homework7

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

在场景中渲染一个平面和一个cube，需要分别给cube和plane的VAO绑定对应的顶点数据，然后进行渲染：

```
    // plane
    glm::mat4 model;
    shader.setMat4("model", glm::value_ptr(model));
    shader.setFloat3("objectColor", glm::value_ptr(glm::vec3(0.7f, 0.7f, 0.7f)));
    glBindVertexArray(planeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);

    // cube
    model = glm::mat4();
    shader.setMat4("model", glm::value_ptr(model));
    shader.setFloat3("objectColor", glm::value_ptr(glm::vec3(0.384f, 0.749f, 0.678f)));
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
```

Shadow Mapping 算法实现：

1)以光源视角渲染场景，得到深度图(DepthMap)，并存储为texture;
2)以camera视角渲染场景，使用Shadowing Mapping算法(比较当前深度值与在DepthMap Texture的深度 值)，决定某个点是否在阴影下。
具体过程如下：

1. 首先需要创建一个2D纹理贴图（Texture），用于后面以光源视角渲染场景时存储深度图（depthMap）

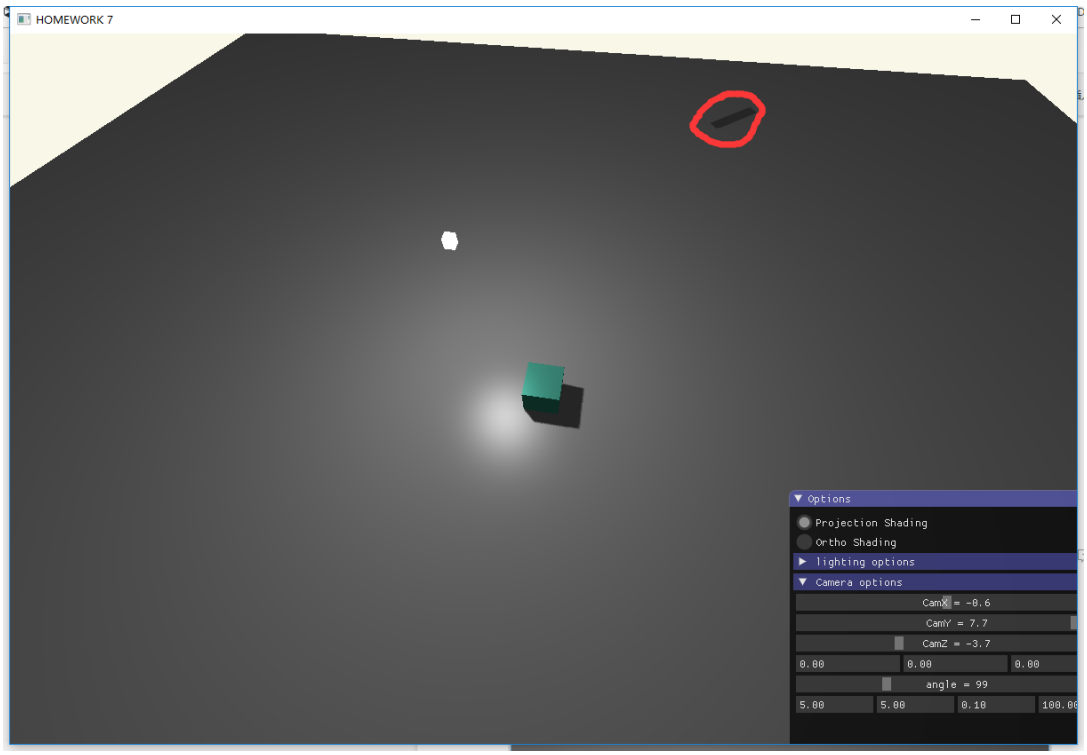
```
    // 创建2D纹理贴图
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
```

```
    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);
    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);
```

这里要注意，在设置纹理 `GL_REPEAT` 参数后，需要添加 `GL_TEXTURE_BORDER_COLOR` 参数来防止纹理重复渲染。如下面代码所示:

```
    // 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
```

若没有上面这段代码，可能会出现一些不应该出现的阴影。



2. 计算将渲染视角空间转移到光源视角空间的变换矩阵，由 projection矩阵 * view矩阵可得:

```
    glm::mat4 lightProjection, lightView;
    glm::mat4 lightSpaceMatrix;
    GLfloat near_plane = 1.0f, far_plane = 7.5f;
    if (mode == 1) {
        // 正交投影矩阵
        lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f,
near_plane, far_plane);
    }
    else {
        // 透射投影矩阵
```

```
        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;
```

3. 修改 glViewPort，进行渲染深度纹理贴图，并将深度信息保存起来。

```
    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);
```

4. 最后修改 glViewPort 回到原始屏幕大小，使用正常渲染场景的方式，利用上一步得到的深度纹理贴图进行渲染。
5. 对于阴影的渲染和判断则在着色器的实现中完成，包含以下过程：
   a. 顶点着色器中添加计算一个变量 FragPosLightSpace ，表示顶点位置在光源观察空间中的位置坐标：

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

   b. 在片段着色器中根据顶点着色器传来的 FragPosLightSpace ，以及通过 Uniform 传进来的深度贴图 ShadowMap ， 算出 closestDepth 和 currentDepth ，判断该点是否在阴影中。

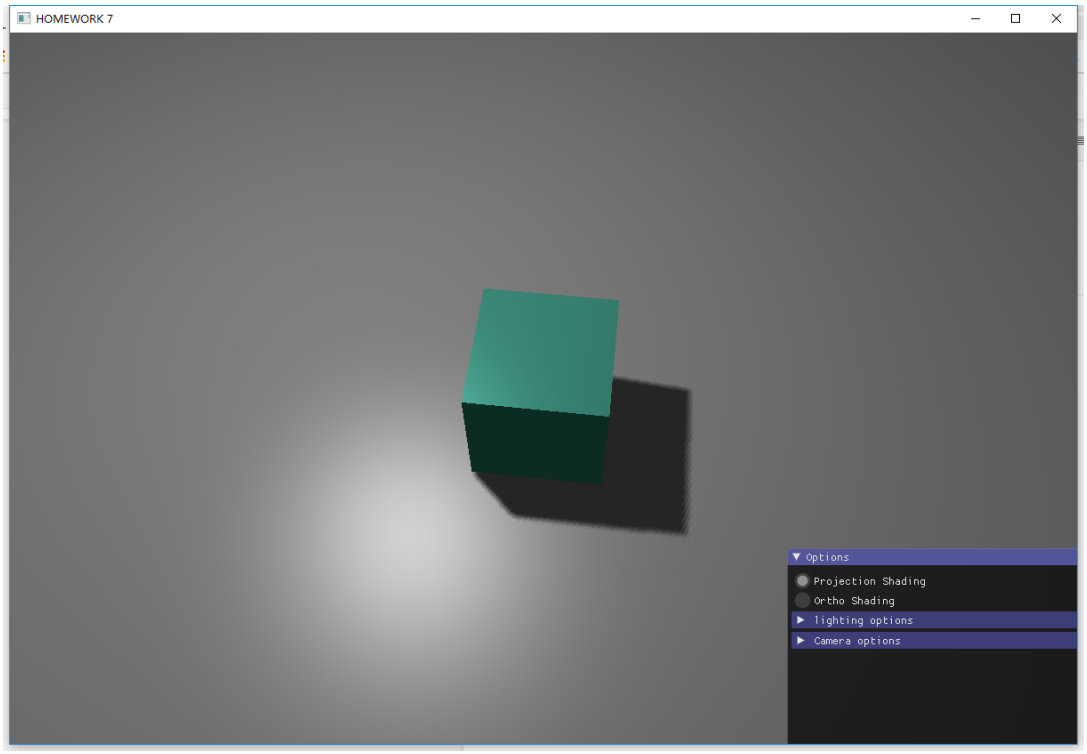
```
   vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
   projCoords = projCoords * 0.5 + 0.5;
   float closestDepth = texture(shadowMap, projCoords.xy).r;
   float currentDepth = projCoords.z;
```

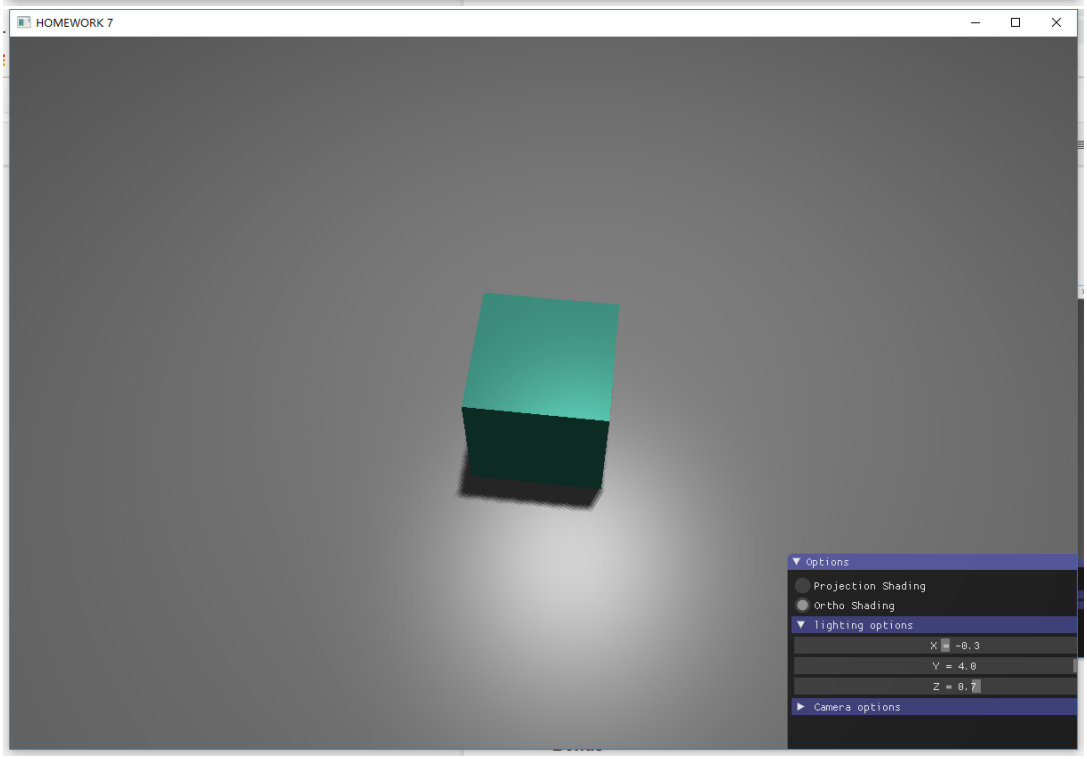
   c. 采用 naive 的方法或者 bias 方法算出表示阴影强度的变量：shadow
   d. 在phong光照模型的计算公式中加上 shadow 的影响，即对于漫反射和镜面反射分量都乘上shadow系数：

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

   e. 由此可以渲染出正确的阴影。

运行效果如下：





Bonus

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

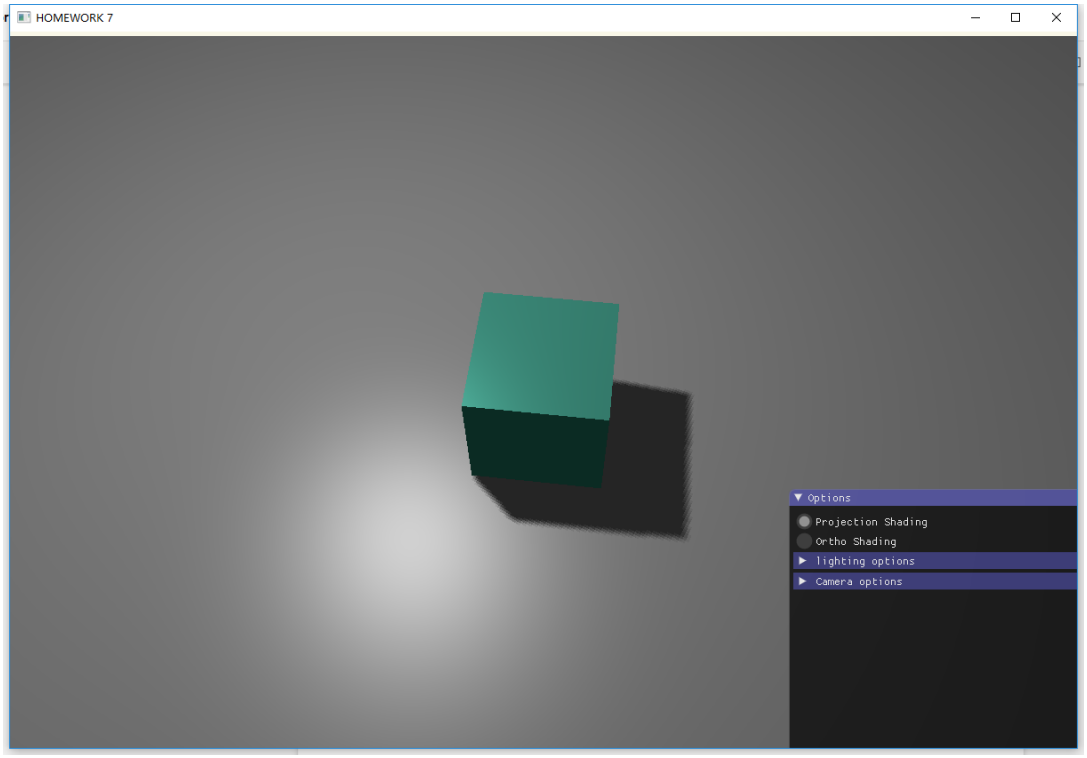
正交/透视投影只需要在GUI中控制显示模式，在render loop中通过判断mode变量来调用glm::ortho 或者 glm::perspective 函数即可实现，并且传入需要的变量参数。

```
    if (mode == 1) {
        // 正交投影矩阵
        lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f,
near_plane, far_plane);
    }
    else {
```

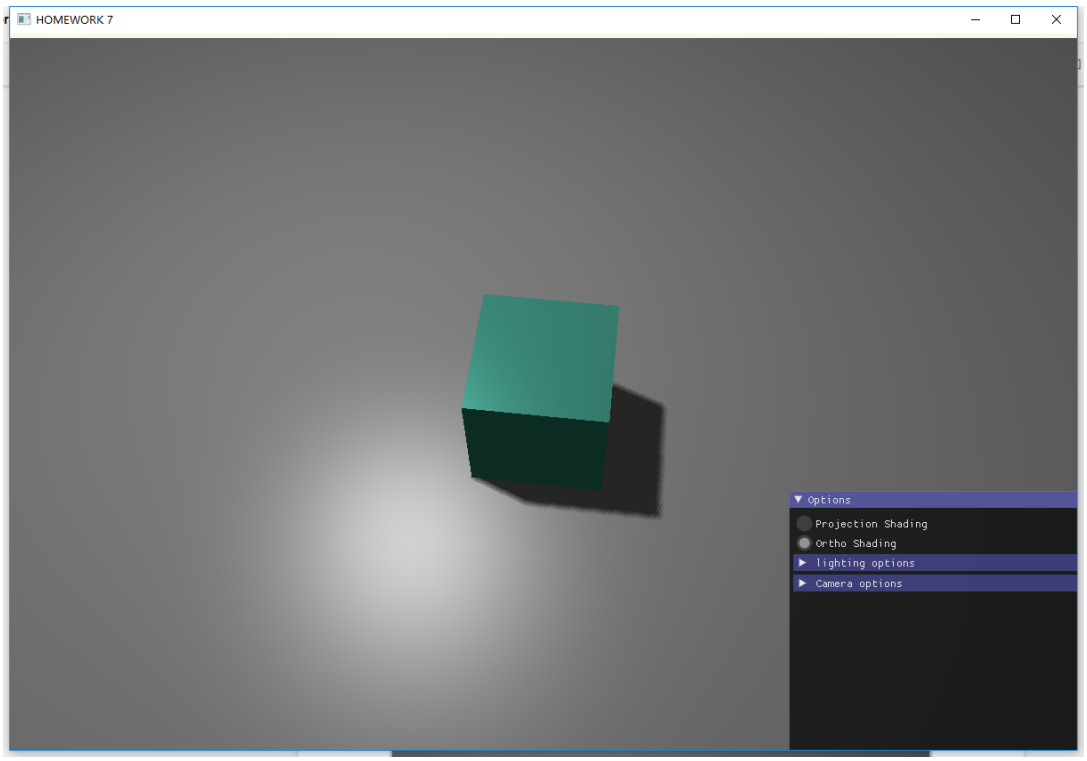
```
        // 透射投影矩阵
        lightProjection = glm::perspective(124.0f, (float)SHADOW_WIDTH / (float)SHADOW_HEIGHT, near_plane, far_plane);
    }

```

透视投影：



正交投影：



优化 Shadow Mapping：

优化1：使用 bias 进行最基本的改进，修复阴影失真

由这篇文章的介绍可知，对于阴影失真的问题，可以使用阴影偏移(shadow bias)的技巧来解决。只需要对表面深度应用一个偏移量，使得所有点的深度值都比表面深度更小，从而片元就不会认为其在表面之下，使得整个表面能够被正确照亮。

```
    float bias = 0.005;
    float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```

优化2: 修复 peter 游移

在渲染深度贴图的时候，可以开启正面剔除以解决上一个优化中bias过大导致的悬浮问题。

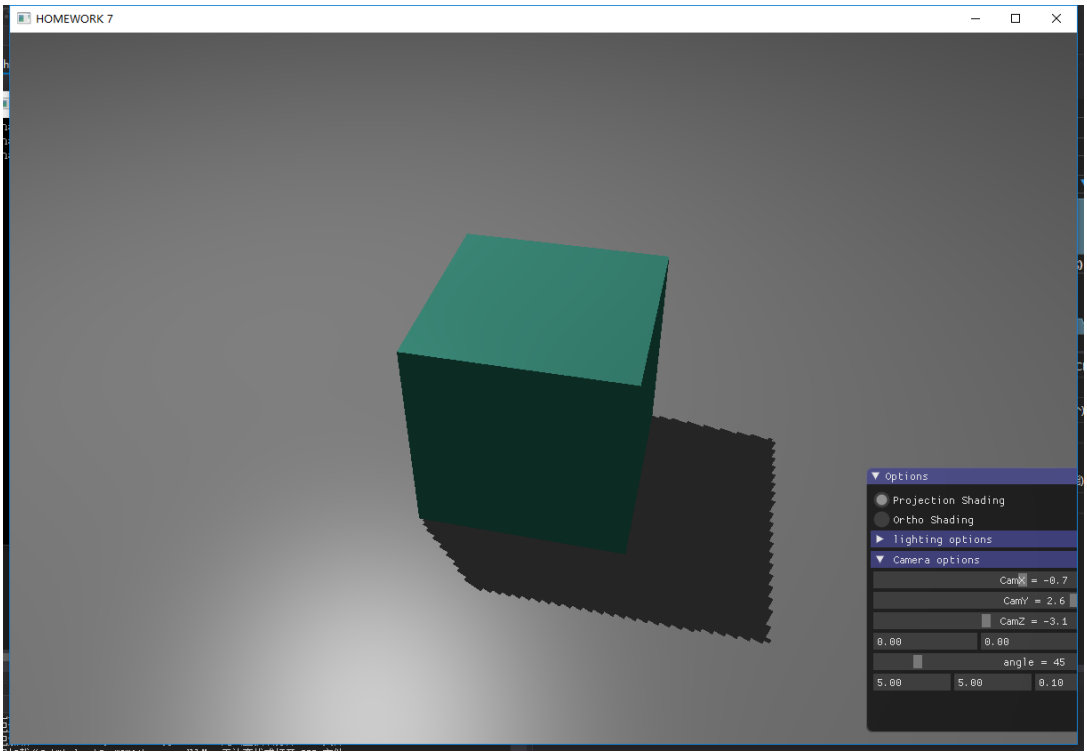
```
    glCullFace(GL_FRONT);
    ...
    glCullFace(GL_BACK);
```

改进3:使用 PCF

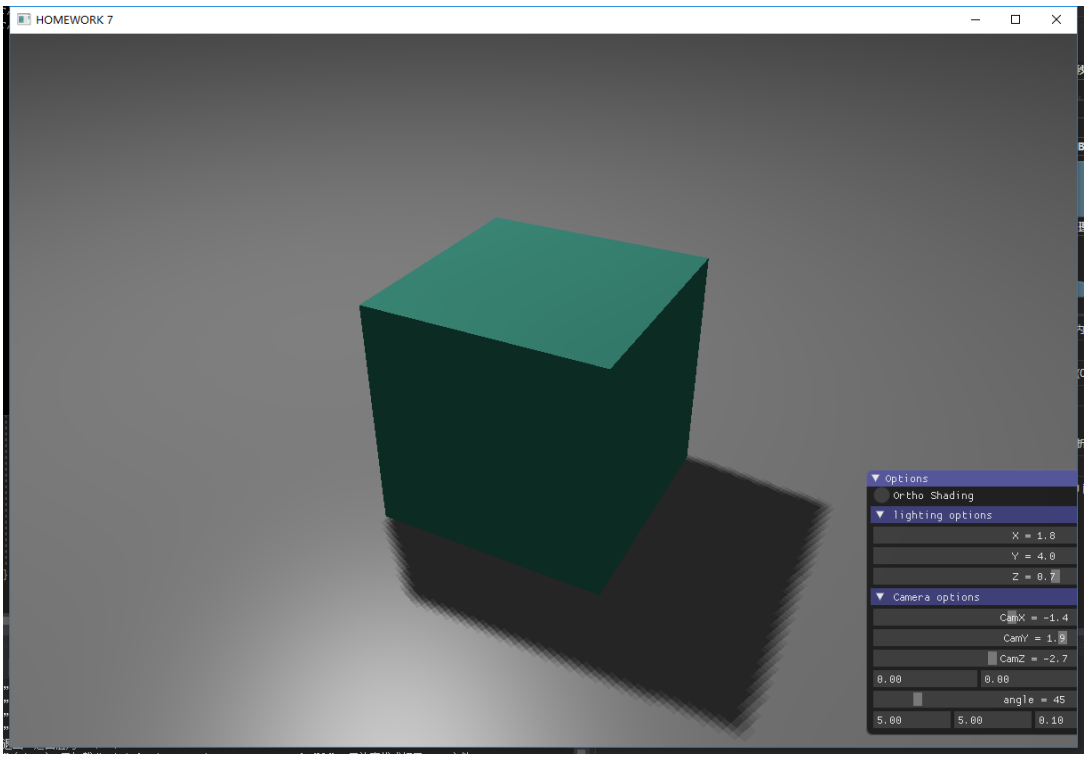
PCF方法能够有效的应用在阴影边缘抗锯齿。主要工作是在片段着色器中，对于每一点的阴影采用对其周围的临近点进行采样取均值，进而达到平滑的效果。这里我使用周围3*3的邻域，计算每一点在深度图中的阴影值，加起来之后做均值。

```
    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;
```

没使用PCF之前：



使用PCF之后：



完整运行效果可以参见gif图。