计算机图形学实验报告

学号：16340054

姓名：戴馨乐

学院：数据科学与计算机学院

作业：Homework7

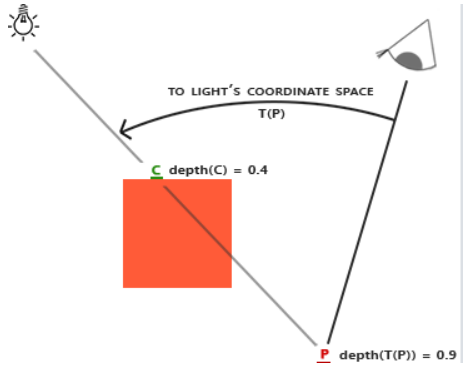
Basic:

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
2. 修改 GUI

阴影映射 Shadow Mapping 的思路：

有一个光源和一些盒子，要从光源处沿着光线的方向能看到的就是亮的，看不到的就是在阴影处。那么找到光线到达的第一个物体，后面的自然都是在阴影下的。假如采用遍历很耗时，那么可以借用深度贴图:

平时透视图都是从观察者的角度出发的，这里从光源的透视图出发，来渲染这个场景，可以得到如下这种透视图:



从光源透视图的角度看出去，P 点的深度大于 C 点，故 P 点应该在阴影中。用这种方式渲染出来的，就是深度贴图。

得到了深度贴图，将其存储为一个纹理，最后和绑定正常纹理那样，绑定到正常渲染的场景上就可以得到了带阴影的场景了。

阴影映射 Shadow Mapping 的详细步骤：

1. 生成深度贴图
   a) 深度贴图是从光的透视图出发，渲染的深度纹理，要将渲染的场景保存成纹理，需要用到帧缓冲：
      i. 生成帧缓冲对象

```
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

      ii. 创建深度贴图的纹理对象，并设置相关的属性。其中，深度贴图的解析度SHADOW_WIDTH,SHADOW_HEIGHT)不同于窗口大小，需要另外设置，这里设置解析度为1024×1024

```
const int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
// 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);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
```

      iii. 将深度贴图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);
```

      这里，深度缓冲没有颜色缓冲，关注的只有深度缓冲，所以设置如下来告诉 OpenGL 这里不需要对任何颜色数据进行渲染

      glDrawBuffer(GL_NONE);

      glReadBuffer(GL_NONE);

   b) 在配置好帧缓冲之后，下一步需要在渲染循环中生成深度贴图。

```
glm::mat4 lightProjection(1.0f);
glm::mat4 lightView(1.0f);
float near_plane = 1.0f, far_plane = 7.5f;
if (ortho)
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
else if (proj)
    lightProjection = glm::perspective(glm::radians(45.0f), (float)width / height, near_plane, far_plane);
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
depthShader.use();
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
// 1. render depth scene
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, woodTexture);
    render_scene(depthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

   可以看到，整个渲染过程分为：

      1) 改变视口大小并清理深度缓冲位

```
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glClear(GL_DEPTH_BUFFER_BIT);
```

2) 因为要从光源的角度来看物体以生成深度缓冲，所以要使用从光源角度出发的透视矩阵lightProjection和视角矩阵lightView，最后可以得到将世界空间转换到光源空间的矩阵lightSpaceMatrix。这里默认使用的是正交投影。

```
glm::mat4 lightProjection(1.0f);
glm::mat4 lightView(1.0f);
float near_plane = 1.0f, far_plane = 7.5f;
if (ortho)
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
else if (proj)
    lightProjection = glm::perspective(glm::radians(45.0f), (float)width / height, near_plane, far_plane);
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
depthShader.use();
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
```

3) 最后，绑定帧缓冲，然后渲染场景，就可以得到深度贴图纹理

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, woodTexture);
    glCullFace(GL_FRONT);
    render_scene(depthShader);
    glCullFace(GL_BACK);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

其中，生成深度贴图所用的 Shader：

```
Shader depthShader("./Shadow_Depth.vs", "./Shadow_Depth.fs");
```

其中顶点着色器直接简单地将坐标转换空间，片段着色器由于没有需要颜色缓冲，所以什么事也不用做。

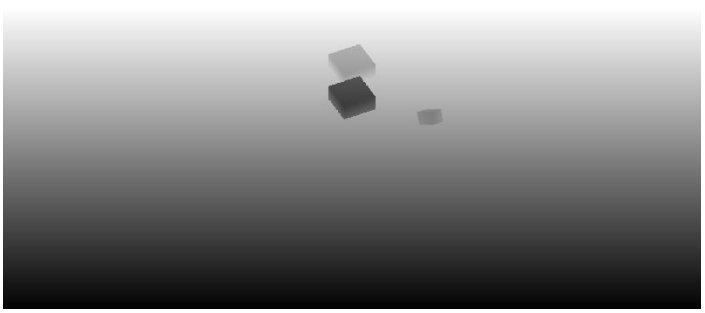
顶点着色器：

```
#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);
}
```

片段着色器：

```
#version 330 core

void main()
{

}
```

将深度贴图可视化之后：



可以看到，这里得到的就是从光源角度看下去的物体，只是没有了颜色缓冲仅仅有深度缓冲，这可以用来计算阴影。

2. 将深度贴图绑定到正常场景中

使用渲染场景的 Shader 来渲染：

```
Shader shader("./Shadow_Mapping.vs", "./Shadow_Mapping.fs");
```

```
// 2. render scene
shader.use();
glm::mat4 projection = glm::perspective(glm::radians(camera.getfov()), (float)width / (float)height, 0.1f, 100.0f);
glm::mat4 view = camera.getViewMatrix();
shader.setMat4("projection", projection);
shader.setMat4("view", view);
shader.setVec3("viewPos", camera.getCameraPos());
shader.setVec3("lightPos", lightPos);
shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, woodTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);
render_scene(shader);
```

这一段代码，就是简单地从摄像机的视角，来渲染物体，特别的是：

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);
```

将深度贴图作为纹理绑定在场景上，相当于给原先的场景加上了阴影的效果。

要成功将阴影加上，还需要在顶点着色器和片段着色器上进行修改。

顶点着色负责：

1. 计算顶点的坐标

```
gl_Position = projection * view * model * vec4(aPos, 1.0f);
```

2. 为了计算光照，需要计算片段的位置，法向量

```
vs_out.FragPos = vec3(model * vec4(aPos, 1.0f));
vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;
```

3. 为了加上纹理，需要传入纹理到片段着色器

```
vs_out.TexCoords = aTexCoords;
```

4. 为了计算阴影，将片段的坐标进行光源空间的转换，因为深度贴图中阴影的计算就是在光源空间中的。

```
vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0f);
```

完整的代码如下：

```
#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 VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} vs_out;

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

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

片段着色器负责：

1. 根据纹理计算物体的颜色

```
vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
```

2. 计算光照

```
// 环境光照
vec3 ambient = 0.3 * color;
// 漫反射
vec3 lightDir = normalize(lightPos - fs_in.FragPos);
float diff = max(dot(lightDir, norm), 0.0f);
vec3 diffuse = diff * lightColor;
// 镜面反射
vec3 viewDir = normalize(viewPos - fs_in.FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = 0.0;
vec3 halfwayDir = normalize(lightDir + viewDir);
spec = pow(max(dot(norm, halfwayDir), 0.0f), 64.0);
vec3 specular = spec * lightColor;
```

3. 计算阴影

```
// 计算阴影
float shadow = ShadowCaculation(fs_in.FragPosLightSpace);
```

其中，阴影的计算想法很简单，就是计算光线所到达的最近深度以及当前这个点的深度，假如当前这个点的深度大，那么就是阴影，设为 1.0；否则就是照亮的，为 0.0。

```
float ShadowCaculation(vec4 fragPosLightSpace) {
    vec3 projCoords = fragPosLightSpace.xyz/ fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
    float closesDepth = texture(shadowMap, projCoords.xy).r;
    float currentDepth = projCoords.z;
    float shadow = currentDepth > closesDepth? 1.0: 0.0;
    return shadow;
}
```

4. 综合光照和阴影得到最终的着色结果

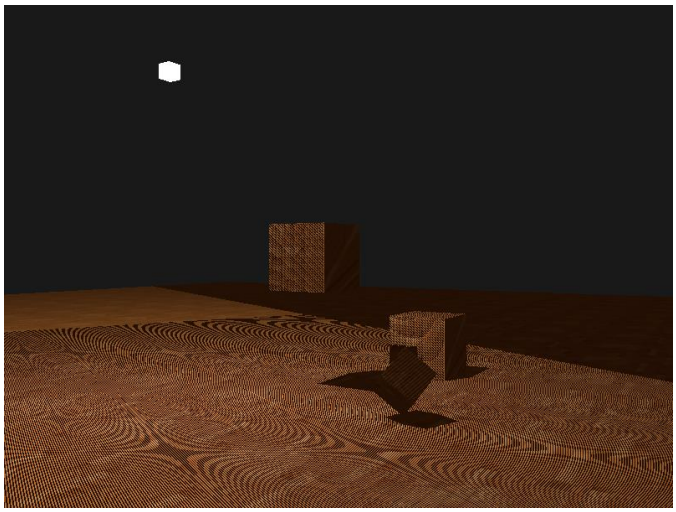
```
vec3 result = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
FragColor = vec4(result, 1.0f);
```

完整代码如下：

```
#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;

float ShadowCaculation(vec4 fragPosLightSpace) {
    vec3 projCoords = fragPosLightSpace.xyz/ fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
    float closesDepth = texture(shadowMap, projCoords.xy).r;
    float currentDepth = projCoords.z;
    float shadow = currentDepth > closesDepth? 1.0: 0.0;
    return shadow;
}
```

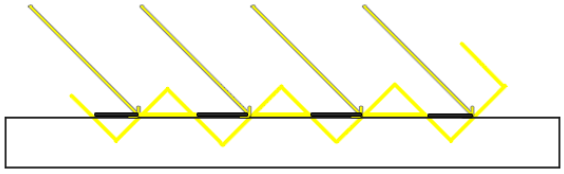
```
void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 norm = normalize(fs_in.Normal);
    vec3 lightColor = vec3(1.0f);
    // 环境光照
    vec3 ambient = 0.3 * color;
    // 漫反射
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, norm), 0.0f);
    vec3 diffuse = diff * lightColor;
    // 镜面反射
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(norm, halfwayDir), 0.0f), 64.0);
    vec3 specular = spec * lightColor;
    // 计算阴影
    float shadow = ShadowCaculation(fs_in.FragPosLightSpace);
    // 融合
    vec3 result = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
    FragColor = vec4(result, 1.0f);
}
```

运行结果：

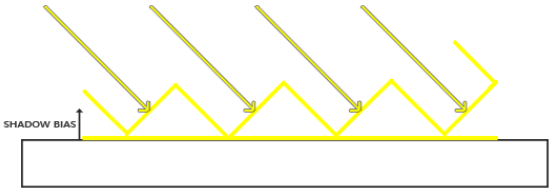


3. 解决阴影失真

这里，可以看到被光照亮的地方出现了一格格的阴影，很不真实。这是因为上述方法产生的深度贴图如下：



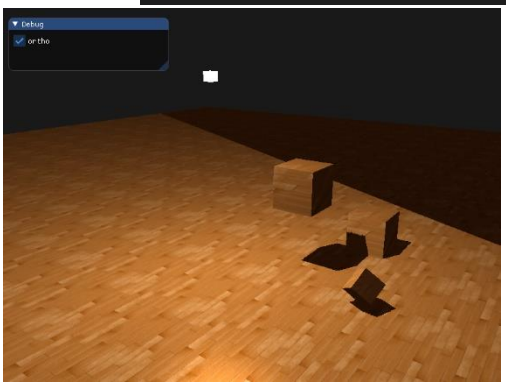
当光源远的时候，片元会从深度贴图的同一个深度值去采样，那么有些片元会被认为在平面下，有些认为在平面上，所以产生了参差交错的，不真实的阴影。解决方法很直观，将深度值整体上移就好了，如下：



那么，只需要在生成场景的 shader 的片段着色器生成阴影时候，增加一个偏移量 bias，就可以解决了。为了这个偏移值可以适应不同角度的光线，所以使用光线的朝向的角度来改变：

```
float bias = max(0.05 * (1.0 - dot(norm, lightDir)), 0.005);
float shadow = currentDepth - bias > closesDepth? 1.0: 0.0;
```

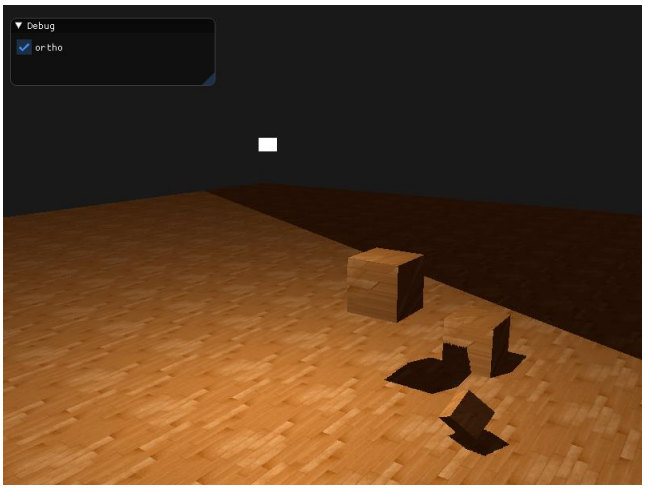
结果如图：



可以看到那种交错的不真实的阴影被去掉了。

修改 ImGUI：

这里只需要调整是使用正交投影还是透视投影，所以只增加了一个 Checkbox 来选择



代码：

```
// ImGUI
ImGui_ImplOpenGL3_NewFrame();
ImGui_ImplGlfw_NewFrame();
ImGui::NewFrame();
{
    ImGui::Checkbox("ortho", &ortho);
}
ImGui::Render();
```

另外，由于为了更自由移动摄像头，所以隐藏了鼠标，这里需要调用出来选择，又需要隐藏起来移动摄像头，所以增加了可以修改鼠标状态的按键设置，这里设置为按下 Space 键，就可以显示或者隐藏鼠标。

```
if (is_press(window, GLFW_KEY_ENTER)) {
    if (show) {
        show = false;
    }
    else {
        show = true;
    }
}
if (show) {
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
}
else {
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
}
```

Bonus：

1. 实现光源在正交/透视两种投影下的 Shadowing Mapping

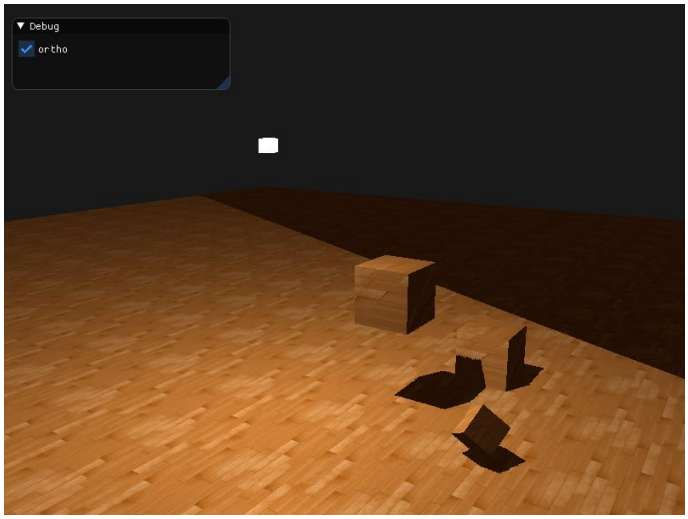
通过上面的 ImGUI 的设置来调整投影模式，如下：

```
if (ortho)
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
else
    lightProjection = glm::perspective(glm::radians(45.0f), (float)width / height, near_plane, far_plane);
```

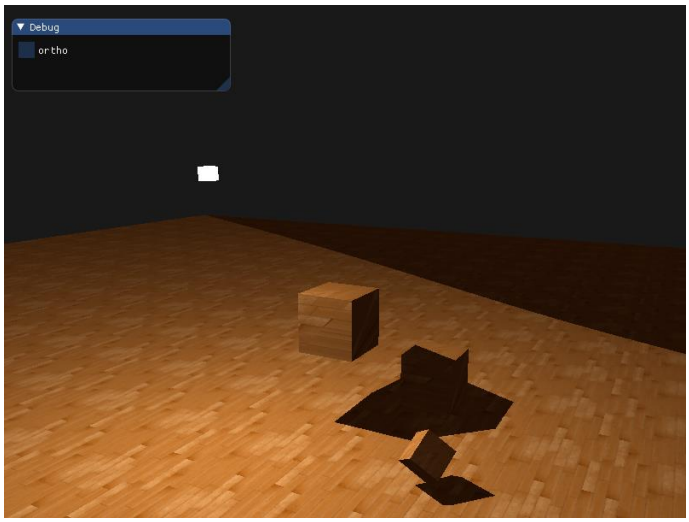
这样就可以修改了

对比如下：

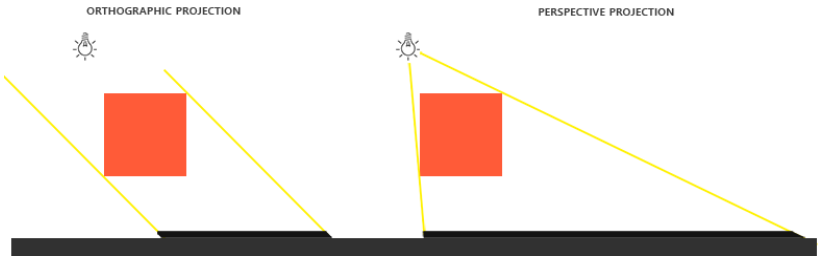
使用正交投影：



使用透视投影：



可以看到，正交投影的阴影面积比起透视投影要少。这是因为正交投影相当于平行光照射在物体上，而透视投影则是从光源出发的光线，对比如下：

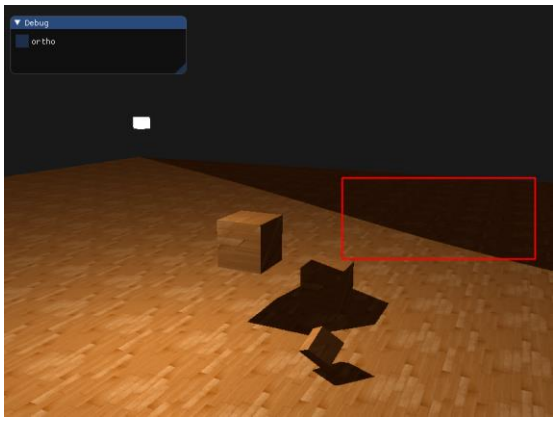


从对比图看到，这和实验结果是相符的。

2. 优化 Shadowing Mapping

a) 解决采样过多问题

从上面实验图看到：



出现了一大块的阴影面积。而我们本来是有环境光照的，所以这些地方其实是对阴影采样过多了，超出光的范围一律视为处在阴影中，不管是不是真的。这个原因是深度贴图的环绕方式是 GL_REPEAT。解决办法是让超出深度贴图的坐标的深度范围都是 1.0，这样它们就不会是被判断在阴影下了。

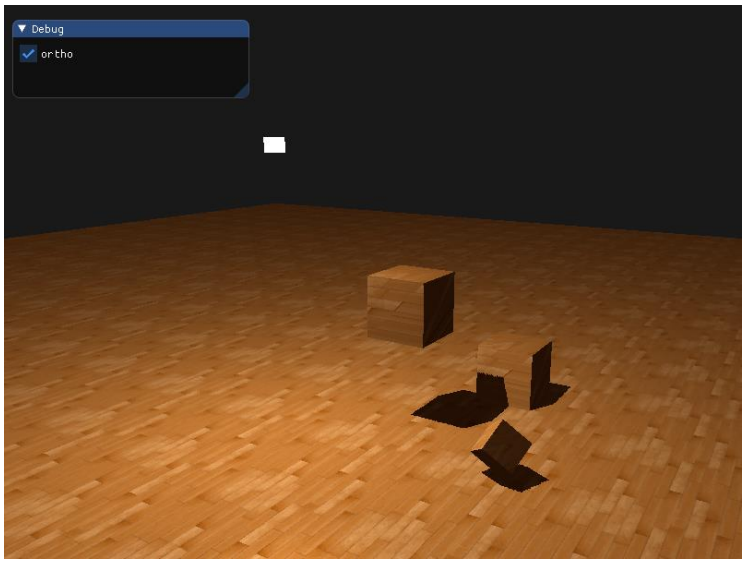
1) 设置深度贴图纹理：将模式设置为 GL_CLAMP_TO_BORDER，然后定义边框的深度 borderColor，最后将这个设置为纹理的边缘的深度。

```
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);
```

2) 修改片段着色器的阴影计算，将深度大于 1.0 的点，阴影都设置为没有阴影，即 shadow 值为 0.0

```
if (projCoords.z > 1.0)
    shadow = 0.0;
```

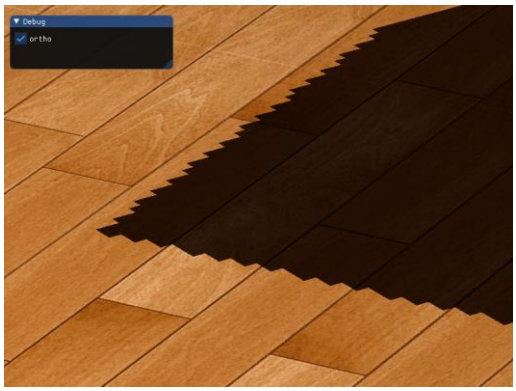
效果：



可以看到，远处的，不真实的阴影都被消除了。

b) 缓解阴影锯齿化的问题

由于图片是像素化的，而且深度贴图的解析度是固定的，那么必定会出现锯齿化，显得不真实。



虽然增加解析度可以缓解这个问题，但是通过另外一种方法 PCF 来解决会更加灵活。该方法是从深度贴图中多次采样，然后融合在一起，结果进行平均化，从而得到了柔和的阴影。这次是从一个3×3的范围中进行采样融合。如下：

```
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;
    }
}
```

效果：

可能设置范围不够，虽然还有锯齿化，但是可以看到是有对周围进行采样融合，得到了柔和的阴影。