universität innsbruck

igs iis

Refraction & Screen Space Reflection Project Report

David Kranebitter, Matthias Feichtinger & Dominik Schäfer

February 14, 2023

Introduction

The task of our project was to implement Refraction and Screen Space Reflection in OpenGL. What happens in Refraction and Screen Space Reflection is, that light is refracted/breaked on reflecting surfaces like for example water.

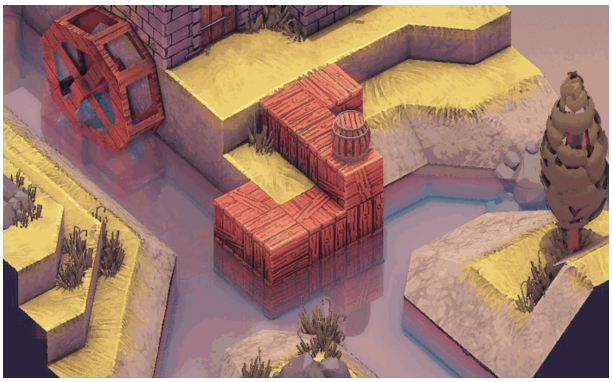

Figure 1: An example how Screen Space Reflection on water looks like [3]

This requires the use of deferred shading techniques and ray-marching to compute reflected texture coordinates along a reflected ray.

To accomplish this, we used two distinct shader programs to first render the scene and all necessary components into a set of textures, then pass those textures along to perform two passes of raymarching with differing levels of detail.

Ultimately, our project did not produce the desired results, though it is capable of displaying discernible reflections.

Background

Ray marching

Ray marching is a method used in screen spaced reflection to trace a reflected ray until it collides with a surface or reaches a predetermined number of iterations.

Every pixel on the screen goes through this procedure once, and the resulting reflections are then combined to create the final reflection color.

In order to progress the ray, the ray marching algorithm samples the distance along the ray to the closest surface at each iteration. The distance function of the scene, which calculates the distance between a given point in space and the closest surface, is used to accomplish this. Every time the ray iterates, the distance function is evaluated and the ray is moved by the calculated distance.

It can be very expensive to perform ray marching, especially for complicated scenes with plenty of reflective surfaces, so therefore we use a modified version of the the Bresenham algorithm.

This algorithm is used to efficiently iterate through the depth buffer in screen space along the path of the ray. To choose which way to step on the x and y axes, the algorithm increments the pixel coordinates and a set of decision variables (z-axis).

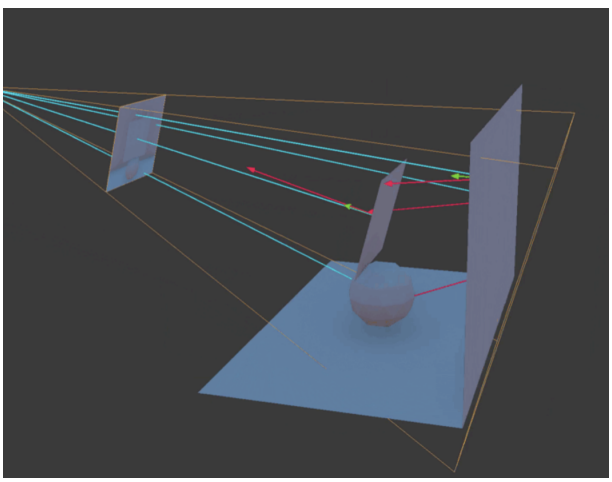

Figure 2: How ray marching looks like [3]

Bresenham algorithm

This algorithm is used to efficiently iterate through the depth buffer in screen space along the path of the ray.

For the algorithm there is the Line Representation from where the algorithm increments. For this the line equation and the implicit line equation is needed. For example the implicit line equation:

$$\begin{aligned} f(x,y) &= \begin{pmatrix} y_1 - y_0 \\ x_0 - x_1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + c = 0 \\ &= n^T \begin{pmatrix} x \\ y \end{pmatrix} + c = 0 \end{aligned} \tag{1}$$

Figure 3: The implicit line representation

To choose which way to step on the x and y axes, the algorithm increments the pixel coordinates and a set of decision variables. (z-axis) For incrementing there must be determined d_{i+1} from d_i.

In case of E we use

$$d_{i+1} = d_i + \Delta y \tag{2}$$

Else in case of NE we use

$$d_{i+1} = d_i + \Delta y - \Delta x \tag{3}$$

The algorithm then checks the depth buffer at each step to see if the ray crosses any scene elements.

The reflection computation then uses the nearest intersection.

Deferred shading

Screen spaced reflection makes use of a technique called deferred shading, which is quite useful as it keeps the computation costs low.

In normal forward shading the shading computation has to repeated for every object in the scene and for every light source affecting that object. On the other hand, deferred shading divides the rendering process into two distinct passes, which greatly improves the rendering performance in scenes.

In the first pass, the objects properties like normal, position and albedo are rendered into the G-buffer.

The G-buffer is a collection of buffers which store various properties of the objects in the scene.

In the second pass, the lighting contribution from each light source is calculated for each pixel and the results are combined to create the final color.

As the lighting computations are done when the objects have already been rendered, deferred shading can manage several lights in a scene much more effectively than forward shading.

Method

At the beginning of this project, we designed a chart, which shows the basic process of how screen spaced reflection can be implemented using deferred shading.

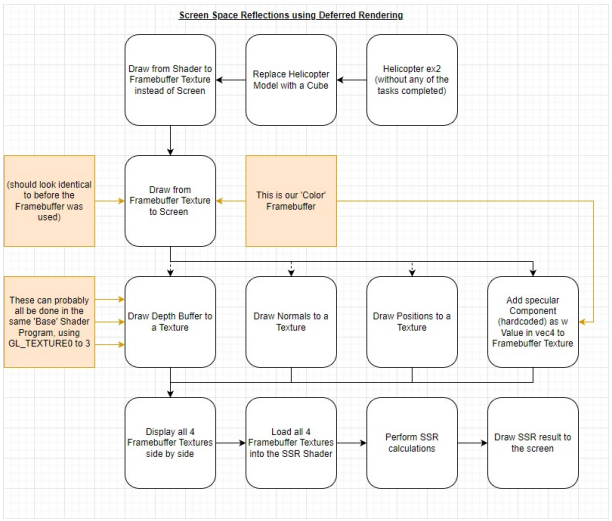

Figure 4: Screen Space Reflections using Deferred Rendering

Our plan was to start with second assignment and to replace the helicopter model with a cube model.

As this project was quite time consuming and challenging, we decided to skip the part, where we would change the model and keep the helicopter.

At the end it wasn't such a bad idea , because the rotation of the helicopter rotor made it easier to debug the SSR shader.

So the next step was to setup the G-buffer where we store our normals, position, color specularity and depth. [5] [4]

The framebuffer textures were then draw to a screen-sized quad, using code provided in the Proseminar [2]

By correctly implementing these steps mentioned above, we should end up with the same result as with

the template code of the second assignment.

After we verified this behavior, we continued by passing the information stored in the G-buffer to gShader which draws each buffer to a texture.

We then displayed each framebuffer texture side by side on the screen, to ensure that everything looks fine.

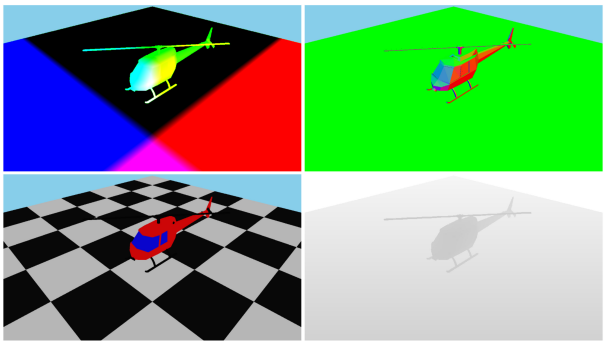

Figure 5: Framebuffer

We proceeded by loading each texture into the SSR shader. The SSR shader performs these calculations in 2 separate passes of increasing granularity, provided a fragment is capable of reflection.

The first pass uses our implementation of Bresenham's line algorithm [1] to march along the ray in screen space, comparing calculated depth values with values fetched from the position texture.

This should have been done through the depth buffer, which ended up producing no reflections. If a hit is found during the first pass, the shader then performs a binary search around the point determined in the first pass, increasing the accuracy of the displayed reflection.

If a hit is determined during this second pass, the resulting uv coordinates are used to determine a reflected fragment's color from the color texture, which is then added to the fragment's final color.

Results

For the implementation of SSR, we have elected to use a reflective ground plane, on which a reflection of the animated helcopter model should be visible. Our project is now capable of displaying wildly incorrect, but still discernible reflections.

Current reflections include the scene's background color, the helicopter's blue and red colorations and the distinct movement of the helicopter's rotor. These results are not satisfactory to us, but they are all we were able to produce until now.

The SSR shader's current faults are likely due to some incorrect use of projections or our failure to properly integrate the depth texture, generated in gShader.

To supplement this lackluster conclusion, we include some images, chronicling our latest attempts at displaying accurate reflections.

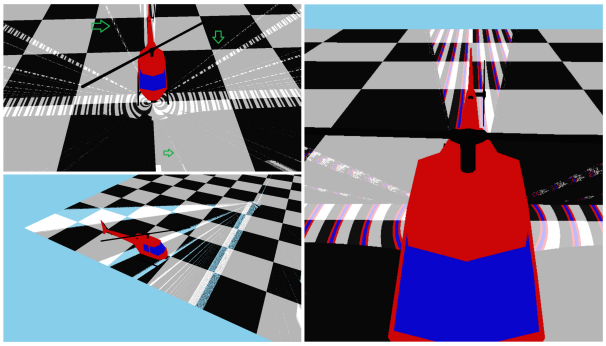

Figure 6: Displayed Reflections with Circular Anomaly

These images show our project displaying all three of the aforementioned reflections, with green arrows denoting the reflections of the spinning rotor blades.

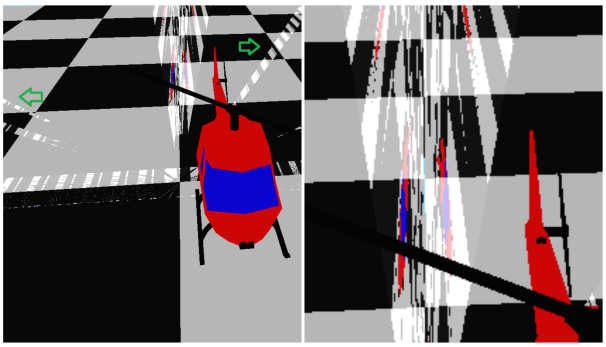

Figure 7: Displayed Reflections after Deprojecting Normals

These images show our project, after adding a deprojection of input normals, removing the reflections' strange circularity. Zooming into the left image reveals a decent reflection of our helicopter, stretched and repeated multiple times.

Conclusion

We've tried multiple times to complete the project and ran into many problems while doing that. Therefore we started from the beginning four times.

The first attempt
At first we tried to follow the "3D Game Shaders For Beginners: Screen Space Reflection (SSR)" [3] page because we thaught it was a great tutorial for our project. We built it on our OpenGL exercise number 3 and we came to the problem that there are many aspects that are not easy to understand and are not covered in the Visual Computing lectures. For example cell shading. Also this tutorial had used 10 shaders, which are also many.

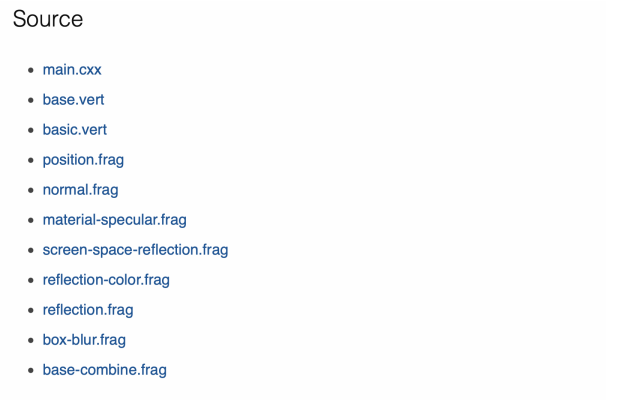

Figure 8: The many sources this website has [3]

The second attempt
After that we tried using the Tiny object loader for our project. The problem was that it is hard to use with our project and we tried to understand the material properties there.

The third attempt
At the third attempt we tried understanding the whole problem and the concept of Screen Space Reflection and then there was the presentation where we had explained the problems of Screen Space Reflection but unfortunately we did not finished our project.

The final attempt
At the final attempt we started from the second helicopter task from the proseminar and we built our project with the tips that have been told to us.

References

[1] Rosetta Code. Bitmap/Bresenham's line algorithm, 2023. https://rosettacode.org/wiki/Bitmap/Bresenham%27s_line_algorithm.

[2] Proseminar Visual Computing. 09framebuffer, 2023. https://lms.uibk.ac.at/auth/RepositoryEntry/5305860954/CourseNode/106545021962578.

[3] David Lettier. 3D Game Shaders For Beginners: Screen Space Reflection (SSR), 2019. https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html.

[4] OGLdev. Deferred Shading - Part 1, Unknown. https://ogldev.org/www/tutorial35/tutorial35.html.

[5] Learn OpenGL. Deferred Shading, Unknown. https://learnopengl.com/Advanced-Lighting/Deferred-Shading.