Homework 7 - Shadowing Mapping

黄俊凯 16340082

Shadowing Mapping

Shadowing Mapping 的原理很简单，总的来说，就是在光的视角，把光能看到的物体都点亮。在 OpenGL 里的实现呢，我们可以分两个步骤来实现：

1. 假设在光的位置放置摄像机，求得 View 和 Projection 矩阵，然后将世界空间中的物体乘以这两个矩阵，最后通过 z-buffer 来获取里光源最近（没有被遮挡）的点的z值
2. 现在考虑正常的摄像机位置，像往常一样对物体进行坐标变换，在应用光照模型时（漫反射和镜面反射），把该点的对应光视角下的坐标z值与对应z-buffer中的值作比较，如果大于，说明它被遮挡了，此时就没有此光线带来的漫反射和镜面反射。

深度贴图

对于步骤一，我们可以自己申请一个帧缓冲，其中只有一个z-buffer，通过深度贴图来实现：

```
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);

// create depth texture
unsigned int 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);

// to avoid the area out of light view frustum have unrealistic shadow
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);

// attach depth texture as FBO's depth buffer
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

然后求光视角下的 View 和 Projection 矩阵（如果是平行光，我们可以使用正交投影，而如果是点光源或者是聚光灯，我们可以使用透视投影）：

```
glm::mat4 lightProj, lightView;
glm::mat4 lightSpaceMatrix;
float near_plane = 1.0f, far_plane = 10.0f;

// type of light
if (lightType == 1) {
    lightProj = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
} else {
    lightProj = glm::perspective(glm::radians(45.0f), (GLfloat)SHADOW_WIDTH /
(GLfloat)SHADOW_HEIGHT, near_plane, far_plane);
}

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

着色器的工作很简单，将物体顶点的坐标都乘以`lightSpaceMatrix`即可，z-buffer里的值，OpenGL会帮我们求得：

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

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

渲染场景

该阶段与平常有一些不一样的地方，如顶点着色器中，需要多计算一次光视角下的顶点坐标（FragPosLightSpace）：

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

然后在片段着色器中，将FragPosLightSpace的z值与深度纹理对应的值进行比较，其中有个地方需要注意，为了确保xyz值在[0,1]间，需要做如下操作：

```
// perform perspective divide (though orthographic doesn't need it)
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
// transform to [0,1] range
projCoords = projCoords * 0.5 + 0.5;
```

将z值和深度纹理中的z值作比较，如果大于的话，说明它被遮挡：

```
// 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;
// to solve the issue of Shadow Acne, we'll shadow bias
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
// check whether current frag pos is in shadow
float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
```

然后根据 shadow 的值，修改像素的漫反射和镜面反射：

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

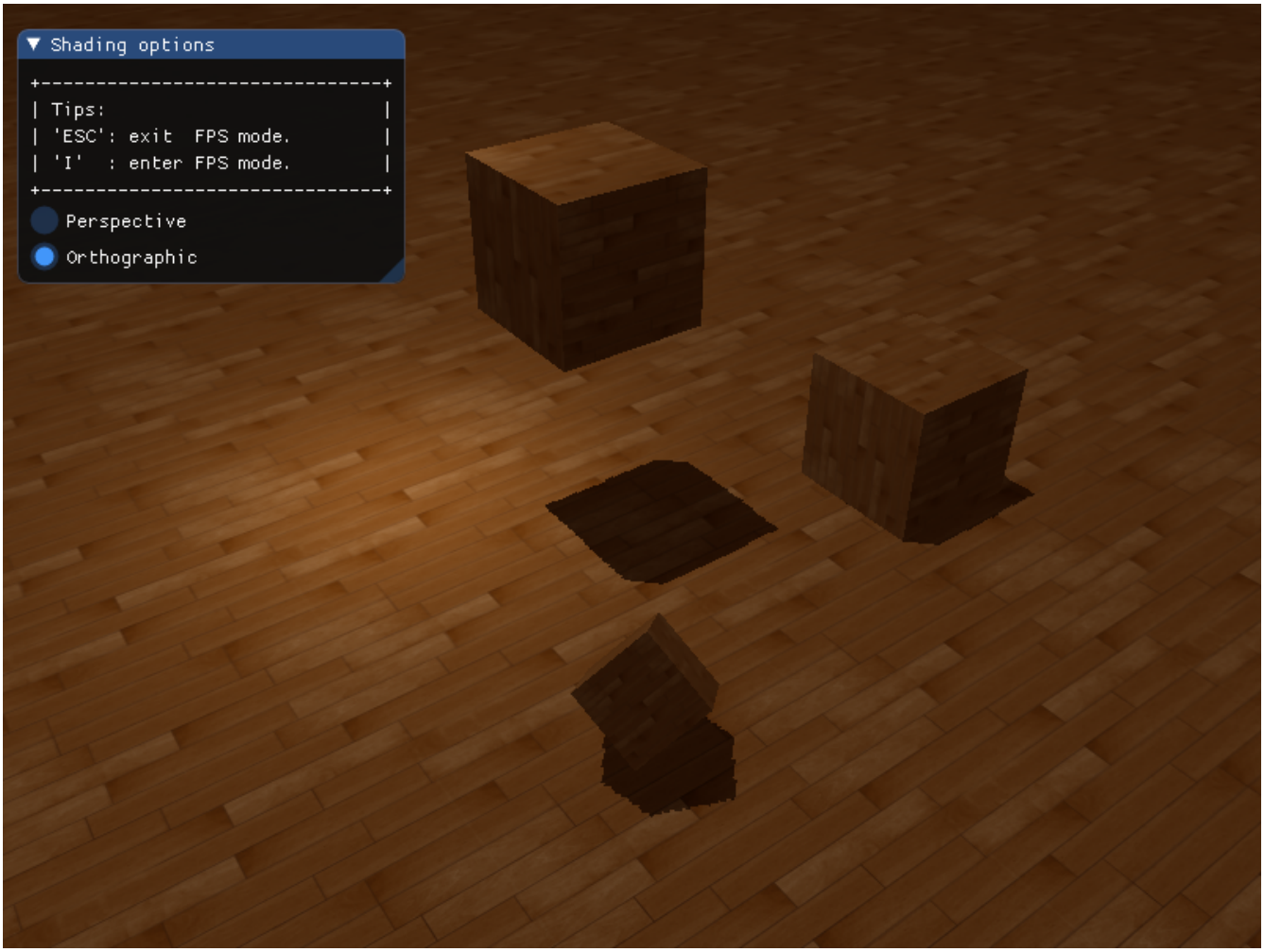
Bonus

实现光源在正交/透视两种投影下的Shadowing Mapping

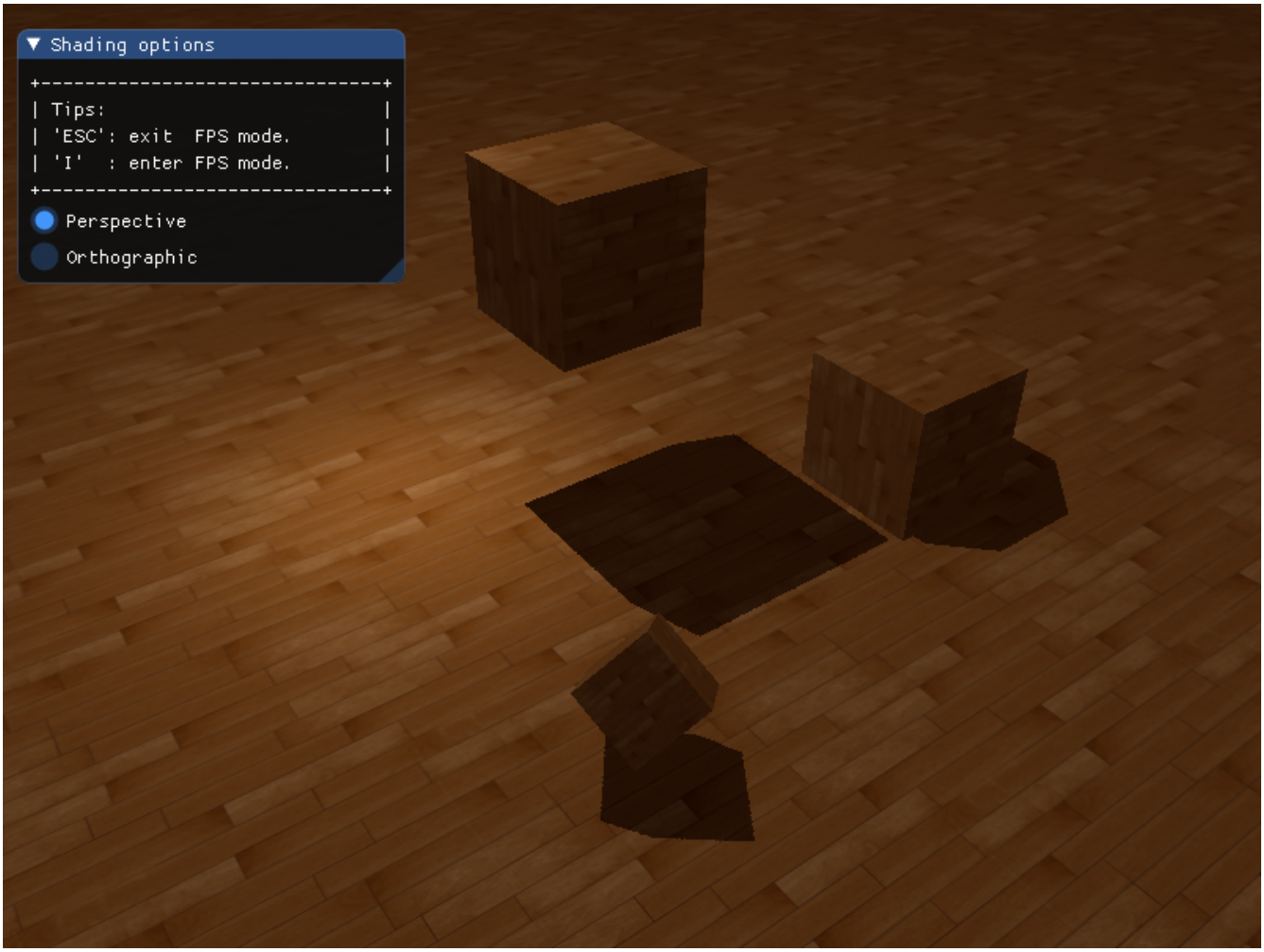
```
// type of light
if (lightType == 1) {
    lightProj = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
} else {
    lightProj = glm::perspective(glm::radians(45.0f), (GLfloat)SHADOW_WIDTH /
(GLfloat)SHADOW_HEIGHT, near_plane, far_plane);
}
```

效果

正交投影：


Shading options
+------------------------------+
| Tips: |
| 'ESC': exit FPS mode. |
| 'I'   : enter FPS mode. |
+------------------------------+
Perspective
Orthographic

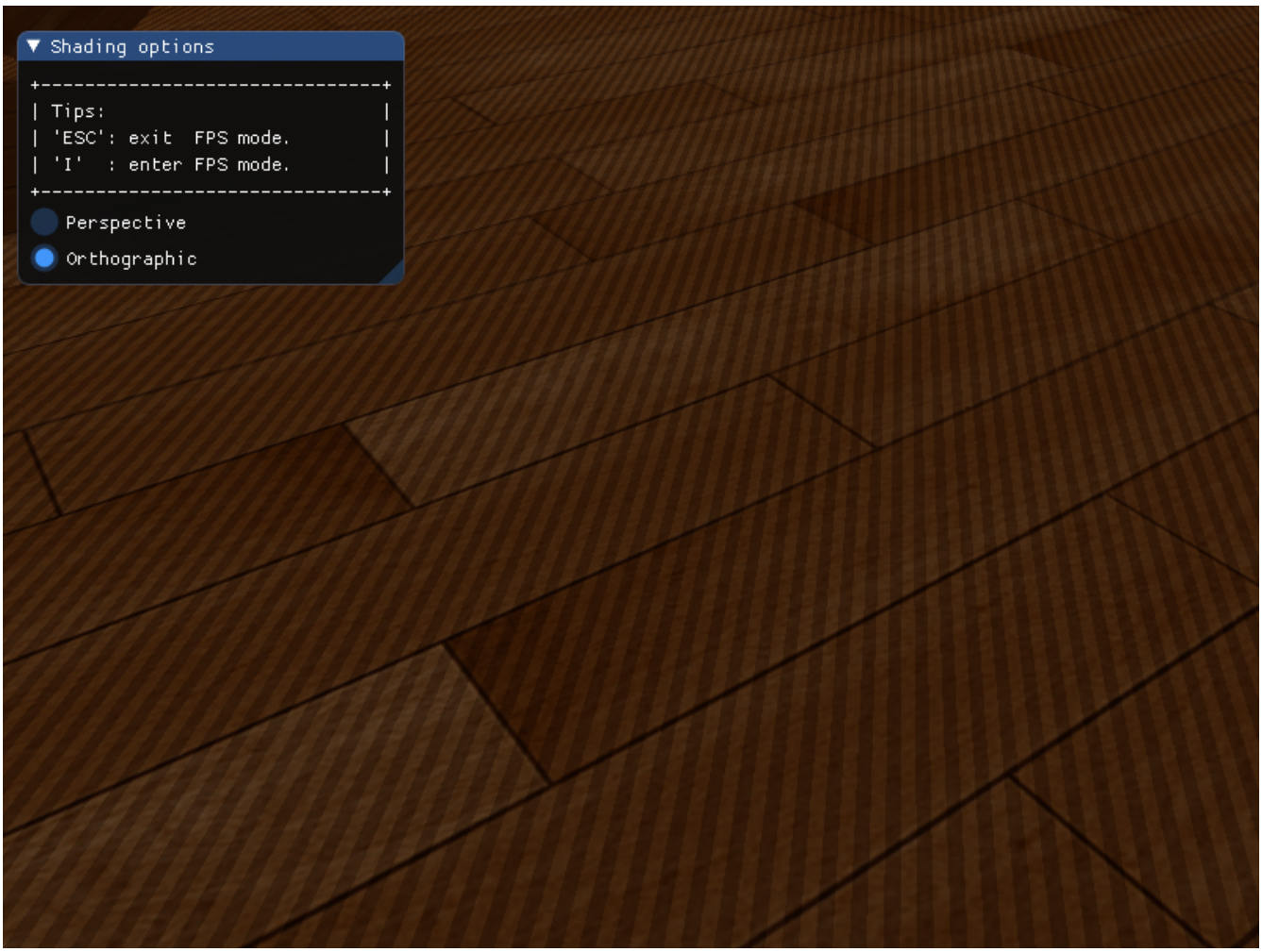
透视投影：



Shadowing Mapping 的优化

shadow bias

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，这样会出现**Shadow Acne**，即下面展示的这种条纹：



通过一个小技巧，即给每个片元加一个bias，从而不会让它们会认为有些处于表面之下:

```
// to solve the issue of Shadow Acne, we'll shadow bias
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
```

解决采样过多

超出光的视锥的投影坐标比1.0大，这样采样的深度纹理就会超出他默认的0到1的范围。根据纹理环绕方式，我们将会得到不正确的深度结果，它不是基于真实的来自光源的深度值。我们可以通过把深度贴图的纹理环绕选项设置为GL_CLAMP_TO_BORDER:

```
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);
```

现在如果我们采样深度贴图0到1坐标范围以外的区域，纹理函数总会返回一个1.0的深度值，阴影值为0.0。

而超出了光的正交视锥的远平面部分，解决这个问题也很简单，只有投影向量的z坐标小于等于1.0，我们才计算shadow，否则直接为0:

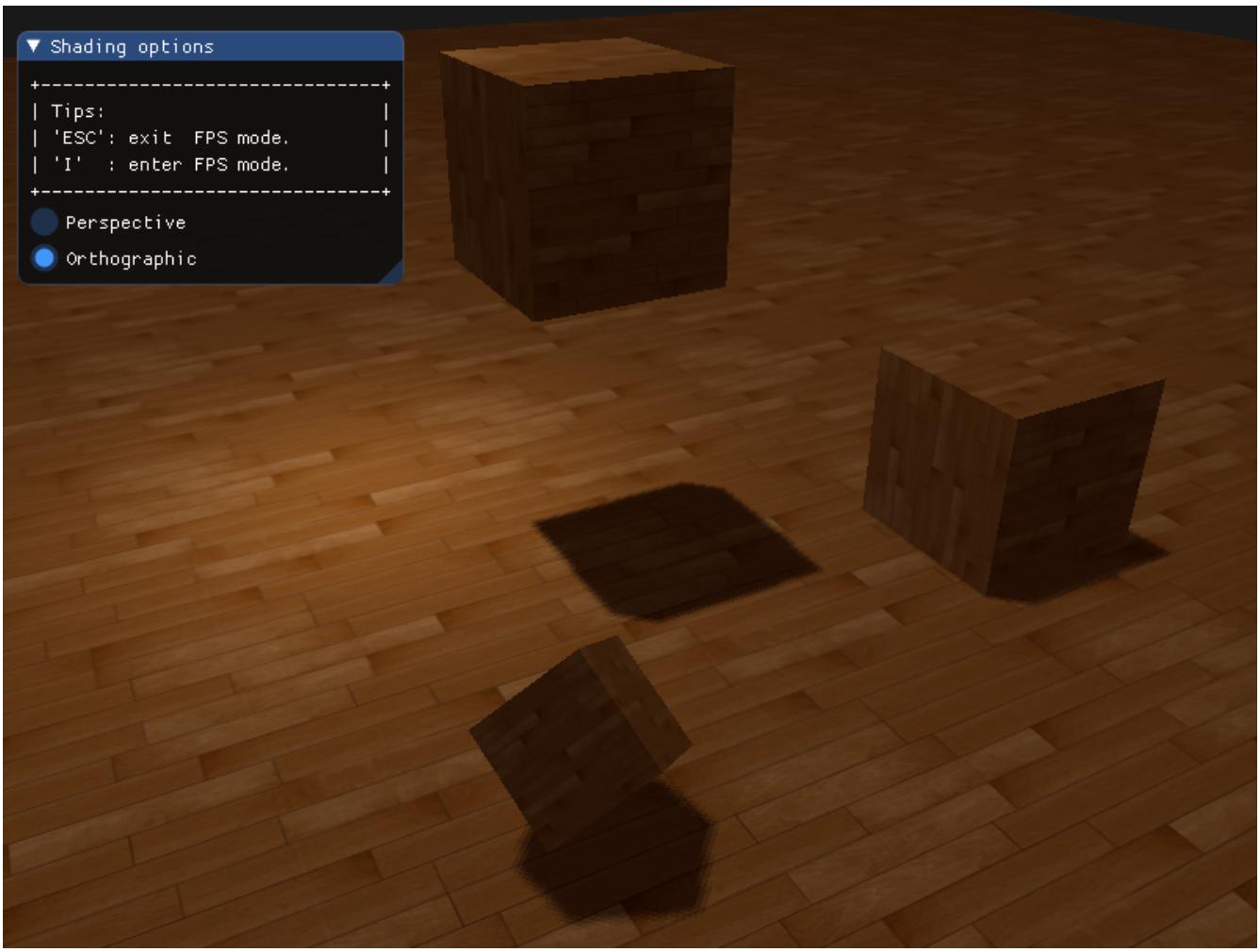
```
float shadow = 0.0;

if (currentDepth <= 1.0) {
    [...]
}
```

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。PCF 的思想很简单，对该片元周围的像素也进行采样，然后做一个平均，就达到了一种模糊平缓的作用：

```
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;
```

最终的效果



更详细的情况请看附带的GIF演示图