计算机图形学 Homework 7

(一) 作业要求

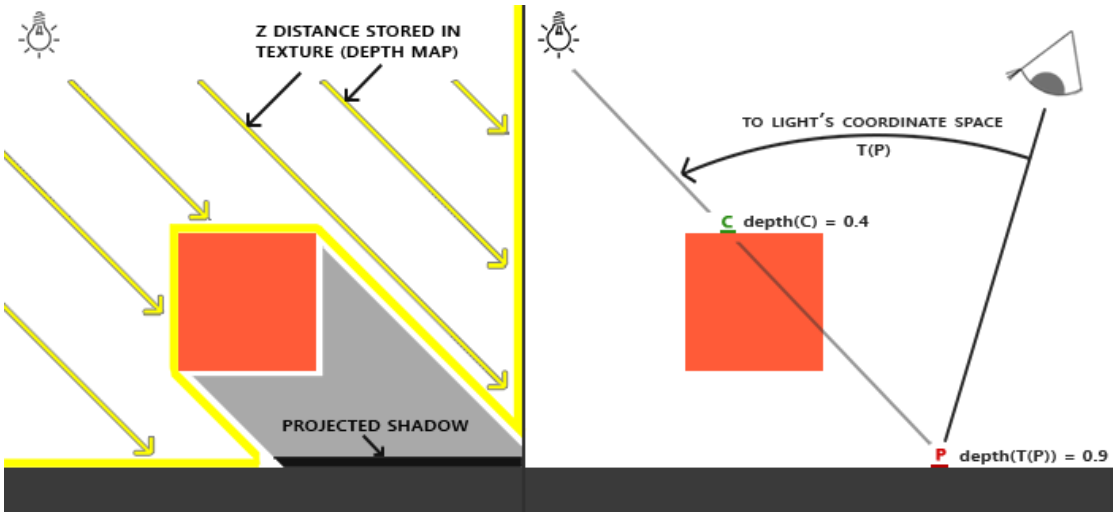
Basic:

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

Bonus:

1. 实现光源在正交/透视两种投影下的Shadowing Mapping
2. 优化Shadowing Mapping (可结合References链接，或其他方法。优化方式越多越好，在报告里说明，有加分)

(二) Shadow Mapping 算法



Shadow Mapping 算法分为两个基本步骤:

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

② 以相机视角渲染场景，比较当前深度值和在深度贴图中存储的的最近点深度值，决定某个点是否在阴影下

1. 创建平面用于显示 shadow

```
// 平面顶点数据
float planeVertices[] = {
    // 位置坐标              // 法线坐标          // 纹理坐标
     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
};
```

```
// 创建平面VAO和VBO
unsigned int planeVBO;
glGenVertexArrays(1, &planeVAO);
glGenBuffers(1, &planeVBO);

// 绑定平面VAO和VBO
glBindVertexArray(planeVAO);
glBindBuffer(GL_ARRAY_BUFFER, planeVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(planeVertices), planeVertices, GL_STATIC_DRAW);

// 设置平面顶点属性指针
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);
```

2. 加载纹理

```
unsigned int marbleTexture = loadTexture("E:/OpenGL/resources/textures/marble2.jpg");
```

loadTexture 函数:

```
// 创建纹理
unsigned int textureID;
glGenTextures(1, &textureID);

// 加载图片
int width, height, nrComponents;
unsigned char *data = stbi_load(path, &width, &height, &nrComponents, 0);


// 绑定纹理
glBindTexture(GL_TEXTURE_2D, textureID);

// 生成纹理
glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);

// 为当前绑定的纹理对象设置环绕和过滤方式
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, format == GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, format == GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

3. 生成深度贴图

a. 创建帧缓冲对象（depthMapFBO）

```
// 创建帧缓冲对象
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

b. 创建深度纹理（depthMap）

```
// 创建深度纹理
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);
float borderColor[] = {1.0, 1.0, 1.0, 1.0};
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

glTexImage2D 函数用于生成纹理。第一个参数为纹理目标。第二个参数为多级渐远纹理的级别，0 表示基本级别。第三个参数为纹理格式，这里设置为 GL_DEPTH_COMPONENT。第四个和第五个参数设置纹理的宽度和高度。第六个参数应该总是被设置为 0。第七个和第八个参数定义纹理图片的格式和数据类型，这里格式设置为 GL_DEPTH_COMPONENT，数据类型设置为 GL_FLOAT。最后一个参数为真正的纹理图片数据，这里设置为 NULL。

纹理的宽度（SHADOW_WIDTH）和高度（SHADOW_HEIGHT）设置为 1024，这是深度贴图的解析度。

c. 将深度纹理（depthMap）作为帧缓冲对象（depthMapFBO）的深度缓冲

```
// 将深度纹理作为帧缓冲对象的深度缓冲
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);
```

glFramebufferTexture2D 函数用于将纹理附加到帧缓冲。第一个参数为帧缓冲的目标。第二个参数为附件类型，这里设置为 GL_DEPTH_ATTACHMENT，表示附加深度缓冲。第三个参数为希望附加的纹理类型。第四个参数为附加的实际纹理。最后一个参数为多级渐远纹理的级别。

调用 glDrawBuffer 和 glReadBuffer 函数将绘制和读取缓冲设置为 GL_NONE，表示不使用任何颜色数据进行渲染。

4. 渲染场景至深度纹理

a. 计算投影矩阵，观察矩阵，T 矩阵

```
// 计算投影矩阵，观察矩阵，T矩阵
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
float near_plane = 1.0f, far_plane = 7.5f;
if (isOrtho) { // 正交投影
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
else { // 透视投影
    lightProjection = glm::perspective(glm::radians(45.0f), (GLfloat)SHADOW_WIDTH / (GLfloat)SHADOW_HEIGHT,
}
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
lightSpaceMatrix = lightProjection * lightView;
```

投影矩阵（lightProjection）间接决定可视区域的范围，所以必须保证投影的视锥体包含需要渲染至深度贴图的物体。glm::ortho 函数设置正交投影矩阵，前四个参数定义了近平面和远平面的大小，第五个和第六个参数定义了近平面和远平面的距离。glm::perspective 函数设置透视投影矩阵，第一个参数定义了 FOV，第二个参数定义了宽高比，第三个和第四个参数定义了近平面和远平面的距离。需要注意的是，这里的宽高比使用了深度贴图的宽度和高度。

观察矩阵（lightView）用来将顶点坐标从世界空间变换到观察空间。这里观察参考点为光源，所以 glm::lookAt 函数的第一个参数为光源位置坐标。

T 矩阵（lightSpaceMatrix）定义为投影矩阵乘以观察矩阵，它用来将顶点坐标从世界空间变换到光源处所见空间。

b. 定义简单深度着色器

顶点着色器：将顶点坐标变换到光源处所见空间

```
#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：因为没有颜色缓冲，所以片段着色器不需要做任何处理

```
#version 330 core

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

```
// 激活简单深度着色器
simpleDepthShader.use();
simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
```

c. 渲染场景

```
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, marbleTexture);
renderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

调整视口大小

激活纹理单元并绑定纹理

调用renderScene函数渲染场景

因为阴影贴图和原来的场景通常有不同的解析度，所以需要调用 glViewport 函数来调整视口大小。

一个纹理的位置值通常称为一个纹理单元（Texture Unit），纹理单元用于在着色器中使用多个纹理。首先调用 glActiveTexture 函数激活相应的纹理单元，然后调用 glBindTexture 函数绑定纹理至当前激活的纹理单元。

glActiveTexture 函数用于激活纹理单元，参数为符号常量 GL_TEXTUREi，i 的取值范围为 0 到 K-1，K 为 OpenGL 支持的最大纹理单元数量。默认情况下当前激活的纹理单元为 0。

glBindTexture 函数用于将纹理绑定到当前激活的纹理单元。

```
void renderScene(const Shader &shader) {
    // 渲染平面
    glm::mat4 model = glm::mat4(1.0f);
    shader.setMat4("model", model);
    glBindVertexArray(planeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    // 渲染立方体
    model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(0.0f, 1.0f, 0.0));
    model = glm::rotate(model, glm::radians(30.0f), glm::vec3(0.0, 1.0, 0.0));
    model = glm::scale(model, glm::vec3(0.5f));
    shader.setMat4("model", model);
    renderCube();
}
```

renderScene 函数使用当前传入的着色器渲染一个平面和一个立方体。

5. 使用生成的深度贴图渲染场景

a. 在顶点着色器中进行坐标转换

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

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

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

输出到片段着色器

T矩阵，将顶点坐标转换到光源处所见空间

lightSpaceMatrix 将顶点坐标从世界空间转换到光源处所见空间。顶点着色器将一个转换到世界空间的顶点位置 vs_out.FragPos 和一个转换到光源处所见空间的顶点位置 vs_out.FragPosLightSpace 传递给片段着色器。

b. 在片段着色器中计算片段颜色

计算环境光照、漫反射光照和镜面光照

```
void main() {
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.8);
    // 环境光照
    vec3 ambient = 0.3 * color; 环境光照使用纹理颜色
    // 漫反射光照
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor; 漫反射光照和镜面光照使用光源颜色
    // 镜面光照
    vec3 viewDir = normalize(viewPos - fs_in.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;
    // 计算阴影
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
                                                    阴影对漫反射光照和镜面光照产生影响
    FragColor = vec4(lighting, 1.0);
}
```

texture 函数用于采样纹理的颜色，第一个参数为纹理采样器，第二个参数为纹理坐标。texture 函数使用之前设置的纹理参数对相应的颜色值进行采样。

片段着色器使用 Blinn-Phong 光照模型来渲染场景，其中环境光照使用纹理颜色，漫反射光照和镜面光照使用光源颜色。如果片段在阴影中，shadow 值为 1；否则为 0。由于阴影不会是全黑的，所以阴影只对漫反射光照和镜面光照产生影响。

计算 shadow 值

```
// 执行透视除法
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
// 变换到[0,1]范围
projCoords = projCoords * 0.5 + 0.5;
// 获得在光源视角下最近点的深度值
float closestDepth = texture(shadowMap, projCoords.xy).r;
// 获得在光源视角下当前片段的深度值
float currentDepth = projCoords.z;
float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;
```

① 将光源处所见空间的位置坐标转换到裁剪空间的标准化设备坐标。在顶点着色器输出一个裁剪空间的位置坐标到 gl_Position 时，OpenGL 将会自动进行透视除法，将坐标范围[-w, w]转换到[-1, 1]。这通过将 x, y, z 分量除以 w 分量实现。由于 FragPosLightSpace 没有通过 gl_Position 传递到片段着色器，所以需要显式执行透视除法。

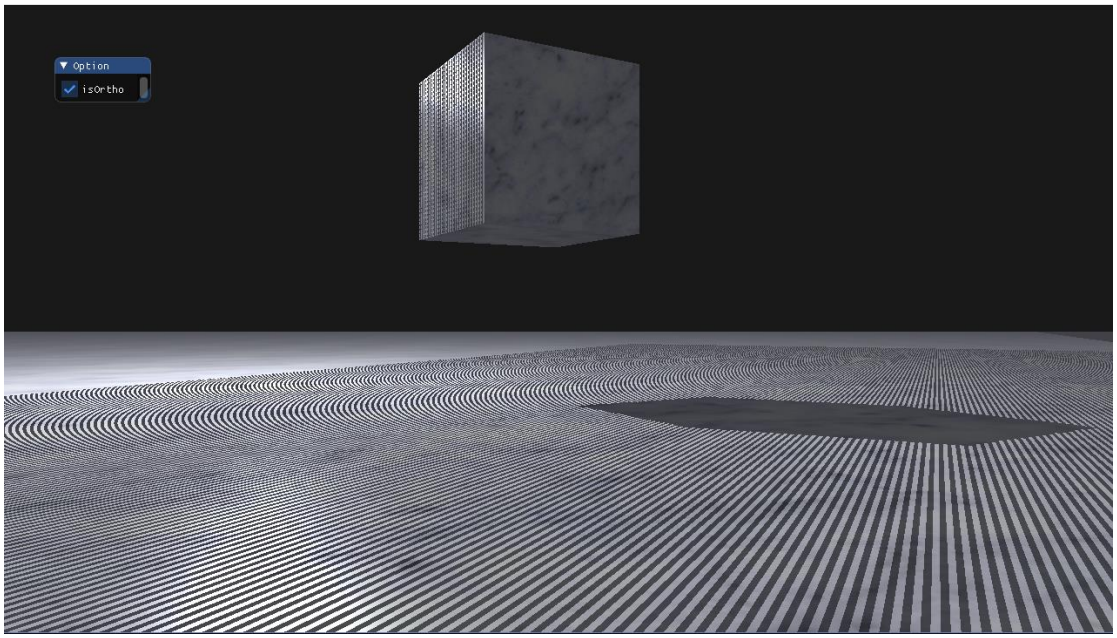
② 将坐标范围从[-1, 1]转换到[0, 1]。深度贴图中存储的深度值范围为[0, 1]，为了和这些深度值进行比较，需要将 z 分量范围转换到[0, 1]。深度贴图的 x 轴和 y 轴范围为[0, 1]，为了获得相应的最近点深度值，需要将 x 分量和 y 分量的范围转换到[0, 1]。

③ 获得在光源视角下最近点的深度值。使用投影坐标 projCoords 的 x 分量和 y 分量索引深度贴图，获得最近点的深度值。

④ 获得在光源视角下当前片段的深度值。当前片段的深度值为投影坐标 projCoords 的 z 分量。

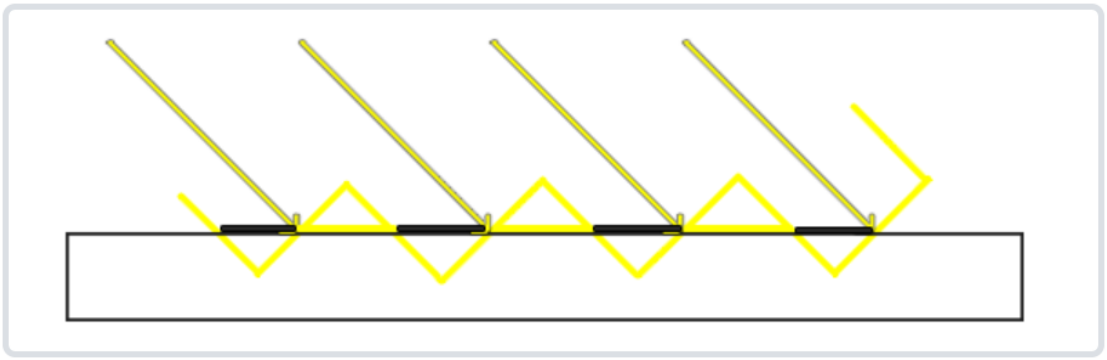
⑤ 比较最近点深度值和当前片段深度值。如果当前片段深度值大于最近点深度值，那么该片段在阴影中，设置 shadow 值为 1；否则为 0。

阴影映射效果

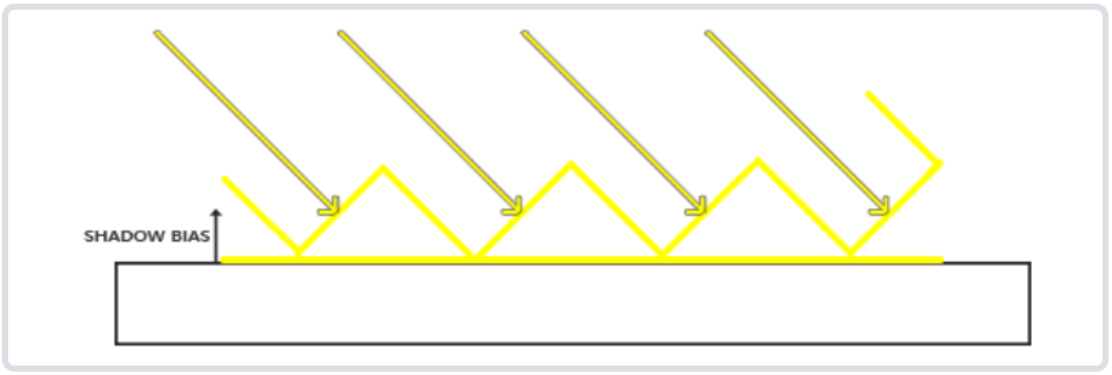


(三) 改进阴影

1. 阴影失真



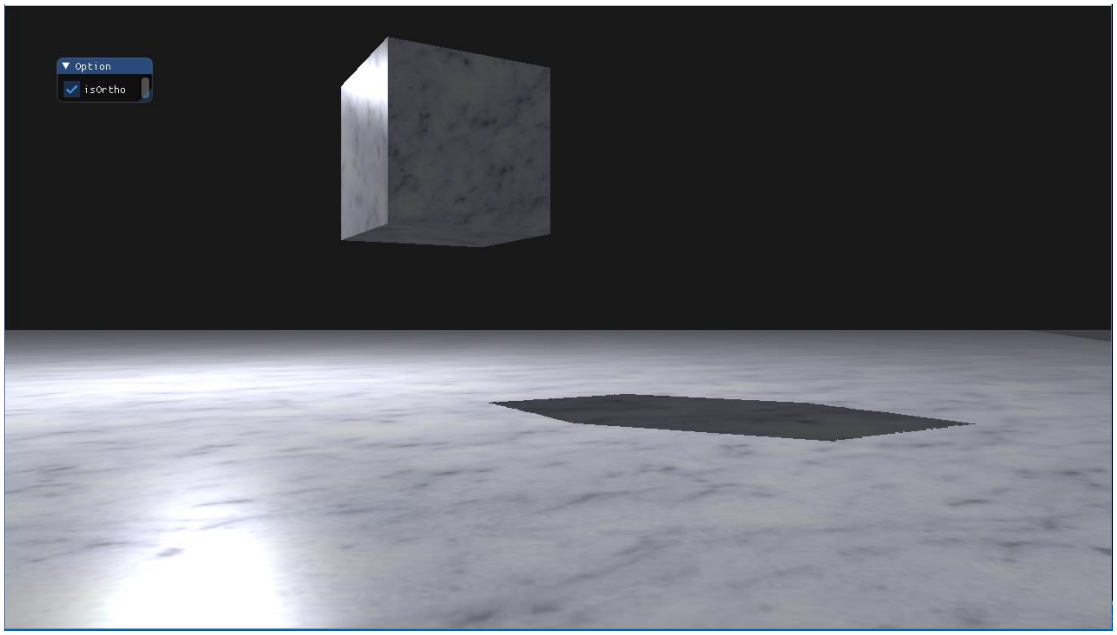
由于阴影贴图受限于解析度，所以多个片段可能对应深度贴图中存储的同一个最近点深度值。如上图所示，多个片段对应深度贴图中存储的同一个最近点深度值。其中，部分片段的深度值小于最近点深度值，所以没有阴影；另一部分片段的深度值大于最近点深度值，所以有阴影。由此产生了条纹效果。




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

对当前片段的深度值（currentDepth）应用一个偏移量（bias）能够解决阴影失真问题。这里设置偏移量的最大值为 0.05，最小值为 0.005。偏移量由光照方向和法线方向确定。光照方向和法线方向的夹角越大，点乘结果越小，偏移量越接近 0.05；光照方向和法线方向的夹角越小，点乘结果越大，偏移量越接近 0.005。

阴影映射效果

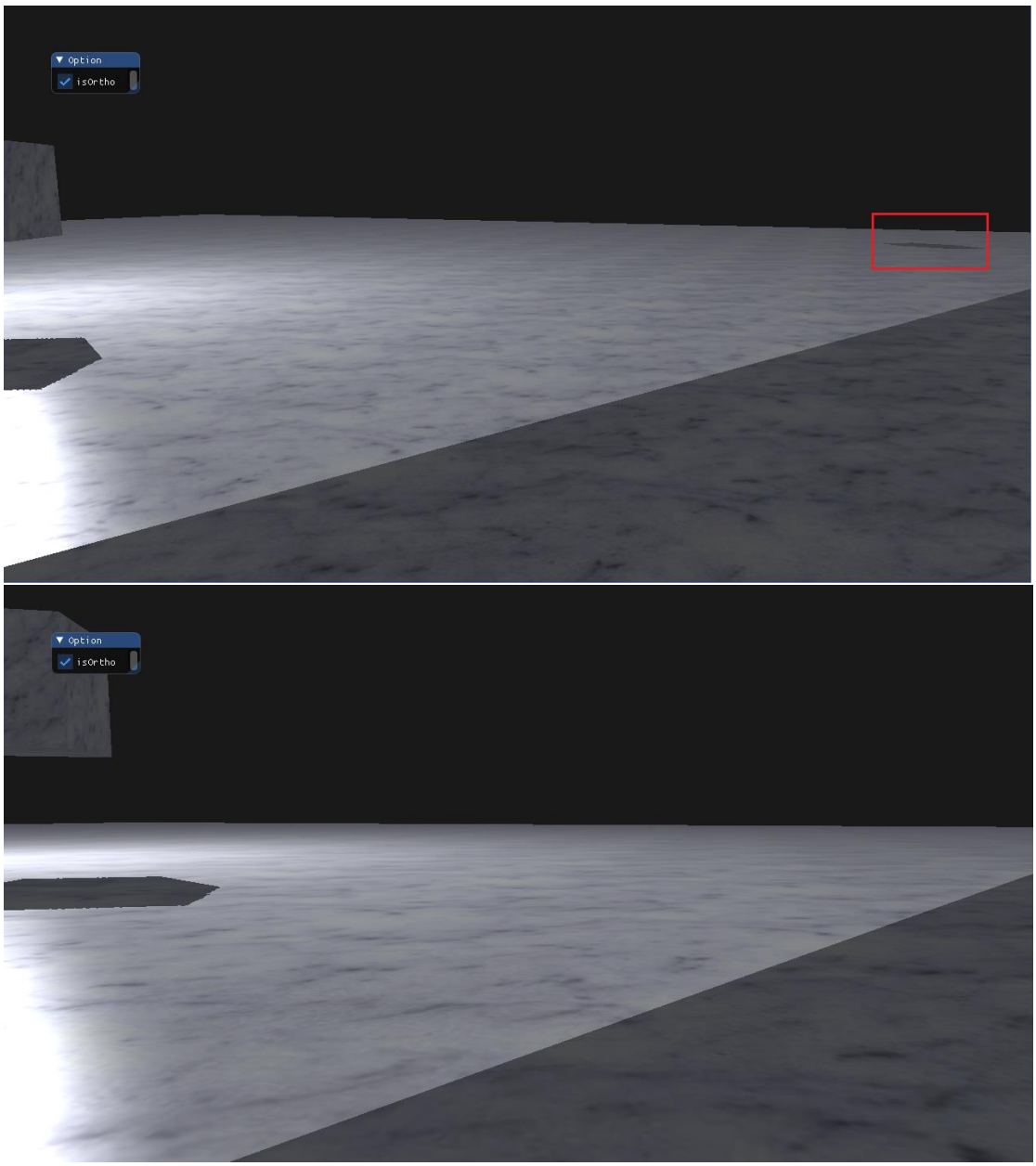


2. 采样过多

超出光源视锥体的投影坐标比 1.0 大，这样采样的深度纹理会超出其[0, 1]范围。如果纹理环绕方式设置不合理，就会得到不正确的深度结果。

将所有超出深度贴图坐标范围的最近点深度值设置为 1.0 可以解决上述问题。将深度贴图的纹理环绕方式设置为 GL_CLAMP_TO_BORDER, 边界颜色设置为{1.0, 1.0, 1.0, 1.0}。这样，如果采样超出深度贴图坐标范围的最近点深度值，将会得到 1.0 的返回结果，光源视锥体不可见的区域将不会处于阴影中。

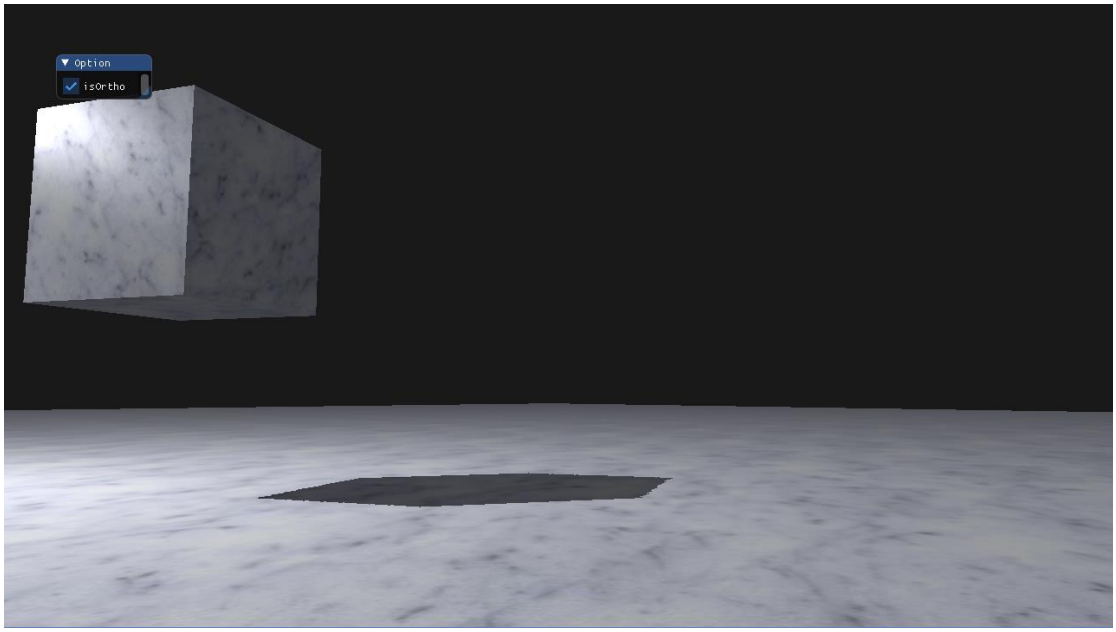
```
// glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
// glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = {1.0, 1.0, 1.0, 1.0};
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```



仍有一部分处于黑暗区域，这是因为那里的坐标超出了光源视锥体的远平面。当一个点比光源视锥体的远平面还要远时，其投影坐标的 z 分量大于 1.0。由于 z 分量大于最近点深度值 1.0，所以 shadow 值为 1.0，该点处于阴影中。

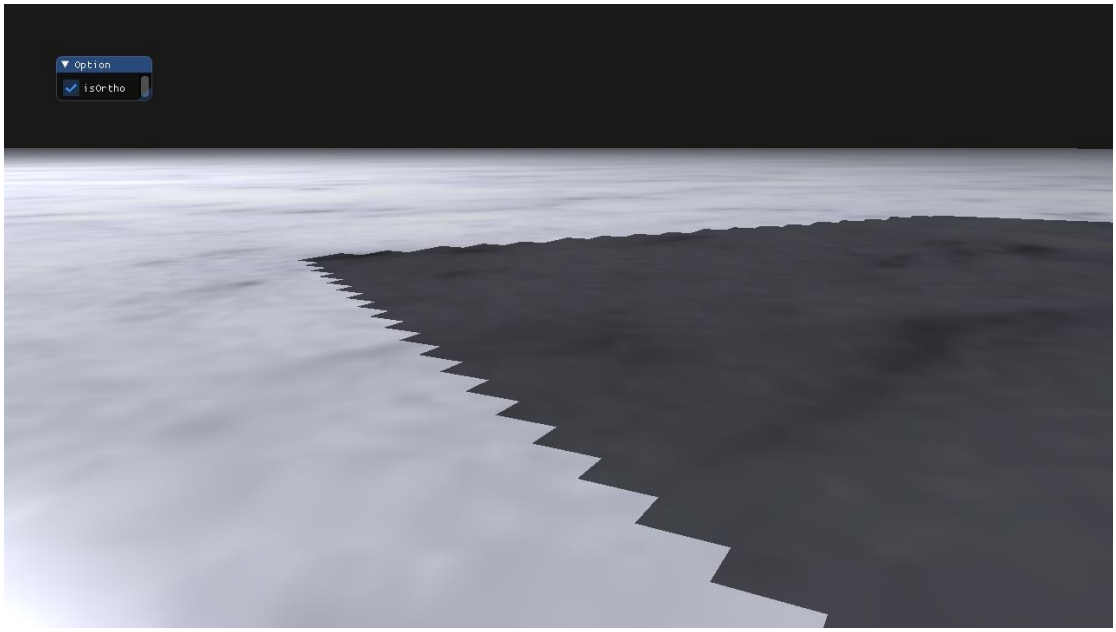
为了解决该问题，如果投影坐标的 z 分量大于 1.0，设置 shadow 值为 0.0。

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



3. PCF

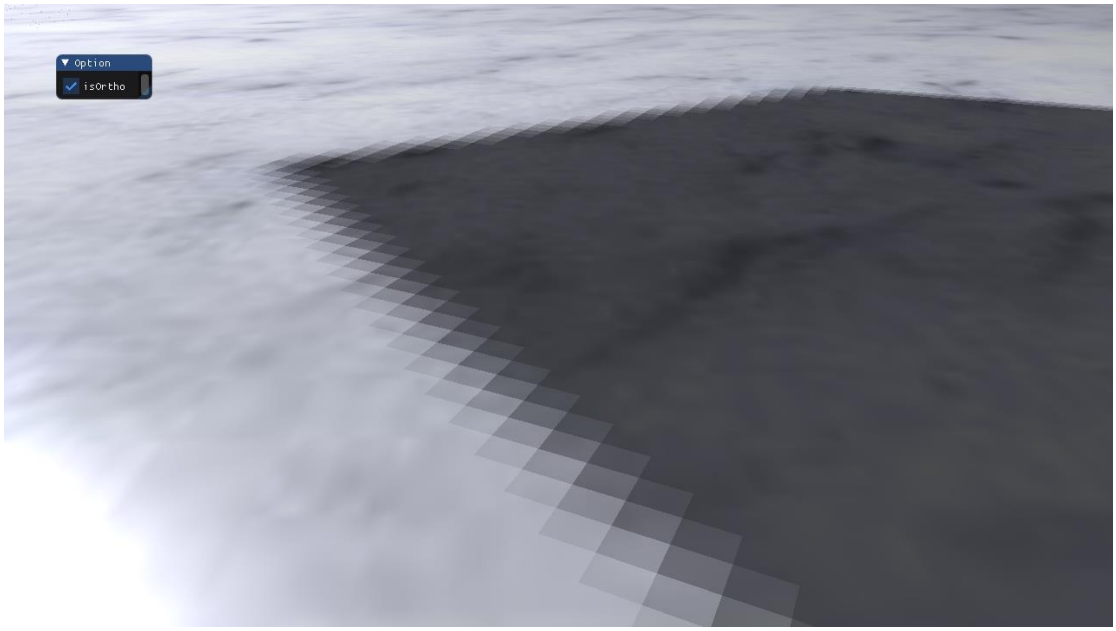
由于阴影贴图受限于解析度，所以多个片段可能对应深度贴图中存储的同一个最近点深度值。如果多个片段从深度贴图的同一个深度值进行采样，这几个片段就会得到同一个阴影，从而产生锯齿边。



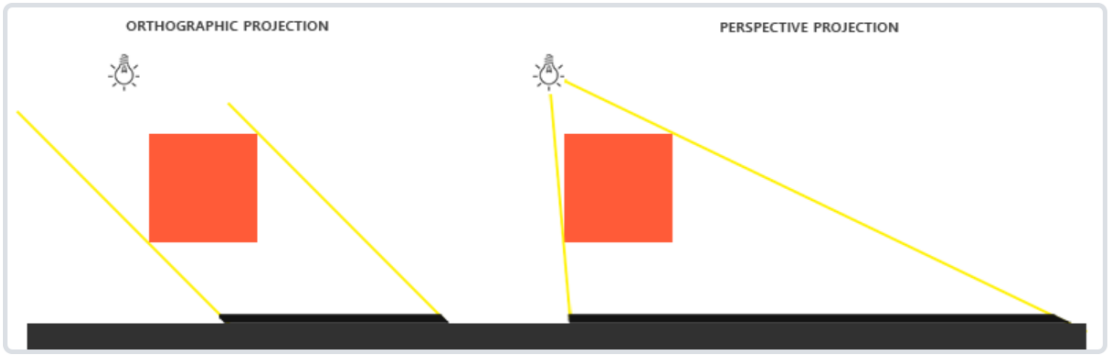
一种解决方案称为 PCF（Percentage-Closer Filtering）。它产生柔和阴影，能够有效减少锯齿块。PCF 从深度贴图中进行多次采样，每一次采样的纹理坐标不同，每个独立的样本可能在也可能不在阴影中。对所有采样结果进行平均化，就能够得到柔和阴影。

```
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;
```

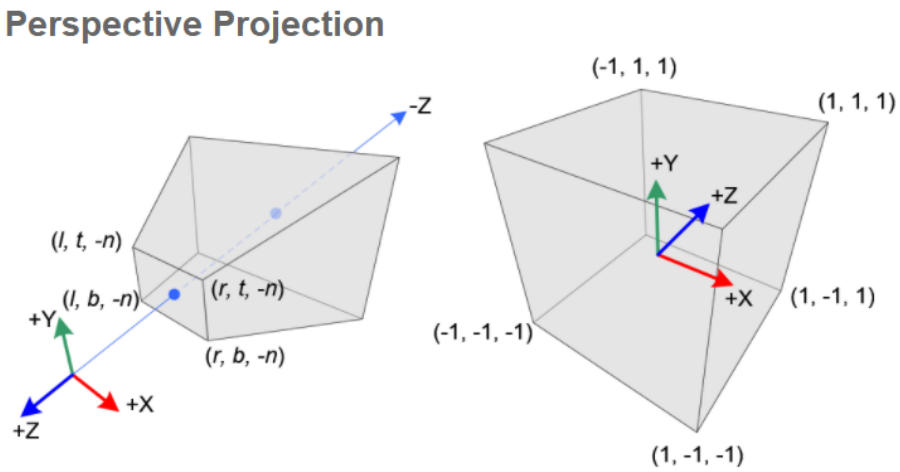
texelSize 函数返回采样器（shadowMap）指定级别多级渐远纹理（0 级）的宽度和高度。用 1 分别除以宽度和高度得到纹理像素的大小。对深度贴图的 9×9 邻域进行采样，得到 9 个深度值 pcfDepth，根据 pcfDepth 计算 shadow 值，平均化后得到最终的 shadow 值。



(四) 正交投影和透视投影

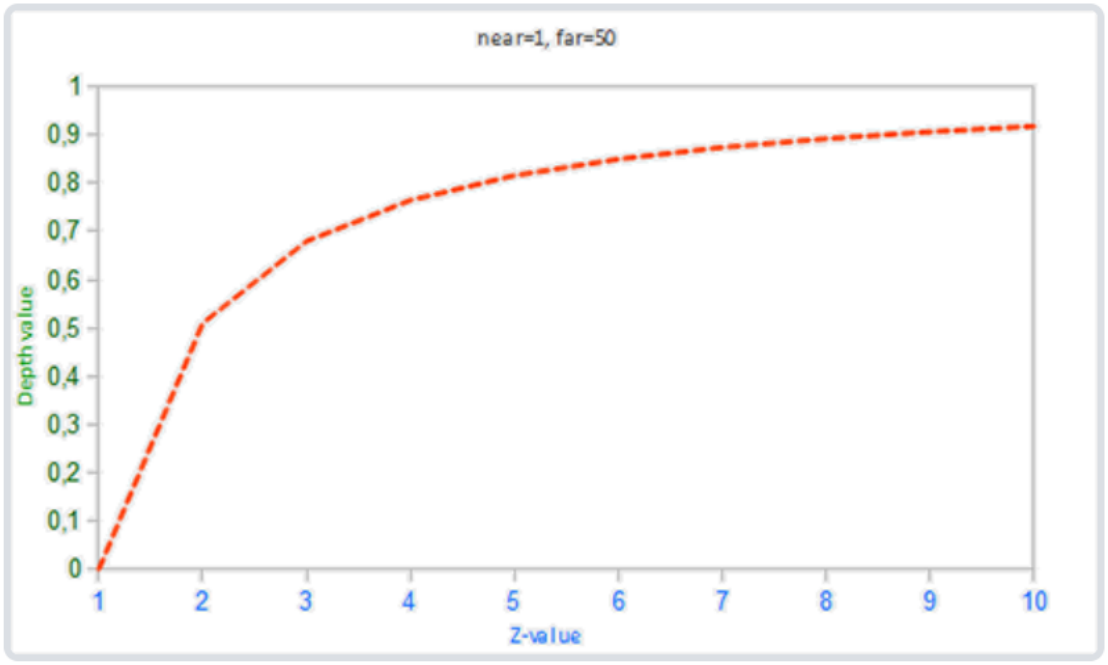


正交投影矩阵不会使用透视对场景进行变形，光线是平行的，通常用于定向光；透视投影矩阵使用透视对所有的顶点进行变形，，通常用于点光源和聚光灯。



Perspective Frustum and Normalized Device Coordinates (NDC)

在透视投影中，位于平截头体内部的一个 3D 点（观察空间）映射到一个立方体（NDC）。x 坐标范围从[l, r]变换到[-1, 1]，y 坐标范围从[b, t]变换到[-1, 1]，z 坐标范围从[-n, -f]变换到[-1, 1]。观察空间坐标使用右手坐标系，NDC 使用左手坐标系。观察空间的摄像机看向-z 轴，NDC 的摄像机看向+z 轴。

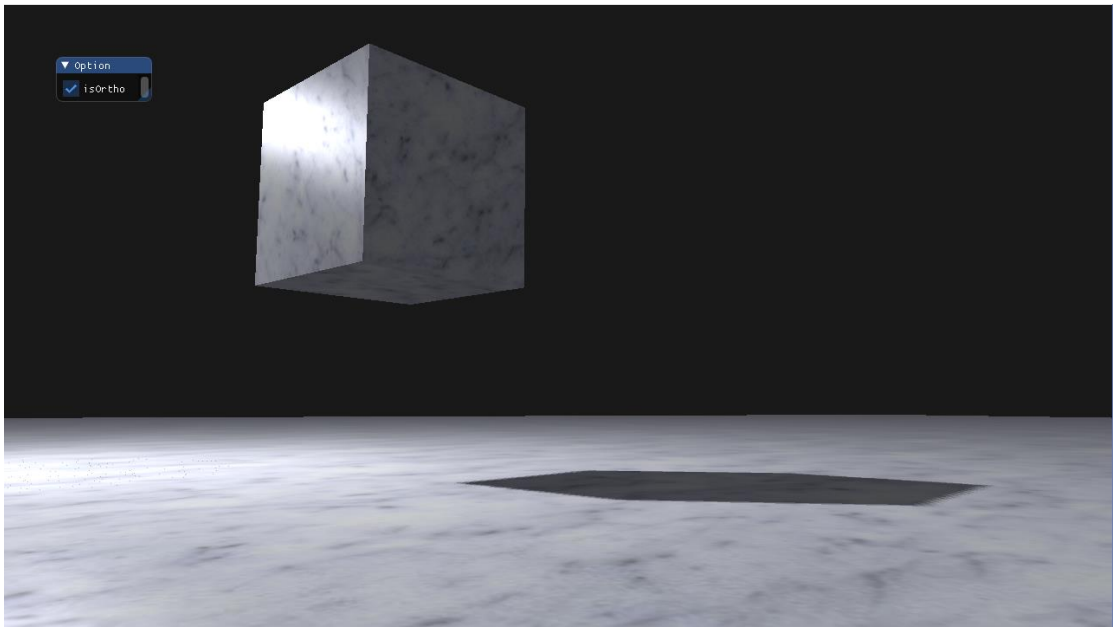


$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$

得到正确的投影性质需要使用一个非线性深度方程，它与 1/z 成正比，在 z 值很小的时候提供高精度，在 z 值很大的时候提供低精度。这个以观察者视角变换 z 值的方程嵌入了在投影矩阵中，当我们将一个顶点坐标从观察空间变换到裁剪空间时，这个非线程方程就被应用了。

正交投影和透视投影另外一个区别是，在使用透视投影的情况下，深度缓冲可视化会得到一个几乎全白的结果，这是因为位于近平面和远平面范围之间的深度值映射成了非线性的深度值，这些非线性深度值大部分接近 1。

正交投影结果



透视投影结果