计算机图形学 Homework7

15331416 赵寒旭

目录

1. 运行结果

光源投影方式：正交投影

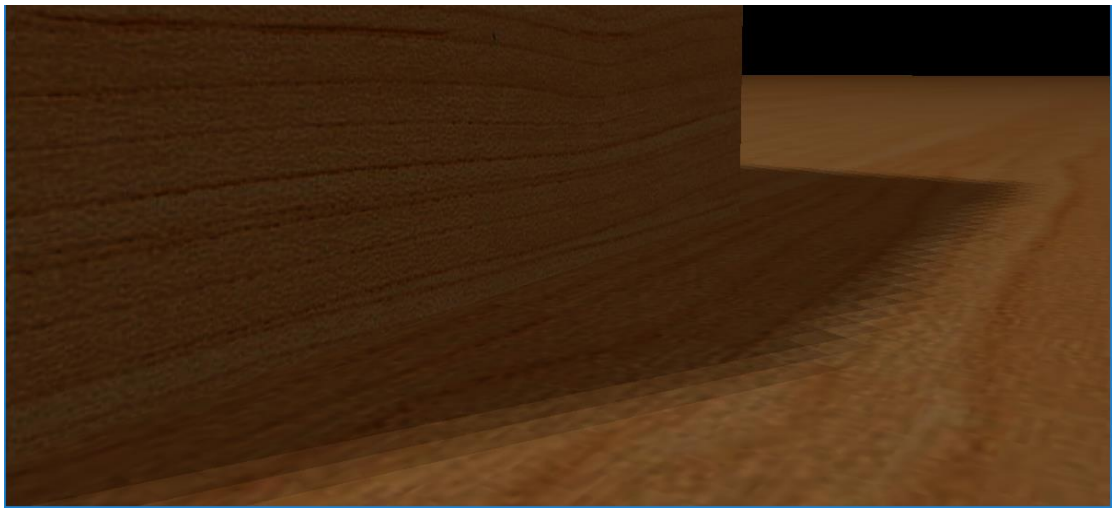
场景：4 个立方体，一块平面

交互方式：WASD 控制前左后右移动，鼠标控制视角，Q 键隐藏鼠标指针，E 键显示鼠标指针。

1）初始界面

2）在立方体表面显示阴影

3）在平面上显示阴影



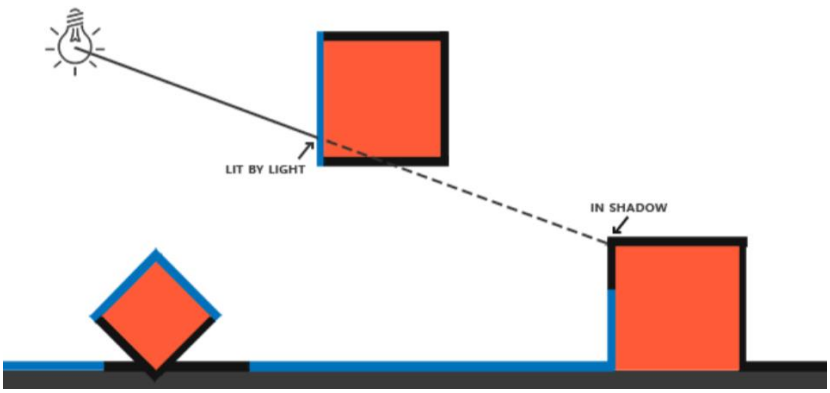
另有视频在 doc 文件夹下。

2. 实现思路

2.1 阴影渲染

本节重点：结合代码解释 Shadowing Mapping 算法。

首先简单描述以下阴影渲染的总体思路和基本原理。



阴影映射即以光的位置为视角进行渲染，所有能看到的表面都被点亮，而看不到的地方认为是被阴影覆盖。如上图所示，标记为蓝色的部分是被光线直射的地方 fragment，黑色部分表示应被渲染为带阴影的 fragment。

对上图射线来说，射线第一次击中物体的点作为**最近点**，如果射线上某一点点比此点距起点更远，这个点就在阴影中。

为了避免性能的消耗，我们借助深度缓冲得到摄像机视角下目标场景的深度值。

我们从光源的透视图来渲染场景，将得到的深度值结果储存到纹理中，对光源透视图所见的最近深度值进行采样，最终使深度值显示从光源的透视图下见到的第一个片元。

即我们得到了从光源出发所有最近点的深度值，并把它们统称为**深度贴图（depth map）**。

具体的阴影渲染过程，可以按照以下两个基本步骤来进行。

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

② 以 camera 视角渲染场景，使用 Shadowing Mapping 算法（比较当前深度值与在 DepthMap Texture 的深度值），决定某个点是否在阴影下。

1）光源空间变换

我们使用一个投影矩阵和一个观察矩阵组合而成的 T 变换将世界坐标转化到光源的可见坐标空间。

使用正交投影矩阵 lightProjection。

```
GLfloat near_plane = 1.0f, far_plane = 7.5f;
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
```

创建观察矩阵 lightView 来变换每个物体，将其转移到光源视角下，使从光源位置看向场景中央。

```
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
```

结合投影矩阵和观察矩阵，我们得到了光空间的变换矩阵 lightSpaceMatrix，作用是将场景中的世界坐标转换为光源空间下的坐标，**此时每个 fragment 坐标的 z 分量即对应它在此光源下的深度**。

```
lightSpaceMatrix = lightProjection * lightView;
```

2）创建深度贴图

我们要将光源视角下的渲染结果存储到深度贴图中，首先需要创建一张深度贴图。

（1）创建一个帧缓冲对象

```
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

（2）创建一个 2D 纹理

因为**只需要考虑深度值**，指定纹理格式为 GL_DEPTH_COMPONENT。同时设置深度贴图的解析度即纹理的宽高分别为 1200 和 800。

```
GLuint depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 1200, 800, 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);
```

（3）把生成的深度纹理作为帧缓冲的深度缓冲

深度贴图仅需要深度值，显式告知 OpenGL 不使用任何颜色数据进行渲染。

```
// 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);
// 显示说明不使用任何颜色数据进行渲染
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

3）以光源视角渲染场景深度信息

在渲染循环中：

```
// 从光的透视图下渲染场景的深度信息
glUseProgram(simpleDepthShader);
setMat4(simpleDepthShader, "lightSpaceMatrix", lightSpaceMatrix);
glViewport(0, 0, display_w, display_h);
// 随窗口大小自动调整
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, display_w, display_h, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
RenderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

（1）顶点着色器（顶点变换到光空间着色器）

将顶点通过空间变换矩阵 lightSpaceMatrix 变换到光空间中。

```
#version 440 core
layout (location = 0) in vec3 aPos;
uniform mat4 lightSpaceMatrix;
uniform mat4 model;
void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}
```

（2）片段着色器（(顶点变换到光空间着色器）

只需要深度缓冲，不需要考虑 fragment 颜色，着色器可以直接置为空。

```
#version 440 core
void main()
{
}
```

（3）RenderCube 函数

初始化顶点坐标数组：以一面为例，分别是六个顶点（正方形由两个三角形组成）的坐标，法向量，纹理坐标。

```
GLfloat vertices[] = {
    // Back face
    -0.4f, -0.4f, -0.4f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f, // Bottom-left
    0.4f, 0.4f, -0.4f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f, // top-right
    0.4f, -0.4f, -0.4f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f, // bottom-right
    0.4f, 0.4f, -0.4f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f,  // top-right
    -0.4f, -0.4f, -0.4f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,  // bottom-left
    -0.4f, 0.4f, -0.4f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f,// top-left
```

链接顶点属性：(对应顶点着色器输入)

```
glBindVertexArray(cubeVAO);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

（4）RenderScene 函数

渲染场景。

```
void RenderScene(int &shader)
{
    glUseProgram(shader);
    // Floor
    glm::mat4 model;
    setMat4(shader, "model", model);
    glBindVertexArray(planeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);
    // Cubes
    model = glm::mat4();
    model = glm::translate(model, glm::vec3(0.0f, 1.0f, 0.0));
    setMat4(shader, "model", model);
    RenderCube();
```

激活传入的着色器，用此着色器分别渲染 floor 和 cube。

4）生成阴影

（1）顶点着色器（主着色器）

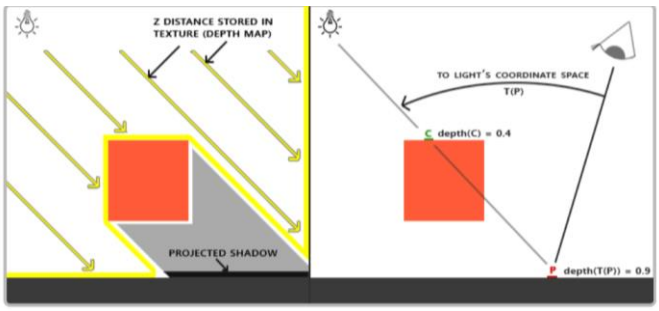
进行光空间的变换。

用空间变换矩阵 lightSpaceMatrix 把坐标从世界坐标转到光空间坐标，其余变换同普通顶点着色器。

```
#version 440 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);
}
```

（2）片段着色器（主着色器）

首先介绍根据深度贴图决定一个 fragment 是否在阴影中的方法。



平行光下，考虑右图 p 点，渲染点 P 处 fragment 时，将 P 坐标转到光空间坐标中，z 坐标对应 P 在此光源下的深度（此处为 0.9）。使用 P 在光源的坐标空间的坐标，可以索引深度贴图，获得光视角中最近点的深度，此处为 C 点，深度 0.4。索引深度贴图的结果 0.4 小于 P 的深度，可以确定点 P 在阴影中。

此处片段着色器相比于普通着色器增加了阴影计算。

```
#version 440 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;
```

考虑到环境光照是固有的，即使是阴影部分也会有这部分的光照，我们将它独立出来加和，计算出的阴影因子不对它起作用。

```
// shadow
float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
FragColor = vec4(lighting, 1.0);
```

shadow 计算值为 0 时，没有阴影，光照正常。

shadow 计算值为 1 时，diffuse 和 specular 分量权重置 0，只有环境光照。

为计算阴影因子 shadow，我们使用 ShadowCalculation 函数：

```
// 执行透视除法
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
// 变换到[0,1]的范围
projCoords = projCoords * 0.5 + 0.5;
// 取得最近点的深度（使用[0,1]范围下的fragPosLight当坐标）
float closestDepth = texture(shadowMap, projCoords.xy).r;
// 取得当前片元在光源视角下的深度
float currentDepth = projCoords.z;
```

① 执行透视除法，将裁切空间坐标的范围[-w,w]转到[-1,1]。

② 为和深度贴图坐标范围匹配，把 projCoords 变换到[0,1]的范围。

③ 得到光源视角下最近的深度 closestDepth

④ 得到 fragment 在光源视角下的深度 currentDepth

⑤ 比较 closestDepth 和 currentDepth，若 currentDepth 值更大，fragment 就在阴影中 shadow = 1.0，反之光照正常，shadow = 0.0。

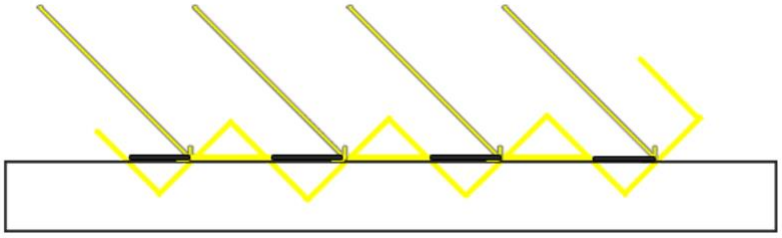
2.2 阴影优化

此部分的优化都在主着色器中的片段着色器里完成。

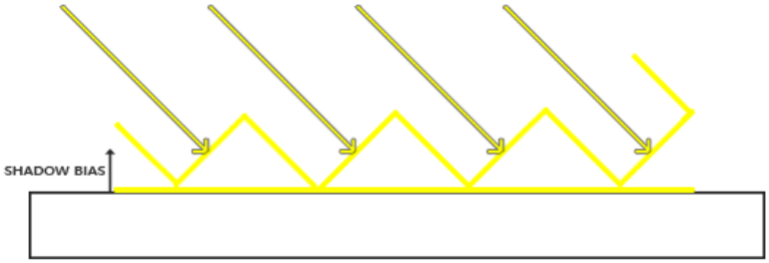
1）阴影偏移

针对阴影失真（Shadow Acne）带来的不真实感，我们使用阴影偏移（Shadow bias）对表面的深度应用一个偏移量，避免 fragment 被错误地认为在表面之下。

光线方向与表面形成夹角时，深度贴图在一个角度下渲染，多个 fragment 会从同一个斜坡的深度纹理像素中采样，有些在表面上侧，有些在表面下侧，部分 fragment 被认为在阴影之中，会产生条纹。



应用偏移量后，所有采样点都获得了比表面深度更小的深度值，不会有 fragment 被认为在表面之下，没有任何阴影。




```
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);
```

最后用 currentDepth 减去 bias 值得到最终深度值。

2）采样过多

默认把光的视锥不可见的区域认为是在阴影中，会使中心区域之外的部分区域显示错误的阴影效果。

当一个点的深度值超过光的远平面，就把 shadow 置为 0，认为没有阴影。

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

3）PCF

此方法用于柔和阴影。从深度贴图中多次采样，每个采样点的计算结果可能都有不同，把多次结果做平均后再为阴影因子 shadow 赋值。

通过对纹理坐标进行偏移，确保每个新样本来自不同的深度值。此处采样得到 9 个值，后对计算结果求平均。

```
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;
        // 判断是否在阴影中，是+1
        shadow += currentDepth - bias > pcfDepth  ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```