计算机图形学第七次作业

一、深度贴图

用深度缓冲的方式，对光源的透析图所见的最近的深度值进行采样，将深度值的结果储存到纹理中。

1、为渲染的深度贴图创建一个帧缓冲对象。

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

2、创建一个 2D 纹理，提供给帧缓冲的深度缓冲使用。

```
//创建一个2D纹理，提供给帧缓冲的深度缓冲使用
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
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_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
//处理采样过多问题
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);
//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);
```

3、将每个世界空间坐标变换到光源空间，对正交投影和透视投影采用不同的投影矩阵。

```
glm::mat4 lightProjection;
glm::mat4 lightSpaceMatrix;
float near_plane = 1.0f, far_plane = 7.5f;
if (isPerspective)
{
    lightProjection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, near_plane, far_plane);
}
else
{
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
glm::mat4 lightView = glm::lookAt(lightPos, glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
//光空间的变化矩阵，将世界空间坐标变换到光源处所见到的空间
lightSpaceMatrix = lightProjection * lightView;
```

4、渲染深度贴图

```
//渲染深度贴图
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, woodTexture);
RenderScene(depthShaderProgram);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

二、渲染阴影

1、在顶点着色器中，传递世界空间顶点位置 vs_out.FragPos 和光空间 vs_out.FragPosLightSpace 给片段着色器。

```
const char *vertexShaderSource = "#version 330 core\n"
"layout (location = 0) in vec3 aPos;\n"
"layout (location = 1) in vec3 aNormal;\n"
"layout (location = 2) in vec2 aTexCoords;\n"
"out vec2 TexCoords;\n"
"out VS_OUT {\n"
"    vec3 FragPos;\n"
"    vec3 Normal;\n"
"    vec2 TexCoords;\n"
"    vec4 FragPosLightSpace;\n"
"} vs_out;"
"uniform mat4 projection;\n"
"uniform mat4 model;\n"
"uniform mat4 view;\n"
"uniform mat4 lightSpaceMatrix;\n"
"void main()\n"
"{\n"
"    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));\n"
"    vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;\n"
"    vs_out.TexCoords = aTexCoords;\n"
"    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);\n"
"    gl_Position = projection * view * model * vec4(aPos, 1.0);\n"
"}\0";
```

2、在片段着色器中，计算一个 shadow 值，用来判断片元是否在阴影中，然后使用 Blinn-Phong 光照模型渲染场景。

3、Shadow 值的计算过程：

（1）将光空间坐标转换为标准化设备坐标，再变换到[0,1]范围。

（2）获得光透视视角下的最近深度和当前片元在光透视视角下的深度。

（3）比较二者，若当前片元深度值大于最小深度值，则在阴影中，反之不在阴影中。

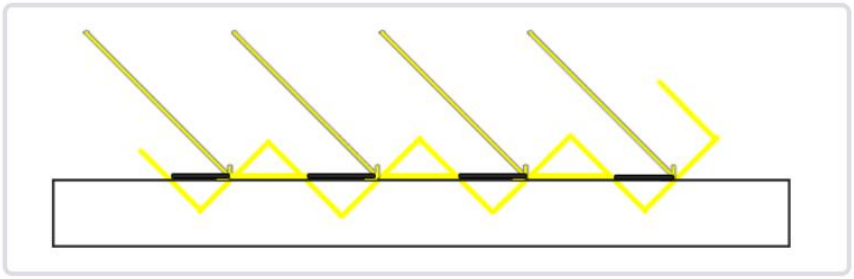
```
"float ShadowCalculation(vec4 fragPosLightSpace)\n"
"{\n"
"    //将光空间坐标转换为标准化设备坐标，再变换到[0,1]范围\n"
"    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;\n"
"    projCoords = projCoords * 0.5 + 0.5;\n"
"    //光透视视角下的最近深度\n"
"    float closestDepth = texture(shadowMap, projCoords.xy).r;\n"
"    //当前片元在光透视视角下的深度\n"
"    float currentDepth = projCoords.z;\n"
"    //比较二者大小\n"
"    float shadow = currentDepth > closestDepth ? 1.0 : 0.0;\n"
"}\n"
```

4、渲染阴影

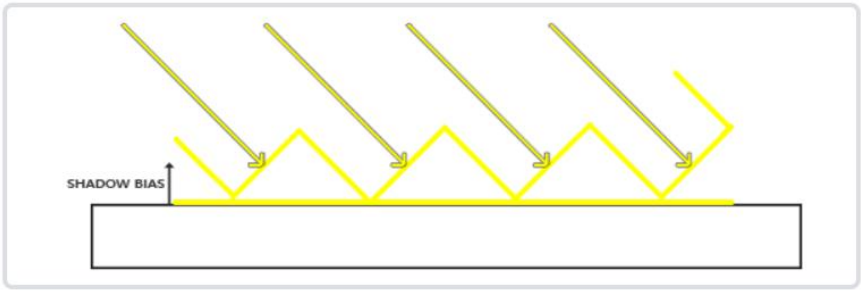
```
//渲染阴影
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(phongShaderProgram);
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
unsigned int phongProjectionLoc = glGetUniformLocation(phongShaderProgram, "projection");
glUniformMatrix4fv(phongProjectionLoc, 1, GL_FALSE, glm::value_ptr(projection));
unsigned int phongViewLoc = glGetUniformLocation(phongShaderProgram, "view");
glUniformMatrix4fv(phongViewLoc, 1, GL_FALSE, glm::value_ptr(view));
unsigned int viewPosLoc = glGetUniformLocation(phongShaderProgram, "viewPos");
glUniform3fv(viewPosLoc, 1, &camera.Position[0]);
unsigned int lightPosLoc = glGetUniformLocation(phongShaderProgram, "lightPos");
glUniform3fv(lightPosLoc, 1, &lightPos[0]);
lightSpaceMatrixLoc = glGetUniformLocation(phongShaderProgram, "lightSpaceMatrix");
glUniformMatrix4fv(lightSpaceMatrixLoc, 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, woodTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene(phongShaderProgram);
```

三、阴影优化

1、阴影失真



如上图所示，阴影贴图受限于解析度，当光源较远时，多个片元可能从深度贴图的同一个值中去采样。当光源以一个角度朝向表面的时候，多个片元就会从同一个斜坡的深度纹理像素中采样，所得到的阴影就会有差异，产生条纹样式。

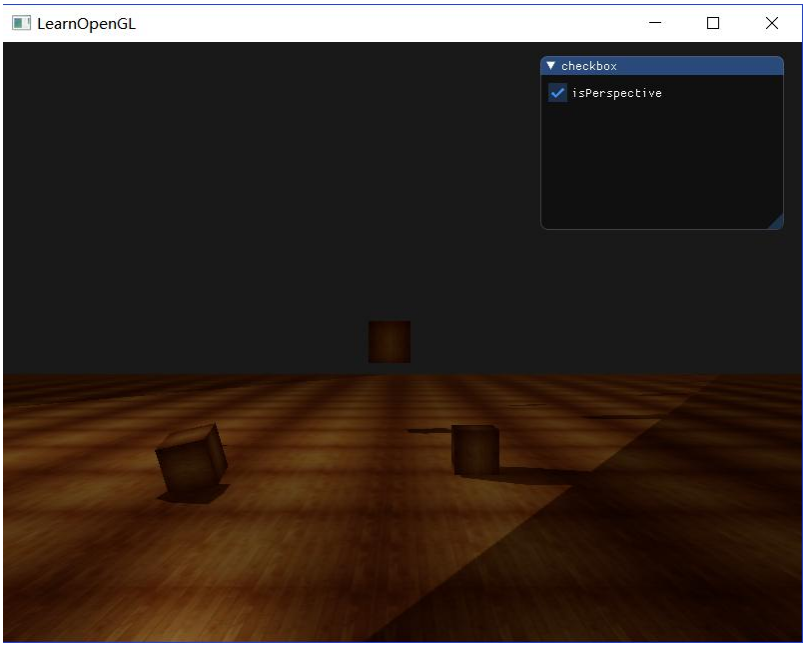


使用阴影偏移可以解决这个问题，对表面的深度应用一个偏移量，使得所有采样点都获得了比表面深度更小的深度值。关于偏移量，可以根据表面朝向光线的角度更改偏移量。

```
"    float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);\n"
```

2、采样过多

以透视投影为例：



可以看到，光照有一块区域全都是阴影，甚至出现了同一个物体被投射出多个阴影的情况。

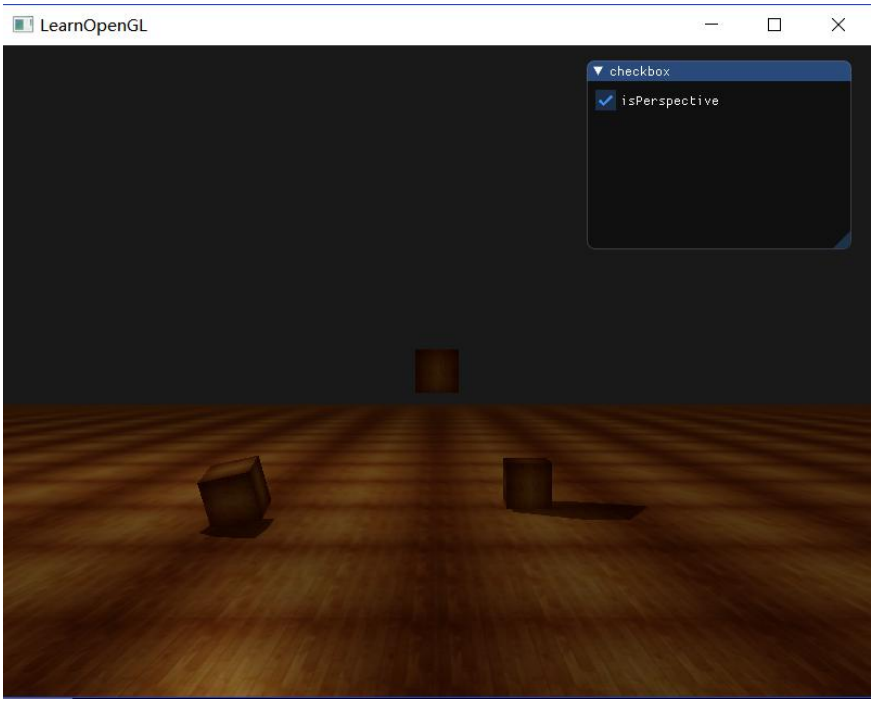
我们把深度贴图的纹理环绕选项设置为 GL_CLAMP_TO_BORDER，使得采样深度贴图[0,1]范围以外的区域时，纹理函数总会返回一个 1.0 的深度值，阴影值为 0.0。

```
//处理采样过多问题
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);
```

同时，我们将比光的远平面还远的点的 shadow 值强制设为 0.0。

```
"    if (projCoords.z > 1.0)\n"
"        shadow = 0.0;\n"
"    return shadow;\n"
```

处理后的运行结果：



3、消除锯齿

从深度贴图中多次采样，根据像素点的位置，采样当前位置及其八邻域的 shadow 值，将所有结果进行平均化，就得到了柔和阴影。

```
"    float shadow = 0.0;\n"
"    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);\n"
"    for (int x = -1; x <= 1; ++x)\n"
"    {\n"
"        for (int y = -1; y <= 1; ++y)\n"
"        {\n"
"            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;\n"
"            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;\n"
"        }\n"
"    }\n"
"    shadow /= 9.0;\n"
```

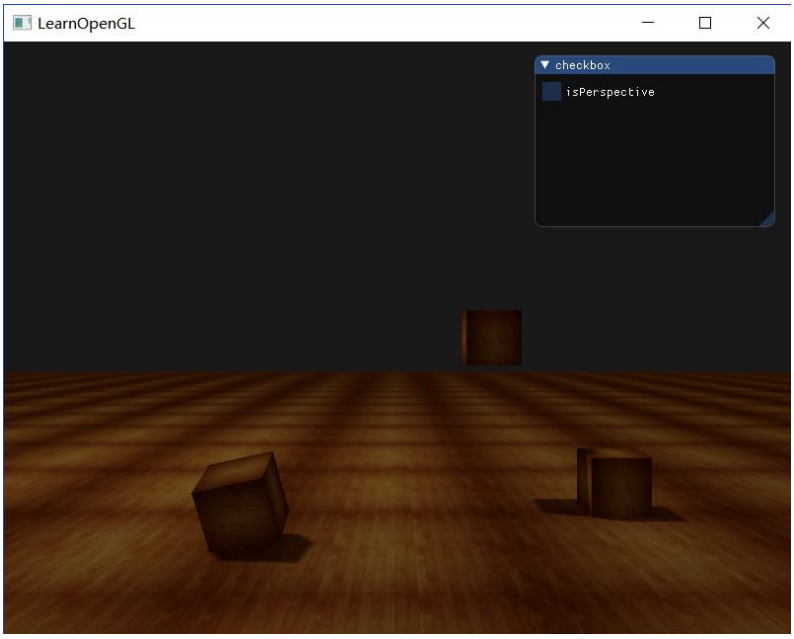
四、正交投影与透视投影

1、根据不同的 lightProjection，实现正交投影和透视投影。

```
if (isPerspective)
{
    lightProjection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, near_plane, far_plane);
}
else
{
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
}
```

2、运行结果

（1）正交投影



（2）透视投影