韩宇潇 16340069

整体代码结构：

```
class HW7
{
public:
    HW7(GLuint, GLuint);
    ~HW7();
    void HW7LinkShader();
    unsigned int getShaderProgram();
    void draw();
    void shadow(Camera, float, float, float);
    void RenderScene(Shader &);
    void RenderCube();
    void RenderQuad();
    unsigned int loadTexture(char const * path);

private:
    unsigned int planeVAO, planeVBO, cubeVAO = 0, cubeVBO, quadVAO = 0, quadVBO;
    GLuint depthMapFBO;
    GLuint depthMap;
    const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
    GLuint SCR_WIDTH, SCR_HEIGHT;
    Shader simpleDepthShader, debugDepthQuad, shader;
    unsigned int woodTexture;
};
```

HW7LinkShader():

用来构建着色器，将在本次作业中用到的顶点着色器和片段着色器连接，在接下来的调用过程中方便使用。在本次作业中构建了 3 个着色器。

getShaderProgram():

返回构建的着色器到 main 函数。

draw():

用于将画平面的数组存入缓存中，方便在接下来的画平面使用。导入图片。实现深度贴图。

shadow():

渲染深度贴图和场景。

RenderScene():

渲染场景，包括画平面和立方体。

RenderCube():

用于将画立方体的数组存入缓存中，进行立方体的渲染。

RenderQuad():

用于将画平面的数组存入缓存中，进行平面的渲染。

loadTexture():

导入图片。

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

实验思路：

Shadowing Mapping 算法：

我们以光的位置为视角进行渲染，我们能看到的东西都将被点亮，看不见的一定是在阴影之中了。假设有一个地板，在光源和它之间有一个大盒子。由于光源处向光线方向看去，可以看到这个盒子，但看不到地板的一部分，这部分就应该在阴影中了。简单的方法是对从光源发出的射线上的成千上万点进行遍历，为了避免这种极端消耗性能的举措，采用相似的方法深度缓冲，使用从光源的透视图来渲染场景，把深度值的结果储存到纹理中，形成深度贴图。在实现过程中，先渲染深度贴图，在渲染场景，使用生成的深度贴图来计算片元是否在阴影之中。

先在 draw()函数中实现深度贴图，从而实现深度缓冲。

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

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);
```

实现场景中各个部分的渲染，平面，正方体，图片导入等。

```
void HW7::RenderScene(Shader &shader){ ... }

void HW7::RenderCube(){ ... }

void HW7::RenderQuad(){ ... }

unsigned int HW7::loadTexture(char const * path){ ... }
```

构建相应的顶点着色器，片段着色器。

实现使用光的透视图进行场景渲染，只需要把顶点变换到光空间，实现过程简单。

顶点着色器：

```
#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：

```
#version 330 core

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

将深度贴图渲染到正方体上，顶点着色器不需要处理，片段着色器需要将深度贴图渲染到四边形上，实现方法有正交/透视两种投影（实现 Bonus1 要求）。

顶点着色器：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

void main()
{
    TexCoords = aTexCoords;
    gl_Position = vec4(aPos, 1.0);
}
```

片段着色器：

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D depthMap;
uniform float near_plane;
uniform float far_plane;

// required when using a perspective projection matrix
float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane - near_plane));
}

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    //FragColor = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // perspective
    FragColor = vec4(vec3(depthValue), 1.0); // orthographic
}
```

实现深度贴图的渲染：

```
GLfloat near_plane = 1.0f, far_plane = 7.5f;
glm::mat4 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), glm::vec3(1.0));
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
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);
```

使用深度贴图渲染场景：

```
debugDepthQuad.use();
debugDepthQuad.setFloat("near_plane", near_plane);
debugDepthQuad.setFloat("far_plane", far_plane);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, depthMap);
//RenderQuad();
```

渲染阴影，在顶点着色器中进行光空间的变换，在片段着色器中使用 Blinn-Phong 光照模型渲染场景，当处于阴影中，diffuse 和 specular 会为零。判断是否处于阴影，使用透视除法，获取最近点的深度，和当前片元在光源视角下的深度进行比较，判断是否处于阴影中。

顶点着色器：

```
#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);
}
```

片段着色器：渲染场景

```
void main() {
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.3);
    // ambient
    vec3 ambient = 0.15 * 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, normal, lightDir);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

    FragColor = vec4(lighting, 1.0);
}
```

判断是否处于阴影中，添加了阴影的优化（实现 Bonus2 要求）：

通过阴影偏移解决了阴影失真：

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

shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
```

解决采样过多：

```
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);
GLfloat borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

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

通过 PCF 解决锯齿边问题：

```
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;
```

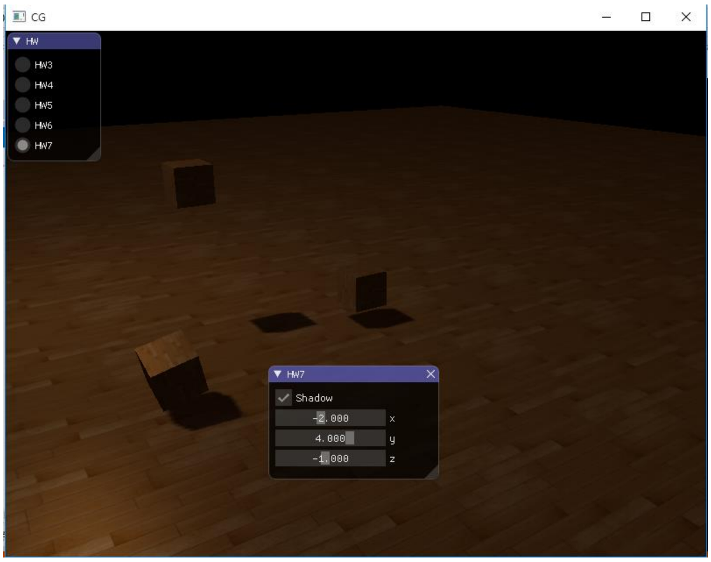
判断是否处于阴影中：

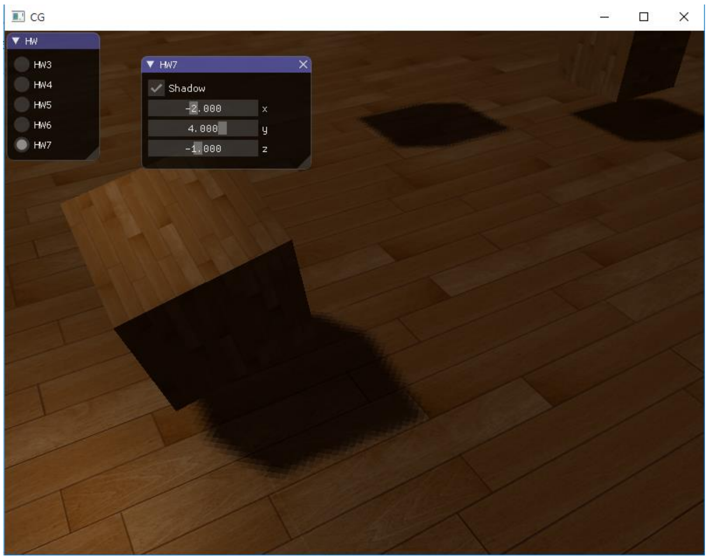
```
float ShadowCalculation(vec4 fragPosLightSpace, vec3 normal, vec3 lightDir) {

    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // check whether current frag pos is in shadow
    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;
}
```

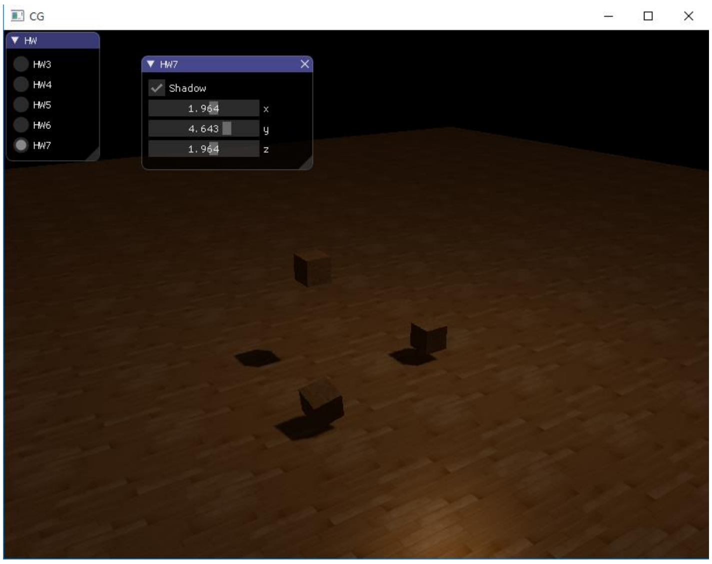
渲染阴影：

```
shader.use();
glm::mat4 projection = glm::perspective(glm::radians(camera.getZoom()), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
shader.setMat4("projection", projection);
shader.setMat4("view", view);
// set light uniforms
shader.setVec3("viewPos", camera.getPosition());
shader.setVec3("lightPos", lightPos);
shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, woodTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene(shader);
```

实验结果：


CG
HW
HW3
HW4
HW5
HW6
HW7
HW7
Shadow
-2.000
x
4.000
y
-1.000
z


CG
HW
HW3
HW4
HW5
HW6
HW7
HW7
Shadow
-2.000
x
4.000
y
-1.000
z


CG
HW
HW3
HW4
HW5
HW6
HW7
HW7
Shadow
1.964
x
4.643
y
1.964
z

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)
实现过程在 Basic 中说明。