OPENGL Assignment 1 实验报告

姓名：林筱涵　　学号：22336141

Task1（基础任务）：实现观察矩阵（View Matrix）、透视投影矩阵（Project Matrix）、视口变换矩阵（Viewport Matrix）的计算。

观察矩阵（View Matrix）:观察矩阵由旋转和平移组成

```
// 计算方向向量
glm::vec3 forward =glm::normalize(camera - target); // 从目标指向摄像机的前向向量
glm::vec3 right = glm::normalize(glm::cross(glm::normalize(worldUp), forward)); // 右向向量
glm::vec3 up = glm::normalize(glm::cross(forward,right));
glm::normalize(up);
// 构造视图矩阵
glm::mat4 vMat = glm::mat4(1.0f);
vMat[0][0] = right.x;
vMat[0][1] = right.y;
vMat[0][2] = right.z;
//vMat[0][3] = 0.0f;
vMat[1][0] = up.x;
vMat[1][1] = up.y;
vMat[1][2] = up.z;
//vMat[1][3] = 0.0f;
vMat[2][0] = forward.x;
vMat[2][1] = forward.y;
vMat[2][2] = forward.z;
//vMat[2][3] = 0.0f;

// 最后一列是相机的平移，实际上是右向量、上向量和前向量的点积


vMat[3][0] = -glm::dot(right, camera);
vMat[3][1] = -glm::dot(up, camera);
vMat[3][2] = -glm::dot(forward, camera);
vMat[3][3] = 1.0f;
```

推导:

1.平移矩阵：摄像机从世界空间中某个坐标移动到原点进行的平移操作，假设某个坐标为(xw,yw,zw),则Mt=

$$\begin{bmatrix} 1 & 0 & 0 & -xw \\ 0 & 1 & 0 & -yw \\ 0 & 0 & 1 & -zw \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.旋转矩阵：平移后，旋转坐标系，使摄像机坐标系abc进一步旋转变成标准坐标系XYZ，即寻找一个矩阵Mr使得

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = Mr * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

解得

$$Mr = \begin{bmatrix} Rx & Ux & Fx & 0 \\ Ry & Uy & Fy & 0 \\ Rz & Uz & Fz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其中R为右向量,U为上向量,F为前向量

3.观察矩阵：观察矩阵就是由平移矩阵和旋转矩阵相乘组合而来，Mt·Mr则可得到观察矩阵

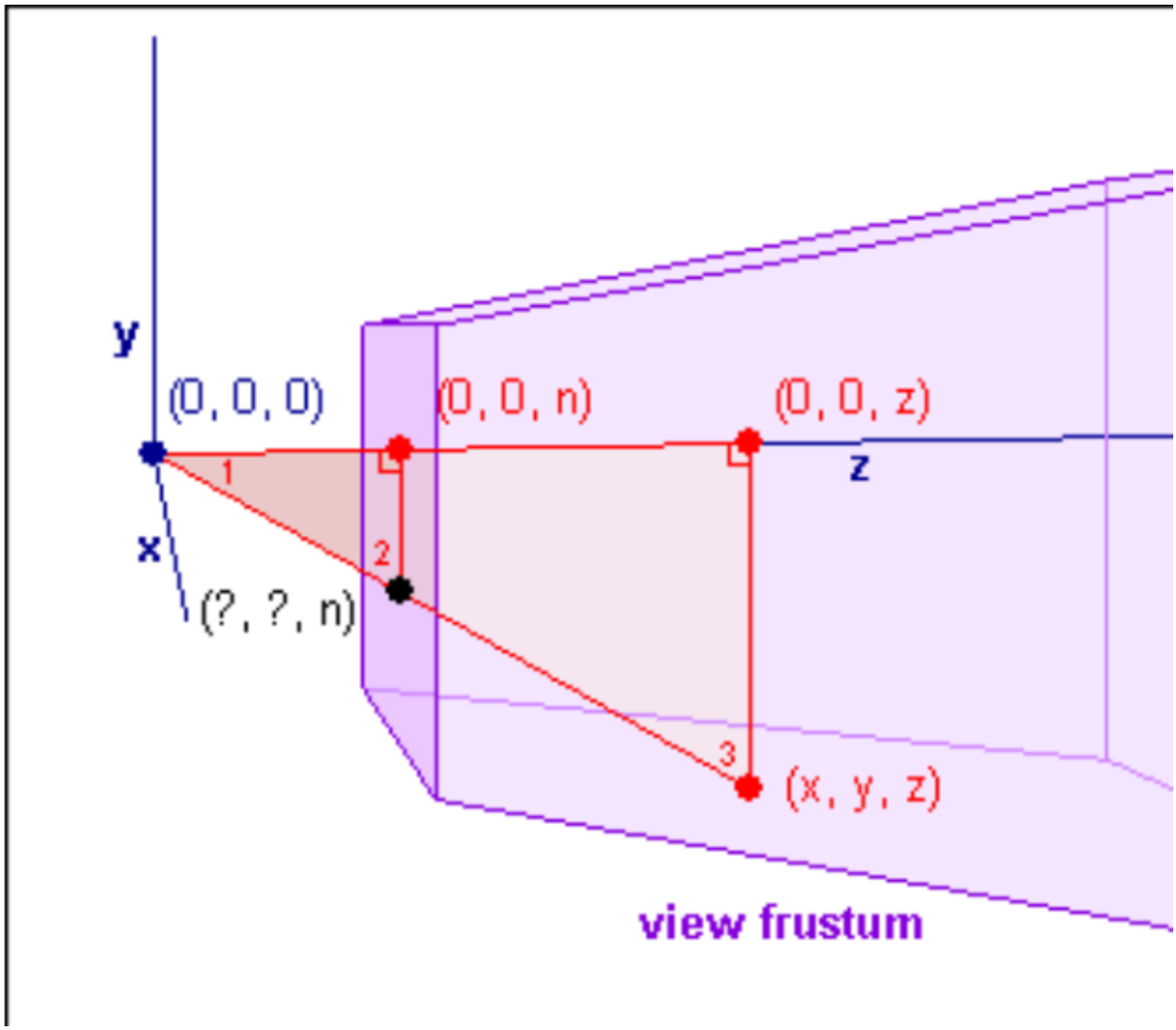
$$\begin{bmatrix} Rx & Ux & Fx & -(R\cdot W) \\ Ry & Uy & Fy & -(U\cdot w) \\ Rz & Uz & Fz & -(F\cdot w) \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{其中} w \text{为} (xw, yw, zw)$$

透视投影矩阵（Project Matrix）

```
glm::mat4 pMat = glm::mat4(0.0f);
float tanHalfFovy = tan(glm::radians(fovy)/ 2.0f);

pMat[0][0] = 1.0f / (aspect * tanHalfFovy);  // 第0列，第0行
pMat[1][1] = 1.0f / tanHalfFovy;             // 第1列，第1行
pMat[2][2] = -(near + far) / (far-near);     // 第2列，第2行
pMat[2][3] = -1.0f;                          // 第2列，第3行
pMat[3][2] = -(2.0f * near * far) / (far-near);  // 第3列，第2行
```

透视投影矩阵将三维场景中的点转投换为投影平面上的二维坐标，从而实现3D场景在屏幕上的透视效果,透视影的基本思想是将3D点(x,y,z)映射到一个齐次坐标系中的2D点(x',y',z')，并通过视角和宽高比等参数来决定投影效果。

1.将(x,y,z)投影到近平面z=n。由于投影点在近平面上，所以它的x坐标范围在[l, r]，y坐标范围在[b, t]。
参考图如下：



得x1=x*n/z;y1=y*n/z

2.把x坐标从[l, r]映射到[-1, 1]，把y坐标范围从[b, t]映射到[-1, 1]。w=r-l,h=t-b可得矩阵

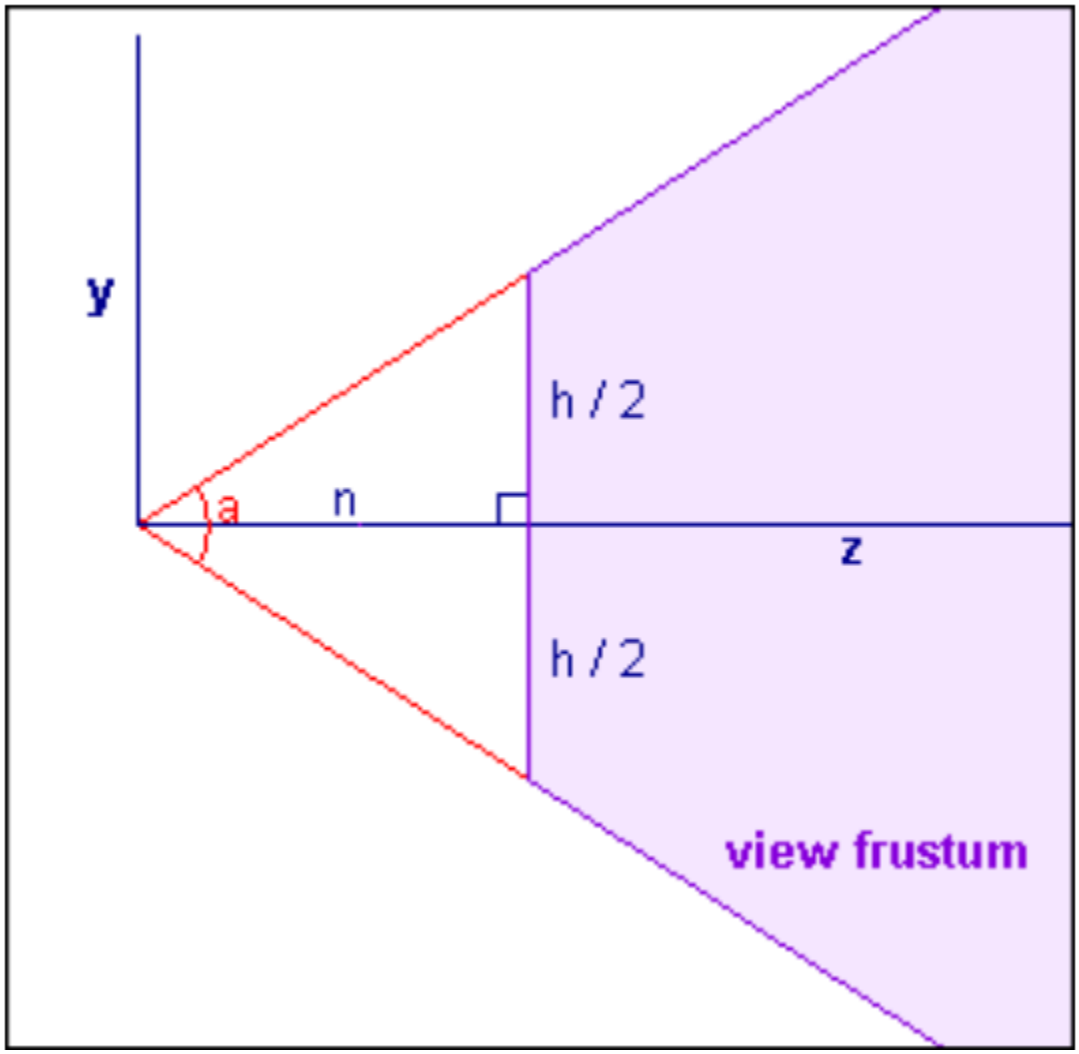
$$P = \begin{bmatrix} 2near/w & 0 & 0 & 0 \\ 0 & 2near/h & 0 & 0 \\ 0 & 0 & -(far+near)/(far-near) & -2*far*near/(far-near) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

其中near是近裁剪面距离，表示从摄像机到场景中最近可以被渲染的物体之间的距离；

far是远裁剪面的距离，表示从摄像机到场景中最远可以被渲染的物体之间的距离；

默认该视域体是对称的且中心是z轴，其中w是视域体的宽，h是视域体的高;

参考下图，可用另一种方式来表达矩阵，即视域体的高可由垂直可视范围的角度fovy定义



令tanHalfFovy=tan(fovy/2)

aspect=w/h

代入得

$$P = \begin{bmatrix} 1/(aspect * tanHalfFovy) & 0 & 0 & 0 \\ 0 & 1/tanHalfFovy & 0 & 0 \\ 0 & 0 & -(far + near)/(far - near) & -2 * far * near/(far - near) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

视口变换矩阵（Viewport Matrix）

经过视图变换和投影变换后，所有的物体位置都变换到了[-1, 1]³的标准立方体里，而视口变换需要经过两个步骤，一是缩放变换，即将标准立方体x轴和y轴长度分别缩放到width和height，由于-1到1的距离是2，所以缩放长度分别为width/2和height/2；二是平移变换，即将标准立方体的中点从原点平移到屏幕中心(width/2,height/2)。

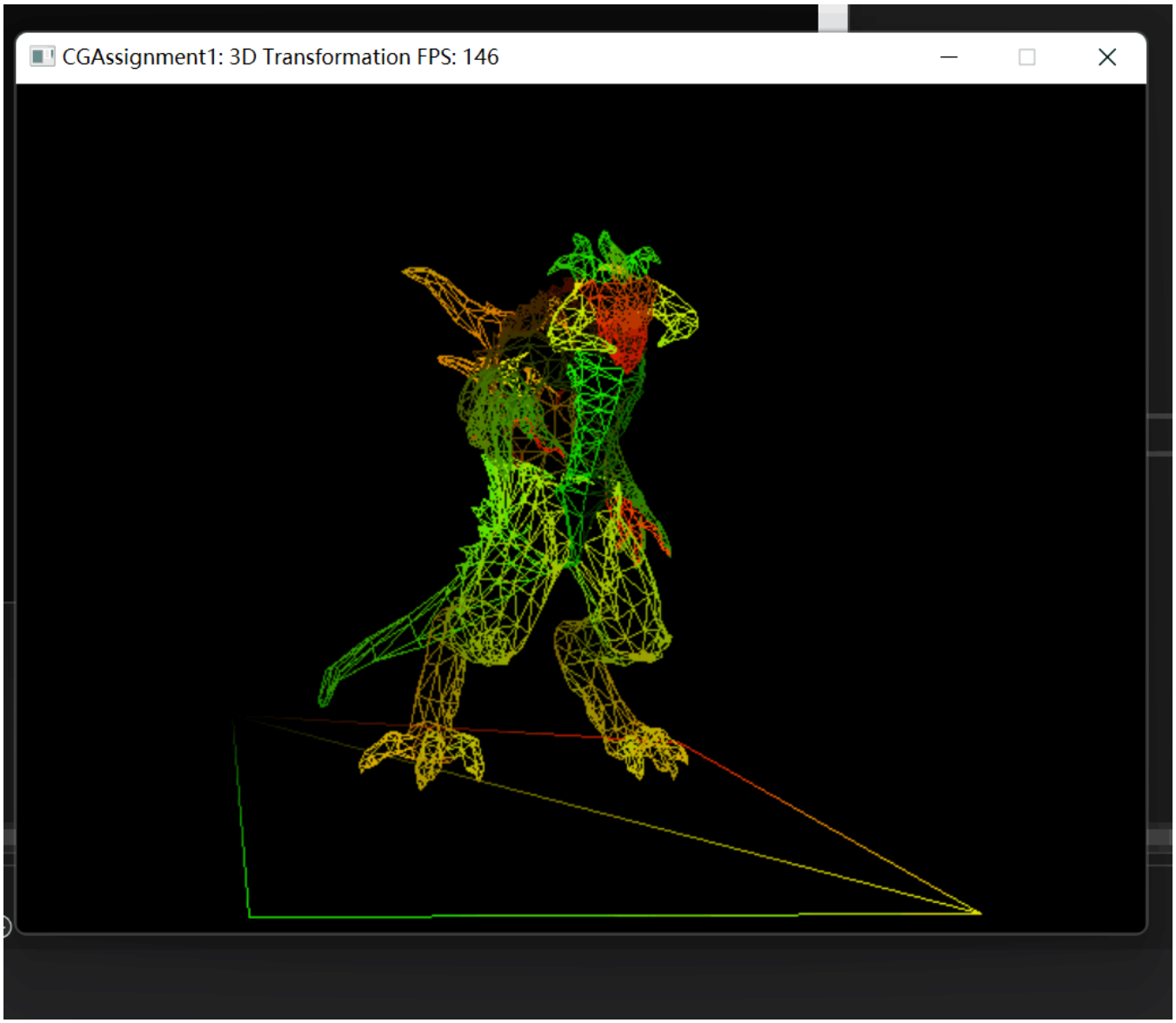
$$M = \begin{bmatrix} width/2 & 0 & 0 & width/2 \\ 0 & height/2 & 0 & height/2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

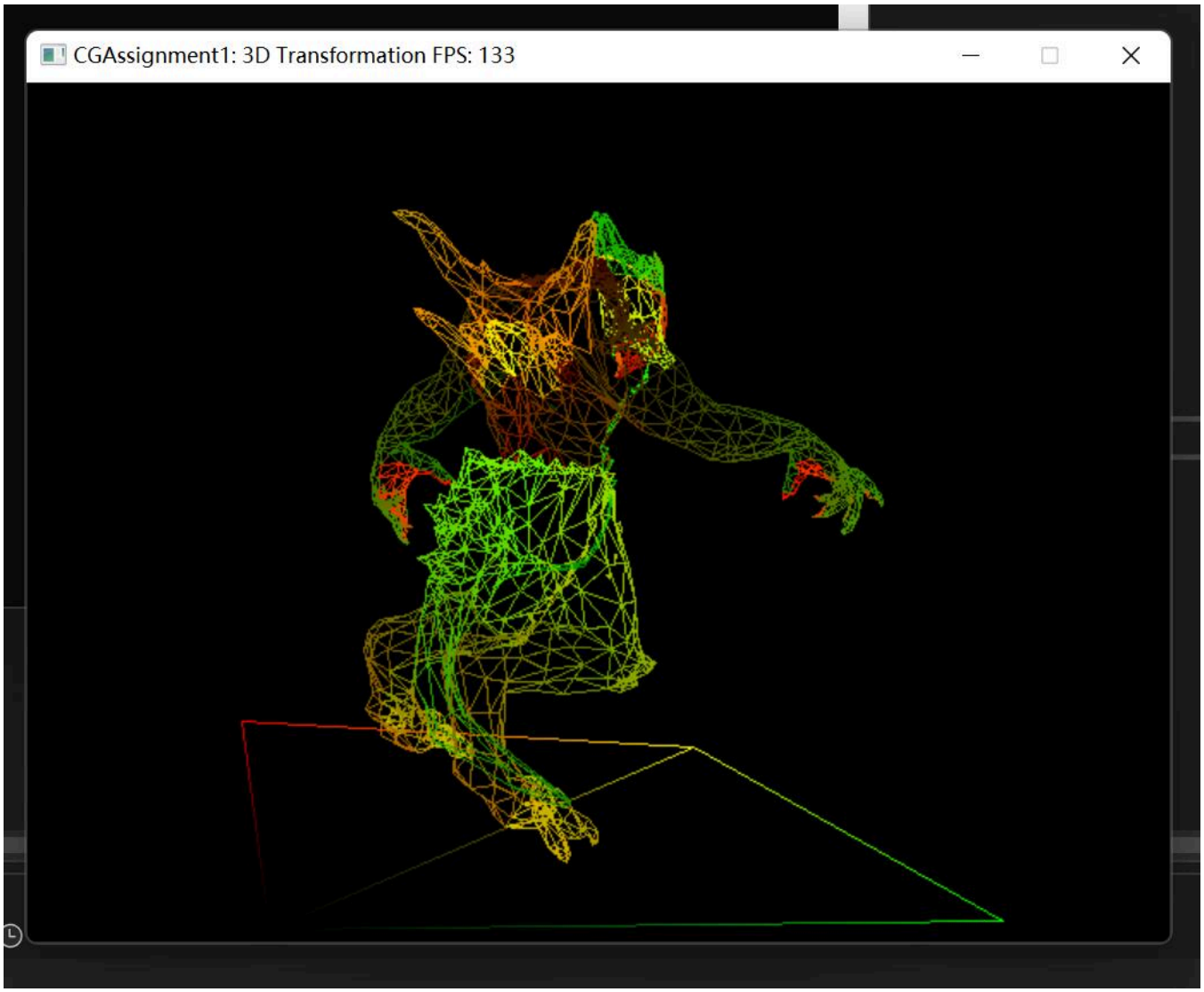
由于在操作中运行后其呈现的图片是上下颠倒的，故将M[1][1]的值改成-height/2。

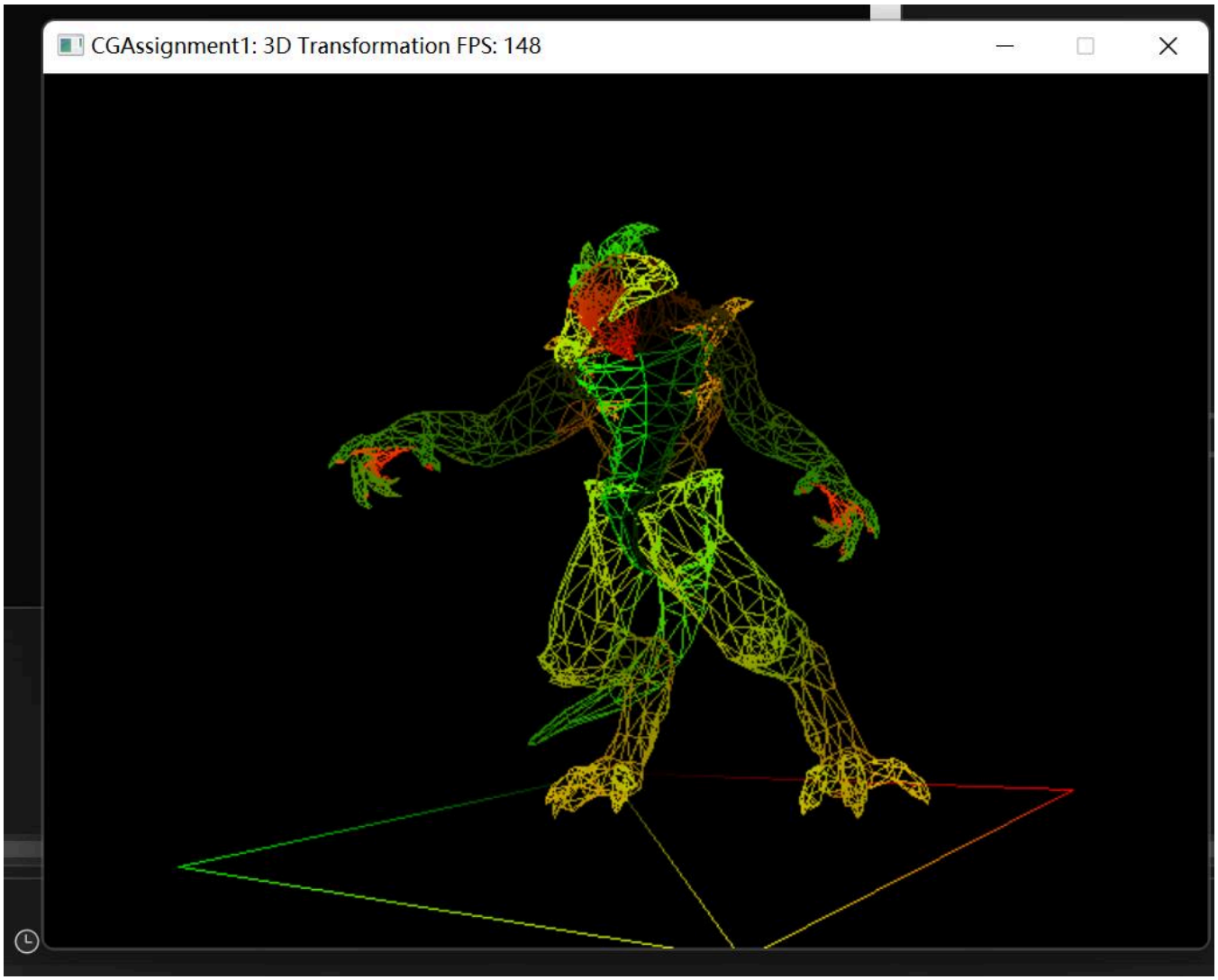
代码如下：

```
//Setup viewport matrix (ndc space -> screen space)
glm::mat4 vpMat = glm::mat4(1.0f);
vpMat[0][0] = width/2.0f;
vpMat[1][1] =- height/2.0f;
vpMat[3][0] = width/2.0f;
vpMat[3][1] = height/2.0f;
```

运行后效果图如下：




CGAssignment1: 3D Transformation FPS: 133



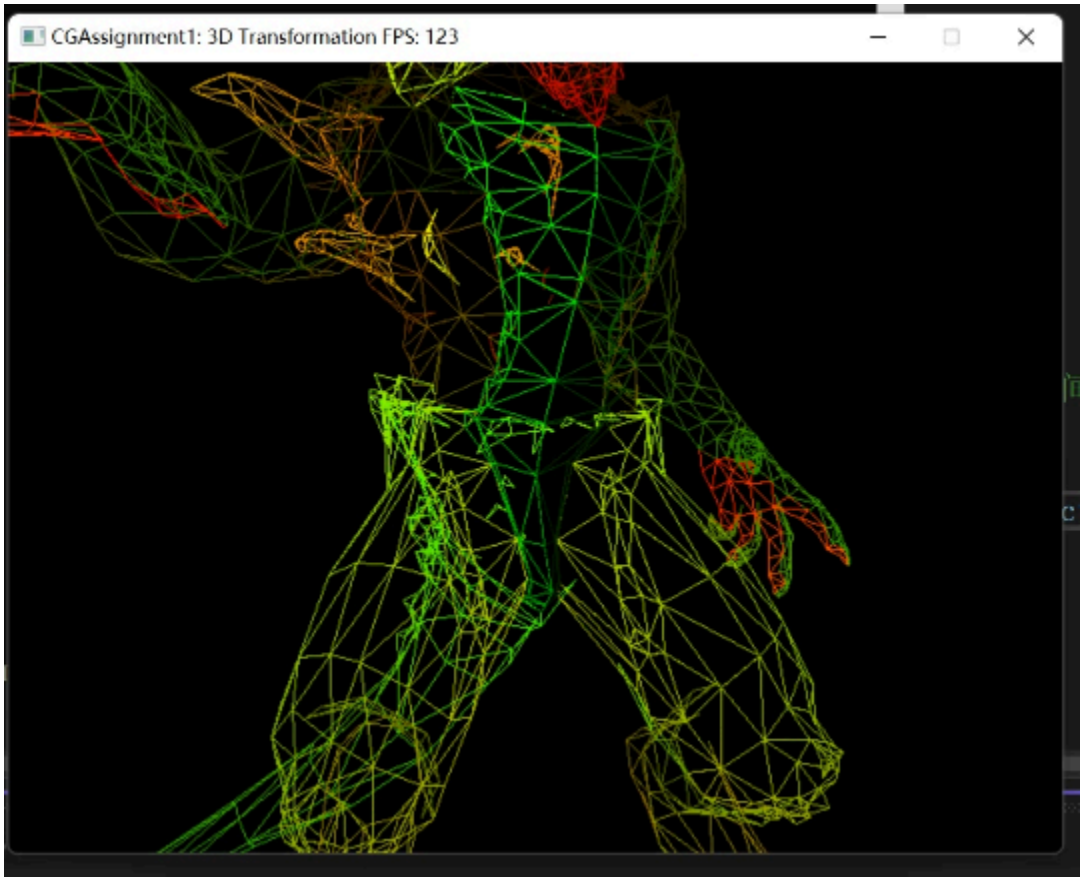
Task2(提升任务)：用 glm::scale 函数实现物体不停地放大、缩小、放大的循环动画，物体先放大至3.0倍、然后缩小至原来的大小，然后再放大至3.0，依次循环下去，要求缩放速度适中，不要太快也不要太慢。

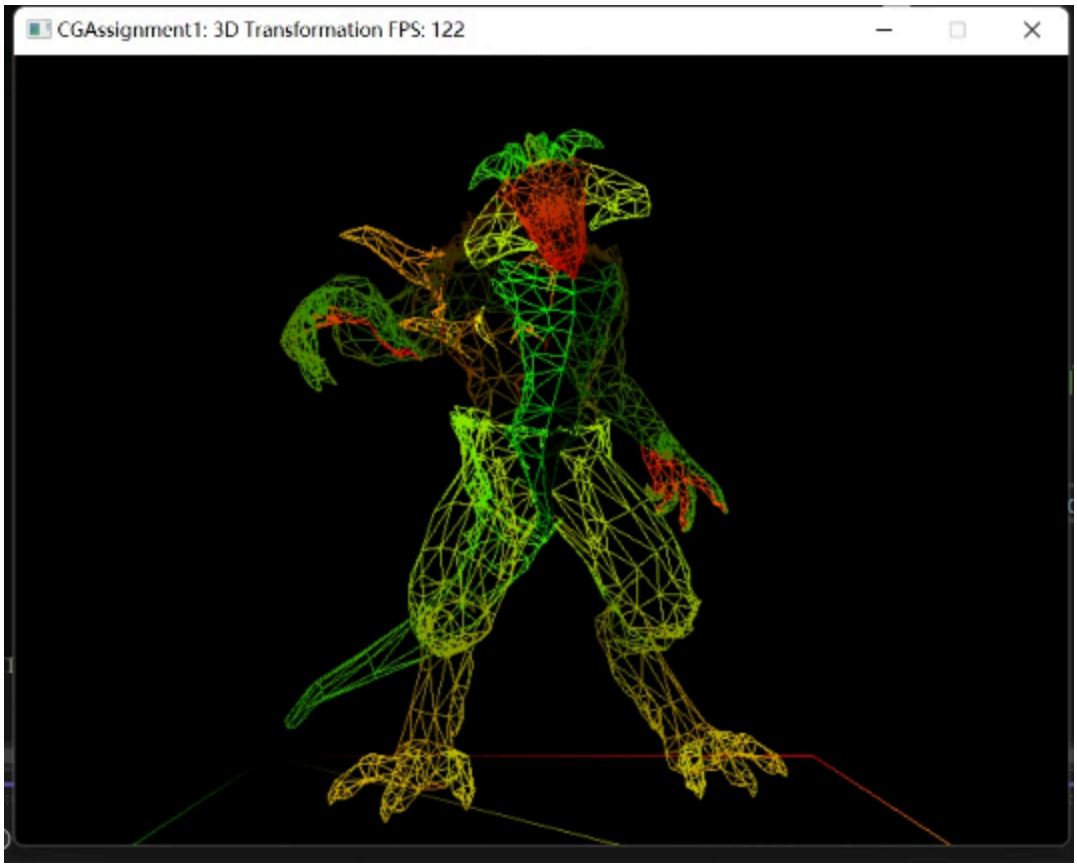
先获取当前时间，用sin() 函数返回time的正弦值，使sin(time) 随时间周期性变化，生成缩放动画，在操作过程中，出现了图像翻转的问题，故用fab()函数使缩放因子只在 [0, 1] 范围内变化，避免出现负缩放导致图像翻转的情况，最后用 glm::scale() 函数通过缩放矩阵对物体进行等比例缩放，实现物体在 1.0 倍大小和 3.0 倍大小之间的循环变化。

代码如下：

```
 // 计算当前时间
float time = getTime();  // 获取当前时间
// 使用 sin 函数，确保缩放因子只在 1.0 到 3.0 之间循环变化
float scaleFactor = 1.0f + fabs(sin(time)) * 2.0f;  // 确保缩放因子在 [1.0, 3.0] 之间

// 使用 glm::scale 函数实现缩放变换
model_mat = glm::scale(glm::mat4(1.0f), glm::vec3(scaleFactor, scaleFactor, scaleFactor));
```

运行结果如下：





Task3(挑战任务)：实现正交投影矩阵的计算，在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。

正交投影将相机空间坐标转换为齐次空间坐标，正交投影的视域体由6个面定义：

left : x = l

right : x = r

bottom : y = b

top : y = t

near : z = n

far : z = f

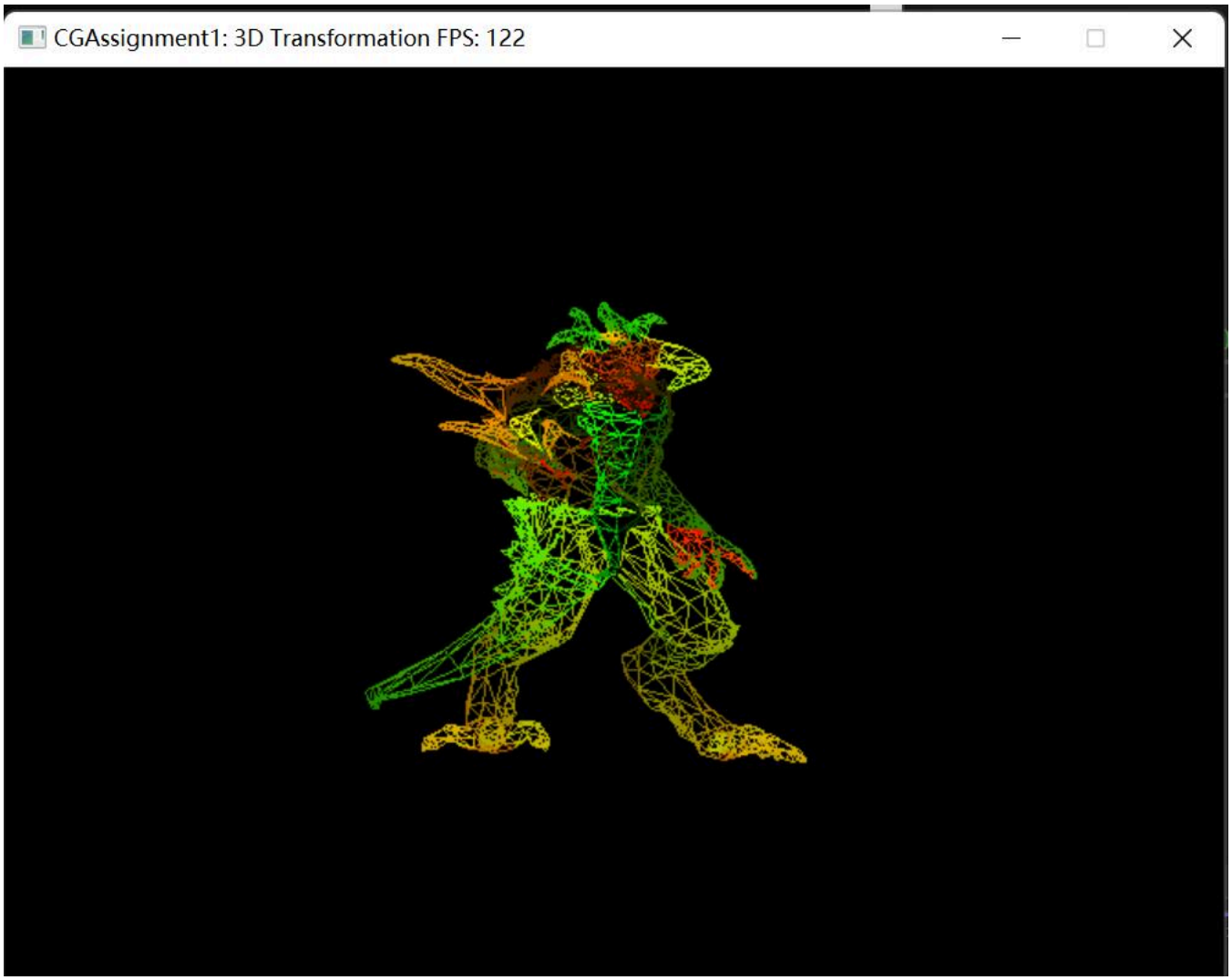
分别计算x,y,z的缩放因子和偏移量，例如对于x，先计算缩放后计算平移，x'=2.0f / (right - left)-(right + left) / (right - left);
2.0f / (right - left)是将相机空间的 x 坐标缩放到标准化设备坐标[-1, 1]中的范围。
而-(right + left) / (right - left)则表示x轴方向的平移，平移范围使其以0为中心，而不是一端为0。
以此类推，可得出正交投影的矩阵：

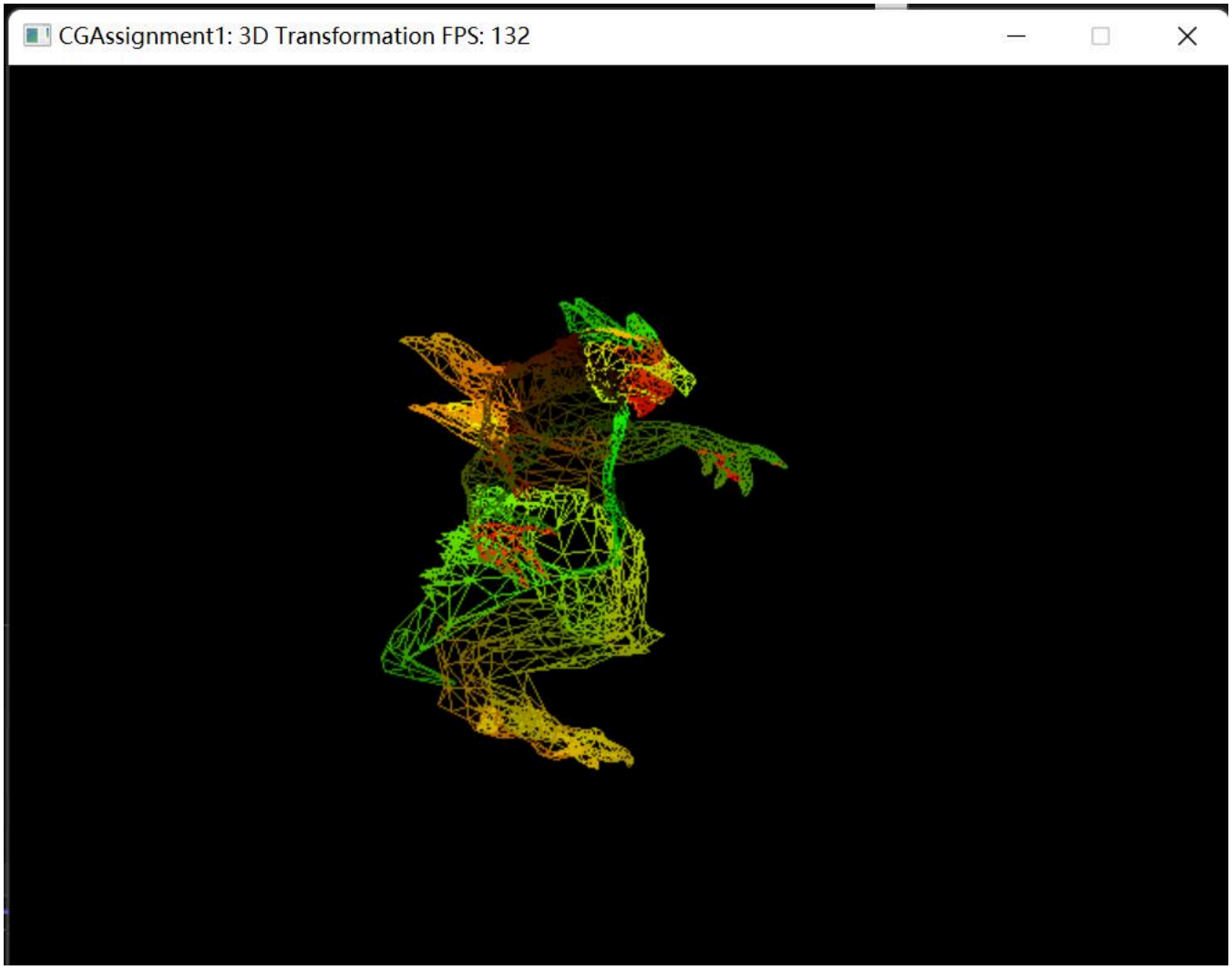
$$P0 = \begin{bmatrix} 2/(right-left) & 0 & 0 & -(right+left)/(right-left) \\ 0 & 2/(top-bottom) & 0 & -(top+bottom)/(top-bottom) \\ 0 & 0 & 1/(far-near) & -(far+near)/(far-near) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

代码如下：

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



//Implement the calculation of orthogonal projection, and then set it to pMat
//glm::mat4 pMat(1.0f);

pMat[0][0] = 2.0f / (right - left);
pMat[1][1] = 2.0f / (top - bottom);
pMat[2][2] = 1.0f / (far - near);

pMat[3][0] = -(right + left) / (right - left);
pMat[3][1] = -(top + bottom) / (top - bottom);
pMat[3][2] = -(far + near) / (far - near);
```

用正交投影时，运行结果如下：


CGAssignment1: 3D Transformation FPS: 122



通过运行结果，可以看出正交投影所呈现的图像更圆整，且当用鼠标滚轮滚动时，其图像大小不会变化，对于正交投影而言，不论物体离视点的距离如何，它们的大小在投影中是恒定的，即无缩放效果。而透视投影会根据物体与摄像机的距离缩放物体的大小，近距离的物体会得到较大的投影，远处的物体则会显得更小，有缩放效果。