学号	姓名	论文规范性（10）	问题分析与调研（30）	方案创新性（20）	实验结果分析与讨论（40）	结课论文总成绩（100）
21301154	胡鑫	7	24	17	33	81

缺少对现有工作的梳理，参考文献较少

Practice of Shadow Mapping

Xinhu

June 19, 2024

Abstract

The produce of shadow is the result of light being blocked. To be more detailed, When the light from a light source cannot reach the surface of an object due to obstruction from other objects, then the object is in shadow. Due to the significant improvement in depth perception of scenes and objects, shadow can make the rendering of 3D scenes more realistic. Besides, in the modern video games, shadow is also a crucial part of performance optimization. In this paper, I will practice two different shadow rendering methods

1 Introduction

Shadow rendering is a crucial part of computer graphics. Its principle is relatively simple. For a fragment, if it's located in a shadow, it means that the camera can see the fragment, but the light source cannot see it; If it is outside the shadow, it means that the camera can see this fragment, and the light source can also see this point.

The overall summary calculation steps are as follows: first, perform a perspective projection on the object from the view of the light source, and only record the depth information of the points visible to the light after the projection. Then perform perspective projection of the current scene from the camera perspective, and project all fragments seen by the camera back onto the light source camera, and it will get a depth map. Then compare the two depth maps to see if the depth of the fragments recorded above is consistent. If consistent, it means that the light source can see the fragment, that is, the fragment is not in the shadow and is visible; On the contrary, the fragment is in shadows and it's not visible.

In this paper, I will implement shadow map and point shadows,

2 Related Works

2.1 Shadow Map

Shadow map is widely used in video games as a method of implementing shadows. Its idea is that we render from the perspective of the position of light source, and everything that can be seen by the light source will be illuminated, while things that cannot be seen will be in shadow. Its implementation principle is relatively simple. Firstly we render the depth map of the light source's perspective, and then we use the generated depth map to calculate whether the fragment is in the shadow.

2.2 Point Shadows

The shadow effect generated by using shadow maps is good, but it is only applicable to directional light, because shadow is generated from a fixed perspective of the light source.

Point light shadow is suitable for point light and it can generate shadows in all directions, and it is also called omnidirectional shadow maps. Point shadow require rendering all scenes from one point light source, so regular 2D depth map can't work. And because the Cubemaps can store environmental data of six faces, it can render the entire scene onto each face of the Cubemaps and sample them as depth values around the point light source. Its algorithm principle is similar to shadow map, it will use a direction vector to sample the Cubemaps, then obtaining the depth of the current fragment, and the following steps are the same as shadow map.

3 Experimental Procedure

The graphics API I used in this experiment is OpenGL.

3.1 Shadow Map

The principle of shadow map is simple, but the code implementation is more complicated. Firstly, we need to generate a depth map. We will use framebuffers to store the rendering results of the scene in a texture. Firstly I create a framebuffer object for the rendered depth map, and create a 2D texture to provide depth buffering for framebuffer. I need to explicitly tell OpenGL that don't use any color data for rendering meanwhile. Before rendering the depth map, I also need to use glViewport to change the parameters of the viewport to fit the size of the shadow map.

The next step is to calculate the transformation of the light source space. Because shadow map is only applicable to directional lights where all rays are parallel, I use an orthogonal projection matrix for the light source. I can use the glm::lookAt function to transform an object into space that is visible from the perspective of the light source.

$$glm :: mat4\ viewMatrix = glm :: lookAt(lightPosition,\ scencePosition,\ upVector);$$

The result of multiplying the orthogonal matrix and the viewMatrix is just the transform matrix of the light source space. When rendering a scene from the perspective of light source, I also need a simple fragment shader, due to the lack of color buffering, the fragment shader don't require any processing, so it can be empty.

After completing the rendering of the depth map, then I can start rendering shadows. Fragment shader use Blinn-Phong lighting model to render the scene, then calculate a shadow value, which is 1.0f when the fragment is in the shadow and 0.0f outside the shadow. Afterwards, multiply the diffuse and specular colors with the shadow value. To avoid shadow becoming completely black, I still need to remove the ambient component from the multiplication. Eventually, activate the shader, bind the appropriate texture, and use the default projection and view matrix, it can render the shadow successfully.

3.2 Point Shadows

The core idea of point shadows is that use cubemaps to store the sampling depth values around the point light source. Firstly I use geometric shader to establish the depth cubemaps, because geometric shader is allowed to use a single rendering process to complete it, and that can save some rendering performance. So I create a cubemaps and generate each face of the cubemaps as a 2D depth value of texture image. Then use the geometric shader to attach the six faces of the cubemaps' texture to the framebuffers, and render scene as normal with shadow mapping.

After setting up the framebuffers and cubemaps, I need to transform all the objects of the scene into the corresponding light source space in six directions of light. So for each face of the cube texture, there needs to be a perspective projection matrix. I use function glm::perspective to create the matrix, and set the field of view parameter to 90 degrees. This is because only 90 degrees can ensure that the field of view is large enough to fill one face of the cube texture appropriately, and all faces of the cube texture can be aligned with other faces at the edge. Meanwhile we still need to provide a different view matrix for each direction. And I create six viewing directions using function glm::lookAt, each looks in one direction of the cube texture in order. Multiplying the perspective matrix and the view matrix yields the corresponding transform matrix.

To render the values to the depth cubemaps, it will need three shaders: vertex shader, fragment shader, and a geometric shader. Vertex shader simply transform vertices into world space and then send them directly to geometric shaders. For the geometric shader, it input a triangle and output six triangles totally. It traverse the six faces of the cube texture, and assign each face as an output face, save the integer of this face to gl_Layer. Then it transform each vertex of world space into the light source space by multiplying the light space transform matrix by FragPos, and generating each triangle. It also need to send the final FragPos variable to the fragment shader for calculating a depth value. For fragment shader, it take the FragPos from the geometric shader, the position vector of the light source, and the far plane value of the visual cone as inputs. And it map the distance between the fragment and the light source to the range of 0 to 1, then write it as the depth value of the fragment.

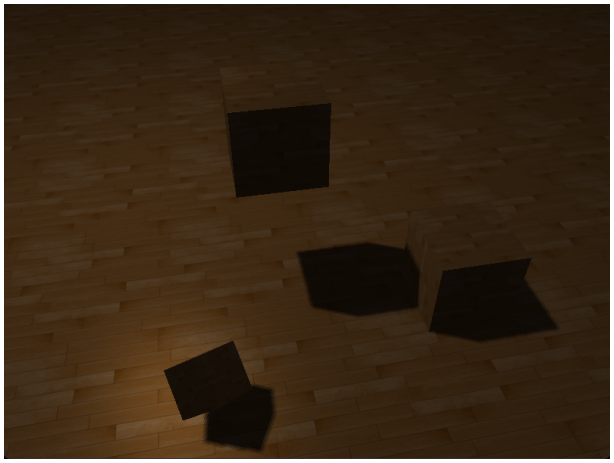

Figure 1: Shadow Map

Activate the framebuffer object attached to the cubemaps, use these shaders to render the scene, and it will get the depth cubemaps.

Then it's time to render the point shaders. This process is similar to the shadow map, the difference is that it don't need the position of fragment and it use a direction vector to sample the depth values. And the fragment shader still uses the Blinn-Phong lighting model. It uses the difference vector between the position of the fragment and the position of the light source as a directional vector to sample the cubemaps. Then calculate the standardized depth value between the light source and its closest visible fragment, and convert it back to the range of 0 to far plane value, which means multiplying it by far plane value. Next, obtain the depth value between the fragment and the light source, then compare it with the closest fragment depth values to see which one is closer, in order to determine whether the current fragment is in the shadow. By using these shaders to render the scene, it can achieve a good shadow effects.

4 Experimental Result

4.1 Shadow Map

The light of shadow map is the global light source. But the effect of the shadow is kind of unreal due to the texture of the shadows presenting a stripe style. The lack of realism in this type of shadow map is called Shadow Acne. The reason for it's generation is that shadow maps are limited by resolution, and when the distance from the light source is relatively far, multiple fragments may be sampled from the same value of the depth map. So when the light source faces the surface at an angle, there will be a problem, and in this case, the depth map is also rendered from this angle. Multiple segments will be sampled from the depth texture pixels of the same slope, some above the floor and some below the floor. In this way, there will be a difference in the shadows. Because of this, some fragments are considered to be in the shadow, while others are not, resulting in the stripe style in the image. We can use a technique called shadow bias to solve this problem. We simply apply an offset to the depth (or depth map) of the surface, so that the fragment is not mistakenly considered to be below the surface.

The most commonly used method to solve Shadow Acne is called shadow bias, which simply applies an offset to the depth map of the surface so that the fragment is not mistakenly considered below the surface. After applying the offset, all sampling points get a depth values smaller than the surface depth, so that the entire surface was correctly lighted without any shadows.

After resolving the problem of Shadow Acne, the rendering result is shown in Figure 1.

4.2 Point Shadow

The light source for point shadows is a point light source, which emits light from the light source to the surrounding area. So I place the light source at the center of the scene. After implementing the

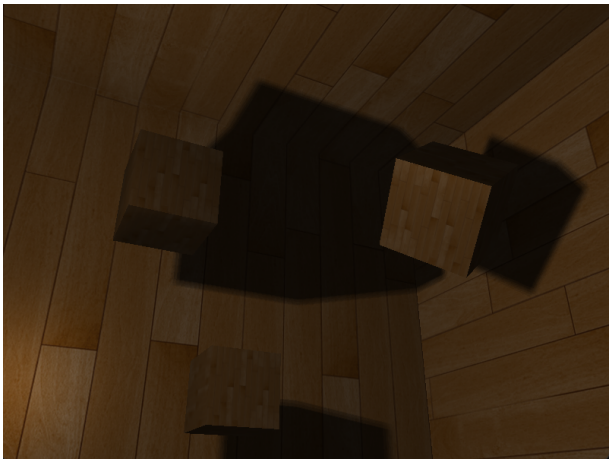

Figure 2: Point Shadow

above experimental process, the effect of the rendering shadow is good enough. So there is no need to optimize point shadows. And the rendering result is shown in Figure 2.

5 Reference

1. Article Shadow Mapping in LearnOpenGL By JoeyDeVries;

2. Vedio OpenGL Game Rendering Tutorial: How Shadow Mapping Works By thebennybox;

3. Article Shadow Mapping for point light sources in OpenGL By sunandblackcat;