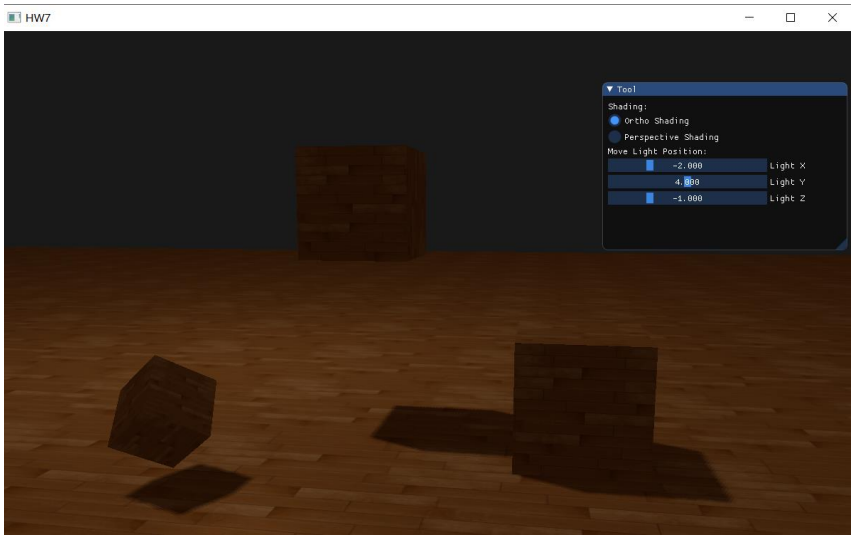
Homework 7 - Shadowing Mapping

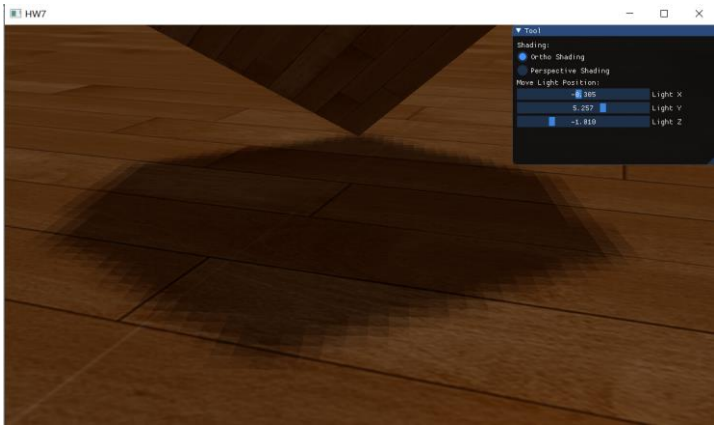
一、实现结果

✧ 正交投影的 Shadowing Mapping



✧ 透视投影的 Shadowing Mapping

✧ 使用 PCF



二、实现思路

1. 要求场景中至少有一个 object 和一块平面(用于显示 shadow)

➢ 在本次实验中在程序场景中放置了一个平面以及三个不同的正方体用来显示阴影。

a. 设置平面的属性数组，拥有平面的顶点的位置发现和纹理应该处于的位置，因为平面是一个矩形，所以平面由两个三角形拼成。之后处理与以前相似设置平面的 VAO 以及 VBO 还有对应变量的位置。

```
// 设置平面的顶点和法线以及纹理的位置
float planeVertices[] = {
    // 位置              // 法线           // 纹理位置
    25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,  25.0f,  0.0f,
    -25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,   0.0f,  0.0f,
    -25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,   0.0f, 25.0f,

    25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,  25.0f,  0.0f,
    -25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,   0.0f, 25.0f,
    25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,  25.0f, 25.0f
};
// 设置平面的VAO和VBO
unsigned int planeVBO;
glGenVertexArrays(1, &planeVAO);
glGenBuffers(1, &planeVBO);
glBindVertexArray(planeVAO);
glBindBuffer(GL_ARRAY_BUFFER, planeVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(planeVertices), planeVertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
// 顶点位置
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
glEnableVertexAttribArray(1);
// 法线向量
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(2);
// 纹理位置
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
glBindVertexArray(0);
```

b. 所有的正方体的顶点位置相同，面的法向量和纹理位置相同，只是摆放的位置以及大小和旋转不同，通过对模型矩阵进行操作从而生成不同位置的正方体。

```
    // 渲染正方体
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(0.0f, 1.5f, 0.0));
    model = glm::scale(model, glm::vec3(0.5f));
    shader.setMat4("model", model);
    renderCube();
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(2.0f, 0.0f, 1.0));
    model = glm::scale(model, glm::vec3(0.5f));
    shader.setMat4("model", model);
    renderCube();
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(-1.0f, 0.0f, 2.0));
    model = glm::rotate(model, glm::radians(60.0f), glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
    model = glm::scale(model, glm::vec3(0.25));
    shader.setMat4("model", model);
    renderCube();
```

2. 在报告里结合代码，解释 Shadowing Mapping 算法

➢ Shadowing Mapping 其实分为主要的两个部分，第一个是渲染深度贴图，第二是个渲染阴影。

1. 渲染深度贴图

➢ 通过光源发出去的光，如果能从光源的透视图来渲染场景，并把深度值的结果储存到纹理中，就可以通过深度值显示从光源的透视图下见到的第一个片元，而其他的片元就会作为阴影，将这些深度值叫做，深度贴图。所以首先需要生成一张深度贴图。

- 为深度贴图创建帧缓冲对象，把生成的深度纹理作为帧缓冲的深度缓冲。

```
// ------------创建深度贴图----------------------------------------
// 创建帧缓冲对象
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// 纹理的宽高，深度贴图的解析度
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
// 创建一个2D纹理，提供给帧缓冲的深度缓冲使用
unsigned int depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
// 纹理格式为深度
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);
// 储存一个边框颜色，所有超出深度贴图的坐标的深度范围是1.0
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);
// 把生成的深度纹理作为帧缓冲的深度缓冲
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
// 不进行颜色数据的渲染
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- 如何将世界坐标下的物体转换到光空间的坐标下，将使用一个光空间的变换矩阵，该变换矩阵利用一个正交投影矩阵和光源到场景中央的 lookAt 矩阵的乘积。首先使用的是一个所有光线都平行的定向光，所以将为光源使用正交投影矩阵，透视图将没有任何变形。也可以使用透视投影矩阵，但是它会将所有顶点根据透视关系进行变形，结果因此而不同。

```
// ------------------光源空间的变换-----------------------------
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
float near_plane = 1.0f, far_plane = 7.5f;
// 光源使用透视矩阵
if(type == 1)
    lightProjection = glm::perspective(glm::radians(45.0f), (GLfloat)SHADOW_WIDTH / (GLfloat)SHADOW_HEIGHT, near_plane, far_plane);
// 光源使用正交投影矩阵，所有光线都平行的定向光
if(type == 0)
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
// 从光源的位置看向场景中央
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
// 变换到从光源视角可见的空间的变换矩阵
lightSpaceMatrix = lightProjection * lightView;
```

- 之后如果是以光的透视图进行场景渲染的时候，在着色器中将顶点变换到光空间，可以得到光到每个片段的深度值。所以用刚才的变换矩阵乘以物体的模型矩阵。

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main()
{
    // 把顶点变换到光空间
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}
```

2. 渲染阴影

- 在像素着色器中，检验一个片元是否在阴影之中。首先需要在顶点着色器中进行光空间的变换。

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec2 TexCoords;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

void main()
{
    // 经变换的世界空间顶点位置
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;
    vs_out.TexCoords = aTexCoords;
    // 把世界空间顶点位置转换为光空间
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

- 使用 Blinn-Phong 光照模型渲染场景，Blinn-Phong 模型和 Phong 的区别就是利用光线与视线夹角一半方向上的一个单位向量，让这个半程向量与法线向量进行点乘，测量的是法线与半程向量之间的夹角，而 Phong 光照模型测量的是观察方向与反射向量间的夹角。Blinn-Phong 光照模型减少了高光断层的现象。

```
// texture函数来采样纹理的颜色
vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
vec3 normal = normalize(fs_in.Normal);
vec3 lightColor = vec3(0.3);
// 环境分量
vec3 ambient = 0.3 * color;
// 漫反射分量
vec3 lightDir = normalize(lightPos - fs_in.FragPos);
float diff = max(dot(lightDir, normal), 0.0);
vec3 diffuse = diff * lightColor;
// 镜面分量
vec3 viewDir = normalize(viewPos - fs_in.FragPos);
vec3 reflectDir = reflect(-lightDir, normal);
float spec = 0.0;
vec3 halfwayDir = normalize(lightDir + viewDir);
// 对表面法线和半程向量进行一次约束点乘
spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
vec3 specular = spec * lightColor;
```

简单的阴影计算

- 加入一个阴影的计算，检查一个片元是否在阴影中，首先需要手动执行透视除法，因为裁切空间的 FragPosLightSpace 并不会通过 gl_Position 传到像素着色器里，而我们需要把光空间片元位置转换为裁切空间的标准化设备坐标。然后将所有的深度和深度贴图深度相比较，所以需要将整个 projCoords 向量都需要变换到[0,1]范围。从 projCoords 坐标直接对应于变换过的 NDC 坐标。将得到光的位置视野下最近的深度。通过比较当前深度与最近的深度从而得出片元是否是阴影。

```
// 执行透视除法
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
// 将NDC坐标变换为0到1
projCoords = projCoords * 0.5 + 0.5;
// 得到光的位置视野下最近的深度
float closestDepth = texture(shadowMap, projCoords.xy).r;
// 获取投影向量的z坐标，它等于来自光的透视视角的片元的深度
float currentDepth = projCoords.z;
```

阴影失真

- 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，使用阴影偏移，对深度贴图应用一个偏移量，使用了偏移量后，所有采样点都获得了比表面深度更小的深度值，整个表面就正确地被照亮。根据表面朝向光线的角度更改偏移量。

```
// 计算阴影偏移
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);
```

采样过多

- 深度贴图的大小是固定的，而超出深度贴图的区域就会变成阴影，所以利用在深度贴图外的区域会返回一个 1.0 的深度值，阴影值为 0.0，利用储存一个边框颜色，然后把深度贴图的纹理环绕选项设置为 GL_CLAMP_TO_BORDER。在超出光的正交视锥的远平面，需要在投影向量的 z 坐标大于 1.0 时把 shadow 的值强制设为 0.0。

```
// 储存一个边框颜色，所有超出深度贴图的坐标的深度范围是1.0
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);
```

```
// 投影向量的z坐标大于1.0，把shadow的值强制设为0.0：
if(projCoords.z > 1.0)
    shadow = 0.0;
```

PCF

- 深度贴图有一个固定的解析度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。利用 PCF 从纹理像素四周对深度贴图采样，然后把结果平均。采样得到 9 个值，它们在投影坐标的 x 和 y 值的周围，为阴影阻挡进行测试，并最终通过样本的总数目将结果平均化。

```
// ---------------------PCF----------------------------------------
// 从纹理像素四周对深度贴图采样，然后把结果平均
float shadow = 0.0;
// 一个单独纹理像素的大小，用以对纹理坐标进行偏移，增加阴影的柔和程度
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;
```

```
// 计算阴影
float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
// diffuse和specular乘以(1-阴影元素)，这表示这个片元有多大成分不在阴影
vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
```

3. 修改 GUI

通过 GUI 可以改变阴影的映射方式是正交还是透视投影，可以改变光的位置来查看两者的差异。

```
//创建gui
ImGui::Begin("Tool");
ImGui::Text("Shading:");
ImGui::RadioButton("Ortho Shading", &type, 0);
ImGui::RadioButton("Perspective Shading", &type, 1);
ImGui::Text("Move Light Position:");
ImGui::SliderFloat("Light X", &lightPosX, -4.0f, 4.0f);
ImGui::SliderFloat("Light Y", &lightPosY, 0.0f, 8.0f);
ImGui::SliderFloat("Light Z", &lightPosZ, -2.0f, 2.0f);
ImGui::End();
```