Assignment 1: 3D Transformation

教学班级	专业（方向）	学号	姓名
2班	计算机科学与技术	21307174	刘俊杰

一、基础任务：

1. 观察矩阵（View Matrix）

源代码:

```
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);

                //mplement the calculation of view matrix, and then set it to vMat
                //  Note: You can use any glm function (such as glm::normalize, glm::cross, glm::d


                //计算视图空间的三个轴向量
                glm::vec3 Z(glm::normalize(camera-target));
                glm::vec3 X(glm::normalize(glm::cross(worldUp, Z)));
                glm::vec3 Y(glm::normalize(glm::cross(Z,X)));

                //旋转矩阵Rotate
                glm::mat4 Rotate = glm::mat4(1.0f);

                Rotate[0][0] = X.x;
                Rotate[1][0] = X.y;
                Rotate[2][0] = X.z;

                Rotate[0][1] = Y.x;
                Rotate[1][1] = Y.y;
                Rotate[2][1] = Y.z;

                Rotate[0][2] = Z.x;
                Rotate[1][2] = Z.y;
                Rotate[2][2] = Z.z;
                Rotate[3][3] = 1.0f;

                //平移矩阵Translate
                glm::mat4 Translate = glm::mat4(1.0f);

                Translate[3][0] = -camera.x;
                Translate[3][1] = -camera.y;
                Translate[3][2] = -camera.z;

                //计算视图矩阵vmat
                vMat = Rotate * Translate;

                return vMat;
        }
```

实验原理

参数:

- camera:摄像机的位置
- target:摄像机目标点
- worldUP:世界空间的上向量

步骤:

1.首先求出了视图空间的三条轴向量

变换到视图空间中时摄像机是朝向视图空间的−Z方向的，所以求视图空间中的Z轴时是摄像机的位置减去目标点的位置。

摄像机右侧的方向是通过叉乘摄像机的上向量和正向向量，最后再使用单位化。

摄像机上方的方向是通过叉乘正向向量和右向量再单位化得到。

```
//计算视图空间的三个轴向量
			glm::vec3 Z(glm::normalize(camera-target));
			glm::vec3 X(glm::normalize(glm::cross(worldUp, Z)));
			glm::vec3 Y(glm::normalize(glm::cross(Z,X)));
```

2.构建旋转矩阵(其实是旋转矩阵的逆，因为摄像机实际上是通过物体的移动来实现的)

$$Rotate^{-1} = \begin{bmatrix} X.x & X.y & X.z & 0 \\ Y.x & Y.y & Y.z & 0 \\ Z.x & Z.y & Z.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
//旋转矩阵Rotate
                glm::mat4 Rotate = glm::mat4(1.0f);

                Rotate[0][0] = X.x;
                Rotate[1][0] = X.y;
                Rotate[2][0] = X.z;

                Rotate[0][1] = Y.x;
                Rotate[1][1] = Y.y;
                Rotate[2][1] = Y.z;

                Rotate[0][2] = Z.x;
                Rotate[1][2] = Z.y;
                Rotate[2][2] = Z.z;
                Rotate[3][3] = 1.0f;
```

3.构建平移矩阵(其实是平移矩阵的逆，因为摄像机实际上是通过物体的移动来实现的)

$$Translate^{-1} = \begin{bmatrix} 0 & 0 & 0 & -camera.x \\ 0 & 0 & 0 & -camera.y \\ 0 & 0 & 0 & -camera.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
//平移矩阵Translate
                glm::mat4 Translate = glm::mat4(1.0f);

                Translate[3][0] = -camera.x;
                Translate[3][1] = -camera.y;
                Translate[3][2] = -camera.z;
```

4.两者相乘得到观察矩阵

$$vMat = Rotate^{-1} * Translate^{-1} = \begin{bmatrix} X.x & X.y & X.z & 0 \\ Y.x & Y.y & Y.z & 0 \\ Z.x & Z.y & Z.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & -camera.x \\ 0 & 0 & 0 & -camera.y \\ 0 & 0 & 0 & -camera.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} X.x & X.y & X.z & -X * camera \\ Y.x & Y.y & Y.z & -Y * camera \\ Z.x & Z.y & Z.z & -Z * camera \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
//计算视图矩阵vmat
		vMat = Rotate * Translate;
```

2. 透视投影矩阵（Project Matrix)

源代码:

```
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);

		//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 Length = tan(fovy / 2.0f);

		pMat[0][0] = 1.0f / (aspect * Length);
		pMat[1][1] = 1.0f / Length;
		pMat[2][2] = -(far + near) / (far - near);
		pMat[2][3] = -1.0f;
		pMat[3][2] = -2.0f * far * near / (far - near);
	pMat[3][3] = 0.0f;


		return pMat;
	}
```

实验原理

推导得到透视投影矩阵:

$$pMat = \begin{bmatrix} \frac{1}{aspect*tan(fovy/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(fovy/2)} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

参数:

- fovy：y 方向的视域角
- aspect: 屏幕的宽高比

- near: 相机到近平面的距离
- far: 相机到远平面的距离

3. 视口变换矩阵（Viewport Matrix)

源代码:

```
glm::mat4 TRRenderer::calcViewPortMatrix(int width, int height)
	{
		//Setup viewport matrix (ndc space -> screen space)
		glm::mat4 vpMat = glm::mat4(1.0f);

		//Implement the calculation of viewport matrix, and then set it to vpMat
		vpMat[0][0] = static_cast<float>(width) / 2;
		vpMat[1][1] = static_cast<float>(-height) / 2;
		vpMat[3][0] = static_cast<float>(width - 1) / 2;
		vpMat[3][1] = static_cast<float>(height - 1) / 2;
		return vpMat;
	}
```

实验原理

视口变换是将3D投射到2D，将(-1,1)规范化坐标范围映射到width*height个像素组成的屏幕坐标，可以通过缩放再平移的方式来实现：

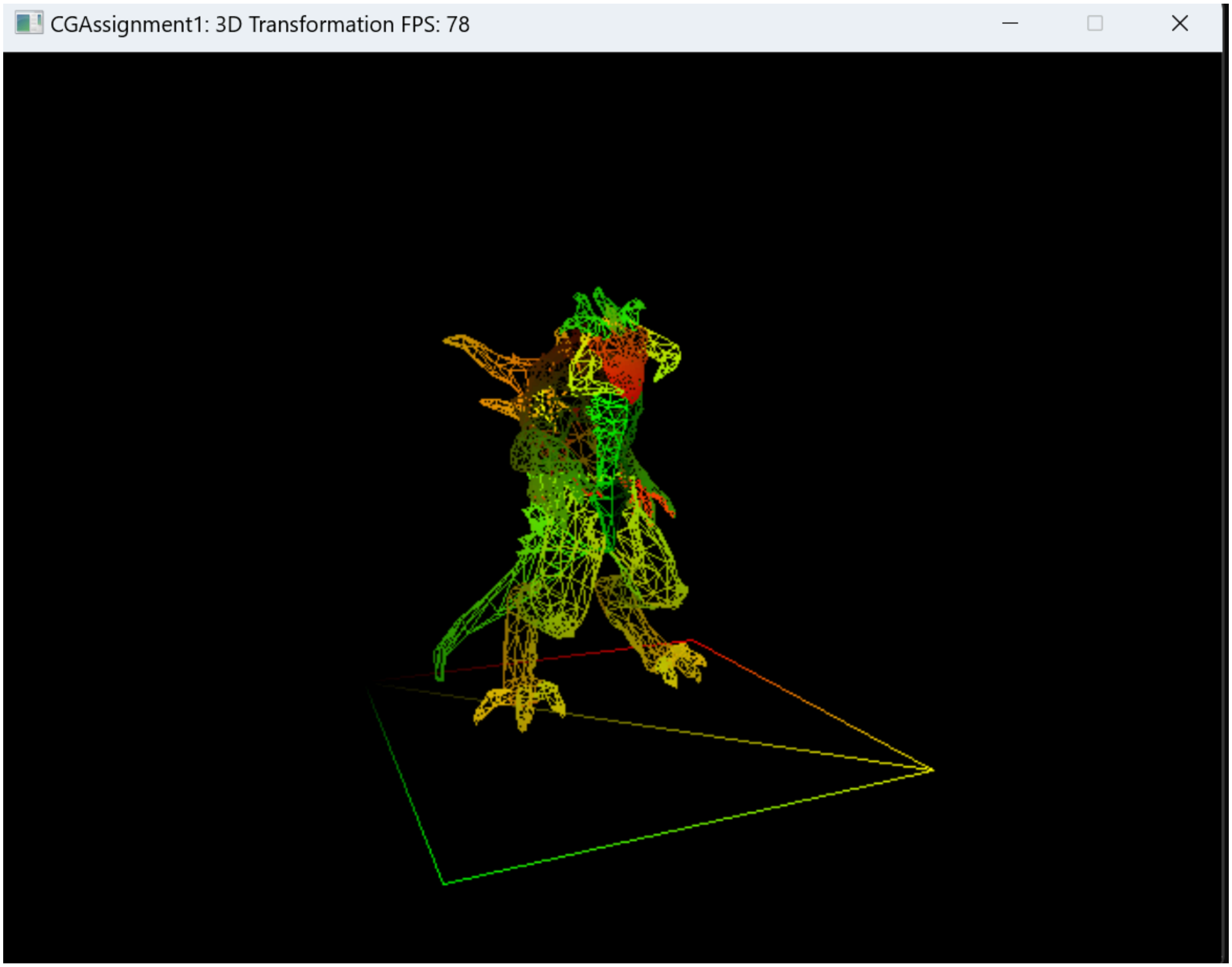
- 缩放：x、y轴从2、2放大到width、height，因此缩放比例为width/2, -height/2(y轴向下为正方向);
- 平移: 缩放后的原点位置不变，这个时候需要通过平移将左下角的点移动到屏幕坐标系的坐标原点，x、y轴向正向移动的距离分别为(width-1)/2、(height/-1)/2，之所以有这个0.5是(因为屏幕坐标原点为左下角像素)

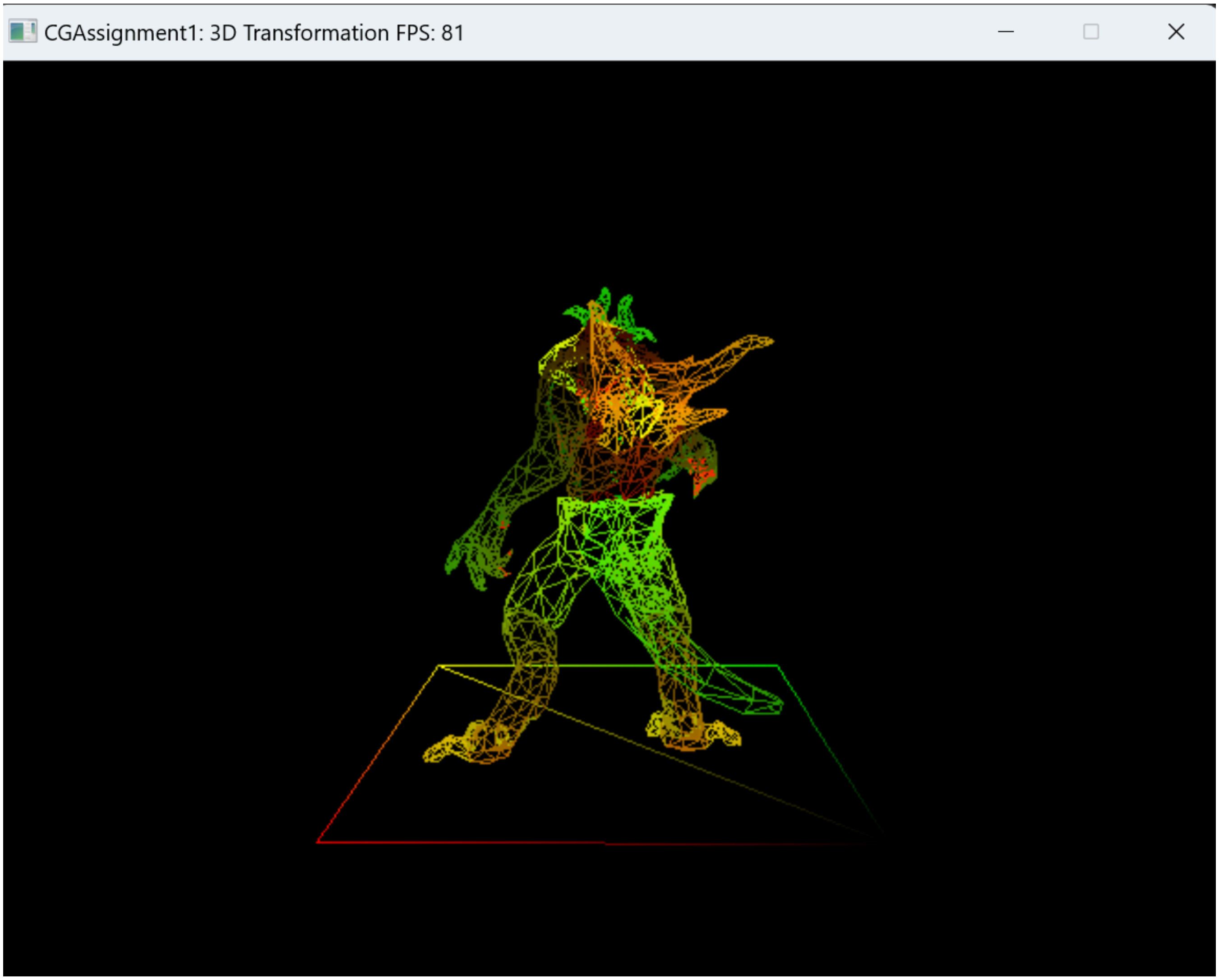
视口变换矩阵:

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

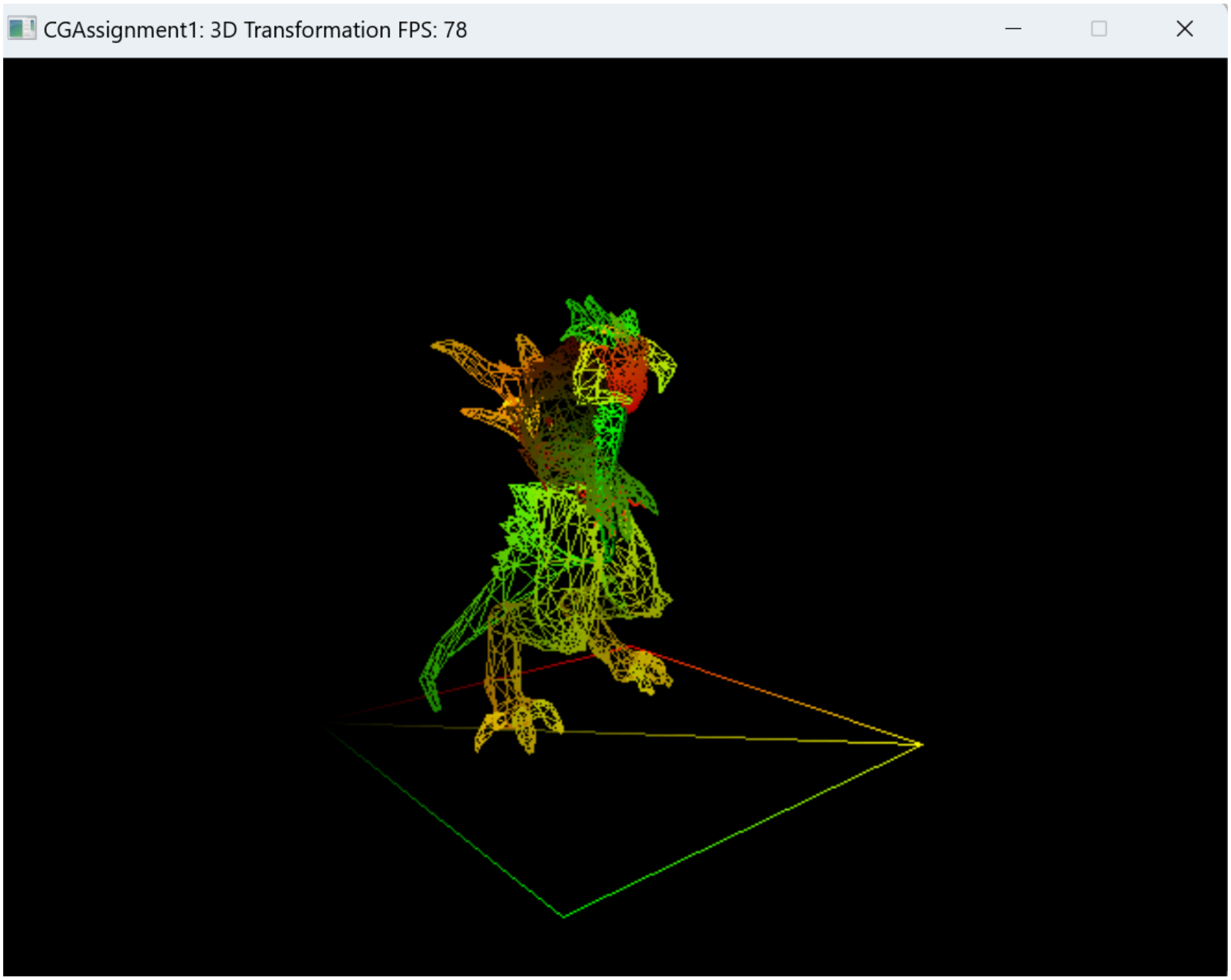
4.效果图(视频在附件中):

三维物体随着程序的运行绕y轴在不停地旋转:

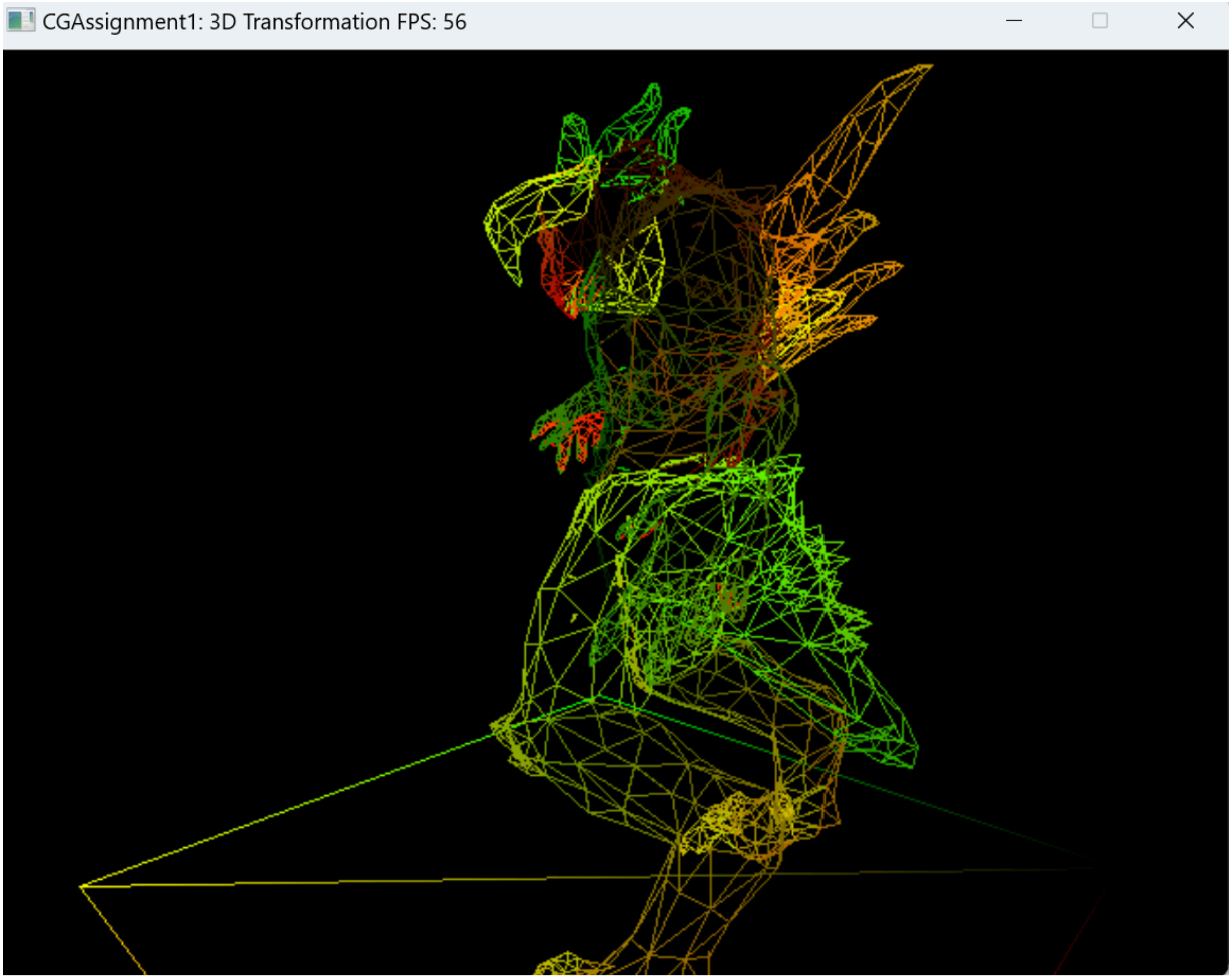




按住鼠标左键并横向拖动鼠标来旋转摄像机:



过滚动鼠标的滚轮来拉近摄像机的位置:



二、提升任务

源代码:

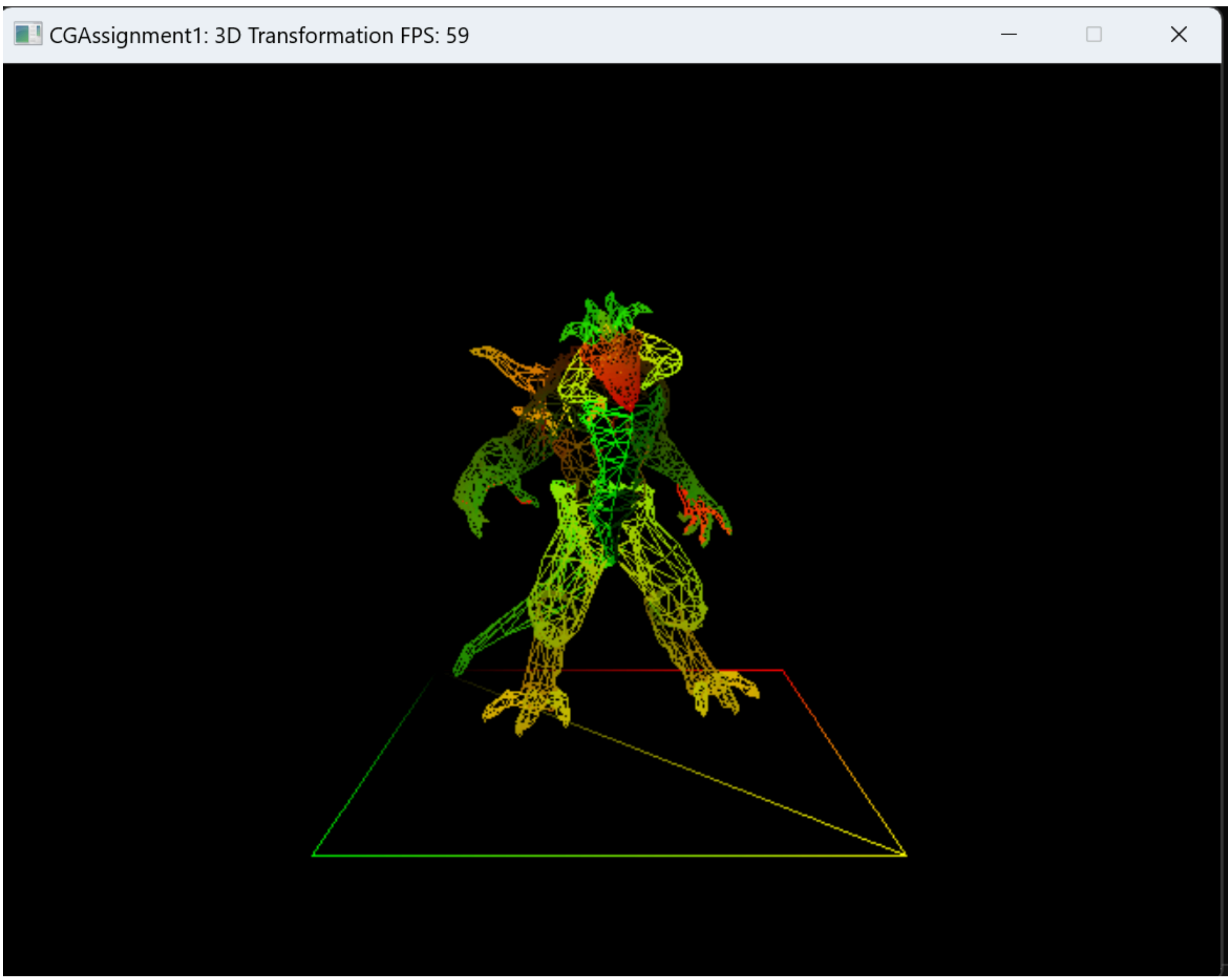
```
//Scale
                        {

                                static float scaleSize = 1.0f;//放缩倍数
                                static float scaleCircle = 0.01f;//放缩倍数的变化
                                if (scaleSize > 3)scaleCircle = -scaleCircle;//开始缩小
                                if (scaleSize < 1)scaleCircle = -scaleCircle;//开始变大
                                scaleSize += scaleCircle;
                                model_mat = glm::scale(glm::mat4(1.0f), glm::vec3(scaleSize, scale

                        }
```

实验原理

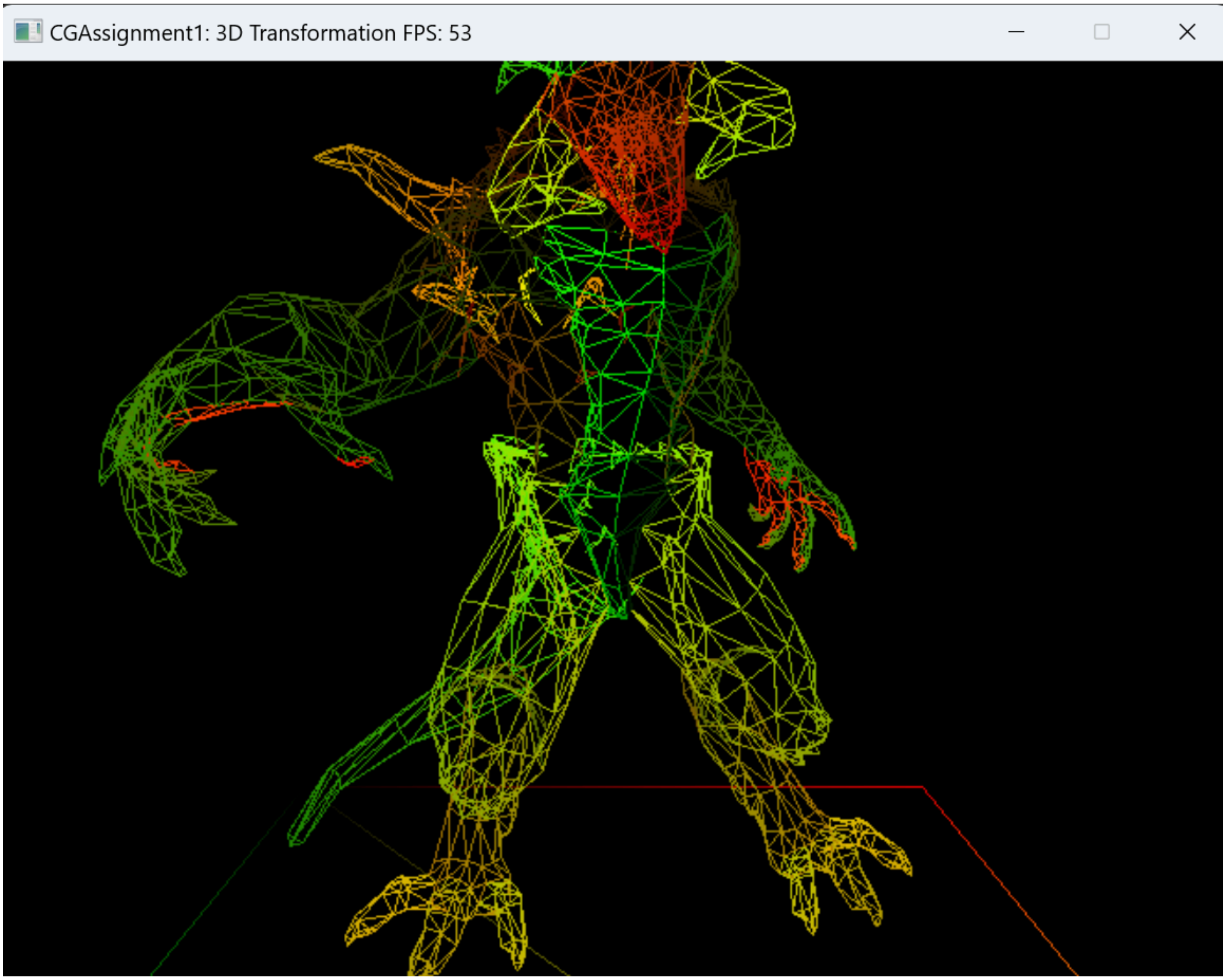
1. 首先定义static变量:放缩倍数scaleSize = 1.0f,
   放缩倍数的变化scaleCircle = 0.01f。
2. 每次判断是该变化缩放方向还是保持此时的放缩倍数的变化值:若scaleSize>3说明该缩小了，scaleSize<1说明该放大了。
3. 通过glm::scale函数进行缩小或者放大。(注意不能传入model_mat，因为其是static变量，仅在定义时声明其为单位矩阵，若使用model_mat则在之前放大或缩小的倍数上再乘以倍数)

效果图(视频在附件中)

物体缩小:



物体放大:


CGAssignment1: 3D Transformation FPS: 53

三、挑战任务

源代码:

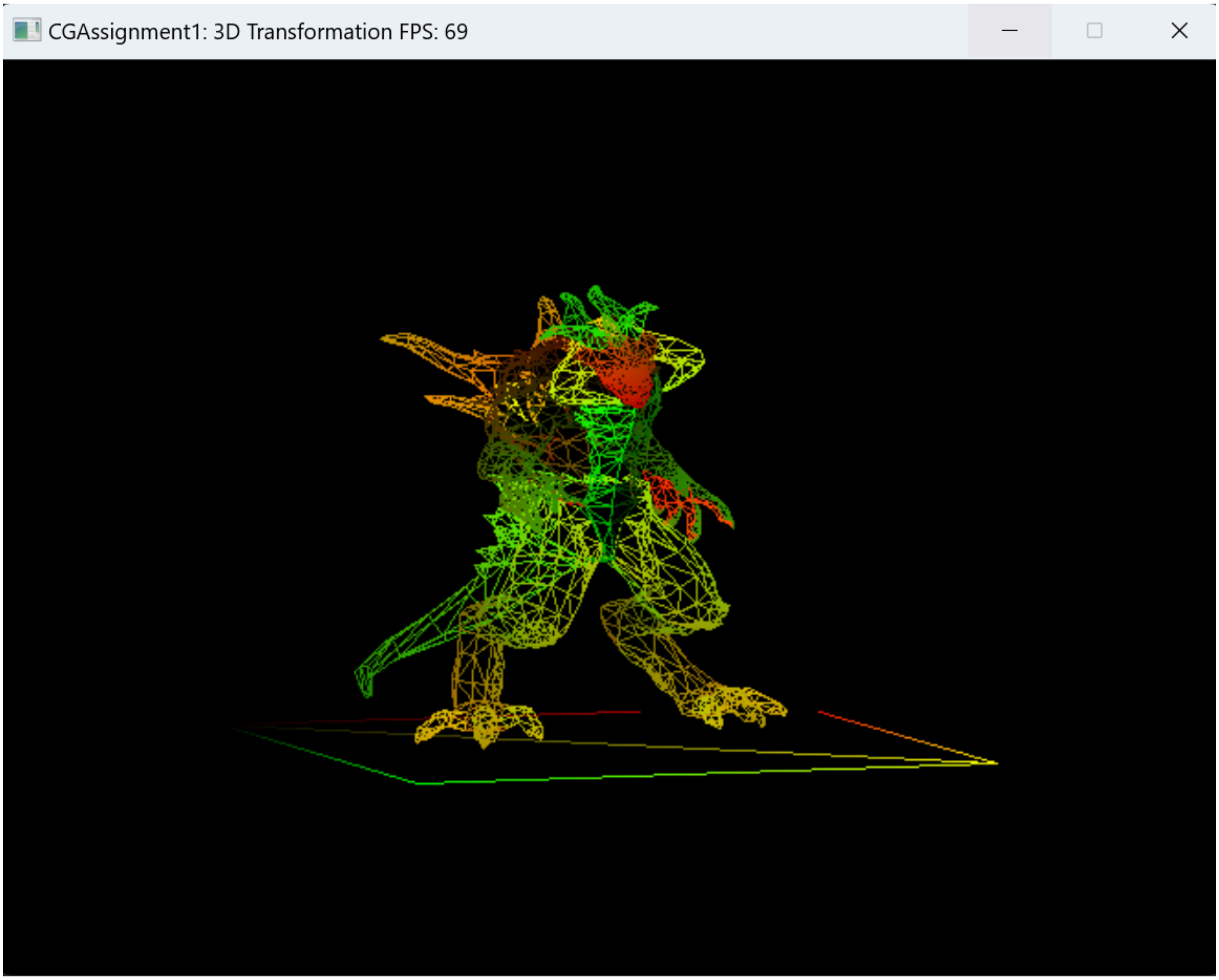
```
glm::mat4 TRRenderer::calcOrthoProjectMatrix(float left, float right, float bottom, float
{
	//Setup orthogonal matrix (camera space -> homogeneous space)
//Implement the calculation of orthogonal projection, and then set it to pMat
	glm::mat4 pMat = glm::mat4(1.0f);
	pMat[0][0] = 2.0f / (right - left);
	pMat[1][1] = 2.0f / (top - bottom);
	pMat[2][2] = 2.0f / (near - far);
	pMat[3][0] = (left + right) / (left - right);
	pMat[3][1] = (bottom + top) / (bottom - top);
	pMat[3][2] = (far+near) / (near - far);

	return pMat;
}
```

实验原理

推导得到正交投影矩阵:

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

效果图(视频在附件中)



效果上:正交矩阵投影相较来说物体更加扁平，并且正交投影缩小物体会使物体消失

四、选做：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。

Answer:

正交投影和透视投影的区别在于投影方式不同。

在正交投影中，物体在三维空间中的坐标将按照一个固定的方向进行投影，所有线段都被投影成与原始物体垂直的平行线。

而在透视投影中，投影是沿观察者的视线方向进行的，使得远离观察者的物体部分缩小，出现远近效果。

(2)从物体局部空间的顶点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？

Answer:

经过投影变换之后，几何顶点的坐标值需要经过:

模型空间、世界空间、相机空间、裁剪空间
、屏幕空间

其中，裁剪空间下的顶点的w值实际上是相机空间下的z坐标值（因为投影会改变z坐标），这个值决定了该顶点是否位于可见的范围内，它的值在[-w, w]的范围内，如果其值小于-w或大于w，该顶点将被裁剪掉，不参与后续处理。

(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？

Answer:

投影变换是将三维空间映射到二维屏幕上的过程，因此，投影后的几何顶点的坐标值不一定在NDC坐标系中。

透视除法是将裁剪空间下的坐标值进行归一化的过程，也就是将w坐标值除以其自身，得到新的xyz坐标值，使得最终的坐标值都在[-1, 1]的范围内。

这个过程可以保证投影变换后的线性关系仍然能够被保留，在之后进行光栅化、像素化等处理操作时能够得到正确的结果。

参考资料:

http://yangwc.com/2019/05/01/SoftRenderer-Math/