Homework 7 - Shadowing Mapping

黄树凯

16340085

作业要求

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

Bonus:

1. 实现光源在正交/透视两种投影下的Shadowing Mapping
2. 优化Shadowing Mapping

实现及结果

绘制立方体和平面

确定立方体和平面的节点坐标数组，包含位置、法向量和纹理属性

```
/******************
    立方体顶点
*******************/
float vertices[] = {
    // positions            // normals        // texture
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f, 0.0f, 0.0f,
    0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f, 1.0f, 1.0f,
    0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f, 1.0f, 0.0f,
    0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f, 1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f, 0.0f, 1.0f,

    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f, 0.0f, 0.0f,
    0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f, 1.0f, 0.0f,
    0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f, 1.0f, 1.0f,
    0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f, 1.0f, 1.0f,
    -0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f, 0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f, 0.0f, 0.0f,

    -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f, 1.0f, 0.0f,
    -0.5f,  0.5f, -0.5f, -1.0f,  0.0f,  0.0f, 1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
```

```
    -0.5f, -0.5f,  0.5f, -1.0f,  0.0f,  0.0f, 0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f, 1.0f, 0.0f,

    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f, 1.0f, 0.0f,
    0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
    0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f, 1.0f, 1.0f,
    0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f, 1.0f, 0.0f,
    0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f, 0.0f, 0.0f,

    -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f, 0.0f, 1.0f,
    0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f, 1.0f, 1.0f,
    0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f, 1.0f, 0.0f,
    0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f, 0.0f, 1.0f,

    -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f, 0.0f, 1.0f,
    0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f, 1.0f, 0.0f,
    0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f, 1.0f, 1.0f,
    0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f, 1.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f, 0.0f, 1.0f,
    -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f, 0.0f, 0.0f
};
/******************
    平面顶点
******************/
float planeVertices[] = {
    // Positions          // Normals         // texture
    40.0f, -0.5f,  40.0f, 0.0f, 1.0f, 0.0f, 25.0f,  0.0f,
    -40.0f, -0.5f, -40.0f, 0.0f, 1.0f, 0.0f,  0.0f, 25.0f,
    -40.0f, -0.5f,  40.0f, 0.0f, 1.0f, 0.0f,  0.0f,  0.0f,

    40.0f, -0.5f,  40.0f, 0.0f, 1.0f, 0.0f, 25.0f,  0.0f,
    40.0f, -0.5f, -40.0f, 0.0f, 1.0f, 0.0f, 25.0f, 25.0f,
    -40.0f, -0.5f, -40.0f, 0.0f, 1.0f, 0.0f,  0.0f, 25.0f
};
```

分别进行配置

```
unsigned int cubeVAO = 0;
unsigned int VBO = 0;
// 生成 VAO, VBO
glGenVertexArrays(1, &cubeVAO);
glGenBuffers(1, &VBO);
// 绑定顶点数组对象
glBindVertexArray(cubeVAO);
// 绑定 VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 位置属性
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
```

```
// 颜色属性
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 *
sizeof(float)));
// 纹理属性
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 *
sizeof(float)));
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);

unsigned int planeVAO = 0;
unsigned int planeVBO = 0;
// 生成 VAO, VBO
glGenVertexArrays(1, &planeVAO);
glGenBuffers(1, &planeVBO);
// 绑定顶点数组对象
glBindVertexArray(planeVAO);
// 绑定 VBO
glBindBuffer(GL_ARRAY_BUFFER, planeVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(planeVertices), &planeVertices, GL_STATIC_DRAW);
// 位置属性
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
// 颜色属性
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 *
sizeof(float)));
// 纹理属性
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 *
sizeof(float)));
glBindVertexArray(0);
```

顶点着色器和片段着色器

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

out vec3 FragPos;
out vec3 Normal;

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

void main() {
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

```
////////////////////////////
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 FragPos;

uniform float ambientStrength;
uniform float diffuseStrength;
uniform float specularStrength;
uniform float shininess;

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;
uniform vec3 objectColor;

void main() {
    // ambient
    vec3 ambient = ambientStrength * lightColor;

    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diffuseStrength * diff * lightColor;

    // specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
    vec3 specular = specularStrength * spec * lightColor;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
}
```

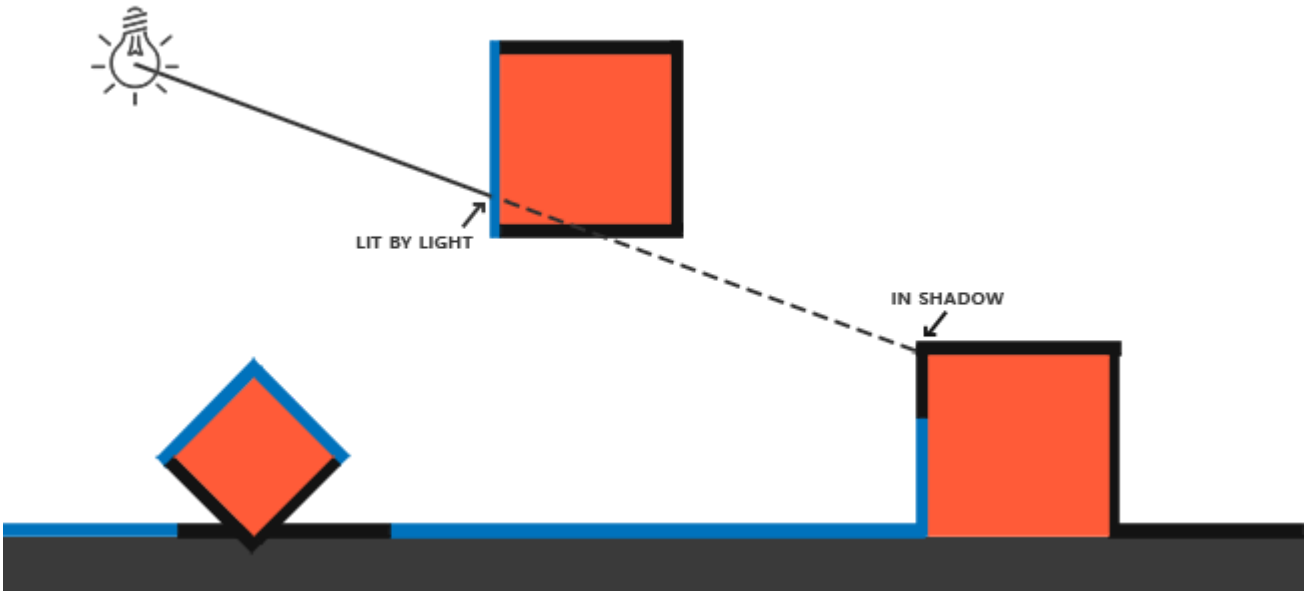
设置位置、颜色属性等并进行渲染

```
// 绘制立方体
cubeShader.use();
cubeShader.setMat4("projection", projection);
cubeShader.setMat4("view", view);
cubeShader.setMat4("model", model);
cubeShader.setVec3("lightPos", lightPos);
cubeShader.setVec3("viewPos", camera.Position);
cubeShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
cubeShader.setBool("shadows", true);
cubeShader.setVec3("objectColor", 1.0f, 0.5f, 0.3f);
cubeShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);

glBindVertexArray(planeVAO);
```

```
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
glBindVertexArray(cubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
```

阴影映射

阴影映射(Shadow Mapping)以光的位置为视角进行渲染，所有能看到的东西都被点亮，看不见的一定在阴影之中。假设有一个地板，在光源和它之间有一个大盒子，由于光源处向光线方向看去，可以看到这个盒子，但看不到地板的一部分，这部分就应该在阴影中。



这里的所有蓝线代表光源可以看到的fragment，黑线代表被遮挡的fragment，它们应该渲染为带阴影的。如果绘制一条从光源出发，到达最右边盒子上的一个片元上的线段或射线，那么射线将先击中悬浮的盒子，最后才会到达最右侧的盒子。结果就是悬浮的盒子被照亮，而最右侧的盒子将处于阴影之中。

为了实现阴影贴图，从一个光的透视图渲染场景，这样就得在光的方向的某一点上渲染场景。深度映射由两个步骤组成：首先，渲染深度贴图，像往常一样渲染场景，使用生成的深度贴图来计算片元 是否在阴影之中。

深度贴图

首先要生成一张深度贴图(Depth Map)。深度贴图是从光的透视图里渲染的深度纹理，用它来计算阴影。将场景的渲染结构存储到一个纹理中，需要用到帧缓冲。

```
// 为渲染的深度贴图创建一个帧缓冲对象
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// 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);

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);
// 生成的深度纹理作为帧缓冲的深度缓冲
float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
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);
```

光源空间的变换

为每个设置合适的投影和视图矩阵，以及相关的模型矩阵。首先，从光的位置的视野下使用不同的投影和视图矩阵来渲染场景。

```
GLfloat near_plane = 1.0f, far_plane = 7.5f;
// 光源使用正交投影，提供一个所有光线都平行的定向光。
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane,
far_plane);

// 创建一个视图矩阵来变换每个物体，把它们变换到从光源视角可见的空间中
glm::mat4 lightView = glm::lookAt(glm::vec(-2.0f, 4.0f, -1.0f), glm::vec3(0.0f),
glm::vec3(1.0));
// 将两个矩阵结合提供一个光空间的变换矩阵，用该矩阵将每个世界空间坐标变换到光源处所见到的那个空间
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
```

渲染至深度贴图

用着色器以光的透视图进行场景渲染，该着色器只负责把顶点变换到光空间

```
// 将一个单独模型的一个顶点，使用lightSpaceMatrix变换到光空间中
#version 330 core
layout (location = 0) in vec3 position;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main() {
    gl_Position = lightSpaceMatrix * model * vec4(position, 1.0f);
}


// 片元不需要任何处理
#version 330 core

void main() {
    // gl_FragDepth = gl_FragCoord.z;
}
```

渲染深度缓冲

```
simpleDepthShader.use();
simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
simpleDepthShader.setMat4("model", model);

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// 渲染平面
glBindVertexArray(planeVAO);
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
// 渲染立方体
glBindVertexArray(cubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

用正交投影的方式来现实深度，将深度贴图渲染到四边形上的像素着色器

```
#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D depthMap;

void main() {
    float depthValue = texture(depthMap, TexCoords).r;
    color = vec4(vec3(depthValue), 1.0);
}
```

渲染阴影

生成深度贴图之后就可以用来生成阴影，在像素着色器中，检查一个片元是否在阴影之中。

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

out vec2 TexCoords;

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

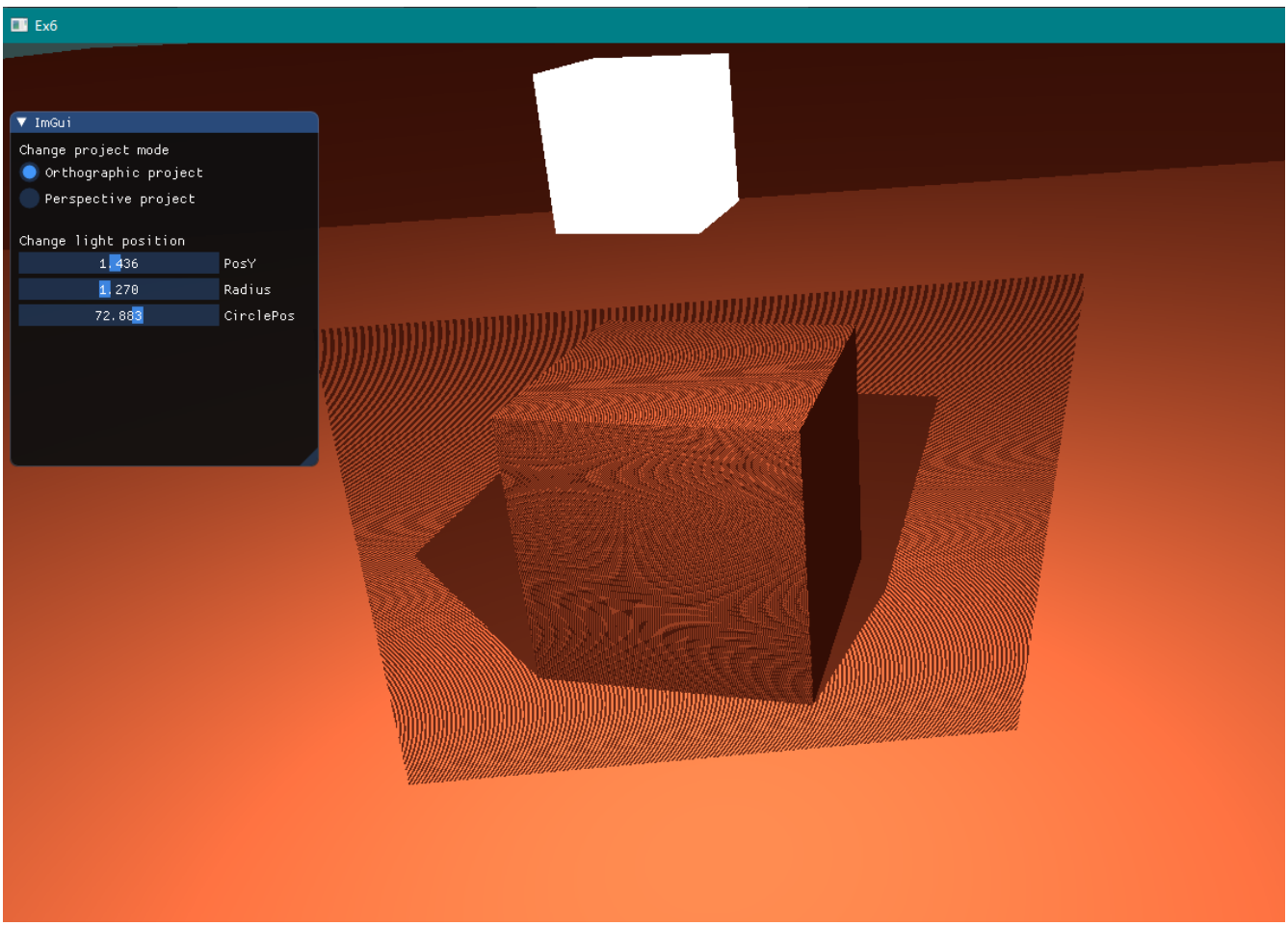
```
uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

void main() {
    gl_Position = projection * view * model * vec4(position, 1.0f);
    vs_out.FragPos = vec3(model * vec4(position, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * normal;
    vs_out.TexCoords = texCoords;
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);
}
```

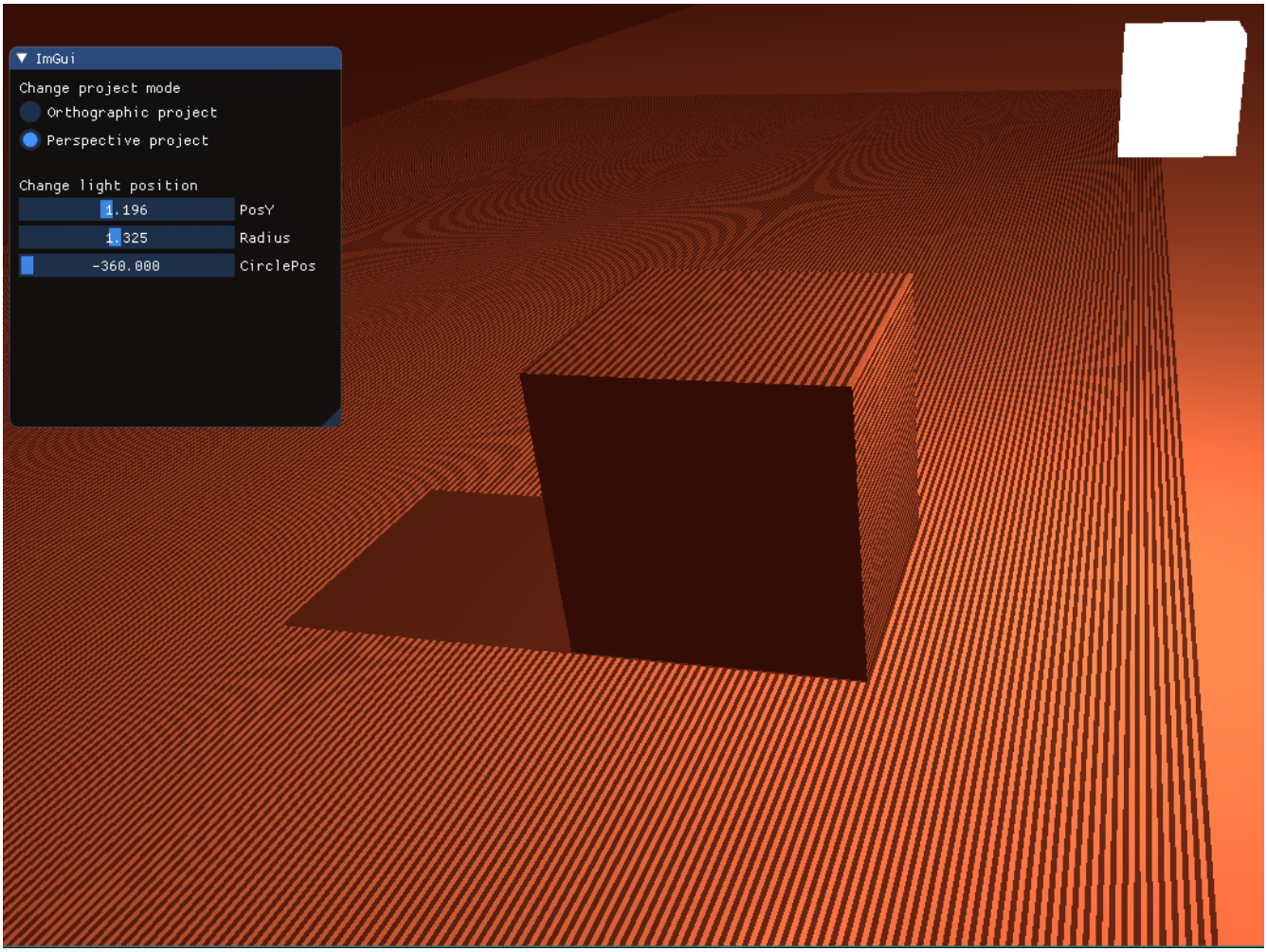
像素着色器使用Blinn-Phong光照模型渲染场景。

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;

uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

//要检查一个片元是否在阴影中，把光空间片元位置转换为裁切空间的标准化设备坐标。当我们在顶点着色器输出一个裁切空间顶点位置到gl_Position时，OpenGL自动进行一个透视除法，将裁切空间坐标的范围-w到w转为-1到1，这要将x、y、z元素除以向量的w元素来实现。
float ShadowCalculation(vec4 fragPosLightSpace) {
    // 执行透视除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 变换到[0,1]的范围
    projCoords = projCoords * 0.5 + 0.5;
    // 取得最近点的深度(使用[0,1]范围下的fragPosLight当坐标)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // 取得当前片元在光源视角下的深度
    float currentDepth = projCoords.z;
    // 检查当前片元是否在阴影中
    float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;

    return shadow;
}

void main() {
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(1.0);
```

```
    // Ambient
    vec3 ambient = 0.15 * color;
    // Diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // Specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // 计算阴影
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    // 接着计算出一个shadow值
    // 当fragment在阴影中时是1.0，在阴影外是0.0。
    // diffuse和specular颜色乘以这个阴影元素
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

    FragColor = vec4(lighting, 1.0f);
}
```

此时的效果如下





阴影优化[bonus]

阴影失真

以上图片中看到图片中有明显的线条样式，这种阴影贴图的不真实感叫做阴影失真。

由于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去采样。当光源以一个角度朝向表面时就会出现这种情况。

使用阴影偏移来解决这个问题。对物体的表面的深度贴图应用一个便宜量，这样片元就不会被错误地认为在表面之下。

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
```

悬浮

使用阴影偏移的一个缺点是你对物体的实际深度应用了平移。偏移有可能足够大，以至于可以看出阴影相对实际物体位置的便宜，这种阴影失真现象称为悬浮。使用正面剔除的方法来解决这个问题

```
glCullFace(GL_FRONT);
RenderSceneToDepthMap();
glCullFace(GL_BACK); // 不要忘记设回原先的culling face
```

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。

在图中看到，光照有一个区域，超出该区域就成为了阴影；这个区域实际上代表着深度贴图的大小，这个贴图投影到了地板上。发生这种情况的原因是我们之前将深度贴图的环绕方式设置成了GL_REPEAT。

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);
GLfloat borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

采样深度贴图0到1坐标范围以外的区域，纹理函数总会返回一个1.0的深度值，阴影值为0.0。结果看起来会更真实。

仍有一部分是黑暗区域。那里的坐标超出了光的正交视锥的远平面。你可以看到这片黑色区域总是出现在光源视锥的极远处。

当一个点比光的远平面还要远时，它的投影坐标的z坐标大于1.0。这种情况下，GL_CLAMP_TO_BORDER环绕方式不起作用，把坐标的z元素和深度贴图的值进行了对比；它总是为大于1.0的z返回true。

只要投影向量的z坐标大于1.0，就把shadow的值强制设为0.0。检查远平面，并将深度贴图限制为一个手工指定的边界颜色，就能解决深度贴图采样超出的问题。

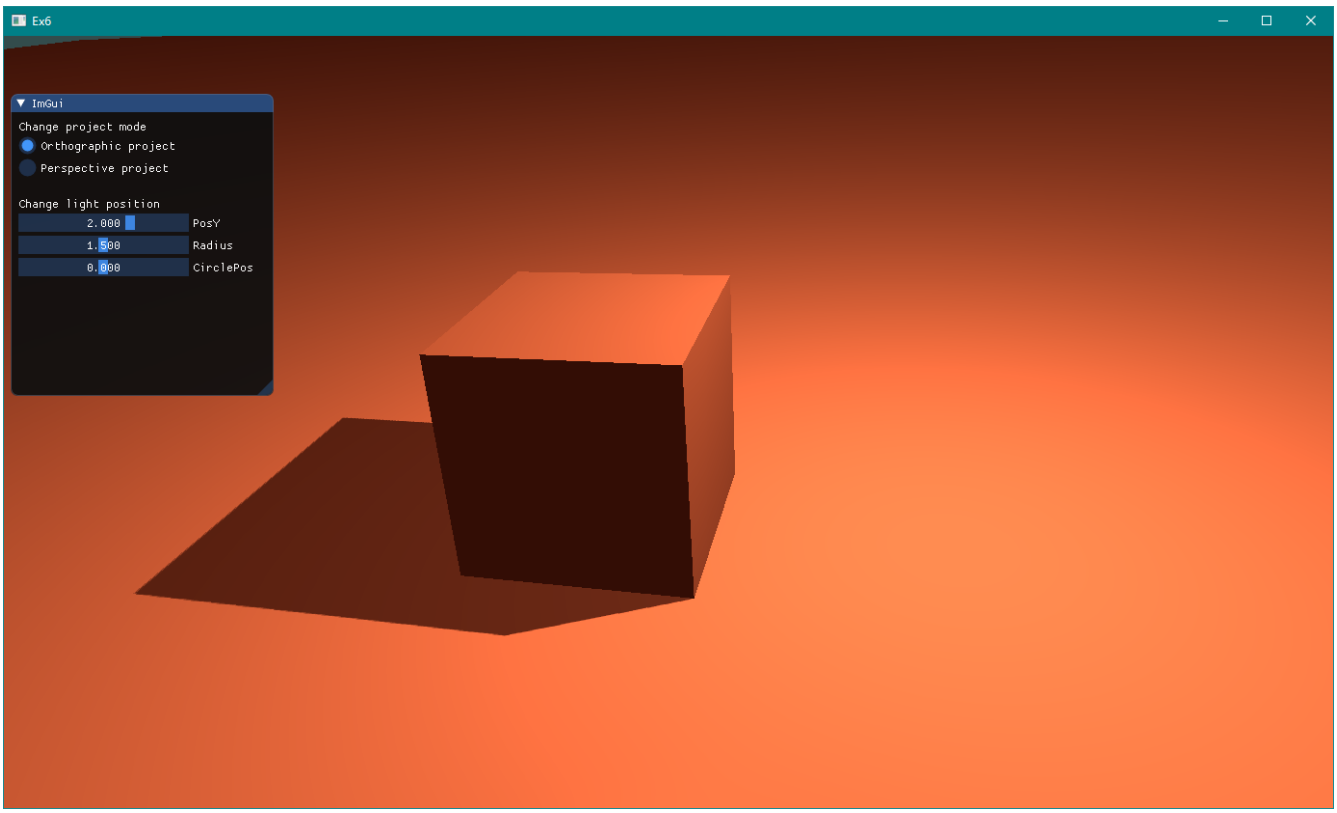
锯齿消除

由于深度贴图有一个固定的解析度，多个片元对应一个纹理像素，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。解决的方法是PCF，一种多个不同过滤方式的组合，能产生柔和阴影，使得它们出现更少的锯齿快和硬边。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在阴影中。所有的次生结构结合在一起进行平均化就得到了柔和阴影。

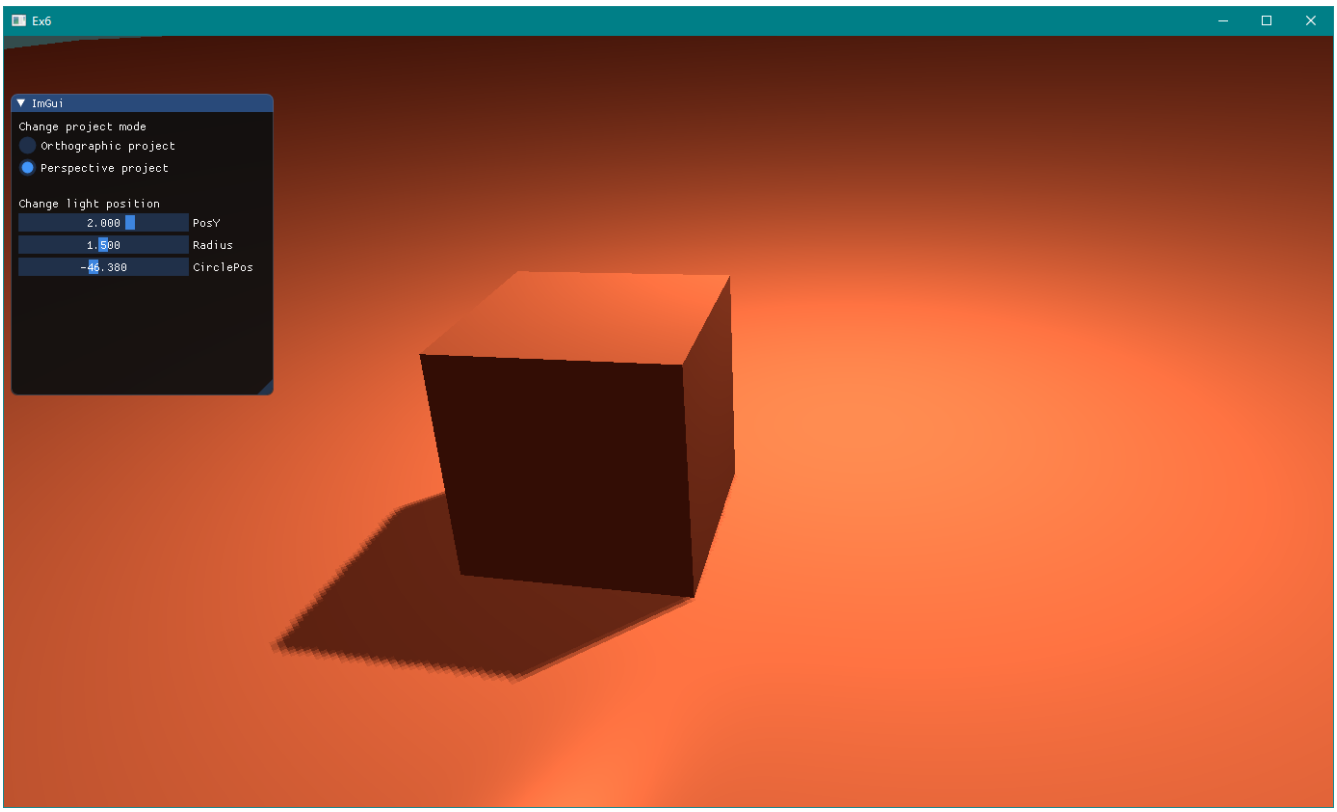
```
// PCF 柔和阴影
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x) {
    for(int y = -1; y <= 1; ++y) {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```

优化阴影结果

光源正交投影

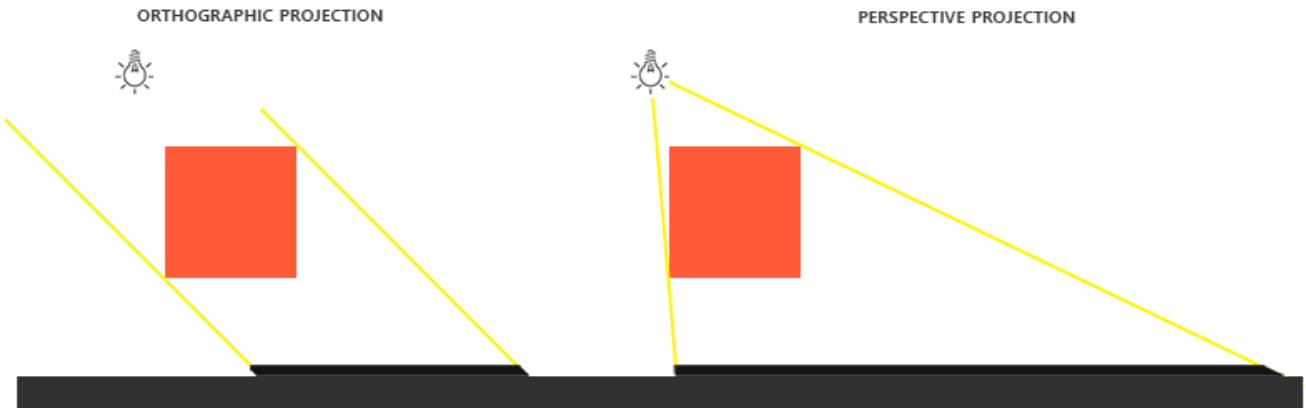


光源透视投影



两种光源投影方式的Shadowing Mapping [bonus]

在渲染深度贴图的时候，正交(Orthographic)和投影(Projection)矩阵之间有所不同。正交投影矩阵并不会将场景用透视图进行变形，所有视线/光线都是平行的，这使它对于定向光来说是个很好的投影矩阵。然而透视投影矩阵，会将所有顶点根据透视关系进行变形，结果因此而不同。下图展示了两种投影方式所产生的不同阴影区域：



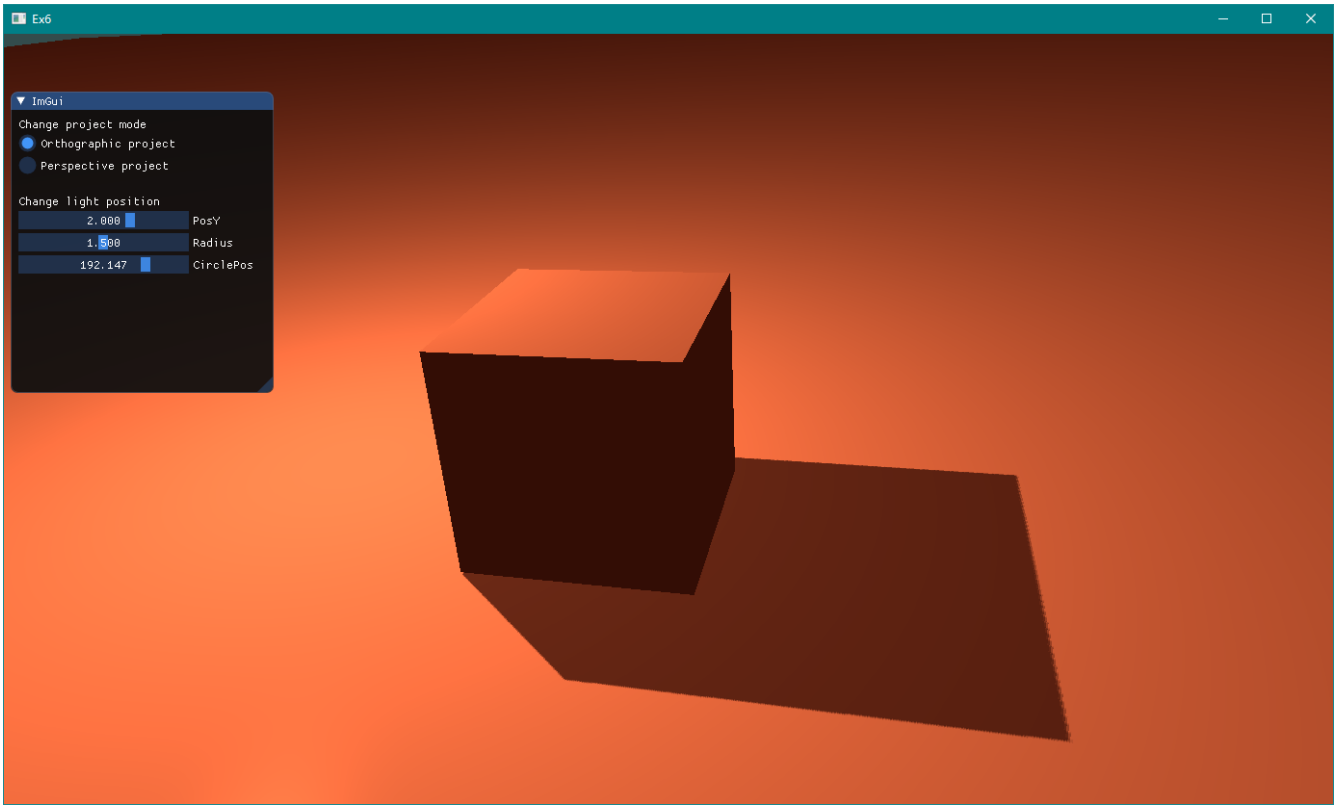
透视投影对于光源来说更合理，不像定向光，它是有自己的位置的。透视投影因此更经常用在点光源和聚光灯上，而正交投影经常用在定向光上。

另一个细微差别是，透视投影矩阵，将深度缓冲视觉化经常会得到一个几乎全白的结果。发生这个是因为透视投影下，深度变成了非线性的深度值，它的大多数可辨范围接近于近平面。为了可以像使用正交投影一样合适的观察到深度值，必须将非线性深度值转变为线性的。

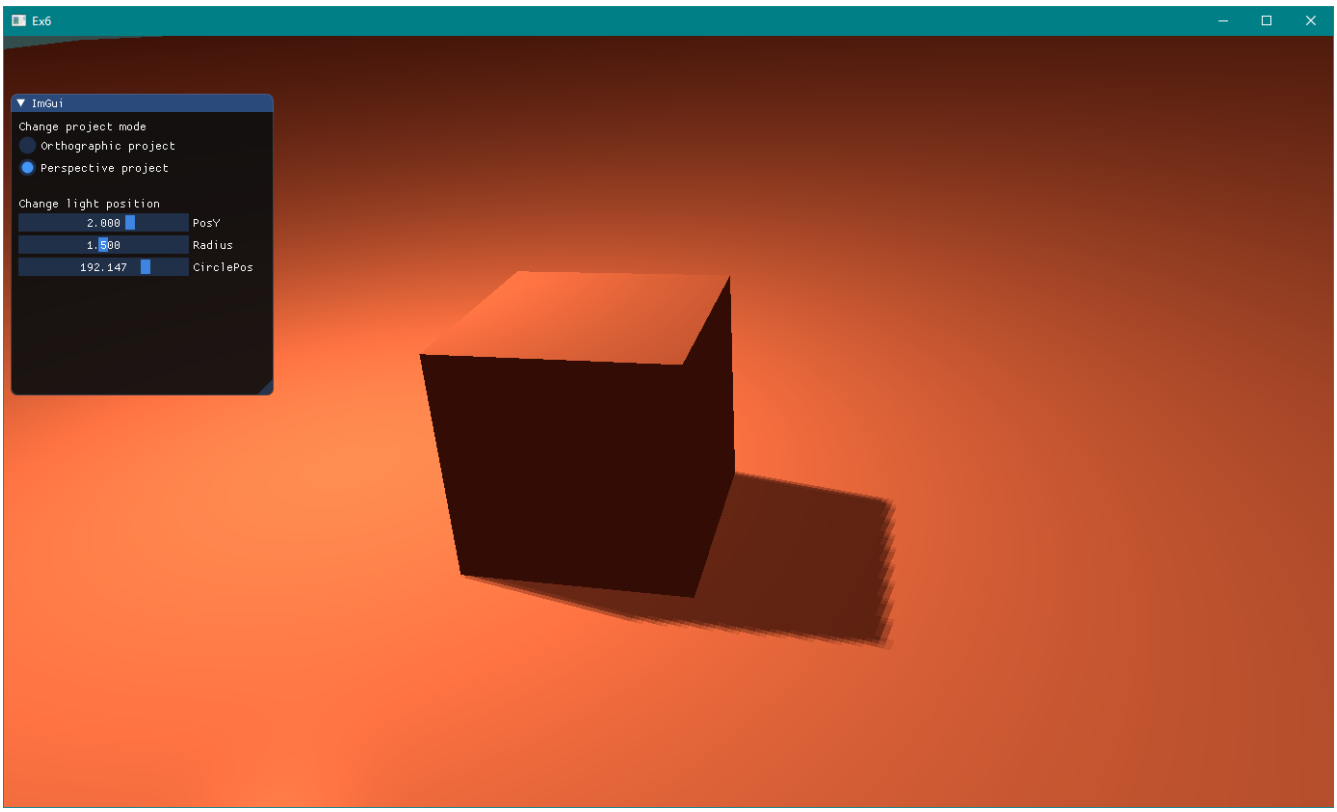
```
#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D depthMap;
uniform float near_plane;
uniform float far_plane;

float LinearizeDepth(float depth) {
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane -
near_plane));
}

void main() {
    float depthValue = texture(depthMap, TexCoords).r;
    color = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // perspective
    // color = vec4(vec3(depthValue), 1.0); // orthographic
}
```

结果对比

正交投影


Ex6
ImGui
Change project mode
Orthographic project
Perspective project
Change light position
2.000
PosY
1.500
Radius
192.147
CirclePos

透视投影


Ex6
ImGui
Change project mode
Orthographic project
Perspective project
Change light position
2.000
PosY
1.500
Radius
192.147
CirclePos