# 计算机图形学第七次作业

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)

## Basic

根据教程，实现阴影的映射需要两步：

- 首先，需要渲染深度贴图
- 使用生成的深度贴图计算片元是否在阴影中，以摄像机方向。

**具体步骤如下**：

1. 一开始我们需要生成一张深度贴图，这个过程需要我们首先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，配置一系列参数，然后把我们生成的深度纹理作为帧缓冲的深度缓冲：

```
//depth map FBO configure
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// - Create depth texture
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);
// 防止纹理贴图在远处重复渲染
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);

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);
```

2. 然后是光源空间的变换，通过使用一个lightMatrix矩阵来变换物体，把他们变换到从光源视角可见的空间中，这个lightMatrix由lightProjection矩阵和lightView矩阵相乘得到，前者是确定光源的投影矩阵，后者是通过视图矩阵来变换物体。而投影矩阵我们可以选择是正交投影还是透视投影。

```
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
GLfloat near_plane = 1.0f, far_plane = 7.5f;
if (mode ==1)
{
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
else
{
    lightProjection = glm::perspective(124.0f, (float)SHADOW_WIDTH /
(float)SHADOW_HEIGHT, near_plane, far_plane);
}
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
lightSpaceMatrix = lightProjection * lightView;
depthShader.use();
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
```

3. 在渲染至深度贴图的时候，重新设置glViewPort，在这一阶段，着色器没有很多工作，对于顶点着色器来说，只需要将一个单独模型的一个顶点，使用lightSpaceMatrix变换到光空间中。而片段着色器会空着，因为运行完之后，深度缓冲会被自动更新。

```
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
RenderScene(depthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glCullFace(GL_BACK);
```

4. 最后，修改viewport为屏幕大小，因为阴影贴图经常和我们原来渲染的场景（通常是窗口解析度）有着不同的解析度，我们需要改变视口（viewport）的参数以适应阴影贴图的尺寸。如果我们忘了更新视口参数，最后的深度贴图要么太小要么就不完整。

**关于阴影的渲染在着色器中完成**：

1. 首先是新建了一个FragPosLightSpace变量，这个变量描述的是顶点位置在光源空间中的坐标：

```
vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);
```

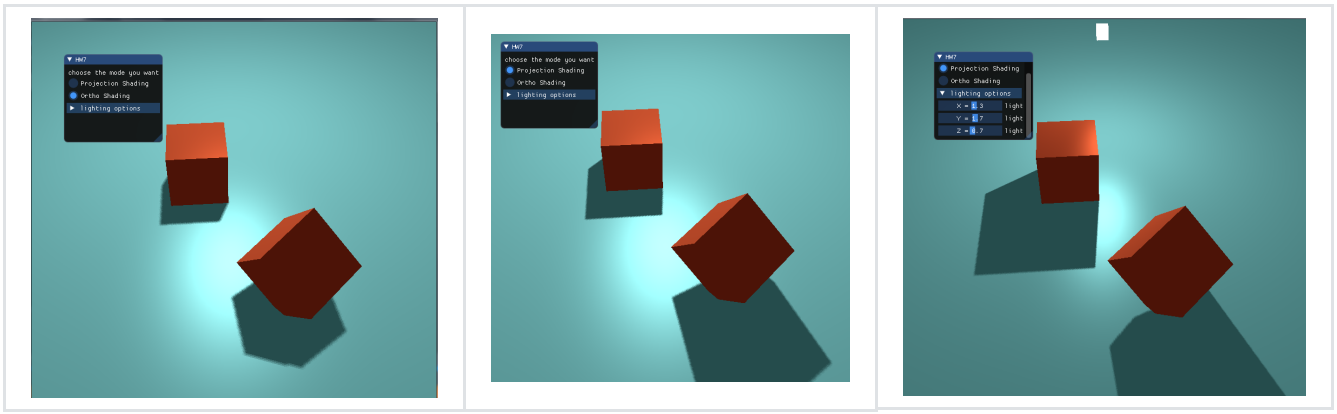
2. 在片段着色器中，首先需要判断一个像素是否在阴影中，通过顶点着色器传进来的FragPosLightSpace以及shadowMap的值得出closeDepth和currentDepth，判断一个片元是否在阴影中。

```
// 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;
```

3. 使用bias方法或者PCF方法或者原生方法求出一个阴影强度分量shadow
4. 最后在计算光照的公式上加上阴影的影响

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

实验结果如下：



## Bonus

实现光源在正交/透视两种投影下的Shadowing Mapping只需要改变lightProjection矩阵即可：

```
if (mode ==1)
{
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
else
{
    lightProjection = glm::perspective(124.0f, (float)SHADOW_WIDTH / (float)SHADOW_HEIGHT,
near_plane, far_plane);
}
```

### 阴影的改进方法

1. 对阴影失真的修复：

使用一个阴影偏移来解决，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，这样整个表面就正确地被照亮，没有任何阴影：

```
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);
```

2. 悬浮修正：

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偏移，我们可以使用一个叫技巧解决大部分的Peter panning问题：当渲染深度贴图时候使用正面剔除（front face culling）：

```
//render depth of scene to texture
glCullFace(GL_FRONT);//正面剔除
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
GLfloat near_plane = 1.0f, far_plane = 7.5f;
if (mode ==1)
{
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
else
{
lightProjection = glm::perspective(124.0f, (float)SHADOW_WIDTH / (float)SHADOW_HEIGHT,
near_plane, far_plane);
}
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
lightSpaceMatrix = lightProjection * lightView;
depthShader.use();
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
RenderScene(depthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glCullFace(GL_BACK);//反面剔除
```

对于本程序而言，使用悬浮修正与否并没有很大的差别。

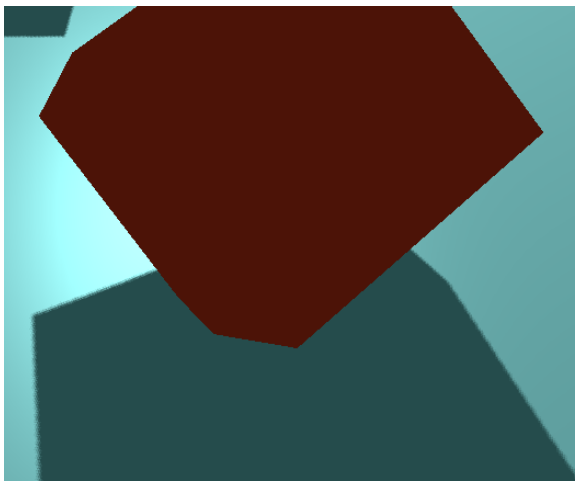
3. 使用PCF：

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，尽可能的消除锯齿状阴影。

这个方法是用在片段着色器中。

```
// 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;
```

效果如下：

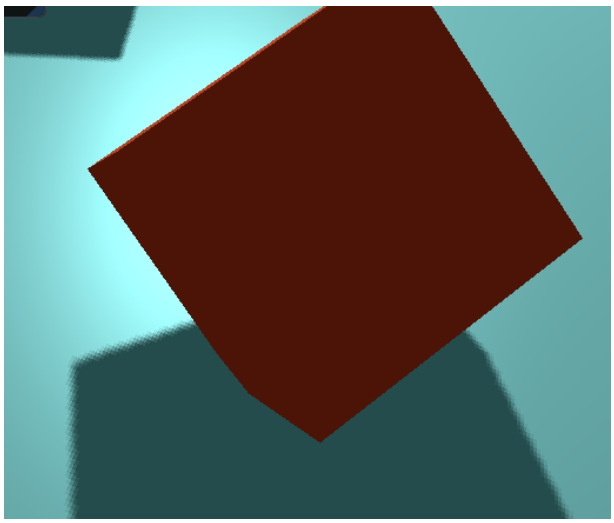


使用PCF

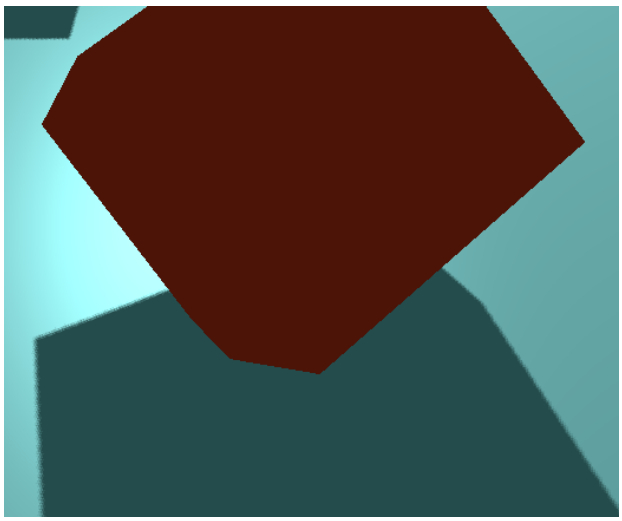
不使用PCF

可以看到，不使用PCF会出现明显的锯齿状阴影。

4. 调整depthMap分辨率



分辨率为1024*1024



分辨率为2048*2048

可以看到边缘更为清晰了。