CGP - Mini-project: Water Shader

Jannick Drews

December 8, 2019

1 Abstract

2 Introduction

A water shader can be built in many different ways, this report will cover how I developed my shader, the challenges I faced and how I reached my goals. To start with I was looking for a unique way to develop the water shader, usually the shader is made using a bump or normal map for the water itself, in this project I was aiming for using voronoi as the bump map.

3 Theory

This section will cover all theory related to the shader. This will include; Voronoi, bump maps, reflections and refractions, fragment and vertex knowledge, fresnel effect, transparency, culling, z-buffer and grabpass.

3.1 Fragment & Vertex

The **fragment** and **vertex** shaders are both shaders but they work in different spaces. The vertex shader works on the local space, object space and afterwards converts the space to the clip space. The fragment shader works by modifying the mesh on the clip space, which will finally be convert to screen space.

3.2 Voronoi

The voronoi effect is made by having, typically a grid of 9 cells which each have a random point, and from that point a distance is calculated to each other cell. Depending on that distance, each pixel will be colored with less and less saturation. You could argue that it is a radial colouration that happens, colouring outwards from the center point.

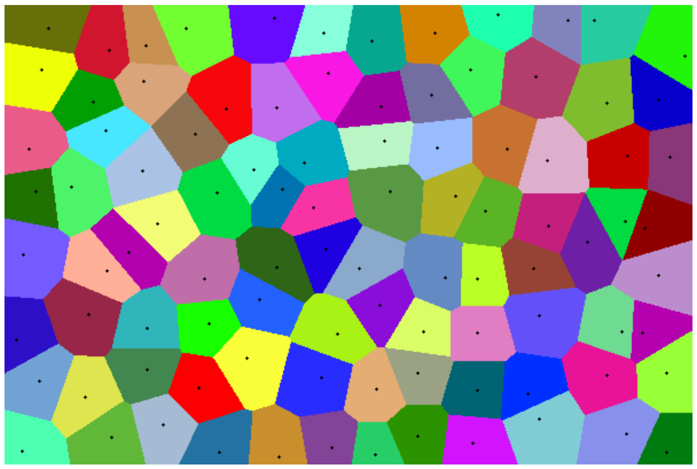

Figure 1: Voronoi[1]

Figure 1 shows the voronoi effect. Each cell is uniquely coloured to better showcase the effect, the voronoi effect is usually coupled with a time variable with a sinus or cosinus function applied to make each cell move with time passed.

I've used this effect on the vertex shader, applying it in object space as to generate a bump map which I utilize to visualize the waves on the object. The voronoi affect is applied via the normals of the mesh.

3.3 Bump maps

Bump maps in shaders is utilized by having a greyscale image or texture tell where the object should modify the verteces positions. Ergo translating the position of the verteces to a new position after shader application. This can be applied to either the vertex or fragment shader respectively.

3.4 Reflections & Refraction

3.4.1 Grabpass

3.5 Fresnel effect

The Fresnel effect is blablabla..
It can be used to visualize reflective objects color, when viewed at a critical angle. The effect is a bright surface, which will form around the object, no matter the camera angle, and is applied in clipspace.

[1]`https://www.codeproject.com/KB/recipes/882739/Fig_1.png`

4 Description & Analysis

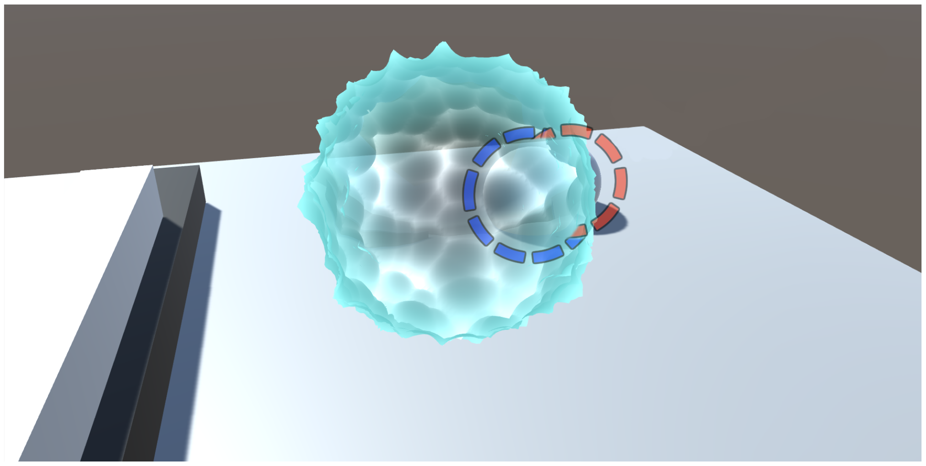

Figure 2: Shader - Refraction illustration

Figure 2 shows a capture of the scene with the shader applied to a sphere. The effect of the Grabpass can be seen as a substitude for refraction, see Reflections & Refraction 3.4 & Grabpass 3.4.1.

The red indicator on the figure shows the original real position of a sphere which is placed behind the object with the shader applied. The blue indicators denote the distortion of the image with the new location of the object being refracted. Technically, the shader does not have any transparent properties, instead a Grabpass is used which is discussed in Grabpass 3.4.1.

This gives a better refraction of the object(subjective), compared to when cubemap refractions are used. Since cubemaps require an object to be in a static position in space or constantly update a cubemap image, the grabpass is a faster and more "headache free" alternative.

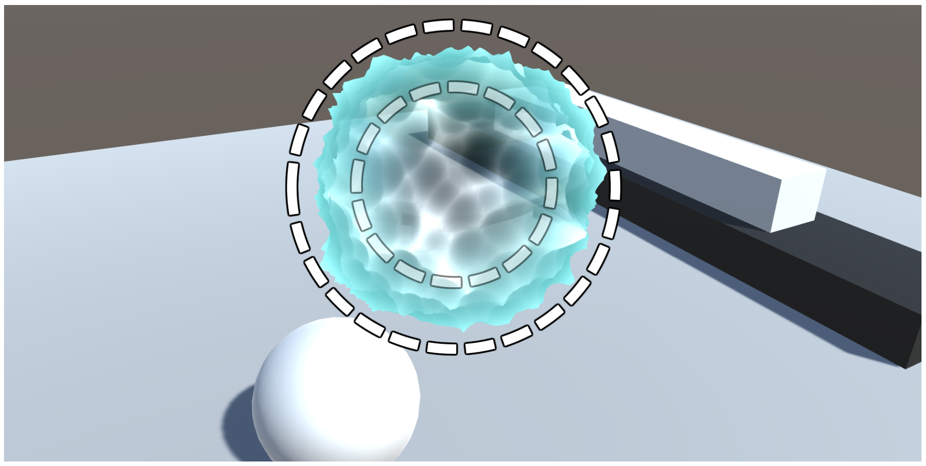
Figure 3: Shader, fresnel effect and additive bumpmap

Figure 3 shows the fresnel effect with some added extra colours from the predefined color of the shader and the voronoi bump map colors.

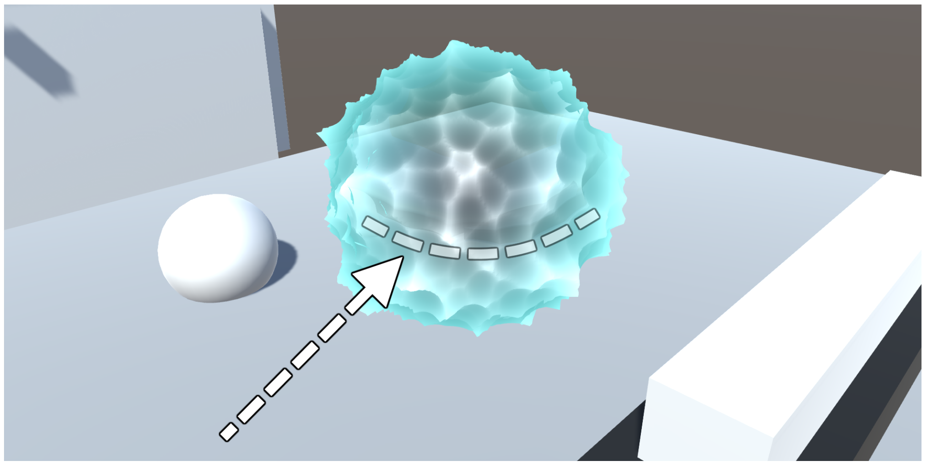
Figure 4: Shader

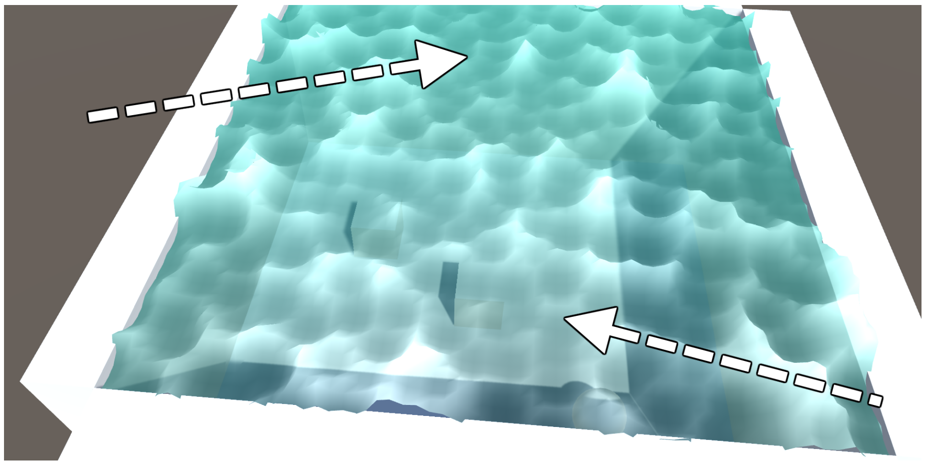
Figure 5: Shader

```
 1  for (int y = -1; y <= 1; y++) {
 2      for (int x = -1; x <= 1; x++) {
 3          for (int z = -1; z <= 1; z++) {
 4              float3 neighbor = float3(float(x), float(y), float(z));
 5              pointt = random2(i_st + neighbor);
 6              pointt = 0.42 + 0.42*sin(_Time.y + 6.2831*pointt);
 7              float3 diff = float3(neighbor + pointt - f_st);
 8              float dist = length(diff);
 9              m_dist = min(m_dist, dist);
10          }
11      }
12  }
```

Figure 6: Voronoi - code implementation in object space

Figure 6 shows a code snippet from the voronoi implementation. Usually this is used on 2D textures, but since we want to convert it to 3D space we need to do a 3D grid and not a 2D. This is however not needed on a plane or a mesh that which has the same normal vector across the polygons.
This code is a very standard or conventional way of implementing voronoi features, besides the additional for-loop on the z axis.

Line 4 to 9 is the core of the function, serving to grab the neighbouring grid to later use as an offset. The `pointt` variable will then hold a random position and have that position modified by time and arbitrary numbers.
The distance difference will then be calculated and used for colouring the cell or grid square, compared to other points in neighbouring grid cells.

5 Discussion

6 Conclusion

7 Bibliography