# 计算机图形学作业报告

廖蕾 16340135

## Basic

- 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

答：

- 渲染一个正方体和平面：
  这里的方法使用到之前通过关键点，渲染三角形的方法去渲染得到平面和正方体的。平面和正方体顶点信息如下：

```
// floor vertices
float floorVertices[] = {
         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,  0.0f,  0.0f,

         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, 25.0f, 25.0f,
        -25.0f, -0.5f, -25.0f, 0.0f, 1.0f, 0.0f,  0.0f, 25.0f
};
float vertices[] = {
        -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 0.0f, 0.0f,
         1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 1.0f, 1.0f,
         1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 1.0f, 0.0f,
         1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 1.0f, 1.0f,
        -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 0.0f, 0.0f,
        -1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f, 0.0f, 1.0f,

        -1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f, 0.0f,
         1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 1.0f, 0.0f,
         1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 1.0f, 1.0f,
         1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 1.0f, 1.0f,
        -1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f, 1.0f,
        -1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f, 0.0f,

        -1.0f,  1.0f,  1.0f, -1.0f,  0.0f,  0.0f, 1.0f, 0.0f,
        -1.0f,  1.0f, -1.0f, -1.0f,  0.0f,  0.0f, 1.0f, 1.0f,
        -1.0f, -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
        -1.0f, -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
        -1.0f, -1.0f,  1.0f, -1.0f,  0.0f,  0.0f, 0.0f, 0.0f,
        -1.0f,  1.0f,  1.0f, -1.0f,  0.0f,  0.0f, 1.0f, 0.0f,
```

```
         1.0f,  1.0f,  1.0f,  1.0f,  0.0f,  0.0f, 1.0f, 0.0f,
         1.0f, -1.0f, -1.0f,  1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
         1.0f,  1.0f, -1.0f,  1.0f,  0.0f,  0.0f, 1.0f, 1.0f,
         1.0f, -1.0f, -1.0f,  1.0f,  0.0f,  0.0f, 0.0f, 1.0f,
         1.0f,  1.0f,  1.0f,  1.0f,  0.0f,  0.0f, 1.0f, 0.0f,
         1.0f, -1.0f,  1.0f,  1.0f,  0.0f,  0.0f, 0.0f, 0.0f,

        -1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f, 0.0f, 1.0f,
         1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f, 1.0f, 1.0f,
         1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f, 1.0f, 0.0f,
         1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f, 1.0f, 0.0f,
        -1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f, 0.0f, 0.0f,
        -1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f, 0.0f, 1.0f,

        -1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f, 0.0f, 1.0f,
         1.0f,  1.0f , 1.0f,  0.0f,  1.0f,  0.0f, 1.0f, 0.0f,
         1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f, 1.0f, 1.0f,
         1.0f,  1.0f,  1.0f,  0.0f,  1.0f,  0.0f, 1.0f, 0.0f,
        -1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f, 0.0f, 1.0f,
        -1.0f,  1.0f,  1.0f,  0.0f,  1.0f,  0.0f, 0.0f, 0.0f
};
```

但是这里需要对我们场景中的物体进行深度贴图。深度贴图是从光的透视图里渲染的深度纹理，用它计算阴影。因为我们需要将场景的渲染结果储存到一个纹理中，我们将再次需要帧缓冲。
我们需要给一个帧缓冲对象，然后创建一个2D纹理，提供给帧缓冲的深度缓冲使用。之后把我们把生成的深度纹理作为帧缓冲的深度缓冲。这里的代码如下：

```
// create depth map
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);

//Create depth texture
unsigned int depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 1024, 1024, 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);
GLfloat borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

- 光源空间的变换
  这里用到的是**正交投影**，是一个所有光线都平行的定向光。为了创建一个视图矩阵来变换每个物体，把它们变换到从光源视角可见的空间中，我们将使用`glm::lookAt`函数；这次从光源的位置看向场景中央。
  使用到的代码如下：

```
glm::mat4 lightSpaceMatrix = glm::mat4(1.0f); // make sure to initialize
matrix to identity matrix first
glm::mat4 lightView = glm::mat4(1.0f);
glm::mat4 lightProjection = glm::mat4(1.0f);
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));
lightSpaceMatrix = lightProjection * lightView;

// render from light's perspective
glCullFace(GL_FRONT);
depthShader.use();
depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
glViewport(0, 0, 1024, 1024);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
```

- 渲染至深度贴图

  这里使用到的顶点着色器一个单独模型的一个顶点，使用lightSpaceMatrix变换到光空间中。代码如下：

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 model;
uniform mat4 lightSpaceMatrix;

void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}
```

  而这里的片段着色器不用做什么事，所以用一个空的着色器就好了：

```
#version 330 core
void main(){}
```

- 渲染阴影

  之后还要将深度贴图渲染到四边形上的像素着色器上，正确地生成深度贴图以后我们就可以开始生成阴影了。像素着色器需要去检验一个片元是否在阴影之中，在顶点着色器中需要进行光空间的变换，这样的代码如下所示：

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in 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()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    vs_out.FragPos = vec3(model * vec4(position, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * normal;
    vs_out.TexCoords = texCoords;
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);
}
```

像素着色器和顶点着色器用同一个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普通的经变换的世界空间顶点位置vs_out.FragPos和一个光空间的vs_out.FragPosLightSpace给像素着色器。
像素着色器使用Blinn-Phong光照模型渲染场景。计算出一个shadow值，当fragment在阴影中时是1.0，在阴影外是0.0。然后，diffuse和specular颜色会乘以这个阴影元素。由于阴影不会是全黑的（由于散射），所以把ambient分量从乘法中剔除。代码如下：

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 objectColor;

uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;
```

```
float ShadowCalculation(vec4 fragPosLightSpace) {
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    float currentDepth = projCoords.z;
    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);
    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;
    if(projCoords.z > 1.0)
        shadow = 0.0;

    return shadow;
}
void main()
{
    vec3 color = objectColor;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.3);
    // ambient
    vec3 ambient = 0.6 * color;
    // diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // specular
    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;
    // calculate shadow
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    shadow = min(shadow, 0.75);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) *
color;
    FragColor = vec4(lighting, 1.0f);
}
```

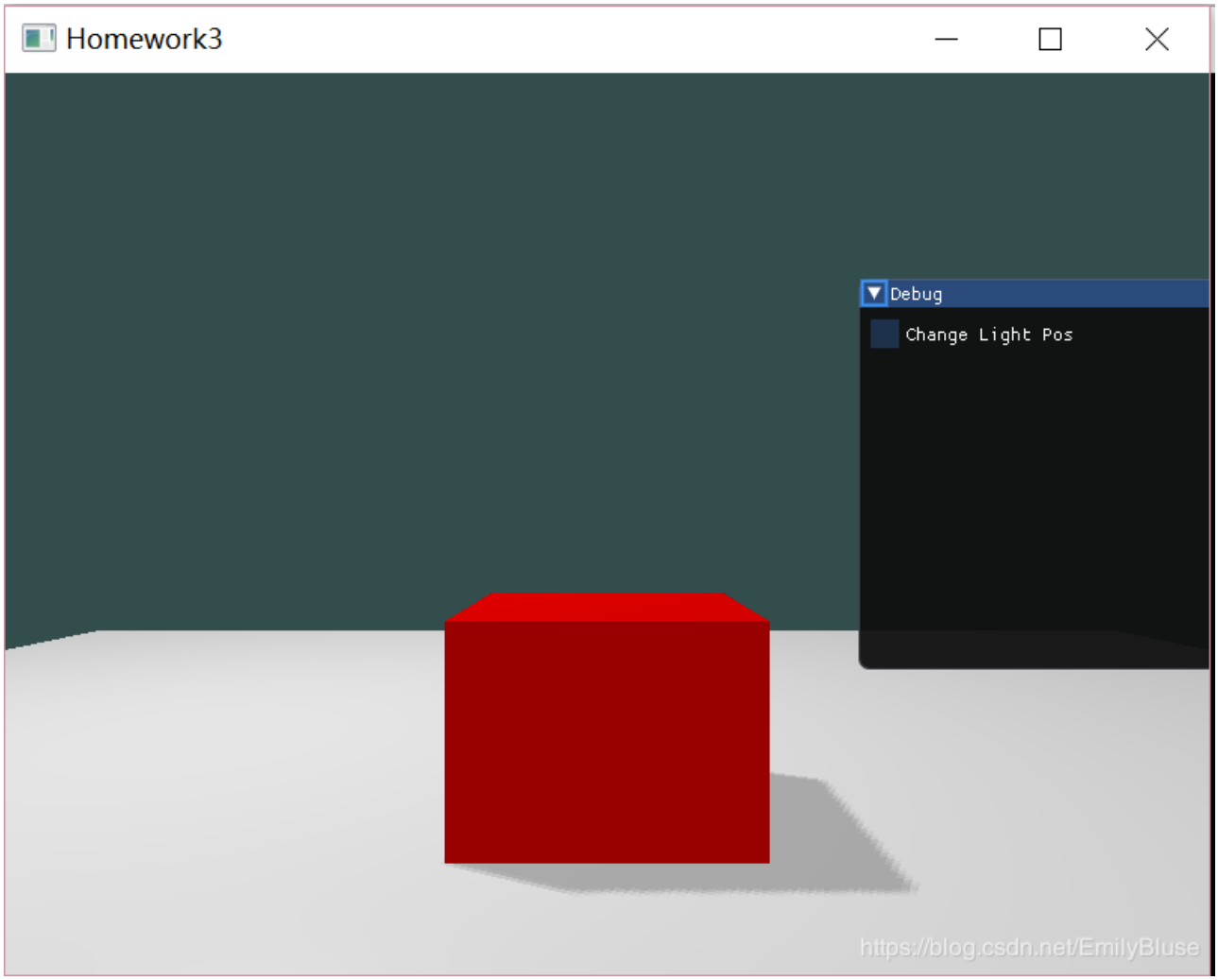
**解释Shadowing Mapping算法**

从上面的代码中，可以看到阴影算法，归根结底是要给出一个`ShadowCalculation`函数：首先要检查一

个片元是否在阴影中，把光空间片元位置转换为裁切空间的标准化设备坐标。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，所以需要做透视除法。因为来自深度贴图的深度在0到1的范围，我们也打算使用projCoords从深度贴图中去采样，所以我们将NDC坐标变换为0到1的范围。有了这些投影坐标，我们就能从深度贴图中采样得到0到1的结果，从第一个渲染阶段的projCoords坐标直接对应于变换过的NDC坐标。我们将得到光的位置视野下最近的深度。为了得到片元的当前深度，需要获取投影向量的z坐标，它等于来自光的透视视角的片元的深度。实际的对比就是简单检查currentDepth是否高于closetDepth，如果是，那么片元就在阴影中。

- 在渲染中：
  我们需要先计算光照模型，然后渲染出平面和立方体，使用深度贴图，计算阴影并渲染出来，就可以得到最后的阴影模型。
  在我的实现过程中，我使用了键盘控制摄像机位置，通过GUI也可以选择光源位置是否可动。这里用到了ImGui。

  实现的效果如下：




  动态图请见附件中mp4文件。