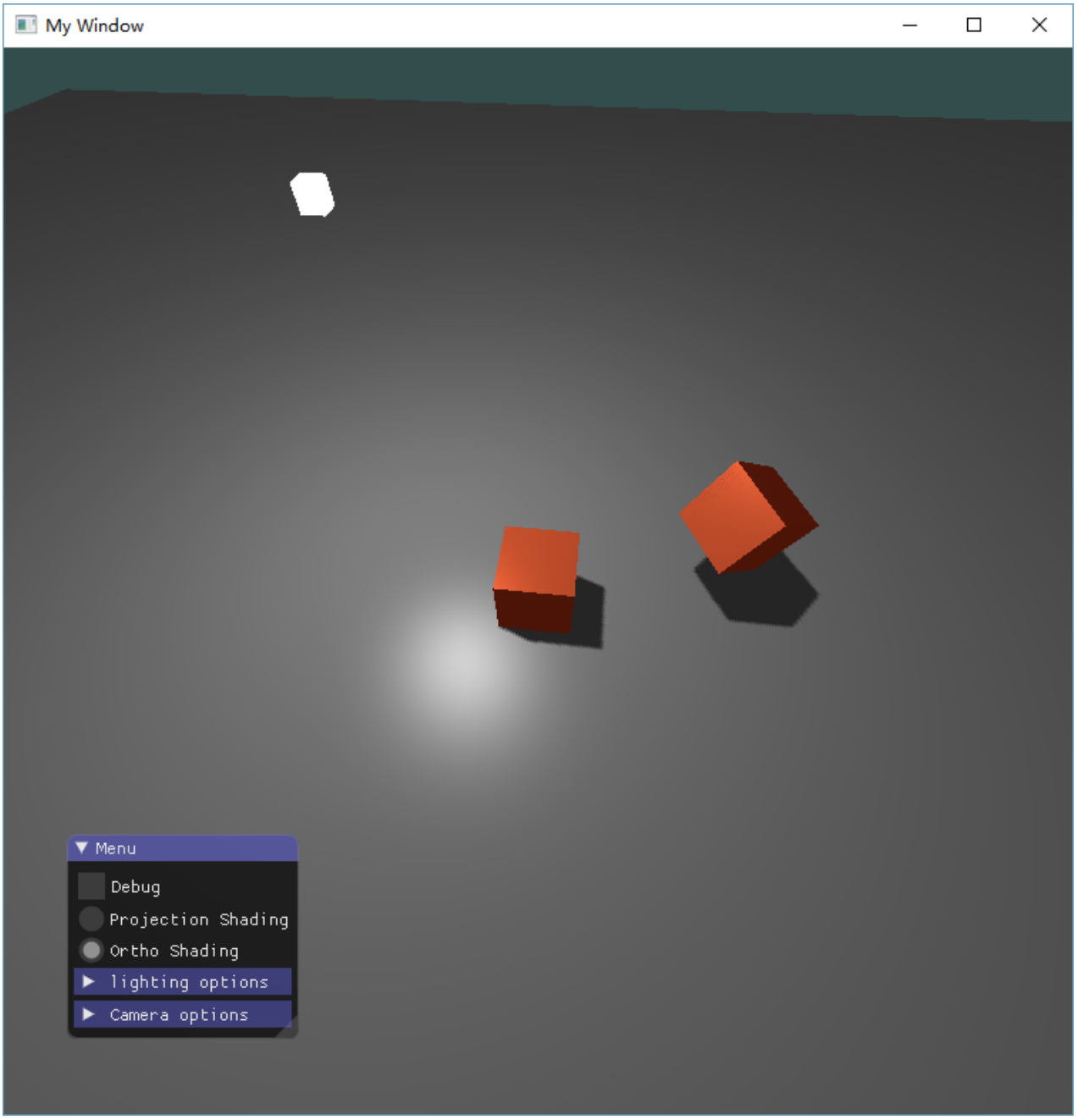
Homework 7 - Shadowing Mapping

> PS：
>
> 1. 本次作业的最终代码放上的已经是完成了 bonus 后的代码。由于工作量过大和为了保证代码简洁性这两点原因，不在 GUI 中设置优化 Shadow Mapping 前后效果对比按钮，仅以截图的形式体现。gif 中展示的是最终成品的效果。
> 2. 本代码不涉及 Camera 类实现 FPS 操控。改为普通的参数设置，可以在 GUI 中进行方便操作，原因是为了在观察不同渲染方法效果的时候保证视角参数的唯一性。

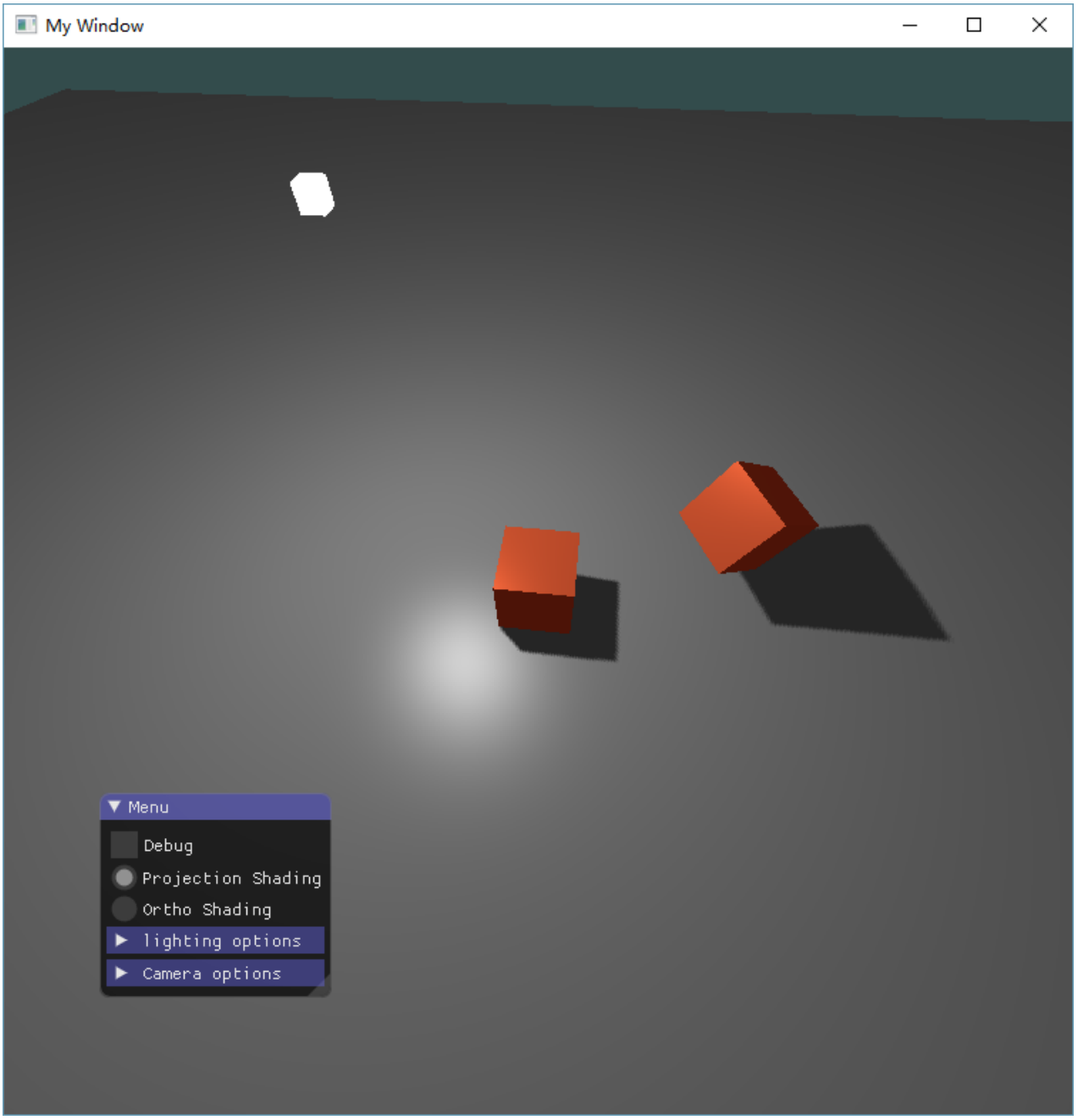
Basic

实现 Shadowing Mapping

最终完成实验截图(包括 Bonus 实现)


My Window
Menu
Debug
Projection Shading
Ortho Shading
lighting options
Camera options

正交投影下的阴影渲染



透视投影下的阴影渲染

Basic 算法实现

总的来说渲染阴影需要两步

1. 以光源方向渲染深度贴图
2. 以摄像机方向，参考深度贴图进行阴影渲染

具体算法过程如下：

1. 创建一个2D纹理贴图，设置好相关的属性和参数。需要注意的是设置纹理 `GL_REPEAT` 参数后，需要补充一个 `borderColor` 防止纹理图片重复渲染，以至于远处重复出现一些无中生有的阴影。

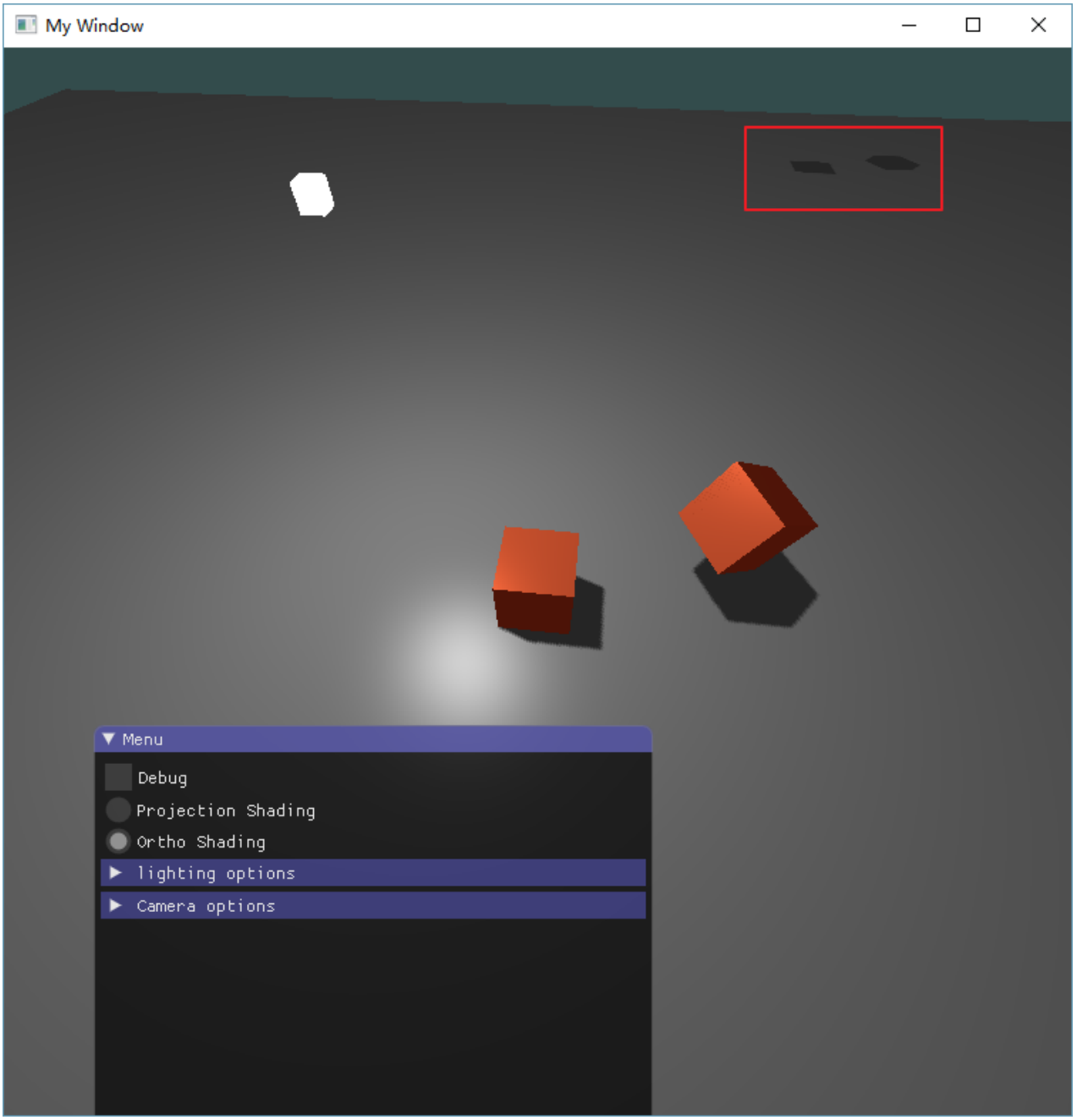
```
// Configure depth map FBO
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// - Create depth texture
GLuint depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);

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);
```

如果没有设置 `borderColor` 会出现如下情况：



2. 通过一个变换 `lightSpaceMatrix` ，将渲染视角空间转移到以光源为视角的空间。其中可以选择透视投影矩阵或者正交投影矩阵来分别模拟点光源或者平行光源。

```
glCullFace(GL_FRONT);
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
GLfloat near_plane = 1.0f, far_plane = 7.5f;
if (mode == 1) {
    // Orthographic
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
else {
    // Projection
    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` ，进行深度纹理贴图的渲染，在这一阶段着色器不需要做太多的操作，因为只需要存储好深度信息。

```
simpleDepthShader.use();
simpleDepthShader.setMat4("lightSpaceMatrix", glm::value_ptr(lightSpaceMatrix));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
RenderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glCullFace(GL_BACK); // 不要忘记设回原先的culling face
```

4. 修改 `glViewPort` 修改回屏幕的大小，然后使用之前渲染好的深度贴图辅助对原图进行渲染即可。

其中，阴影渲染的判断和实际操作细节在着色器中完成，具体过程如下：

1. 在顶点着色器中，除了计算顶点正常着色需要的要素之外，还需要多计算一个变量：顶点位置在光源观察空间中的位置坐标 `FragPosLightSpace`

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

2. 在片段着色器中，通过从顶点着色器拿来的 `FragPosLightSapce` , 以及通过 `Uniform` 传进来的深度贴图 `ShadowMap` ，算出 `closestDepth` 和 `currentDepth` , 综合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;
```

3. 根据 naive 的方法或者 bias 或者 PCF 方法算出一个表示阴影强度的分量 `shadow`

4. 在原始的计算光照的公式中加上 `shadow` 的影响。

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

至此，阴影可以被正确渲染。

Bonus

实现光源在正交/透视两种情况下的 Shadowing Mapping

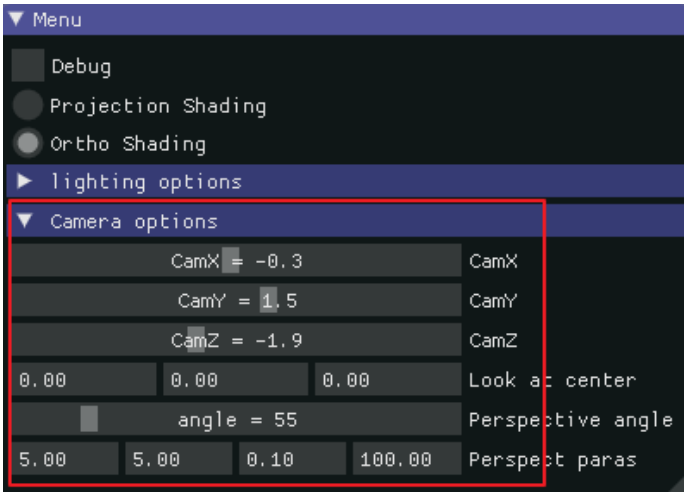
整个过程只需要修改光源观察空间的 `lightProjection` 变换矩阵即可实现

```
glCullFace(GL_FRONT);
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
GLfloat near_plane = 1.0f, far_plane = 7.5f;
if (mode == 1) {
    // Orthographic
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
else {
    // Projection
    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;
```

实验结果截图参见本报告最初的两张截图。

优化 Shadowing Mapping

细节观察的相机参数



改进0：使用 bias 进行最基本的改进，修复阴影失真

> 我们可以用一个叫做阴影偏移（shadow bias）的技巧来解决这个问题，我们简单的对表面的深度（或深度贴图）应用一个偏移量，这样片元就不会被错误地认为在表面之下了。
> 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。

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

引用自此篇文章，由于非常基础也非常关键，缺少了这一步的阴影基本没法看，故不再赘述。

改进1： 修复 peter 游移

对于立方体，开启正面剔除，可以解决由于 bias 设置过大而产生的悬浮问题。

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

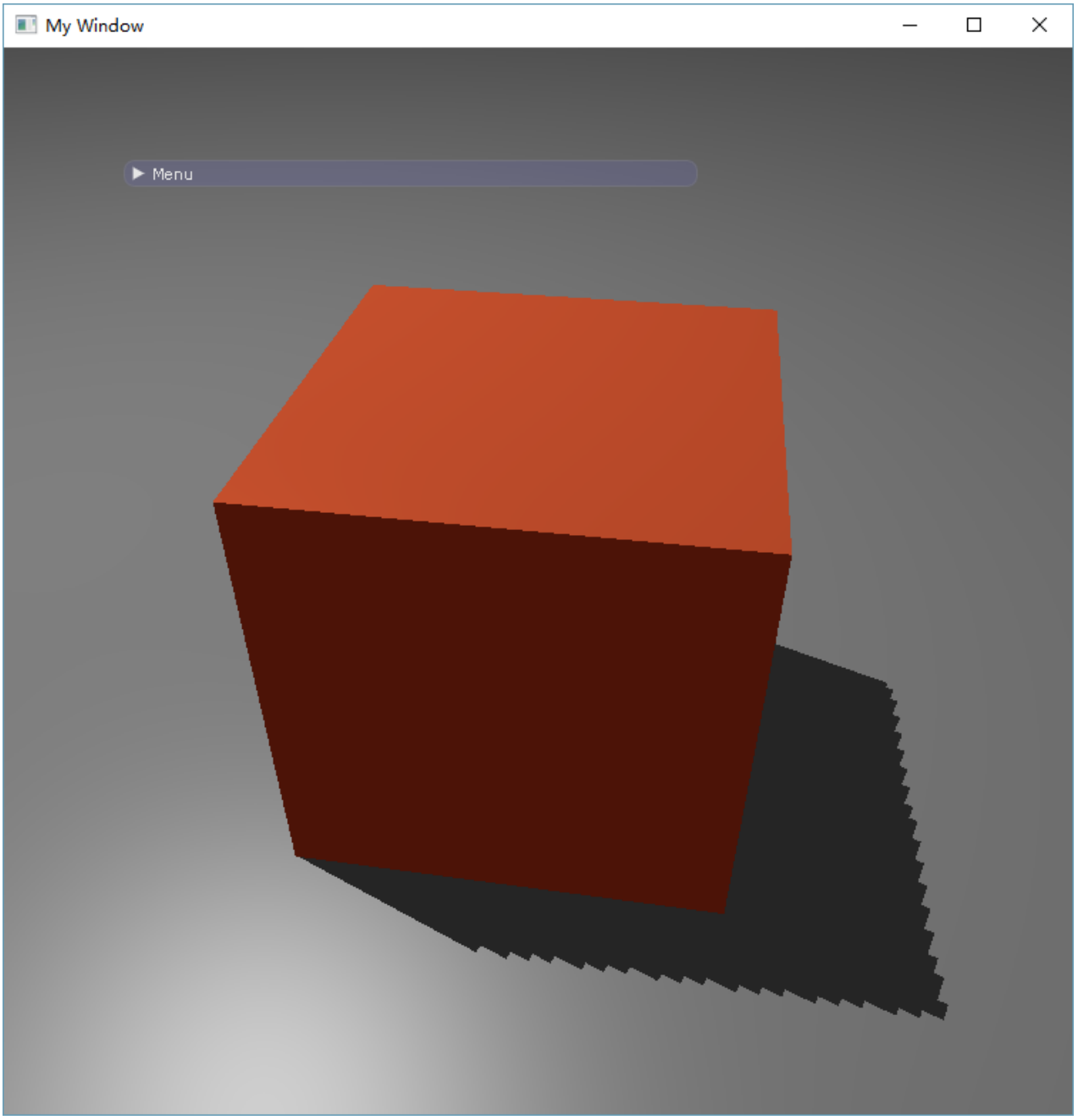
由于在本实验代码中采用的 bias 并不大，所以实验效果前后差距并不明显，故不截图。

改进2：使用 PCF

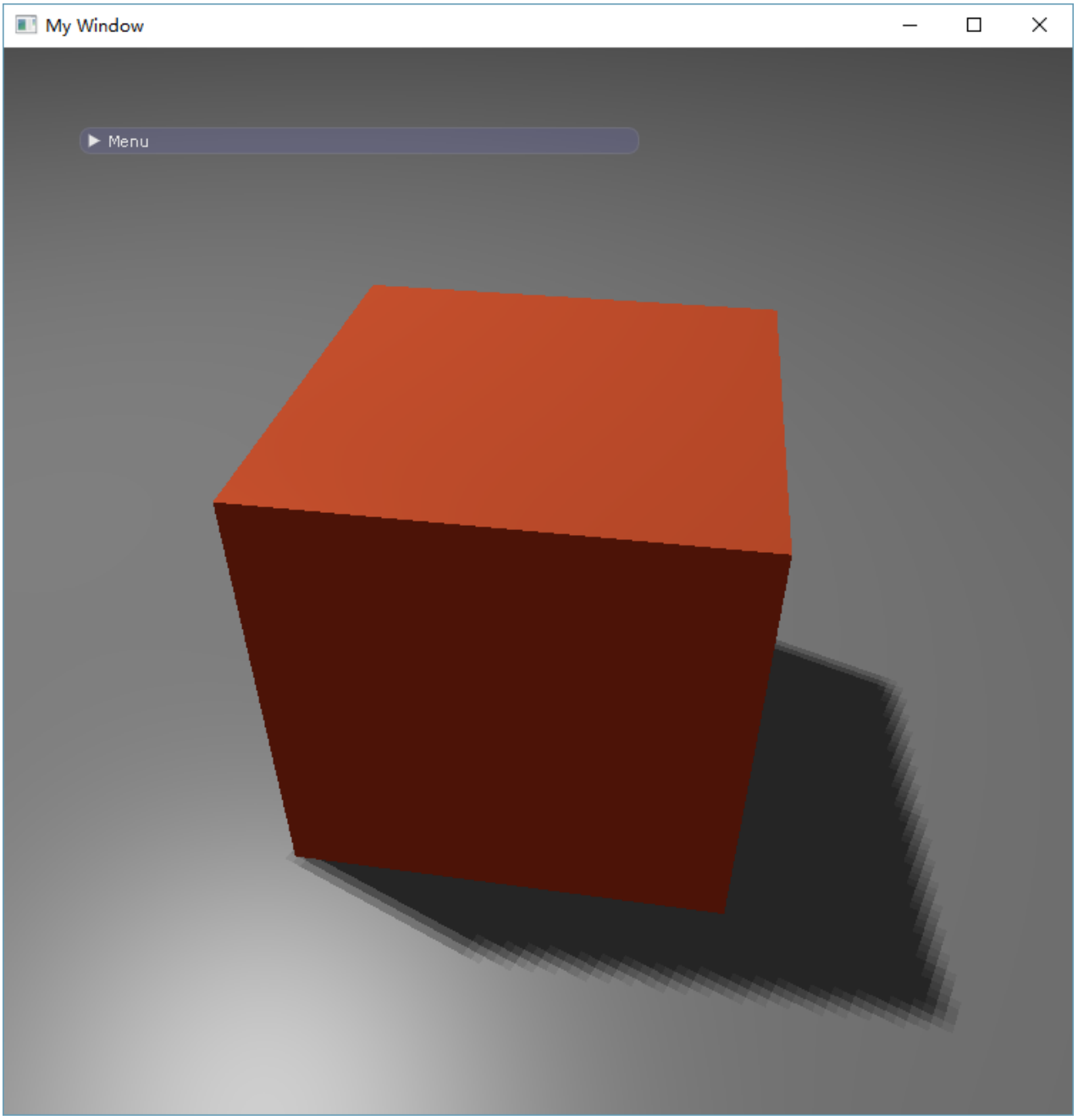
PCF的方法是**在片段着色器中**，对于每一点的 shadow， 并不是直接只取“在”或者“不在”二值，而是对其周围的临近点同时进行采样作平均，以达到平滑的效果。**这样做的好处是可以缓解一下阴影边缘明显的锯齿化**

```
float ShadowCalculation(vec4 fragPosLightSpace) {
    ...
    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;
    ...
    return shadow;
}
```



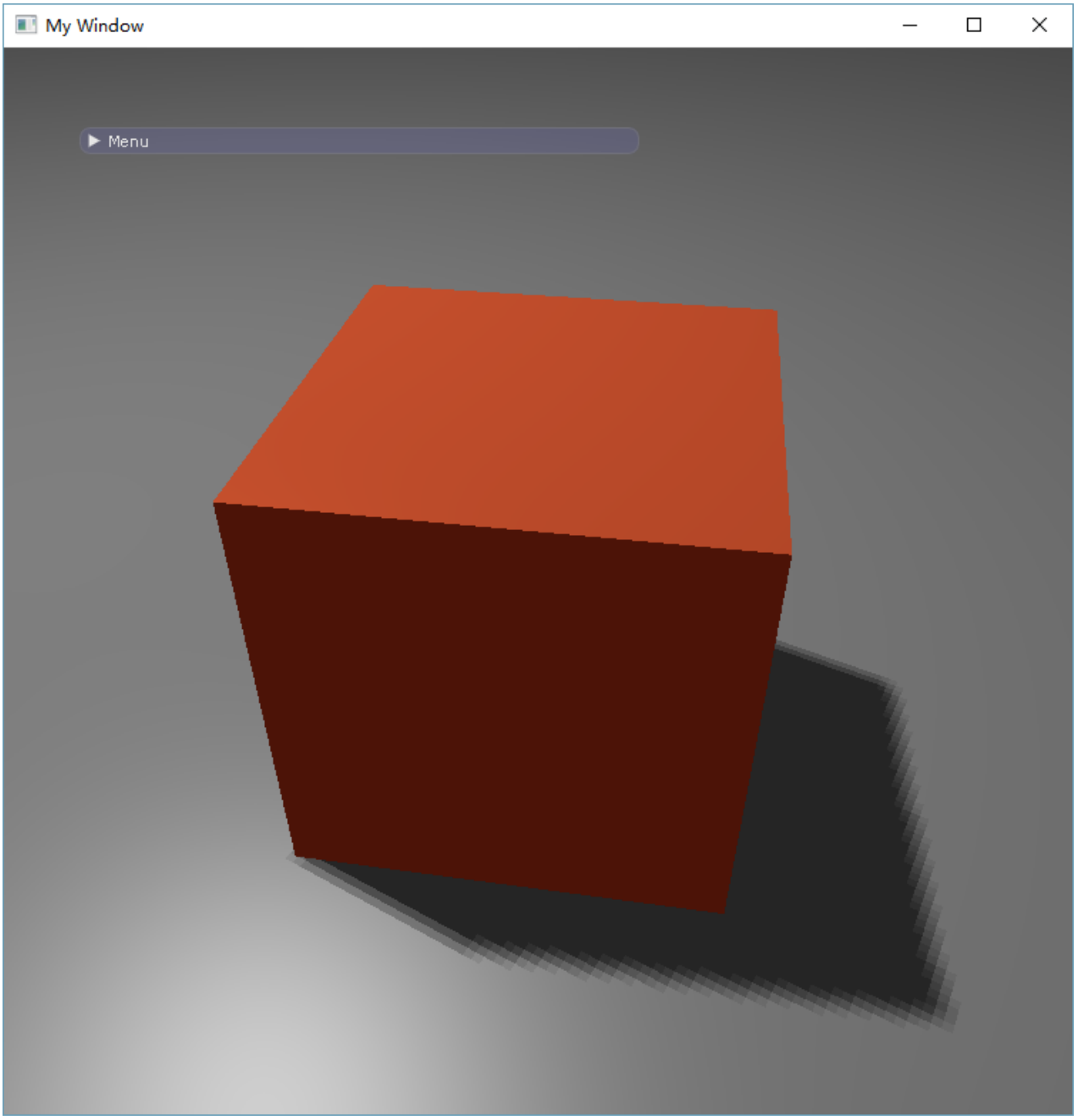
deepthMap 分辨率1024x1024, 没有使用PCF



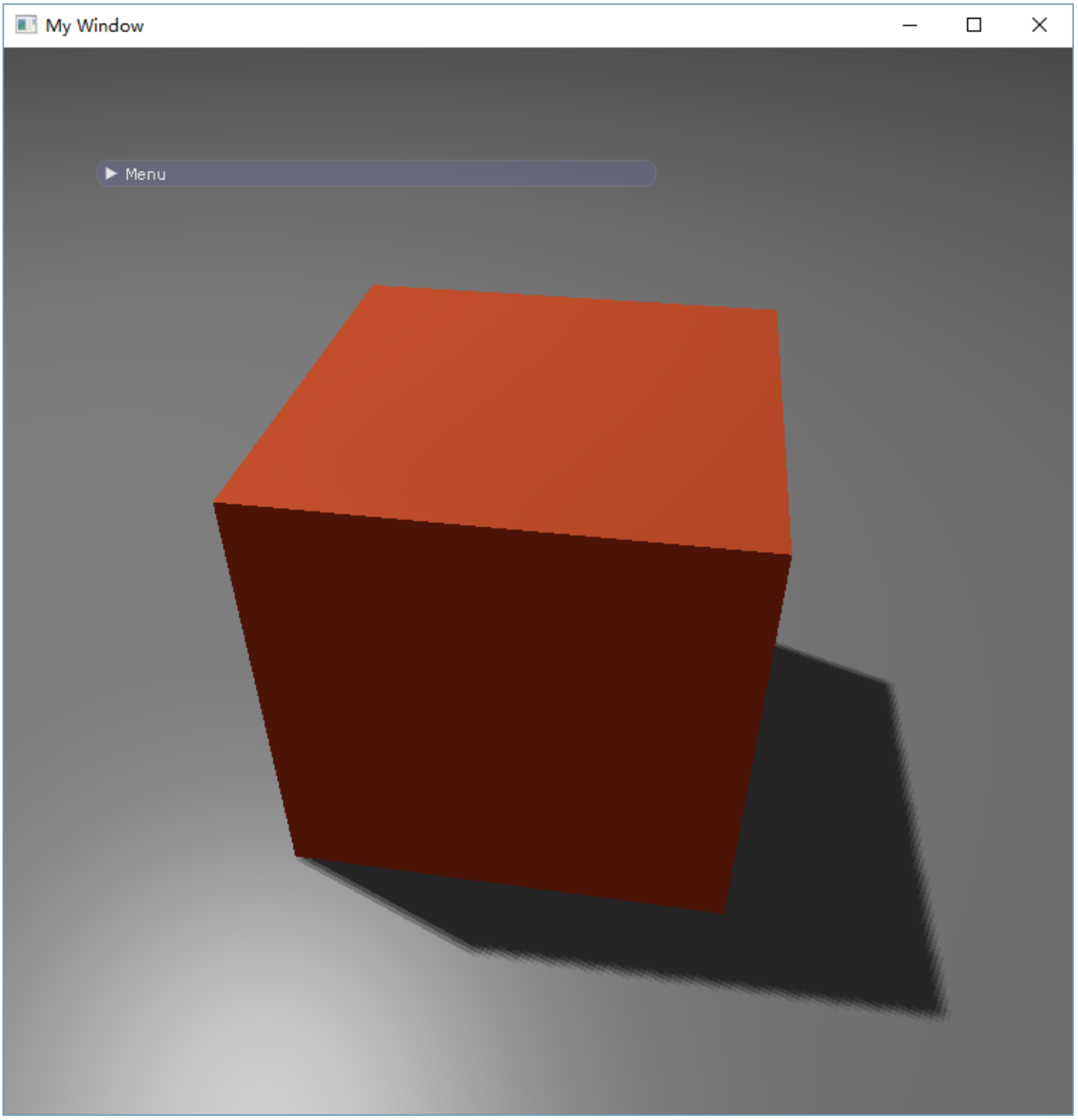
deepthMap 分辨率1024x1024, 使用PCF

改进3：增加 deepthMap 的分辨率

提高深度贴图的分辨率，会让纹理的像素点排列得更加紧密，由此可以提高阴影的渲染效果。 通过增加 `SHADOW_WIDTH` 和 `SHADOW_HEIGHT` 即可做到。



deepthMap 分辨率为 1024x1024



deepthMap 分辨率为 2048x2048