# Homework 7 - Shadowing Mapping

> 主函数在main.cpp，shader.h和camera.h为之前的shader类和camera类，h7shader2.vs和h7shader2.fs主要用来把顶点切换到光空间，用来渲染深度贴图，h7shader1.vs和h7shader1.fs用深度贴图来渲染场景产生阴影。

首先先创建场景中的object和平面，在这里物体的渲染需要用到贴图，所以在顶点的数据中需要新增纹理坐标。平面的数据以及绑定缓冲：

```
    // set up vertex data (and buffer(s)) and configure vertex attributes
    float planeVertices[] = {
        // positions            // normals         // texcoords
        25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,  25.0f,  0.0f,
        -25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,   0.0f,  0.0f,
        -25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,   0.0f, 25.0f,

        25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,  25.0f,  0.0f,
        -25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,   0.0f, 25.0f,
        25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,  25.0f, 25.0f
    };
    // plane VAO
    unsigned int 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, 8 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
    glBindVertexArray(0);
```

立方体数据：

```
// renderCube() renders a 1x1 3D cube in NDC.
unsigned int cubeVAO = 0;
unsigned int cubeVBO = 0;
void renderCube()
{
    // initialize (if necessary)
    if (cubeVAO == 0)
    {
        float vertices[] = {
            // back face
            -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 0.0f, 0.0f, // bottom-left
             1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 1.0f, 1.0f, // top-right
             1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 1.0f, 0.0f, // bottom-right
             1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 1.0f, 1.0f, // top-right
            -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 0.0f, 0.0f, // bottom-left
            -1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 0.0f, 1.0f, // top-left
            // front face
            -1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f, 0.0f, // bottom-left
             1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 1.0f, 0.0f, // bottom-right
             1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 1.0f, 1.0f, // top-right
             1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 1.0f, 1.0f, // top-right
            -1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f, 1.0f, // top-left
            -1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f, 0.0f, // bottom-left
            // left face
            -1.0f,  1.0f,  1.0f, -1.0f,  0.0f,  0.0f, 1.0f, 0.0f, // top-right
            -1.0f,  1.0f, -1.0f, -1.0f,  0.0f,  0.0f, 1.0f, 1.0f, // top-left
            -1.0f, -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, 0.0f, 1.0f, // bottom-left
            -1.0f, -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, 0.0f, 1.0f, // bottom-left
            -1.0f, -1.0f,  1.0f, -1.0f,  0.0f,  0.0f, 0.0f, 0.0f, // bottom-right
            -1.0f,  1.0f,  1.0f, -1.0f,  0.0f,  0.0f, 1.0f, 0.0f, // top-right
            // right face
             1.0f,  1.0f,  1.0f,  1.0f,  0.0f,  0.0f, 1.0f, 0.0f, // top-left
             1.0f, -1.0f, -1.0f,  1.0f,  0.0f,  0.0f, 0.0f, 1.0f, // bottom-right
             1.0f,  1.0f, -1.0f,  1.0f,  0.0f,  0.0f, 1.0f, 1.0f, // top-right
             1.0f, -1.0f, -1.0f,  1.0f,  0.0f,  0.0f, 0.0f, 1.0f, // bottom-right
             1.0f,  1.0f,  1.0f,  1.0f,  0.0f,  0.0f, 1.0f, 0.0f, // top-left
             1.0f, -1.0f,  1.0f,  1.0f,  0.0f,  0.0f, 0.0f, 0.0f, // bottom-left
            // bottom face
            -1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f, 0.0f, 1.0f, // top-right
             1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f, 1.0f, 1.0f, // top-left
             1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f, 1.0f, 0.0f, // bottom-left
             1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f, 1.0f, 0.0f, // bottom-left
            -1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f, 0.0f, 0.0f, // bottom-right
            -1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f, 0.0f, 1.0f, // top-right
            // top face
```

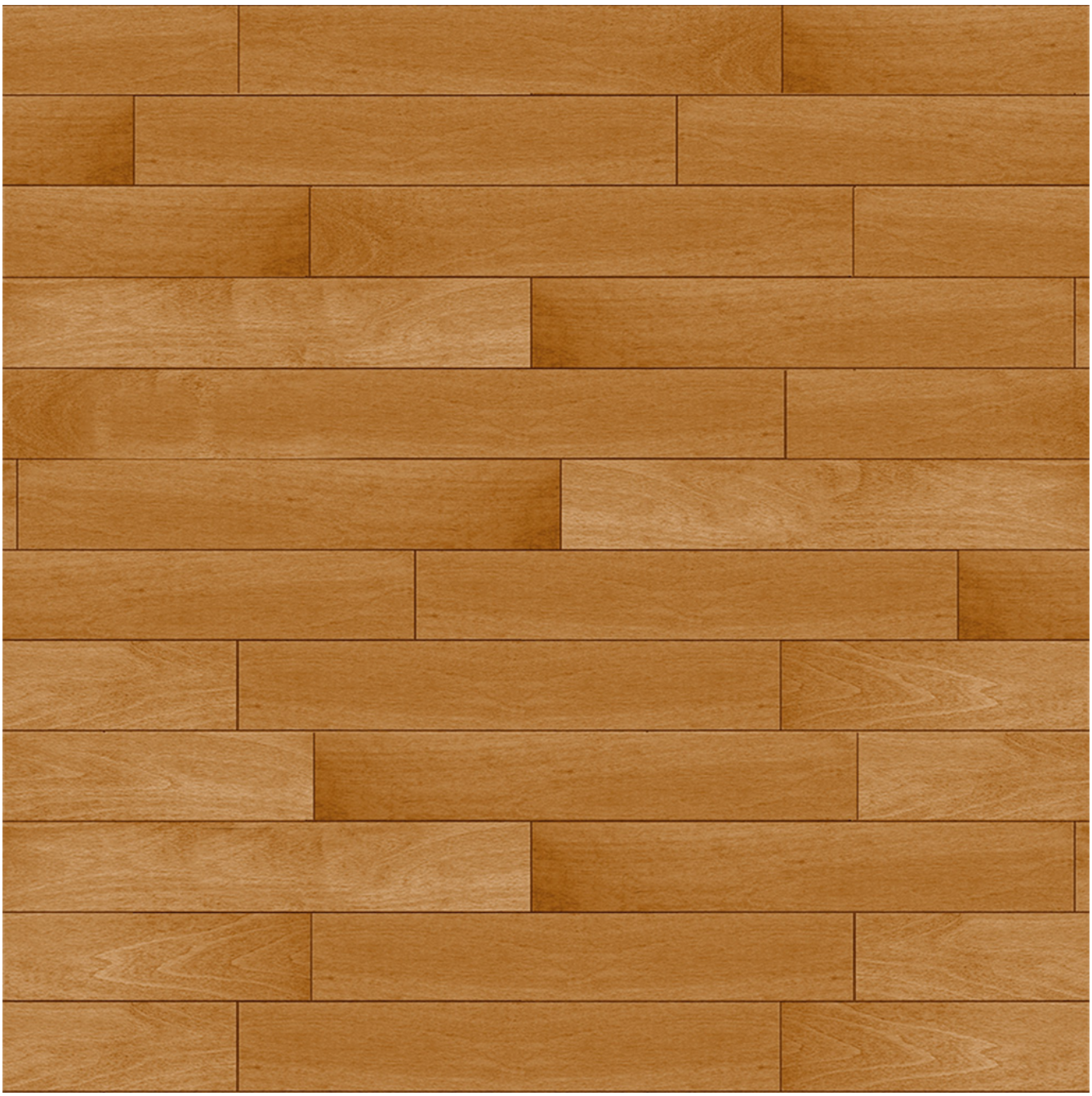
```
            // top face
            -1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f, 0.0f, 1.0f, // top-left
             1.0f,  1.0f , 1.0f,  0.0f,  1.0f,  0.0f, 1.0f, 0.0f, // bottom-right
             1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f, 1.0f, 1.0f, // top-right
             1.0f,  1.0f,  1.0f,  0.0f,  1.0f,  0.0f, 1.0f, 0.0f, // bottom-right
            -1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f, 0.0f, 1.0f, // top-left
            -1.0f,  1.0f,  1.0f,  0.0f,  1.0f,  0.0f, 0.0f, 0.0f  // bottom-left
        };

        glGenVertexArrays(1, &cubeVAO);
        glGenBuffers(1, &cubeVBO);
        // fill buffer
        glBindBuffer(GL_ARRAY_BUFFER, cubeVBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
        // link vertex attributes
        glBindVertexArray(cubeVAO);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindVertexArray(0);
    }
```

```
    // render Cube
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
}
```

这里把立方体数据和缓冲的绑定放到函数里方便多次调用，然后是渲染平面和立方体的函数：

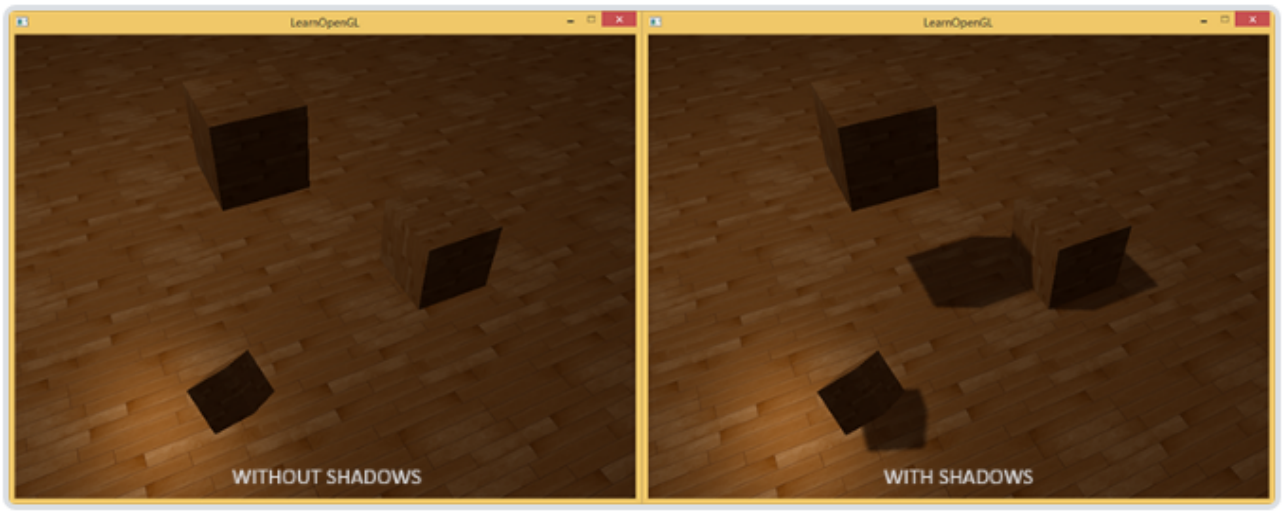
```
// renders the 3D scene
void renderScene(const Shader &shader)
{
    // floor
    glm::mat4 model = glm::mat4(1.0f);
    shader.setMat4("model", model);
    glBindVertexArray(planeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    // cubes
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(0.0f, 1.5f, 0.0));
    model = glm::scale(model, glm::vec3(0.5f));
    shader.setMat4("model", model);
    renderCube();
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(2.0f, 0.0f, 1.0));
    model = glm::scale(model, glm::vec3(0.5f));
    shader.setMat4("model", model);
    renderCube();
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(-1.0f, 0.0f, 2.0));
    model = glm::rotate(model, glm::radians(60.0f), glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
    model = glm::scale(model, glm::vec3(0.25));
    shader.setMat4("model", model);
    renderCube();
}
```

这里用到的贴图如下所示：

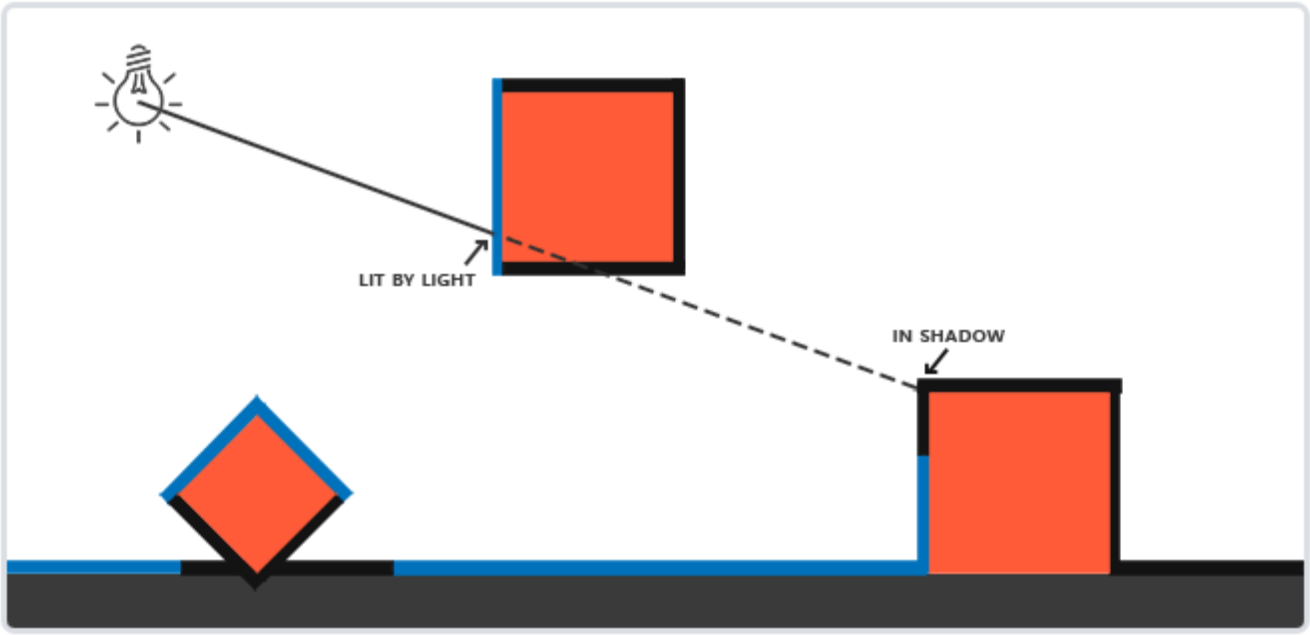


使用了贴图相当于物体材质的颜色就是贴图的颜色了。

阴影是光线被阻挡的结果；当一个光源的光线由于其他物体的阻挡不能够达到一个物体的表面的时候，那么这个物体就在阴影中了。阴影能够使场景看起来真实得多，并且可以让观察者获得物体之间的空间位置关系。场景和物体的深度感因此能够得到极大提升，下图展示了有阴影和没有阴影的情况下的不同：



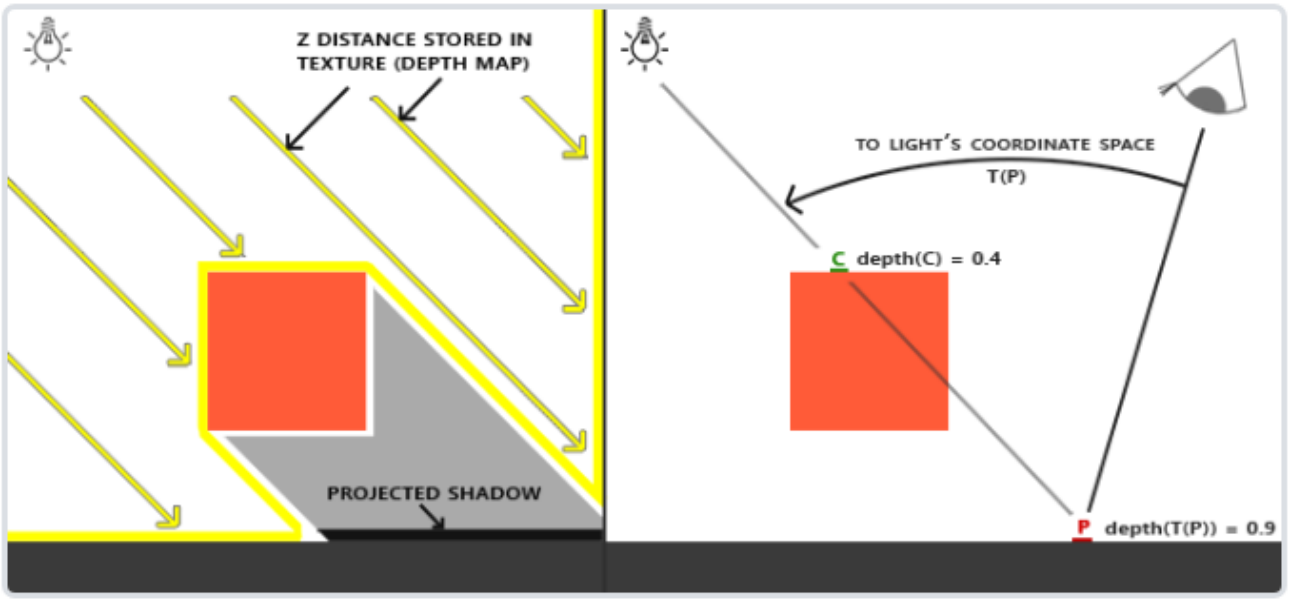

阴影映射(Shadow Mapping)背后的思路非常简单：我们以光的位置为视角进行渲染，我们能看到的东西都将被点亮，看不见的一定是在阴影之中了。假设有一个地板，在光源和它之间有一个大盒子。由于光源处向光线方向看去，可以看到这个盒子，但看不到地板的一部分，这部分就应该在阴影中了。




这里的所有蓝线代表光源可以看到的fragment。黑线代表被遮挡的fragment：它们应该渲染为带阴影的。如果我们绘制一条从光源出发，到达最右边盒子上的一个片元上的线段或射线，那么射线将先击中悬浮的盒子，随后才会到达最右侧的盒子。结果就是悬浮的盒子被照亮，而最右侧的盒子将处于阴影之中。

我们希望得到射线第一次击中的那个物体，然后用这个最近点和射线上其他点进行对比。然后我们将测试一下看看射线上的其他点是否比最近点更远，如果是的话，这个点就在阴影中。对从光源发出的射线上的成千上万个点进行遍历是个极端消耗性能的举措，实时渲染上基本不可取。我们可以采取相似举措，不用投射出光的射线。我们所使用的是非常熟悉的东西：深度缓冲。

深度缓冲里的一个值是摄像机视角下，对应于一个片元的一个0到1之间的深度值。如果我们从光源的透视图来渲染场景，并把深度值的结果储存到纹理中会怎样？通过这种方式，我们就能对光源的透视图所见的最近的深度值进行采样。最终，深度值就会显示从光源的透视图下见到的第一个片元了。我们管储存在纹理中的所有这些深度值，叫做深度贴图（depth map）或阴影贴图。




左侧的图片展示了一个定向光源（所有光线都是平行的）在立方体下的表面投射的阴影。通过储存到深度贴图中的深度值，我们就能找到最近点，用以决定片元是否在阴影中。我们使用一个来自光源的视图和投影矩阵来渲染场景就能创建一个深度贴图。这个投影和视图矩阵结合在一起成为一个TT变换，它可以将任何三维位置转变到光源的可见坐标空间。

> 定向光并没有位置，因为它被规定为无穷远。然而，为了实现阴影贴图，我们得从一个光的透视图渲染场景，这样就得在光的方向的某一点上渲染场景。

在右边的图中我们显示出同样的平行光和观察者。我们渲染一个点P处的片元，需要决定它是否在阴影中。我们先得使用T把P变换到光源的坐标空间里。既然点P是从光的透视图中看到的，它的z坐标就对应于它的深度，例子中这个值是0.9。使用点P在光源的坐标空间的坐标，我们可以索引深度贴图，来获得从光的视角中最近的可见深度，结果是点C，最近的深度是0.4。因为索引深度贴图的结果是一个小于点P的深度，我们可以断定P被挡住了，它在阴影中了。

深度映射由两个步骤组成：首先，我们渲染深度贴图，然后我们像往常一样渲染场景，使用生成的深度贴图来计算片元是否在阴影之中。

## 深度贴图

第一步我们需要生成一张深度贴图(Depth Map)。深度贴图是从光的透视图里渲染的深度纹理，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，我们将再次需要帧缓冲。

首先，我们要为渲染的深度贴图创建一个帧缓冲对象：

```
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

然后，创建一个2D纹理，提供给帧缓冲的深度缓冲使用：

```
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;

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);
```

生成深度贴图不太复杂。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。

把我们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);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

我们需要的只是在从光的透视图下渲染场景的时候深度信息，所以颜色缓冲没有用。然而帧缓冲对象不是完全不包含颜色缓冲的，所以我们需要显式告诉OpenGL我们不适用任何颜色数据进行渲染。我们通过将调用glDrawBuffer和glReadBuffer把读和绘制缓冲设置为GL_NONE来做这件事。

合理配置将深度值渲染到纹理的帧缓冲后，我们就可以开始第一步了：生成深度贴图。两个步骤的完整的渲染阶段，看起来有点像这样：

```
// 1. 首选渲染深度贴图
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    ConfigureShaderAndMatrices();
    RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 2. 像往常一样渲染场景，但这次使用深度贴图
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene();
```

这段代码隐去了一些细节，但它表达了阴影映射的基本思路。这里一定要记得调用glViewport。因为阴影贴图经常和我们原来渲染的场景（通常是窗口解析度）有着不同的解析度，我们需要改变视口（viewport）的参数以适应阴影贴图的尺寸。如果我们忘了更新视口参数，最后的深度贴图要么太小要么就不完整。

## 光源空间的变换

前面那段代码中一个不清楚的函数是`ConfigureShaderAndMatrices`。它是用来在第二个步骤确保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。然而，第一个步骤中，我们从光的位置的视野下使用了不同的投影和视图矩阵来渲染的场景。

因为我们使用的是一个所有光线都平行的定向光。出于这个原因，我们将为光源使用正交投影矩阵，透视图将没有任何变形：

```
GLfloat near_plane = 1.0f, far_plane = 7.5f;
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane,
far_plane);
```

为了创建一个视图矩阵来变换每个物体，把它们变换到从光源视角可见的空间中，我们将使用glm::lookAt函数；这次从光源的位置看向场景中央。

```
glm::mat4 lightView = glm::lookAt(glm::vec(-2.0f, 4.0f, -1.0f), glm::vec3(0.0f),
glm::vec3(1.0));
```

二者相结合为我们提供了一个光空间的变换矩阵，它将每个世界空间坐标变换到光源处所见到的那个空间；这正是我们渲染深度贴图所需要的。

```
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
```

这个lightSpaceMatrix正是前面我们称为TT的那个变换矩阵。有了lightSpaceMatrix只要给shader提供光空间的投影和视图矩阵，我们就能像往常那样渲染场景了。然而，我们只关心深度值，并非所有片元计算都在我们的着色器中进行。为了提升性能，我们将使用一个与之不同但更为简单的着色器来渲染出深度贴图。

## 渲染至深度贴图

当我们以光的透视图进行场景渲染的时候，我们会用一个比较简单的着色器，这个着色器除了把顶点变换到光空间以外，不会做得更多了。这个简单的着色器叫做`simpleDepthShader`，就是使用下面的这个着色器：

```
#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);
}
```

这个顶点着色器将一个单独模型的一个顶点，使用lightSpaceMatrix变换到光空间中。

由于我们没有颜色缓冲，最后的片元不需要任何处理，所以我们可以简单地使用一个空像素着色器：

```
#version 330 core

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

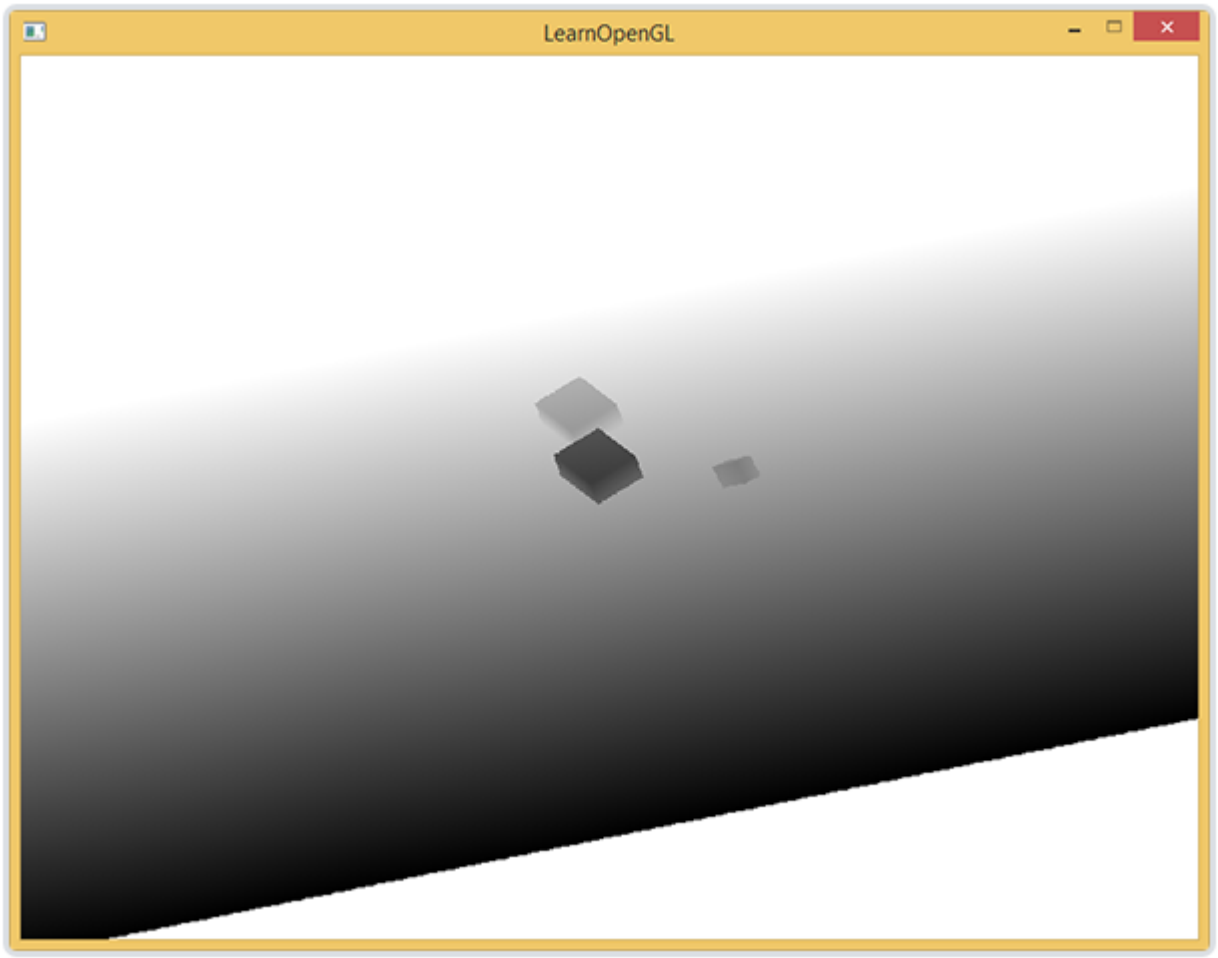
这个空像素着色器什么也不干，运行完后，深度缓冲会被更新。我们可以取消那行的注释，来显式设置深度，但是这个（指注释掉那行之后）是更有效率的，因为底层无论如何都会默认去设置深度缓冲。

渲染深度缓冲现在成了：

```
simpleDepthShader.Use();
glUniformMatrix4fv(lightSpaceMatrixLocation, 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    RenderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

这里的RenderScene函数的参数是一个着色器程序（shader program），它调用所有相关的绘制函数，并在需要的地方设置相应的模型矩阵。

最后，在光的透视图视角下，很完美地用每个可见片元的最近深度填充了深度缓冲。通过将这个纹理投射到一个2D四边形上（和我们在帧缓冲一节做的后处理过程类似），就能在屏幕上显示出来，我们会获得这样的东西：




将深度贴图渲染到四边形上的像素着色器：

```
#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);
}
```

要注意的是当用透视投影矩阵取代正交投影矩阵来显示深度时，有一些轻微的改动，因为使用透视投影时，深度是非线性的。

## 渲染阴影

正确地生成深度贴图以后我们就可以开始生成阴影了。这段代码在像素着色器中执行，用来检验一个片元是否在阴影之中，不过我们在顶点着色器中进行光空间的变换：

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

```
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec2 TexCoords;
out vec3 FragPos;
out vec3 Normal;
out vec4 FragPosLightSpace;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

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

这儿的新的地方是FragPosLightSpace这个输出向量。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FragPos和一个光空间的FragPosLightSpace给像素着色器。

像素着色器使用Blinn-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。

```
#version 330 core
out vec4 FragColor;

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;
in vec4 FragPosLightSpace;


uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

float ShadowCalculation(vec4 fragPosLightSpace)
{
   [...]
}

void main()
{
    vec3 color = texture(diffuseTexture, TexCoords).rgb;
```

```
    vec3 normal = normalize(Normal);
    vec3 lightColor = vec3(0.3);
    // ambient
    vec3 ambient = 0.3 * color;
    // diffuse
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // calculate shadow
    float shadow = ShadowCalculation(FragPosLightSpace);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

    FragColor = vec4(lighting, 1.0);
}
```

像素着色器大部分是从高级光照教程中复制过来，只不过加上了个阴影计算。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。

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：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    [...]
}
```

返回了片元在光空间的-1到1的范围。

> 当使用正交投影矩阵，顶点w元素仍保持不变，所以这一步实际上毫无意义。可是，当使用透视投影的时候就是必须的了，所以为了保证在两种投影矩阵下都有效就得留着这行。

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： （译者注：这里的意思是，上面的projCoords的xyz分量都是[-1,1]（下面会指出这对于远平面之类的点才成立），而为了和深度贴图的深度相比较，z分量需要变换到[0,1]；为了作为从深度贴图中采样的坐标，xy分量也需要变换到[0,1]。所以整个projCoords向量都需要变换到[0,1]范围。）

```
projCoords = projCoords * 0.5 + 0.5;
```

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：

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

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。

```
float currentDepth = projCoords.z;
```

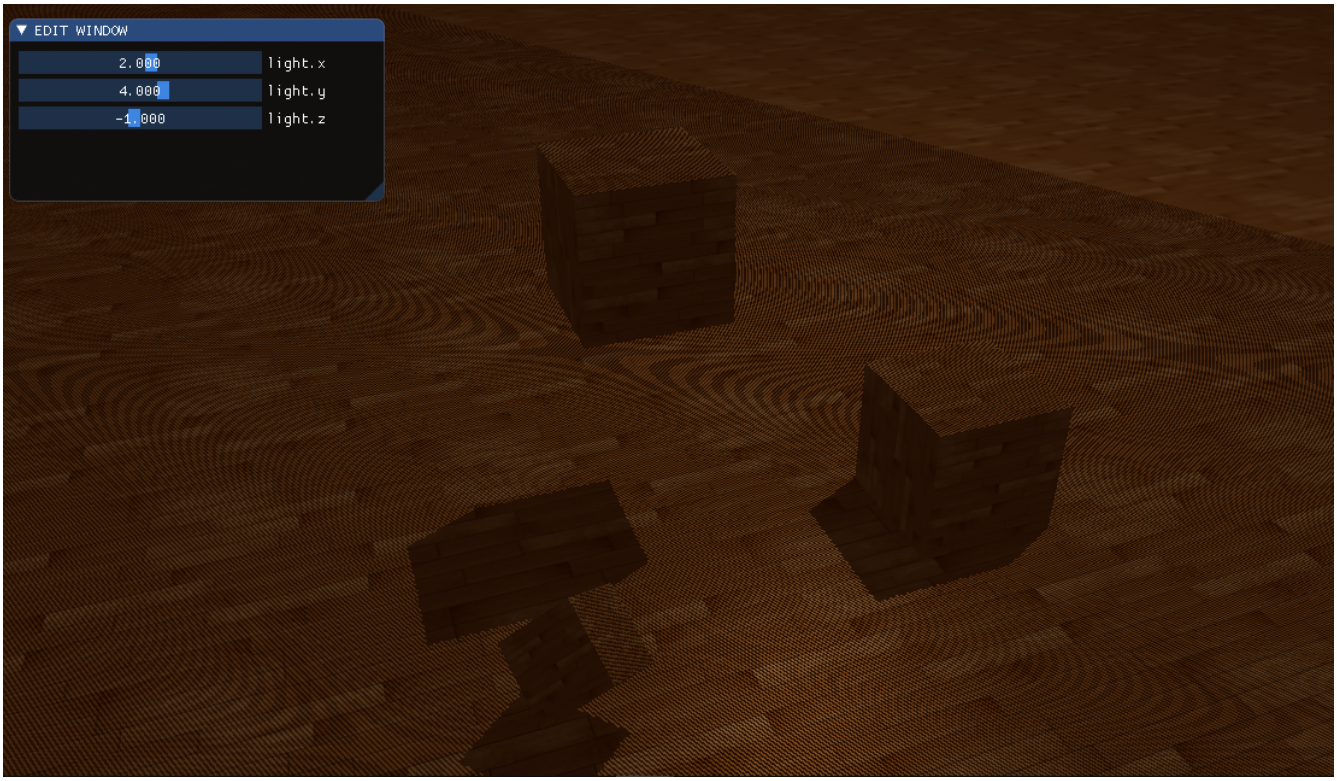
实际的对比就是简单检查currentDepth是否高于closetDepth，如果是，那么片元就在阴影中。

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

完整的shadowCalculation函数是这样的：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 变换到[0,1]的范围
    projCoords = projCoords * 0.5 + 0.5;
    // 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;
    // 取得当前片元在光源视角下的深度
    float currentDepth = projCoords.z;
    // 检查当前片元是否在阴影中
    float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;

    return shadow;
}
```

激活这个着色器，绑定合适的纹理，激活第二个渲染阶段默认的投影以及视图矩阵，结果如下图所示：




这里可以通过gui改变光源的位置。