Real-Time Rendering of Ocean Water With Dynamic Skybox

Aaron Hornby
10176084

- **Introduction**

The overall focus of this term project was to attempt the implementation of real-time rendering of a simulation of ocean water/waves in a photorealistic manner. An application was to be developed to support Windows, Linux and macOS desktop devices with hardware capable of supporting OpenGL 4.1. This version of OpenGL was primarily chosen as it is the latest version supported by modern macOS systems. The repository holding the project files was to be open-source on GitHub with simple download instructions for an easy entry for interested parties wishing to learn the techniques presented in this paper. The code itself was to be well-documented and logical to follow and learn from. Due to discovering the importance of how the ambience of a quality skybox adds to the scene, I also chose to implement a relatively simple dynamic skybox. The sandbox environment was designed to be highly extensible for future work accompanied by a user-interface fit with numerous tweakable parameters. A user is hopefully able to play with the demo itself while cross-referencing this paper in order to further understand the techniques. These parameters both influence the physical mesh representation of the water surface and different layers of the visual shading. Furthermore, while many other water renderers focus on flatter water, I wished to implement larger wave systems to discover how they react to the shading algorithms which primarily focus on approximating the surface with a flat plane.

The results should provide a well-documented open-source repository that can be studied by other students or others with interest in this topic. The project will primarily benefit hobbyist game developers who wish for an easier entry point into a highly popular feature of many games. The project is motivated by the prevalence of water technology in games, movies and other interactive media. High quality water is usually one of the most poignant appeal points in the visuals of a game. The challenge of this project was due to the interlinking of aspects of rendering, modeling and animation. While the rendering component was the main focus, the application can serve as a good reference to those interested in any of the three disciplines.

Every water renderer is different in implementation, yet many of the features are fairly similar. Water visually has many subtleties or layers to it in order to achieve a dynamic look. Traditionally, water has been rendered mainly as a texture-mapped quad. This doesn't look very realistic and isn't very extensible to many modern use cases. Nowadays, water is typically modeled as a large mesh or grid surface with actual displacement or even just a flat plane with purely visual techniques such as bump/normal maps or du/dv maps that give the appearance of a rougher or distorted surface. In the movie industry where pre-rendering frames is standard and thus higher quality water is possible, the main techniques used rely on the Navier-Stokes

equations of fluid dynamics or advanced particle simulations. For real-time systems it seems that Fast-Fourier-Transform (FFT) waves are becoming the standard since they allow a large variation in movement of the surface with sharper peaks.

The movie-standard techniques are state of the art when done correctly, however they are inherently limited due to the complicated foundation of physics propping them up and for their infeasibility for real-time systems. Some approaches may simply apply global environment mapping onto a static water surface which is a reasonable approach for surfaces that are either far away or not the focus of the application. These systems are easy to extend with more layered effects, as outlined in this paper and is a relatively simple process to achieve. FFT waves were considered as an alternate implementation later on in this project's development, however were decided against for two reasons. The first being that FFT waves are best implemented with compute shaders which require OpenGL 4.3+, unavailable for macOS systems and thus conflicting with the goal of multi-platform. Second, once Gerstner waves were implemented and heightmap noise was repurposed by being layered on top, the priority shifted to incorporating more rendering features.

I used an iterative approach for building up the project beginning with implementing the projected grid concept outlined by Johanson [2004]. It was necessary to implement the grid first due to it acting as the foundation for the entire simulation. In contrast to the CPU implementation in Johanson's demo, my application is written such that the grid positions are interpolated and normals are computed in a vertex shader. This was a logical improvement, since it removes the action of rebinding an entire grid into VRAM every frame. Once the grid was fully functional, a Perlin noise texture was mapped onto the grid to act as a heightmap resulting in a heightfield. Quickly, it became evident that this approach had the downside of a very repetitive appearance due to tiling. As a replacement, a Gerstner wave system was implemented directly in the same vertex shader, in accordance with Finch [2007]. This allowed for directional waves, useful for swells, allowing for many different physical configurations through modifying the wave parameters and combining them. Continuing, the layering process of visual effects was expanded and refined.

- **Methodology/Implementation**

Skybox Shading:

Due to the skybox being dynamic, the six composed textures in the cubemap must be updated every frame. This is accomplished by rendering the scene six times with a field of view of 90 degrees and aspect of 1, centering the camera eye at the world origin and rotating the forward vector of the camera to point in six orthogonal directions.

The skybox is shaded in four layers (in the following order):

1. Space/Star Layer:

This layer simply renders the textures in a typical skybox fashion, mapping them onto the inside of a cube.

2. Atmosphere/Sun Layer (Skysphere):

First, I painted a multi-stop 1D gradient that stores information of two colours based on the time of day in the simulation. The left half of the gradient designates the “sun peak colours”, while the right half represents the “sun horizon colours”. For the sun peak colour, from hours 0 to 12, the texture sampling variable (u) ranges from 0.0 to 0.5 and then reverses back to 0.0 in hours 12 to 24. The sampling variable for the sun horizon colour is 1-u. The overall shading is a slightly modified Lambertian diffuse algorithm where points on the sphere near the sun (peak) assume the sun peak colour, those 90 degrees from the sun assume the sun horizon colour and 180 degrees from the sun is black. All points in between get linearly interpolated between these three colour stops. For convenience, the sun was generated as a typical Phong specular highlight, that serves as a satisfactory effect for this implementation. In future versions, the sun could be replaced with an HDRI billboarded sprite for more realism. The underlying geometry began as a UV sphere, which presented diamond-like artifacts at the seams as well as a large variation in the sun’s size. In an attempt to reduce these issues, it was replaced with an icosphere which managed to alleviate the artifacts, however the size is still an issue that requires further investigation.

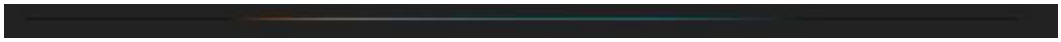


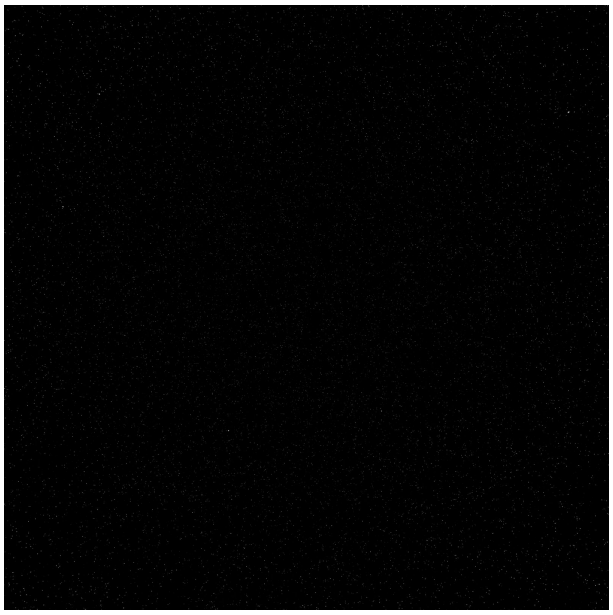
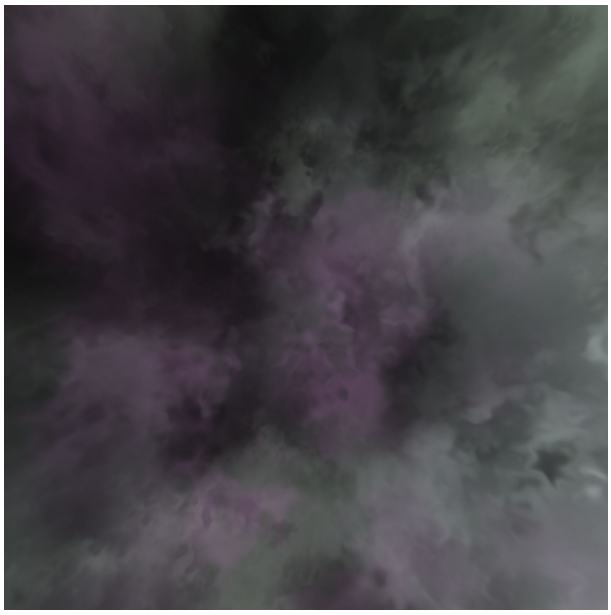
Figure 1 - The gradient texture.

3. Cloud Layer:

First a raw colour is sampled from an ordinary skybox of nebulae. This colour is converted to a grayscale intensity from 0.0 to 1.0. If the intensity is less than the cloud proportion user parameter, that fragment is considered part of a cloud and gets assigned its grayscale colour which fades towards a constant minimum intensity as the sun lowers in the sky. Other fragments get assigned a uniform gray colour, scaled by the overcast strength which fades in opacity as the sun lowers in the sky.

4. Fog Layer:

The user is able to set a global fogging colour that applies to the entire scene based on the distance from the camera. The RGB channels act as the tint, while the A applies as the density. The tint linearly fades towards black as the sun lowers in the sky. Since the skybox is considered infinitely far away, the densest fog colour is applied uniformly through rendering an alpha-blended screen-space quad.

Figures 2 (left) - one of the star textures. Figure 3 (right) - one of the base nebulae textures.

Water Shading:

The water is shaded in a composition of many layers (all these visual elements are combined in the fragment shader for the water surface):

1. Tint

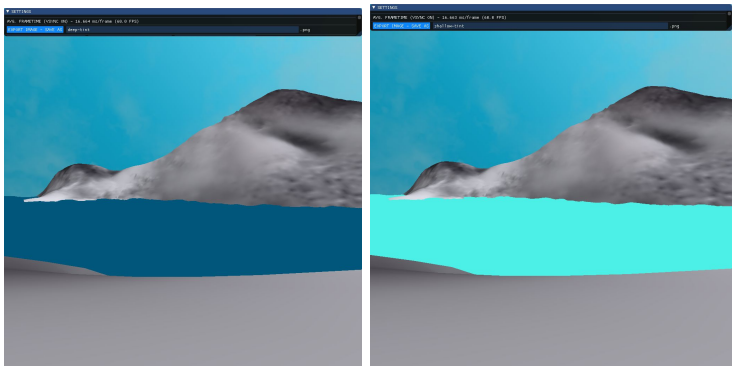


Figure 3 (left) - deep blue tint. Figure 4 (right) - shallow turquoise tint.

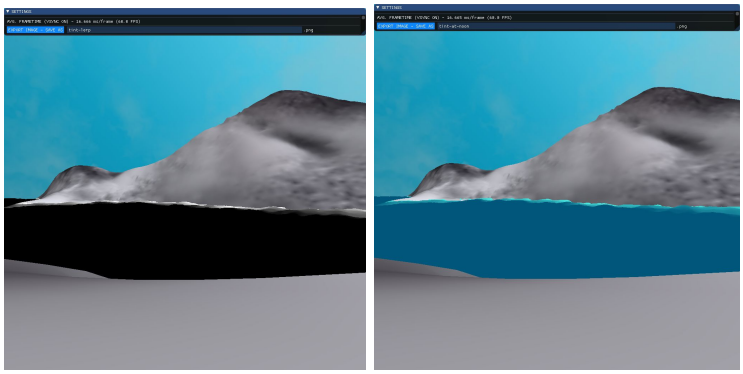


Figure 5 (left) - interpolation of the tint (black means deep, whiter means more turquoise). Figure 6 (right) - resulting tint layer.

```
const vec3 DEEP_TINT_COLOUR_AT_NOON = vec3(0.0f, 0.341f, 0.482f);

const vec3 SHALLOW_TINT_COLOUR_AT_NOON = vec3(0.3f, 0.941f, 0.903f);

float tintInterpolationFactor = 0.0f == tintDeltaDepthThreshold ? 0.0f : hack_skybox_in_back *
(1.0f - clamp(deltaDepthClamped / tintDeltaDepthThreshold, 0.0f, 1.0f));

vec3 tintColourAtNoon = mix(DEEP_TINT_COLOUR_AT_NOON,
SHALLOW_TINT_COLOUR_AT_NOON, tintInterpolationFactor);

vec3 tintColourAtCurrentTime = clamp(sunPosition.y, 0.0f, 1.0f) * tintColourAtNoon;
```

2. Local Refractions

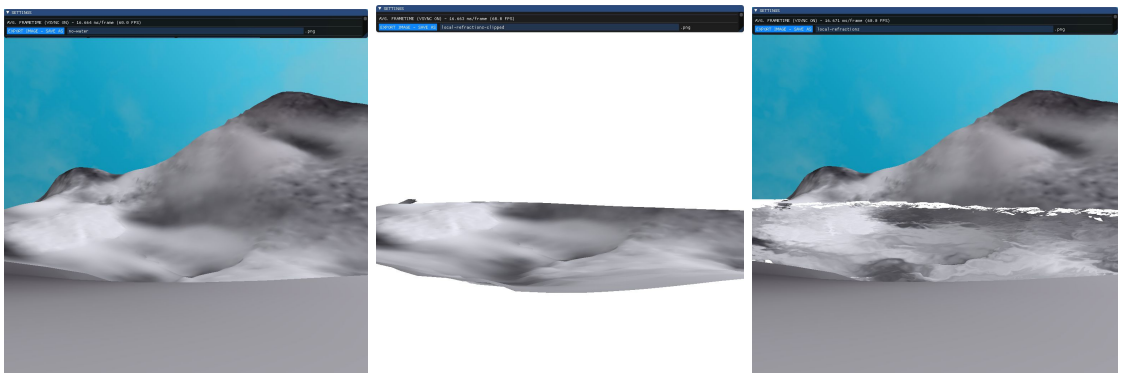


Figure 7 (left) - no water. Figure 8 (middle) - clipped refractive surface (notice how it looks more shallow). Figure 9 (right) - distorted refractions.

```
const float LOCAL_REFRACTIONS_DISTORTION_STRENGTH = 0.1f;

float localRefractionsDistortionScalar = LOCAL_REFRACTIONS_DISTORTION_STRENGTH *
(1.0f - viewVecDepthClamped);
```

vec2 uvLocalRefractionsYDistorted = mix(uvViewportSpace, uvViewportSpaceHeight0, localRefractionsDistortionScalar);

vec2 uvLocalRefractionsXZDistortion = localRefractionsDistortionScalar * vec2(-normalVecInViewSpaceOnlyYaw.x, normalVecInViewSpaceOnlyYaw.z);

vec2 uvLocalRefractions = clamp(uvLocalRefractionsYDistorted + uvLocalRefractionsXZDistortion, vec2(0.0f, 0.0f), vec2(1.0f, 1.0f));

vec4 localRefractionColour = texture(localRefractionsTexture2D, uvLocalRefractions);

3. Depth & Clarity

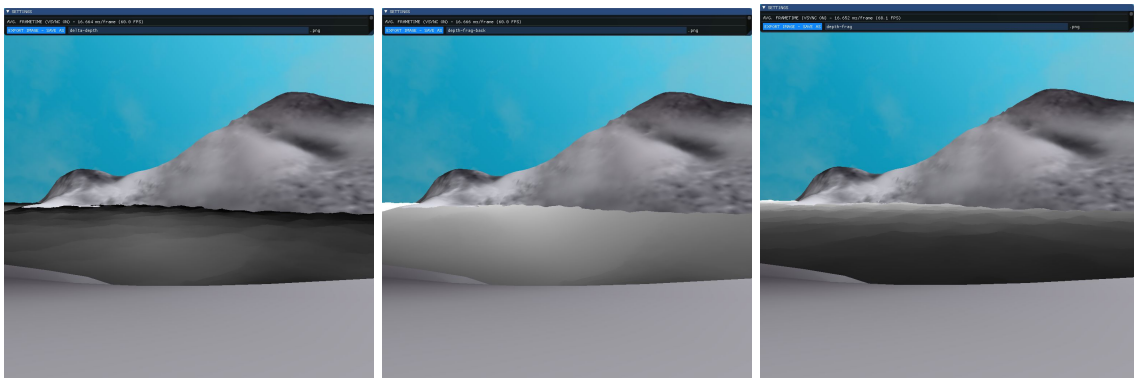


Figure 10 (left) - the depth delta from taking the difference in depth of the back buffer fragments (figure 11 - middle) and the water fragments (figure 12 - right).

4. Global Reflections

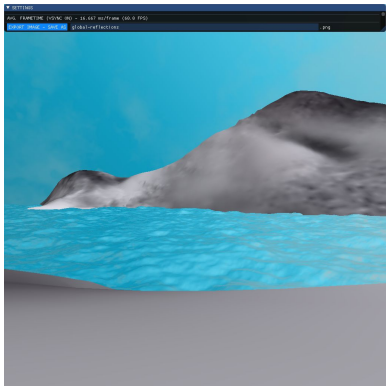


Figure 12 - skybox reflections onto the water surface.

vec3 R = reflect(-viewVec, normal);

vec4 skybox_reflection_colour = vec4(texture(skybox, R).rgb, 1.0f);

5. Specular sun highlights

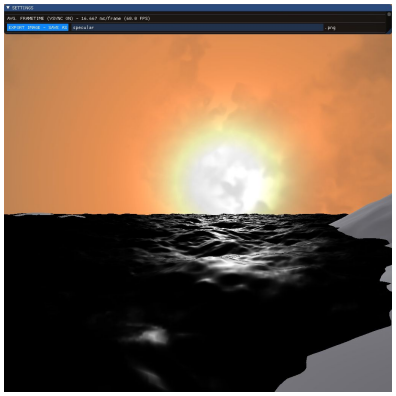
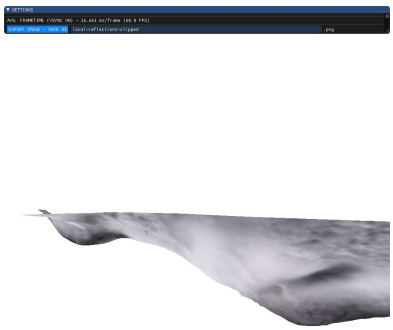


Figure 13 - Phong specular highlighting on the water surface.

const vec3 SUN_BASE_COLOUR = vec3(1.0f, 1.0f, 1.0f);

vec4 sun_reflection_colour = vec4(sunStrength * SUN_BASE_COLOUR * pow(clamp(dot(R, sunPosition), 0.0f, 1.0f), sunShininess), 1.0f);

6. Local Reflections



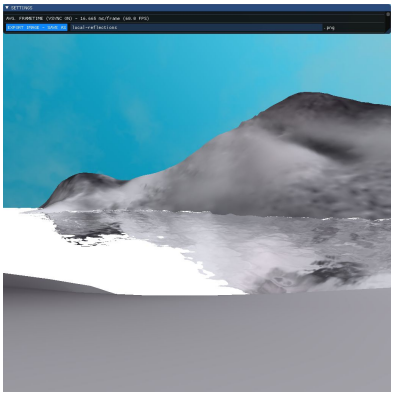


Figure 14 - clipped reflection surface. Figure 15 - distorted reflections.

const float LOCAL_REFLECTIONS_DISTORTION_STRENGTH = 0.1f;

float localReflectionsDistortionScalar = LOCAL_REFLECTIONS_DISTORTION_STRENGTH * (1.0f - viewVecDepthClamped);

vec2 uvLocalReflectionsYDistorted = mix(uvViewportSpace, uvViewportSpaceHeight0, localReflectionsDistortionScalar);

vec2 uvLocalReflectionsXZDistortion = localReflectionsDistortionScalar * vec2(normalVecInViewSpaceOnlyYaw.x, -normalVecInViewSpaceOnlyYaw.z);

vec2 uvLocalReflections = clamp(uvLocalReflectionsYDistorted + uvLocalReflectionsXZDistortion, vec2(0.0f, 0.0f), vec2(1.0f, 1.0f));

vec4 localReflectionColour = texture(localReflectionsTexture2D, uvLocalReflections);

7. Fresnel Reflectance

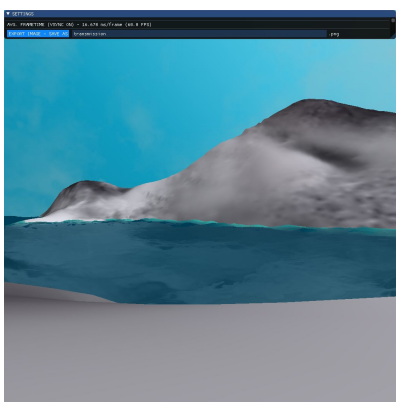
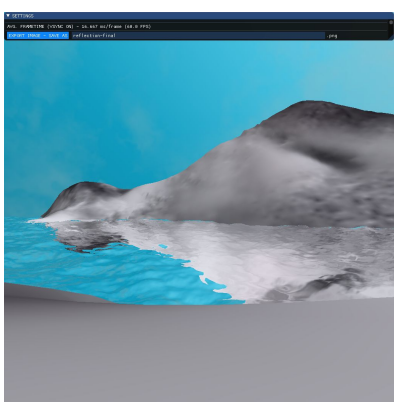
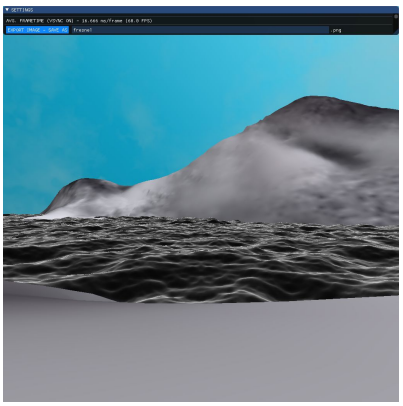




Figure 16 (left) - final transmission layer. Figure 17 (middle) - final reflection layer. Figure 18 (right) - fresnel interpolation values (black means more transmissive, white means more reflective)

```
float fresnel_f_0 = 0.02f;

float fresnel_cos_theta = dot(normal, viewVec);

float fresnel_f_theta = fresnel_f_0 + (1.0f - fresnel_f_0) * pow(1.0f - fresnel_cos_theta, 5);

vec3 waterFresnelTransmissionColour = mix(tintColourAtCurrentTime,
localRefractionColour.rgb, waterClarity * localRefractionColour.a * (1.0f - deltaDepthClamped));

vec3 waterFresnelReflectionColour = mix(skybox_reflection_colour.rgb +
sun_reflection_colour.rgb, localReflectionColour.rgb, localReflectionColour.a);

colour = vec4(mix(waterFresnelTransmissionColour, waterFresnelReflectionColour,
fresnel_f_theta), 1.0f);
```

8. Soft Edges

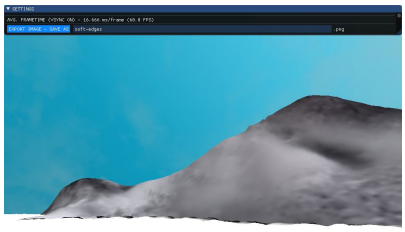



Figure 18 - white means hard surface (alpha = 1.0), darker means alpha fading towards 0.0.

```
float edgeHardness = 0.0f == softEdgesDeltaDepthThreshold ? 1.0f : 1.0f - (hack_skybox_in_back * (1.0f - clamp(deltaDepthClamped / softEdgesDeltaDepthThreshold, 0.0f, 1.0f)));

colour.a = edgeHardness;
```

9. Fog

```
vec4 fogColour = fogColourFarAtCurrentTime;

fogColour.a = viewVecDepthClamped >= fogDepthRadiusFar ? fogColour.a : viewVecDepthClamped <= fogDepthRadiusNear ? 0.0f : fogColour.a * ((viewVecDepthClamped - fogDepthRadiusNear) / (fogDepthRadiusFar - fogDepthRadiusNear));

colour.rgb = mix(colour.rgb, fogColour.rgb, fogColour.a);
```

- **Results**

For the base skyboxes, they were originally loaded in as 4096x4096 textures however the render passes to update the textures proved to be a huge bottleneck on the GPU. They were subsequently replaced with 2048x2048 textures which of course frees a large sum of memory and alleviated the stress on the GPU.

The final demo was capable of running at >60FPS on a 64-bit Windows 10, AMD Ryzen 5 1600 CPU, Radeon RX 570 GPU desktop computer. It was also tested on an older spec surface book and managed to run at ~25FPS.

All things considered the performance goal was achieved, but leaves some room for improvement.

- **Discussion**

The approach of layering effects is promising notably due to the fact that it is extensible for similar effects. For example, implementation of shoreline foam, sand wetting, surface splashes would all consider a depth-based approach for the water-surface to local object intersections.

Due to the layering nature of my approach, an improvement in performance could be achieved for large scenes by transitioning to a deferred rendering pipeline.

There are many more features that could be and are planned to be implemented. These include, but are not limited to, shoreline foaming, underwater effects (e.g. light attenuation, shallow caustic textures), boat wakes, sand wetting on shores, cheap particle texture splashing on steep contact surfaces, buoyancy/rocking of floating objects, improved animation, adding more realism to the skybox, ordered dithering to reduce colour banding, and a transition to a deferred rendering pipeline.

This project served as a great learning tool in improving my knowledge working in an OpenGL environment. I learned how the mere combination of many different visual elements can aid in producing a more natural look and feeling, more so than simply refining one element to be perfect. It further reinforced the idea that compromises must always be made and it is an art more so than a science in finding a desirable balance.

- **Conclusion and Future Work**

In the short-term, I wish to continue implementing the features that I had planned to implement for the project deadline, such as shoreline foam, height mapped terrain, a more populated scenery, etc. Eventually, I believe it would be beneficial to author several blog entries outlining the process to help guide novices, as well as refining the documentation in the GitHub repository. In the long term, I would like to experiment with more advanced techniques such as FFT or small particle simulations. I would also like to overhaul a past video game project to incorporate my water system or even develop an entirely new short game with an ocean surface being the central focus.

- **References**

Claes Johanson, “Real-time water rendering - Introducing the projected grid concept”. Lund University. 2004.
https://fileadmin.cs.lth.se/graphics/theses/projects/projgrid/projgrid-hq.pdf

Mark Finch, Cyan Worlds, “Chapter 1. Effective Water Simulation from Physical Models”. Nvidia GPU Gems. 5th Printing, September, 2007.
https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models

Robert Golias, Lasse Staff Jensen, “Deep Water Animation and Rendering”. GDCE. 2001.
https://www.gamasutra.com/view/feature/3036/deep_water_animation_and_rendering.php?print=1

- **Appendix**

Project files (notably shader code with better specifics) is available in the public GitHub repository:
https://github.com/TexelBox/wave-tool