作业二：光照与纹理映射

一、 问题 1：球面三角化与地球的纹理贴图

1.1 球面三角化

首先将创造球面三角化参数，生成一个可以在 OpenGL 中渲染的球体，通过外层遍历纬线将纬线索引映射到 $[0,\pi]$, 内层遍历经线，将经线索引映射到 $[0，2\pi]$，得到顶点的三维坐标，然后利用纬线和经线的顶点索引得到三角形的索引。最终得到顶点的三维坐标、顶点的纹理坐标和三角形索引.

1.2 create sphere 代码

```
def create_sphere(radius, segments, rings):
   vertices = []
   indices = []
   for i in range(rings + 1):
      theta = i * np.pi / rings
      sin_theta = np.sin(theta)
      cos_theta = np.cos(theta)

      for j in range(segments + 1):
         phi = j * 2 * np.pi / segments
         sin_phi = np.sin(phi)
         cos_phi = np.cos(phi)

         x = -cos_phi * sin_theta
         y = -cos_theta
         z = sin_phi * sin_theta

         # 计算纹理坐标
         u = j / segments
         v = i / rings

         vertices.extend([radius * x, radius * y, radius * z, x, y, z, u, v])

         if i < rings and j < segments:
         a = i * (segments + 1) + j
         b = a + segments + 1
         indices.extend([a, b, a + 1, b, b + 1, a + 1])

   return np.array(vertices, dtype=np.float32), np.array(indices, dtype=np.uint32)
```

1.3 纹理贴图

1.3.1 纹理加载

加载纹理图片生成 OpenGL 的纹理对象，并使用 glTexImage2D 创建纹理，代码如下（为方便后续使用此处已进行纹理过滤并生成 Mipmap):

```python
def load_texture(filename):
   image = Image.open(filename)
   image = image.transpose(Image.FLIP_TOP_BOTTOM)
   image = image.convert("RGB")
   img_data = image.tobytes()

   texture = glGenTextures(1)
   glBindTexture(GL_TEXTURE_2D, texture)

   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image.width, image.height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, img_data)
   glGenerateMipmap(GL_TEXTURE_2D)

   # 设置纹理参数
   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_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR)
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)

   return texture
```

后续利用着色器程序绑定纹理进行绘制即可。得到图像如下（在程序中生成的是可以旋转的动图，此处取亚非大陆与美洲大陆两帧，且为方便后续使用，此处已实现 Blinn-Phong 模型）：

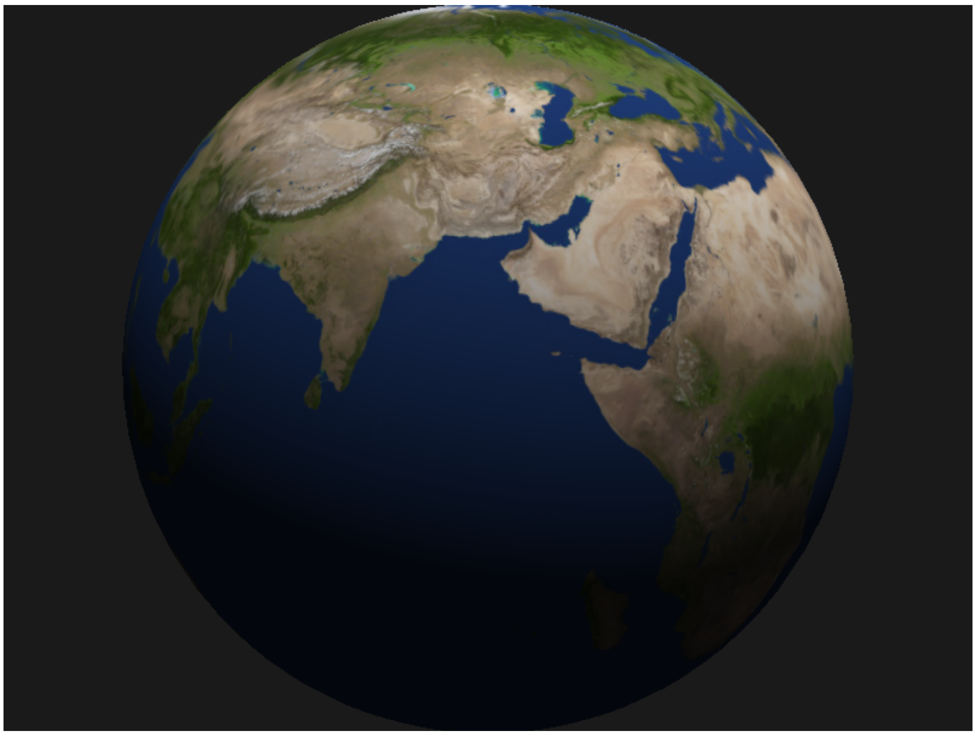
图 1　**Asia Africa**

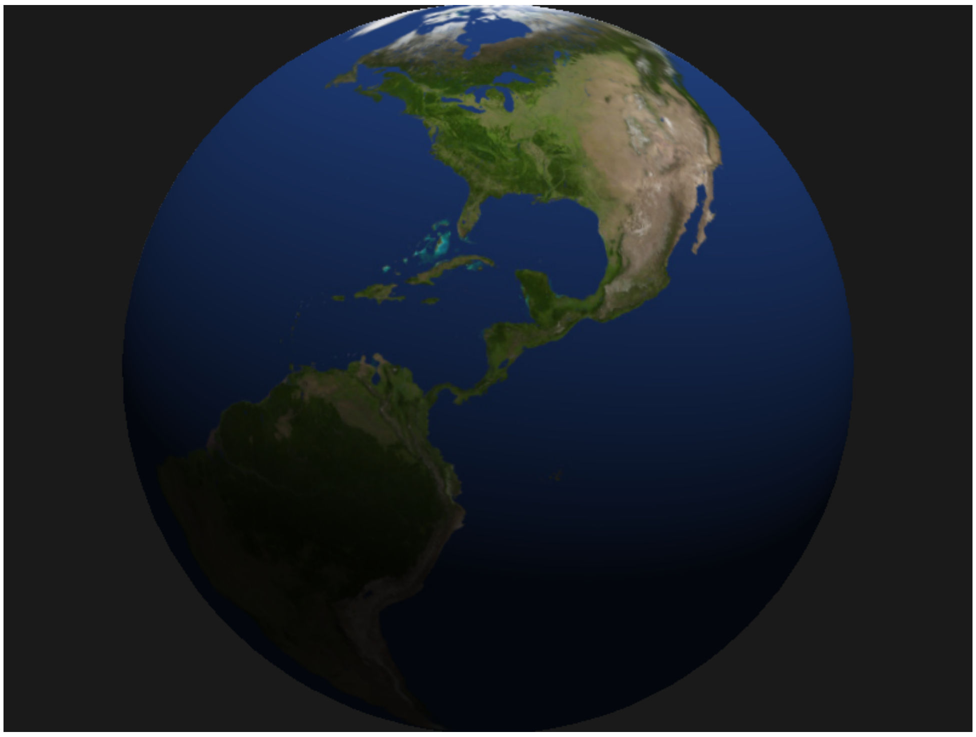
图 2　**America**

二、 问题 2：纹理贴图反走样

2.1 实现过程

1. 使用 MIPMAP

```
glGenerateMipmap(GL_TEXTURE_2D)
```

2. 使用线性过滤

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR)
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
```

3. 启用各向异性过滤

```
max_anisotropy = glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY, max_anisotropy)
```

4. 启用抗锯齿功能

```
glEnable(GL_POLYGON_SMOOTH)
glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST)
```

5. 启用多重采样

```
glutSetOption(GLUT_MULTISAMPLE, 8)
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH | GLUT_MULTISAMPLE )
```

效果图如下所示：

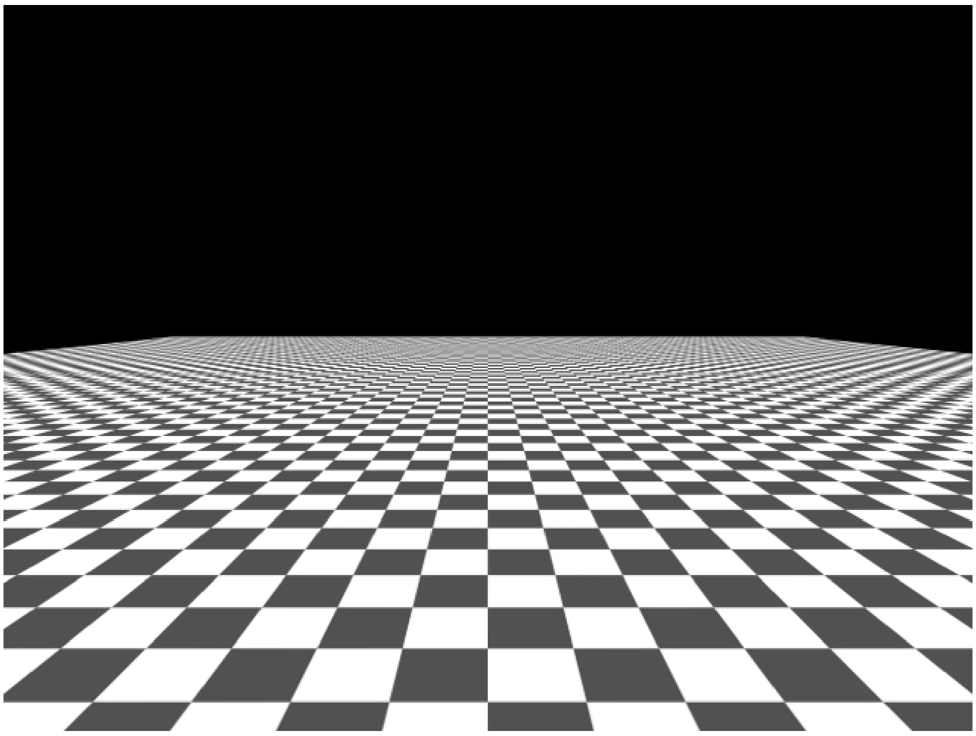
图 3　**chessboard**

三、 问题 3：Blinn-Phong 模型

3.1 Blinn-Phong

几何使用球面，球面创建过程见问题 1。为实现 Blinn-Phong 模型，要对片段着色器代码进行修改。

1.Ambient Lighting：$I_{ambient} = k_a * I_{light}$,$k_a$ 为环境光反射系数，这里取 0.1，I_{light} 为环境光源的颜色.

2.Diffuse Lighting:$I_{diffuse} = k_d * I_{light} * max(N \cdot L, 0)$,$k_a$ 为漫反射系数，这里取 1，N 为表面法线，L 为光源方向

3.Specular Lighting：$I_{specular} = k_s \cdot I_{light} \cdot max(N \cdot H, 0)^{\alpha}$，$k_a$ 为镜面反射系数，这里取 0.5，N 为表面法线，α 为光滑度，这里取 32，L 为半角向量。

半角向量 H= $\frac{V+L}{|V+L|}$，其中 V 为视线方向，L 为光源方向，Blinn-Phong 中以此向量代替反射向量以简化计算

片段着色器代码如下：

```
FRAGMENT_SHADER = """
#version 430


in vec3 FragPos;
in vec3 VertexNormal;
```

```
out vec4 FragColor;

uniform vec3 lightPos;
uniform vec3 lightColor;
uniform vec3 viewPos;
uniform vec3 objectColor;
uniform float samplingFactor;

void main()
{
   // 模拟像素采样位置调整
   vec2 pixelPosition = gl_FragCoord.xy * samplingFactor;

   // 法线和光照计算
   vec3 norm = normalize(VertexNormal);
   vec3 lightDir = normalize(lightPos - FragPos);

   // 环境光
   float ambientStrength = 0.1;
   vec3 ambient = ambientStrength * lightColor;

   // 漫反射
   float diff = max(dot(norm, lightDir), 0.0);
   vec3 diffuse = diff * lightColor;

   // 镜面反射
   float specularStrength = 0.5;
   vec3 viewDir = normalize(viewPos - FragPos);
   vec3 reflectDir = reflect(-lightDir, norm);
   float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
   vec3 specular = specularStrength * spec * lightColor;

   vec3 result = (ambient + diffuse + specular) * objectColor;

   FragColor = vec4(result, 1.0);
}
"""
```

然后在主函数中初始化并绘制即可。

3.2 采样频率

1. 顶点：在模型的顶点处进行光照计算。
2. 三角形：基于顶点进行插值计算。
3. 像素：在每个像素点进行光照计算。

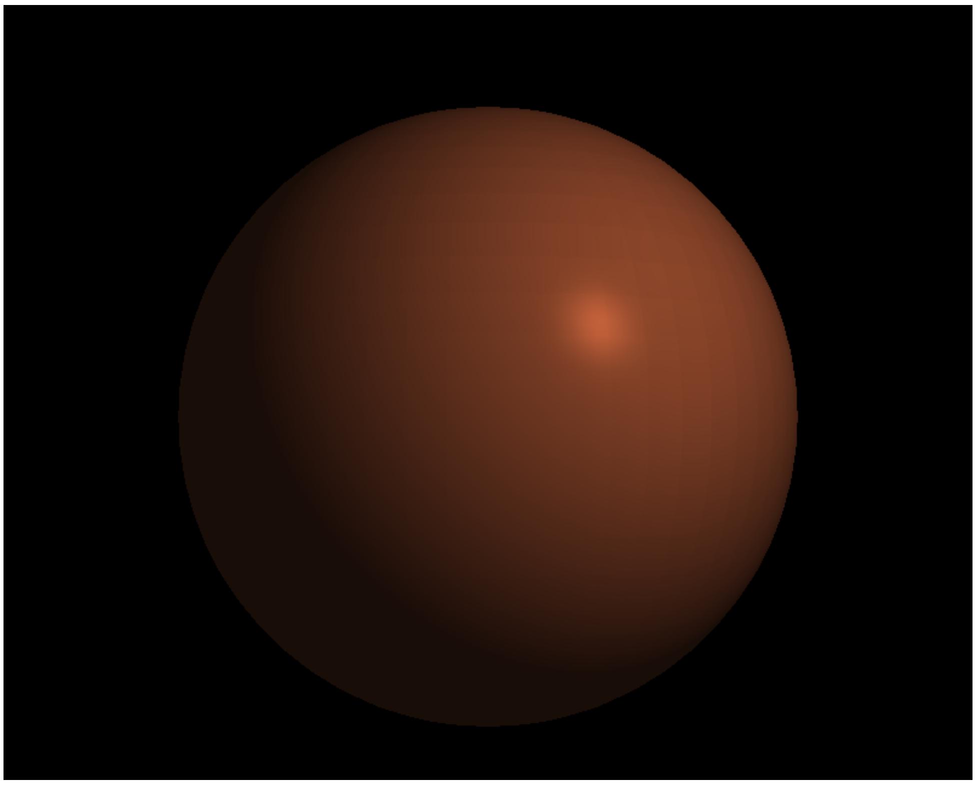

图 4　**vertex**

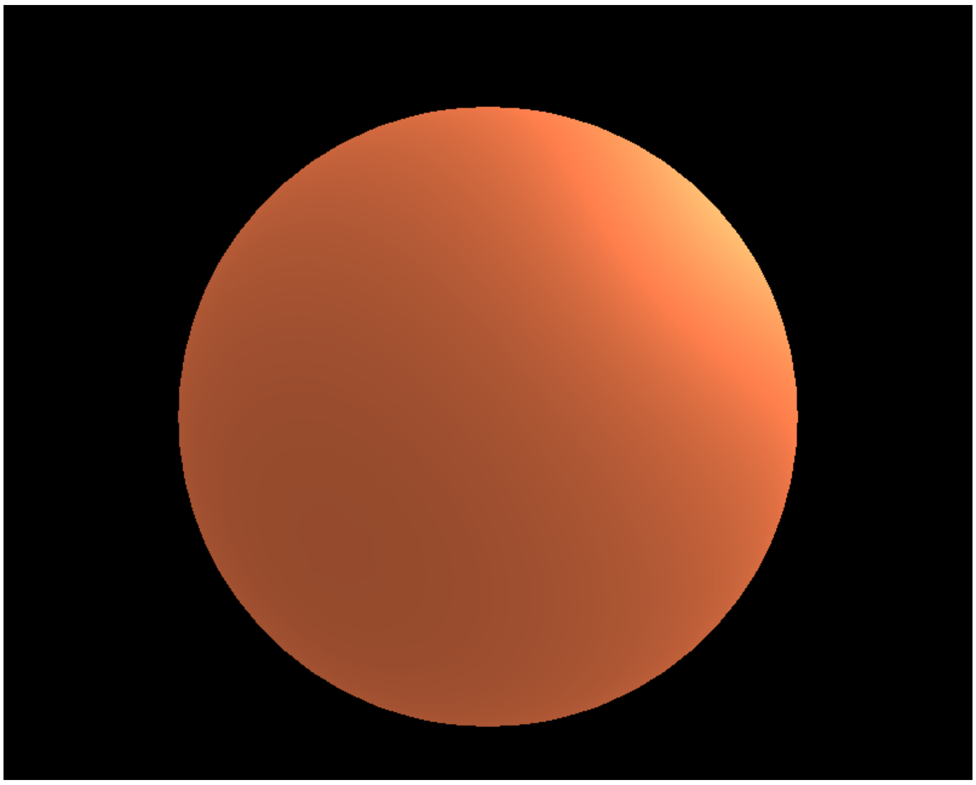

图 5　**triangle**

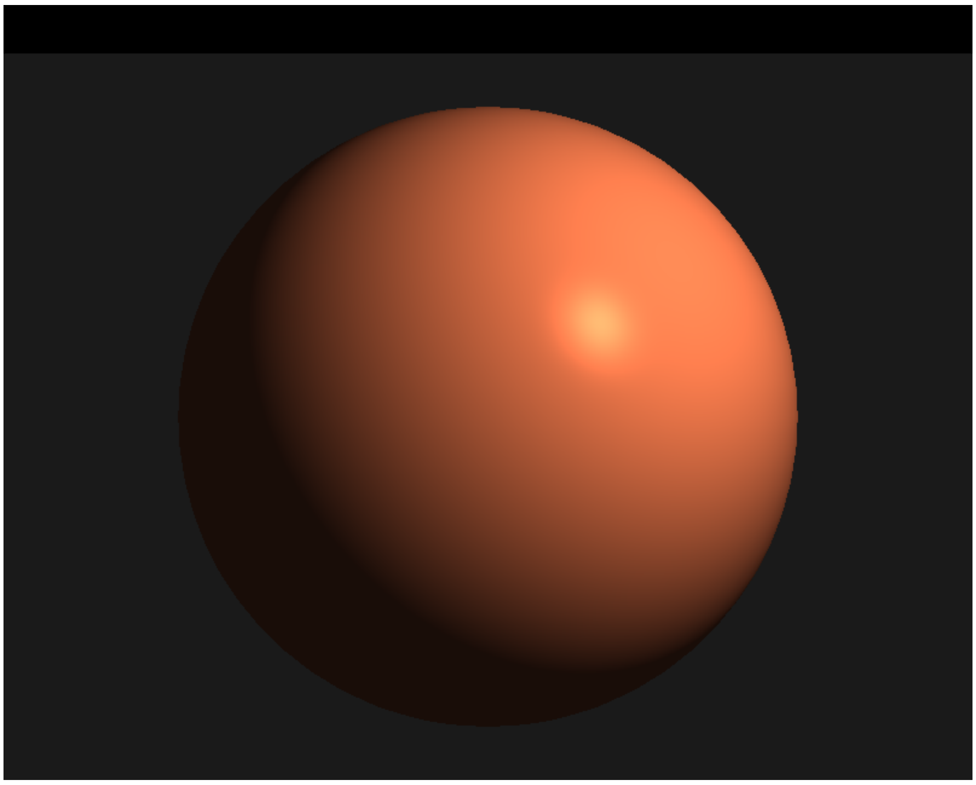

图 6　**pixel**

四、 问题 4：Bump Mapping

在问题 1 的基础上稍作修改即可。

在片段着色器中，提取凹凸贴图的高度信息，并计算法线扰动，其公式如下：

$$n^{'} = n + \frac{B_u(n \times P_v) - B_v(n \times P_u)}{||n||}$$

在着色器中计算法线扰动代码如下：

```
FRAGMENT_SHADER = """
#version 430
uniform sampler2D tex0; // Diffuse texture
uniform sampler2D tex1; // Bump map
uniform vec3 lightPos;
uniform vec3 lightColor;
uniform vec3 eyePos;

in vec3 fragPos;
in vec3 vt_norm;
in vec3 vt_T;
in vec2 vt_texcoord;

void main() {
    vec3 vt_B = cross(vt_norm, vt_T); // Bitangent vector
    float bumpScale = 0.01;
    float delta = 0.001;

    float bump = texture(tex1, vt_texcoord).r;
    float left = texture(tex1, vt_texcoord + vec2(-delta, 0.0)).r * bumpScale;
    float right = texture(tex1, vt_texcoord + vec2(delta, 0.0)).r * bumpScale;
    float up = texture(tex1, vt_texcoord + vec2(0.0, delta)).r * bumpScale;
    float down = texture(tex1, vt_texcoord + vec2(0.0, -delta)).r * bumpScale;

    float du = (left - right) / (2.0 * delta);
    float dv = (down - up) / (2.0 * delta);

    float bumpFactor = 0.5;
    vec3 adjustedNormal = vt_norm + bumpFactor * (du * vt_B - dv * vt_T);
    adjustedNormal = normalize(adjustedNormal);

    vec3 ambient = 0.1 * lightColor;

    vec3 lightDir = normalize(lightPos - fragPos);
    vec3 viewDir = normalize(eyePos - fragPos);
    vec3 reflectDir = reflect(-lightDir, adjustedNormal);

    float diff = max(dot(adjustedNormal, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
    vec3 specular = 0.5 * spec * lightColor;

    vec3 textureColor = texture(tex0, vt_texcoord).rgb;
    vec3 result = (ambient + diffuse + specular) * textureColor;

    gl_FragColor = vec4(result, 1.0);
}
"""
```

其效果图如图所示：

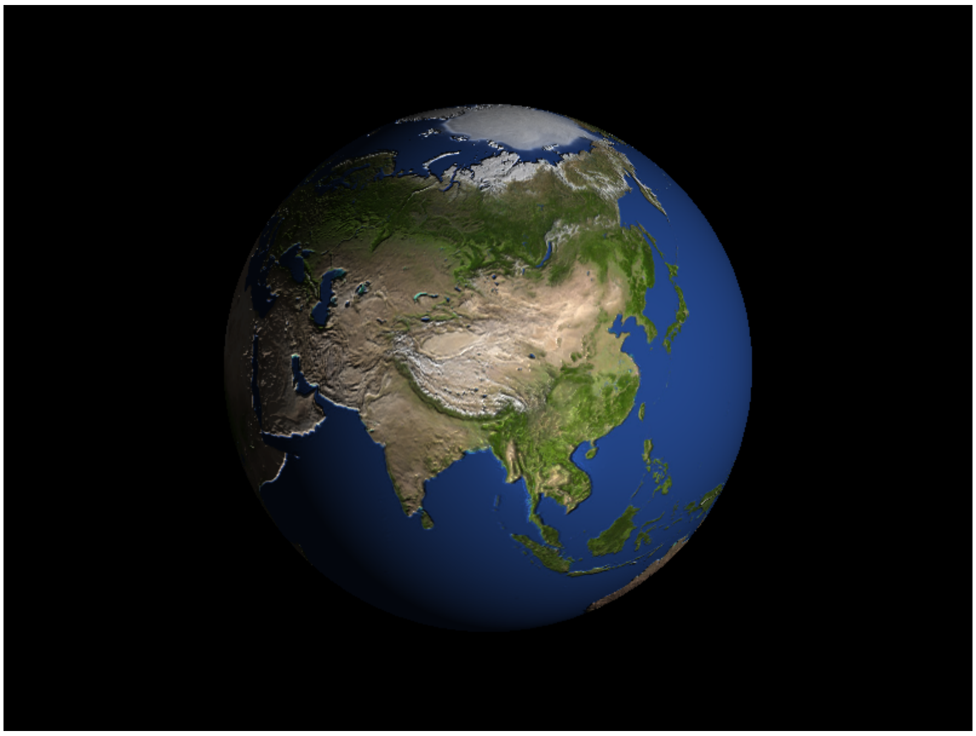
图 7　Shadow

五、 问题 5：Shadow Mapping

首先绘制平面，代码如下：

```
def plane(width=10.0, height=10.0):
  points = [
  -width/2, -1, -height/2,
  width/2, -1, -height/2,
  width/2, -1, height/2,
  -width/2, -1, height/2
  ]
  texcoords = [0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0]
  norms = [0.0, 1.0, 0.0] * 4
  indices = [0, 1, 2, 0, 2, 3]

  return np.array(points, np.float32), np.array(texcoords, np.float32), np.array(norms,
      np.float32), np.array(indices, np.uint32)
```

创建两个片段着色器，分别用于光源视角和摄像机视角的渲染。

光源视角的片段着色器需要建立 Blinn-Phong 模型的光照并且计算阴影因子。该片段着色器代码如下：

```
FRAGMENT_SHADER1 = """
#version 430
```

```
uniform vec3 lightPos; // 光源在世界坐标下的位置
uniform vec3 lightColor; // 光源的颜色
uniform bool isShadowPass; // 控制是否是阴影传递
uniform vec3 materialSpecular;
uniform vec3 lightSpecular;

in vec3 fragPosition; // 片段位置
in vec3 fragNormal; // 片段法线
in vec2 vt_texcoord; // 片段纹理坐标
in vec4 ShadowCoords; // 片段在光源空间中的坐标

out vec4 FragColor;



void main() {

  if (isShadowPass) {
    // 不需要输出颜色，因为我们在渲染深度值
    discard;
  } else {
    vec3 norm = normalize(fragNormal);
    vec3 lightDir = normalize(lightPos - fragPosition);
    float distance = length(lightPos - fragPosition);
    float attenuation = 1.0 / (distance * distance);

    // 环境光
    float ambientStrength = 0.01;
    vec3 ambient = ambientStrength * lightColor;

    // 漫反射
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = lightColor * attenuation * diff;

    vec3 kd = vec3(1, 1, 1); // 设置一个简单的颜色

    // Specular
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(lightDir, reflectDir), 0.0), 32);
    vec3 specular = lightSpecular * (spec * materialSpecular);



    vec3 color = ambient + kd * diffuse + specular;

    // 计算阴影因子
    //float shadow = ShadowCalculation(ShadowCoords);
    // 应用阴影
    //color *= (1.0 - shadow); // 如果片段在阴影中, 则降低亮度
    FragColor = vec4(color, 1.0);
  }
}
"""
```

摄像机视角的片段着色器则是在以上基础上加上了一个阴影计算函数，通过获取深度计算一个片段是否在阴影中。阴影计算函数代码如下：

```
  float ShadowCalculation(vec4 fragPosLightSpace)
  {
    // 执行透视除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;

    // 将投影坐标从 [-1,1] 映射到 [0,1]
    projCoords = projCoords * 0.5 + 0.5;

    // 获取最近片段的深度（使用xy分量）
    float closestDepth = texture(shadowMap, projCoords.xy).r;

    // 获取当前片段的深度
    float currentDepth = projCoords.z;

    // 检查当前片段是否在阴影中
    float bias = 0.005;
    float shadow = currentDepth - bias > closestDepth? 1.0 : 0.0;
    //shadow = closestDepth==1? 1.0 : 0.0;

    return shadow;
  }
```

先以光源的视角渲染，计算投影和视图矩阵，并将矩阵将传递到着色器中，然后进行阴影传递，再传递光源位置和颜色到着色器中，最后绘制场景。

接下来从摄像机视角渲染，步骤与光源视角类似。

渲染代码如下：

```
  def render():
    glClearColor(0, 0, 0, 1)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glEnable(GL_DEPTH_TEST)
    glDepthFunc(GL_LESS)
    glUseProgram(shaderProgram1)
    # 绑定深度贴图帧缓冲区
    glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO)
```

```
glViewport(0, 0, width, length)
glClear(GL_DEPTH_BUFFER_BIT)
# 设置光源视角的投影和视图矩阵
lm_mat, lmvp_mat=calc_lmvp(width, length)
light_space_matrix =lmvp_mat
# 从光源视角渲染场景
mvp_loc = glGetUniformLocation(shaderProgram1, "MVP")
m_loc = glGetUniformLocation(shaderProgram1, "M")
glUniformMatrix4fv(mvp_loc, 1, GL_FALSE, glm.value_ptr(lmvp_mat))
glUniformMatrix4fv(m_loc, 1, GL_FALSE, glm.value_ptr(lm_mat))
light_space_loc = glGetUniformLocation(shaderProgram1, "lightSpaceMatrix")
glUniformMatrix4fv(light_space_loc, 1, GL_FALSE, glm.value_ptr(light_space_matrix))
glUniform1i(glGetUniformLocation(shaderProgram1, "isShadowPass"), 0) # 设置为阴影传递
# 传递光源位置和颜色
lightPos = np.array([3.0, 4.0, 0.0], np.float32)
lightColor = np.array([17.0, 17.0, 17.0], np.float32)
glUniform3f(glGetUniformLocation(shaderProgram1, "lightPos"), lightPos[0], lightPos[1],
    lightPos[2])
glUniform3f(glGetUniformLocation(shaderProgram1, "lightColor"), lightColor[0],
    lightColor[1], lightColor[2])
# 绘制球体
model = glm.mat4(1.0)
model_loc = glGetUniformLocation(shaderProgram1, "model")
glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm.value_ptr(model))
glBindVertexArray(VAO)
glDrawElements(GL_TRIANGLES, Num_T, GL_UNSIGNED_INT, None)
# 绘制平面
model = glm.translate(glm.mat4(1.0), glm.vec3(0, -1.0, 0))
glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm.value_ptr(model))
glBindVertexArray(plane_VAO)
glDrawElements(GL_TRIANGLES, Num_Planes, GL_UNSIGNED_INT, None)
glUseProgram(0)
# 绑定默认帧缓冲区
glBindFramebuffer(GL_FRAMEBUFFER, 0)
glViewport(0, 0, width, length)
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
# 从摄像机视角渲染场景
glUseProgram(shaderProgram)
mvp_loc = glGetUniformLocation(shaderProgram, "MVP")
m_loc = glGetUniformLocation(shaderProgram, "M")
m_mat, mvp_mat = calc_mvp(800, 600)
glUniformMatrix4fv(mvp_loc, 1, GL_FALSE, glm.value_ptr(mvp_mat))
glUniformMatrix4fv(m_loc, 1, GL_FALSE, glm.value_ptr(m_mat))
light_space_loc = glGetUniformLocation(shaderProgram, "lightSpaceMatrix")
glUniformMatrix4fv(light_space_loc, 1, GL_FALSE, glm.value_ptr(light_space_matrix))
glUniform1i(glGetUniformLocation(shaderProgram, "isShadowPass"), 0)
# 传递光源位置和颜色
```

```
lightPos = np.array([3.0, 4.0, 0.0], np.float32)
lightColor = np.array([17.0, 17.0, 17.0], np.float32)
glUniform3f(glGetUniformLocation(shaderProgram, "lightPos"), lightPos[0], lightPos[1],
    lightPos[2])
glUniform3f(glGetUniformLocation(shaderProgram, "lightColor"), lightColor[0],
    lightColor[1], lightColor[2])
# 传递深度贴图
glActiveTexture(GL_TEXTURE0)
glBindTexture(GL_TEXTURE_2D, depthMap)
glUniform1i(glGetUniformLocation(shaderProgram, "shadowMap"), 0)
# 设置为非阴影传递
# 绘制球体
model = glm.mat4(1.0)
glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm.value_ptr(model))
glBindVertexArray(VAO)
glDrawElements(GL_TRIANGLES, Num_T, GL_UNSIGNED_INT, None)
# 绘制平面
model = glm.translate(glm.mat4(1.0), glm.vec3(0, -1.0, 0))
glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm.value_ptr(model))
glBindVertexArray(plane_VAO)
glDrawElements(GL_TRIANGLES, Num_Planes, GL_UNSIGNED_INT, None)
glUseProgram(0)
glutSwapBuffers()
```

最终得到效果图如下：

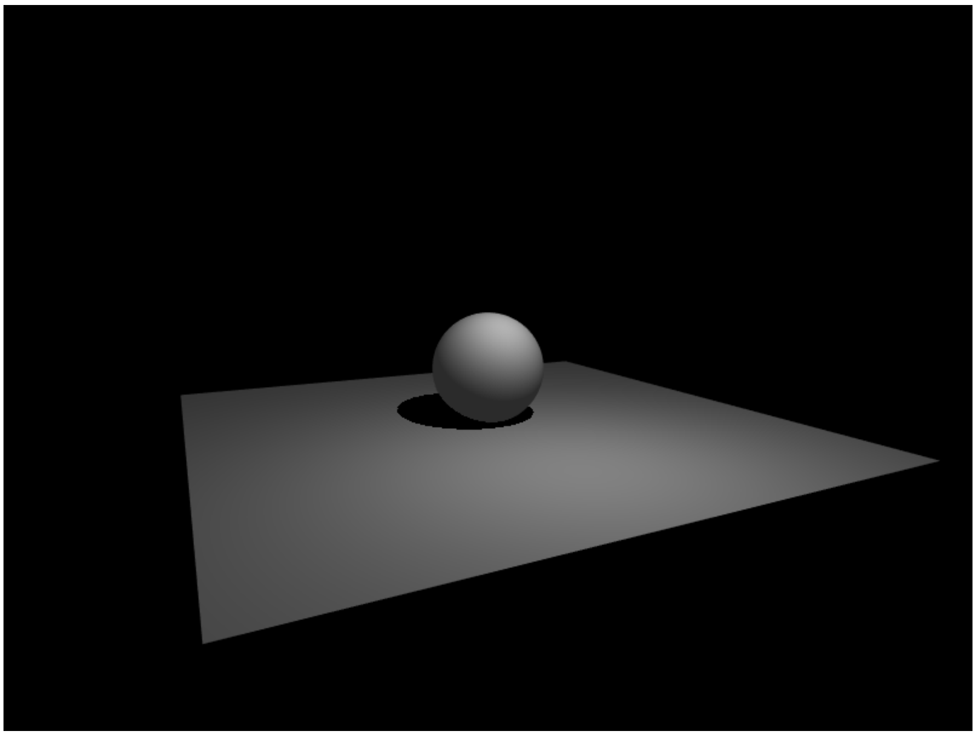

图 8　**Shadow**