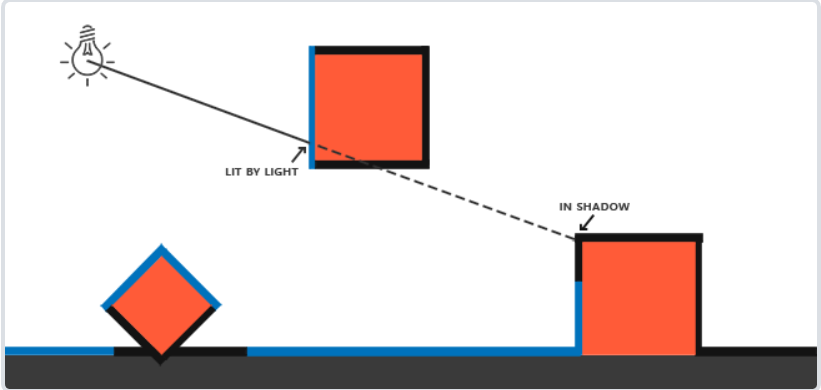
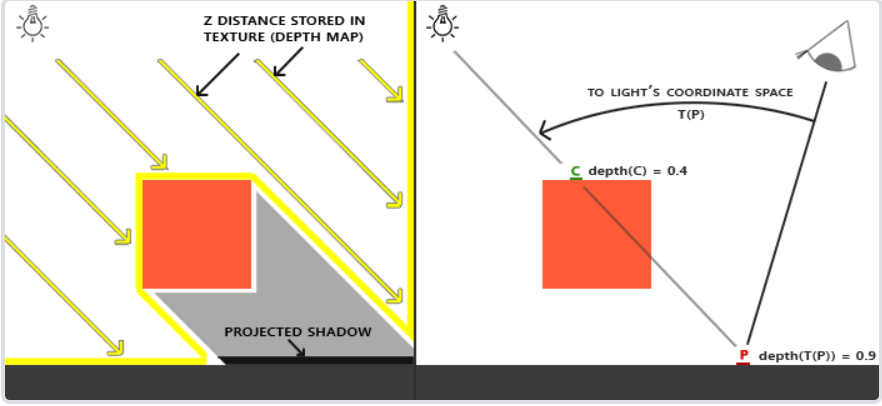


1. 实现**Shadow mapping**

首先，我们要在**光源**的角度进行渲染，得到光源位置的**深度贴图**的信息，其中深度贴图中的每一个像素点代表从光源位置看去，能看到的**最近的物体**的深度。也就是说在进行正式的渲染之前，要**先**进行一遍**渲染**，生成从光源位置的深度贴图。这里渲染得到的数据保存在我们自己创建的**深度缓冲**中。



接下来进行**正式的渲染**，即对屏幕坐标中的每一点的**颜色缓冲**进行填充。这次渲染的观察点是下图所示的**人眼**所在的位置。比如当前要渲染点**P**，在进行**shadow mapping**时，我们要决定**P**点**是否在阴影中**，首先我们要将**P**点转换到**光空间**中，即从光源看向**P**点的坐标，然后将深度缓冲中该坐标的**深度值**与**P**点的**z坐标**进行比较，如果**P**点处于光源能看见的**最近点**的**后面**，则**P**点在阴影中，否则不在阴影中。



**实现：**

首先我们要生成保存**深度信息**的**帧缓冲depthMapFBO**，调用函数**glGenFramebuffers**即可生成**帧缓冲**。

接下来创建**2 D纹理**，提供给帧缓冲使用。因为我们只关心深度值，我们要把纹理格式指定为**GL\_DEPTH\_COMPONENT**。我们还要把纹理的高宽设置为1024：这是深度贴图的解析度。

帧缓冲和纹理都生成成功之后就可以调用**glFramebufferTexture2D**函数将生成的纹理与深度贴图的缓冲绑定起来，这里的参数**GL\_DEPTH\_ATTACHMENT**即表示该纹理用于存储图像的**深度信息**。

由于这里还不需要读写颜色缓冲区的数据，所以调用函数

**glDrawBuffer(GL\_NONE)**和**glReadBuffer(GL\_NONE)**即可。

const GLuint SHADOW\_WIDTH **=** 1024**,** SHADOW\_HEIGHT **=** 1024**;**

GLuint depthMapFBO**;**

glGenFramebuffers**(**1**,** **&**depthMapFBO**);**

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\_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**);**

接下来便可以渲染**深度贴图**了。首先我们看一下生成深度贴图所用的着色器。

顶点着色器**ShadowMappingShader.v**

**position**依旧代表顶点的位置

**lightSpaceMatrix**代表将坐标转换成光空间的**view**与**projection**矩阵想乘。

**model**矩阵即模型矩阵

这里顶点着色器的作用就是将顶点转换到**光空间**。

**#**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**);**

**}**

片段着色器**ShadowMappingShader.f**

这里的片段着色器什么都不做，运行完后，**深度缓冲**会被更新，底层会**默认**去设置深度缓冲。

#version 330 core

void main**()**

**{**

**}**

主函数中进行深度缓冲渲染：

Shader depthShader**(**"ShadowMappingDepth.v"**,** "ShadowMappingDepth.f"**);**

**bindVAO();**

float nearPlane **=** 1.0f**;**

float farPlane **=** 7.5f**;**

int display\_w **=** 1024**;**

int display\_h **=** 1024**;**

float left **=** **-**10**;**

float right **=** 10**;**

float bottom **=** **-**10**;**

float top **=** 10**;**

**while** **(!**glfwWindowShouldClose**(**window**))**

**{**

…

glm**::**mat4 lightProjection**,** lightView**;**

glm**::**mat4 lightSpaceMatrix**;**

lightProjection **=** glm**::**ortho**(**left**,** right**,** bottom**,** top**,** nearPlane**,** farPlane**);**

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

lightSpaceMatrix **=** lightProjection **\*** lightView**;**

depthShader**.**useProgram**();**

depthShader**.**setMatrix**(**"lightSpaceMatrix"**,** lightSpaceMatrix**);**

glViewport**(**0**,** 0**,** SHADOW\_WIDTH**,** SHADOW\_HEIGHT**);**

glBindFramebuffer**(**GL\_FRAMEBUFFER**,** depthMapFBO**);**

glClear**(**GL\_DEPTH\_BUFFER\_BIT**);**

renderScene**(**depthShader**);**

glBindFramebuffer**(**GL\_FRAMEBUFFER**,** 0**);**

…

**}**

其中**bindVAO**用来对所用**VAO**进行绑定。这里主要有三个**VAO**

分别是**平面**对应的VAO、**立方体**的VAO和用来**调试深度缓冲的矩形**的VAO。

glm**::**vec3 lightPos**(-**2.0f**,** 4.0f**,** **-**1.0f**);**

GLuint cubeVAO**;**

GLuint planeVAO**;**

GLuint quadVAO **=** 0**;**

void bindVAO**()** **{**

float vertices**[]** **=** **{**

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

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

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

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

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

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

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

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

0.5f**,** 0.5f**,** 0.5f**,** 0.0f**,** 0.0f**,** 1.0f**,**

0.5f**,** 0.5f**,** 0.5f**,** 0.0f**,** 0.0f**,** 1.0f**,**

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

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

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

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

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

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

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

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

0.5f**,** 0.5f**,** 0.5f**,** 1.0f**,** 0.0f**,** 0.0f**,**

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

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

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

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

0.5f**,** 0.5f**,** 0.5f**,** 1.0f**,** 0.0f**,** 0.0f**,**

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

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

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

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

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

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

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

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

0.5f**,** 0.5f**,** 0.5f**,** 0.0f**,** 1.0f**,** 0.0f**,**

0.5f**,** 0.5f**,** 0.5f**,** 0.0f**,** 1.0f**,** 0.0f**,**

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

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

**};**

unsigned int VBO**;**

glGenVertexArrays**(**1**,** **&**cubeVAO**);**

glGenBuffers**(**1**,** **&**VBO**);**

glBindVertexArray**(**cubeVAO**);**

glBindBuffer**(**GL\_ARRAY\_BUFFER**,** VBO**);**

glBufferData**(**GL\_ARRAY\_BUFFER**,** **sizeof(**vertices**),** vertices**,** GL\_STATIC\_DRAW**);**

glEnableVertexAttribArray**(**0**);**

glVertexAttribPointer**(**0**,** 3**,** GL\_FLOAT**,** GL\_FALSE**,** 6 **\*** **sizeof(**float**),** **(**void**\*)**0**);**

glEnableVertexAttribArray**(**1**);**

glVertexAttribPointer**(**1**,** 3**,** GL\_FLOAT**,** GL\_FALSE**,** 6 **\*** **sizeof(**float**),** **(**void**\*)(**3 **\*** **sizeof(**float**)));**

//glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray**(**0**);**

GLfloat planeVertices**[]** **=** **{**

// Positions // Normals

25.0f**,** **-**0.5f**,** 25.0f**,** 0.0f**,** 1.0f**,** 0.0f**,**

**-**25.0f**,** **-**0.5f**,** **-**25.0f**,** 0.0f**,** 1.0f**,** 0.0f**,**

**-**25.0f**,** **-**0.5f**,** 25.0f**,** 0.0f**,** 1.0f**,** 0.0f**,**

25.0f**,** **-**0.5f**,** 25.0f**,** 0.0f**,** 1.0f**,** 0.0f**,**

25.0f**,** **-**0.5f**,** **-**25.0f**,** 0.0f**,** 1.0f**,** 0.0f**,**

**-**25.0f**,** **-**0.5f**,** **-**25.0f**,** 0.0f**,** 1.0f**,** 0.0f

**};**

GLuint planeVBO**;**

glGenVertexArrays**(**1**,** **&**planeVAO**);**

glGenBuffers**(**1**,** **&**planeVBO**);**

glBindVertexArray**(**planeVAO**);**

glBindBuffer**(**GL\_ARRAY\_BUFFER**,** planeVBO**);**

glBufferData**(**GL\_ARRAY\_BUFFER**,** **sizeof(**planeVertices**),** **&**planeVertices**,** GL\_STATIC\_DRAW**);**

glEnableVertexAttribArray**(**0**);**

glVertexAttribPointer**(**0**,** 3**,** GL\_FLOAT**,** GL\_FALSE**,** 6 **\*** **sizeof(**GLfloat**),** **(**GLvoid**\*)**0**);**

glEnableVertexAttribArray**(**1**);**

glVertexAttribPointer**(**1**,** 3**,** GL\_FLOAT**,** GL\_FALSE**,** 6 **\*** **sizeof(**GLfloat**),** **(**GLvoid**\*)(**3 **\*** **sizeof(**GLfloat**)));**

glBindVertexArray**(**0**);**

**}**

**renderScene**函数用来渲染场景中的物体，这里包括**一个平面**和平面**上方**的**一个立方体**。并且在调用此函数**渲染物体之前**，要调用

**glViewport(**0**,** 0**,** SHADOW\_WIDTH**,** SHADOW\_HEIGHT**)**设置深度缓冲的**分辨率**

**glBindFramebuffer(**GL\_FRAMEBUFFER**,** depthMapFBO**)绑定**深度缓冲

**glClear(**GL\_DEPTH\_BUFFER\_BIT**)清空**深度缓冲。

void renderScene**(**Shader **&**shader**)**

**{**

// Floor

glm**::**mat4 model**(**1.0**);**

shader**.**setModel**(**model**);**

glBindVertexArray**(**planeVAO**);**

glDrawArrays**(**GL\_TRIANGLES**,** 0**,** 6**);**

glBindVertexArray**(**0**);**

// Cubes

model **=** glm**::**mat4**(**1.0**);**

model **=** glm**::**translate**(**model**,** glm**::**vec3**(**0.0f**,** 1.5f**,** 0.0**));**

shader**.**setModel**(**model**);**

// Render Cube

glBindVertexArray**(**cubeVAO**);**

glDrawArrays**(**GL\_TRIANGLES**,** 0**,** 36**);**

glBindVertexArray**(**0**);**

**}**

接下来就可以对场景中的物体进行**Shadow mapping**了，依然先看**着色器**

顶点着色器**ShaowMapping.v**

这里与之前**Phong光照模型**不太相同的地方是多了个**FragPosLightSpace**变量，我们用同一个**lightSpaceMatrix**，把**世界空间**顶点位置转换为**光空间**。顶点着色器传递一个普通的经变换的世界空间顶点位置**vs\_out.FragPos**和一个光空间的**vs\_out.FragPosLightSpace**给像素着色器。

**#**version 330 core

layout **(**location **=** 0**)** in vec3 position**;**

layout **(**location **=** 1**)** in vec3 normal**;**

out VS\_OUT **{**

vec3 FragPos**;**

vec3 Normal**;**

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**.**FragPosLightSpace **=** lightSpaceMatrix **\*** vec4**(**vs\_out**.**FragPos**,** 1.0**);**

**}**

**ShadowMapping.f**

片段着色器使用**Phong**光照模型渲染场景。我们接着计算出一个**shadow**值，当fragment在阴影中时是1.0，在阴影外是0.0。然后，**diffuse**和**specular**颜色会乘以这个阴影元素。由于阴影不会是全黑的（由于散射），我们把ambient分量从乘法中剔除。我们声明一个**shadowCalculation**函数，用它计算阴影。像素着色器的最后，我们我们把**diffuse**和**specular**乘以**(1-阴影元素)**，这表示这个片元有**多大成分不在阴影中**。这个像素着色器还需要两个额外输入，一个是**光空间的片元位置**和第一个渲染阶段得到的**深度贴图**。

这里已经包含了对于阴影的一些**优化。**

#version 330 core

**out** vec4 FragColor**;**

**in** VS\_OUT **{**

vec3 FragPos**;**

vec3 Normal**;**

vec4 FragPosLightSpace**;**

**}** fs\_in**;**

uniform sampler2D shadowMap**;**

uniform vec3 lightPos**;**

uniform vec3 viewPos**;**

uniform bool shadows **=** true**;**

**float** ShadowCalculation**(**vec4 fragPosLightSpace**)**

**{**

vec3 projCoords **=** fragPosLightSpace.xyz / fragPosLightSpace.w**;**

projCoords **=** projCoords **\*** 0.5 **+** 0.5**;**

**float** closestDepth **=** texture**(**shadowMap**,** projCoords.xy).r**;**

**float** currentDepth **=** projCoords.z;

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**);**

**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**;**

**if(**projCoords.z > 1.0**)**

shadow **=** 0.0**;**

**return** shadow**;**

**}**

void main**()**

**{**

vec3 color **=** vec3**(**0.0f**,** 0.8f**,** 0.0f**);**

vec3 normal **=** normalize**(**fs\_in.Normal);

vec3 lightColor **=** vec3**(**0.4**);**

**//** Ambient

vec3 ambient **=** 0.2 **\*** 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);

**float** spec **=** 0.0**;**

vec3 halfwayDir **=** normalize**(**lightDir **+** viewDir**);**

spec **=** pow**(max(**dot**(**normal**,** halfwayDir**),** 0.0**),** 64.0**);**

vec3 specular **=** spec **\*** lightColor**;**

**float** shadow **=** shadows **?** ShadowCalculation**(**fs\_in.FragPosLightSpace) : 0.0**;**

shadow **=** **min(**shadow**,** 0.75**);**

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

FragColor **=** vec4**(**lighting**,** 1.0f**);**

**}**

首先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。由于裁切空间的**FragPosLightSpace**并不会通过**gl\_Position**传到像素着色器里，我们必须**自己做**透视除法：

因为来自深度贴图的深度在0到1的范围，我们也打算使用**projCoords**从深度贴图中去采样，所以我们将**NDC坐标**变换为0到1的范围：

有了这些投影坐标，我们就能从深度贴图中采样得到0到1的结果，从第一个渲染阶段的projCoords坐标直接对应于变换过的NDC坐标。我们将得到光的位置视野下最近的深度：

为了得到片元的当前深度，我们简单获取投影向量的z坐标，它等于来自光的透视视角的片元的深度。

改进：

1. **阴影失真：**

因为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中去采样。图片每个斜坡代表深度贴图一个单独的纹理像素。你可以看到，多个片元从同一个深度值进行采样。

虽然很多时候没问题，但是当光源以一个角度朝向表面的时候就会出问题，这种情况下深度贴图也是从一个角度下进行渲染的。多个片元就会从同一个斜坡的深度纹理像素中采样，有些在地板上面，有些在地板下面；这样我们所得到的阴影就有了差异。因为这个，有些片元被认为是在阴影之中，有些不在，由此产生了图片中的条纹样式。

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。我们可以这样实现这个偏移：

1. **悬浮**

这个阴影失真叫做悬浮(Peter Panning)，因为物体看起来轻轻悬浮在表面之上（译注Peter Pan就是童话彼得潘，而panning有平移、悬浮之意，而且彼得潘是个会飞的男孩…）。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）你也许记得在面剔除教程中OpenGL默认是背面剔除。我们要告诉OpenGL我们要剔除正面。

因为我们只需要深度贴图的深度值，对于实体物体无论我们用它们的正面还是背面都没问题。使用背面深度不会有错误，因为阴影在物体内部有错误我们也看不见。

为了修复peter游移，我们要进行正面剔除，先必须开启**GL\_CULL\_FACE**：

**glCullFace(GL\_FRONT);**

**renderScene(depthShader);**

**glCullFace(GL\_BACK);**

1. **PCF**

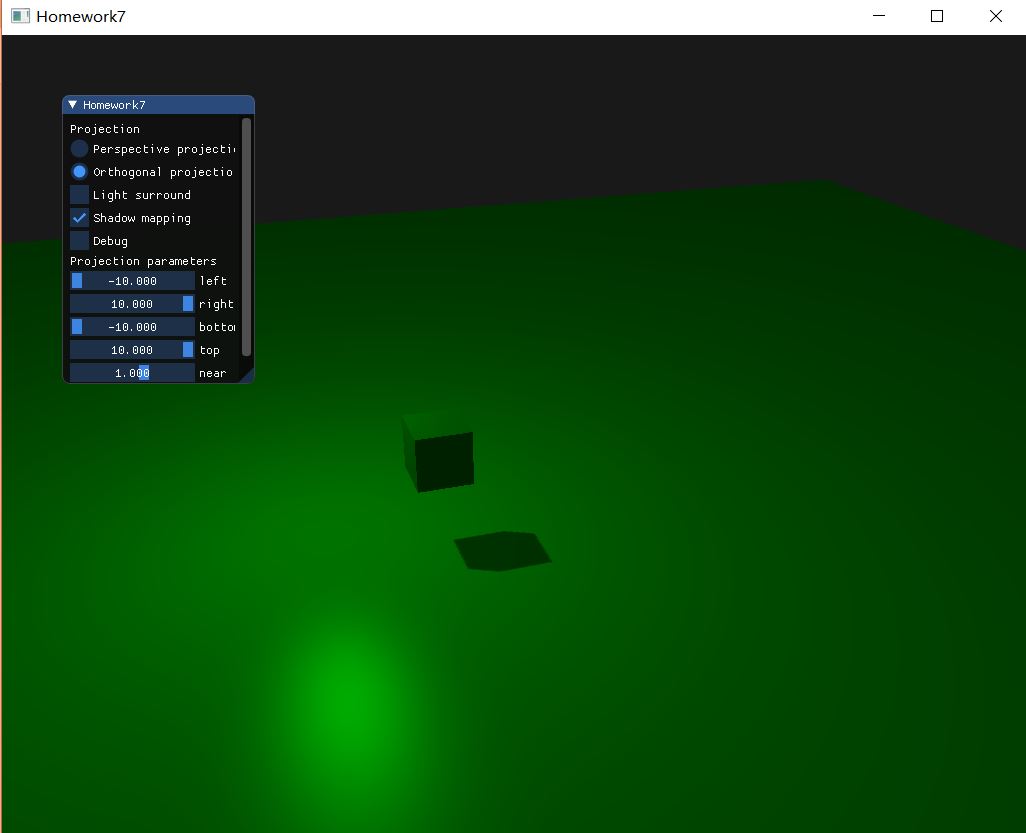
因为深度贴图有一个固定的解析度，多个片元对应于一个纹理像素。结果就是多个片元会从深度贴图的同一个深度值进行采样，这几个片元便得到的是同一个阴影，这就会产生锯齿边。

你可以通过增加深度贴图解析度的方式来降低锯齿块，也可以尝试尽可能的让光的视锥接近场景。

另一个（并不完整的）解决方案叫做PCF（percentage-closer filtering），这是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。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。

一个简单的PCF的实现是简单的从纹理像素四周对深度贴图采样，然后把结果平均起来：

**效果：这里实现的是正交投影：**

****

1. 实现光源的不同**投影**方式

**GUI**中加入选项，并可调节投影的参数，具体做法与之前实现投影的作业无区别。

int projMode **=** 0**;**

bool shadow **=** **true;**

bool debug **=** **false;**

const int PERSPECTIVE **=** 0**;**

const int ORTHOGONAL **=** 1**;**

ImGui**::**Begin**(**TITLE**);**

ImGui**::**Text**(**"Projection"**);**

ImGui**::**RadioButton**(**"Perspective projection"**,** **&**projMode**,** PERSPECTIVE**);**

ImGui**::**RadioButton**(**"Orthogonal projectio"**,** **&**projMode**,** ORTHOGONAL**);**

ImGui**::**Checkbox**(**"Light surround"**,** **&**isLightSurround**);**

ImGui**::**Checkbox**(**"Shadow mapping"**,** **&**shadow**);**

ImGui**::**Checkbox**(**"Debug"**,** **&**debug**);**

**if** **(**projMode **==** PERSPECTIVE**)** **{**

ImGui**::**Text**(**"Projection parameters"**);**

ImGui**::**SliderFloat**(**"radian"**,** **&**radian**,** 1**,** 89**);**

ImGui**::**SliderFloat**(**"near"**,** **&**nearValue**,** **-**5**,** 5**);**

ImGui**::**SliderFloat**(**"far"**,** **&**farValue**,** 5**,** 150**);**

isOrthogonal **=** **false;**

**}**

**else** **if** **(**projMode **==** ORTHOGONAL**)** **{**

ImGui**::**Text**(**"Projection parameters"**);**

ImGui**::**SliderFloat**(**"left"**,** **&**left**,** **-**10**,** 10**);**

ImGui**::**SliderFloat**(**"right"**,** **&**right**,** **-**10**,** 10**);**

ImGui**::**SliderFloat**(**"bottom"**,** **&**bottom**,** **-**10**,** 10**);**

ImGui**::**SliderFloat**(**"top"**,** **&**top**,** **-**10**,** 10**);**

ImGui**::**SliderFloat**(**"near"**,** **&**nearPlane**,** **-**5**,** 5**);**

ImGui**::**SliderFloat**(**"far"**,** **&**farPlane**,** 5**,** 150**);**

isPerspective **=** **false;**

**}**

ImGui**::**End**();**

glm**::**mat4 lightProjection**,** lightView**;**

glm**::**mat4 lightSpaceMatrix**;**

//GLfloat nearPlane = 1.0f, farPlane = 7.5f;

**if** **(**projMode **==** ORTHOGONAL**)** **{**

lightProjection **=** glm**::**ortho**(**left**,** right**,** bottom**,** top**,** nearPlane**,** farPlane**);**

**}**

**else** **{**

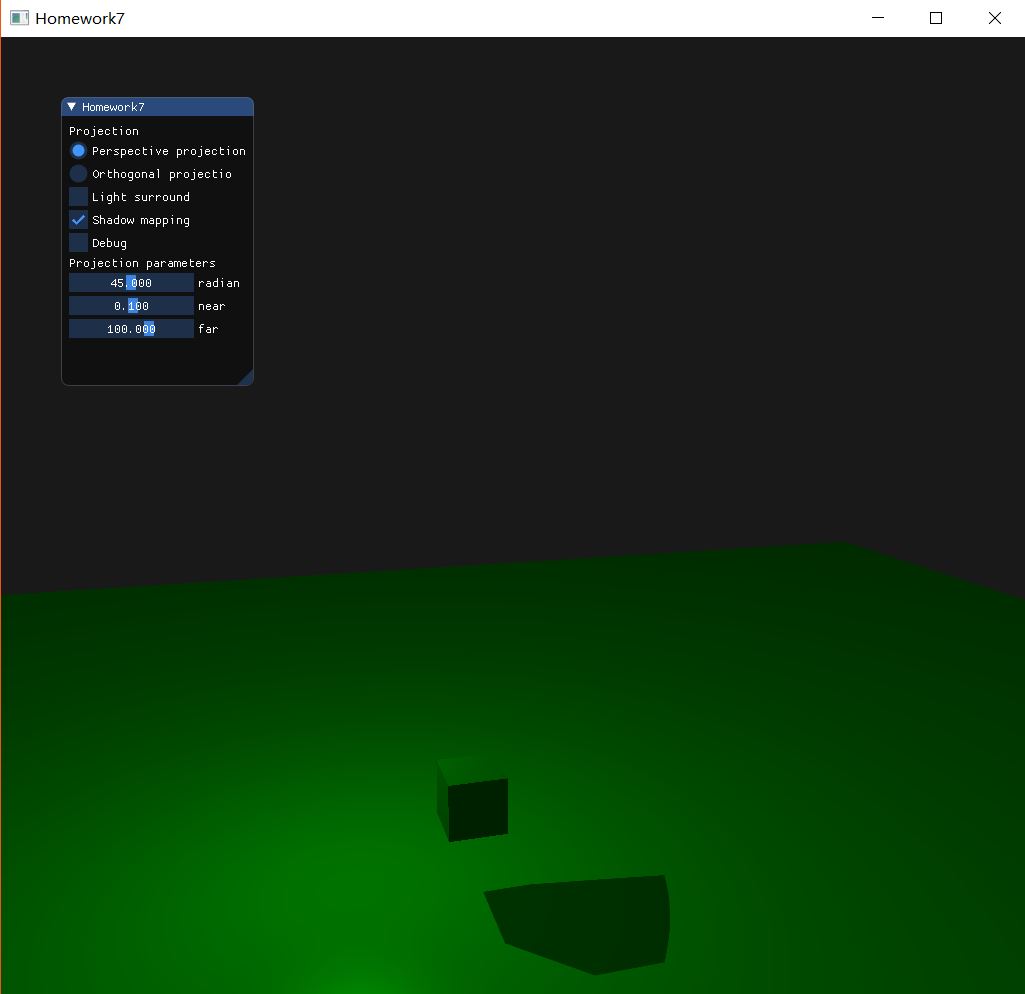
lightProjection **=** glm**::**perspective**(**radian**,** **(**float**)**display\_w **/** **(**float**)**display\_h**,** nearValue**,** farValue**);**

**}**

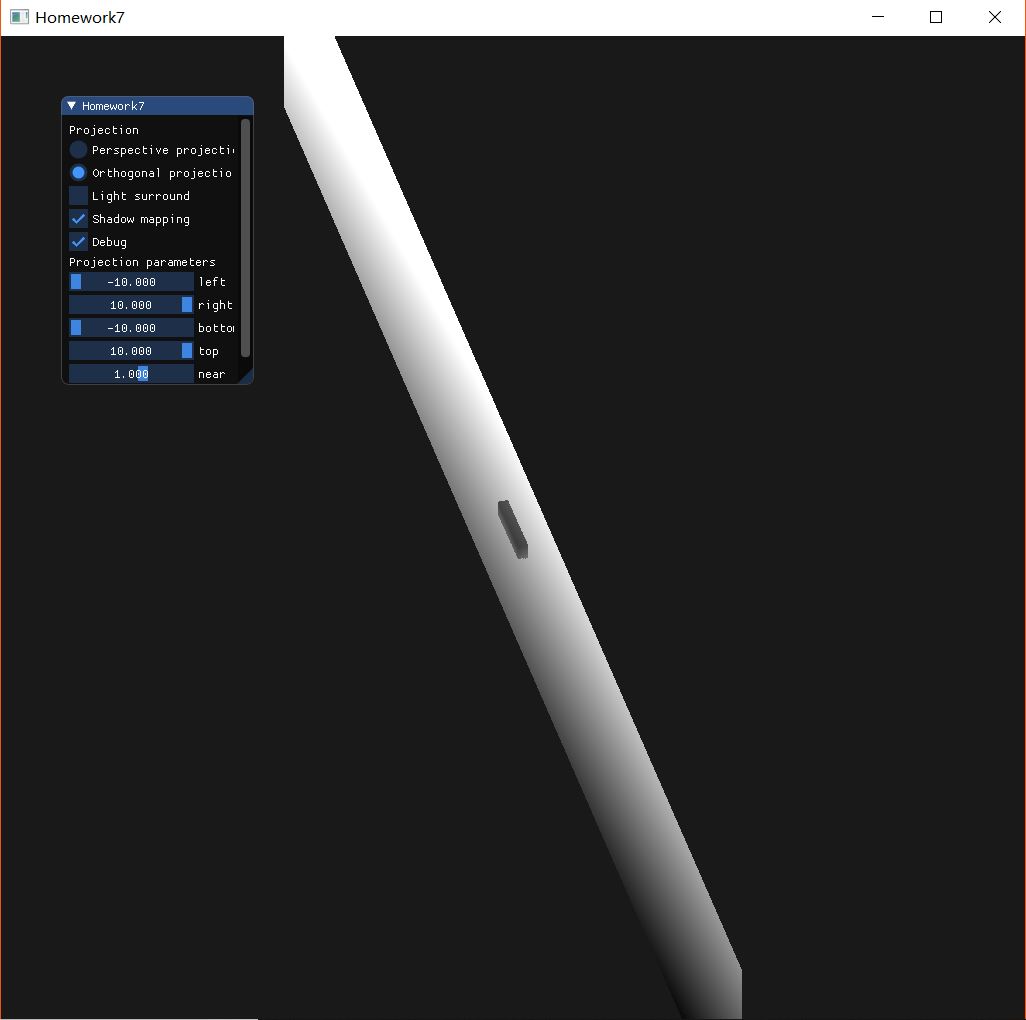
lightView **=** glm**::**lookAt**(**lightPos**,** glm**::**vec3**(**0.0f**),** glm**::**vec3**(**0.0**,** 1.0**,** 0.0**));**

lightSpaceMatrix **=** lightProjection **\*** lightView**;**

**透视投影：**

****

此外，还实现了在**光源位置观察**渲染的深度贴图，将深度信息转化为**颜色的深浅**展示，**GUI**中选择**Debug**即可在光源位置看深度缓冲的信息：



选择**Light surround**选项可以使**光源**在**椭圆**上移动，看到**实时阴影**的**渲染效果**。具体效果在**视频**中展示。