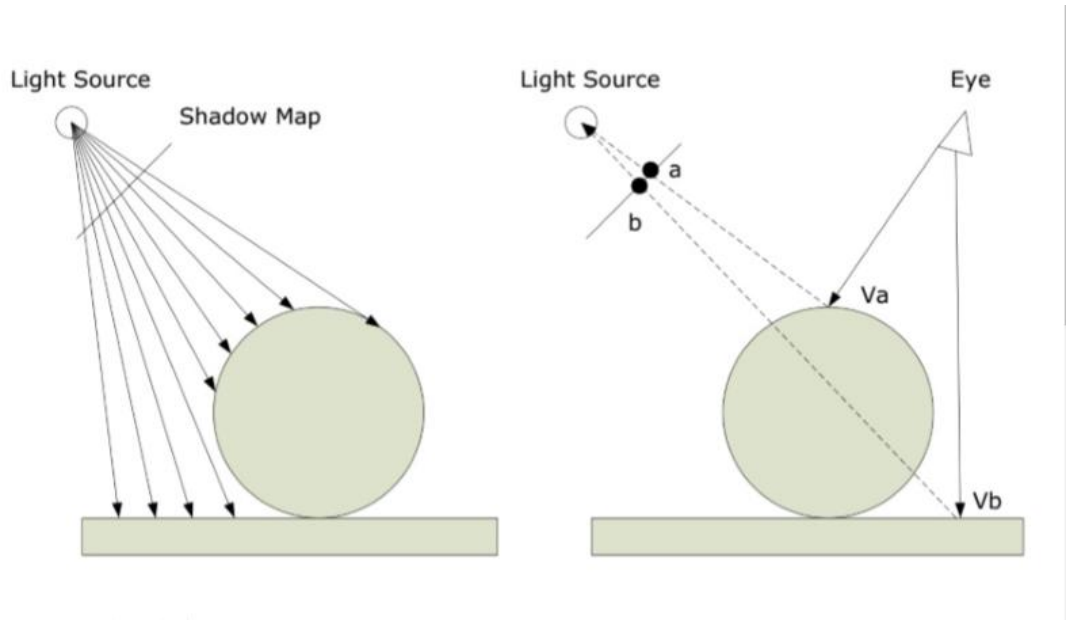
CG 第七次作业

软件工程数字媒体 15331101 洪鹏圳

Basic:

1. 实现 Shadowing Mapping

Shadowing Mapping 示意图如下：



其分为两个步骤：

（1）以光源视角渲染场景，得到深度贴图（DepthMap），并存储为 texture

（2）以 camera 视角渲染场景，使用 Shadow 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。

（1）深度贴图

我们首先需要生成一张深度贴图，深度图是从光的透视图里渲染深度纹理，用它计算阴影。因为我们需要将场景的渲染结果储存到一个纹理中，我们为渲染的深度贴图创建一个帧缓冲对象，同时把深度贴图的纹理的高宽设置为 1024（这是深度贴图的解析度）：

```
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
GLuint depthMapFBO;//阴影贴图的帧缓冲对象
glGenFramebuffers(1, &depthMapFBO);
```

接着我们创建一个 2D 纹理，提供给帧缓冲的深度缓冲使用，记得把纹理格式指定为 GL_DEPTH_COMPONENT：

```
//创建深度贴图
GLuint depthMap;
glGenTextures(1, &depthMap);//第一个参数表示输入生成纹理的数量
glBindTexture(GL_TEXTURE_2D, depthMap);//绑定纹理
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);//生成深度纹理
//为当前绑定的纹理对象设置环绕、过滤方式
//设置环绕方式
//glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);// set texture wrapping to GL_REPEAT (default wrapping method)
//glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
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);
//设置过滤方式
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

接着把生成的深度纹理作为帧缓冲的深度缓冲：

```
//把生成的深度纹理作为帧缓冲的深度缓冲
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
//不用任何颜色数据进行渲染
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

因为我们只需要从光的透视图下渲染场景时候的深度信息，因此颜色缓冲没有用。我们通过调用 glDrawBuffer 和 glReadBuffer 函数把读和绘制缓冲设置为 GL_NONE 来实现不使用任何颜色数据进行渲染。

然后我们需要设定光源的投影方式，用来确保在渲染物体的时候为每个物体设置了合适的投影和视图矩阵以及相关的模型矩阵，在这里，我使用的光源投影方式为正交投影，计算正交投影的投影矩阵：

```
//从光的位置的视野下使用不同的投影和视图矩阵来渲染场景
glm::mat4 lightProjection, lightView;//光源的投影矩阵和视图矩阵
glm::mat4 lightSpaceMatrix;//光空间的变换矩阵，将每个世界空间坐标变换到光源处所见到的空间
//将光源设置为正交投影
GLfloat near_plane = 1.0f, far_plane = 7.5f;
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);//设置正交投影矩阵
```

为了使每个物体能够变换到光源视角可见的空间中，我们需要通过 glm:lookAt 函数计算视图矩阵：

```
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));//设置视图矩阵，为从光源的位置看向场景中央
```

接着我们需要计算光空间变换矩阵，该矩阵是将每个世界空间坐标变换到光源处所见到的空间：

```
lightSpaceMatrix = lightProjection * lightView;//计算光空间变换矩阵
```

接着我们声明一个着色器 simpleDepthShader，用该着色器将顶点变换到光空间：

```
Shader simpleDepthShader("shadow_mapping_depth.vs", "shadow_mapping_depth.fs");//光空间转换着色器
```

其顶点着色器 shadow_mapping_depth.vs 为：

```
//光空间转换顶点着色器，将顶点从世界坐标变换到光空间坐标
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 anormal;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 lightSpaceMatrix;//光空间变换矩阵
uniform mat4 model;//模型变换矩阵

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

可以看到我们通过把 lightSpaceMatrix 与世界坐标系相乘将顶点从世界坐标变换到光空间坐标。

其片段着色器 shadow_mapping_depth.fs 为：

```
//光空间转换片段着色器，将顶点从世界坐标变换到光空间坐标，没有颜色缓冲，该片段着色器为空像素着色器
#version 330 core

void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}
```

因为没有颜色缓冲，因此在主函数中不用做任何操作，该着色器为空像素着色器。我们也可以把注释去掉，此时我们是显示设置深度，但是这样效率比较低，因为底层会默认去设置深度缓冲。

接着我们便可以渲染深度贴图（注意我们需要改变视口的参数以适应阴影贴图的尺寸）：

```
  //从光的点渲染场景
  simpleDepthShader.use();
  simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

  //1.首先渲染深度贴图
  glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);//改变视口的参数以适应阴影贴图的尺寸
  glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
  glClear(GL_DEPTH_BUFFER_BIT);
  RenderScene(simpleDepthShader);//渲染场景
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

其中 RenderScene 函数是绘制整个场景函数：

```
//渲染场景函数
void RenderScene(Shader &shader) {
    //**************渲染平面**************
    glm::mat4 model;//变换矩阵
    shader.setMat4("model", model);
    glBindVertexArray(planeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);

    //**************渲染三个立方体**************
    //第一个立方体
    model = glm::mat4();
    model = glm::translate(model, glm::vec3(0.0f, 1.5f, 0.0f));
    shader.setMat4("model", model);
    RenderCube();
    //第二个立方体
    model = glm::mat4();
    model = glm::translate(model, glm::vec3(2.0f, 0.0f, 1.0f));
    shader.setMat4("model", model);
    RenderCube();
    //第三个立方体
    model = glm::mat4();
    model = glm::translate(model, glm::vec3(-1.0f, 0.0f, 2.0f));
    model = glm::rotate(model, 60.0f, glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
    model = glm::scale(model, glm::vec3(0.5, 0.5, 0.5));
    shader.setMat4("model", model);
    RenderCube();
}
```

我们已经渲染了一个深度贴图，接下来我们通过将这个纹理投射到一个 2D 四边形上来检查该深度贴图是否正确：

先声明一个调试着色器：

```
Shader debugDepthQuad("debug_quad_depth.vs", "debug_quad_depth.fs");//调试着色器
```

其顶点着色器 debug_quad_depth.vs 为：

```
//四边形顶点着色器
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

uniform mat4 model;//变换矩阵
uniform mat4 view;//观察矩阵
uniform mat4 projection;//投影矩阵

out vec2 TexCoords;

void main()
{
    gl_Position = vec4(aPos, 1.0f);//获得变换后的位置
    TexCoords = vec2(aTexCoord.x, aTexCoord.y);
}
```

其片段着色器 debug_quad_depth.fs 为：

```
#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D depthMap;

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    color = vec4(vec3(depthValue), 1.0);
}
```

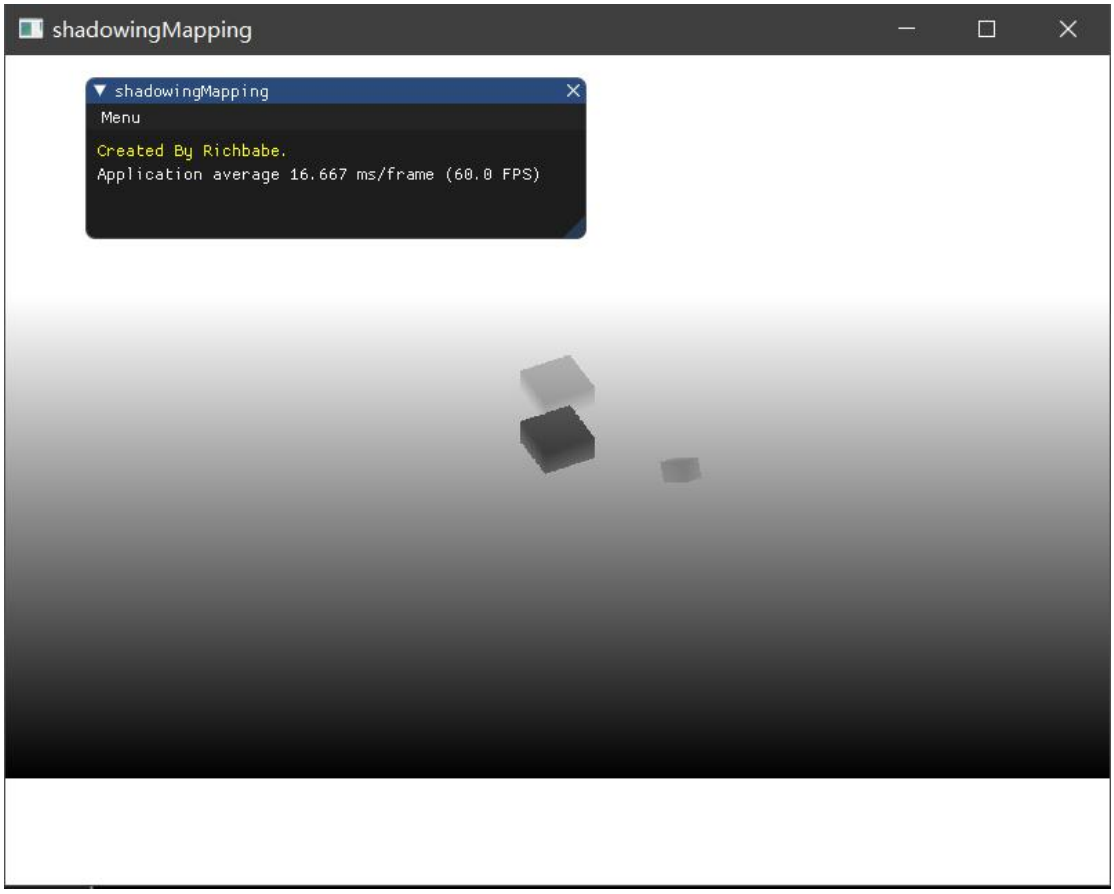
接着在主函数里渲染该四边形：

```
//Debug
debugDepthQuad.use();
debugDepthQuad.setFloat("near_plane", near_plane);
debugDepthQuad.setFloat("far_plane", far_plane);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderQuad();
```

其中渲染四边形的函数如下：

```
//画四边形函数
void RenderQuad() {
    if (quadVAO == 0) {
        //平面四个顶点
        GLfloat quadVertices[] = {
            //顶点                    //纹理坐标
            -1.0f, 1.0f, 0.0f,       0.0f, 1.0f,
            -1.0f, -1.0f, 0.0f,      0.0f, 0.0f,
            1.0f, 1.0f, 0.0f,        1.0f, 1.0f,
            1.0f, -1.0f, 0.0f,       1.0f, 0.0f,
        };
        glGenVertexArrays(1, &quadVAO);
        glGenBuffers(1, &quadVBO);
        glBindVertexArray(quadVAO);
        glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), &quadVertices, GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), (GLvoid*)0);
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
    }
    glBindVertexArray(quadVAO);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glBindVertexArray(0);
}
```

运行结果为：



可以看到我们成功将深度贴图的纹理投射到四边形上。

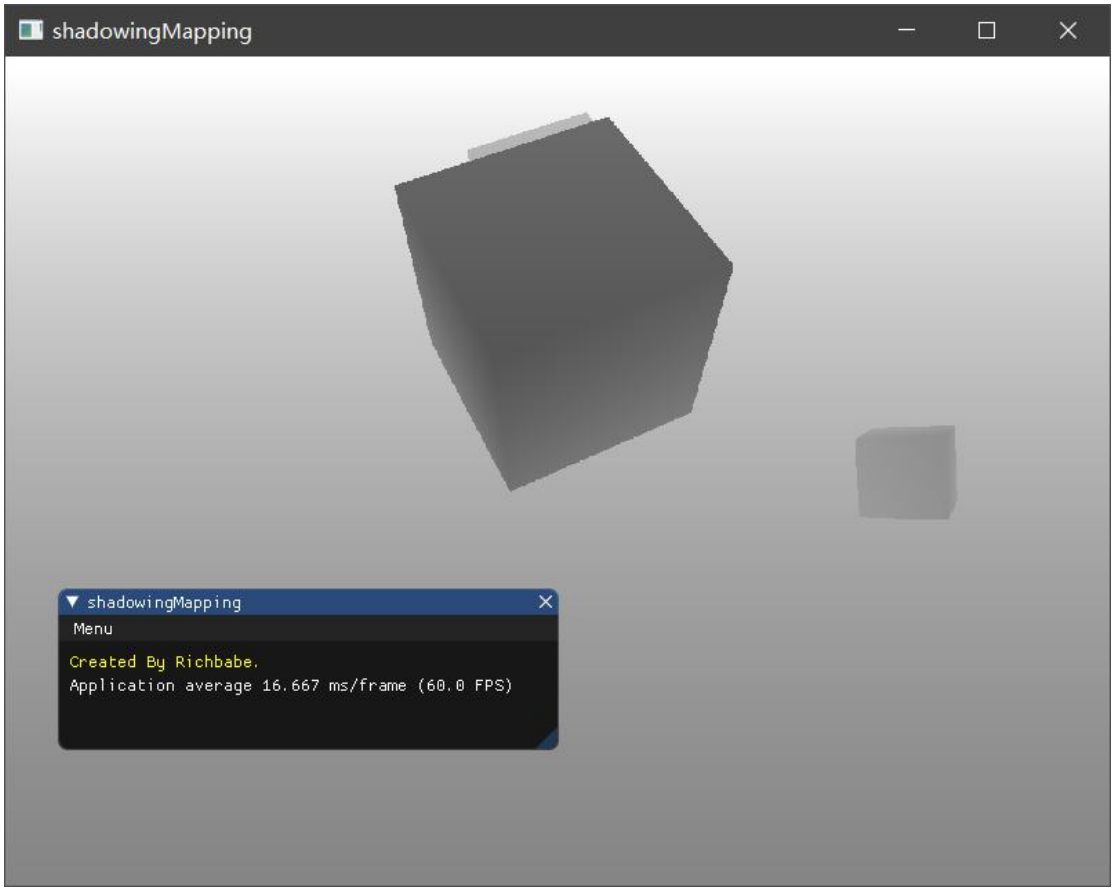
上面我们使用的光源投影方式为正交投影，当我们使用透视投影时，我们需要做如下改变，首先将光源的投影矩阵换成透视投影矩阵：

```
lightProjection = glm::perspective(camera.Zoom, (float)screenWidth / (float)screenHeight, near_plane, far_plane);//设置透视投影矩阵
```

接着把之前正交投影的非线性深度值转换为线性深度值：

```
//四边形片段着色器
#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D depthMap;//阴影纹理

uniform float near_plane;//近平面
uniform float far_plane;//远平面

float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane - near_plane));
}

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;//深度值
    color = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // 透视投影
    //color = vec4(vec3(depthValue), 1.0);// 正交投影
}
```

运行结果为：



到这里我们已经完成步骤（1），接下来我们进入步骤（2）渲染阴影。

（2）渲染阴影

首先我们声明一个像素着色器 shader:

```
Shader shader("shadow_mapping.vs", "shadow_mapping.fs");//像素着色器
```

其顶点着色器 shadow_mapping.vs 如下：

```
//像素顶点着色器
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

uniform mat4 lightSpaceMatrix;//光空间变换矩阵
uniform mat4 projection;//投影变换矩阵
uniform mat4 view;//视图变换矩阵
uniform mat4 model;//模型变换矩阵

out vec2 TexCoords;//纹理

out VS_OUT {
    vec3 FragPos;//世界坐标系中顶点位置
    vec3 Normal;//法向量
    vec2 TexCoords;//纹理坐标
    vec4 FragPosLightSpace;//光空间坐标系中顶点位置
} vs_out;

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

可以注意到，我们新增了两个输出向量，一个是 FragPos，其表示世界坐标系中顶点位置，其通过变换矩阵与局部坐标相乘得到。另一个是 FragPosLightSpace，其表示光空间坐标系中的顶点位置，其通过光空间变换矩阵(lightSpaceMatrix)与世界坐标系相乘得到。

其片段着色器 shadow_mapping.fs 如下：

```
//像素片段着色器
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;

uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;
```

```
void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;//物体颜色
    vec3 normal = normalize(fs_in.Normal);//法向量
    vec3 lightColor = vec3(1.0);//光源颜色
    //*********Blinn-Phong光照模型*********
    //环境光
    vec3 ambient = 0.15 * color;//计算环境光照强度
    //漫反射
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);//计算入射光向量（片段位置指向光源位置）
    float diff = max(dot(lightDir, normal), 0.0);//计算漫反射分量，为入射光向量与法向量的余弦值（大于0）
    vec3 diffuse = diff * lightColor;//计算漫反射光照强度
    //镜面反射
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);//计算观察向量（片段位置指向观察者所在位置）
    vec3 reflectDir = reflect(-lightDir, normal);//计算反射光向量
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;//计算镜面反射向量
    //计算阴影
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace,normal,lightDir);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
    FragColor = vec4(lighting,1.0f);
}
```

可以看到其所使用的光照模型为 Blinn-Phong 光照模型，与 Phong 光照模型不同在于镜面反射。接着我们需要加上阴影的计算，首先通过 ShadowCalculation 函数计算得到阴影元素，如果在阴影中则该阴影元素为 1，如果不再则为 0。然后把漫反射和镜面反射的光照强度的和乘以（1-阴影元素），表示这个片元有多大成分不在阴影中。

接着我们来看看 ShadowCalculation 函数是怎么计算得到阴影元素的：

首先，要检查一个片元是否在阴影中，我们需要把光空间的片元位置转换为裁剪空间的标准化设备坐标。当我们在顶点着色器输出一个裁剪空间顶点位置到 gl_Position，OpenGL 会自动进行一个透视除法，将裁剪空间坐标的范围[-w,w]归一化到[-1,1]，其通过将 x，y，z 元素除以 w 元素来实现。而光空间片元坐标 FragPosLightSpace 并没有通过 gl_Position 传到片段着色器中，因此我们需要自己做透视除法：

```
//计算阴影元素
float ShadowCalculation(vec4 fragPosLightSpace,vec3 normal,vec3 lightDir)
{
    //执行透视除法(当使用透视投影时用到)
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;//返回了片元在光空间的-1到1的范围内的坐标
```

需要注意的是，当使用正交投影矩阵时，顶点的 w 元素仍保持不变，这一步实际上没有意义，但是当使用透视投影矩阵时，顶点的 w 元素会改变，因此我们必须保留改行使得阴影计算函数在两种投影矩阵下都有效。

因为深度贴图中的深度范围为[0,1]，因此我们需要将执行了透视除法的光空间坐标 projCoords 的 x,y,z 分量变换到[0,1]：

```
//变换到[0,1]的范围
projCoords = projCoords * 0.5 + 0.5;
```

接着我们通过 projCoords 的 x,y 分量获取光的位置视野下的最近深度：

```
//取得最近点的深度(使用[0,1]范围下的fragPosLight当坐标)
float closestDepth = texture(shadowMap, projCoords.xy).r;
```

接着获取片元的当前深度，其深度值即为投影向量 projCoords 的 z 坐标（等价于来自光的透视视角的片元深度）：

```
//取得当前片元在光源视角下的深度
float currentDepth = projCoords.z;
```

最后我们对比当前深度 currentDepth 是否高于最近点深度 closestDepth，如果是，则说明该片元在阴影中，函数返回 1，如果不是则返回 0：

```
// 检查当前片元是否在阴影中
float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;

return shadow;
```

然后我们在源代码中激活着色器 shader,设置其投影矩阵、视图矩阵：

```
//2.像往常一样渲染场景，但这次使用深度贴图
glViewport(0, 0, screenWidth, screenHeight);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shader.use();
glm::mat4 projection = glm::perspective(camera.Zoom, (float)screenWidth / (float)screenHeight, 0.1f, 100.0f);//投影矩阵
glm::mat4 view = camera.GetViewMatrix();//视图矩阵
shader.setMat4("projection", projection);
shader.setMat4("view", view);
```

接着将光源参数传入片段着色器：

```
//将光源参数传进像素片段着色器
shader.setVec3("lightPos", lightPos);
shader.setVec3("viewPos", camera.Position);
shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
```

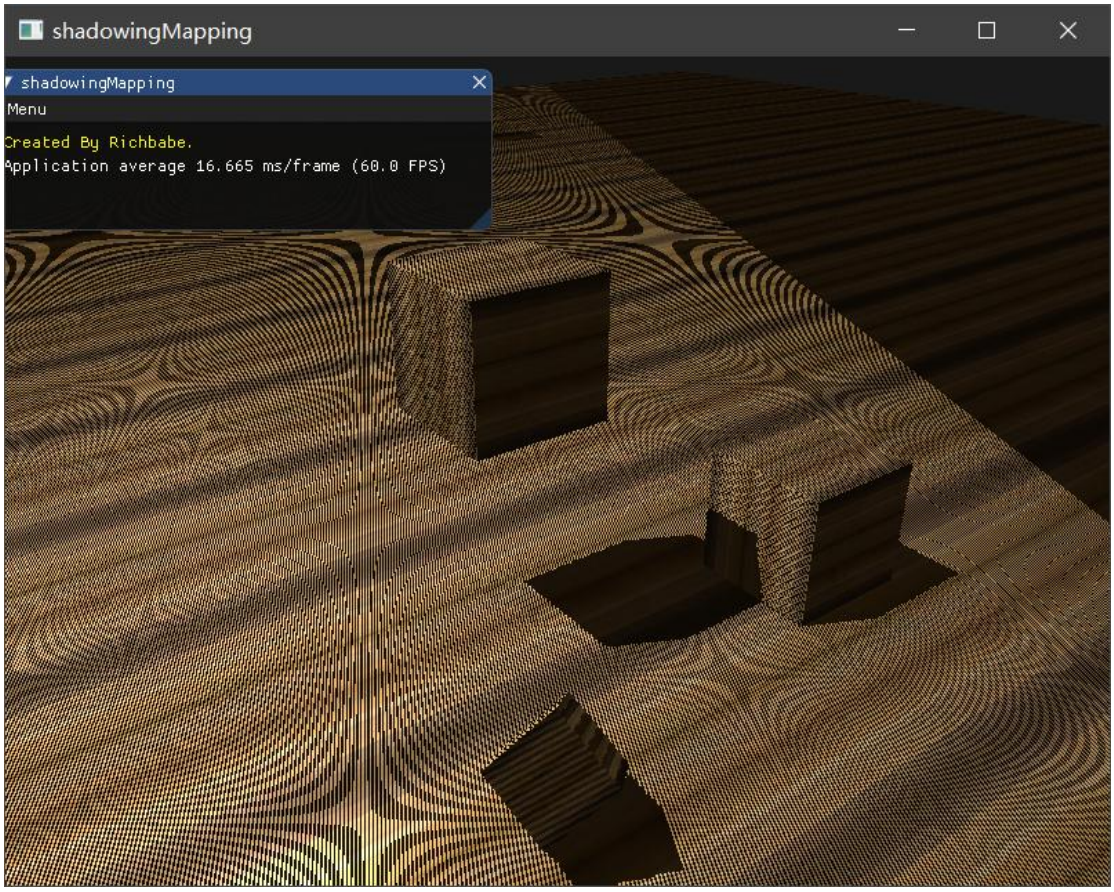
激活自身纹理和阴影纹理并绑定：

```
glActiveTexture(GL_TEXTURE0);//激活物体自身纹理
glBindTexture(GL_TEXTURE_2D, woodTexture);//绑定物体自身纹理
glActiveTexture(GL_TEXTURE1);//激活阴影纹理
glBindTexture(GL_TEXTURE_2D, depthMap);//绑定阴影纹理
```

最后渲染场景：

```
RenderScene(shader);//渲染场景
```

运行结果为：



可以看到地板上和立方体上已经生成了阴影，但效果不是很好，我们接下来对其进行优化。

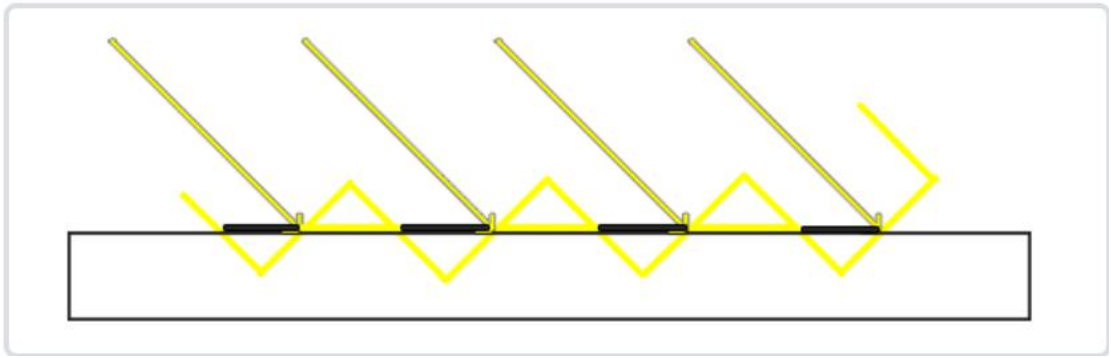
Bonus:

2. 优化 Shadowing Mapping

（1）优化 1:解决阴影失真

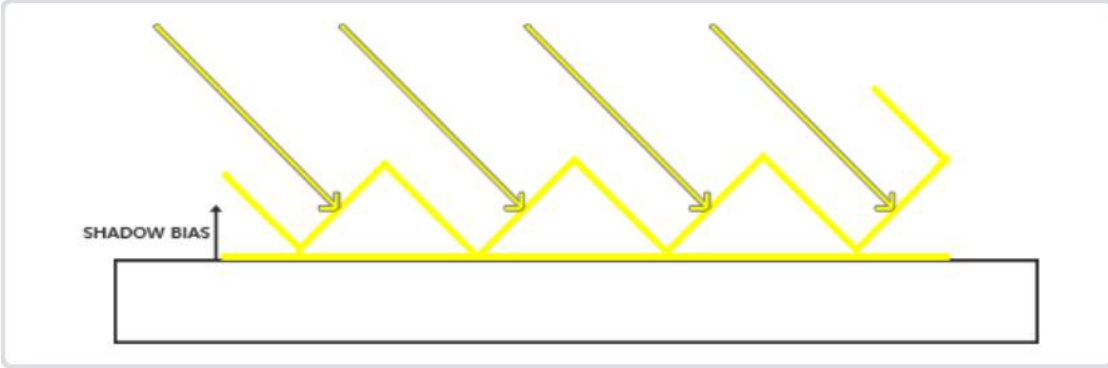
从上面的运行结果我们可以看到在地板上和立方体上有很多线条样式：

这正是阴影失真（Shadow Acne）导致的，其成因如下：



由于阴影贴图受限于解析度，因此当距离光源较远时多个片元就会从深度贴图的同一个值去采样。图中的每个斜坡代表深度贴图中一个单独的纹理像素，多个片元从同一个深度值进行

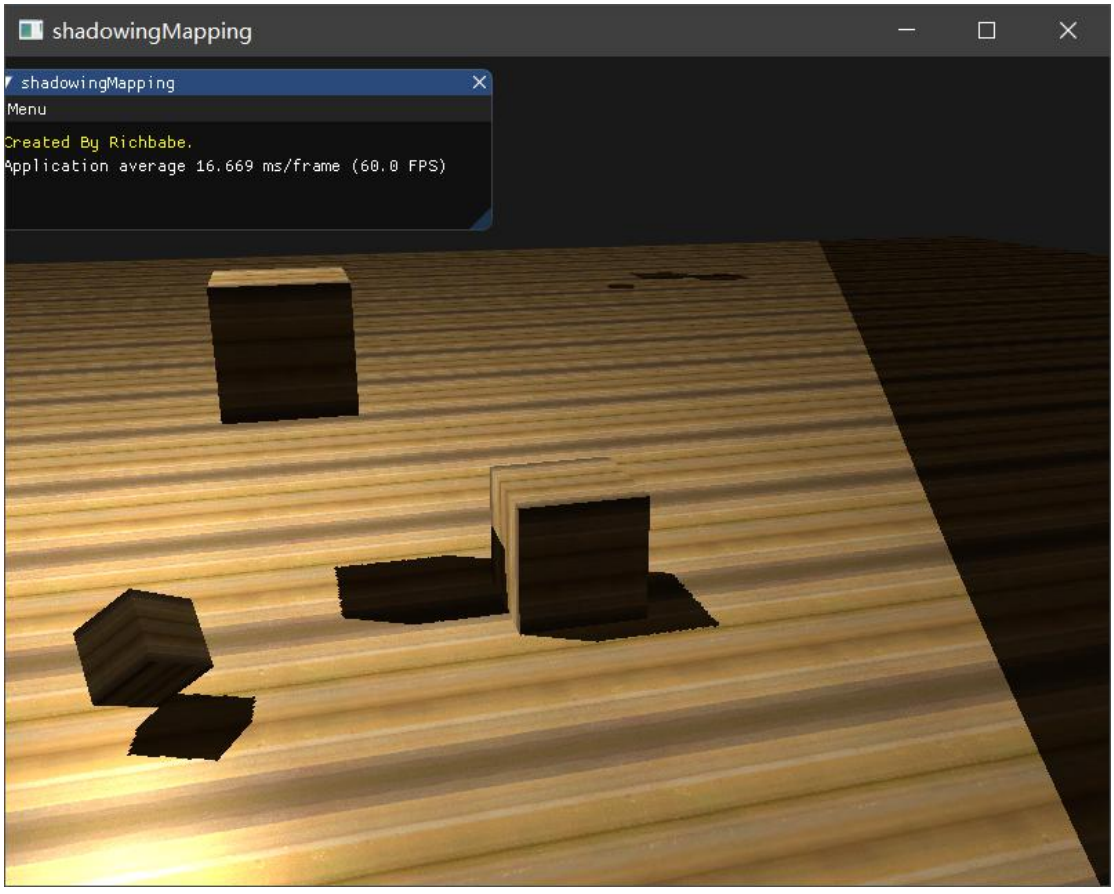
采样。当光源以一个角度照向平面时，深度贴图就会从一个角度下进行渲染，此时多个片元就会从同一个斜坡的深度纹理像素中采样，有些在地板上，有些在地板下，因此有些片元被认定在了阴影之中，有些不在，因此便会出现图片中的条纹样式。
我们通过阴影偏移（shadow bias）来解决这个问题，我们对深度贴图应用一个偏移量，因此片元就不会被错误地认为在表面之下了：



在像素片段着色器中，我们通过以下办法来实现偏移量：

```
//检查当前片元是否在阴影中
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);//偏移量(运用阴影偏移)
float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```

其中偏移量的最大值是 0.05，最小值是 0.005，其取值基于表面法线和光照方向。当地板和光源垂直时，法向量和光照方向的余弦值为 1，此时偏移量取最小值。当光源是斜着照向地板时，偏移量的值会变大。在使用了阴影偏移后，运行结果为：

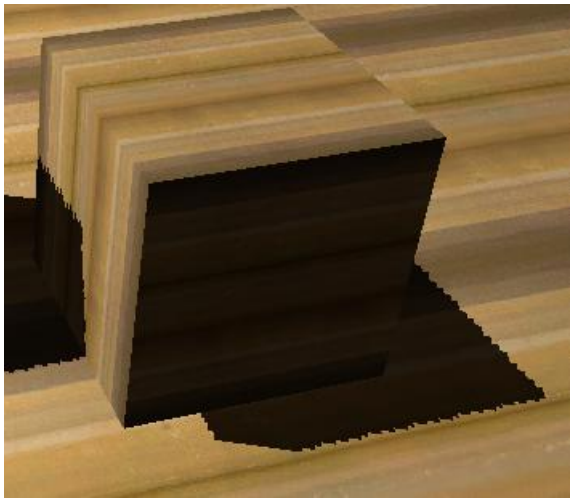


（2）优化 2：解决悬浮

当我们把偏移量的设置太大时，比如：

```
//检查当前片元是否在阴影中
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.1);//偏移量(运用阴影偏移)
```

我们会发现阴影相对实际物体位置的偏移过大：

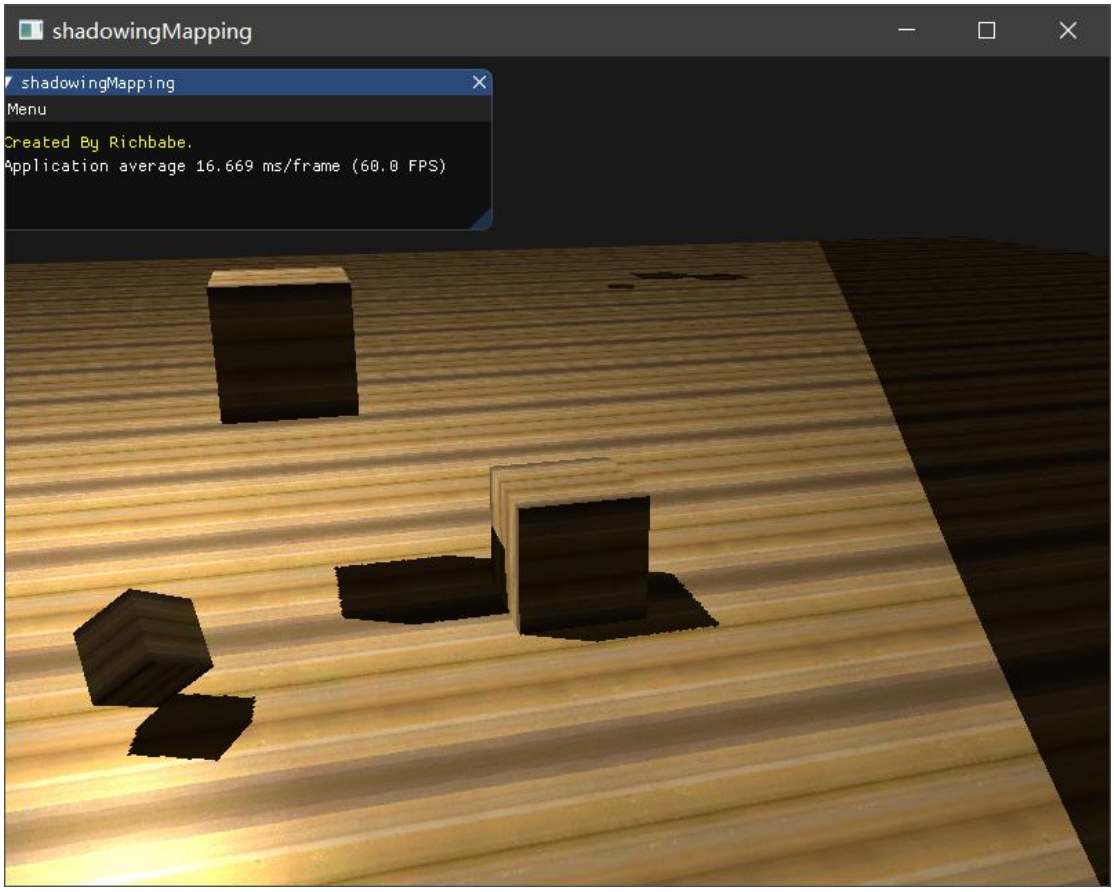


这种现象称为悬浮(Peter Panning)，因为物体看起来是悬浮在表面上的（而实际紧贴表面）。要解决悬浮现象，我们可以在渲染深度贴图时使用正面剔除(front face culling)，在渲染之后再把剔除方式改为 OpenGL 默认的背面剔除：

```
//1.首先渲染深度贴图
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);//改变视口的参数以适应阴影贴图的尺寸
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glCullFace(GL_FRONT);
RenderScene(simpleDepthShader);//渲染场景
glCullFace(GL_BACK);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

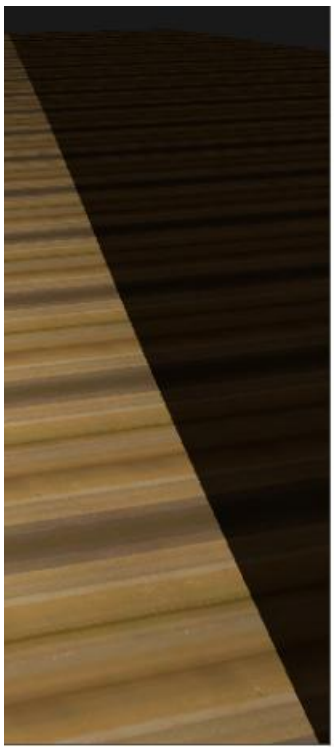
这种方法只能针对实体物体，但是对于地板来说，其是一个单独的平面，不会被完全剔除，因此要避免悬浮现象，最好还是找到合适的偏移值。

（3）优化 3：解决采样过多



从这张图可以看到在远处也存在着三个立方体的阴影：

而且超出光照区域就自动变成阴影：



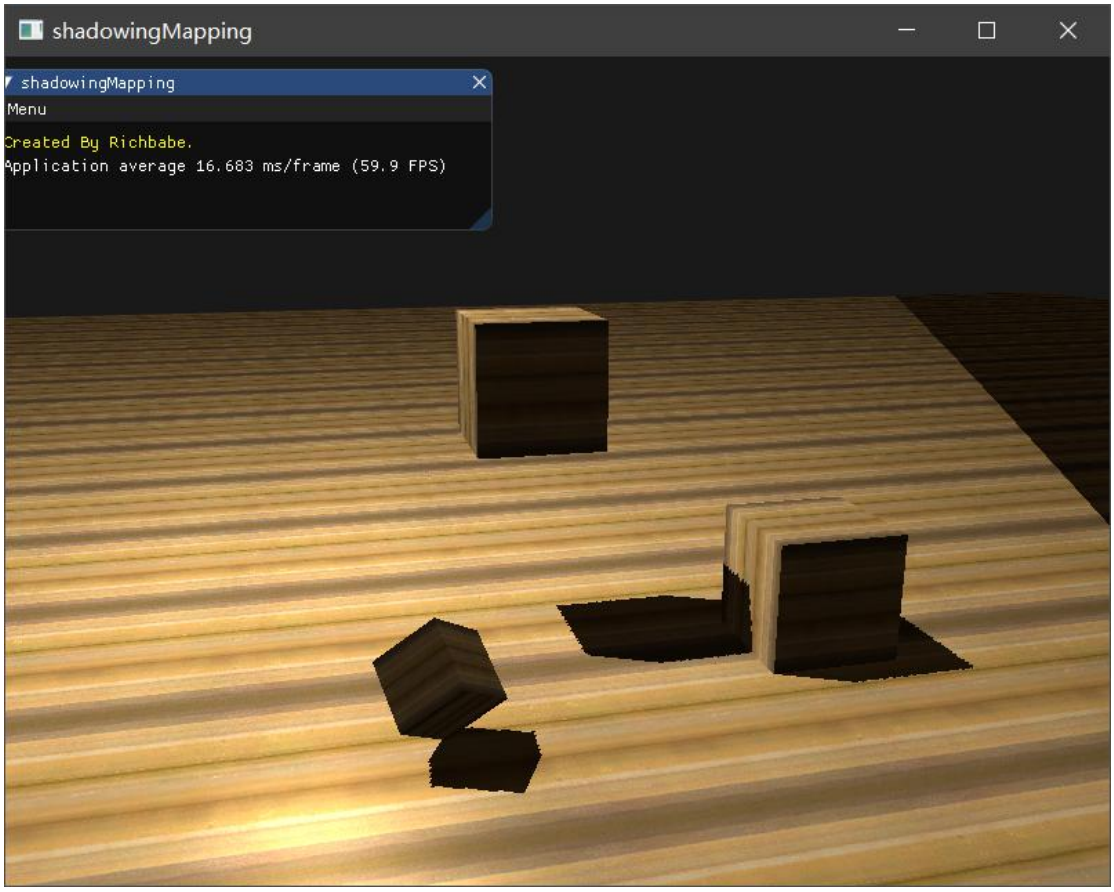
这是因为坐标超出 1.0 时，采样的深度纹理就会超出他默认的 0 到 1 的范围。根据我们之前设置的深度贴图的环绕方式：

```
//设置环绕方式
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);// set texture wrapping to GL_REPEAT (default wrapping method)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

我们将深度贴图的环绕方式设置为了 GL_REPEAT,因此当超出深度贴图的范围时，其就会重复之前的贴图，因此在远处也生成三个立方体的阴影。要解决这个问题，我们可以将所有超出深度贴图的坐标设置其深度是 1.0，然后储存一个边框颜色，把深度贴图的纹理环绕方式设置为 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，当超出深度贴图的范围时，纹理函数总会返回一个 1.0 的深度值，阴影值为 0.0，运行效果如下：

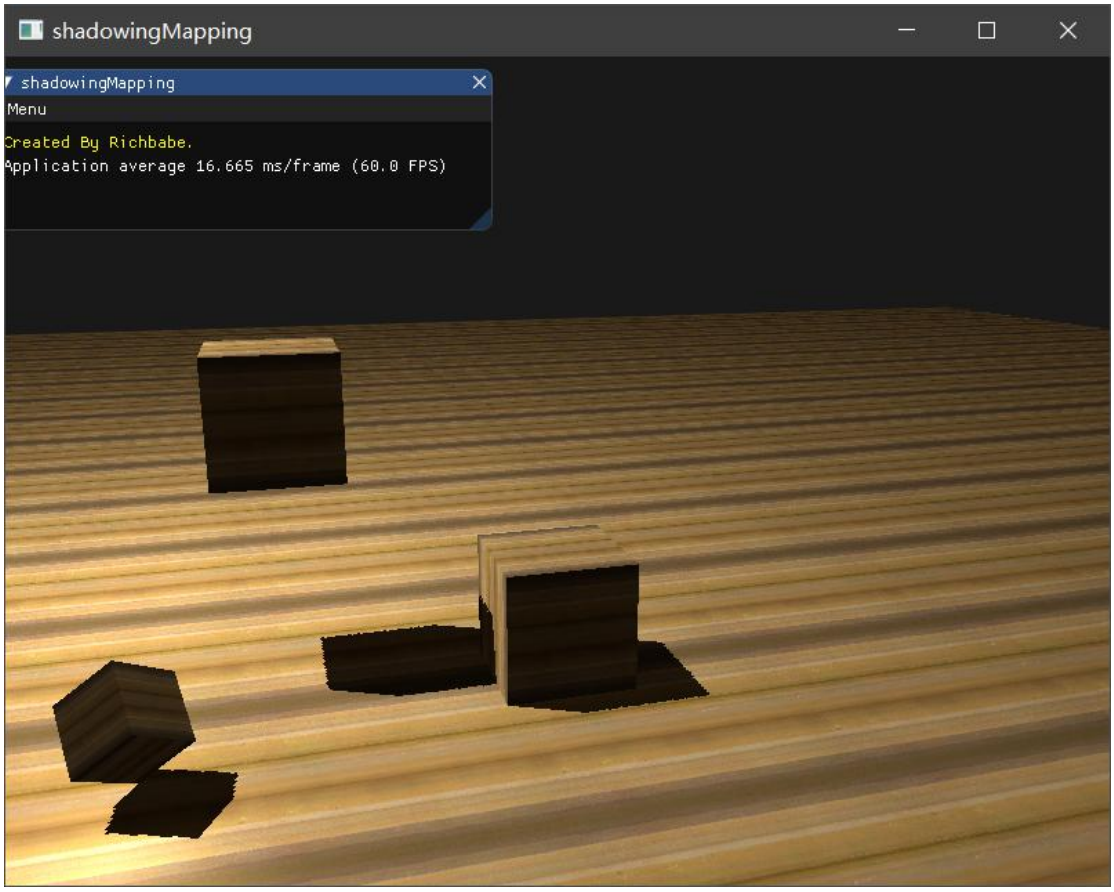


可以看到远处已经没有立方体的三个阴影，但是在右边仍有一部分是黑暗区域。这是因为黑暗区域的坐标超出了光的正交视锥的远平面（之前我把远平面设置为 7.5f）。当一个点比光的远平面还要远时，他的投影坐标的 z 坐标大于 1.0。这种情况下，GL_CLAMP_TO_BORDER 环绕方式就不起作用，因为我们把坐标的 z 元素和深度贴图的值进行对比，当大于 1.0 时就会默认该坐标在阴影之中。

要解决这个问题，我们可以在像素片段着色器中加一个判断，当投影向量的 z 坐标大于 1.0 时，把 shadow 的值设为 0：

```
//当点比光的远平面还要远时，阴影不起作用
if(projCoords.z > 1.0){
    shadow = 0.0;
}

return shadow;
```

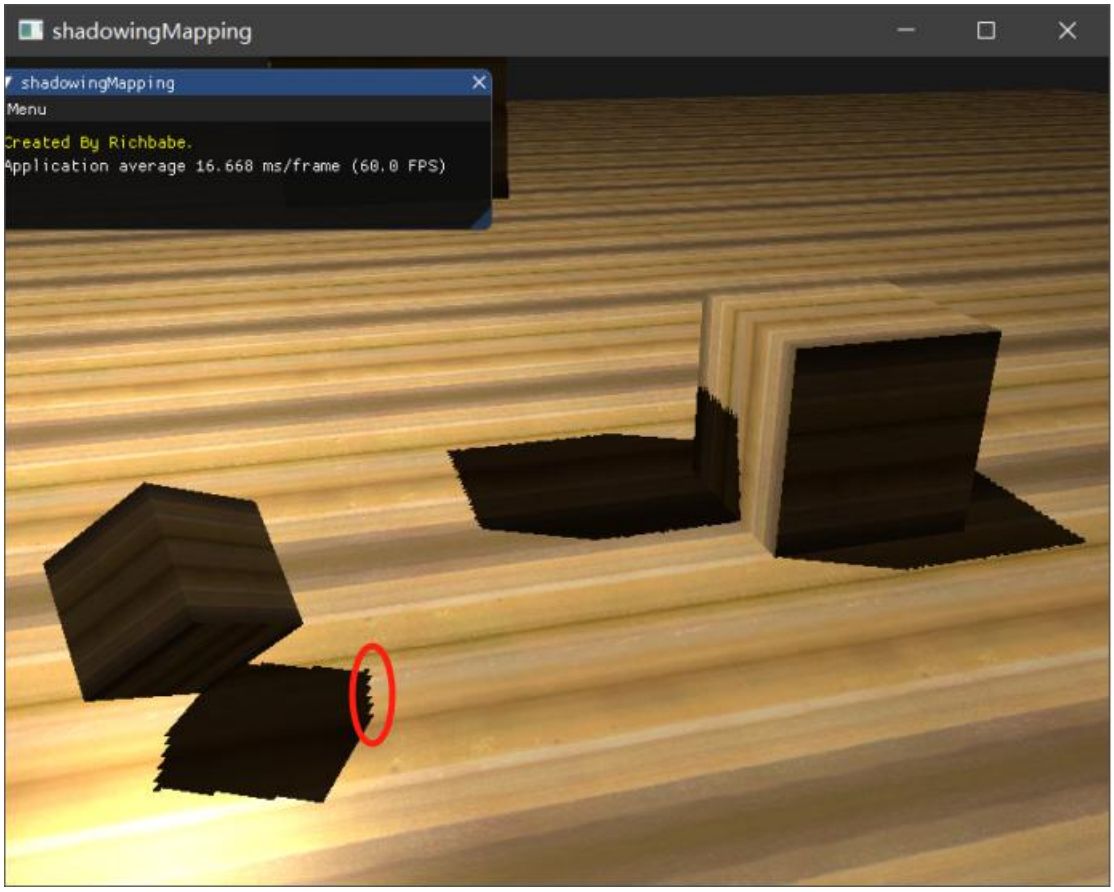
运行效果为：



可以看到采样过多的问题已经被解决了，不过这意味着只有在深度贴图范围以内的被投影的片元坐标才有投影，任何超出范围的都没有投影。

（4）优化 4：解决抗锯齿

当放大阴影时，我们可以看到阴影映射对解析度的依赖变得很明显。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：



我们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。

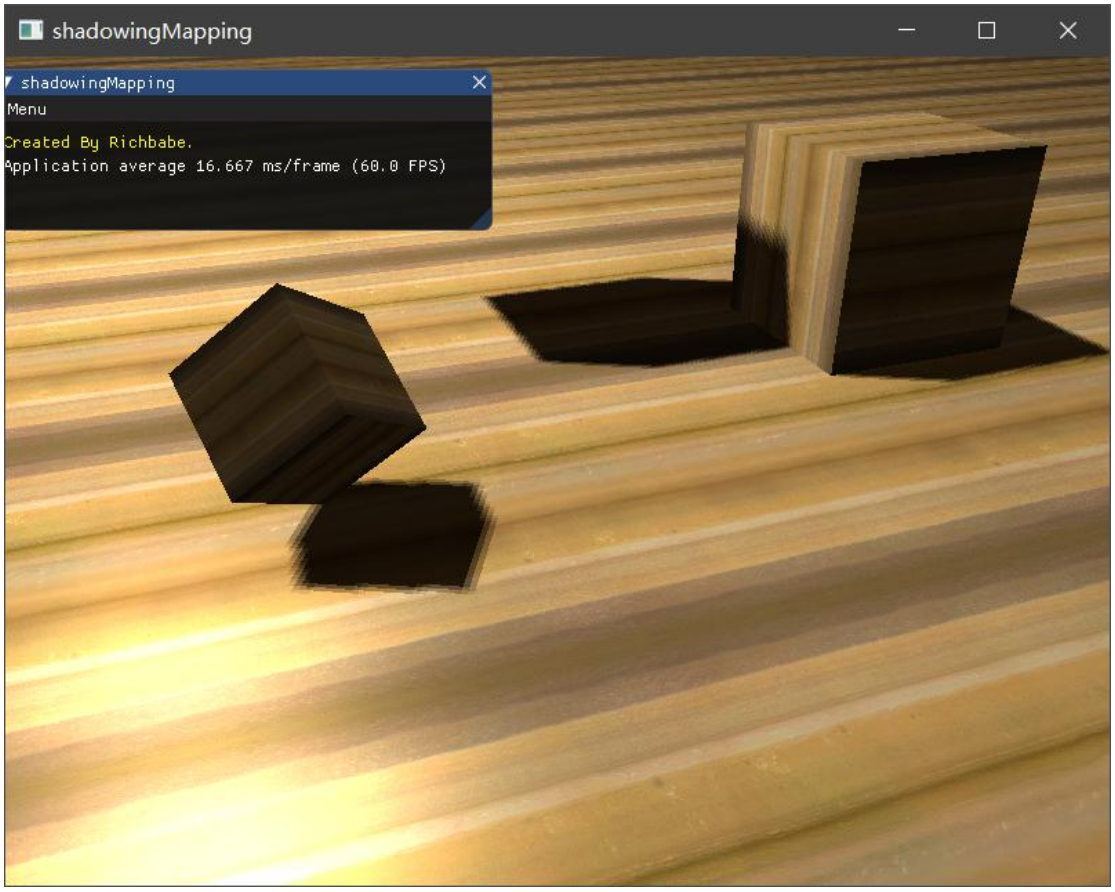
在这里我采用另一个解决方案：PCF（percentage-closer filtering），这是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能也可能不在阴影中。所有的次生结果结合在一起，取平均值，就可以得到柔和阴影。

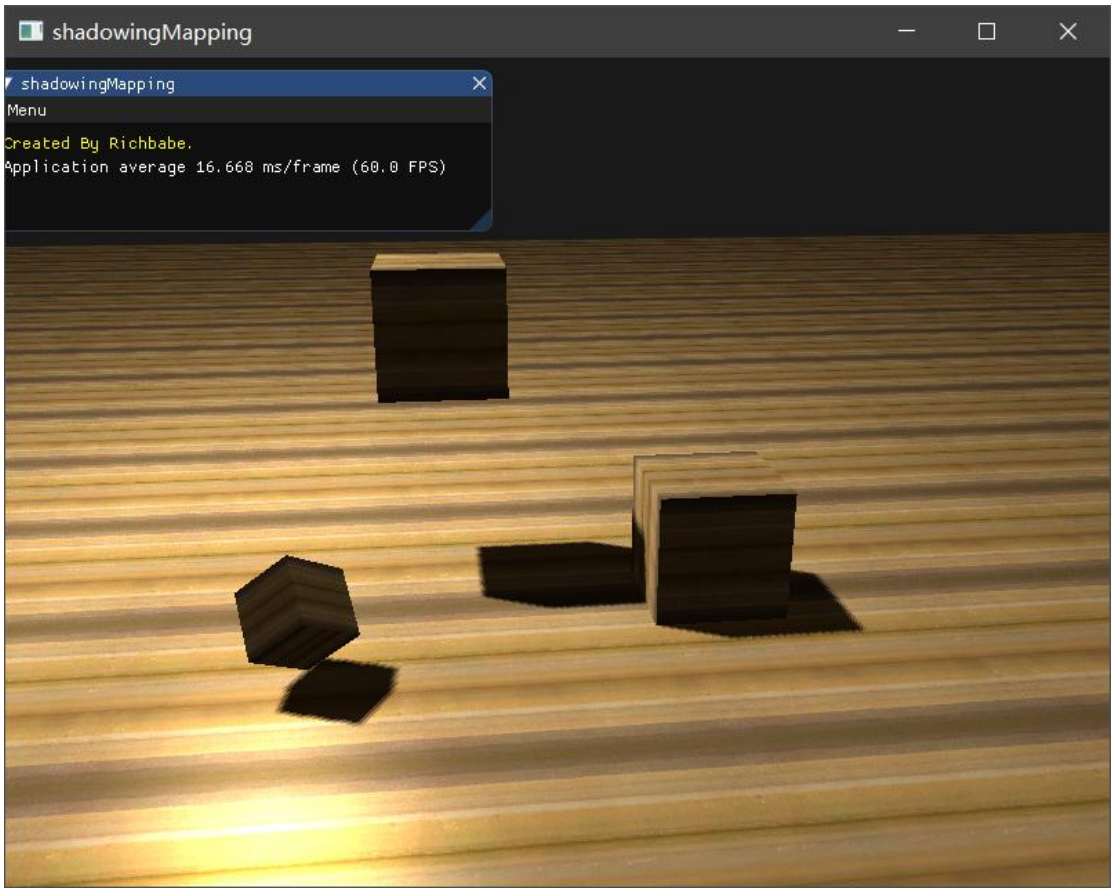
一个简单的 PCF 实现是从纹理像素四周对深度贴图采样，然后把结果平均起来：

```
//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;
```

这个 textureSize 返回一个给定采样器纹理的 0 级 mipmap 的 vec2 类型的宽和高。用 1 除以它返回一个单独纹理像素的大小，我们用以对纹理坐标进行偏移，确保每个新样本来自不同的深度值。在这里我们采样得到 9 个值 1，他们在投影坐标的 x 和 y 值的周围，为阴影阻挡进行测试，并对样本总数目结果取平均值。

运行结果为：


shadowingMapping
shadowingMapping
Menu
Created By Richbabe.
Application average 16.667 ms/frame (60.0 FPS)


shadowingMapping
shadowingMapping
Menu
Created By Richbabe.
Application average 16.668 ms/frame (60.0 FPS)

可以看到如果从远一点的距离看去，阴影效果好了很多，但是放大后阴影贴图仍有不真实感，但相对与之前锯齿状的阴影已经好了很多了。
具体的效果可以看演示视频。