计算机图形学实验报告

学号：16340054

姓名：戴馨乐

学院：数据科学与计算机学院

作业：第四次作业

Basic:

1. **画一个立方体(cube)：边长为 4， 中心位置为(0, 0, 0)。分别启动和关闭深度测试 `glEnable(GL_DEPTH_TEST)` 、 `glDisable(GL_DEPTH_TEST)` ，查看区别，并分析原因。**

   实现思路：

   渲染一个立方体的方法并不难，只需要修改顶点数组 vertices，将立方体所需要的顶点添加进数组即可。这里画的是一个正方体，一共 36 个顶点。OpenGL 画立方体同样是通过画三角形的方式来画，用一个个三角形拼凑出立方体。

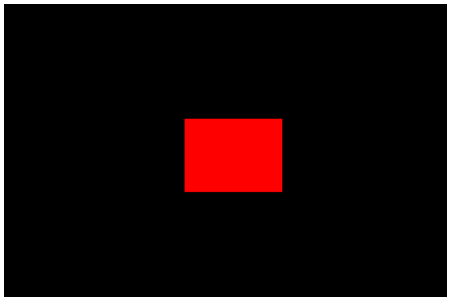
   顶点数组：

```
float vertices[] = {
// 后面
    -0.1f, -0.1f, -0.1f,  1.0f, 0.0f, 0.0f,
     0.1f, -0.1f, -0.1f,  1.0f, 0.0f, 0.0f,
     0.1f,  0.1f, -0.1f,  1.0f, 0.0f, 0.0f,
     0.1f,  0.1f, -0.1f,  1.0f, 0.0f, 0.0f,
    -0.1f,  0.1f, -0.1f,  1.0f, 0.0f, 0.0f,
    -0.1f, -0.1f, -0.1f,  1.0f, 0.0f, 0.0f,
// 前面
    -0.1f, -0.1f,  0.1f,  1.0f, 0.0f, 0.0f,
     0.1f, -0.1f,  0.1f,  1.0f, 0.0f, 0.0f,
     0.1f,  0.1f,  0.1f,  1.0f, 0.0f, 0.0f,
     0.1f,  0.1f,  0.1f,  1.0f, 0.0f, 0.0f,
    -0.1f,  0.1f,  0.1f,  1.0f, 0.0f, 0.0f,
    -0.1f, -0.1f,  0.1f,  1.0f, 0.0f, 0.0f,
// 左面
    -0.1f,  0.1f,  0.1f,  0.0f, 1.0f, 0.0f,
    -0.1f,  0.1f, -0.1f,  0.0f, 1.0f, 0.0f,
    -0.1f, -0.1f, -0.1f,  0.0f, 1.0f, 0.0f,
    -0.1f, -0.1f, -0.1f,  0.0f, 1.0f, 0.0f,
    -0.1f, -0.1f,  0.1f,  0.0f, 1.0f, 0.0f,
    -0.1f,  0.1f,  0.1f,  0.0f, 1.0f, 0.0f,
// 右面
     0.1f,  0.1f,  0.1f,  0.0f, 1.0f, 0.0f,
     0.1f,  0.1f, -0.1f,  0.0f, 1.0f, 0.0f,
     0.1f, -0.1f, -0.1f,  0.0f, 1.0f, 0.0f,
     0.1f, -0.1f, -0.1f,  0.0f, 1.0f, 0.0f,
```

   画立方体：

```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

   最后，得到的结果：



然而看到的却是一个 2D 的正方形。这个原因是我们没有调整视角的原因，目前是正视这个正方体，看到的是它的正面，我设置是红色。
旋转下正方体调整下视角：
设置调整矩阵：

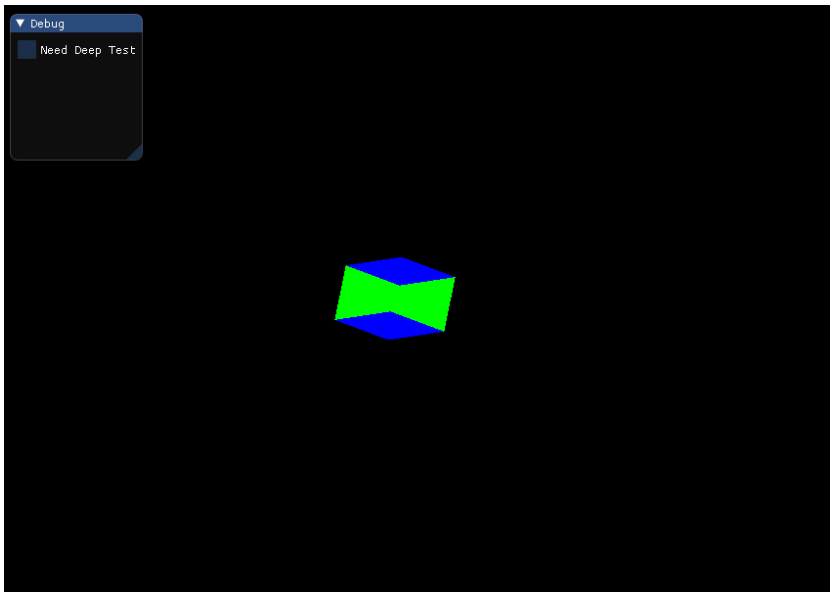
```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));
```

传入顶点着色器：

```
const char* vertexShaderCode = "#version 330 core\n"
    "layout(location = 0) in vec3 aPos;\n"
    "layout(location = 1) in vec3 aColor;\n"
    "out vec3 myColor;\n"
    "uniform mat4 model;\n"
    "uniform mat4 view;\n"
    "uniform mat4 projection;\n"
    "uniform mat4 transform;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = projection * view * model * vec4(aPos, 1.0f);\n"
    "   gl_Position = transform * gl_Position;\n"
    "   myColor = aColor;"
    "}\0";
```

```
glUniformMatrix4fv(glGetUniformLocation(MyShader.ID, "model"), 1, GL_FALSE, glm::value_ptr(model));
```

最后得到的结果是：



这里我们看到，这并不是我们熟悉的正方体，这个原因是没有开启深度测试。OpenGL 默认是关闭深度测试的，会导致上面的结果。

开启深度测试：

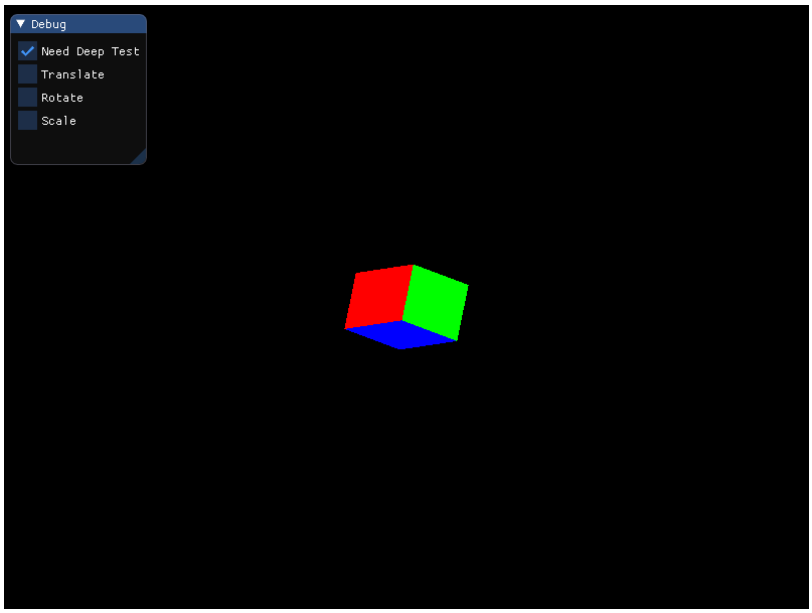
通过 glEnable( )或者 glDisable( )开启或者关闭指定的属性。GL_DEPTH_TEST 就是深度测试。

```
if (is_deep_test)
    glEnable(GL_DEPTH_TEST);
else
    glDisable(GL_DEPTH_TEST);
```

另外还需要再每次渲染迭代之前，清楚深度缓冲。

```
if (is_deep_test) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

得到的结果：



可以看到了这就是熟悉的正方体了。

问题分析：

为什么没有开启深度测试会出现那种情况？

OpenGL 将每个点的深度信息都存储在一个 Z 缓冲中。什么是深度信息？就是指示这个平面在上面还是在下面的信息，可以理解为通过比较两个位于同一位置的点的深度信息，决定哪个在上哪个在下。假如没有开启深度测试，那么就不会进行比较这些深度信息，而是简单的覆盖，所以导致渲染出了上面那种奇怪的正方体。而开启深度测试之后，OpenGL 会将片段的深度信息和 Z 缓冲进行比较，从而决定是丢弃还是覆盖。这样，就可以渲染出了一个我们肉眼常见的正方体了。

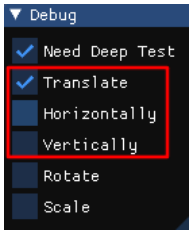
2. **在 GUI 里添加菜单栏，可以选择各种变换**
   a) **平移(Translation)：使画好的 cube 沿着水平或垂直方向来回移动。**
   b) **旋转(Rotation)：使画好的 cube 沿着 XoZ 平面的 x=z 轴持续旋转。**
   c) **放缩(Scaling)：使画好的 cube 持续放大缩小。**

GUI 工具栏展示说明：

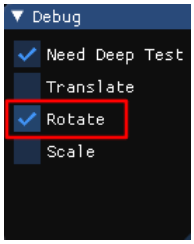
在开启深度测试的前提下：

进行平移：

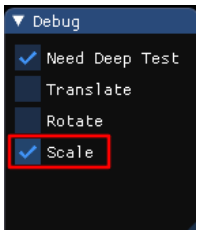
可以选择是水平来回移动还是垂直来回移动



进行旋转：



进行缩放：



变换的实现思路：

变换都是基于单位矩阵上来进行的。首先需要一个基础的4×4的单位矩阵：

```
glm::mat4 trans = glm::mat4(1.0f);
```

1. **平移：**

通过将位移向量$(T_x \quad T_y \quad T_z)$和单位矩阵$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$结合起来，得到需要的平移矩阵

$$\begin{bmatrix}1&0&0&T_x\\0&1&0&T_y\\0&0&1&T_z\\0&0&0&1\end{bmatrix}\cdot\begin{pmatrix}x\\y\\z\\1\end{pmatrix}=\begin{pmatrix}x+T_x\\y+T_y\\z+T_z\\1\end{pmatrix}$$

这个组合工作，OpenGL 提供了相应的 API 可供使用，通过 `glm::translate()`函数来进行组合。

不过需要实现的功能是可以来回移动，这需要当正方体移动到边缘的时候，可以改正方向来反向移动。

这里需要一个变量 `direction` 来记录方向：

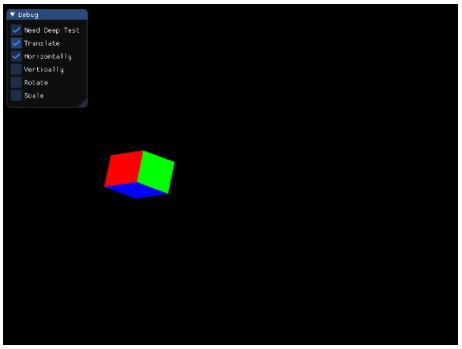
```cpp
bool direction = true;
/*
    true: 向上或者向右
    false: 向下或者向左
*/
```

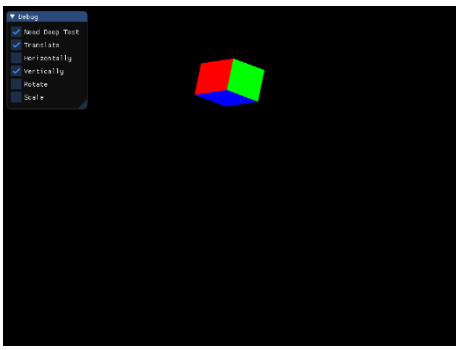
实现的逻辑是，使用一个变量 pre_step 来记录上一次的平移向量，然后这次的移动在上一次的移动的基础上，计算下一步的平移向量。在计算最后的平移矩阵前，我们还需要计算假如这么移动，是否到达了边缘，假如到达了需要修改方向，使得正方体可以来回移动。

```cpp
if (ishor) {
    if (pre_ishor != ishor) {
        pre_ishor = ishor;
        glm::vec3 pre_step = glm::vec3(0.0f, 0.0f, 0.0f);
    }
    pre_isver = isver = false;
    pre_ishor = true;
    glm::vec3 p_step = glm::vec3(0.05f, 0.0f, 0.0f), n_step = glm::vec3(-0.05f, 0.0f, 0.0f);
    glm::vec4 p_axis = glm::vec4(0.1f, 0.1f, 0.1f, 1.0f), n_axis = glm::vec4(-0.1f, -0.1f, -0.1f, 1.0f);
    if (direction) {
        glm::vec3 step = pre_step_h + p_step;
        glm::vec4 next = glm::translate(trans, step)*p_axis;
        if (next.x <= 1.0f) {
            trans = glm::translate(trans, step);
            pre_step_h = step;
        }
        else {
            step = pre_step_h + n_step;
            trans = glm::translate(trans, step);
            pre_step_h = step;
            direction = false;
        }
    }
    else {
        glm::vec3 step = pre_step_h + n_step;
        glm::vec4 next = glm::translate(trans, step)*n_axis;
        if (next.x >= -1.0f) {
            trans = glm::translate(trans, step);
            pre_step_h = step;
        }
        else {
            step = pre_step_h + p_step;
            trans = glm::translate(trans, step);
            pre_step_h = step;
            direction = true;
        }
    }
}
```

这是关于水平移动的实现，垂直移动同理。

实现结果：





2. 旋转：

旋转比起平移更加复杂一点，绕$(R_x \quad R_y \quad R_z)$轴旋转的旋转矩阵为：

$$\begin{bmatrix} \cos\theta + {R_x}^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + {R_y}^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + {R_z}^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

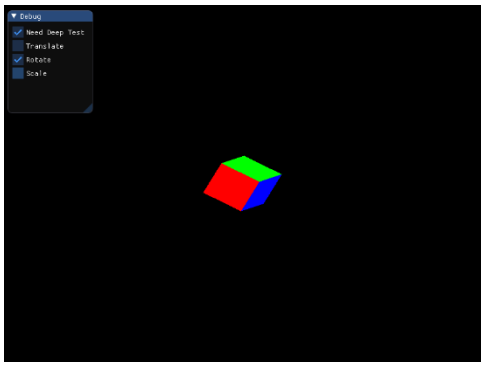
不过这个在 glm 库中，已经封装起来了，进行旋转只需要调用 glm::rotate( ) 函数，传入的参数有 3 个：

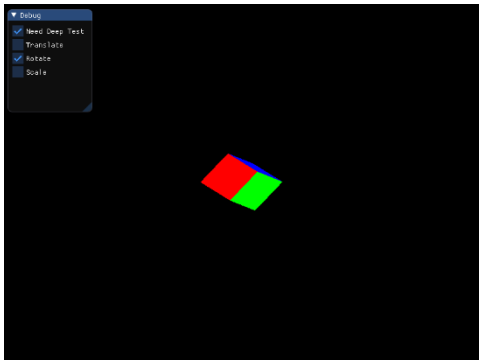
1. 单位矩阵
2. 旋转的角度（弧度制，可以通过 glm 库的 glm::radians( )函数进行转换）
3. 旋转的轴（vec3 的向量）

```
trans = glm::rotate(trans, (float)glfwGetTime()*glm::radians(50.0f), glm::vec3(1.0f, 0.0f, 1.0f));
```

这里让这个正方体绕着 x=z 的轴来进行旋转，每次旋转的角度是50^o。另外，为了让其保持旋转，通过 glfwGetTime( )来通过时间来更新角度。

运行结果：





3. 缩放：

缩放和平移一样，也是通过一个缩放向量$(S_1 \quad S_2 \quad S_3)$和单位矩阵结合得到

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

这个在 OpenGL 中同样有相应的 API 来帮助我们组合，使用 glm::scale()函数来组合，传入的参数是单位矩阵和缩放变量

```
trans = glm::scale(trans, tmp_scale);
```

这里需要放大到一定程度，然后缩小到原始大小的一定程度，然后又放大，一直重复这个过程。

实现逻辑：

首先定义最大的放大倍数为 3 倍，最小缩小到原始大小的 0.8 倍。

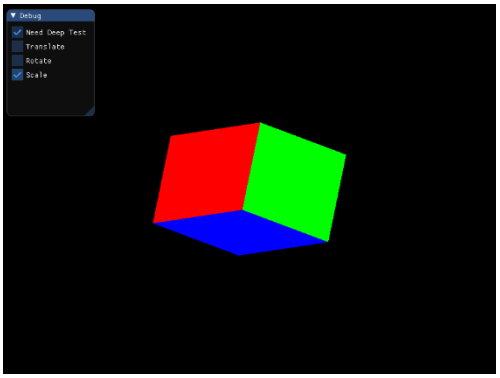
定义一个变量记录我们需要放大还是缩小

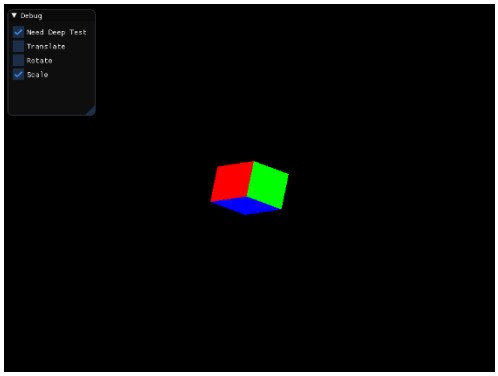
```
bool scale = true;
/*
    true: 放大
    false: 缩小
*/
```

在计算最后的变换矩阵之前，首先计算下一步的缩放变量。假如当前需要放大，放大超过了 3 倍，那么需要修改状态为缩小，使用前一个缩放变量来计算变换矩阵；否则用新的缩放变量来计算变换矩阵；缩小同理。

```
else if (is_scale) {
    is_scale = true;
    is_trans = is_rotate = false;
    glm::vec3 tmp_scale;
    float p_scale = 1.1, n_scale = 0.95;
    glm::vec4 axis = glm::vec4(0.1f, 0.1f, 0.1f, 1.0f);
    if (scale) {
        tmp_scale = pre_scale * p_scale;
        if (tmp_scale.x <= 3.0f && tmp_scale.y <= 3.0f) {
            trans = glm::scale(trans, tmp_scale);
            pre_scale = tmp_scale;
        }
        else {
            scale = false;
            trans = glm::scale(trans, pre_scale);
        }
    }
    else {
        tmp_scale = pre_scale * n_scale;
        if (tmp_scale.x >= 0.8 && tmp_scale.y >= 0.8) {
            trans = glm::scale(trans, tmp_scale);
            pre_scale = tmp_scale;
        }
        else {
            scale = true;
            trans = glm::scale(trans, pre_scale);
        }
    }
}
```

运行结果：





3. **结合 Shader 谈谈对渲染管线的理解**

OpenGL 的 shader 渲染管线使用一个通道来处理数据，渲染数据。在这个管道中，每一个步骤都有输入和输出，而且上一个步骤的输出，就是下一个步骤的输入。

OpenGL 的 shader 渲染管线，分为 4 个步骤处理。

1. 顶点处理步骤

   这一个步骤中，输入的是需要画出来的顶点的坐标以及相关属性，如颜色，纹理等等。这个步骤可以对顶点数据进行一系列处理，如各种变换，放缩，旋转，设置颜色等等。执行完这一步骤之后，输出的是需要展现的顶点数据。

   这一步骤的 shader 叫做顶点 shader

2. 几何处理步骤

这一步骤处理点和点之间的关系，以及其余的附加信息，如拼接方式，是三角形还是正方形等等。这一个步骤的输入是顶点处理步骤输出的一系列点，输出的是那些点按照我们的要求拼接起来的图形。

3. 裁剪步骤
   这一步骤输入的是需要画的一系列图形。然后根据裁剪空间对输入的几何体元进行裁剪，超出的会被裁剪丢弃。输出的是在裁剪空间中的几何体元。
4. 光栅化和片元操作步骤
   这一步会将输入进来的几何体元进行光栅化，即将连续的几何体元可以用一个个像素点表示。这产生了许多片元，在片元 shader 的处理下，这些片元会被填充颜色，纹理等等，然后就可得到了我们看到的图形。

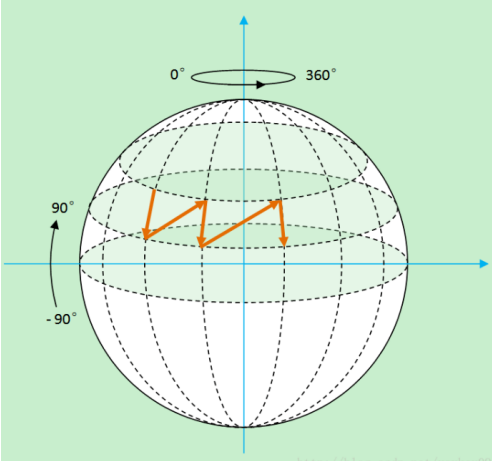
总而言之，渲染管线可以看作一个流水线工作，一步步紧密相连，将一些顶点坐标数据渲染成我们需要的图形。

Bonus:

1. **将以上三种变换相结合，打开你们的脑洞，实现有创意的动画**
   所做的动画：地球绕着太阳转

   分析:

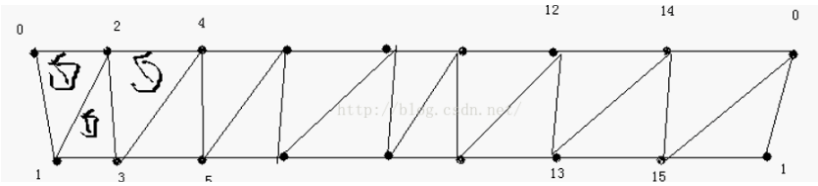
   1. 球体的生成
      球体的生成方法如下:



将一个球体切割成N层，根据角度来切分。比如第 1 层，其圆周上的点连接球心的直线与球体的法线成θ角，第 2 层则是$2\times\theta$角。由此可以确定每一层的半径r以及当前层的y坐标。

然后对于每一层，将圆周均分成n段，即角度被均分为β度一份。然后可以求得每一个点的x，z坐标。这样可以求出了构成球体所需要的点的坐标。

那么，下一步需要用三角形将点连接起来，连接方式如下:



0 和 1 是不同层圆周的对应点，其余一样，即他们在各自圆周对应的角度是一样的。三角形连接有以下规律:

从上面那层的点2n开始，需要有 2 个三角形:

$$(2n, 2n-1, 2n+1)$$
$$(2n, 2n+1, 2n+2)$$

其中边缘点需要特殊处理。

这样画的目的是为了保证都是逆时针画的，防止被 OpenGL 剔除。

2. 涉及的变换
   a) 放缩变换

   地球是要比太阳小的，所以地球应该缩小一些，这里我设置的是缩小 2 倍

```
glm::mat4 sizeScale[2];
glm::mat4 temp = glm::mat4(1.0f);
sizeScale[0] = glm::scale(temp, glm::vec3(1.0f, 1.0f, 1.0f));
sizeScale[1] = glm::scale(temp, glm::vec3(0.5f, 0.5f, 0.5f));
```

   b) 平移变换

   由于实体生成都是在原点，所以需要将地球这个实体挪下位置。这里我是在 x 轴平移了一些距离。

```
glm::vec3 translations[2];
translations[0] = glm::vec3(0.0f, 0.0f, 0.0f);
translations[1] = glm::vec3(-1.0f, 0.0f, 0.0f);
```

   c) 旋转变换

   地球绕着太阳旋转，这肯定得用到了旋转变换。这里我需要地球绕着原点变换，所以可以直接使用旋转矩阵而不需要进行移回来旋转后再移回去。

```
glm::mat4 posMatrix[2];
glm::mat4 befor[2], after[2];
posMatrix[0] = glm::rotate(temp, (float)glfwGetTime()*glm::radians(20.0f), glm::vec3(0.0f, 1.0f, 1.0f));
//posMatrix[1] = glm::translate(temp, glm::vec3(0.8f, 0.0f, 0.0f));
posMatrix[1] = glm::rotate(temp, (float)glfwGetTime()*glm::radians(50.0f), glm::vec3(0.0f, 1.0f, 1.0f));
//posMatrix[1] = glm::translate(posMatrix[1], glm::vec3(-0.8f, 0.0f, 0.0f));
```

3. 效果图: