragment shader. For that, the paper suggests following the method described in the article 'Normal Mapping Without Precomputed Tangents' by Christian Schüler.

Schüler explains a method that obtains tangents and bitangents from perturbed normals. Tangent and bitangents are the gradient of the texture coordinates. So, putting them as a function of the position we get this:

$$T = \nabla u \quad B = \nabla v$$

$$du = T \cdot dp \quad dv = B \cdot dp$$

Introducing the fragment differences in the equations above we get the following linear systems.

$$\Delta uv_1 = dFdx(uv); \Delta uv_2 = dFdy(uv); \Delta p_1 = dFdx(p); \Delta p_2 = dFdy(p)$$

$$\begin{cases} \Delta uv_1.x = T \cdot \Delta p_1 \\ \Delta uv_2.x = T \cdot \Delta p_2 \\ T \cdot (\Delta p_1 \times \Delta p_2) = 0 \end{cases}$$

$$\begin{cases} \Delta uv_1.y = B \cdot \Delta p_1 \\ \Delta uv_2.y = B \cdot \Delta p_2 \\ B \cdot (\Delta p_1 \times \Delta p_2) = 0 \end{cases}$$

Note that the 3rd equation is introduced to maintain the orthogonality with the normal. Now solve for T and B:

$$T = \begin{pmatrix} \Delta p_1 \\ \Delta p_2 \\ \Delta p_1 \times \Delta p_2 \end{pmatrix}^{-1} \begin{pmatrix} \Delta uv_1.x \\ \Delta uv_2.x \\ 0 \end{pmatrix}$$

$$B = \begin{pmatrix} \Delta p_1 \\ \Delta p_2 \\ \Delta p_1 \times \Delta p_2 \end{pmatrix}^{-1} \begin{pmatrix} \Delta uv_1.y \\ \Delta uv_2.y \\ 0 \end{pmatrix}$$

With these systems, it is possible to obtain tangent and bitangents, which is in fact what the paper uses. However, computing an inverse matrix per pixel each time can be quite expensive. That is why, in a follow up article in 2013, 'Followup: Normal Mapping Without Precomputed Tangents', Schüler explains some simplifications, which turn out in the next equation

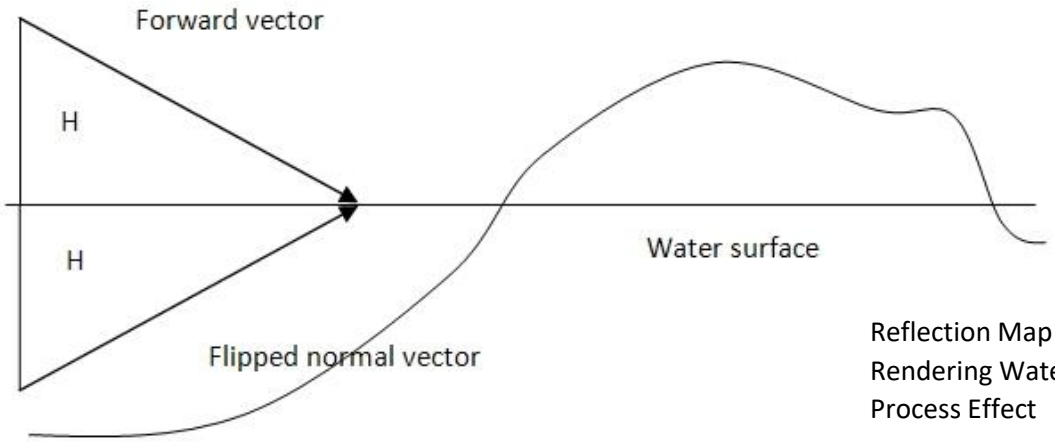
$$T = \frac{1}{|\Delta p_1 \times \Delta p_2|^2} \begin{pmatrix} \Delta p_{1\perp} \\ \Delta p_{2\perp} \\ \Delta p_1 \times \Delta p_2 \end{pmatrix}^T \begin{pmatrix} \Delta uv_1.x \\ \Delta uv_2.x \\ 0 \end{pmatrix}$$

$$B = \frac{1}{|\Delta p_1 \times \Delta p_2|^2} \begin{pmatrix} \Delta p_{1\perp} \\ \Delta p_{2\perp} \\ \Delta p_1 \times \Delta p_2 \end{pmatrix}^T \begin{pmatrix} \Delta uv_1.y \\ \Delta uv_2.y \\ 0 \end{pmatrix}$$

Finally, this article applies some optimizations. As the last row of the matrix will be multiplied by zero, it can be skipped. He also treats the determinant as a scalar to preserve the relation between the lengths of both vectors. For that, he takes the maximum out of both tangent and bitangent.

Optics: Reflection

Reflection are obtained in the most naïve way. The paper suggests to render the whole scene with a flipped camera below the sea level.



Reflection Map Scheme – Rendering Water as a Post-Process Effect

On the water pass, sample the reflection map by projecting the water surface position onto the flipped view. To give the sensation of irregular reflections, modify slightly the textures applying sinusoidal movements as well as using the surface normal to transform it. To have it animated is important to use time in this modification.

Optics: Refraction

The refraction makes uses of the result of the lighting pass to get the information of the bottom of the sea. Once again, this textures needs to be modified to be more irregular and hence, more realistic. However, the refraction will also be in charge of giving the actual color to the water itself. First, perform a mix between the refracted texture and the surface color (water color in shallow waters). As the mixing factor, use the division between accumulated water (the length between the fragment position and the water surface) and visibility (variable set by user to determine how visible is the depth of the water).

$$color = mix\left(refractColor, surfaceColor, clamp\left(\frac{length(Surface-Fragment)}{Visibility}, 0, 1\right)\right)$$

Once the mix is computed, another mix is needed. This time, it will be used to differentiate shallow from deep waters. For that, use the previous result and a new user defined big depth color. The mixing factor this time will be a division between the depth of the water (difference between the y components of surface and fragment positions) and extinction.

$$color = mix\left(color, bigDepthColor, clamp\left(\frac{(surface.y - fragment.y)}{Extinction}, 0, 1\right)\right)$$

Extinction tries to mimic the fact that not all colors fade out at the same rate on water. For example, the green and blue colors fade out much slower than the reddish ones. So, the extinction is a variable that the user needs to set to each color component.

Specular

Specular component helps to give water volume. The paper suggests to compute the reflection of the view and use this to compute the dot product with the light direction. After that, the author uses the next modified version of the specular:

```
float dotSpec = clamp(dot(mirrorEye.xyz, -lightDir) * 0.5 + 0.5, 0.0, 1.0);

specular = (1.0 - fresnel) * clamp(-lightDir.y, 0.0, 1.0) * ((pow(dotSpec, 512.0)) * (shininess
* 1.8 + 0.2));

specular += specular * 25 * clamp(shininess - 0.05, 0.0, 1.0);
```

Fresnel

The Fresnel term is needed to add reflection and refraction colors. This terms describes how much light refracts and reflects when passing from one medium to another, in this case, from air to water. It can be computed as follows:

$$c = \cos(\alpha) * \frac{n_a}{n_b}$$

$$g = \sqrt{1 + c^2 - \left(\frac{n_a}{n_b}\right)^2}$$

$$R(\alpha) = \frac{1}{2}\left(\frac{g-c}{g+c}\right)^2 \left(1 + \left[\frac{c(g+c) - \left(\frac{n_a}{n_b}\right)^2}{c(g-c) + \left(\frac{n_a}{n_b}\right)^2}\right]^2\right)$$

Where α is the angle between the normal and the incident ray and $\frac{n_a}{n_b}$ is the difference between indices of refraction between air and water. However, these computations are really expensive, so it proposes to use the following simplification:

$$R(\alpha) = R(0) + \big(1 - R(0)\big) * (1 - \cos(\alpha))^5$$

Where R(0) is equal to $\frac{n_a}{n_b}$. So, finally to add all the results do the following:

$$FinalColor = fresnel * refractColor + (1 - fresnel) * reflectionColor + Specular$$

There is a problem, though. These will give us hard waves, which is not what realistic water looks like. To correct this, make a last blend between the obtained color and the refraction, using the multiplication between the accumulated water and shore hardness (a user defined variable) as the lerping factor. In this way, the user can select how hard the waves should be.

Foam

Finally, the paper suggests implementing foam. There are two types of foam: foam dependent on depth or on height. The first one is applied by having a range defined by the user, where if the height is higher than the minimum height, then foam will be applied. Instead, if it is in the in-between, use linear lerp to fade out little by little the foam texture. The one that depends on height is computed with the following function:

$$FoamHeight = clamp\left(\frac{CurrentLevel - H_0}{H_{max} - H_0}, 0{,}1\right)$$

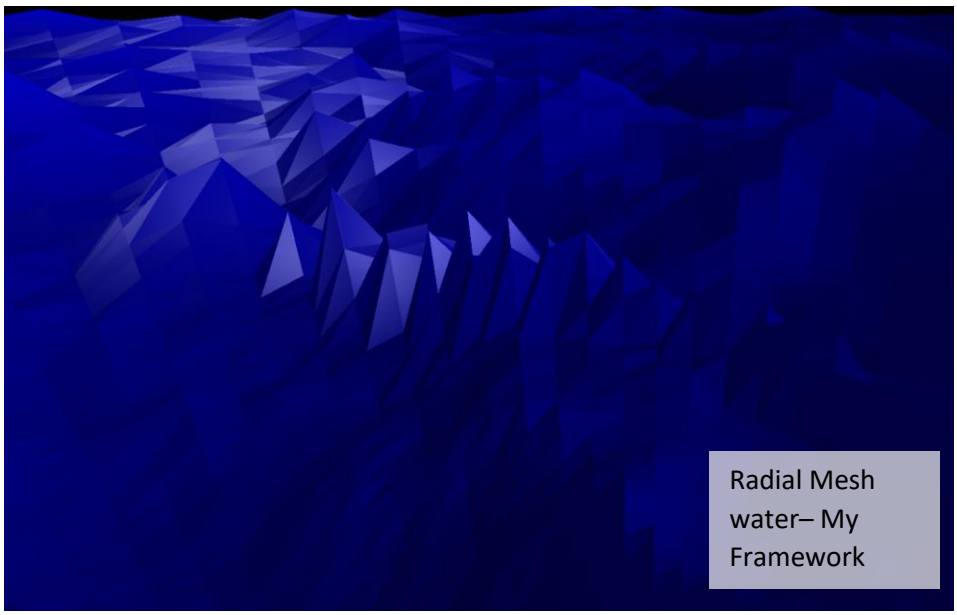
Both foams need to be added with the final color.

MY IMPLEMENTATION

At the beginning, I started working with the first paper. But, later on, I decided to change the paper. This are the results I was getting with this approach.

First paper's implementation:

I started by creating the radial mesh on the CPU as explained before. So, whenever I was drawing the water, I was rendering these created radial grid on the position of the camera (ignoring the y-coordinate). In this way, I get the LOD. Then, in the vertex shader, I transform these points to Cartesian and modify them using the height given by the height map. These are the results:



Radial Mesh water– My Framework

Of course, this is a very early implementation, but the radial mesh works well. The problems came with the normal. The paper does not explain how we can obtain them. It just says that by tilting to the camera is more than enough. Moreover, as I was working with geometry, smoothing water was headache, as the only solution I could find was by increasing the quality, which goes against the idea of having a radial mesh. So, finding for possible solution, I came across with the second paper.

Implementing the second paper:

I discarded everything except from the height map pre-pass, as this second paper does not specify how the height maps are obtained.

Rendering Pipeline

I needed to once again reorganize my pipeline. I decided to render the water after lighting. In this way I was simplifying the use of frame buffers. But, its drawback is that the water cannot be affected by

other lights than the sun. The height map and reflection pass do not really depend on any pass, so I just do them before the water pass. The water rendering needs a lot of textures: Depth buffer from the GBuffer, result of the lighting (used as back buffer on refraction), reflection map, height map, normal map and 2 for the foam (one per type). The computations will take place only on a fragment shader.

Water Surface

Once in the fragment shader, I compute the world position of the depth buffer position as seen in class. I decided to make the computations on world space as it was easier for me to understand.

Regarding the intersection between the plane and the view vector, at the beginning I was doing a complete intersection with dot products. However, taking into the account the plane is flat and that its normal is only pointing upwards, I decided to make these computations on the y-axis.

Pseudocode:

```
//Take view to the fragment
const vec3 view = normalize(pos - cameraPosition);

//Intersection between view and the sea level plane
const float denominator = view.y;

//Distance from camera to intersection
float d = (currentLevel - cameraPosition.y) / denominator;

//Compute intersection in sea level
vec3 surface = cameraPosition + view * d;

//Loop with intersections intersections
for(int i = 0; i < N; ++i)
{
	float height = texture2D(Bump, (surface.xz + view.xz) / size).x;
	height *= MaxWaveHeight / N;

	currentLevel += height;

	//Distance to the new sea level
	d = (currentLevel - cameraPosition.y) / denominator;

	//Compute new surface point
	surface = cameraPosition + view * d;
}
```

Note that I am dividing the texture coordinates by a size. This is to scale the height map and have it repeating more or less times. Also, it may happen that N is not enough for our desired max wave height. In this cases just increase its value.

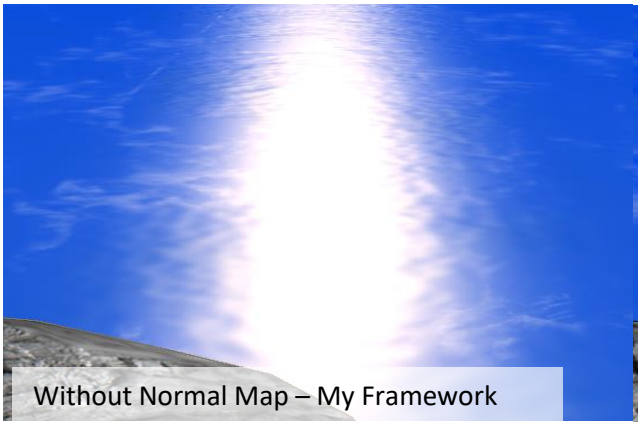
Artifacts with low iterations – My Framework

Normal

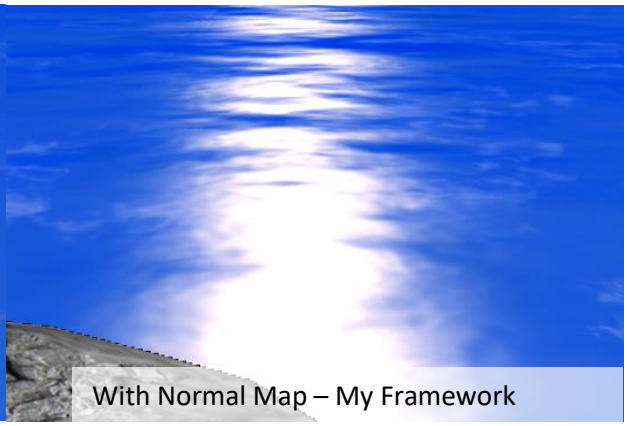
Once the surface is computed, computing the normal of the height map is straightforward:

```
//Compute the texture coordinates to be used from this point on
vec2 realUV = surface.xz / size;

//Take neighbour info
float westH = texture2D(Bump, realUV - vec2(mapIncrement.x, 0)).x;
float eastH = texture2D(Bump, realUV + vec2(mapIncrement.x, 0)).x;
float northH = texture2D(Bump, realUV + vec2(0.0, mapIncrement.y)).x;
float southH = texture2D(Bump, realUV - vec2(0.0, mapIncrement.y)).x;

vec3 normalHeightMap = normalize(vec3((westH - eastH), NormalScale, (southH - northH)));
```

However, as I explained before this will not give good results. It will look like plastic. So to fix it, I implemented normal maps.



Without Normal Map – My Framework



With Normal Map – My Framework

So compute the normal from the normal map as explained by Schüler on its article. The following code it is taken from the article. It is not that hard to get as it follows the formula explained above:

```
//Take the edges of the surface
vec3 dViewX = dFdx(view);
vec3 dViewY = dFdy(view);
vec2 dU = dFdx(realUV);
vec2 dV = dFdy(realUV);

//Compute the perpendiculars of these edges
vec3 dViewYPerp = cross( dViewY, normalHeightMap );
vec3 dViewXPerp = cross( normalHeightMap, dViewX );

//Solve system to get tangent and bitangent
vec3 tangent   = dViewYPerp * dU.x + dViewXPerp * dV.x;
vec3 biTangent = dViewYPerp * dU.y + dViewXPerp * dV.y;

//Compute the inverse of the determinant,
//but only taking the scalar of both tangent and bitangent
float inverseDet = 1.0 / sqrt(max(dot(tangent, tangent), dot(biTangent, biTangent)));

//Compute the tangent to world matrix
mat3 tangentToWorld = mat3(tangent * inverseDet, biTangent * inverseDet, normalHeightMap);

//Return the normal from the normal map
return normalize(tangentToWorld * (2.0 * texture2D(NormalMap, realUV).rgb - 1.0));
```

Finally, using these function, I sampled with different frequencies the normal map, following the same idea of the height maps, to get a good enough result.

Optics

To get the reflection, I am rendering the geometry with a camera underwater and rotated as explained above. However, to this reflected scene, I am not applying lighting. I only apply global ambient. In this way, fidelity is lost in favor of performance. Also, I only need the diffuse texture, so using a render buffer and a color attachment is more than enough.

Once the scene is rendered, I project the surface point and get the texture coordinates in the following way:

```
//Project surface onto the flipped camera perspective
vec4 reflectionUV = WorldToWindowReflection * vec4(surface.x, surface.y - currentLevel,
surface.z, 1.0);

//Perspective division
reflectionUV.xy = reflectionUV.xy / reflectionUV.w;

//Between 0 and 1 to use it as texture coordinates
reflectionUV.xy = 0.5 * reflectionUV.xy + 0.5;

//Sligthly modify the texture coordinates to acess the reflection map
/*
...
*/

const vec3 reflectionColor = texture2D(ReflectionMap, modifiedReflectionUV.xy).rgb;
```

As we can see on the code, the process is really simple. Obtaining the refraction is even easier as it is only to get the color from the diffuse texture and apply the formulas developed above. It is important to mention, though, that the modifications on the reflection and refraction map need to be carefully done. In my case, I was applying a sin to both axes, but this led to artifacts when moving the camera (as the sinusoidal transformation was also moving).

There is not much to mention on the Fresnel computation. Just compute it with the formula explained above, taking into account that the view vector needs to be recomputed, as we are no longer working with the original geometry, but the water surface. Also, I am computing the specular component with Blinn Phong instead of the one suggested, as it already gives good results.

Foam

Foam is really simple to obtain. I use different textures for height and depth foam. Both scalars are obtained as the following:

```
//Height foam
float foamH = clamp((currentLevel - FoamMinHeight) / (MaxWaveHeight - FoamMinHeight),
0.0, 1.0) * 0.2;

//Depth foam
float foamDepth = 0.0;

if(waterDepth < FoamAlways)
	foamDepth = 1.0;
else if(waterDepth > FoamAlways && waterDepth < FoamStart)
{
	foamDepth = (waterDepth - FoamAlways) / (FoamStart - FoamAlways);
	foamDepth = mix(1.0, 0.0, foamDepth);
}
```

However, how I apply them is quite different. Instead of following the paper recommendation, I decided to apply the height foam by adding it to the color, whereas the depth is mixed at the end of the code using as the mixing factor the foam depth we just computed.

HOW TO USE

I used the framework of the assignments, so you will find similar controls. These are the controls:

W, S: Go forward and backward

A, D: Go left and right

Mouse Wheel: Go up and down

Mouse Right Button (Held): Rotate Camera

R: Restart Scene

F5: Reload Shaders

On the demo you will find a lot of variables that can be changed. This is what they are and do:

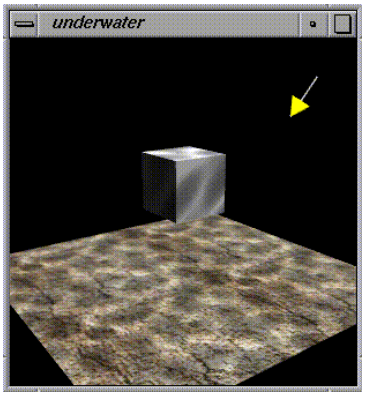
- Display: Shows what it is rendering onto screen.

- Wave:
  - Wave Type: Different height maps to choose from.
  - Modify Geometry: This is a little experiment mixing both approaches. It is totally wrong.
  - Max wave height
  - Iterations: The iterations that the algorithm does to find the wave surface.
  - Wave Texture Size: These value will be used to scale the height map texture.
  - Normal Scale: These will control how smooth or rough the normal is
- Refraction:
  - Visibility: Makes the shallow waters more crystalline or muddier
  - Extinction: Controls how each color component extincts in the deep water.
  - Shore Hardness: Controls how hard the water is. If it is 0, the result would be just the refraction.
- Specular:
  - Shininess
  - Specular or sun color
- Water Surface color: Color of shallow waters
- Depth Water color: Color of deep water
- Foam:
  - Min Height: From this height to the max amplitude, foam will be blended.
  - Min Depth: If depth is smaller than this, then foam will be applied.
  - Dies out Depth: This is the depth where the foam starts to appear.

OTHER IMPLEMENTATIONS

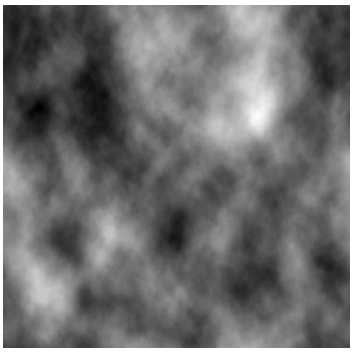
On this report I presented two approaches, both of them explaining how to obtain convincing water when the camera is above it. But, what happens if we go below water? I think that the next logical step, would be implementing a render pass when the camera is below a wave. In fact, my implementation does not check if the camera is underwater, which leads to a strange result.

To improve the look underwater, the most important thing are caustics. Caustics are optical effects created when light intersects a transparent object, in our case water. To obtain this kind of effects, the easiest way is by applying a caustic texture onto the surface, but from the sun's light direction perspective. As in reflection and refraction, it is important to slightly transform it over time to animate it. If this effect is added with an atmospheric attenuation and a refraction of what it is above water, the results will be really amazing.



Underwater Caustics – *OpenGL rendering of Underwater Caustics*

Apart from that, there is a way to obtain more realistic waves: Fast Fourier Transforms. The cost will increase quite a lot, but the patterns created from sampling repeatedly the same height map will most likely disappear. Of course, this is a whole new topic to explore. But, take into account that even this approach will not be able of creating concave waves. Here is a height map created with FFT:



Height map created with FFT – Simulating Ocean Water

PROBLEMS

As you have noticed, one of the biggest problem that I had is not knowing how to continue the previous paper. Now I think that I would be able of making something with that approach, but when I was implementing, no.

The second approach, I find it easier and better explained. At first I was not understanding correctly what a specific line was doing. To overcome this issue, I started thinking line by line what I was doing, which improved my understanding on the paper and on the code. It is important to mention, that the article "hides" some details on the sample code and I needed to understand why the author was making certain things. But, the algorithms are quite simple in my opinion, so following it was quite easy.

The harder part for me, was getting tangent and bitangents. So, that is why, I decided to read the original articles to understand (and even improve) the suggestion of the paper. These articles are really good documented, which helps a lot.

Finally, I want to remark that it is not easy to get good looking water. Even if the code is well implemented, I found myself self-doubting about if it was correct or not. The final results are great, but could improve a lot.

CONCLUSION

Overall the experience has been great. I do not like to left an implementation half way, but the second paper was more appealing. Rendering as a post processing effect gives much more flexibility, as its render pass can be located wherever you want. In my case, I implemented it after lighting, but easily it could be implemented before of it and modify the GBuffer so that lighting can be applied in the light pass. Making interactions between objects and water is easy too, as we only need to change the height map to

show them. Moreover, very different waters can be obtained just by changing variables. I like this flexibility.

Regarding performance, a lot of texture samplings need to be done. However, I think that overall the algorithm is quite fast, as it only relays on height maps to work. Also, because it is implemented as a post processing effect, in most cases, many computations will be saved, as the points that are not affected by the water will be discarded.

On the negative side, that much flexibility, makes it harder to obtain good results. A big part of my work has been tweaking values until I found something interesting. So, if I wanted to use it for a game, for example, I would for sure need to expend a lot of time adjusting it to it.

BIBLIOGRAPHY

Kryachko, Y. (2005). Using Vertex Texture Displacement for Realistic Water Rendering. *GPU Gems 2*. NVIDIA.

Wojciech Toman and Myopic Rhino (2009). *Rendering Water as a Post-process Effect.* Gamedev.net*.* Retrieved December 14, 2020, https://www.gamedev.net/articles/programming/graphics/rendering-water-as-a-post-process-effect-r2642/

Christian Schüler (2013). Followup: *Normal Mapping Without Precomputed Tangents.* The Tenth Planet*.* Retrieved December 13, 2020, http://www.thetenthplanet.de/archives/1180

3Bulue1Brown (2018). *But what is the Fourier Transform? A visual introduction.* Youtube. Retrieved December 13, 2020, https://youtu.be/spUNpyF58BY

Mark Kilgard. *OpenGL-rendering of Underwater Caustics.* OpenGL. Retrieved December 14, 2020, https://www.opengl.org/archives/resources/code/samples/mjktips/caustics/

Tessendorf, J. (2001). Simulating Ocean Water.