Assignment 1: 3D Transformation

学号	姓名
20337025	崔璨明

Task1

实现观察矩阵（View Matrix）的计算，该函数在`TRRenderer.cpp`文件中，在`main.cpp`中被调用，简述你是怎么做的。

首先，在`calcViewMatri()`函数中，传入的参数`camera`是照相机位置（观察位置），参数`target`是观察的物体的位置，参数`worldUp`是世界向上向量。

首先计算观察平面法向量$\overrightarrow{N}$，即观察平面的z_{view}轴，计算方式是用物体位置减去照相机位置，然后进行归一化得到单位向量：

```
glm::vec3 z = glm::normalize(target - camera);
```

接着确定x_{view}，计算方式是将照相机方向向量与世界向上向量叉乘，将得到的结果归一化使之成为单位向量：

```
glm::vec3 t=glm::cross(z, worldUp);
glm::vec3 x = glm::normalize(t);
```

接着确定观察向上向量，即y_{view}，将x_{view}和z_{view}叉乘可以得到：

```
glm::vec3 y=glm::cross(x,z);
```

接下来进行世界坐标系到观察坐标系的转换，分为两个步骤：

1. 平移观察坐标系原点到世界坐标系原点
2. 进行旋转，让x_{view}、y_{view}、z_{view}对应到x_w、y_w、z_w。

设$P=(x_0,y_0,z_0)$为观察坐标系原点，则将观察坐标系原点到世界坐标系原点的变换为：

$$T_v=\begin{bmatrix}1 & 0 & 0 & -x_0\\ 0 & 1 & 0 & -y_0\\ 0 & 0 & 1 & -z_0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

将观察坐标系叠加到世界坐标系的组合旋转变换矩阵为

$$R_v = \begin{bmatrix} x_x & x_y & x_z & 0 \\ y_x & y_y & y_z & 0 \\ z_x & z_y & z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

两个矩阵相乘即可得到坐标变换矩阵：

$$M_v = RT = \begin{bmatrix} x_x & x_y & x_z & -\vec{X} \cdot P_0 \\ y_x & y_y & y_z & -\vec{Y} \cdot P_0 \\ z_x & z_y & z_z & -\vec{Z} \cdot P_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

P_0代表从世界坐标系原点到观察原点的向量，即：

$$-\vec{X} \cdot P_0 = -x_0 x_x - y_0 x_y - z_0 x_z$$

$$-\vec{Y} \cdot P_0 = -x_0 y_x - y_0 y_y - z_0 y_z$$

$$-\vec{Z} \cdot P_0 = -x_0 z_x - y_0 z_y - z_0 z_z$$

根据上述过程编写相应的代码如下：

```
vMat[0][0] = x.x;
vMat[1][0] = x.y;
vMat[2][0] = x.z;
vMat[3][0] = -glm::dot(x, camera);
//---------------------
vMat[0][1] = y.x;
vMat[1][1] = y.y;
vMat[2][1] = y.z;
vMat[3][1] = -glm::dot(y, camera);
//---------------------
vMat[0][2] = -z.x;
vMat[1][2] = -z.y;
vMat[2][2] = -z.z;
vMat[3][2] = glm::dot(z, camera);
//---------------------
vMat[0][3] = 0;
vMat[1][3] = 0;
vMat[2][3] = 0;
vMat[3][3] = 1;
```

Task2

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。简述你是怎么做的。

在函数`calcPerspProjectMatrix()`中，fovy为视角，near为近端的z坐标，far为远端的z坐标。
则透视投影规范化变换矩阵如下：

$$M_{normsymmpers} = \begin{bmatrix} \frac{cot(\frac{\theta}{2})}{aspect} & 0 & 0 & 0 \\ 0 & cot(\frac{\theta}{2}) & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & -\frac{2near-far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

其中$\theta = fovy$。

代码实现如下：

```
float tan = std::tan(fovy / 2);
pMat[0][0] = 1/(aspect*tan);
pMat[0][1] = 0;
pMat[0][2] = 0;
pMat[0][3] = 0;
//----------------
pMat[1][0] = 0;
pMat[1][1] = 1 / tan;
pMat[1][2] = 0;
pMat[1][3] = 0;
//----------------
pMat[2][0] = 0;
pMat[2][1] = 0;
pMat[2][2] = (near + far) / (near - far);
pMat[2][3]=-(2*near - far) / (near - far);
//----------------
pMat[3][0] = 0;
pMat[3][1] = 0;
pMat[3][2] = -1;
pMat[3][3] = 0;
```

Task3

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。该函数在 TRRenderer 的构造函数中被调用。简述你是怎么做的。如果Task1、Task2和Task3实现正确，那么你的运行的效果将是本文档开头贴出来的图片，而且三维物体随着程序的运行绕y轴在不停地旋转。你可以通过按住鼠标左键并横向拖动鼠标来旋转摄像机，通过滚动鼠标的滚轮来拉近或拉远摄像机的位置。请贴出你的实现效果。

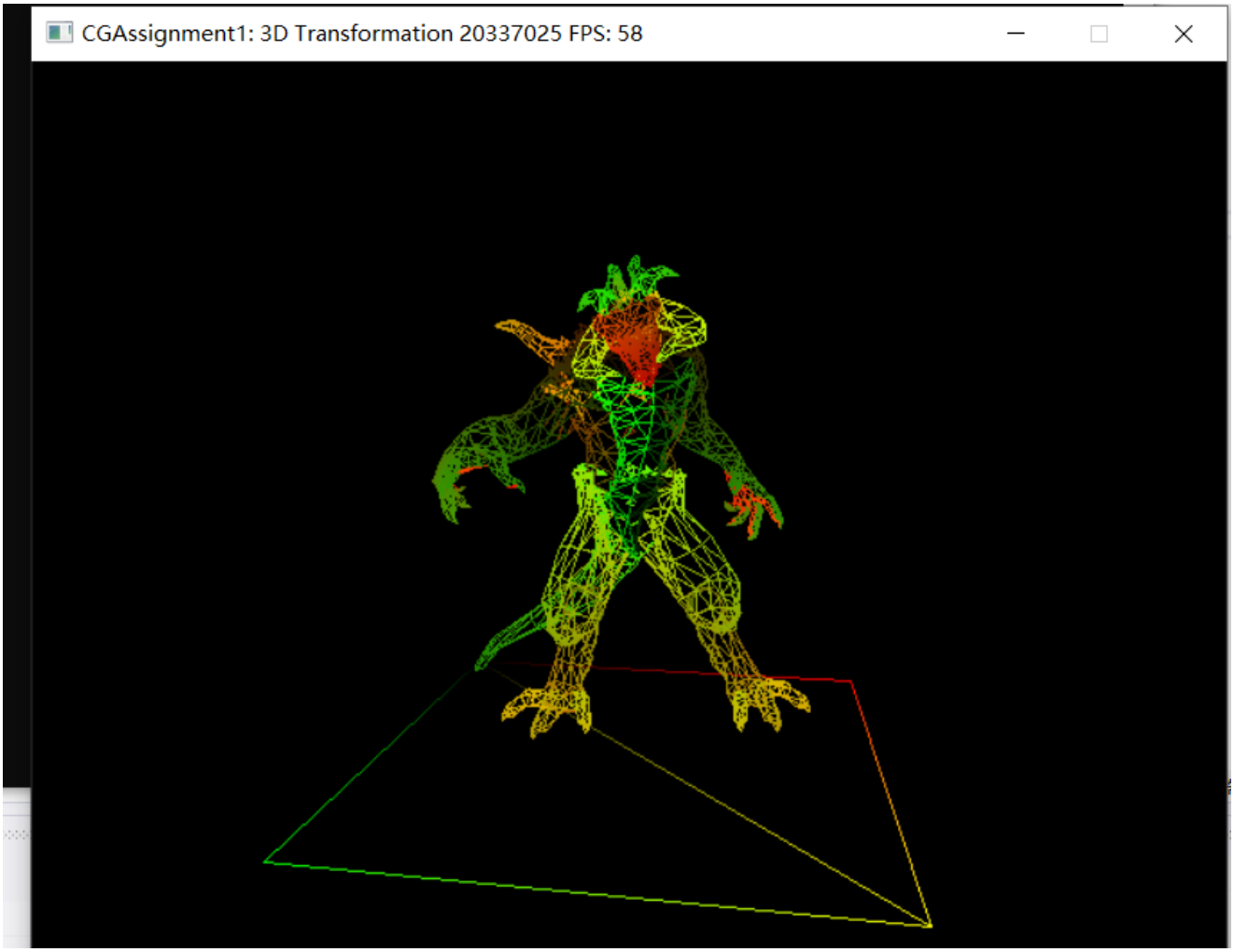
视口变换矩阵将$[-1,1]*[-1,1]$的物体转换到$[width]\times[0,\text{height}]$的屏幕坐标系上，具体计算如下：

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

代码如下：

```
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
        vpMat[0][0] = width / 2.0f;
        vpMat[1][0] = 0;
        vpMat[2][0] = 0;
        vpMat[3][0] = width / 2.0f;

        vpMat[0][1] = 0;
        vpMat[1][1] = -height / 2.0f;
        vpMat[2][1] = 0;
        vpMat[3][1] = height / 2.0f;

        vpMat[0][2] = 0;
        vpMat[1][2] = 0;
        vpMat[2][2] = 1/2;
        vpMat[3][2] = 1/2;

        vpMat[0][3] = 0;
        vpMat[1][3] = 0;
        vpMat[2][3] = 0;
        vpMat[3][3] = 1;

        return vpMat;
}
```

运行代码，效果如下：



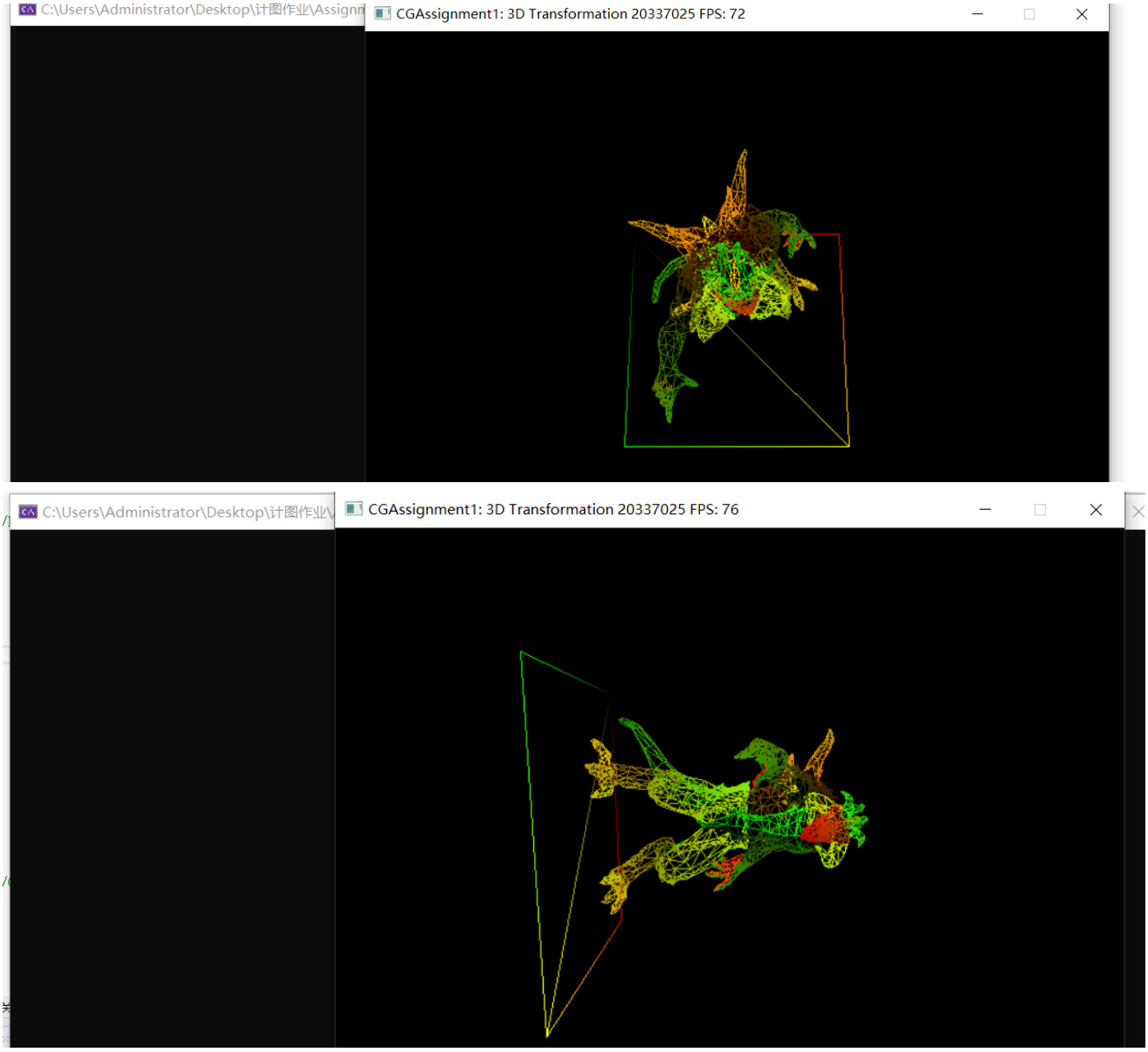
Task4

程序默认物体绕着y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。

只需要修改该行代码即可：

```
//绕x轴旋转：
model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f, glm::vec3(1,
0, 0));
//绕y轴：
model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f, glm::vec3(0,
1, 0));
//绕z轴：
model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f, glm::vec3(1,
0, 0));
```

效果截图如下：



Task5

仔细体会使物体随着时间的推进不断绕y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）

通过提示可知，可以使用`glm::scale`函数进行缩放，如`trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));`，代表将trans在xyz三个方向上都放大0.5倍。

编写代码如下：

```
float k = 1.0005f;//每次放大的倍数
bool change = true;//true为放大，false为缩小
float now = 1;//记录当前缩放的倍数
```

```
while (!winApp->shouldWindowClose()){
        ...
                //Scale
                {
                        //Task 5: Implement the scale up and down
animation using glm::scale function
                        float mul = 1;
                        //放大
                        if (change) {
                                //使用deltaTime计算每一帧放大的倍数
                                mul = std::powf(k, (float)deltaTime);
                                now *= mul;
                                //
                                if (now >= 2) {
                                        change = false;
                                        mul /= now / 2;
                                }
                        }
                        //缩小
                        else {
                                mul = std::powf(1/k, (float)deltaTime);
                                now *= mul;
                                if (now <= 1) {
                                        change = true;

                                        mul /= now;
                                }
                        }
                        model_mat = glm::scale(model_mat, glm::vec3(mul,
mul, mul));
                }
                //根据倍数缩放
                renderer->setModelMatrix(model_mat);
        }
        ...
}
```

效果如附件的动态图所示。

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。简述你是怎么做的。

根据正投影的规范化矩阵：

$$M_{ortho,norm} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & -\frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{near+far}{near-far} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

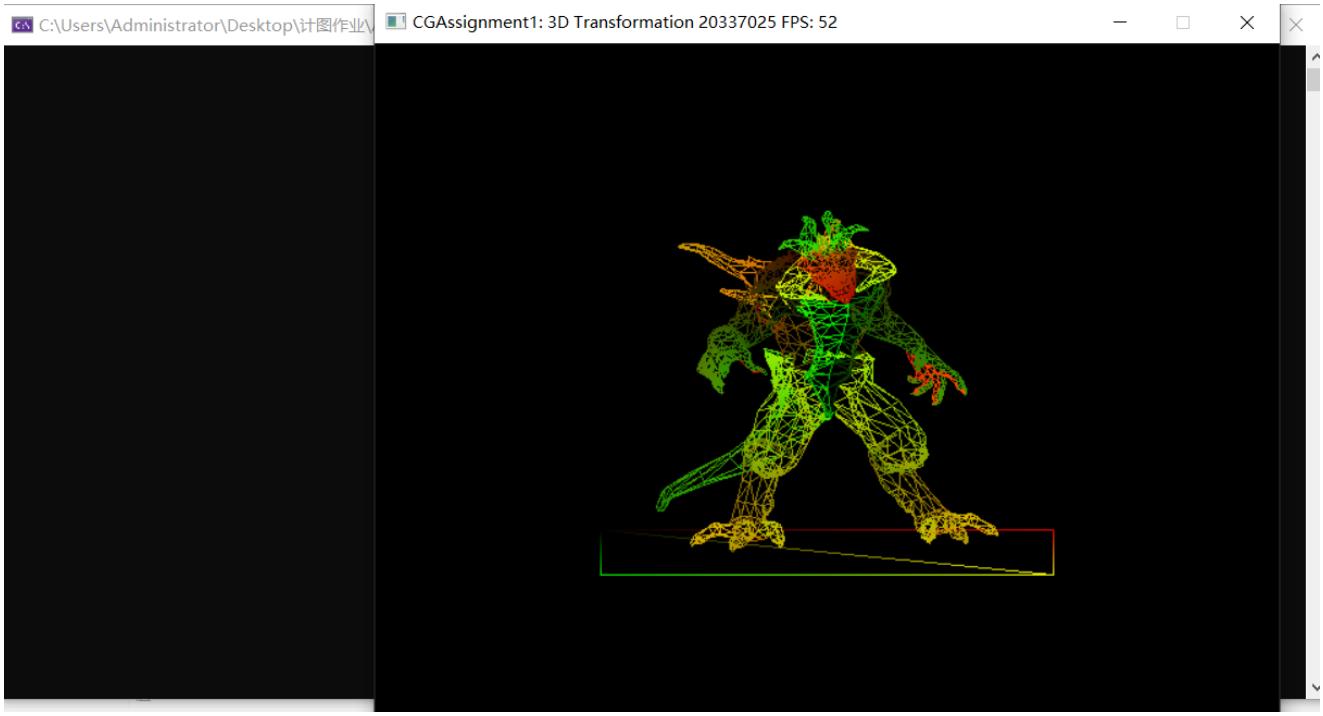
编写相应的代码：

```
//Task 6: Implement the calculation of orthogonal projection, and then set
it to pMat
pMat[0][0] = 2 / (right - left);
pMat[1][0] = 0;
pMat[2][0] = 0;
pMat[3][0] = -(right + left) / (right - left);
//-------------------------
pMat[0][1] = 0;
pMat[1][1] = 2 / (top - bottom);
pMat[2][1] = 0;
pMat[3][1] = -(top + bottom) / (top - bottom);
//-------------------------
pMat[0][2] = 0;
pMat[1][2] = 0;
pMat[2][2] = -2 / (far - near);
pMat[3][2] = (near + far) / (near - far);
//-------------------------
pMat[0][3] = 0;
pMat[1][3] = 0;
pMat[2][3] = 0;
pMat[3][3] = 1;
```

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。

使用正交投影时，移动滚轮的效果始终如下：



而使用透视投影时，移动滚轮会放大和缩小。

这说明透视投影中则会随着摄像机的远近而缩小放大，而正交投影的物体的大小不随着摄像机的远近而改变。

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。
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？
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？

答：
1）正交投影变换用一个长方体来取景，并把场景投影到这个长方体的前面。这个投影不会有透视收缩效果

透视投影变换跟也是把一个空间体（指的是以投影中心为顶点的透视四棱锥）投影到一个二维图像平面上。跟正交投影不同的是，透视投影并不保持距离和角度的相对大小不变，所以平行线的投影并不一定是平行的了。因此透视有透视收缩效果：远些的物体在图像平面上的投影比近处相同大小的物体的投影要小一些。

两种投影法的本质区别在于透视投影的投影中心到投影面之间的距离是有限的，而平行投影的投影中心到投影面之间的距离是无限的。

2）需要经过物体局部空间、世界空间、观察空间、裁剪空间、屏幕。
裁剪空间下的顶点的w值是观察空间下的z值，w = -z。其意义是对xyz进行缩放。

3）几何顶点的坐标值不是被直接变换到了NDC坐标，而是要先输出在Clip Space，然后GPU自己做透视除法变到了NDC（取值范围[-1,1]）。

透视除法就是将Clip Space顶点的4个分量都除以w分量，经过透视除法后，在视锥范围内的顶点的x与y会被映射到[-1，1]，而z会被映射到[0，1]，这样就能很方便地直接进行剔除，不在这个范围内的顶点就被剔除掉，并且也能够通过映射后的结果计算顶点在屏幕上的坐标。