《计算机图形学》实验 7 实验报告

学生	刘沅昊	学号	15331220
学院	数据科学与计算机学院	年级专业	16 级软件工程 (数字媒体技术)

实验内容：

Homework

Basic:

1. 实现方向光源的Shadowing Mapping:
    - 要求场景中至少有一个object和一块平面(用于显示shadow)
    - 光源的投影方式任选其一即可
    - 在报告里结合代码，解释Shadowing Mapping算法
2. 修改GUI

Bonus:

1. 实现光源在正交/透视两种投影下的Shadowing Mapping
2. 优化Shadowing Mapping (可结合References链接，或其他方法。优化方式越多越好，在报告里说明，有加分)

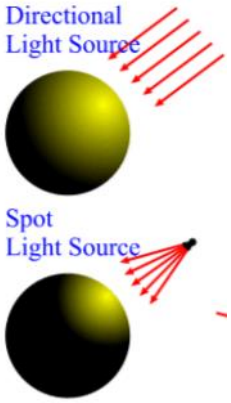
实验结果：

具体效果请查看 record.gif

Basic：

1. 实现方向光源的 Shadowing Mapping

在我的理解中，方向光源就是平行光，平行光投影的阴影就是正交的，点光源的投影就是透视的。如下方课件里面的光源一样，



这和 Bonus 第一条原理挺像。

Shading Mapping 的过程：

s1：采用帧缓冲生成深度贴图（从光的透视图里渲染的深度纹理），用于计算阴影。

```
// configure depth map FBO
// -----------------------
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// create depth texture
unsigned int depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
// attach depth texture as FBO's depth buffer
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

首先，我们要为渲染的深度贴图创建一个帧缓冲对象。然后，创建一个 2D 纹理，提供给帧缓冲的深度缓冲使用。纹理的高宽设置为 1024：这是深度贴图的解析度。合理配置将深度值渲染到纹理的帧缓冲后，我们就可以开始第一步了：生成深度贴图。

```
// 2. render scene as normal using the generated depth/shadow map
// --------------------------------------------------------------
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shader.use();
glm::mat4 projection = glm::perspective(angle, (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = glm::lookAt(
    glm::vec3(camPos[0], camPos[1], camPos[2]),
    glm::vec3(lookAtCenter[0], lookAtCenter[1], lookAtCenter[2]),
    glm::vec3(0, 1, 0)
);
shader.setMat4("projection", projection);
shader.setMat4("view", view);
// set light uniforms
shader.setVec3("viewPos", glm::vec3(camPos[0], camPos[1], camPos[2]));
shader.setVec3("lightPos", lightPos);
shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, woodTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);
renderScene(shader);
```

s2：设置光源为正交投影，由于投影矩阵间接决定可视区域的范围，以及哪些东西不会被裁切，所以需要确保投影视锥（frustum）的大小，以包含打算在深度贴图中包含的物体。当物体和片元不在深度贴图中时，它们就不会产生阴影。

```
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
lightSpaceMatrix = lightProjection * lightView;
// render scene from light's point of view
simpleDepthShader.use();
simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
```

s3：渲染深度缓冲。当以光的透视图进行场景渲染的时候，会用一个比较简单的着色器，这个着色器除了把顶点变换到光空间以外，不会做得更多了。

```
        // render scene from light's point of view
        simpleDepthShader.use();
        simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

        glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
        glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
        glClear(GL_DEPTH_BUFFER_BIT);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, woodTexture);
        renderScene(simpleDepthShader);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

s4：进行阴影计算作为 shadow 值，当 fragment 在阴影中时是 1.0，在阴影外是 0.0。

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range
    fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // calculate bias (based on depth map resolution and slope)
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
    // check whether current frag pos is in shadow
    // float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
    // PCF
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    for(int x = -1; x <= 1; ++x)
    {
        for(int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) *
             texelSize).r;
            shadow += currentDepth - bias > pcfDepth  ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;

    // keep the shadow at 0.0 when outside the far_plane region of the
    light's frustum.
    if(projCoords.z > 1.0)
        shadow = 0.0;

    return shadow;
}
```

这里需要注意的是，光的视锥不可见的区域一律被认为是处于阴影中，不管它真的处于阴影之中。出现这个状况是因为超出光的视锥的投影坐标比 1.0 大，这样采样的深度纹理就会超出他默认的 0 到 1 的范围。根据纹理环绕方式，将会得到不正确的深度结果，它不是基于真实的来自光源的深度值。解决方法是让所有超出深度贴图的坐标的深度范围是 1.0，这样超出的坐标将永远不在阴影之中。我们可以储存一个边框颜色，然后把深度贴图的纹理环绕选项设置为 GL_CLAMP_TO_BORDER。

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

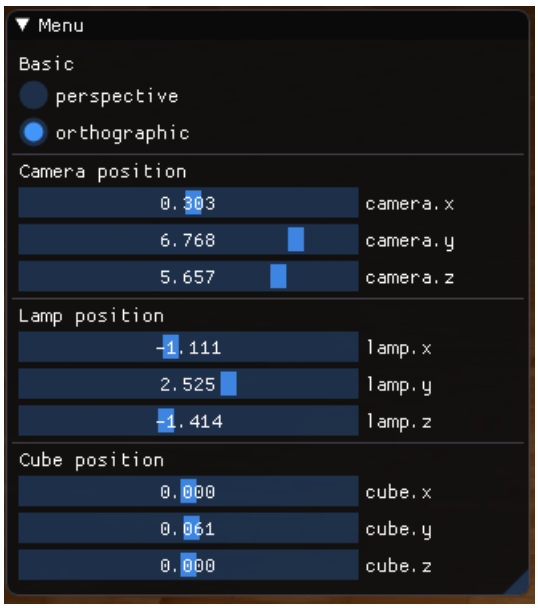
但是会导致物体移动到一个区域外的阴影消失，如下图所示。

s5：把 diffuse 和 specular 乘以（1-阴影元素），表示这个片元有多大成分不在阴影中

```
72      float shadow = ShadowCalculation(fs_in.FragPosLightSpace);

73      vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) *
        color;
```

2. 修改 GUI

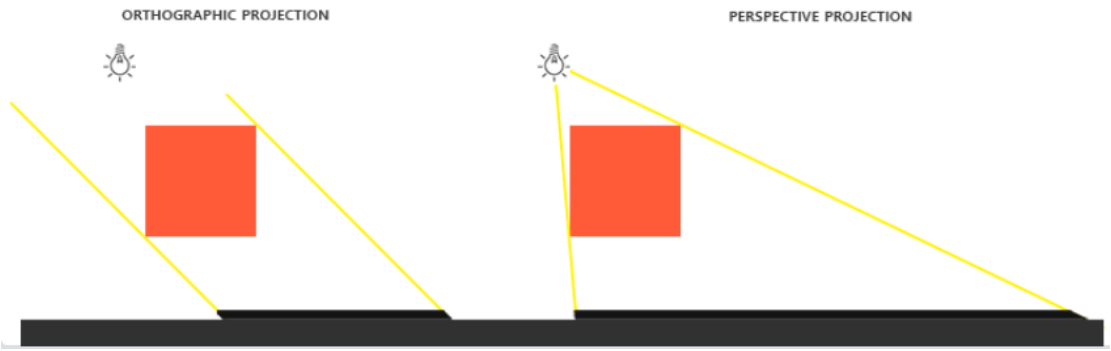



```
207         {
208             ImGui::Begin("Menu");
209             ImGui::Text("Basic");
210             ImGui::RadioButton("perspective", &mode, 1);
211             ImGui::RadioButton("orthographic", &mode, 2);
212             ImGui::Separator();
213             ImGui::Text("Camera position");
214             ImGui::SliderFloat("camera.x", &camPos[0], -10.0f, 10.0f);
215             ImGui::SliderFloat("camera.y", &camPos[1], -10.0f, 10.0f);
216             ImGui::SliderFloat("camera.z", &camPos[2], -10.0f, 10.0f);
217             ImGui::Separator();
218             ImGui::Text("Lamp position");
219             ImGui::SliderFloat("lamp.x", &lightPos.x, -10.0f, 10.0f);
220             ImGui::SliderFloat("lamp.y", &lightPos.y, -10.0f, 10.0f);
221             ImGui::SliderFloat("lamp.z", &lightPos.z, -10.0f, 10.0f);
222             ImGui::Separator();
223             ImGui::Text("Cube position");
224             ImGui::SliderFloat("cube.x", &cubePosition.x, -3.0f, 3.0f);
225             ImGui::SliderFloat("cube.y", &cubePosition.y, -3.0f, 3.0f);
226             ImGui::SliderFloat("cube.z", &cubePosition.z, -3.0f, 3.0f);
227 
228             ImGui::End();
229         }
```

添加 GUI 可以调整正交、透视两种阴影，以及光源、物体、摄像机的位置。

Bobus：

1. 实现光源在正交/透视两种投影下的 Shadowing Mapping

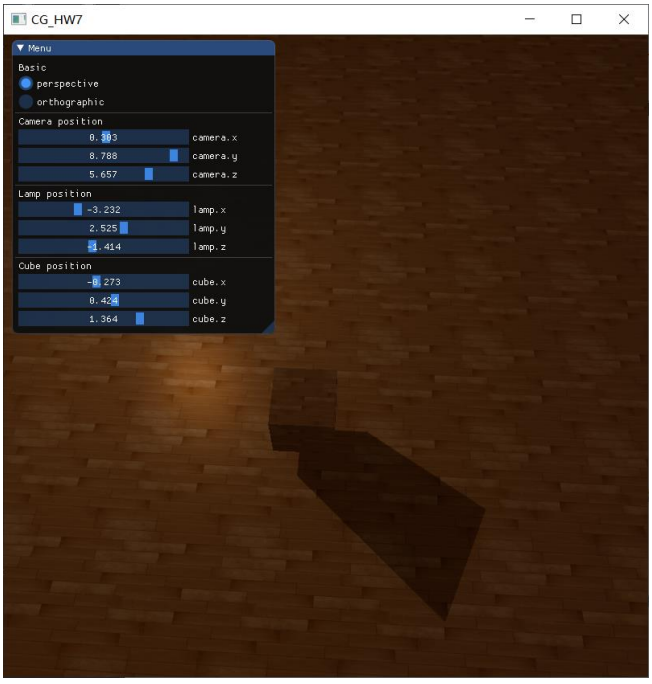


透视投影矩阵，会将所有顶点根据透视关系进行变形，结果因此而不同。透视投影对于光源来说更合理，不像定向光，它是有自己的位置的。透视投影因此更经常用在点光源和聚光灯

上，而正交投影经常用在定向光上。

```
245         float near_plane = 1.0f, far_plane = 7.5f;
246         if (mode == 1) {
247             lightProjection = glm::perspective(glm::radians(45.0f), (GLfloat)SHADOW_WIDTH / (GLfloat)SHADOW_HEIGHT, near_plane, far_plane);
248         }
249         else if (mode == 2) {
250             lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
251         }
```

图表 1 正交投影

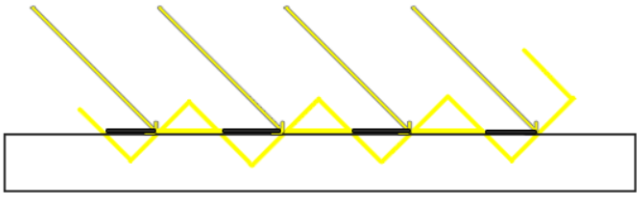


图表 2 透视投影

2. 优化 Shadowing Mapping

1. 阴影失真

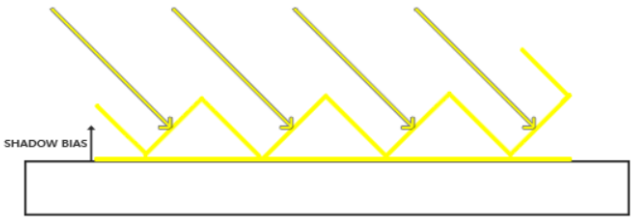
可以看到地板四边形渲染出很大一块交替黑线。这种阴影贴图的不真实感叫做阴影失真(Shadow Acne)，下图解释了成因：



因为阴影贴图受限于解析度，在距离光源比较远的情况下，多个片元可能从深度贴图的同一个值中去采样。图片每个斜坡代表深度贴图一个单独的纹理像素。

虽然很多时候没问题，但是当光源以一个角度朝向表面的时候就会出问题，这种情况下深度贴图也是从一个角度下进行渲染的。多个片元就会从同一个斜坡的深度纹理像素中采样，有些在地板上面，有些在地板下面；这样我们所得到的阴影就有了差异。因为这个，有些片元被认为是在阴影之中，有些不在，由此产生了图片中的条纹样式。

可以用一个叫做阴影偏移（shadow bias）的技巧来解决这个问题，简单的对表面的深度（或深度贴图）应用一个偏移量，这样片元就不会被错误地认为在表面之下了。



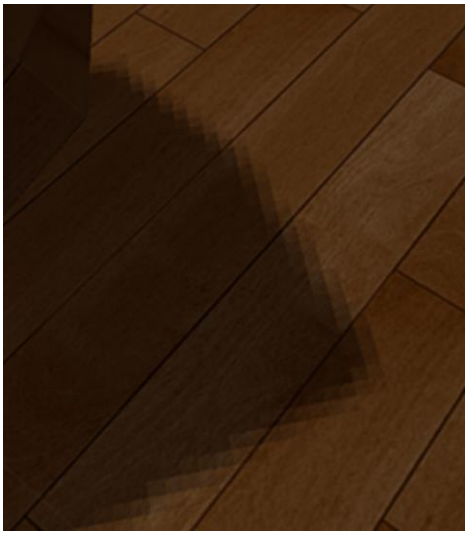
2. 采样过多

光的视锥不可见的区域一律被认为是处于阴影中，不管它真的处于阴影之中。出现这个状况是因为超出光的视锥的投影坐标比 1.0 大，这样采样的深度纹理就会超出他默认的 0 到 1 的范围。根据纹理环绕方式，我们将会得到不正确的深度结果，它不是基于真实的来自光源的深度值。这在上面 Shadowing Mapping s4 提到解决方法。

3. PCF

因为深度贴图有一个固定的解析度，多个片元对应于一个纹理像素。结果就是多个片元会从深度贴图的同一个深度值进行采样，这几个片元便得到的是同一个阴影，这就会产生锯齿边。

可以通过增加深度贴图解析度的方式来降低锯齿块，也可以尝试尽可能的让光的视锥接近场景。另一个解决方案叫做 PCF（percentage-closer filtering），这是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能在也可能不再阴影中。所有的次生结果接着结合在一起，进行平均化，我们就得到了柔和阴影。



从稍微远一点的距离看去，阴影效果好多了，也不那么生硬了。如果放大，仍会看到阴影贴图解析度的不真实感，但通常对于大多数应用来说效果已经很好了。