@TOC

姓名：伍斌
学号：17343124

---

# Task 1

**实现观察矩阵（View Matrix）的计算，如下所示，该函数在 TRRenderer.cpp 文件中。**

```
    glm::mat4 TRRenderer::calcViewMatrix(glm::vec3 camera, glm::vec3
 target, glm::vec3 worldUp)
    {
        //Setup view matrix (world space -> camera space)
        glm::mat4 vMat = glm::mat4(1.0f);

        //Task 1: Implement the calculation of view matrix, and then
 set it to vMat
        //  Note: You can use any glm function (such as glm::normali
ze, glm::cross, glm::dot) except glm::lookAt

        glm::vec3 zAxis = glm::normalize(target - camera); // forwar
d vector
        glm::vec3 xAxis = glm::normalize(glm::cross(zAxis, worldUp))
; // side vector
        glm::vec3 yAxis = glm::normalize(glm::cross(xAxis, zAxis));
// up vector
        glm::mat4 vMat1(
            glm::vec4(xAxis.x, yAxis.x, -zAxis.x, 0.0), // 第一列
            glm::vec4(xAxis.y, yAxis.y, -zAxis.y, 0.0), // 第二列
            glm::vec4(xAxis.z, yAxis.z, -zAxis.z, 0.0), // 第三列
            glm::vec4(-glm::dot(camera , xAxis), -glm::dot(camera, y
Axis), glm::dot(camera, zAxis), 1.0f)  // 第四列
        );
        vMat = vMat1;
        return vMat;
    }
```

**简述你是怎么做的。**
直接计算目标矩阵：
首先根据课上的知识可以得出，取坐标转换矩阵的过程即将世界坐标系旋转和平移至于相机坐标系重合，这样这个旋转RR和平移TT矩阵的组合矩阵 M = T * R ，就是将相机坐标系中坐标变换到世界坐标系中坐标的变换矩阵，那么所求的视变换矩阵（世界坐标系中坐标转换到相机坐标系中坐标的矩阵）

$$view = M^{-1}$$

其中R就是上面求得的side、up、forward基向量构成的矩阵，如下：

$$R = \begin{bmatrix} s[0] & u[0] & -f[0] & 0 \\ s[1] & u[1] & -f[1] & 0 \\ s[2] & u[2] & -f[2] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 0 & 0 & 0 & eye_x \\ 0 & 0 & 0 & eye_y \\ 0 & 0 & 0 & eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



计算得到观察矩阵为：

$$view = R^T * T^{-1} = \begin{bmatrix} s[0] & s[1] & s[2] & -dot(s, eye) \\ u[0] & u[1] & u[2] & -dot(u, eye) \\ -f[0] & -f[1] & -f[2] & dot(f, eye) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 把S、U和F向量用更熟悉的向量替代为xAxis、yAxis和zAxis，更容易理解。

---

## Task 2

**实现透视投影矩阵（Project Matrix）的计算，如下所示，该函数在 TRRenderer.cpp 文件中。**

```
    glm::mat4 TRRenderer::calcPerspProjectMatrix(float fovy, float aspect, float near, float far)
    {
        //Setup perspective matrix (camera space -> clip space)
        glm::mat4 pMat = glm::mat4(1.0f);

        //Task 2: Implement the calculation of perspective matrix, and then set it to pMat
        //  Note: You can use any math function (such as std::tan) except glm::perspective
```

```
        float rFovy = fovy * M_PI / 180;
        const float tanHalfFovy = tanf(static_cast<float>(rFovy * 0.
5f));

        glm::mat4 pMat1(
            glm::vec4(1.0f / (aspect * tanHalfFovy), 0.0, 0.0, 0.0),
 // 第一列
            glm::vec4(0.0, 1.0f / (tanHalfFovy), 0.0, 0.0), // 第二列
            glm::vec4(0.0, 0.0, -(far + near) / (far - near), -1.0f)
, // 第三列
            glm::vec4(0.0, 0.0, -(2.0f * near * far) / (far - near),
 0.0)  // 第四列
        );

        pMat = pMat1;
        return pMat;
    }
```

**简述你是怎么做的。**

fov：纵向的视角大小

aspect：裁剪面的宽高比

near：近裁剪面离摄像机的距离n

far：远裁剪面离摄像机的距离f

通过上述几个参数可获得近裁剪面高度和宽度：

$$tan(fov/2) = \frac{H/2}{near} \Rightarrow H = 2 * near * tan(fov/2)$$
$$W/H = aspect \Rightarrow W = 2 * aspect * near * tan(fov/2)$$

求点P在近裁剪面的投影点P'的坐标：

$$\because \frac{-near}{z} = \frac{x'}{x} = \frac{y'}{y}$$
$$\therefore x' = -x * near/z$$
$$y' = -y * near/z$$

因为$x' \in [-W/2, W/2], y' \in [-H/2, H/2]$，要使二者范围在$[-1, 1]$内，需分别除$W/2, H/2$：

$$x'' = \frac{x'}{W/2} = \frac{-x * near/z}{aspect * near * tan(fov/2)} = \frac{-x}{z * aspect * tan(fov/2)}$$
$$y'' = \frac{y'}{H/2} = \frac{-y * near/z}{near * tan(fov/2)} = \frac{-y}{z * tan(fov/2)}$$

假设$z'' \in [-1, 1]$，则P''为$(\frac{-x}{z*aspect*tan(fov/2)}, \frac{-y}{z*tan(fov/2)}, z'')$

根据上述P''的坐标，可得需要找到一个矩阵使：（为方便求解，四维列向量每一项乘以-z）

$$\begin{bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{aspect*tan(fov/2)} \\ \frac{y}{tan(fov/2)} \\ -z * z'' \\ -z \end{bmatrix}$$

求解可得：

$$m00 = \frac{1}{aspect * tan(fov/2)}$$
$$m11 = \frac{1}{tan(fov/2)}$$
$$m32 = -1$$
$$\because m22 * z + m23 = -z * z'', \text{且} z = -near\text{时}，z'' = -1; z = -far\text{时}，z'' = 1$$
$$\therefore m22 = \frac{-near - far}{far - near}$$
$$m23 = \frac{-2 * near * far}{far - near}$$

透视投影变换矩阵：

$$PerspProjectMatrix = \begin{bmatrix} \frac{1}{aspect*tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(fov/2)} & 0 & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2*near*far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

## Task 3

**实现视口变换矩阵（Viewport Matrix）的计算，如下所示，该函数在 TRRenderer.cpp 文件中。**

```
glm::mat4 TRRenderer::calcViewPortMatrix(int width, int height)
{
    //Setup viewport matrix (ndc space -> screen space)
```

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

        //Task 3: Implement the calculation of viewport matrix, and
then set it to vpMat

        glm::mat4 vpMat1(
            glm::vec4(0.5f * width, 0.0, 0.0, 0.0), // 第一列
            glm::vec4(0.0, -0.5f * height, 0.0, 0.0), // 第二列
            glm::vec4(0.0, 0.0, 0.0, 0.0), // 第三列
            glm::vec4(0.5f * width, 0.5f * height, 0.0, 1.0f)  // 第
四列
        );

        vpMat = vpMat1;
        return vpMat;
    }
```

**简述你是怎么做的。**

由课上老师所给，当左上角作为坐标顶点时：

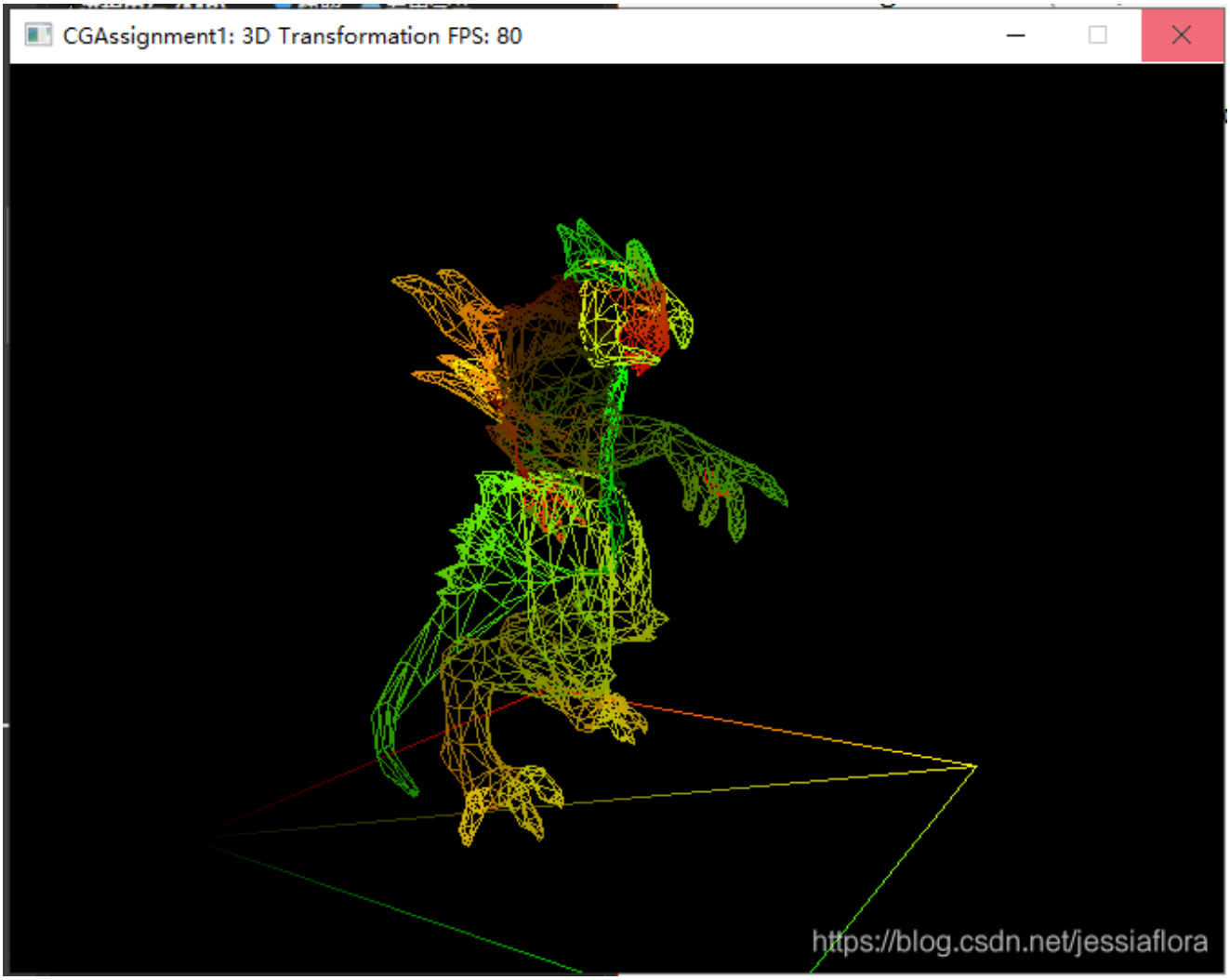
$$ViewportMatrix = \begin{bmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & -\frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

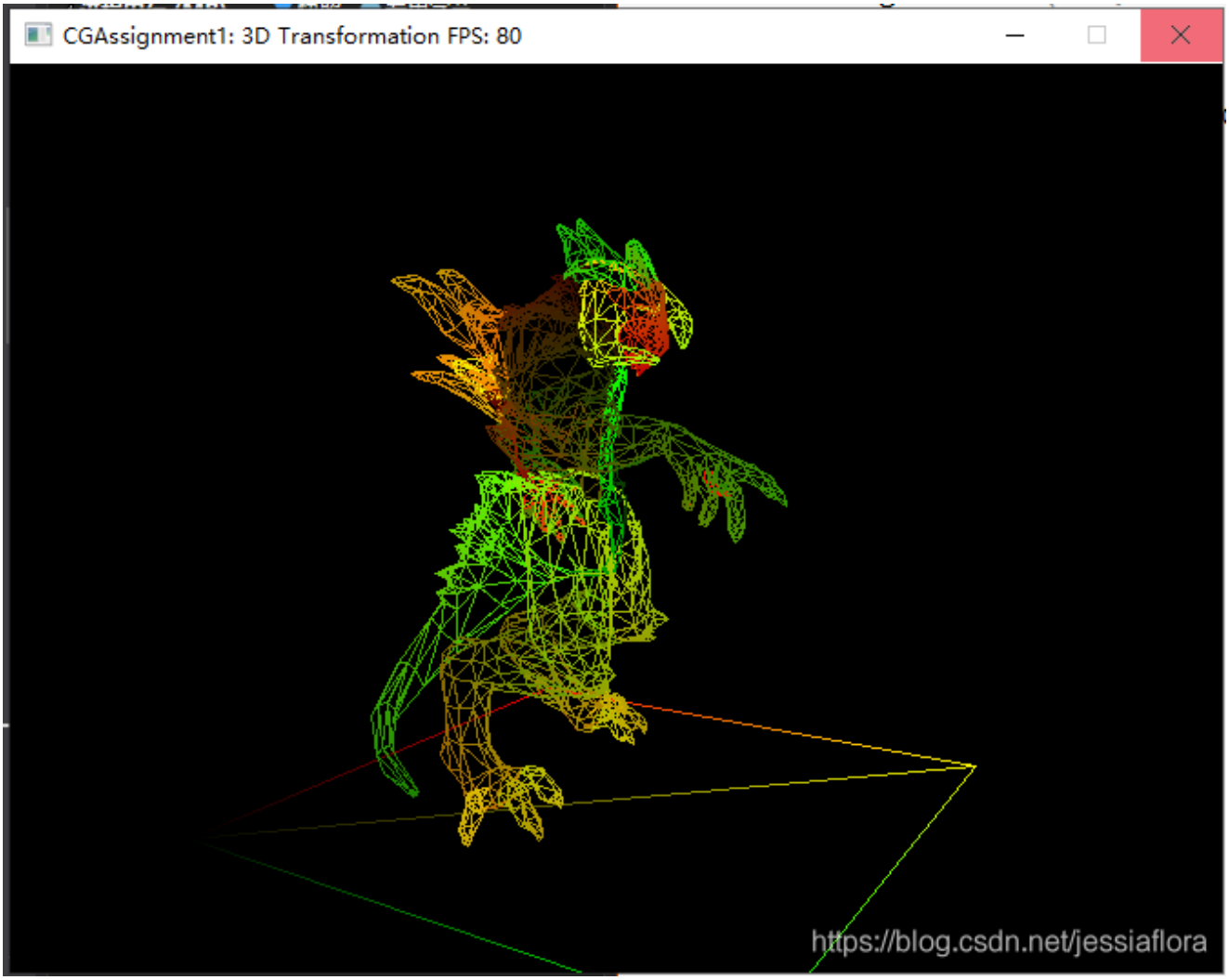
（这一步取决于视口坐标系是左下角还是左上角，在实验过程中我一开始误用了左上角坐标系的视口矩阵：

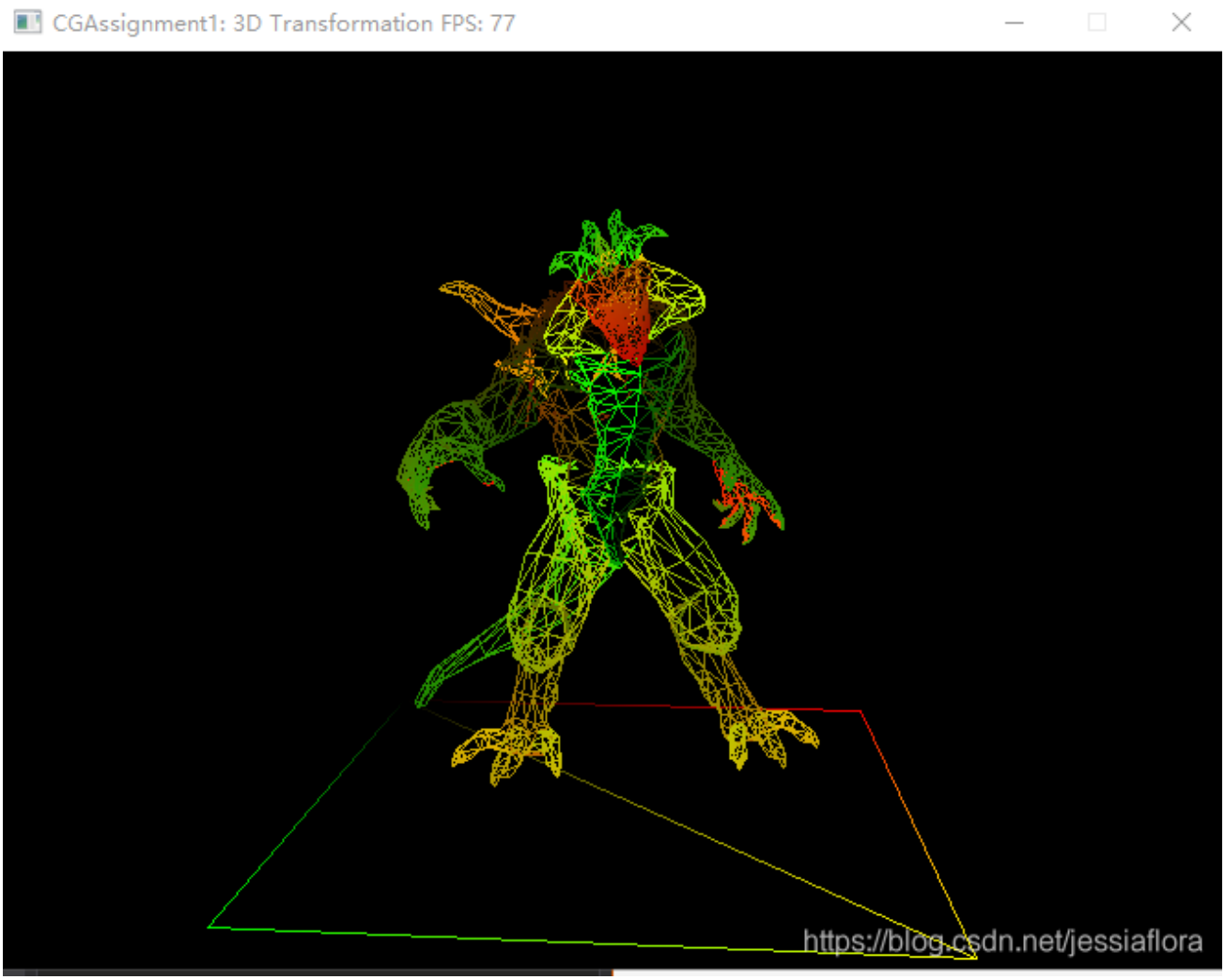
$$ViewportMatrix = \begin{bmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

导致我一开始显示的图形是上下左右颠倒的。）

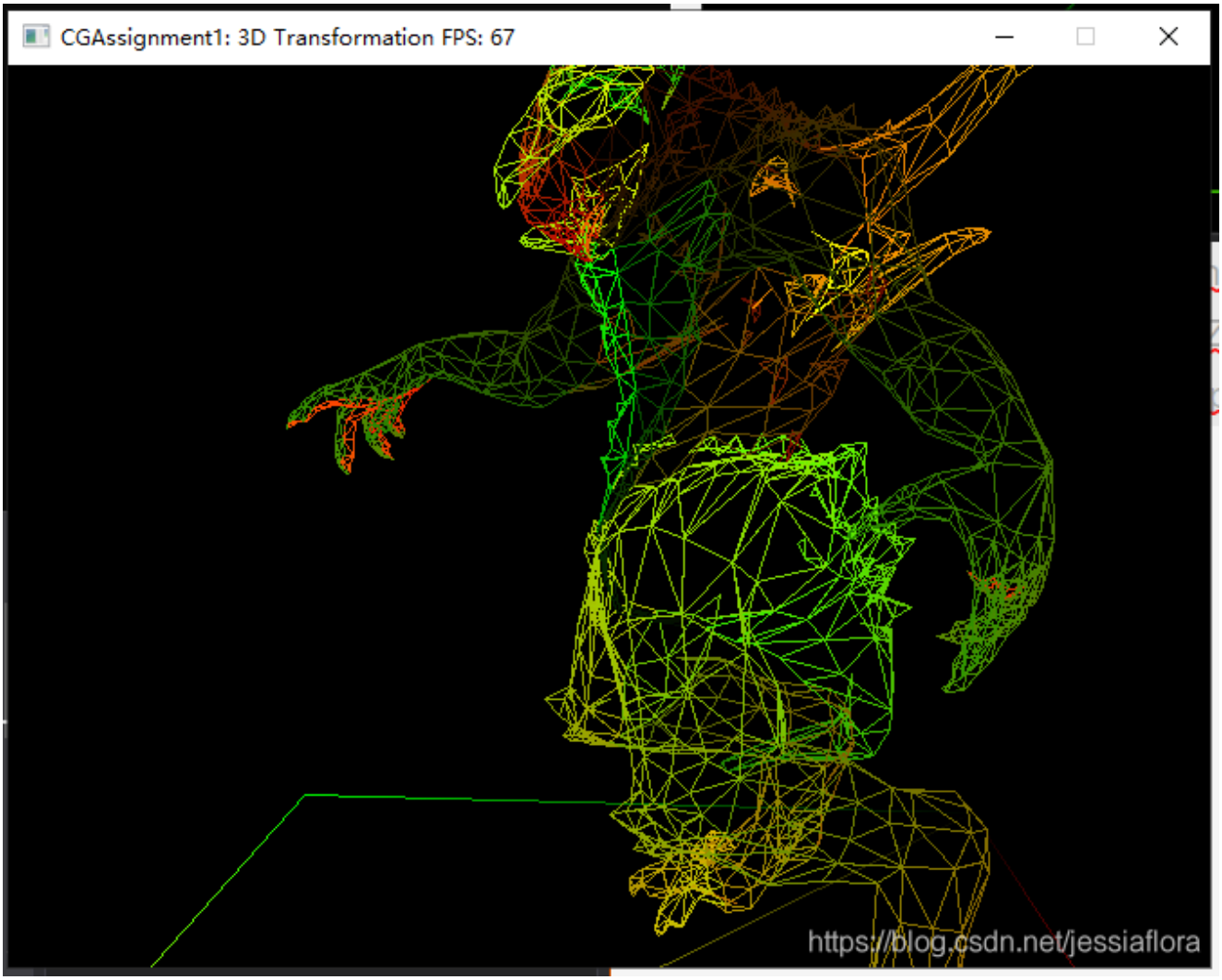
**贴出你的实现效果**


CGAssignment1: 3D Transformation FPS: 80
https://blog.csdn.net/jessiaflora

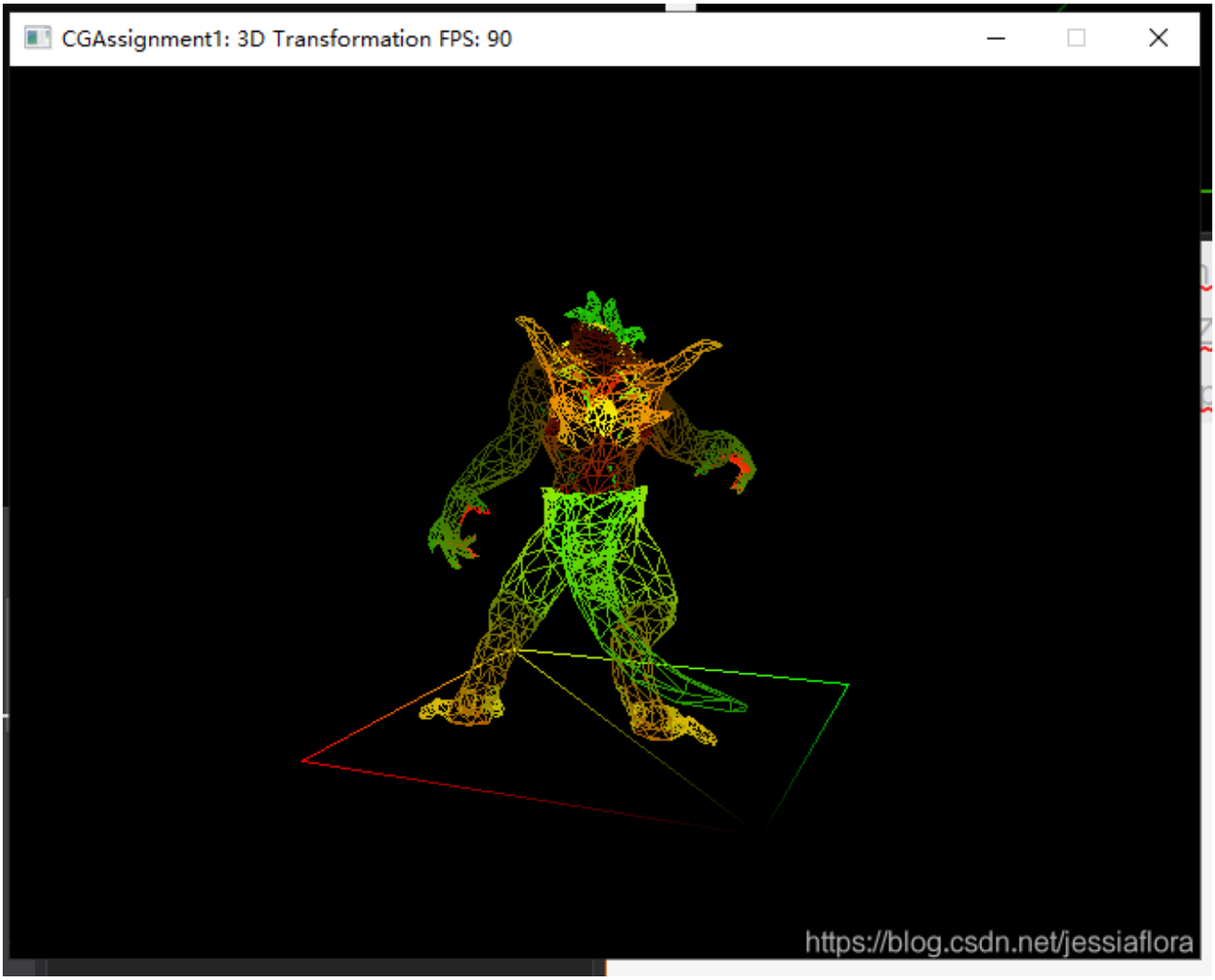

CGAssignment1: 3D Transformation FPS: 80
https://blog.csdn.net/jessiaflora


CGAssignment1: 3D Transformation FPS: 77
https://blog.csdn.net/jessiaflora

放大：


CGAssignment1: 3D Transformation FPS: 67
https://blog.csdn.net/jessiaflora

缩小：



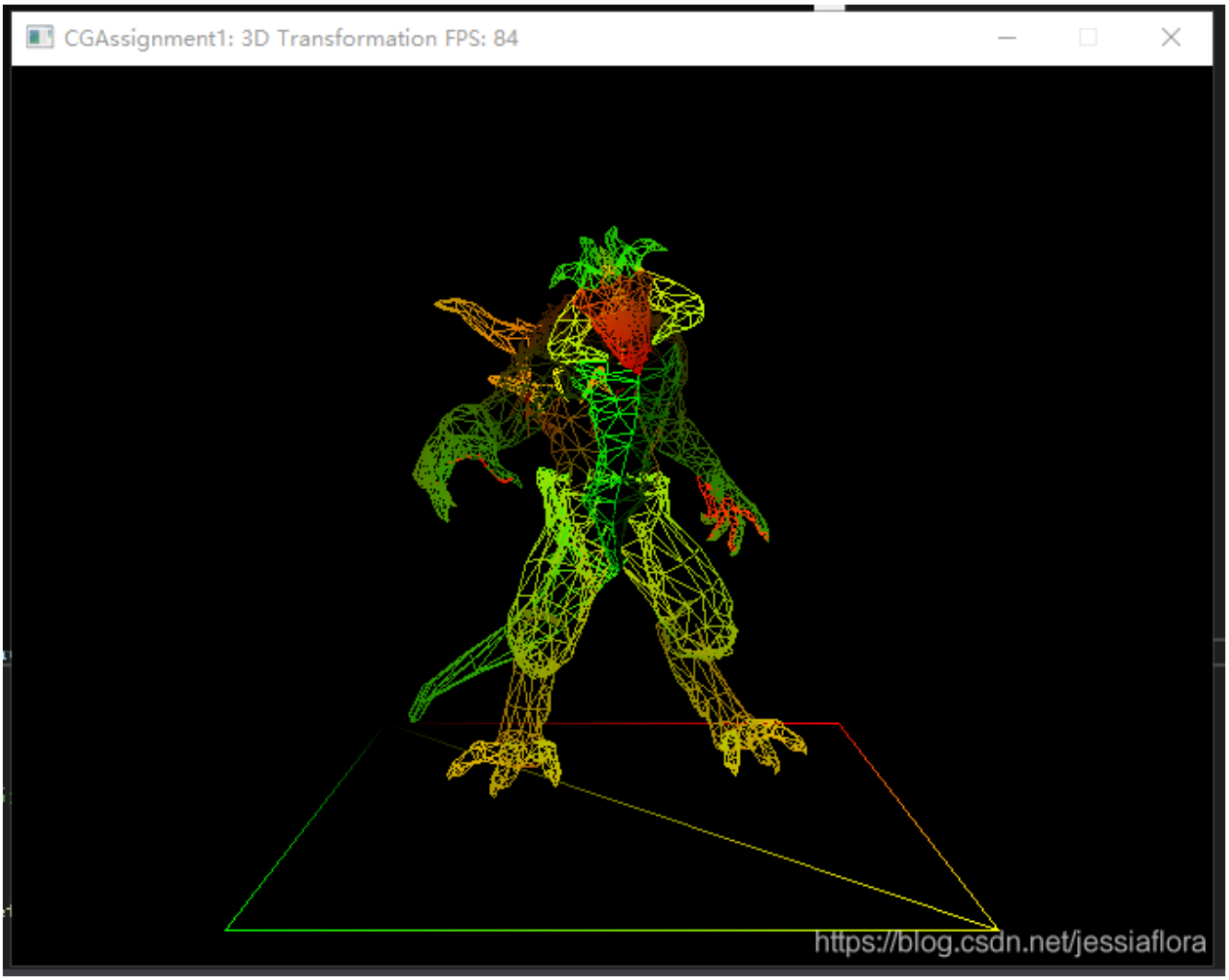

---

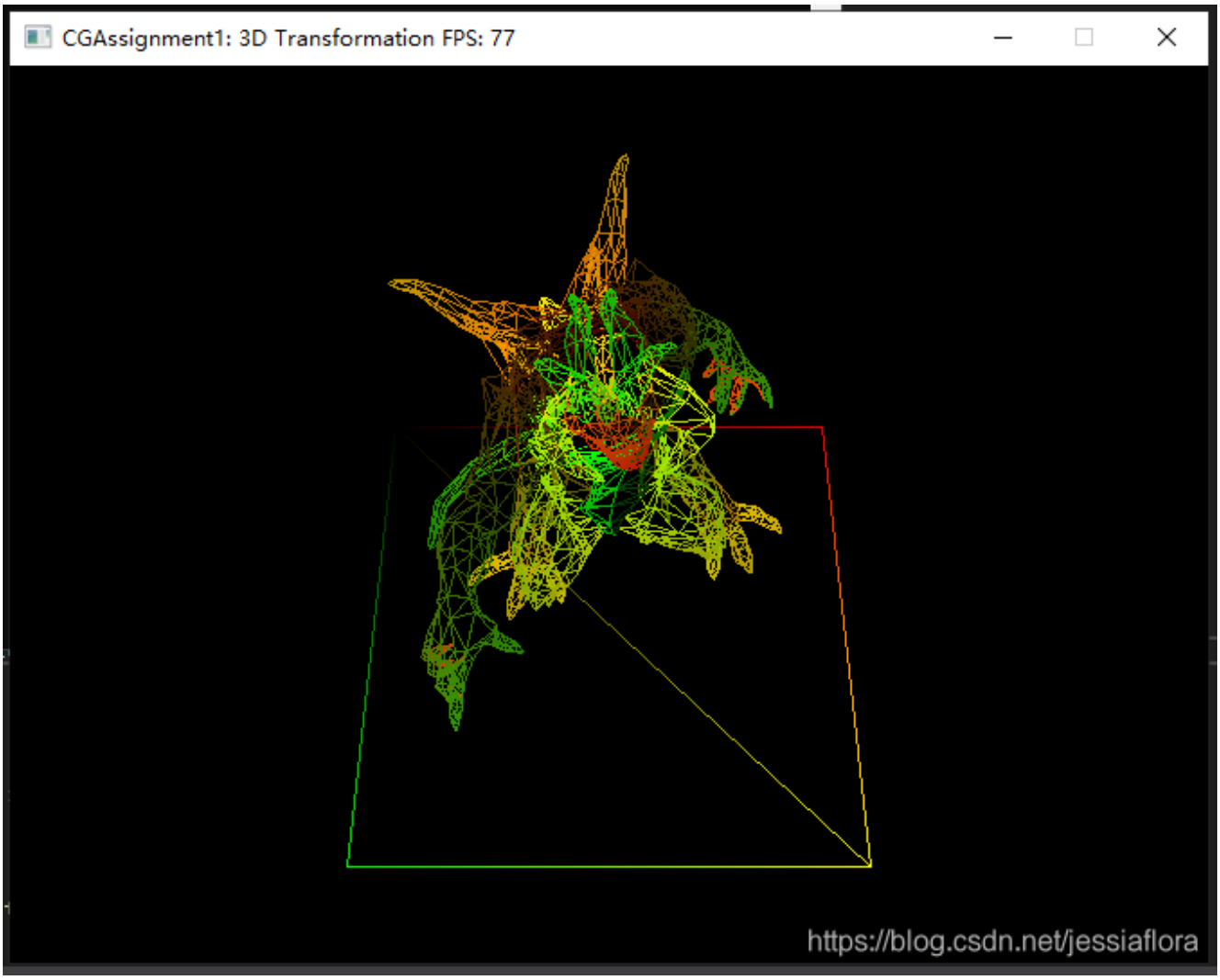
## Task4

**程序默认物体绕着 轴不停地旋转，请你在 main.cpp 文件中稍微修改一下代码，使得物体分别绕 x 轴和 z 轴旋转。贴出你的结果。**

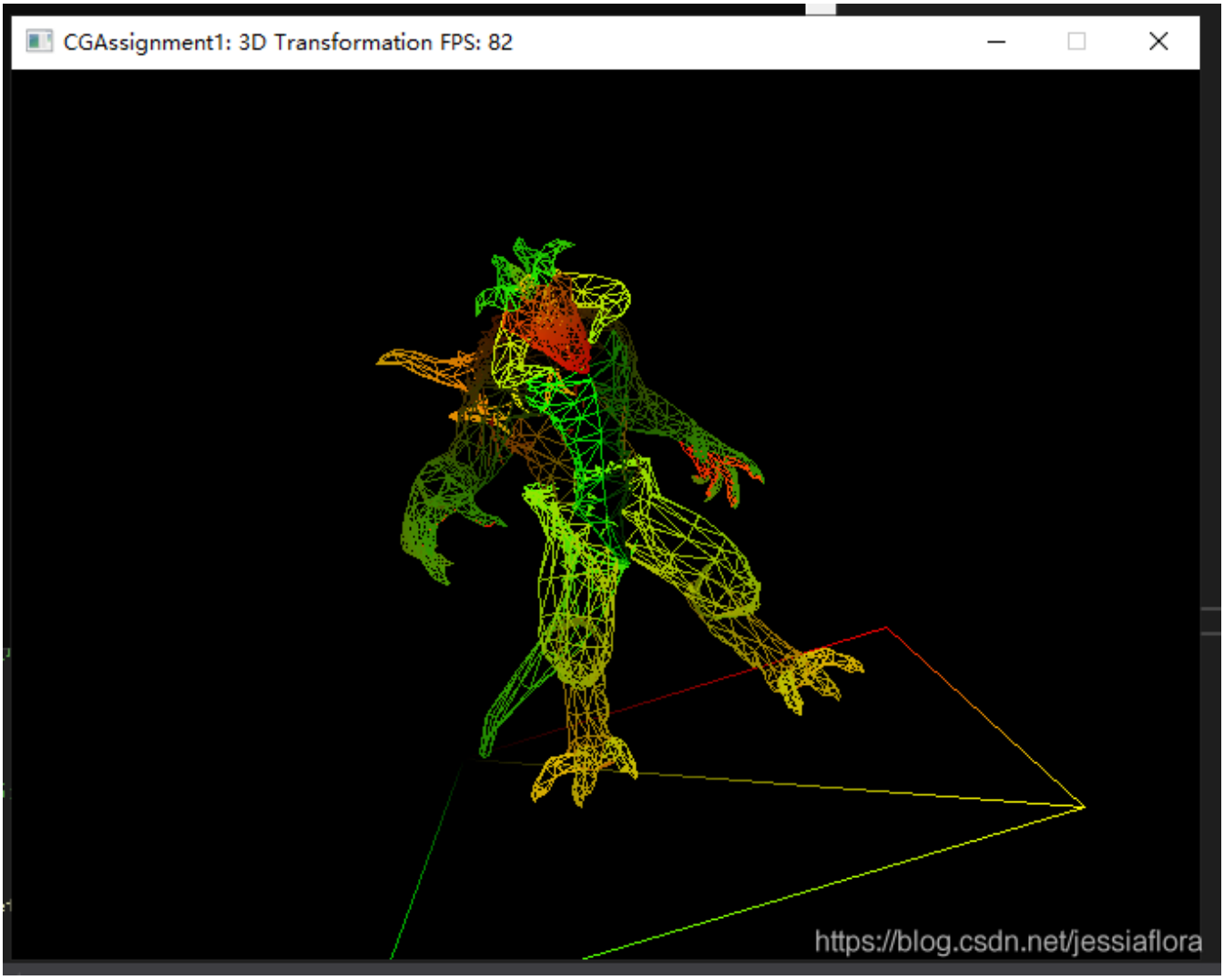
```
    //Rotation
    {
        //Task 4: Make the model rotate around x axis and z axis, respectively
        //model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f, glm::vec3(0, 1, 0));//绕y轴
        model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f, glm::vec3(1, 0, 0));//绕x轴
        //model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f, glm::vec3(0, 0, 1));//绕z轴
            }
```

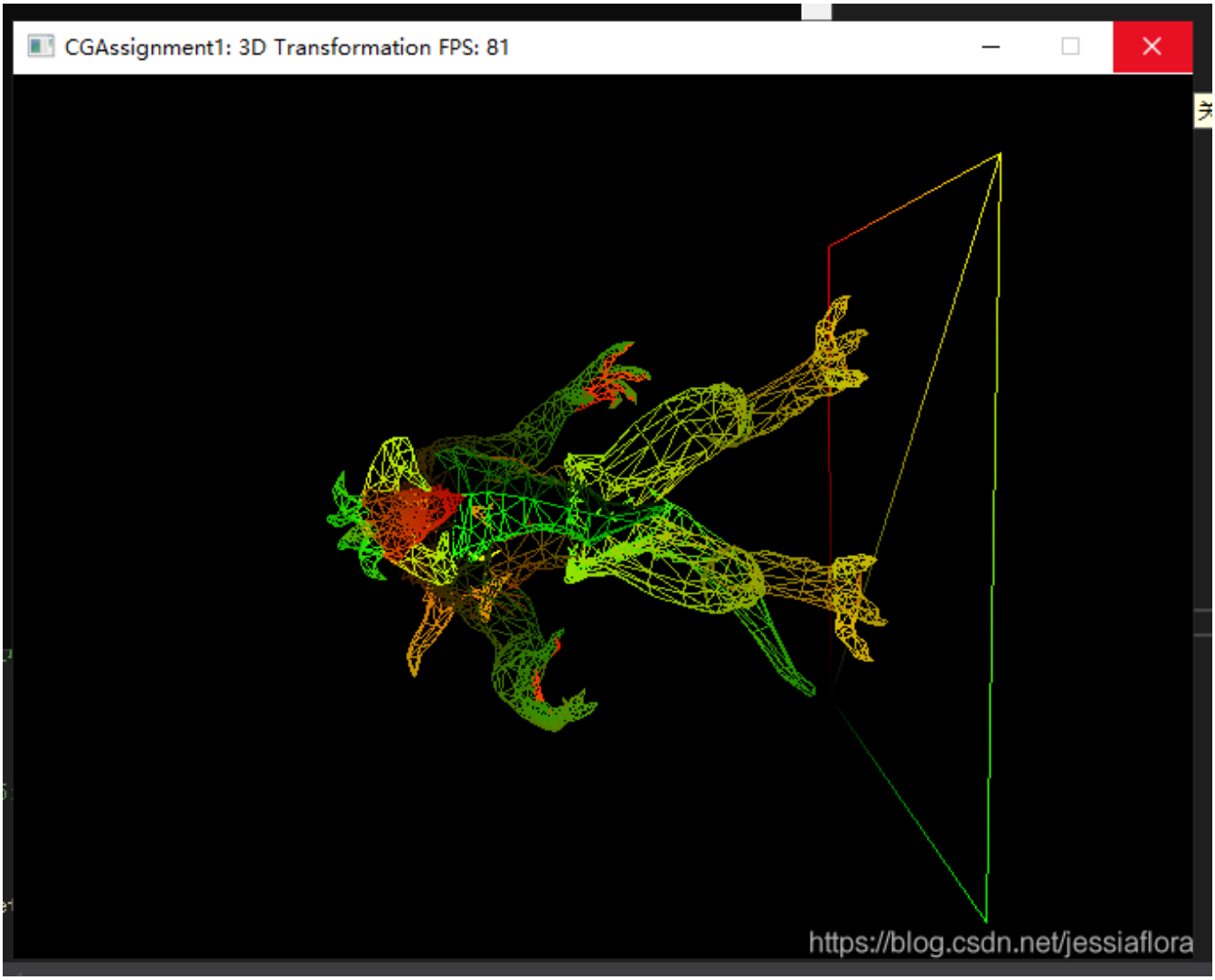
绕x轴旋转：


CGAssignment1: 3D Transformation FPS: 84
https://blog.csdn.net/jessiaflora


CGAssignment1: 3D Transformation FPS: 77
https://blog.csdn.net/jessiaflora

绕z轴旋转：


CGAssignment1: 3D Transformation FPS: 82
https://blog.csdn.net/jessiaflora




---

## Task5

**仔细体会使物体随着时间的推进不断绕 轴旋转的代码，现在要你用 glm::scale 函数实现物体不停地放大、缩小、放大的循环动画，物体先放大至 2.0倍、然后缩小至原来的大小，然后再放大至2。0，依次循环下去，要求缩放速度适中，不要太快也不要太慢。贴出你的效果，说说你是怎么实现的。（请注释掉前面的旋转代码）**

```
//Scale
    {
        //Task 5: Implement the scale up and down animation using gl
m::scale function
        model_mat = glm::translate(model_mat, glm::vec3(-0.5f, 0.5f,
 0.0f));
        GLfloat scaleAmount = sin(glfwGetTime())* 0.25 + 0.75;
        model_mat = glm::scale(model_mat, glm::vec3(scaleAmount, sca
leAmount, scaleAmount));
    }
```

理论上是根据glfwGetTime()取时间，利用sin得到周期，再调整其区间至取值0.5 ~ 1这个区间（即

放大2倍～恢复原状），利用glm::scale函数得到符合题意的周期性变化。但是由于我的电脑无法运行glfwGetTime()所在的glfw库，所以没能得出实验效果。

---

## Task6

**现在要求你实现正交投影矩阵的计算，如下所示，该函数在 TRRenderer.cpp 文件中。**

```
    glm::mat4 TRRenderer::calcOrthoProjectMatrix(float left, float r
ight, float bottom, float top, float near, float far)
    {
        //Setup orthogonal matrix (camera space -> homogeneous space
)
        glm::mat4 pMat = glm::mat4(1.0f);

        //Task 6: Implement the calculation of orthogonal projection
, and then set it to pMat

        glm::mat4 pMat1(
            glm::vec4(2.0f / (right - left), 0.0, 0.0, 0.0), // 第一
列
            glm::vec4(0.0, 2.0f / (top - bottom), 0.0, 0.0), // 第二
列
            glm::vec4(0.0, 0.0, -2.0f / (far - near), 0.0), // 第三列
            glm::vec4(-(right + left) / (right - left), -(top + bott
om) / (top - bottom), -(far + near) / (far - near), 1.0f)  // 第四列
        );

        pMat = pMat1;

        return pMat;
    }
```

**简述你是怎么做的。**
对于正交投影的映射：
l = left; r = right;
b = bottom; t = top;
n = near; f = far;
只需做简单的线性映射：
x轴方向$[l,r]->[-1,1]$;
y轴方向$[b,t]->[-1,1]$;
z轴方向$[-n,-f]->[-1,1]$;

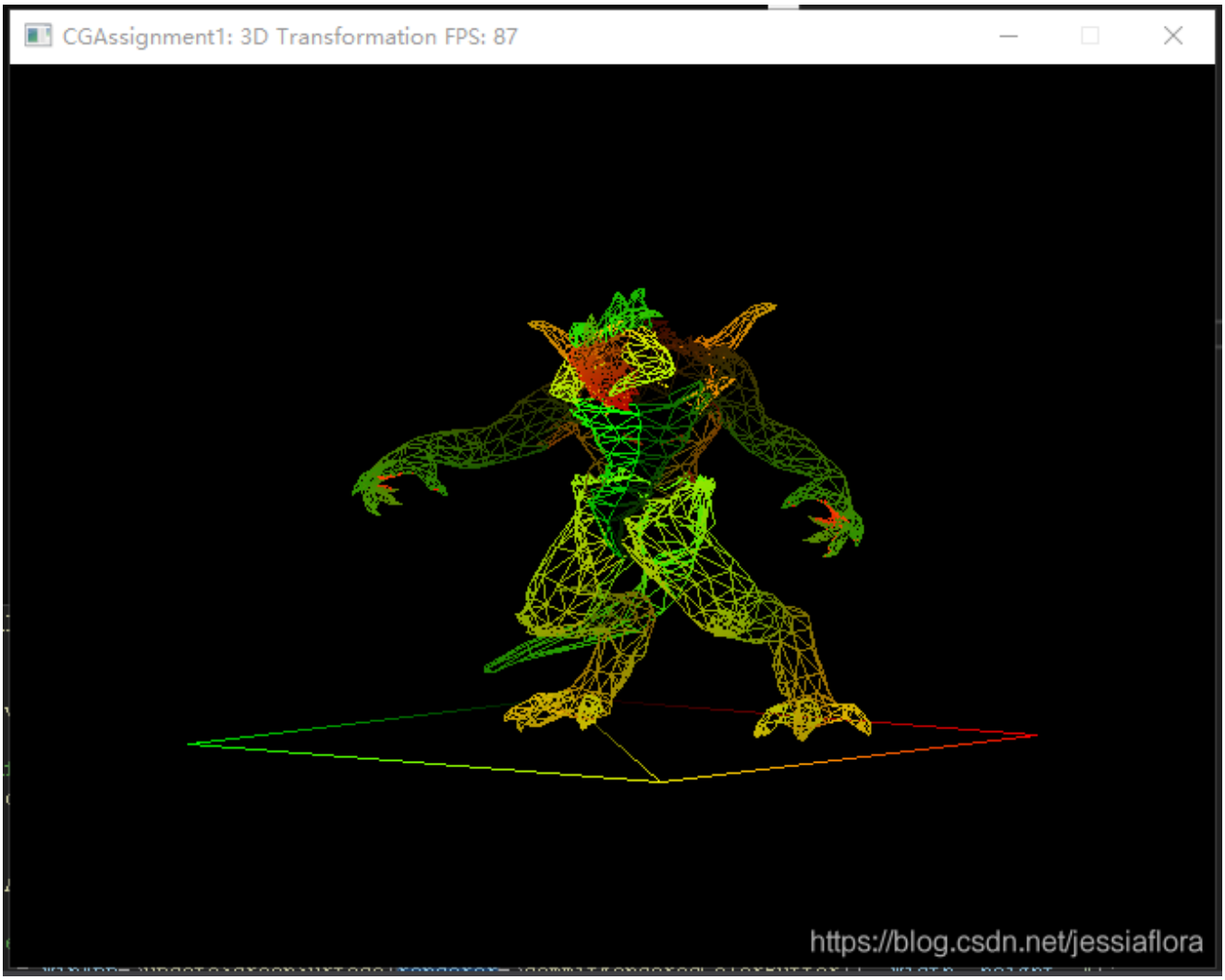
正交投影变换矩阵

$$OrthoProjectMatrix = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

---

## Task7

**实现了正交投影计算后，在 main.cpp 的如下代码中，分别尝试调用透视投影和正交投影函数，通过滚动鼠标的滚轮来拉近或拉远摄像机的位置，仔细体会这两种投影的差别。**




两种投影的区别：
透视投影：虽然模型的四周在跟随旋转不断放缩，但使得整个模型更具有立体感。
正交投影：更加方正，模型的四周不会跟随旋转不断放缩。虽然完美保留了模型的长宽比，但是显得非常呆板。

---

## Task8

完成上述的编程实践之后，请你思考并回答以下问题：
**（1）请简述正交投影和透视投影的区别。**
使用透视投影照相机获得的结果是类似人眼在真实世界中看到的有“近大远小”的效果；而使用正交

投影照相机获得的结果就像我们在数学几何学课上老师教我们画的效果，对于在三维空间内平行的线，投影到二维空间中也一定是平行的。

**（2）从物体局部空间的顶点到最终的屏幕空间上的顶点，需要经历哪几个空间的坐标系？裁剪空间下的顶点的w值是哪个空间坐标下的z值？它有什么空间意义？**
从物体局部空间的顶点到最终屏幕空间上的顶点，需要经历：
局部空间坐标系->世界坐标系->照相机坐标系->裁剪坐标系->NDC规范化设备坐标系->屏幕坐标系。
裁剪空间下的顶点的w值是view space（照相机空间）坐标下的z值，代表了顶点的深度。

**（3）经过投影变换之后，几何顶点的坐标值是被直接变换到了NDC坐标（即xyz的值全部都在$[-1,1]$范围内）吗？透视除法（Perspective Division）是什么？为什么要有这么一个过程？**
不是，直接投影在NDC坐标系，需要经历透视除法。
**透视除法**：
首先裁剪空间和NDC空间不是一个概念，裁剪空间是一个顶点乘以MVP矩阵之后所在的空间，Vertex Shader的输出就是是裁剪空间上，接着GPU会裁剪，剔除裁剪空间外的顶点，然后GPU再做透视除法将顶点转化到NDC。
透视除法：将裁剪空间的顶点的四个分量都除以w分量，就从裁剪空间转化到NDC空间了。
NDC是一个长宽高取值范围为$[-1,1]$的立方体，其意义在于顶点最后要输出到屏幕空间。