计算机图形学第七次作业

16340028 陈思航

题目

Basic:

- 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
- 修改GUI

Bonus:

- 实现光源在正交/透视两种投影下的Shadowing Mapping
- 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)

阴影

阴影的产生的原因是光线被阻挡。如果从光源射出的部分光线被物体阻挡无法到达物体的表面，那物体该表面就在阴影之中。我们还需要"阴影映射"这一概念，即从光源位置为视觉进行渲染，可以看见的部分则进行电量（进行渲染），而看不见的部分在阴影之中。

在阴影映射中，我们需要测试光线方向上物体的其它点是否有比最近点更远，如果有则更远的点出于阴影之中。为了避免光线遍历带来的极高计算量，我们通过z-buffer即深度缓冲进行。**在深度缓冲中的一个值是摄像机视角下，对应于一个片元的一个0到1之间的深度值。**z-buffer中深度值显示从光源的透视图下见到第一个片元。

深度映射由两个部分组成：

- 渲染深度贴图，在光的方向上的某一点进行场景的渲染。
- 渲染场景，与之前作业无异。

深度贴图

在这一步骤中，我们需要从光的透视图中渲染深度纹理。我们需要为渲染的深度贴图创建帧缓冲对象。之后创建2D纹理，提供给深度缓冲使用，其中纹理格式定义为`GL_DEPTH_COMPONENT`，高度和宽度均为1024。

```
/ 为渲染的深度贴图创建一个帧缓冲对象
    GLuint depthMapFBO;
    glGenFramebuffers(1, &depthMapFBO);
    // 创建深度纹理
    GLuint 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_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

下一步，我们将生成的纹理作为深度缓冲。在这个过程中，将`glDrawBuffer`与`glReadBuffer`设置为`GL_NONE`，因为我们仅需要其深度信息而不需要颜色信息。

```
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);
```

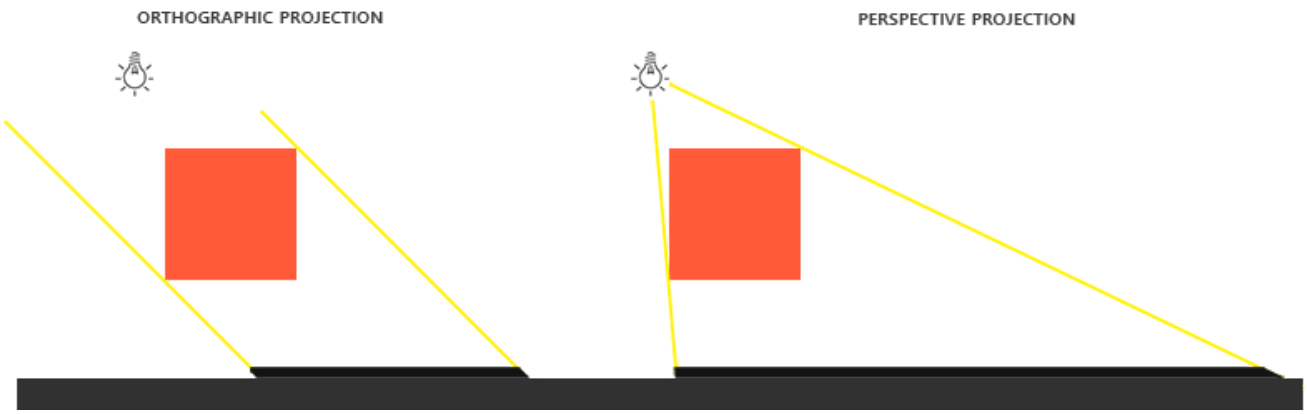
渲染

在while循环中，我们需要先渲染深度贴图，之后再渲染设定好的场景。首先，我们需要进行相对于光源位置投影类型的选择，选择透视投影还是正交投影。

```
 // 将为光源使用正交或是投视投影矩阵
        GLfloat near_plane = 1.0f, far_plane = 7.5f;
        if (currentModel == orthoType) {
            lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane,
far_plane);
        }
        else {
            lightProjection = glm::perspective(100.0f, (float)SHADOW_WIDTH /
(float)SHADOW_HEIGHT, near_plane, far_plane);
        }
```

正交投影与透视投影

正交投影矩阵不会讲场景用透视图进行变形，所有光线是平行的，而对于透视投影，会根据近大远小的透视关系进行变形。透视投影相对于光源更加合理，特别是点光源或是聚光灯上。



光源空间变换

我们需要从光源位置的视野来投影场景中的不同物体。即我们需要通过透视投影矩阵或者正交投影矩阵进行物体表面点位置的变换。投影矩阵间接决定可视区域，而如果图元不在贴图区域中则不产生阴影，所以我们需要保证投影视锥的大小。同时，为了使得物体表面点能够变换到光源位置视角的可见空间中，我们使用`glm::lookAt`函数。

```
        lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
        lightSpaceMatrix = lightProjection * lightView;

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

其中`lightSpaceMatrix`为投影与`lookAt`矩阵结合的变换矩阵，只要给`shader`提供光空间的投影与视图矩阵，我们就可以渲染深度贴图了。

```
// 渲染深度贴图
        glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
        glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
        glClear(GL_DEPTH_BUFFER_BIT);
        RenderScene(simpleDepthShader);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);

        glCullFace(GL_BACK);
```

使用深度贴图渲染场景

在渲染前我们需要修改`glViewport`，因为阴影贴图与我们渲染场景有不同的解析度。为了防止深度贴图太小或者不完整，务必修改`glViewport`。

渲染深度贴图：

```
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
```

渲染场景：

```
glViewport(0, 0, windowsWidth, windowsHeight);
```

场景渲染与之前没有很大的差异，需要设定好边境颜色等参数。之后通过视图矩阵和投影矩阵进行点从世界坐标到视图坐标的变换。

```
// 背景颜色
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
        glClear(GL_DEPTH_BUFFER_BIT);
        glEnable(GL_DEPTH_TEST);
        shader.use();
        glm::mat4 projection(1.0f);
        glm::mat4 view(1.0f);
        glm::mat4 model(1.0f);
        view = glm::lookAt(Camera::getInstance()->Position, Camera::getInstance()->Position
+ Camera::getInstance()->Front, Camera::getInstance()->Up);
        projection = glm::perspective(glm::radians(Camera::getInstance()->Zoom),
(float)windowsWidth / (float)windowsHeight, 0.1f, 100.0f);
```

```
// 设置可变的着色器参数
        shader.setMat4("projection", projection);
        shader.setMat4("view", view);
        shader.setVec3("viewPos", Camera::getInstance()->Position);
        shader.setVec3("lightPos", lightPos);
        shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, depthMap);
```

在这之后，我们需要画出物体：

```
// 画出物体
RenderScene(shader);
```

该函数的参数是一个着色器程序，调用所有相关的绘制函数，在需要的地方设置相应的模型矩阵。最终在光源位置对应的透视图视角下，用每个可见片元的最近深度填充了深度缓冲。将纹理映射到2D的四边形上从而在屏幕上显示。

RenderScene 渲染场景

与之前的作业一样，设定物体的位置、旋转轴、旋转角度以及颜色，并进行渲染。作业中需要渲染平面以及立方体。

```
void RenderScene(Shader &shader)
{
    // 平面
    glm::mat4 model(0.5f);
    shader.setMat4("model", model);
    shader.setVec3("objectColor", glm::vec3(1.0f, 0.0f, 0.0f));
    glBindVertexArray(planeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);

    // 物体1
    model = glm::mat4(1.0f);
```

```
    model = glm::rotate(model, ((float)glfwGetTime() * 0.5f), glm::vec3(1.0f, 1.0f, 0.0f));
    model = glm::translate(model, glm::vec3(1.0f, 3.0f, 2.0f));
    shader.setMat4("model", model);
    shader.setVec3("objectColor", glm::vec3(0.0f, 1.0f, 1.0f));
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
}
```

除此之外，我们还需要画出光源的立方体，同上次作业一样：

```
// 画出光源
        lampShader.use();
        model = glm::translate(model, lightPos);
        model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
        lampShader.setMat4("projection", projection);
        lampShader.setMat4("model", model);
        lampShader.setMat4("view", view);

        glBindVertexArray(lightVAO);
        glDrawArrays(GL_TRIANGLES, 0, 36);
```

ImGui

这一部分，需要添加投影类型的选择，包括正交投影与透视投影。初次之外还添加了是否光源移动的选项。

```
/**
        * 使用IMGUI
        */
        ImGui_ImplOpenGL3_NewFrame();
        ImGui_ImplGlfw_NewFrame();
        ImGui::NewFrame();
        ImGui::Begin("Attributes");
        ImGui::Text("Chooce projection type");
        ImGui::RadioButton("ortho type", &currentModel, orthoType);
        ImGui::RadioButton("perspective type", &currentModel, perspectiveType);
        ImGui::Separator();
        ImGui::Checkbox("Move the light", &isMove);

        ImGui::End();
```

渲染至深度贴图

通过`shadow_mapping_depth`着色器，将顶点变换到光源位置对应的空间。顶点着色器如下：

```
#version 450 core
layout (location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}
```

该顶点着色器将模型中的顶点通过矩阵`lightSpaceMatrix`变换到光空间中。

同时我们需要定义一个空的段着色器：

```
#version 450 core

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

运行完该段着色器后（我们仅需要其深度值而不需要其颜色值），深度缓冲被更新。

阴影渲染

在顶点着色器`shadow.vs`中，我们依旧利用立方体的位置、法向量以及纹理坐标。将顶点变换到光空间。`FragPosLightSpace`向量是输出到段着色器中的，它是变换矩阵`lightSpaceMatrix`将世界坐标系中的顶点位置映射到光空间对应坐标系中的结果。

```
#version 450 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()
```

```
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0f));
    vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;
    vs_out.TexCoords = aTexCoords;
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0f);

}
```

在段着色器中，光照模型选用`Blinn-Phong`。其中的`diffuseTexture`为0，即阴影外为0.0。而`shadowMap`为1，即fragment在阴影中为1.0。`diffuse`和`specular`颜色会乘以该阴影元素。由于散射，阴影并非全黑，所以将ambient分量从乘法中剔除。

```
#version 450 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;
uniform vec3 objectColor;


float ShadowCalculation(vec4 fragPosLightSpace, vec3 normal, vec3 lightDir)
{
...
}

void main()
{
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.5f);
    // ambient环境光
    vec3 ambient = 0.4 * lightColor;
    // diffuse 漫反射
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // specular 镜面
    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, normal, lightDir);

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

在阴影计算部分，我们通过`shadowCalculation`函数进行阴影计算，而在段0着色器的最后，将`diffuse`和`specular`乘以`(1-阴影元素)`，以表示该片元不在阴影中的占比。该着色器另外的两个输入`TexCoords`与`FragPosLightSpace`为渲染得到的深度贴图与偏远位置。

shadowCalculation函数

在该函数中，首先要需要将裁切空间坐标的范围变为[-1, 1]，即将x、y、z元素除以w。在这之后，将其从[-1, 1]变换到[0, 1]，从而获得投影坐标。其投影坐标直接对应变换过的NDC坐标，从而得到光源位置下的深度。

```
     // 执行透视算法，将将w转化为(-1, 1)
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 从(-1,1)变换到(0,1)
    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;
```

之后，通过获取投影向量的z坐标，等于来自光源位置透视视角片元的深度。如果高于`closetDepth`则设为1，因为片元在阴影中，否则为0。

```
float currentDepth = projCoords.z;
float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;
```

函数整体如下（含有改进部分）：

```
float ShadowCalculation(vec4 fragPosLightSpace, vec3 normal, vec3 lightDir)
{
    // 执行透视算法，将将w转化为(-1, 1)
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 从(-1,1)变换到(0,1)
    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;
    // 简单获取投影向量的z坐标，等于来自光的透视视角的片元的深度
    float currentDepth = projCoords.z;

    // 避免阴影失真
    // 使用点乘
    float bias = max(0.5 * (1.0 - dot(normal, lightDir)), 0.005);

    // 从纹理像素四周对深度贴图采样，并取其平均值
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
```

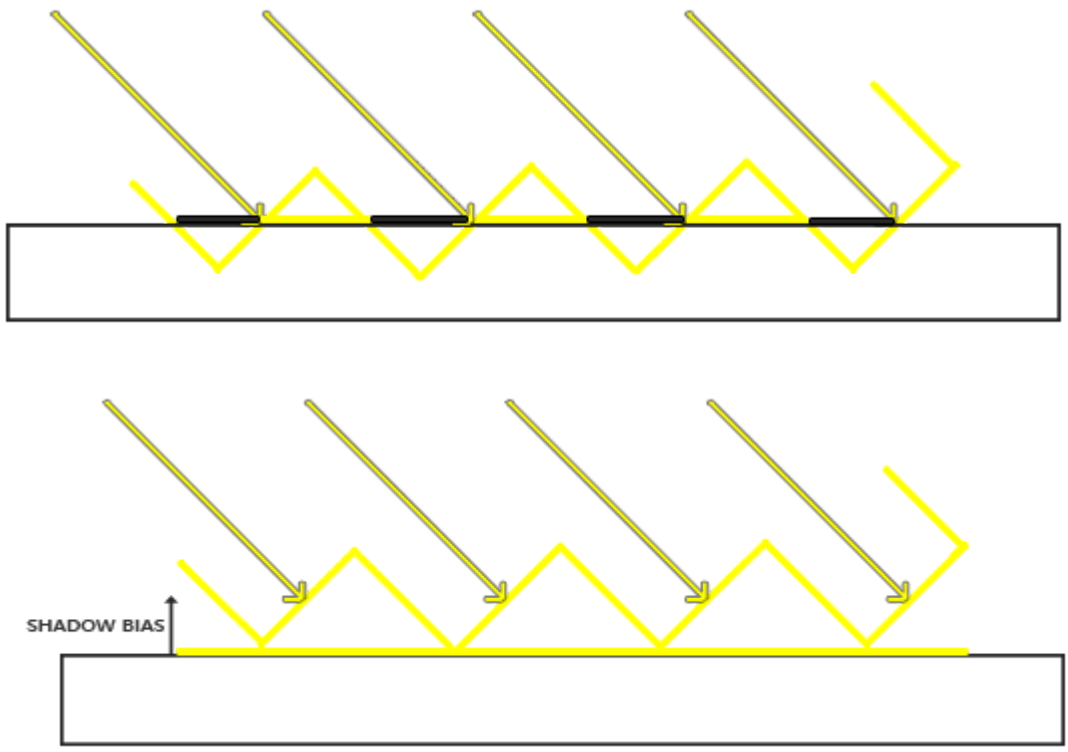
```
    for(int x = -1; x <= 1; ++x)
    {
        for(int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
            shadow += currentDepth - bias > pcfDepth  ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;

    // 只投影向量的z坐标大于1.0则shadow的值强制设为0.0
    if(projCoords.z > 1.0){
        shadow = 0.0;
    }
    return shadow;
}
```

改进阴影计算过程

阴影失真避免

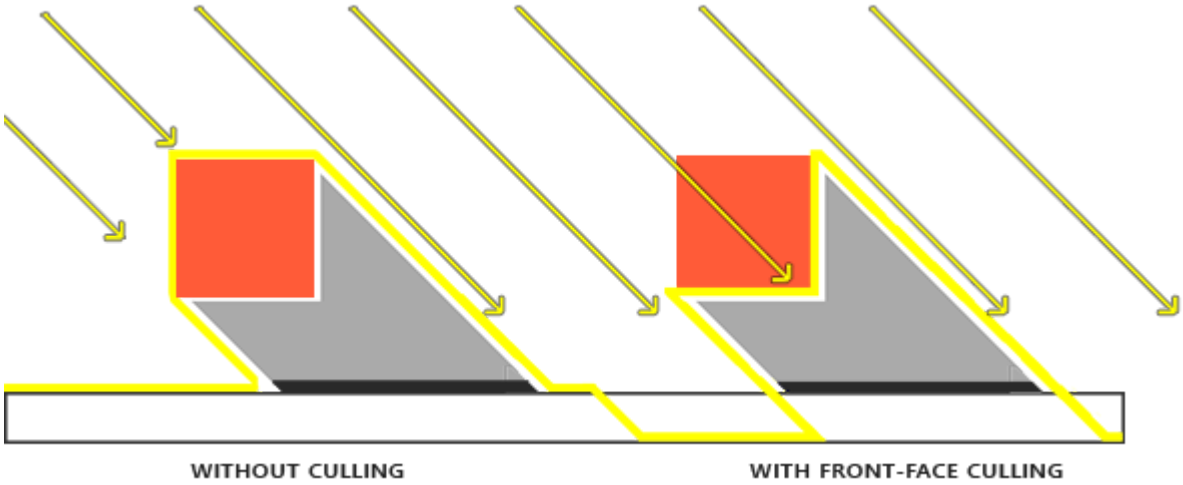
阴影失真是因为阴影贴图受限于解析度，在距离光源比较远的时候，多个片元可能从深度的同一个值进行采样。而光源以一个角度朝向表面的时候，多个片元从同一个斜坡的深度纹理像素中进行采样，阴影产生了差异，即阴影偏移。我们通过点乘获得偏移量（最小0.005，最大为0.5）。如果物体表面几乎与光源垂直，偏移量最小，否则很大。

```
// 使用点乘
    float bias = max(0.5 * (1.0 - dot(normal, lightDir)), 0.005);
```



避免悬浮

使用阴影偏移可能会导致悬浮问题，此时物体看起来悬浮在表面上面，也称作`Peter Panning`。我们需要剔除正面解决该问题，因为只需要深度贴图的深度值，实体物体使用其正面或背面都可以。即使是背面深度出现错误，因为我们看不见，所以也不会被发现。



过程如下：

```
// 为了修复peter游移，进行正面剔除
glCullFace(GL_FRONT);
...
glCullFace(GL_BACK);
```

这样便可以解决悬浮问题，但是在地板时是无效的，因为这种方法只针对实体物体。地面是一个单独的平面，不会被完全剔除。也就是说，正面剔除完全移除了地板。另外，接近阴影的物体也会出现不正确的效果。需要考虑到何时使用正面剔除对物体才有意义。

避免采样过多

在阴影计算中，光源位置视锥不可见的区域都被认为出于阴影中，这样可能出现物体表面处于阴影之中的情况。此时超出光的视锥的投影坐标比1.0大，采样的深度纹理超过了[0, 1]。深度贴图的纹理环绕方式设置为`GL_CLAMP_TO_BORDER`，并且让超过1.0的深度贴图的坐标设置为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);
GLfloat borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

结果就是深度贴图[0, 1]范围之外的区域总会返回一个0.0或是1.0的深度值。但此时仍有部分区域坐标超出了光的正交视锥的远平面。我们需要在投影向量的z坐标大于1.0将其设为0.0即可。

```
// 只投影向量的z坐标大于1.0则shadow的值强制设为0.0
    if(projCoords.z > 1.0){
        shadow = 0.0;
    }
```

之后，只有在深度贴图范围以内被投影的坐标才会显示为阴影。

PCF

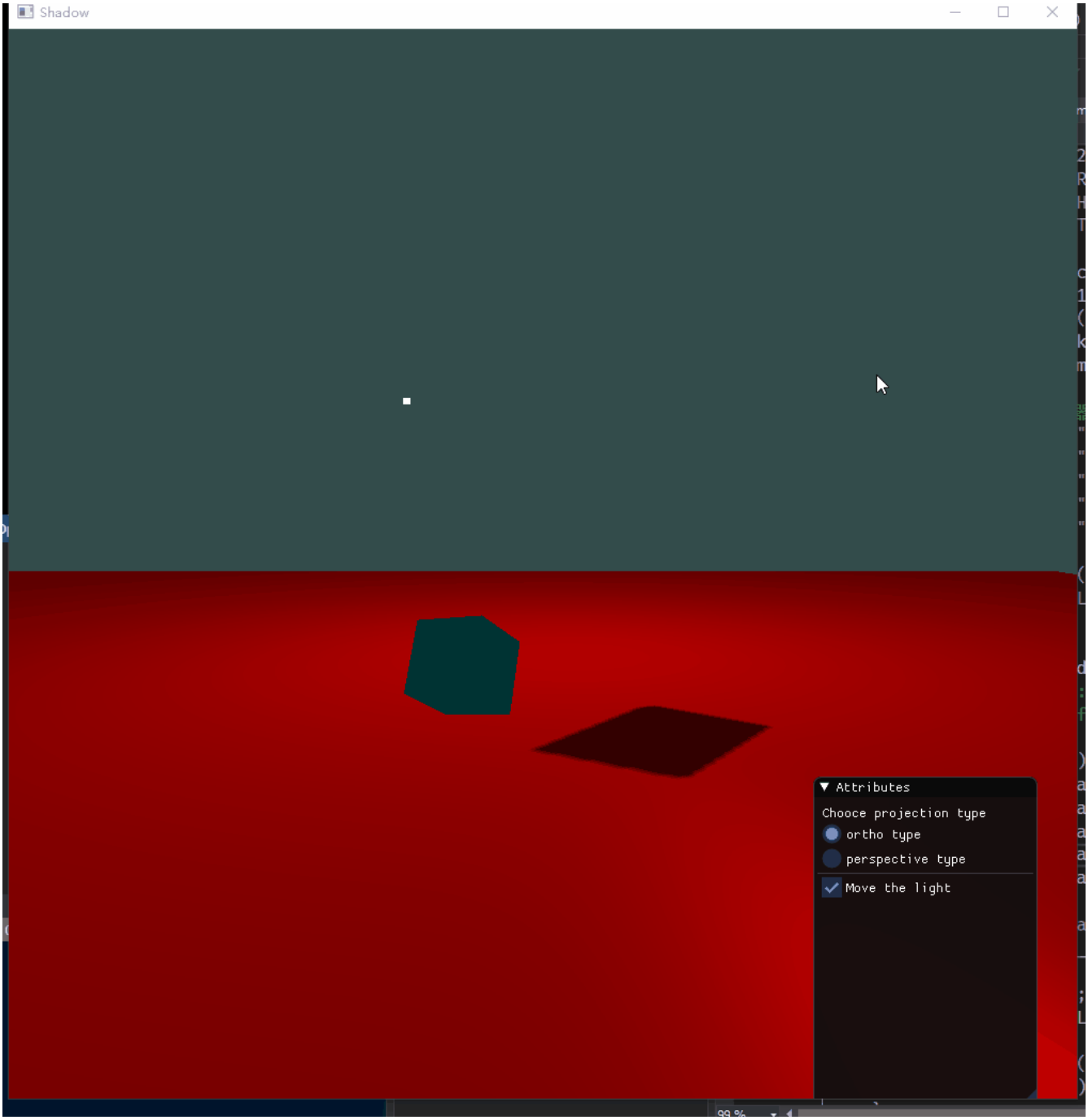
在放大的时候，阴影映射对解析度依赖变大。这是因为渲染深度贴图的时候，解析度是固定的，即多个片元对应一个纹理像素，结果多个片元从深度贴图的相同深度进行采样，产生了锯齿。

（以上图来自于learn openGL）

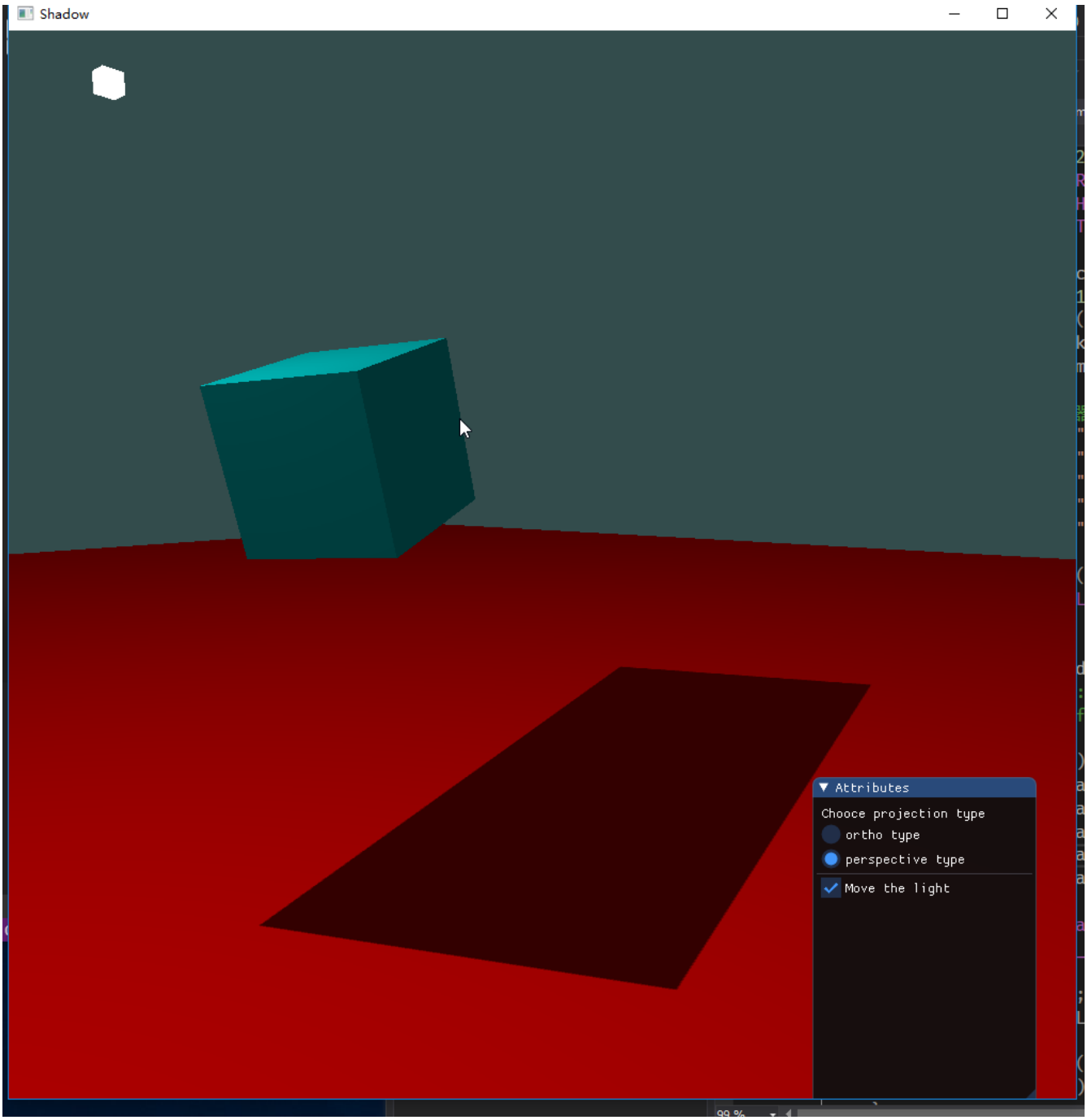
通过PCF能够较好解决这个问题，它通过多个不同过滤方式的组合，产生比较柔和的阴影以避免锯齿。也就是说，从深度贴图中进行多次采样（这里采样区域大小为3×3），而每次采样纹理坐标都不同，也就是说独立样本可以出现在阴影中，也可以在阴影外。最后对结果求出均值(即`shadow = shadow / 9`)，得到比较柔和的阴影。但是在放大时因为解析度的问题依旧存在不真实感。

实验结果

正交投影：


Shadow
Attributes
Chooce projection type
ortho type
perspective type
Move the light

透视投影：


Shadow
Attributes
Chooce projection type
ortho type
perspective type
Move the light