Homework 7 - Shadowing Mapping

Basic:

1. 实现方向光源的Shadowing Mapping:
    - 要求场景中至少有一个object和一块平面(用于显示shadow)
    - 光源的投影方式任选其一即可
    - 在报告里结合代码，解释Shadowing Mapping算法

Shadow Mapping

阴影渲染的两大基本步骤：以光源视角得到深度；以camera视角判断是否为阴影。

- **前期准备**：深度图

  要首先准备一个存储深度图的texture：

```
//存储的buffer
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
const GLuint SHADOW_WINDTH = 1024, SHADOW_HEIGHT = 1024;
//纹理深度图
GLuint depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WINDTH, SHADOW_HEIGHT, 0,
GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
//环绕方式：镜像重复
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
//纹理过滤：放大缩小时使用最近邻
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap,
0);
//对于深度图来说，颜色缓冲是没有用的，设置读和绘制缓冲为GL_NONE
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
//默认设置
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- **第一次渲染**，得到深度图

  顶点着色器：

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

uniform mat4 lightSpaceMatrix;  //光源观察矩阵
uniform mat4 model; //model矩阵

void main()
{
    //将视点设置到光源位置
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}
```

片段着色器：

```
#version 330 core
void main()
{
    //什么都不做，默认会设置深度缓冲
}
```

渲染：

```
//采用正交投影
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane,
far_plane);
//得到光源的观察矩阵
glm::mat4 lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0f, 1.0f,
0.0f));
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
simpleDepthShader.use();
//着色器赋值
simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
//视图尺寸不是窗口尺寸
glViewport(0, 0, SHADOW_WINDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO); //将得到的深度信息存储到depthMapFBO中
glClear(GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, myTexture);    //纹理

//渲染物体：palne和cube，二者的model矩阵是不同的
    glm::mat4 model = glm::mat4(1.0f);
    simpleDepthShader.setMat4("model", model);
    glBindVertexArray(planeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);

    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(0.0f, 0.5f, 0.0f));
    simpleDepthShader.setMat4("model", model);
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
```

```
//恢复默认值
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

这样得到的深度信息就会存储到与depthMapFBO关联的问题depthMap中去。

- **第二次渲染**，得到实际场景

  顶点着色器：

  在输出结构体中的两个视点下的坐标是为了便于与第一次渲染得到的深度图进行比较，因为深度图是光源视点下的。

```
#version 330 core
layout (location = 0) in vec3 aPos;//顶点坐标
layout (location = 1) in vec3 aNormal;//法线
layout (location = 2) in vec2 aTexCoords;//贴图坐标

//输出结构体
out VS_OUT {
    vec3 FragPos;//摄像机视点下的坐标
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;//光源视点下的坐标
} vs_out;

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

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);//坐标转换

    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);
}
```

  片段着色器：

  采用Phong Shading进行计算阴影。

```
#version 330 core
out vec4 FragColor;
//输入结构体，对应顶点着色器
in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;
```

```
uniform sampler2D diffuseTexture;//贴图
uniform sampler2D shadowMap;//深度图

uniform vec3 lightColor;//光
uniform vec3 lightPos;//光的位置
uniform vec3 viewPos;//摄像机位置

//可调的变量
uniform float ambientStrength;
uniform float specularStrength;
uniform float diffuseStrength;
uniform int specN;

//判断当前片元是否在阴影下
float ShadowCalculation(vec4 fragPosLightSpace)
{
    //转换到[-1, 1]
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    //转换到[0,1],便于和深度比较
    projCoords = projCoords * 0.5 + 0.5;
    //得到最近深度
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    //片元的当前深度
    float currentDepth = projCoords.z;
    //片元是否在阴影中
    float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;

    return shadow;
}

void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;//贴图
    vec3 norm = normalize(fs_in.Normal);
    //ambient
    vec3 ambient = ambientStrength * lightColor;
    //fidduse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor * diffuseStrength;
    //specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 halfwayDir = normalize(lightDir + viewDir);
    float spec = pow(max(dot(norm, halfwayDir), 0.0), specN);
    vec3 specular = specularStrength * spec * lightColor;
    //阴影
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    //1-shadow表示是否需要加入diffuse和specular量
    vec3 result = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

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

渲染：

```
//视图尺寸是窗口尺寸
glViewport(0, 0, view_width, view_height);
//这里是个坑
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

shader.use();
//摄像机观察矩阵
glm::mat4 projection = glm::perspective(45.0f, (float)width / (float)height, 0.1f,
100.0f);
glm::mat4 view = camera.getView();
//着色器赋值
shader.setMat4("projection", projection);
shader.setMat4("view", view);
shader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
shader.setVec3("lightPos", lightPos);
shader.setVec3("viewPos", camera.getPos());
shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
shader.setFloat("ambientStrength", ambientStrength);
shader.setFloat("specularStrength", specularStrength);
shader.setFloat("diffuseStrength", diffuseStrength);
shader.setInt("specN", specN);
//纹理贴图
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, myTexture);
//深度图
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);

//渲染物体：palne和cube，注意model要和第一次渲染保持一致
        model = glm::mat4(1.0f);
        shader.setMat4("model", model);
        glBindVertexArray(planeVAO);
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindVertexArray(0);

        model = glm::mat4(1.0f);
        model = glm::translate(model, glm::vec3(0.0f, 0.5f, 0.0f));
        shader.setMat4("model", model);
        glBindVertexArray(cubeVAO);
        glDrawArrays(GL_TRIANGLES, 0, 36);
        glBindVertexArray(0);
```

- 坑

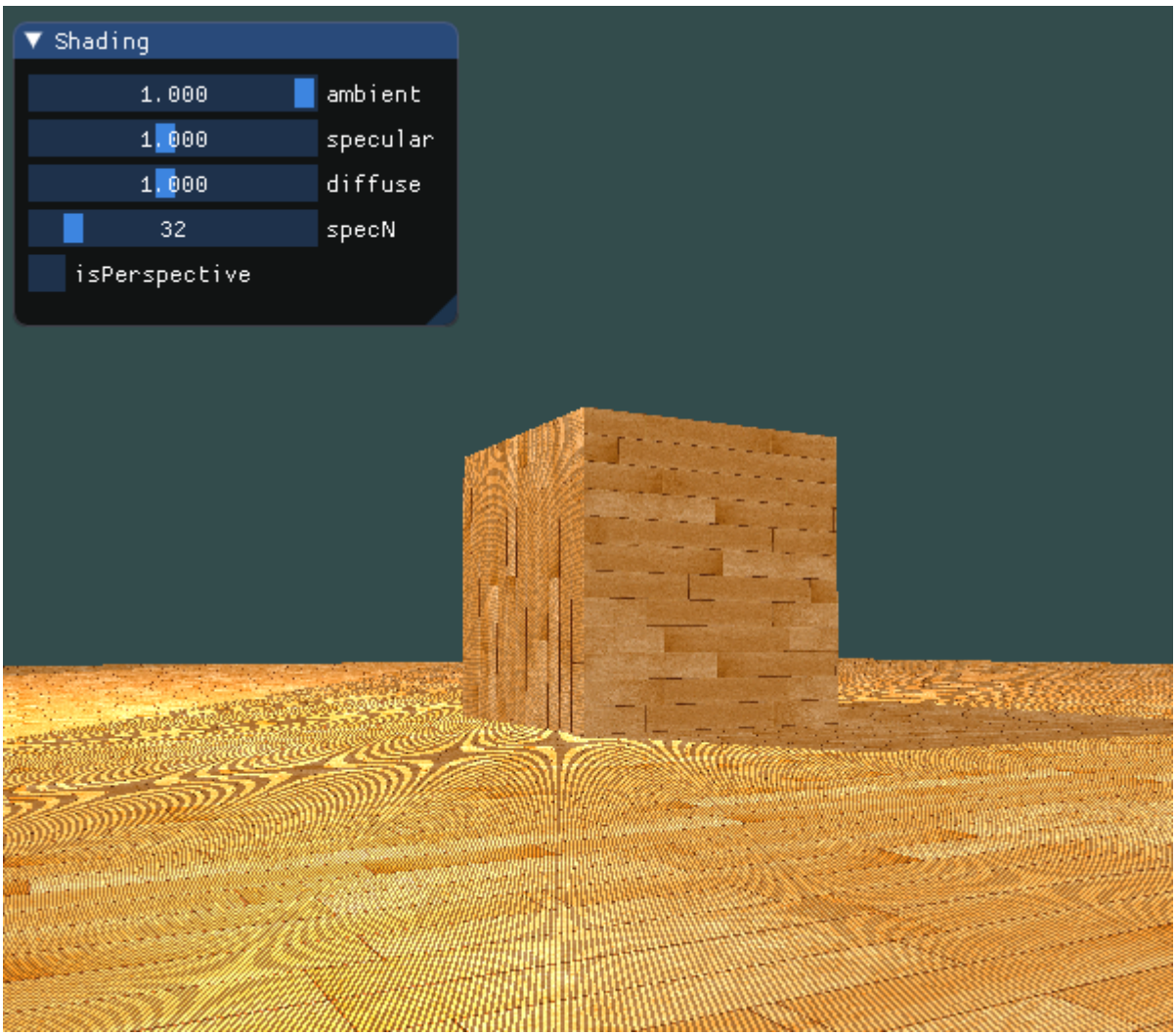
  这里有个坑，注意在第一次渲染之后记得清空深度缓冲：

  ```
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  ```

  如果不清空，窗口里就什么都不会显示，我就是一直不知道为什么窗口里只有背景和GUI，就是没有物体，大费周章检查了各种地方，结果是写成了：

```
glClear(GL_COLOR_BUFFER_BIT) //没有清空depth_buffer
```

效果



2. 修改GUI

在上面的第二次渲染的着色器中可以看到与Phong Shading有关的变量，是可以通过GUI调节的，和上次作业类似。

效果见上图

Bonus

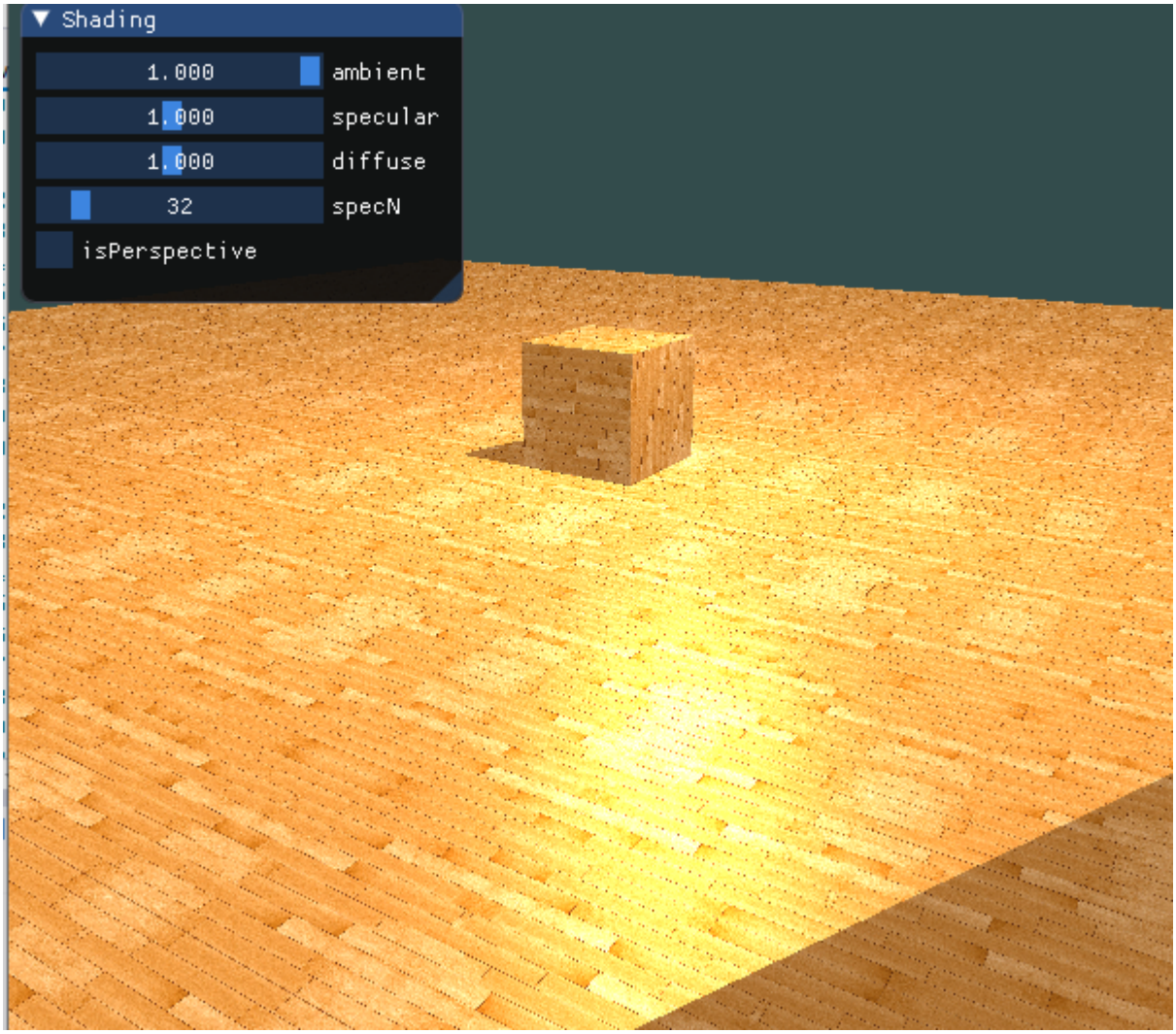
1. 优化Shadowing Mapping

去除条纹

贴图上的条纹是因为光线与表面呈夹角地采样时，有些就会在表面上面，有些在表面下面，因此采用一个偏移量让采样面比之前浅一点，保证不会在表面之下。偏移量的决定最好根据入射光防线来决定合适的值：

```
//在ShadowCalculation函数中
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);//计算偏移量
float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;//比较深度
```

效果：

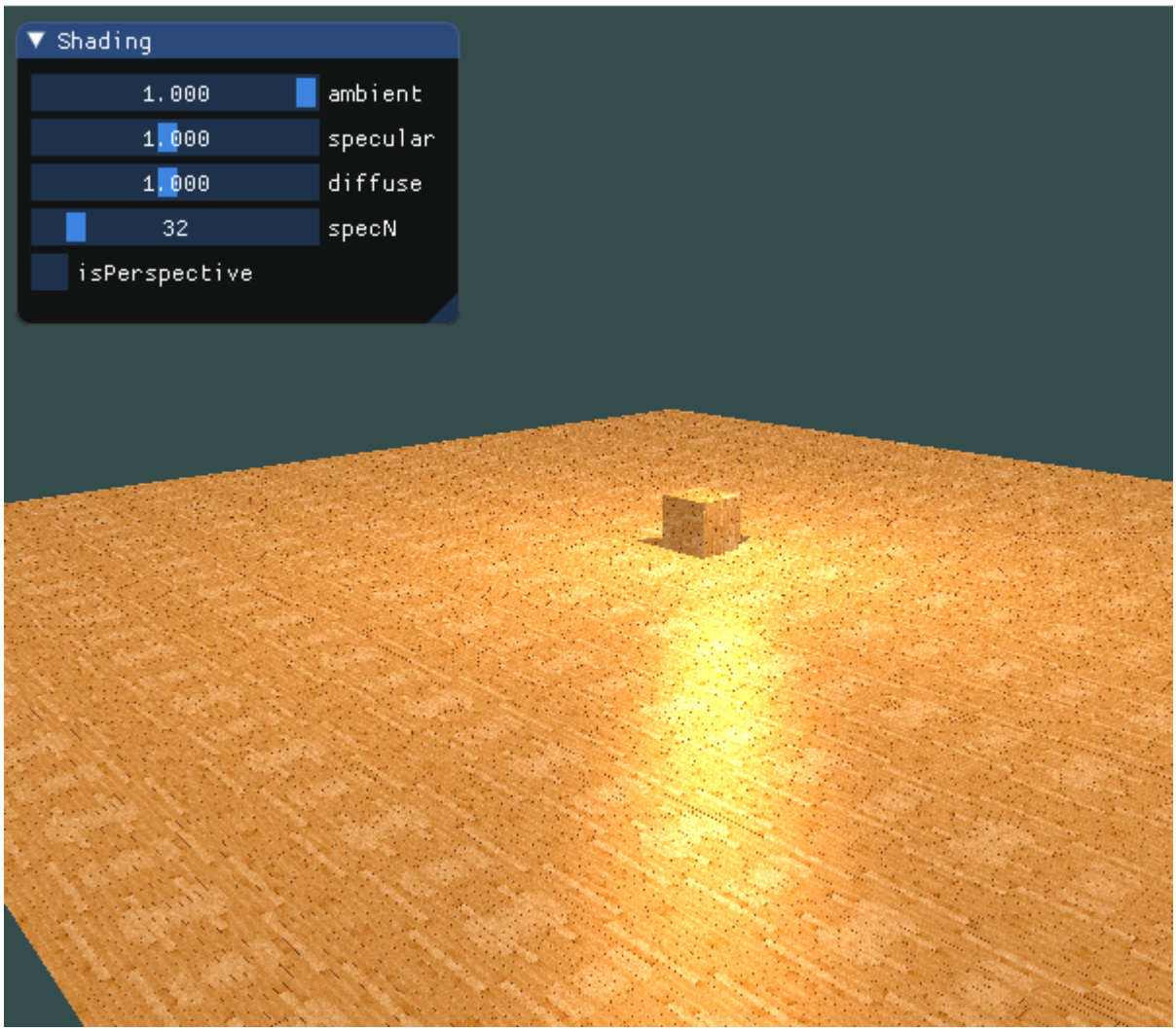


去除远处的阴影

可以看到平面的一部分在阴影中，这是因为当平面坐标超出了光的正交视锥的远平面，即使没有任何东西也会被判定为处在阴影中，由于它们的投影坐标的z坐标大于1.0，因此只要判断后返回false即可：

```
//在ShadowCalculation函数中
if(projCoords.z > 1.0)
        shadow = 0.0;
```

效果：



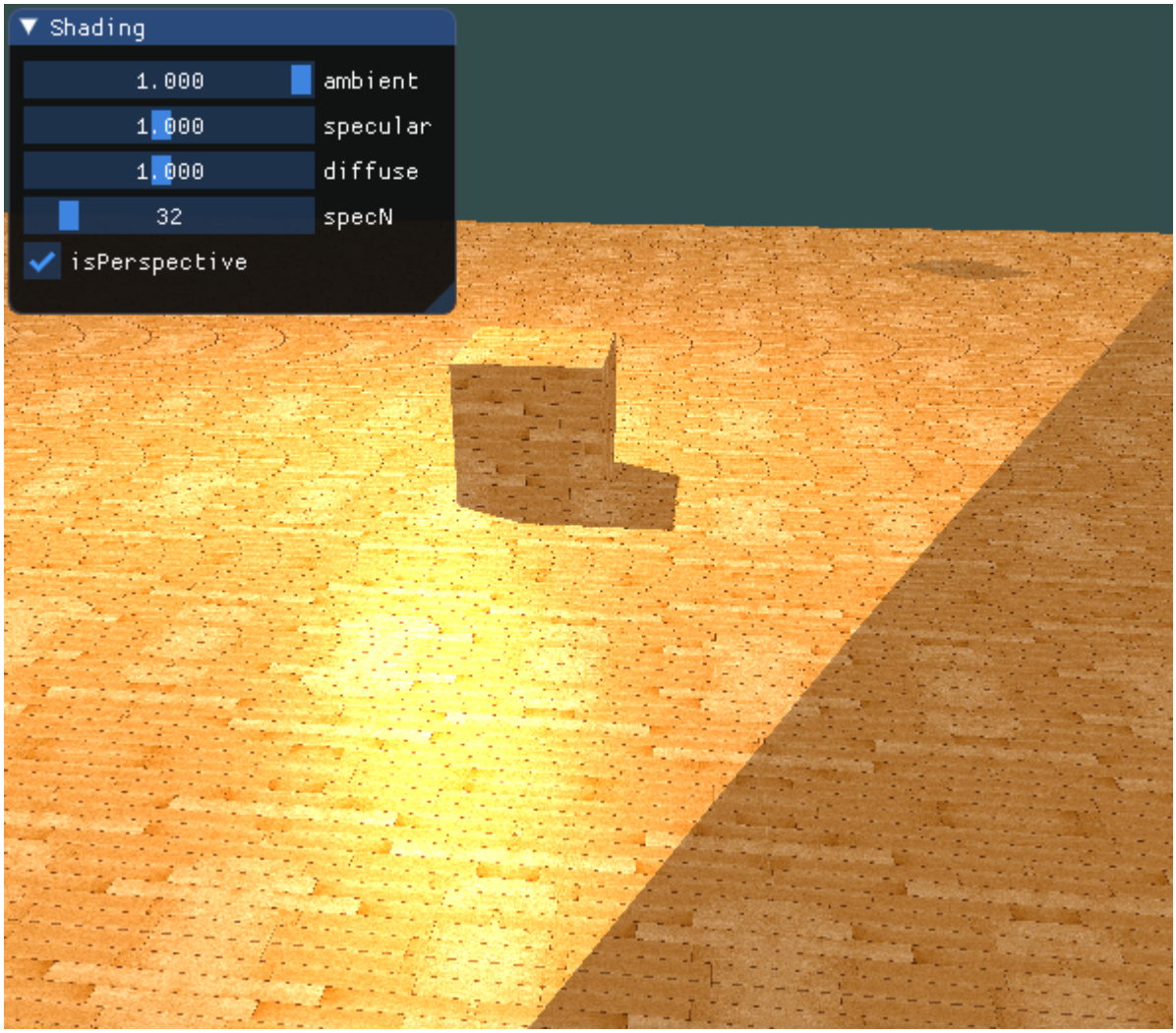
可以看到远处没有了阴影

2. 实现光源在正交/透视两种投影下的Shadowing Mapping

光源的正交/透视影响着第一次渲染时光源的观察矩阵：

```
if (isPerspective)
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
else
    lightProjection = glm::perspective(80.0f, (float)SHADOW_WINDTH / (float)SHADOW_HEIGHT,
0.1f, 150.0f);
```

透视效果（此时还没有去除远处阴影，正交效果见上上图）：


Shading
1.000
ambient
1.000
specular
1.000
diffuse
32
specN
isPerspective