计算机图形学作业 7 实验报告

16340203 谭江华

基本要求：

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（完成）
2. 优化 Shadowing Mapping（使用了阴影偏移技巧来进行优化以及解决了采样过多的问题）

实现思路&算法：

1. 首先要先绘制一块平面用于显示 shadow。两个三角形即可形成一块 plane。

```
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 VBO
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);
```

2. 然后需要导入贴图。

```
// 导入贴图
unsigned int woodTexture;// = loadTexture(FileSystem::getPath("resources/wood.png").c_str());
glGenTextures(1, &woodTexture);
int width, height, nrComponents;
unsigned char *data = stbi_load("resource/wood.png", &width, &height, &nrComponents, 0);
if (data)
{
    GLenum format;
    if (nrComponents == 1)
        format = GL_RED;
    else if (nrComponents == 3)
        format = GL_RGB;
    else if (nrComponents == 4)
        format = GL_RGBA;

    glBindTexture(GL_TEXTURE_2D, woodTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
    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);

    stbi_image_free(data);
}
else
{
    std::cout << "Texture failed to load at path " << std::endl;
    stbi_image_free(data);
}
```

3. 根据 Shadowing Mapping 的方法，首先需要生成一张深度贴图，即从光的透视图里渲染的深度纹理，用它计算阴影。而这需要我们将场景的渲染结果存储到一个纹理中去，所以需要再次使用帧缓冲。

```
// 为渲染的深度贴图创建一个帧缓冲对象
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

之后创建 2D 纹理，提供给帧缓冲的深度缓冲使用，这里做了一点优化，把深度贴图的纹理环绕选项设置为 GL_CLAMP_TO_BORDER，这样就不会出现超出深度贴图坐标的深度大于 1.0 的情况，可以保证不会有多余的阴影出现。

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

然后把生成的深度纹理作为帧缓冲的深度缓冲，这里由于帧缓冲对象也包括有颜色缓冲，但是不需要使用颜色缓冲，所以使用 glDrawBuffer 和 glReadBuffer 把读和绘制缓冲设置为 GL_NONE。

```
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);
```

4. 上面已经合理地配置完了，将深度值渲染到了纹理的帧缓冲里面了。现在进入到渲染循环中，通过设置 GUI 来选择时进行正交投影的渲染还是透视投影的渲染。在正交投影中，首先需要生成深度贴图。由于是正交投影，所以 lightProjection 是 ortho，如果选择的是透视投影，则是 perspective。这里注意，lightSpaceMatrix 是一个变换矩阵，可以将每个世界空间坐标变换到光源处所见到的那个空间，有了 lightSpaceMatrix 只要给 shader 提供光空间的投影和视图矩阵，就能像往常那样渲染场景了。

```
// 从光的透视图进行场景的深度贴图渲染
glm::mat4 lightProjection, lightView, lightSpaceMatrix;
float near_plane = 1.0f, far_plane = 7.5f;
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
// 光空间的变换矩阵，可以将每个世界空间坐标变换到光源处见到的空间坐标
lightSpaceMatrix = lightProjection * lightView;
// 渲染至深度贴图
simpleDepthShader.use();
simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

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(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

这里进行场景渲染所使用的着色器是很简单的，只做了将顶点变换到光空间的工作。

```
#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);
}
```

```
#version 330 core

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

5. 上面那段使用了一个自定义函数，renderScene，用来对场景进行渲染，包括了渲染平面和渲染两个立方体。

```
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, 1.5f, 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(-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();
}
```

6. 在我们生成完深度贴图之后，就可以开始生成阴影了，在渲染循环中首先要重新设置 viewport 并且调用 glClear 刷新屏幕。

```
// 重新设置viewport
glViewport(0, 0, window_size_x, window_size_y);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

然后利用生成的深度贴图来重新渲染场景。

```
// 利用生成的深度贴图来重新渲染场景
shader.use();
shader.setMat4("projection", projection);
shader.setMat4("view", view);
shader.setVec3("lightPos", lightPos2);
shader.setVec3("viewPos", camera);
shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, woodTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);
renderScene(shader);
```

生成阴影的工作主要是在片段着色器中实现的，需要检验一个片元是否在阴影中。但是需要提前在顶点着色器中先进行光空间的变换。

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

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经过变换的世界空间顶点位置 vs_out.FragPos 和一个光空间的 vs_out.FragPosLightSpace 给片段着色器。

之后在片段着色器中，先使用函数根据顶点着色器传过来的 FragPosLightSpace 算出阴影值，再使用 Blinn-Phong 光照模型渲染场景。

```
// calculate shadow
float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
```

在计算阴影值的函数中，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，但当使用正交投影矩阵时，不需要做透视除法，但为了使得透视投影矩阵也能使用该着色器，所以保留了这一步。

```
// perform perspective divide
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
```

这样就返回了片元在光空间的-1 到 1 的范围。

因为来自深度贴图的深度是在 0 到 1 的范围，所以使用 projCoords 从深度贴图中去采样，这里整个 projCoords 向量都需要变换到[0,1]范围。

```
// transform to [0,1] range
projCoords = projCoords * 0.5 + 0.5;
```

之后通过这些坐标获取光的位置视野下最近的深度，并且比较它与片元的当前深度，据此得到该片元的阴影值。这里加入了另一个优化，那就是阴影偏移，即简单的对深度贴图应用一个偏移量，这样片元就不会被错误地认为在表面之下了。

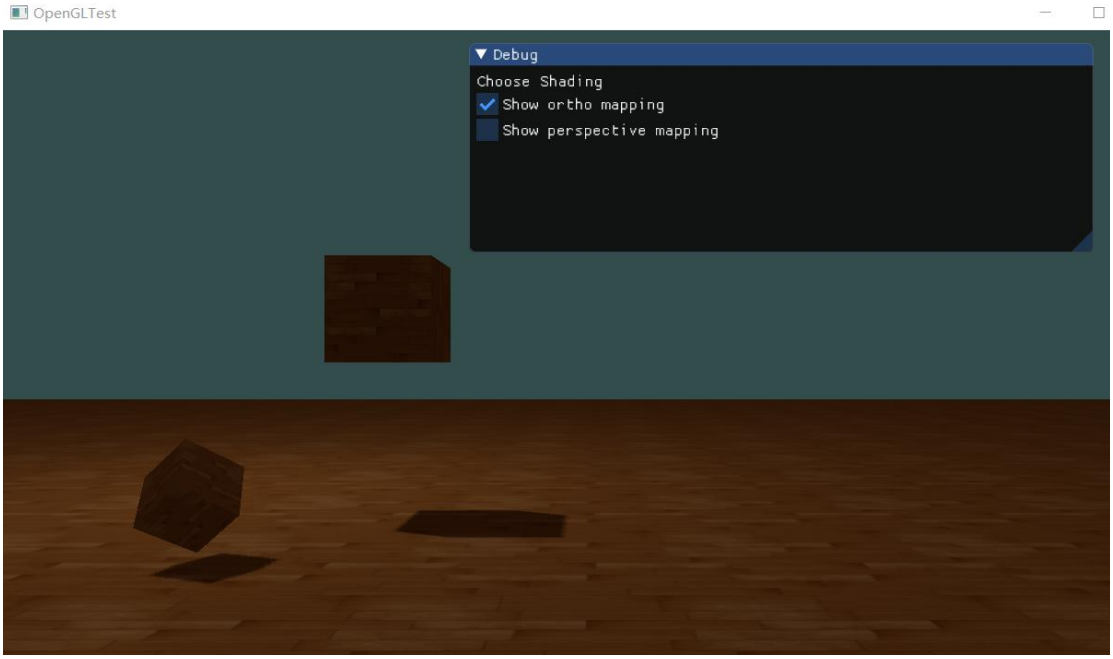
```
// get closest depth value from light's perspective (using [0,1] range fragPosLight
float closestDepth = texture(shadowMap, projCoords.xy).r;
// get depth of current fragment from light's perspective
float currentDepth = projCoords.z;
// calculate bias (based on depth map resolution and slope)
vec3 normal = normalize(fs_in.Normal);
vec3 lightDir = normalize(lightPos - fs_in.FragPos);
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
// check whether current frag pos is in shadow
// float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
// PCF
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).
        shadow += currentDepth - bias > pcfDepth  ? 1.0 : 0.0;
    }
}
```

7. 在利用上面的着色器计算完之后，就可以再次调用 renderScene 来进行场景渲染了，这样就得到了一个优化过的，更加真实的阴影效果。

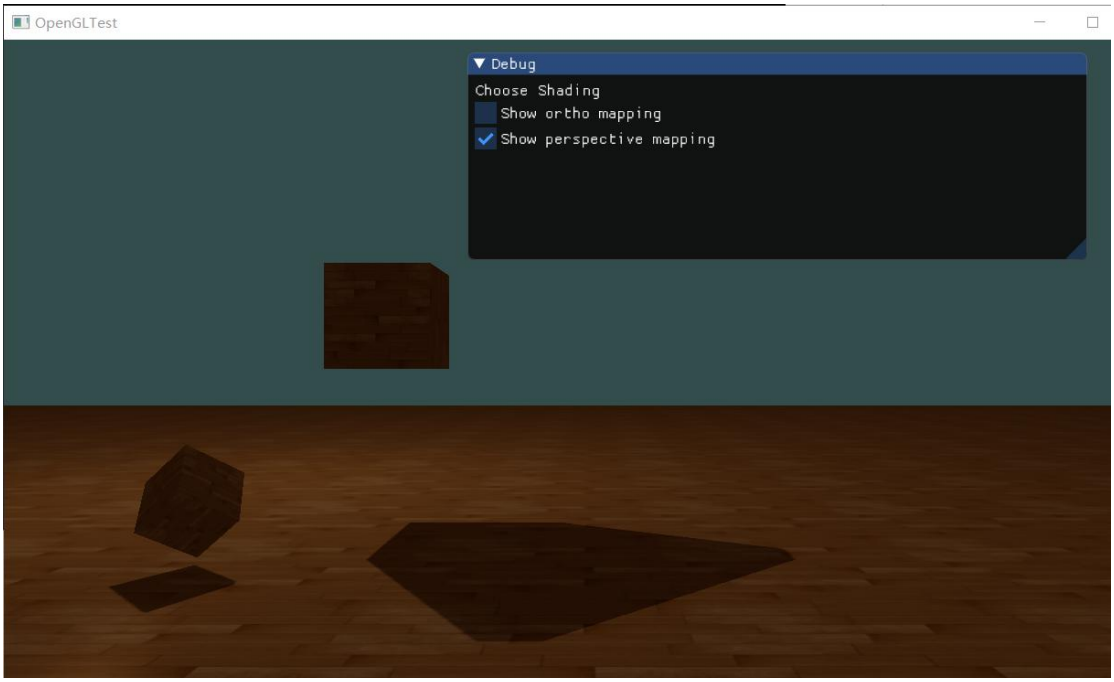
```
// 利用生成的深度贴图来重新渲染场景
shader.use();
shader.setMat4("projection", projection);
shader.setMat4("view", view);
shader.setVec3("lightPos", lightPos2);
shader.setVec3("viewPos", camera);
shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, woodTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);
renderScene(shader);
```

最终效果如图：

正交投影下的 Shadowing Mapping



透视投影下的 Shadowing Mapping


OpenGLTest
Debug
Choose Shading
Show ortho mapping
Show perspective mapping