REAL-TIME WATER RENDERING

CS 562

Gabriel Mañeru
Senior RTIS Student at
Digipen Institute of Technology
Europe - Bilbao
gabriel.m@digipen.edu

December 18, 2019

ABSTRACT

Rendering water surfaces are a common need in games, asides from games that focuses on fluids as it's main mechanic, it is indeed a key element that grants a realistic ambience to oceans, fast rapids, calm lakes or fountains.

1 Introduction to the problem

The objective is to implement a water tool that will come in handy in most of the situations. This will take into account high quality shading and fine geometric detail. Analysing the different types of implementation we can divide it into three groups:

- The first group can be categorize as Oceans. These implementations are robust and homogeneous. They can handle large chunks of water using realistic wave-like behaviours. However, they rely on baked mathematical equations that makes them be difficult to customize and adapt to a specific scene.
- The other side of the spectrum are Flat Surfaces. They are the most common water implementation as it requires few resources and give a decent result without compromising a big performance cost. It relies on having a more complex pixel shader that gives dynamism and realism to the surface with very little vertex displacement.
- The future tends to a real simulation as Positional Based Fluids. This approach is the most faithful to real the water behaves. However, those are still too expensive for complex games but there is currently a lot of research about it.

2 Implementation proposed by the reference paper

Reference Paper:
Using Vertex Texture Displacement for Realistic Water Rendering

The model of the reference paper is based on a flat surface which vertices' heights are defined using tiled height maps. For simulating the dynamism of the water the texture coordinates are displaced over a certain direction. Then the sampled heights of the maps are accumulated, as in a Fourier synthesis, defining the current vertex height.

For sampling it uses a radial grid centered at the camera position and tessellated in such a way that it gives more detail the closer the is viewer (simple Level-Of-Detail).

The shading involves using more height maps for lightning calculations. This reproduce faithfully fine detail of high-frequency waves. In addition it adds local shading created by buoyant objects such as choppiness or foam generation.

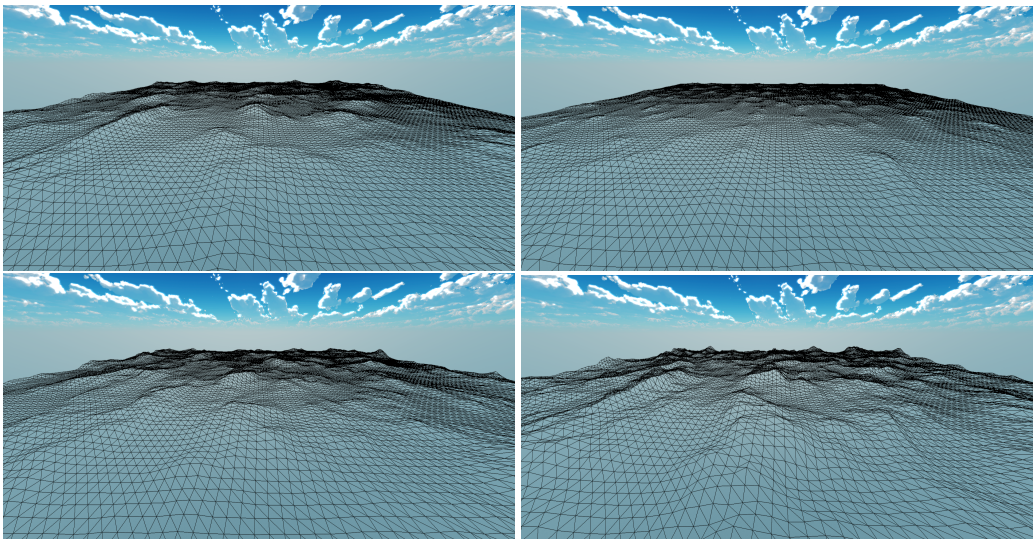
3 Implementation Details

3.1 Model

By observing water in real life, although at a first glance it seams to be a chaotic system, by taking a closer look at it we can derive its behavioural composition into discrete wave combinations. We can appreciate its rhythmic movement due to the wave oscillation and represent it using multiple types of layers such as standing waves, backward waves and a superposition of different frequencies.

3.1.1 Global Motion

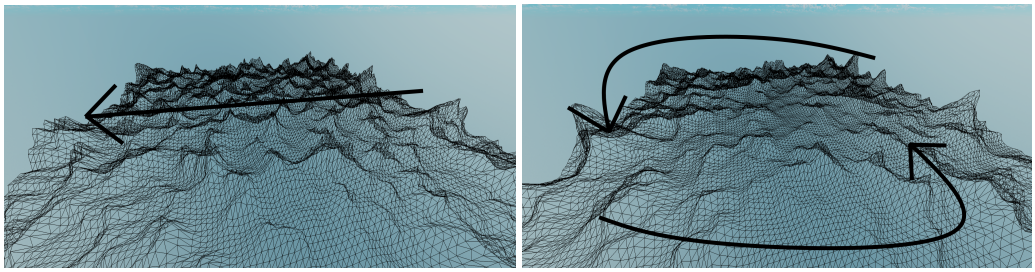
For creating the model I have used that same process: a superposition of those layers for computing the actual height for each vertex. A layer defines and represents an actual wave by its speed, wavelength and amplitude (or height). Its combination already represents a faithful global motion similar to the Ocean implementation, but we can only control its direction globally. If we want to represent for example, a river that adapts to the scene and follows its geometry, we should be able to transform this global motion and turn it into a local motion where the user may specify and control the flow of the river.



Three different waves and its concatenation

3.1.2 Local Motion

The way I control this flow is through a 2D-vector field (which could be easily loaded and stored from a texture). These vectors define the displacement direction of the texture coordinates. These results in height maps flowing correctly. However, this introduces another problem: I'm currently using non tile-able height map created programmatically using Perlin noise, which means, that when repeating the height map there will be a discontinuity in the wave's height. Also, the farther the texture coordinates are displaced, the greater distortion the image will have. In order to avoid those problems, we interpolate between several height maps per layer: we use an height map for a certain time and when the image is not usable anymore we have ready another one. They are interpolated at the same rate in order to simulate the effect of being in-phase, which results in a layer that flows indefinitely other a certain vector field.



Different flow maps (better showcase in demo)

Combining the global motion of the ocean and the local motion of the vector field, we can create any kind of water, from simulating steady oceans to rapid rivers.

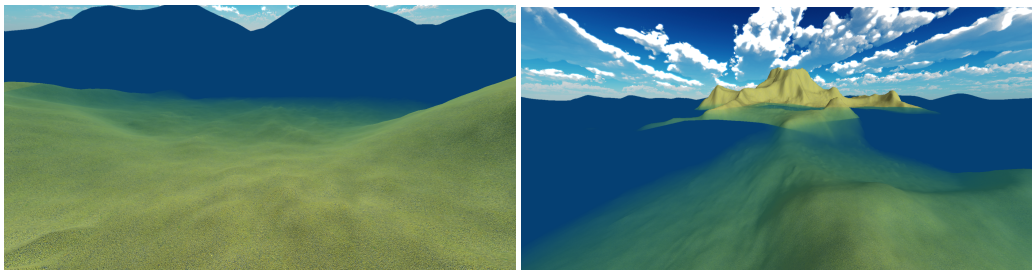
3.2 Shading

The shading part is implemented in a Deferred shading pipeline including the water as a post-process effect after the lightning pass. The gBuffer properties are stored in view space in order to avoid extra computations.

3.2.1 Shoreline color

To remove the flat blue color of the water, I first implemented some gradient that represents the evolution of the water's composition from the shoreline to the deep ocean. I have used the position of the sea bottom and the position of the current water vertex to extract the $ShoreLength$ that tries to approximate the distance travelled by the light under the water. Then, by defining a certain $ShoreDistance$ which represents the limit of the shore, we can then compute the $ShoreFactor$ going from 0 to 1.

Then, the color influence of the water ($ShoreColor$) is simply a linear interpolation of both of the shore colors using that $ShoreFactor$.



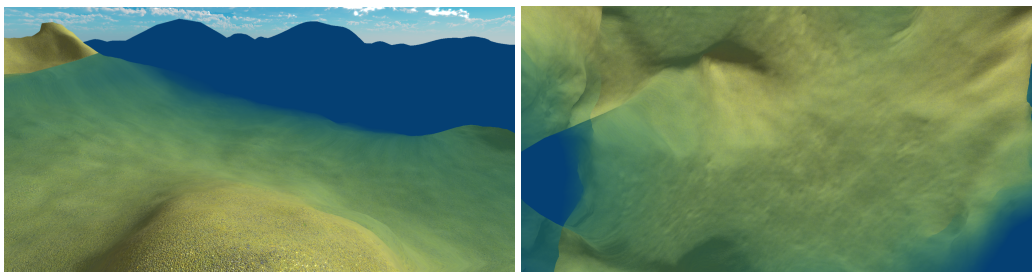
Water color varying from shoreline to deep ocean

3.2.2 Water Refraction

For computing the refracted color of the sea bottom, we start by refracting the $vView$ vector by the $vNormal$ vector of the water surface, obtaining the $vRefractView$. Then, we compute the difference between $vRefractView$ and $vView$, which is the refraction's deviance and we scale that vector by the $ShoreFactor$ in order to control the refraction close to the shoreline.

We are creating some artifacts when computing the refraction in this way, however, we can handle it as there will be more shading layers on top of it. The importance of the refraction is to give dynamism to the scene rather than having a perfect simulation.

The output ($RefractedColor$) is then linearly interpolated with the $ShoreColor$ in order to create the base $WaterColor$. This color will tend to be transparent in the shoreline with very little deformations.

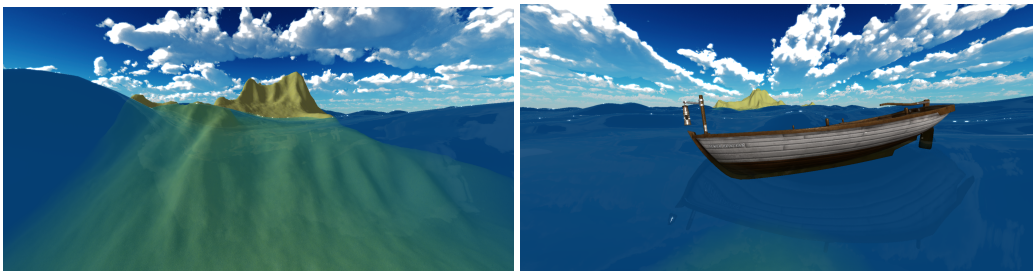


Sea bottom affected by water refraction

3.2.3 Reflections

For the reflections I implemented Screen Space Reflection in order to faithfully represent local objects over the water. This algorithm consists on, marching along a certain direction while sampling over a texture until you hit your corresponding texel to reflect. If the ray goes out of bounds or just passes some limits, you just take the color of the skybox to recreate the global reflection of the sky.

Then the implementation consists on computing the $vReflectView$ by reflecting the $vView$ over the $vNormal$ of the water surface. We then run the algorithm from the $vPosition$ along the $vReflectView$ and we obtain the $ReflectedColor$. Then, we compute the factor that represents how aligned the $vNormal$ and the $vView$ are. We use it to interpolate the final color between the water color and the $ReflectedColor$.



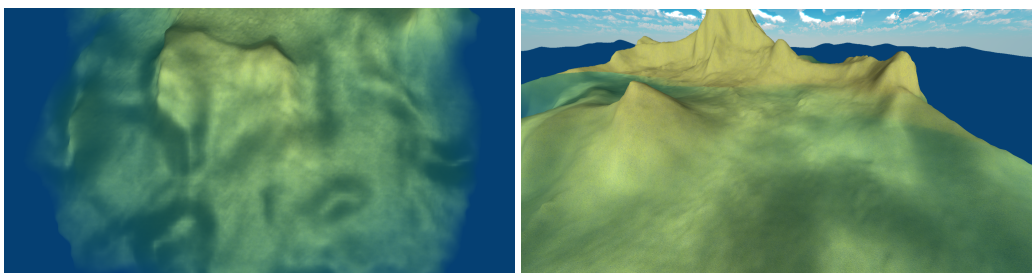
Sky and boat reflected into the water

3.2.4 Caustics

The current problem is that apart from the refraction, the sea bottom color doesn't vary too much. I wanted to add more dynamism to it and a the simpler way to go is adding fake caustics. Caustics are an accumulation of refracted rays, we could try to fully implement it but that involves ray-tracing per frame the influence of each vertex onto the others.

Instead, what we could do, is to only take into account the direct influence of the current vertex for the caustic calculation, which reduces the problem to only checking the orientation of the normal, that corresponding to the sea bottom. This implementation will not add bright spot but only dark areas, still that would be enough for us as we only want to add dynamism.

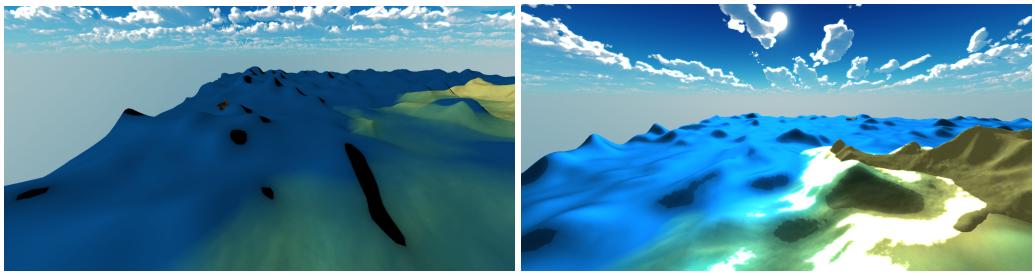
Then, the implementation requires a pre-pass that computes the shadow of the water. Also we need to compute the world texture coordinates for accessing that shadow map, which consists on bringing the sea-bottom position to world space and us it as texture coordinates. Once we know how much that texel is affected by the shadow ($CausticValue$), we just interpolate between the minimum and maximum value of the caustic ($CausticInterval$) and we multiply its result to the $RefractedColor$.



Caustic shadows in the water

3.2.5 Phong Lightning

Apart from its reflection, I also wanted to add a bit of influence of the sun. I added then the diffuse calculations that make the wave brighter when facing the sun and the specular factor which (along the bloom filtering of the engine) blurs the reflection of the sun and highly increase its appeal. Also, the specular interacts with the PBR texture of the sand creating small bright spots near the shoreline.



Phong Lightning effect on the waves (slightly hidden under shading)

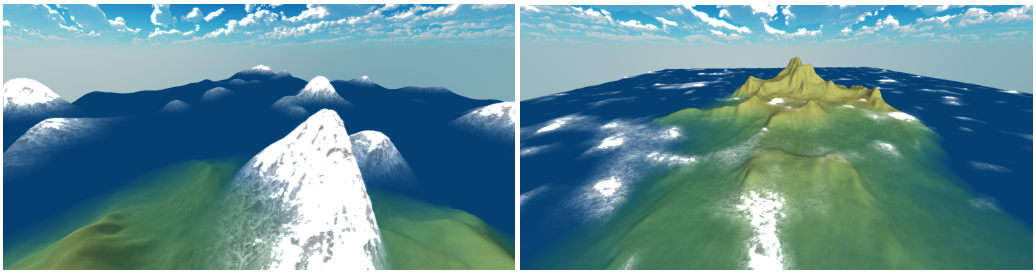
3.2.6 Buoyancy

An top of it, I wanted to add objects near the surface. In order to take advantage of the screen space reflections, I implemented a simple buoyancy for a boat to showcase more faithfully its reflection on the water surface.

The actual implementation is not physically accurate as this wasn't it's intention.

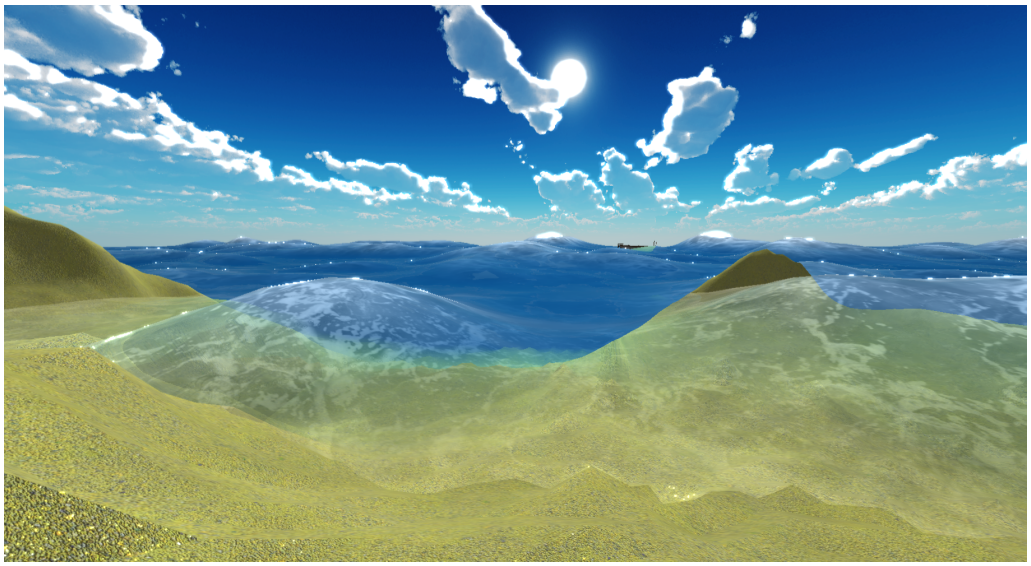
3.2.7 Foam

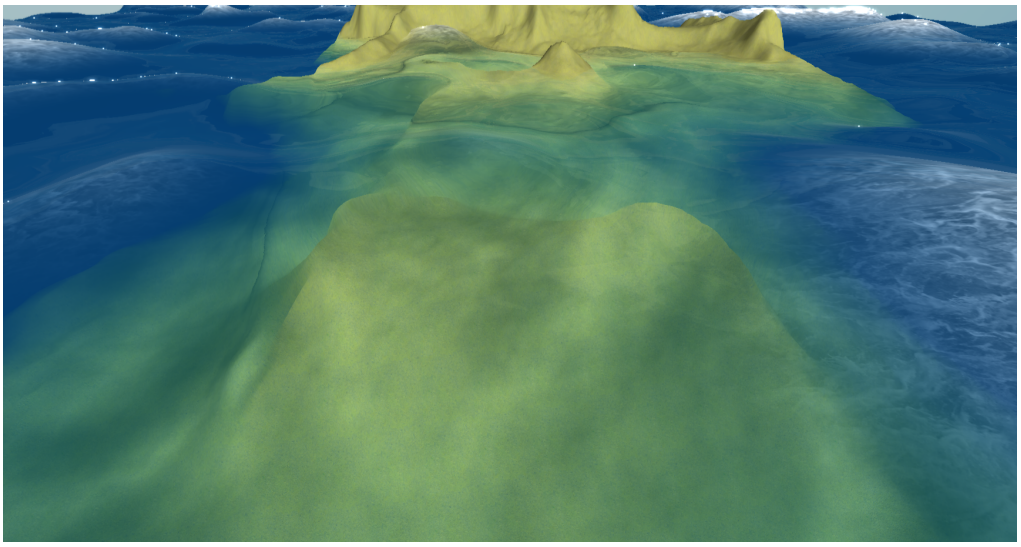
For the foam I implemented it right-away using the vertex height: I just have two height limits where the foam starts and when it gets to its maximum intensity. By smooth stepping between those three values I get the a foam factor, which I use in order to interpolate between the color of the water and a foam texture. It's purpose is not to faithfully represent foam, as I would go with a particle system to do so, but to add a bit of texture to the wave instead of letting them being completely flat.

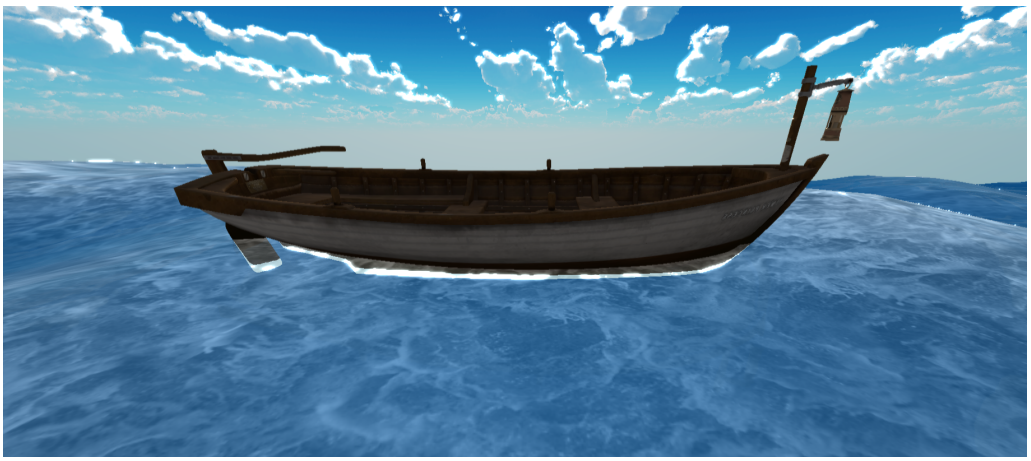


Foam texturing over the waves

4 Demo Showcase







5 Possible Improvements

- There is still a lot of performance that could be improved from creating the model, a level-of-detail and in the shading.
- Flow map creation tool in order to design the scenes both direction and height of the water over the space.
- Second diffuse pass for drawing objects over the water in order to avoid artifacts when doing the refraction.
- Shadows for objects in the surface of shallow waters.
- Particle system for improving foam realism

6 Conclusion

By combining the performance of the global motion and the flexibility of the localise flow we are able to simulate highly interactive and realistic water. However, hardware and algorithms are recently evolving into the particle simulation which will add the next step into the water realism whenever its performance will be manageable.

References

[1] Yuri Kryachko. Using Vertex Texture Displacement for Realistic Water Rendering In *GPU Gems 2, Chapter 18, Nvidia Corporation*, 2005.

[2] Jerry Tessendorf. Simulating Ocean Water. In *Simulating Nature: Realistic and Interactive Techniques*, 2001.

[3] Carlos Gonzalez-Ochoa. Rendering rapids in Uncharted 4. In *Advances in real-time rendering, part I, SIGGRAPH 2016 Courses*, 2016.