Rendering Project (by SurRender Your Weapons)

Oscar Czernuszyn u7282674
Harrison Bailey u7285036
Shuang Liu u7602954
Jinghan Gao u7264563

The Australian National University

1 Introduction

For our final project, we implemented a system for the rendering of the motivational image seen in Figure 1 sourced from Quentin Kuenlin's submission in the 2017 EPFL Rendering Competition [11].

We chose this image of two planes flying over water as it provides a slew of complex rendering techniques for us to explore. These include (1) realistic texturing and path tracing, (2) naturalistic lighting of the sky and the sun, (3) water simulation with bumped reflection, (4) cloud simulation and (5) motion blur.

Each of these effects brings its own implementation challenges. For example, rendering clouds requires modeling semi-transparent volumetric media, which can be computationally intensive with Monte Carlo methods. Motion blur, on the other hand, typically involves temporal sampling over moving objects or camera paths, increasing the complexity of other ray generation strategies. We believe that by incorporating all of these techniques simultaneously, the reference provides an appropriate range of challenging problems to attempt to solve, offering the opportunity to push the limits of our computer graphics understanding.

Figure 1: Motivational Image

2 Methods

The primary objects used for this scene were the airplane object and texture [2], the bridge object [1], and the sky texture map [3]. We utilised the CLAB 4 source code as an initial framework to build off, with a few modifications made to allow for scaling and rotating objects in the scene and applying textures using the STB Image library [4].

2.1 Path Tracing

To achieve realistic global illumination, we implemented the path tracing algorithm which estimates the rendering equation using Monte Carlo integration. This algorithm encodes the following approximation:

$$\begin{aligned} L_o(\mathbf{x}, \omega_o) &= L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o)\, L_i(\mathbf{x}, \omega_i)\, (\omega_i \cdot \mathbf{n})\, d\omega_i \\ &\approx L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{i=1}^{N} \frac{f_r(\mathbf{x}, \omega_i, \omega_o)\, L_i(\mathbf{x}, \omega_i)\, (\omega_i \cdot \mathbf{n})}{p(\omega_i)} \end{aligned}$$

where $\mathbf{x}$ is a point in space and $\mathbf{n}$ is the surface normal. We terminate the recursion of the algorithm with Russian Roulette termination, which unlike fixed-depth termination, does not introduce bias [14].

To simulate light's interaction with materials of varying properties, we tested two bidirectional reflectance distribution functions f_r (BRDFs): a modified Phong BRDF for physically based rendering (PBR) [12] and GGX-based Cook–Torrance BRDF [17][19].

For each random Monte Carlo sample w_o, instead of sampling uniformly from the hemisphere, we use importance sampling which samples from distributions more likely to contribute to the overall BRDF of a point [14]. This speeds up convergence by minimising the variance of the estimation.

Phong BRDF. The modified Phong BRDF takes the form of a weighted sum between a diffuse and specular component:

$$f_r(\omega_i, \omega_o) = k_d f_{r,d}(\omega_i, \omega_o) + k_s f_{r,s}(\omega_i, \omega_o) = k_d \frac{1}{\pi} + k_s \frac{(n+2)}{2\pi} (\mathbf{r} \cdot \omega_o)^\alpha.$$

where k_d, k_s are the diffuse and specular coefficients satisfying $k_d + k_s \leq 1$; α is the specular exponent; and $\mathbf{r}$ is the perfect specular reflective direction.

Since this BRDF combines multiple distributions, we use multiple importance sampling which, at each ray bounce, selects one of multiple sampling distributions. For Phong, we randomly choose either a diffuse or specular distribution with likelihood k_d, k_s, respectively. The probability distribution functions for each distribution are

$$\begin{aligned} p_d(\omega_i, \omega_o) &= \frac{1}{\pi} (\omega_i \cdot \mathbf{n}) \\ p_s(\omega_i, \omega_o) &= \frac{\alpha + 1}{2\pi} (\mathbf{r} \cdot \omega_o)^\alpha \end{aligned}$$

We then use the inverse function method [8] to compute the sampled direction in code.

Cook-Torance BRDF. The Cook-Torance BRDF is a physically-based reflection model based on microfacet theory. It is given by the Fresnel-weighted sum:

$$\begin{aligned} f_r(\omega_i, \omega_o) &= (1 - F) \cdot f_{r,d} + F \cdot f_{r,s} \\ &= \frac{1 - F(\omega_o, \mathbf{h})}{\pi} + \frac{D(\mathbf{h})\, F(\omega_o, \mathbf{h})\, G(\omega_i, \omega_o)}{4(\mathbf{n} \cdot \omega_i)(\mathbf{n} \cdot \omega_o)} \end{aligned}$$

where $\mathbf{h}$ is the halfway vector between ω_i and ω_o; D describes the distribution of the surface normals; F is the Fresnel term; and G is the geometric attenuation function which accounts for self-shadowing and masking effects of microfacet surfaces.

As for the choice of D, we chose the GGX distribution (also called Trowbridge–Reitz distribution):

$$D_{\text{GGX}}(\mathbf{h}) = \frac{\alpha^2}{\pi \left[(\mathbf{n} \cdot \mathbf{h})^2 (\alpha^2 - 1) + 1 \right]^2}$$

where α is surface roughness. For F, we use Schlick's approximation [15] given by

$$F(\omega_o, h) = F_0 + (1 - F_0)(1 - \omega_o \cdot \mathbf{h})^5.$$

Finally, for G, we use Disney's approximation to the Smith geometry term for GGX [6] given by

$$G_1(\omega) = \frac{n \cdot \omega}{(n \cdot \omega)(1-k)+k}$$
$$G(\omega_i, \omega_o) = G_1(\omega_i) \cdot G_1(\omega_o)$$

where $k = \frac{(\alpha+1)^2}{8}$. Again, we use multiple importance sampling and inverse function method to compute the direction. The PDF for the diffuse component remains the same as in Phong method, but the PDF for the specular component is

$$p_s(\omega_i, \omega_o) = \frac{D(\mathbf{h})(\mathbf{n} \cdot \mathbf{h})}{4\,(\omega_o \cdot \mathbf{h})}.$$

2.2 Natural Light

Creating a realistic natural lighting environment required splitting the sky into two primary components: the background texture and the sun.

Skydome. The first of these required finding a method of applying a sky texture to some arbitrary point in the distance which we chose to accomplish through a skydome. This involved creating a sphere with a large enough radius that the whole scene could fit inside, then applying a texture to the inside of this sphere and choosing it to be a light emitting material. To apply a texture to the inside of the sphere, the face normals had to be inverted so that the framework knew not to apply them to the outside instead, leading to an additional Boolean parameter being added to the provided `Sphere.cpp` code to determine this orientation.

Sun light. The process for creating the sun was more involved. We split this task into creating the sun itself and the light bloom effect that surrounds it. Creating the physical sun was just a matter of creating a high light emitting sphere near the bounds of the skydome. For the bloom, we utilised the fact that the sun's high emission value meant that most rays that ultimately hit the sun had brightness value greater than 1 (determined by taking the dot product of the pixel's colour and the Rec 709's standard for human brightness perception [18]). This enables us to isolate and process only the brightest parts of the image. After completing the main rendering pass, we applied a post-processing pass to simulate bloom, based on the following equation:

$$B(x, y) = I(x, y) + \lambda \cdot [K(x, y) \otimes \max\left(I(x, y) - T,\, 0\right)]$$

where $B(x, y)$ is the pixel's intensity after the bloom is added, $I(x, y)$ is the prior colour value, λ is a scaling factor controlling the intensity of the bloom, T is the brightness threshold (set by default to 1), $K(x, y)$ is the box filter, and $\otimes$ represents the convolution operator.

This bloom pass involved thresholding the frame buffer to extract the high-intensity regions, then applying a resolution-scaled box blur to the result. This blurred light buffer was then added back to the original image, creating the effect of light bleeding into the sky and bridge while giving a more realistic reflection on the water.

2.3 Water Simulation

In the early stages of the project, we modeled the water as a flat, highly specular plane with a texture map loaded onto it. This approach was provisional and ultimately insufficient for the level of realism we wished to achieve.

To simulate a realistic water surface, we required bump mapping: a method of perturbing the surface normals according to the derivative of a wave function. We tested two such functions: Gerstner waves [9] and Tessendorf waves [16].

Gerstner Waves. Gerstner waves model realistic-looking ocean waves using trochoidal waveforms. The surface height at the horizontal point $\mathbf{x} = (x, z)$ and time t is given by the sum of a predetermined set of sinusoidal waves:

$$h(\mathbf{x}, t) = \sum_{i=1}^{N} A_i \sin(\mathbf{k}_i \cdot \mathbf{x} - \omega_i t + \phi_i)$$

where $\mathbf{k_i} = k_i \mathbf{d_i}$ is a wave vector denoting a direction and speed; $\omega_i = \sqrt{g k_i}$ is the angular frequency from deep water dispersion relation; and ϕ_i is the phase shift.

At render time, we take the derivative of this height function in the x and z directions using the finite difference method to derive the bumped normal.

Tessendorf waves. Tessendorf's method simulates realistic ocean surfaces by summing a large number of linear sinusoidal waves in the frequency domain and transforming them into the spatial domain using an inverse Fast Fourier Transform (IFFT). The surface height at position $\mathbf{x} = (x, z)$ and time t is given by:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t) e^{i\mathbf{k}\cdot\mathbf{x}}$$

where $\mathbf{k}$ is a wave vector and $\tilde{h}(\mathbf{k}, t)$ is the complex amplitude of the wave of vector $\mathbf{k}$. This amplitude is computed as

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k}) e^{i\omega(\mathbf{k})t} + \tilde{h}_0^*(-\mathbf{k}) e^{-i\omega(\mathbf{k})t}$$

where $w(\mathbf{k}) = \sqrt{g\|\mathbf{k}\|}$ and $\tilde{h}_0(\mathbf{k})$ is the initial random amplitude. This amplitude is sampled using a Phillips spectrum $P(\mathbf{k})$:

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P(\mathbf{k})}$$

where ξ_r and ξ_i are Gaussian random variables. The Phillips spectrum defines the energy distribution across wave vectors:

$$P(\mathbf{k}) = A \frac{e^{-1/(\|\mathbf{k}\|L)^2}}{\|\mathbf{k}\|^4} \left(\frac{\mathbf{k} \cdot \hat{\mathbf{w}}}{\|\mathbf{k}\|} \right)^2$$

where A is a constant controlling wave amplitude, $L = \frac{V^2}{g}$ is the largest wave scale based on wind speed V, and $\hat{\mathbf{w}}$ is the wind direction.

In our implementation, we use the Tessendorf model (leveraging the FFTW library [10]) to generate a 256×256 tileable height map as seen in Figure 2 for the water plane, saved as an image. At render time, we load this height map onto the water surface, and take the derivative as before to generate the bumped normal.

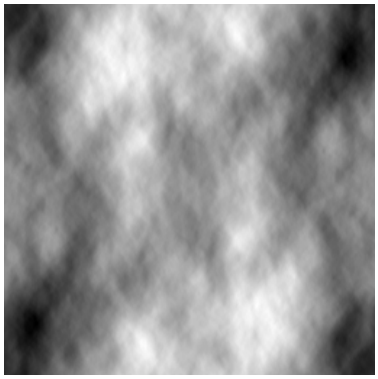


Figure 2: Example Tessendorf height map

2.4 Cloud Simulation

When determining the most appropriate method for creating clouds, three approaches were tested. The first used a OpenGL point sprites with alpha blending [5], creating small circles placed at random positions with random alpha values. The second experimented with ray marching for volumetric clouds, following Sebastian Lague's tutorial [13] and Scratchapixel 's sample code[7].

However, in the end we settled on a particle-based system based on generating a large number of small ellipsoid objects with random sizes, opacities and locations within a set of boundaries, essentially creating our own particle generator. To better emulate both realistic clouds and the ones seen in the reference image, these boundaries were generated in a radial shape with an origin specified by user input along with a direction and length. They were then placed such that half of each cloud was below the water plane to only show the top half and better replicate the reference.

The formula for this radial distribution is defined as follows:

$$\boldsymbol{p}_{\text{final}} = \boldsymbol{c} + \begin{bmatrix} \frac{a}{2}\sin\phi\cos\theta \\ \frac{b}{2}\cos\phi \\ \frac{c}{2}\sin\phi\sin\theta \end{bmatrix} + L \cdot U_3^{0.7} \cdot \frac{\begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}}{\left\| \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \right\|}$$

where c is the center position vector, a, b, c are the ellipsoid radii along the x, y, z axes, respectively, L is the radiation length and the U_i are independent and identically distributed normal random variables. Using these, $\theta = 2\pi U_1$, $\phi = \cos^{-1}(2U_2 - 1)$, $opacity = L \cdot U_3^{0.7}$, d_x is the radiation direction, $d_y = 0.8U_4 - 0.4$ and $d_z = 0.8U_5 - 0.4$.

For each particle in the specified count, the location is determined by the a, b and c radii combined with a randomly selected point along a standard uniform distribution. The specific location of each along the total radiation length is chosen such that the cloud gets less dense the further away it is from the origin, leading to the clouds becoming more transparent around the edges. The size of each particle is then selected randomly and uniformly according to the user input minimum and maximum size with an additional random value selected and applied to the x scale so that they are all slightly horizontally elliptical. This combined with the layered semi-opaque objects leads to the clouds feeling "fluffy" while still allowing some light to come through and thus improving on the static grey clouds seen in the reference.

2.5 Motion Blur

The reference image utilises the illusion of motion blur to give the effect of the camera moving alongside the airplanes and thus make the image feel more dynamic. Specifically, this is done through keeping the airplanes themselves seemingly still while the water has a blur effect applied (to maintain that the camera and airplanes are moving in sync). We realised that while it was technically possible to physically translate both the airplane objects and the camera in the scene, averaging the pixel values in these two states, it was far easier to instead leave everything else static and just "move" the water.

This was accomplished through utilising a time parameter t attached to each cast ray, representing a randomly sampled point within 1 unit of time. This number could then be multiplied with an input representing the speed of the airplanes/camera $\mathbf{v}$ to create an offset of the water's height map texture coordinate, $\varphi = \mathbf{v}t$. Specifically, the offset was only applied to the height of the texture coordinate as this made the blur unidirectional on the z-axis. By taking the average of all of these marginally offset points for each of the rays sampled for each pixel, the water developed a directional blur proportional to the input speed, allowing the user to control the implied speed to achieve their desired output.

3 Experiments and Results

3.1 Path Tracing

As stated previously, we considered two reflectance models for our path tracing algorithm: modified Phong BRDF and Cook-Torrance GGX BRDF. A simplified scene comparison is shown in 3.

The Cook-Torrance model significantly improves the realism of the image. In particular, the perfect mirror reflection of the water is much better captured than under the Phong BRDF, largely due to the inclusion of the Fresnel term which increases the reflectivity of the water at grazing angles. The water under the Phong model, while moderately realistic, is more uniformly reflective, resembling a metallic material more than the ocean surface. The airplane is also brighter on the Cook-Torrance model and better captures the details of the scene in the specular reflection, although we do lose some definition in the ridges on the wings and tail. While the Cook-Torrance model raises the complexity and rendering time, we find this cost to be offset by the gains in realism achieved by accurately modeling microfacet-based specular reflection, energy conservation, and Fresnel effects.

(a) Phong BRDF

(b) Cook-Torrance GGX BRDF

Figure 3: Comparison of sky texture candidates

3.2 Natural Light

When creating a realistic lighting simulation it was important to find an image that best reflected the sky that was used in the reference. Initially we had hoped to create the orange/purple effect through the renderer but after some experimentation, we realised it would be substantially easier without compromising results to just choose a sky texture that already contains this.

(a) Initial sky texture attempt

(b) New sky texture to better emulate reference

Figure 4: Comparison of sky texture candidates

Once we had selected an appropriate sky, we realised it was still important to have some semblance of an accurate sun rather than relying on the texture for this as well to create a better reflection on the water and improve the sun's light blooming effect.

(a) Without sun

(b) With sun

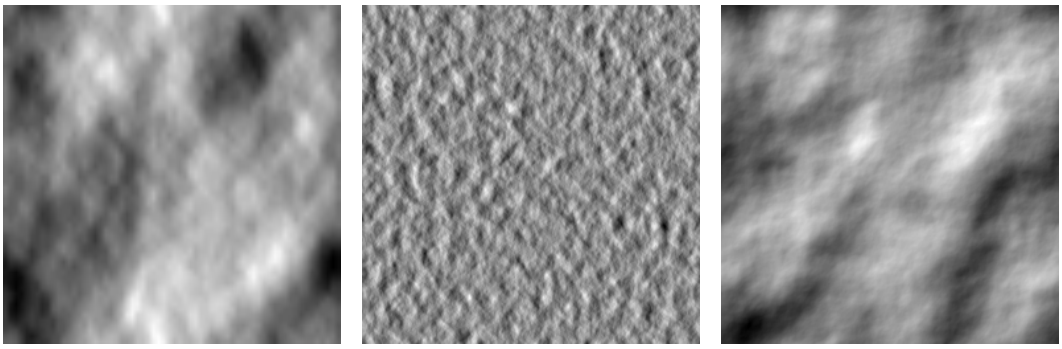
Figure 5: Comparison of skydome with and without an additional sphere object for the sun

There was also an attempt at using an environment map rather than the skydome and while this returned quite similar results in terms of the quality of sky imagery, it did not allow for the atmospheric lighting that the sphere provided. Hence, we chose to remain with the sunset texture sky and sphere object sun combined approach.

3.3 Water Simulation

For the Gerstner method, we tested a variety of wave combinations, designing the individual parameters (including amplitude, direction, phase, etc.) of each wave by hand. Figure 10 shows one such 5-wave configuration.

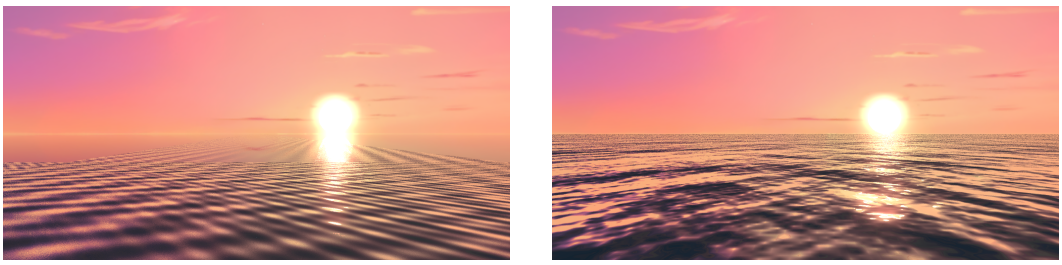
For Tessendorf simulation, the appearance of the water surface is parameterized by the grid size N, the largest wave size L, the wind speed V, the wind direction D, and the horizontal and vertical scale of the bump map when applied to the water surface. Figure 6 shows several configurations.



(a) With grid size of 64×64 (b) With wind speed of 10m/s (c) With diagonal wind direction

Figure 6: Comparison of generated heightmaps

We found that a grid size of 256 with a wind speed of 30m/s in the horizontal direction, and a small vertical scale gave the optimal parameters for water conditions that were not too choppy nor too smooth. The height map generated under this configuration was previously shown in Figure 2.



(a) With Gerstner waves (b) With Tessendorf waves

Figure 7: Comparison of water surface simulation techniques

As seen in Figure 10, the Tessendorf method presents a significant improvement in realism over Gerstner waves. This can be largely attributed to the stochastic initialization of the wave amplitudes and the substantial increase in the number of sinusoids considered (5 vs $256^2 = 65,536$). While the Gerstner approach requires less memory overhead (since it doesn't have to store a height map) and the computation of the Gerstner derivative is much faster, the use of the FFT algorithm in the Tessendorf approach significantly attenuates the increase in total render time. In fact, we found the height map generation to take $\sim$30-40ms on average, which is quick enough to potentially facilitate future real-time rendering projects. On these grounds, we decided that the Tessendorf simulation was the superior approach.

3.4 Cloud Simulation

Accurately rendering clouds is challenging due to their semi-transparent structure and complex geometry. We evaluated three methods for cloud simulation: OpenGL point sprite generator, ray marching, and a particle system. Two approaches were ultimately not suitable for our project, while the third was successfully integrated.

Point sprites. The first technique produced soft, dense cloud shapes in real time however, it was not compatible with our path tracer due to the differences in rendering pipelines. Attempts to use the particle cloud as a PNG texture also resulted in a loss of depth and realism and so this too did not make it into the final render.

Ray marching. The ray marching approach 8 produced realistic volume effects and allowed light to scatter within the clouds. However, while we also observed promising results for this attempt in

a separate environment, integrating this method into our main renderer caused problems, such as clouds obscuring key objects in the scene.

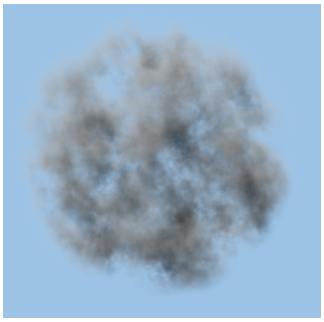


Figure 8: Ray Marching Cloud

Particle system. Our final approach distributed thousands of semi-transparent spheres in a radial formation and thus managed to fit seamlessly into our rendering pipeline while still offering comparable results to fully volumetric approaches. This created soft, transparent cloud edges 9 while still creating a feeling of dynamic "fluffy" objects. While this method is perhaps less physically accurate, it is more efficient and straightforward to implement and still offered results that feel arguably even more realistic than those statically produced in the reference.

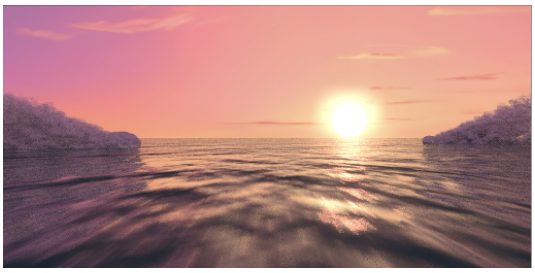
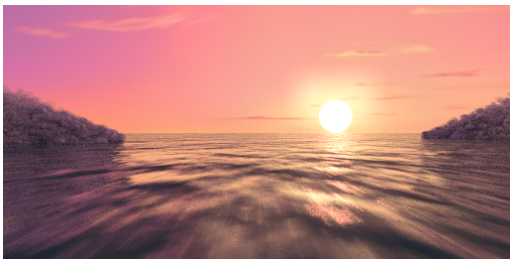


Figure 9: Cloud Effect

3.5 Motion Blur

The only parameter that has a major impact on motion blur is the flow speed which translates to the amount of blur applied to the water. The default flow speed of 0.0008f leads an expected quality of results however as you increase this value the impact on the blur begins to become quite abstract 10.



(a) Flowspeed of 0.001f

(b) Flowspeed of 0.01f

Figure 10: Comparison of water surface simulation techniques

3.6 Overall

The final render for the project, combining all of the aforementioned techniques, was generated as a 1024 x 512 resolution image with a sample per pixel rate of 512. Total render time was 50m 40s using multi-threading and can be seen in Figure 11.

Figure 11: Final scene render. 1024 x 512 resolution, 512 samples per pixel.

4 Conclusion

Through the utilisation of a combination of various computer graphics techniques such as Monte Carlo path-tracing, Tessendorf wave simulation, natural lighting through both atmospheric and point based lighting, particle simulation for clouds and height map offsets for motion blur, we have managed to largely recreate the reference image and even improve on it in some areas.

However, some of the key limitations come in the reduced mirror reflectance on the airplane objects and the predictable shape of the clouds. That is, the airplanes in the reference image manage to combine their texture with a mirror-like surface, leading to a more notable metallic quality that feels more realistic of modern airplanes. The reference's clouds are also slightly more dynamic in their shape, even if they still feel like static objects. We believe that they have imported cloud objects rather than generating them dynamically, allowing greater flexibility in their shape at the cost of them not looking as realistic otherwise. A possible future implementation would have the clouds be generated not according to a radial cone formula, but rather through a preset cloud image that could act as the distribution's pdf.

Additionally, one of the key missing features of our output is the lack of motion blur on the airplane propellors. While the specific airplane models that we chose hide these propellors and hence this is not as detrimental to the final result, the ability to instead show them would lead to a feel of greater motion in the scene and further the dynamic rendering capabilities.

Despite these limitations, we believe that our final result has managed to not only replicate the reference but refine certain aspects further, demonstrating a shift in focus from these missed areas to ones that still work to create a realistic, believable scene.

References

[1] Golden Gate Bridge. https://free3d.com/3d-model/golden-gate-bridge-93300.html, 2014.

[2] Airplane v2. https://free3d.com/3d-model/airplane-v2--659376.html, 2018.

[3] Skybox Clouds 12. https://freestylized.com/skybox/sky_clouds_12/, 2024.

[4] Sean T. Barret. stb library. https://github.com/nothings/stb/tree/master, 2024.

[5] Michal Bubnar. Particle System. https://www.mbsoftworks.sk/tutorials/opengl3/23-particle-system/, 2013.

[6] Brent Burley and Walt Disney Animation Studios. Physically-Based Shading at Disney. In *ACM SIGGRAPH 2012*, pages 1–7, 2012.

[7] Jean Colas. Volume Rendering For Developers. https://github.com/scratchapixel/scratchapixel-code/tree/main/volume-rendering-for-developers, 2022.

[8] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[9] Alain Fournier and William T. Reeves. A Simple Model of Ocean Waves. *Computer Graphics (SIGGRAPH)*, 20(4):75–84, 1986.

[10] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. doi: 10.1109/JPROC.2004.840301.

[11] Quentin Kuenlin. EPFL 2017 Rendering Competition: Flirting with Disaster | Realistic Graphics Lab. https://rgl.epfl.ch/pages/courses/cs440/sp17/competition2017, 2017.

[12] Eric P. Lafortune and Yves D. Willems. Using the Modified Phong Reflectance Model for Physically Based Rendering. In *Compugraphics*, pages 103–120, 1994.

[13] Sebastian Lague. Coding Adventure: Ray Marching. https://www.youtube.com/watch?v=Cp5WWtMoeKg, 2019.

[14] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[15] Christophe Schlick. An Inexpensive BRDF Model for Physically-Based Rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.

[16] Jerry Tessendorf. Simulating Ocean Water. In *ACM SIGGRAPH Courses*, 2001.

[17] Timothy S. Trowbridge and Kurt P. Reitz. Computer graphics and reflectance models. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 455–462, 1975.

[18] International Telecommunication Union. Parameter values for the HDTV standards for production and international programme exchange. In *Recommendation ITU-R BT.709-6*, 2015.

[19] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, page 195–206. Eurographics Association, 2007.