universität innsbruck

igs iis

Visual Computing Project Report

June 12, 2025

Introduction

The project addresses the challenge of rendering realistic reflections in real-time graphics, particularly focusing on water surfaces. While ray tracing offers high-fidelity results, it is computationally expensive and impractical for real-time scenarios. This project explores Screen-Space Reflections (SSR) also known as ray marching as a performant alternative, which reuses information from the screen-space to simulate reflections. The implementation extends an OpenGL application of the 3rd task to include SSR, enhancing the visual realism of the scene with minimal performance overhead. For the basic understanding of screen space reflection techniques, we referred to the online resources given in the project assignment including tutorials by Lettier [2], Virtex Edge Design [1], and the explanation of the linear tracing method by Sugulee [4], in addition to utilizing all relevant lecture slides. Furthermore, the GitHub Repository of Smoliakov [3] helped for the implementation, especially for the binary search refinement.

Background

The realism in computer graphics is often predicated on how accurately lighting phenomena are simulated. One such phenomenon is reflection, particularly on water surfaces, which is challenging due to the dynamic nature of both the reflective surface and the objects being reflected. *The Blinn-Phong illumination model* is a fundamental theory in simulating the specular highlights and is represented by the formula:

$$I_S = k_S M_S I_L (n \cdot h)^m$$

where I_S is the specular intensity, k_S is the specular reflection coefficient, M_S is the specular color of the material, I_L is the intensity of the light, n is the surface normal, h is the half-way vector between the view direction and light direction, and m is the shininess factor. This model is crucial for rendering realistic materials and is a building block for more complex rendering techniques like Screen Space Reflection (SSR).

Method

The method employed in our task was SSR, a technique used to simulate reflective surfaces within the viewable screen space. The implementation involved rendering the scene's objects, capturing their depth and color information, and then projecting their potential reflections on reflective surfaces like water.

The reflection vector r is computed based on the view direction v and the normal at the water surface. The reflection ray is then marched in view-space with a step size, checking against the depth buffer to find a match that signifies a reflection. If the z-depth of the current step is within a certain bias of the object's depth, the reflected color is retrieved from a color buffer.

Our GLSL shader uses the Blinn-Phong model to calculate the diffuse and specular components of the light interaction with the water surface. It also integrates a binary search to refine the intersection point of the reflection ray, enhancing the accuracy of the reflection.

The method is composed of a multi-pass rendering technique. The process is divided into several steps.

Overview of Rendering Passes

1. **Render Scene (Boat Only)**: The first pass involves rendering the boat model into a custom framebuffer. This pass captures both color and depth information, which are stored as textures for later use. The color texture stores the visual representation of the boat, while the depth texture records the distance from the camera to each pixel on the boat's surface.

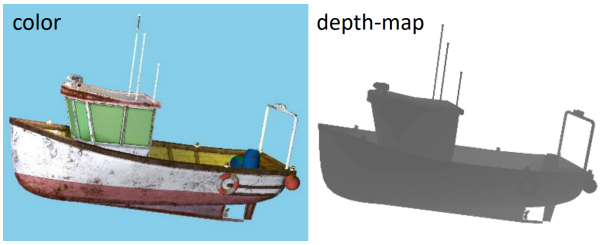

Figure 1: Pass 1 - Render boat into custom framebuffer (color, depth)

2. **Render Scene (Reflective Surface)**: In the second pass, the water surface is rendered. This pass utilizes the previously stored depth information to calculate reflections of the boat on the water's surface by implementing SSR.

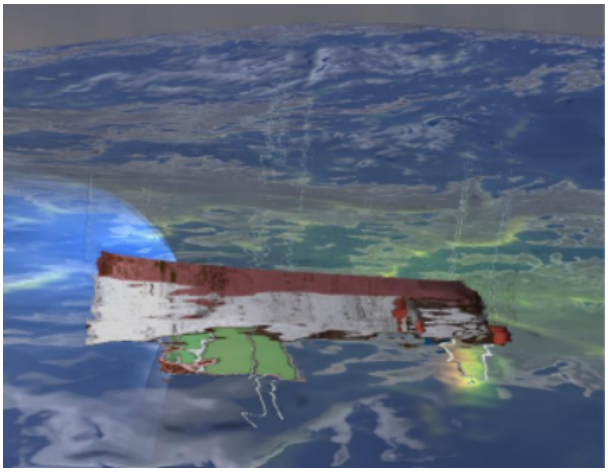

Figure 2: Pass 2 - Render reflective Surface with SSR

3. **Render Scene (Boat Only)**: The final pass involves rendering the boat model again, this time on top of the reflective water surface. The reflections generated in the previous pass are combined with the direct view of the boat to complete the scene.

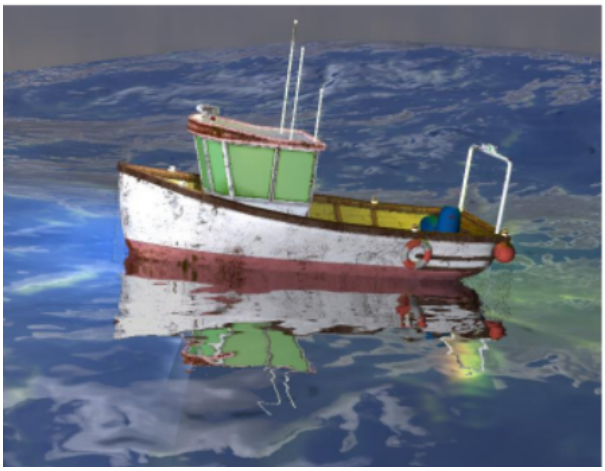

Figure 3: Pass 3 - Render boat into default framebuffer

Framebuffer Implementation

The creation of an Framebuffer object (FBO) involves several steps. Initially, an FBO is generated and bound as the current rendering target. Subsequently, two textures are created and attached to the FBO: one to store color information (color buffer) and another for depth information (depth buffer). These textures are crucial as they store the rendered image and depth data from the scene. We implement a struct which holds the GLInts for the FBO, the color texture and the depth texture. This is done so that we can swap current FBO and then use the textures obtained from the first pass in subsequent passes.

The sizes of the textures are the current window size. If the window is resized, the custom FBO of the scene is deleted and one with the new window size is created. According to some forums online, actually resizing the textures can lead to artifacts.

Both textures use linear interpolation for magnification and minimization. The wrapping mode of the color texture is set to *Clamp To Border* with the color black for both dimensions. Together with the clear color being black, this leads to the sampled color being black except when sampling at a position covered by the boat.

After the textures are configured and attached to the FBO, the default framebuffer is bound and the rendering process can continue using the textures as sources for further rendering steps or effects. This approach is particularly advantageous when multiple rendering passes or complex visual effects are required, as it provides flexibility and control over the rendering pipeline.

The FBO and its associated textures are eventually deleted when they are no longer needed, freeing up GPU resources. The use of FBOs is a cornerstone of advanced OpenGL rendering techniques, enabling the creation of visually rich and dynamic graphics.

Screen Space Reflections (rendering pass 2)

1. **Determine View Direction and Reflection Vector**: The view direction vector (v) is the vector from the camera position towards the position of the fragment in world space. The reflection vector (r) is essential for tracing the path of light as it reflects off the water surface. The reflection vector is calculated using the formula:

$$r = reflect(v, n)$$

where n is the normal at the water's surface, and $reflect$ is a function implemented in GLSL that computes the reflection direction for an incident vector and a normal. Because both vectors are already computed for the Environment

Mapping using a Skybox, we simply reuse the vectors. However, we need to transform both the position of the fragment and the reflection vector from world into view space so that the z-value corresponds to the depth.

2. **Walk Along Reflection Vector**: This step involves a loop where we "walk" or march along the reflection vector for a number of steps starting from the fragment position to find potential intersection points with the boat model. The walk is implemented as a raymarching algorithm in view space.

   (a) **Project Sample to Screen-Space**: The current point on the reflection vector is projected to screen space, obtaining corresponding UV coordinates for texture sampling. By projecting the point into screen space with the projection matrix, we get the normalized device coordinates (NDC) of the point. In order to sample from our precomputed color and depth buffer, we need to transform the NDC, which are between -1.0 and 1.0 for each dimension (after perspective division), into the space between 0.0 and 1.0. By using these transformations, we get the current marching point in UV coordinates (x_s).

   (b) **Retrieve Depth Value from Buffer**: The depth buffer is sampled at the calculated UV coordinates to retrieve the stored depth at the current step on the reflection. The sampled depth is stored as values between 0.0 and 1.0 as the x-value of the depth buffer. In order to compare the z-value of our current marching point ($z(x)$) with the sampled depth of the boat ($\hat{d}(x_s)$), we have to transform the stored depth into view space. First, we calculate the NDC by transforming each coordinate of the vector $(u, v, \hat{d}(x_s))$ into the space between -1.0 and 1.0. We transform the NDC into view space by using the inverse of the projection matrix. At last, we return the z-value of our transformed vector after the perspective division ($d(x_s)$).

   (c) **Retrieve Reflected Color**: If the absolute difference between the depth of the sample ($z(x)$) and the transformed depth from the buffer ($d(x_s)$) is less than a predefined threshold (δ), the reflected color ($c(x_s)$) is fetched from the boat's color texture by using the current UV coordinates. This condition is checked as follows:

$$|z(x) - d(x_s)| < \delta$$

   The retrieved color is then modulated with the water's shininess for simulating the strength of the reflection.

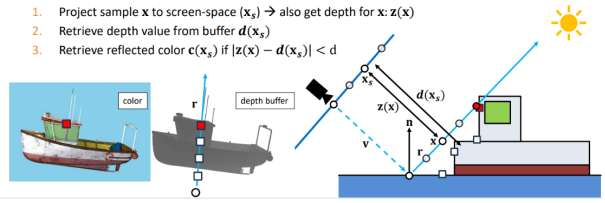

Figure 4: Raymarching

Binary Search Refinement

In our implementation, once an initial hit is detected within a certain threshold (the distanceBias), a binary search with a smaller threshold commences to refine this intersection. The raymarching algorithm itself works the same way as explained before. The only difference is that the step size is halved after each iteration. If there is no hit, the update works as follows: If the depth of the current point on the reflection is smaller than the sampled depth, that means we have not reached the intersection yet and we need to step further along the reflection vector by adding the step to the current position. Otherwise, we take a step back along the reflection by subtracting the step. This is critical for avoiding artifacts in the rendered image that can occur if the intersection point is not accurate, especially at sharp angles and edges where the reflection needs to transition smoothly between different surfaces (see Fig. 5).

Results

The scene set for the experiment was a simple model of a boat floating on water under a simulated sun. The boat's reflections on the water surface were rendered using SSR, and the visual fidelity was assessed.

Visual results showed that SSR could produce convincing reflections of the boat on the water surface. The reflections included both static aspects of the boat and dynamic changes in the water surface due to simulated environmental conditions. The performance was measured in terms of GPU runtime per frame, demonstrating the method's real-time capabilities. We used an approach where for each frame the elapsed time from the previous frame is printed. Because of this, we don't have such a significant decrease in performance due to busy waiting until the elapsed time can be queried. On our test architecture with an AMD Radeon RX 6900 XT graphics card we achieved an average GPU time of about 4 ms per frame. The biggest outliers are around 9 to 10 ms, maybe due to context switches during execution. The

mode is 3.4 ms. By changing the parameters of the linear search (doubling maxSteps, dividing stepSize by 4 and adapting the threshold accordingly) the reflections are much smoother but the average GPU time rises to 5 ms per frame. Interestingly enough, the GPU time for the more accurate version is much more stable so that the deviation usually lies at 0.2 when the scene is at rest.

Figure 5: With Binary Search (Top) - Without Binary Search (Bottom)

Conclusion

The SSR technique was successfully implemented, demonstrating its efficacy in rendering realistic reflections in a dynamic scene. Challenges included managing the computational complexity to ensure real-time performance and fine-tuning the binary search for intersection refinement.

Limitations of the method include its reliance on screen-space information, which means reflections of objects not currently rendered on the screen cannot be simulated. Furthermore, because only the visible

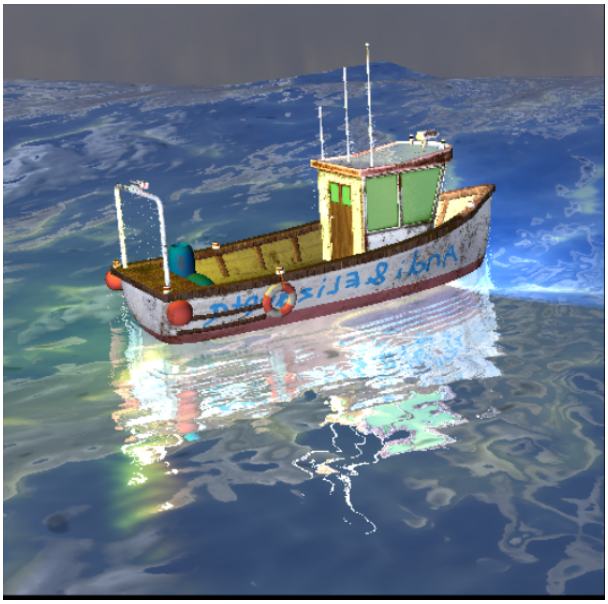

Figure 6: End result

parts of the reflected objects are rendered there are problems when the reflection would in reality hit a part of the object which is not rendered because it is covered by other objects or the object itself. In our case, this happens when the view direction is very steep. That way, the inside of the boat is reflected on the water surface even though it should reflect the outside (see Fig. 7). This is an inherent problem of SSR. Improvements could include integrating the SSR with other global illumination techniques to enhance the overall lighting simulation, as well as optimizing the shader code to improve performance on less powerful graphics hardware.

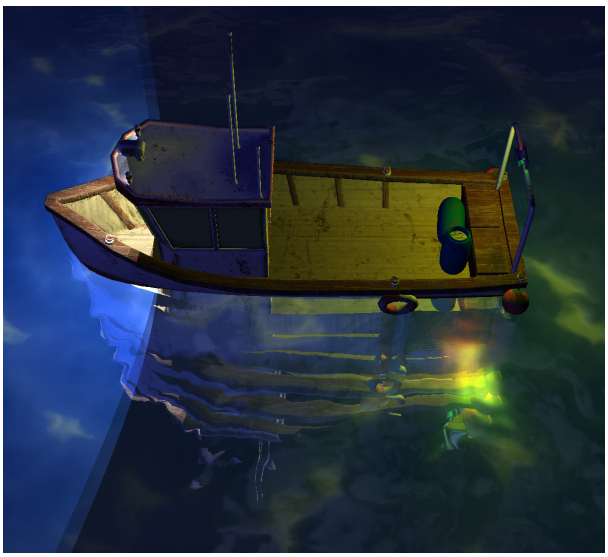

Figure 7: Wrong reflection due to steep view direction

References

[1] Virtex Edge Design. Shader series: Basic screen space reflections. `https://virtexedge.design/`

shader-series-basic-screen-space-reflections/, 2019. Accessed: 2024-01-22.

[2] David Lettier. 3d game shaders for beginners. https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html, 2020. Accessed: 2024-01-22.

[3] Daniil Smoliakov. ScreenSpaceReflection. https://github.com/RoundedGlint585/ScreenSpaceReflection, 2021. Accessed: 2024-01-22.

[4] Sugulee. Performance optimizations for screen space reflections technique part 1: Linear tracing method. https://sugulee.wordpress.com/2021/01/16/performance-optimizations-for-screen/-space-reflections-technique-part-1-/linear-tracing-method/, 2021. Accessed: 2024-01-22.