Game Development Practical Report

Zhengjiang Hu

30.07.2019

Table of Content

1. Introduction

Our team is ZGL. We have developed together a first-person portal game. In this game, the player is in a maze where there are portals on the walls of the maze. The portal can teleport players to different locations of the maze that are not spatially connected or even different maze that are physically far from each other. The main features of this game include the generation of maze, rendering of the portal's view, recursive portal, i.e. portal inside portal, and several techniques of the rendering pass.

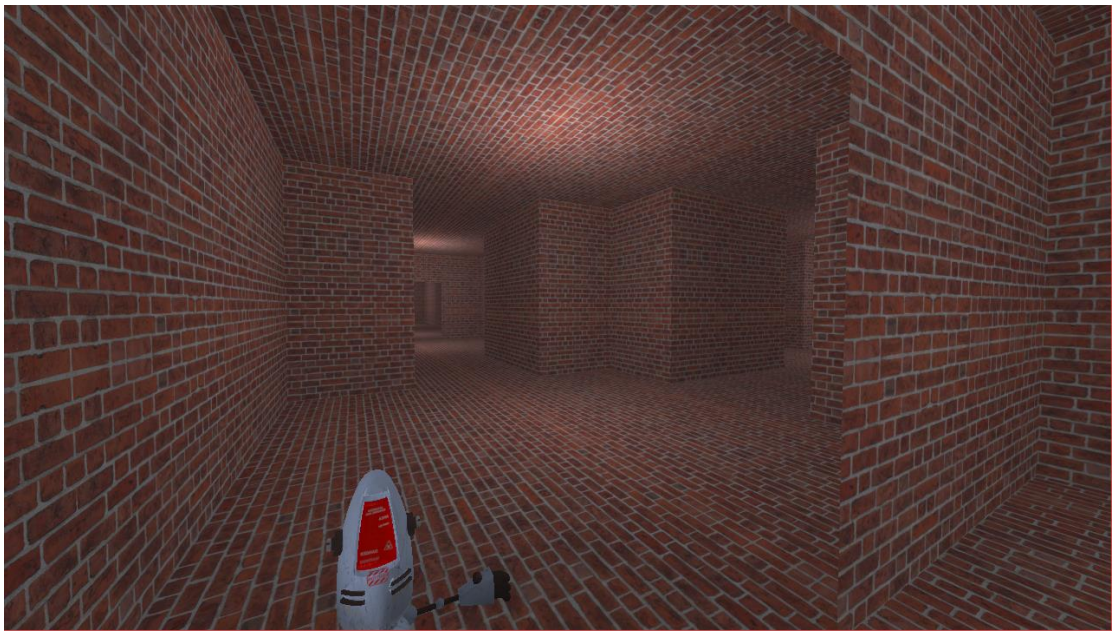

Figure 1 Portal Maze Game

While the other team members focus mainly on the physical effect, basic game framework and portals, I focus mainly on the techniques used for rendering pass. In our game, I have implemented some features including deferred rendering, depth clearing of specific area, oren-nayar diffuse reflection, cook-torrance specular reflection, FXAA and so on. There is also something on which I invest time but did not work out at the end such as clustered shading.

1.1. Deferred Shading

In this game project, we decided to choose deferred shading, which is the fundamental feature of modern real time rendering, as our basic pattern of the rendering pass. The reason why we chose deferred shading as our rendering pass technique is mainly due to the advantage that deferred shading separated the native rendering pass into 2 different rendering passes which can increase the performance.

The traditional forward pass has a big disadvantage that it will compute the

lighting effect for whatever is passed from the vertex shader, even those that are actually not necessary to be rendered. This disadvantage will increasingly affect the performance as there are multiple light sources in the scene.

Deferred shading, on the other hand, aims to react to this disadvantage by separating the whole rendering pass into two sub-passes which are called “Geometry Pass” and “Lighting Pass”. As the name of the technique indicated the lighting computation is deferred to the second pass. This technique takes advantage of the feature of the rendering pipeline itself such as depth culling. Therefore, in DS we can firstly render all the geometry data to textures which are also called geometry buffer in the first pass and then do the lighting calculations only on data stored in geometry buffer. Since the G-buffer only stores front most objects after the depth test, we have avoided problem of rendering unnecessary objects that are unable to be seen from the camera which increases the performance a lot.

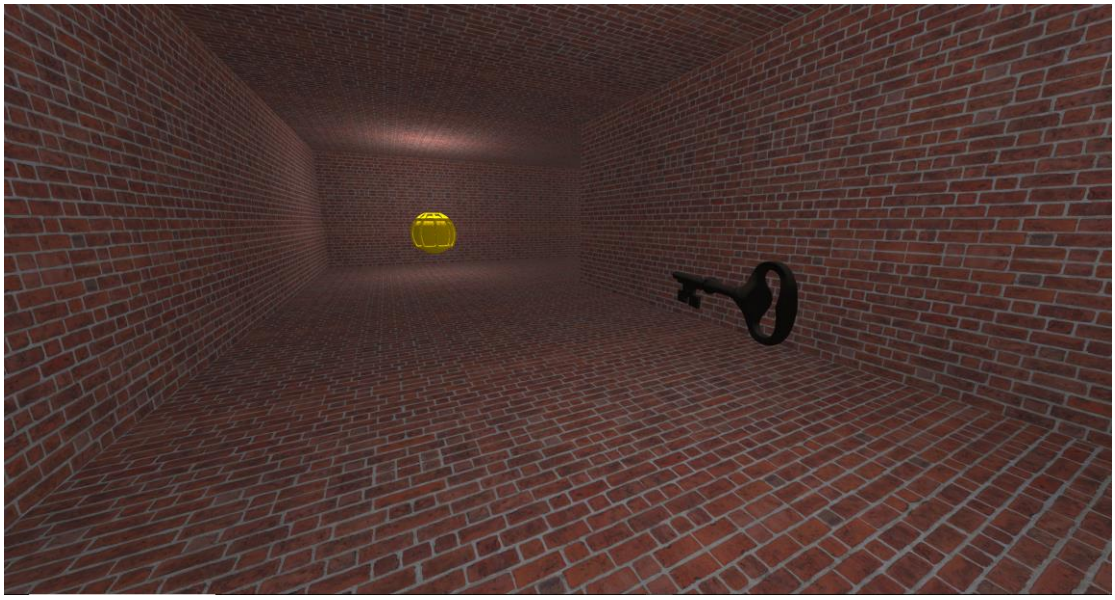

Figure 2 The effect of deferred shading. One can see no difference from forward shading but DS has dropped unnecessary objects in geometry pass before doing lighting calculations.

To implement deferred shading in our game, I have decided to build two pairs of shaders which will separately handle the geometry pass and lighting pass. During the initialization of the rendering, I created several color textures for storing the necessary geometry data such as normal, position and albedo. I also created one depth-stencil buffer for storing the information of depth and stencil since the portal needs stencil buffer to be functioning. After attaching those textures to the framebuffer, the initialization is basically done.

During the geometry pass, I let the vertex shader of the geometry pass only perform necessary space transformation and then pass the data to fragment shader. The fragment shader does nothing else but some basic processing and

then packing the data into the textures bound to the framebuffer. Further, it will automatically drop the fragment that failed passing the depth test. At the end, those textures only store the information of most front objects in the scene from the point of view.

During the lighting pass, firstly I drew a full-screen quad onto the screen which kind of serve as a canvas. By drawing the quad, the fragment shader interpolates the position pixel by pixel covering the whole screen space so that I can give information to every fragment of the scene. Secondly, I read and unpack data from the textures in the previous pass and do all the necessary lighting computation such as oren-nayar and cook-torrance reflections.

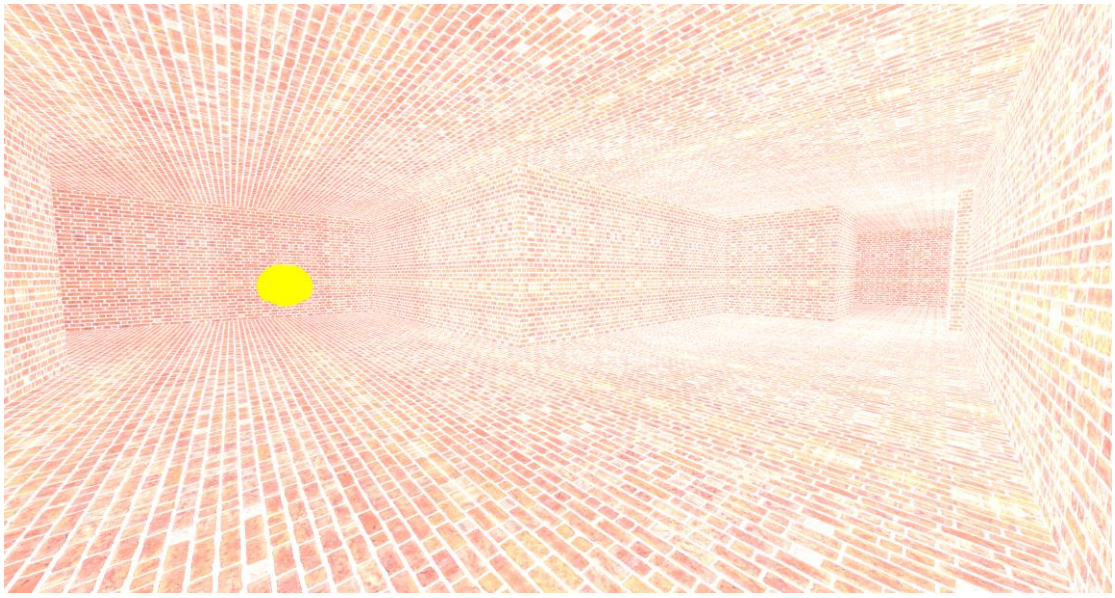

Figure 3 The texture of albedo data

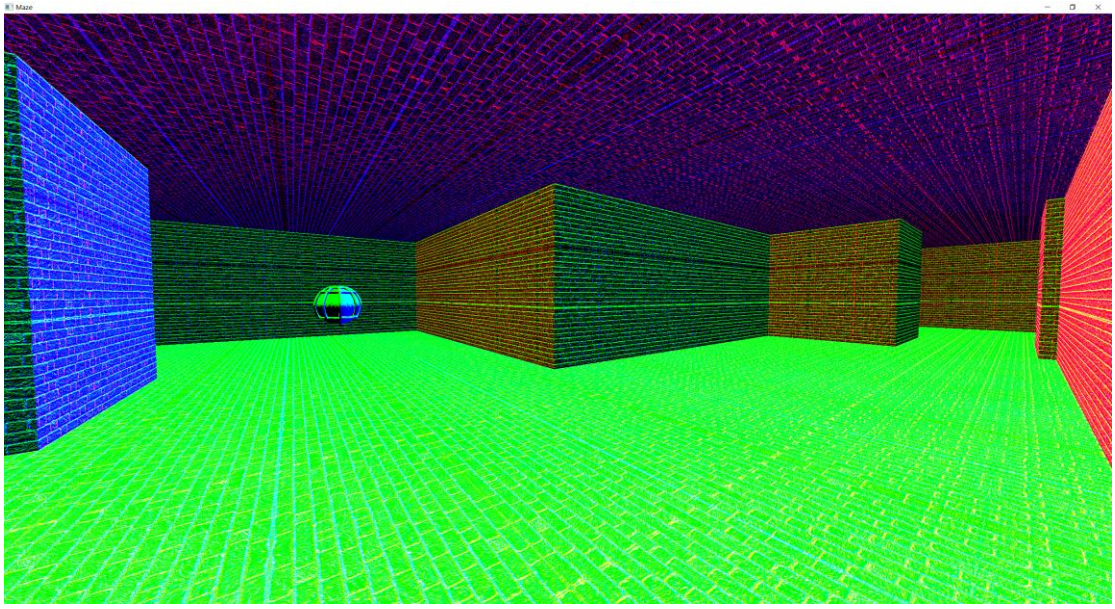

Figure 4 The texture of normal data

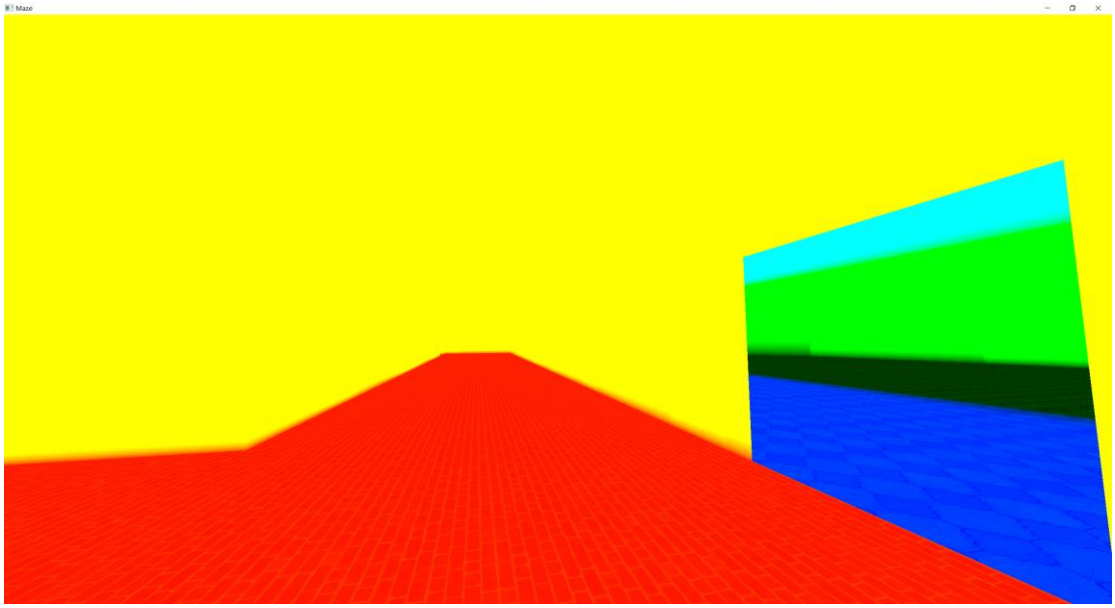

Figure 5 The texture of position data

As shown in figure 3, 4, 5, the textures separately store the geometry data in the geometry pass. The lighting pass use these data to perform lighting which yields the final result shown in figure 2.

1.2. Depth clearing of specific area

I implemented a small technique which helps to clear the depth value of a certain area. During the early stage, I also helped implementing the basic portal rendering. Our portal firstly uses stencil buffer to mark the area where the view of portal appears. Then the content of the portal view is rendered into the marked area. However, before rendering it into the given area, the depth value needs to be cleared first otherwise it will mix up with content of the normal scene. We found the problem that the depth clearing command built in OpenGL can only clear the whole depth buffer and it often caused the normal scene to be flushed which end up with only having the rendering of the portal view but no content of the normal scene left.

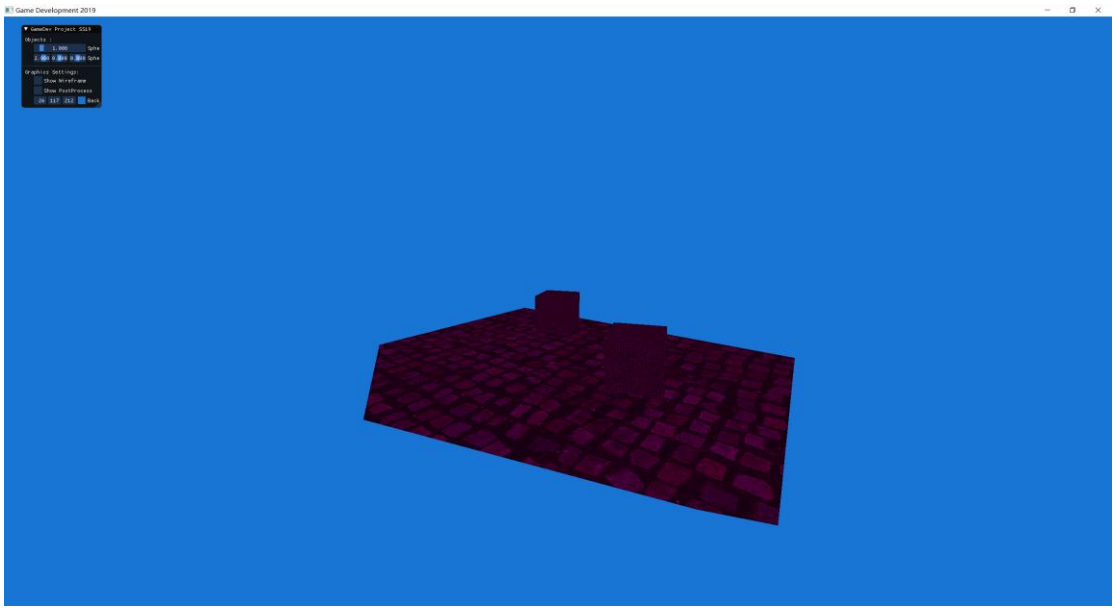

Figure 6 The normal scene is flushed by the background color after calling clear depth buffer bit. This result specifically depends on our implementation of the rendering and there maybe no such problem if the implementation is different.

Therefore, we do not want to clear the whole depth buffer. By doing this, I played trick with stencil buffer like what we did for portals. Before rendering the view of the portal, I created another shader to draw a full-screen quad. During the rendering of the quad, the shader writes specific depth value (1.0 in our case) to the built-in gl_FragDepth which will subjectively overwrite the value stored during depth testing. Due to the marking of the stencil buffer, only the area of the portal will actually update the depth value. At the end we achieve the same effect as clearing buffer bit without flushing the normal scene.

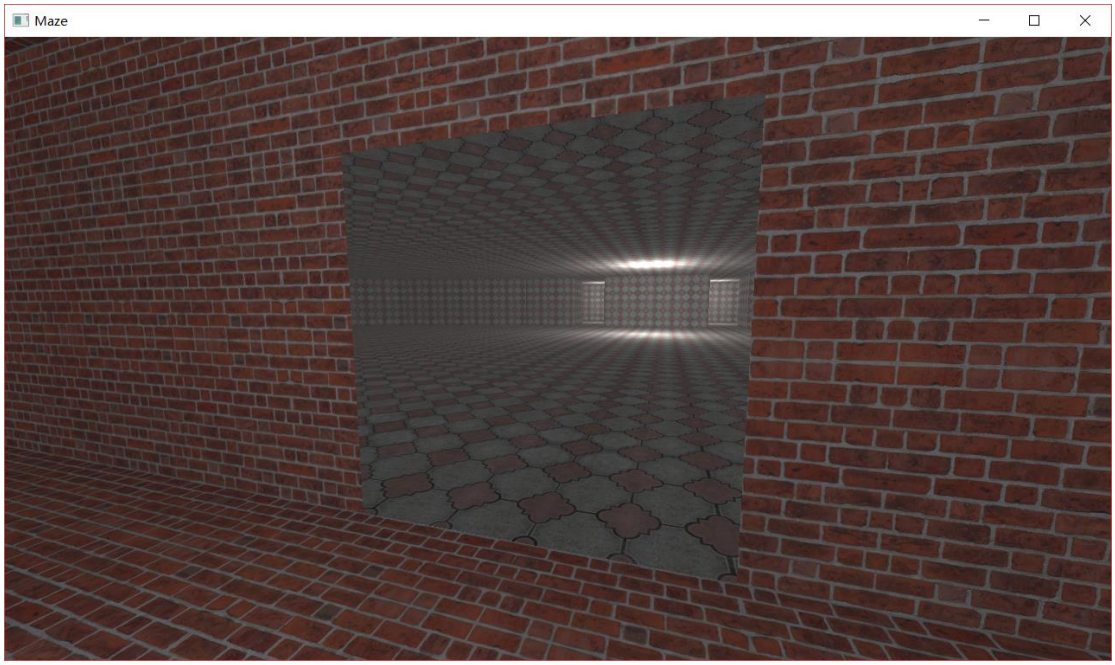

Figure 7 The view of the portal is correctly rendered, depth tested and the normal scene is also reserved.

1.3. Outline of the portal

I also implemented adding outline to objects. Here, in our case, the portal. We finally decided not to use it because we wanted to make the portal less noticeable to confuse the player but the function is implemented.

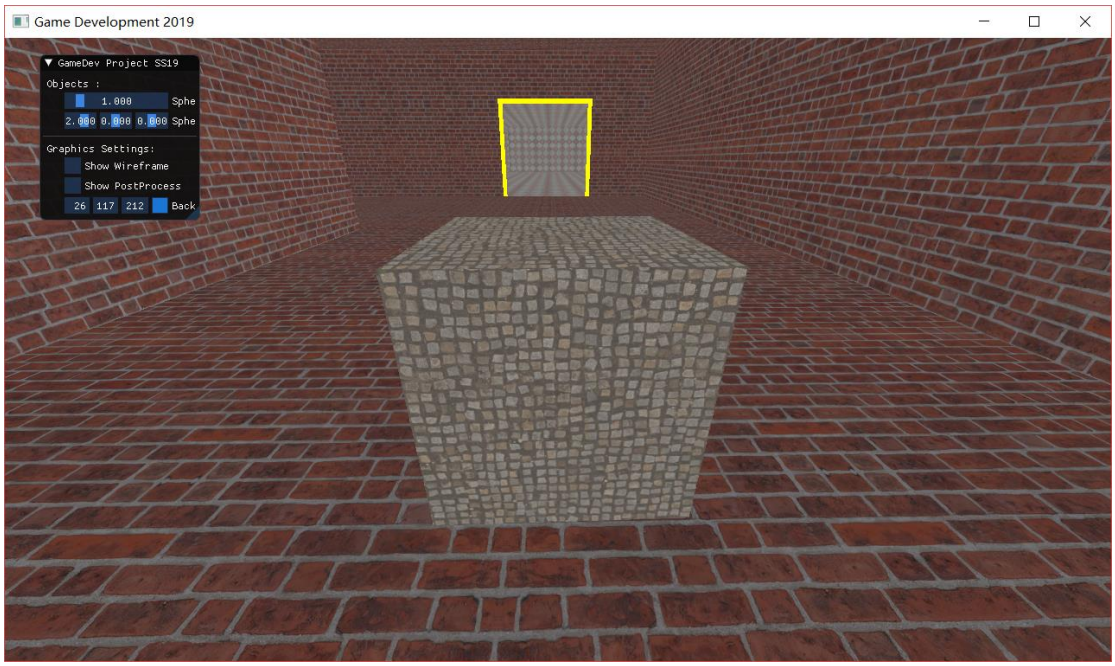

Figure 8 Outline of the portal

I use a separated shader to render this outline. After we use stencil to mark the area of the portal while drawing the portal object, I draw a slightly enlarged portal before actually render the content of the portal which will enable the stencil testing. Then we enable the stencil testing and the area corresponding to the normal portal will be rendered with the content of the portal. However, the enlarged part will be left ending up with the outline of the portal.

1.4. Oren-Nayar Diffuse Reflection

I have implemented oren-nayar reflection model. This model takes the roughness of the surface into consideration which means the model does not assume the surface to be perfectly flat and the diffuse reflection simply happens in all direction on the surface. The model use V-shaped micro facet to simulate the micro unevenness of the surface. The facets are perfectly specular and they are distributed throughout the surface of the object which is modeled by some probability distribution function proposed by Torrance and Sparrow [1]. I use the formula referred from Wikipedia to implement this model [1].

The advantage of this model is that it considers the roughness of the model so that the effect of diffuse reflection is more realistic.

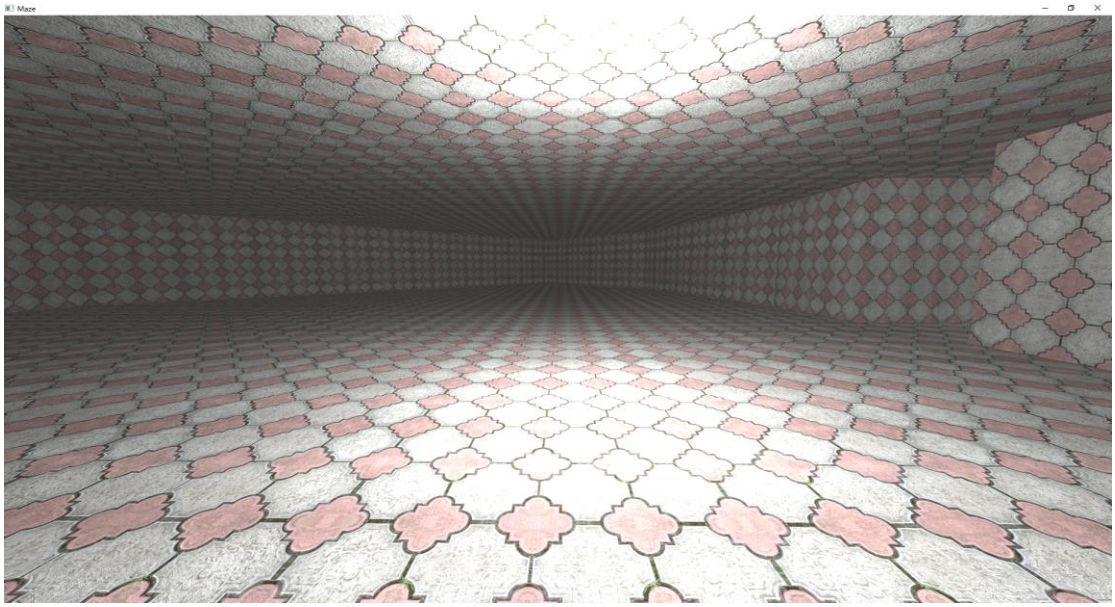

Figure 9 Simple lambert diffuse

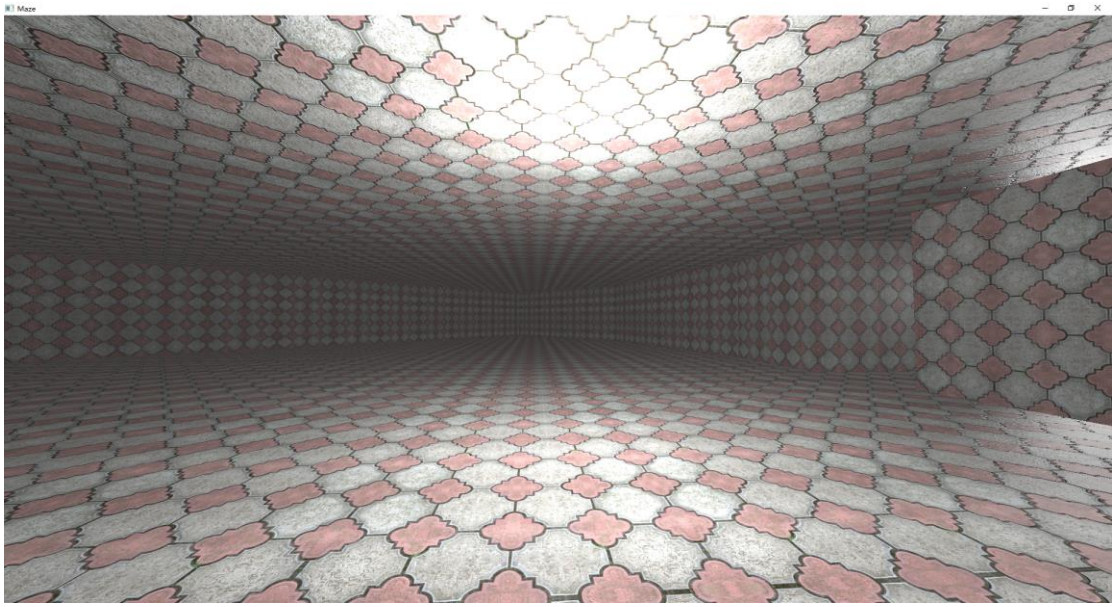

Figure 10 Oren-Nayar diffuse

As we can see from the comparation of figure 9 and 10, when it is close to the light source the scene is overexposed with simple lambert diffuse model. However, the oren-nayar model keeps this problem away making the diffuse object more realistic even when it is close to the light source.

1.5. Cook-Torrance Specular Reflection

Similar to oren-nayar diffuse model, I also implemented the cook-torrance specular model. The cook-torrance model uses same idea as considering the micro structure of the surface. The difference from oren-nayar model is that the facets are considered to be perfectly diffuse. The distribution is modeled by probability distribution known as Beckmann model. And the model also

simulates the geometry attenuation and the specular term. The implementation is done by following the formula of this model [2]. The result is shown in figure 11.

The model gives more detailed specular effect. However, considering that our game does not have specular object, we did not use it at the end. But the method is implemented and backed up.

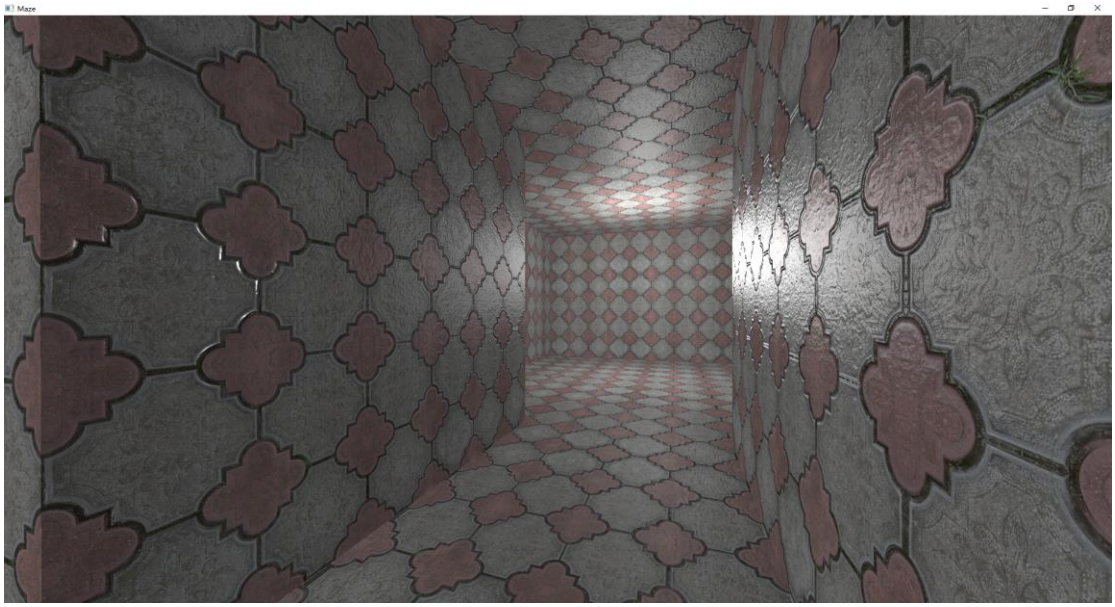

Figure 11 Cook-Torrance specular reflection

1.6. FXAA

I implemented fast approximate anti-aliasing, the technique that is suitable for anti-aliasing in deferred rendering. FXAA aims to find the edge of the object and then blur the pixel around the edge in order to handle the artifacts. I have tried two ways of FXAA. The final implementation is learned from the book "OpenGL ES 3.0 Cookbook" [3]

The first step is to convert the RGB color into luminosity for convenience of edge detection. Afterwards, for each pixel the neighboring pixels are considered. If the difference of the luminosity is larger than a threshold, the pixel is considered as edge pixel which requires FXAA. Otherwise, the shader does nothing but outputting the original color. When the pixel is determined to be edge pixel, the shader computes the direction of the edge using the formula in the book (page 409) [3]. After the direction of the edge is computed, an attenuation term is used to decide for how long the edge should be and thus the blurring. Then the formula (page 410) [3] for blurring the edge is applied. It firstly takes 2 samples to blur the edge and then use 4 samples to blur the edge. If the 4-sample blurred pixel is smaller than the minimum luminosity or larger than the maximum luminosity of the neighborhood, it will use the 2-sample

blurring instead indicating that the 4-sampled blurring is covering too much area which does not belong to the edge. The effect is shown in figure 11 and 12.

Figure 12 Without FXAA, the artifacts are clear and sharp

Figure 13 With FXAA, the artifacts are blurred, though still there. In general, the artifacts will be even less seen from a farther distance

1.7. Other technique

During the lab course, I also tried to implement other features but some did not succeed at the end. For example, I tried a lot on clustered shading (one can see in the branch “cluster_shading” in the respository). The clustered shading reacts to the disadvantage of simple deferred shading that the lighting computation of every light source is performed every time. Similar to the existence of unnessary objects, many light sources are also unnecessary for some objects which is out of the range of the light source. Clustered shading clusters the space into subspace called clusters and assign necessary light sources to the clusters. For each pixel belonging to certain cluster only relevant light sources are computed. This allows dramatically large number of light sources in the scene to be possible.

I use compute shaders to cluster the space and assign light sources, and store the result using shader storage buffer object. Then I tried to retrieve the light source and do lighting in the fragment shader. However, I did not manage to successfully make it correct in limited time because the overall algorithm is very complicated which is a pity.

2. Summary

This game has many features including what I have implemented. The project has experienced much refactoring for the purpose of cleaning code and the improvement of the performance. Therefore, the code might be different from what is described here but the general logic is consistent. There are also many things I studied and tried but finally did not succeed. For example, I experimented with clustered shading but finally did not manage to implement it due to the complexness.

3. Reference

[1] Wikipedia, “ Oren – Nayar reflectance model, ” [Online]. Available: https://en.wikipedia.org/wiki/Oren%E2%80%93Nayar_reflectance_model. [Date: 25 07 2019].

[2] Wikipedia, “ Specular highlight, ” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Specular_highlight#Cook%E2%80%93Torrance_model. [Date: 25 07 2019].

[3] P. Singh, “ Anti-aliasing Techniques, ” From *OpenGL ES 3.0 Cookbook*, Packt Publishing Ltd, 2015, pp. 405 - 411.