Assignment 1: 3D Transformation

22336226

王泓沣

基础任务

1. 观察矩阵（View Matrix）实现

Source Code:

```
    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);

        //Implement 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::dot) except glm::lookAt

        //zAxis
        glm::vec3 forward = glm::normalize(target-camera);
        //xAxis
        glm::vec3 right = glm::normalize(glm::cross(worldUp, forward));
        //yAxis
        glm::vec3 up = glm::cross(forward, right);

        vMat[0][0] = right.x;
        vMat[1][0] = right.y;
        vMat[2][0] = right.z;
        vMat[0][1] = up.x;
        vMat[1][1] = up.y;
        vMat[2][1] = up.z;
        vMat[0][2] = forward.x;
        vMat[1][2] = forward.y;
        vMat[2][2] = forward.z;
        vMat[3][0] = -glm::dot(right, camera);
        vMat[3][1] = -glm::dot(up, camera);
        vMat[3][2] = glm::dot(forward, camera);
        vMat[0][3] = 0;
        vMat[1][3] = 0;
        vMat[2][3] = 0;
        vMat[3][3] = 1;
        return vMat;
    }
```

Parameter Analysis:

`camera`：照相机位置；

`target`：所观察的物体的位置；

`worldUp`：世界向上向量；

`forward`：观察平面z向量的反方向，由`camera`-`target`得到；

`right`：观察平面x向量，由`forward`×`worldUp`得到；

`up`：观察平面y向量，由`forward`×`right`得到；

Transform the axis

1. 将观察坐标系原点平移到世界坐标系原点

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

2. 旋转观察坐标轴至与世界坐标轴重合

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

3. 两个矩阵相乘的得到坐标变换矩阵：

$$M_v = R_v T_v = \begin{bmatrix} x_x & x_y & x_z & -\vec{x} \cdot P_0 \\ y_x & y_y & y_z & -\vec{y} \cdot P_0 \\ z_x & z_y & z_z & -\vec{z} \cdot P_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其中

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

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

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

2. 透视投影矩阵（Project Matrix）实现

Source Code:

```
    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 tanHalfFovy = std::tan(fovy / 2.0f);

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

        return pMat;
    }
```

Parameter Analysis

`fovy`：视角；

`near`：近端z坐标；

`far`：远端z坐标；

透视投影规范化变换矩阵：

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

3. 视口变换矩阵（Viewport Matrix）实现

Source Code:

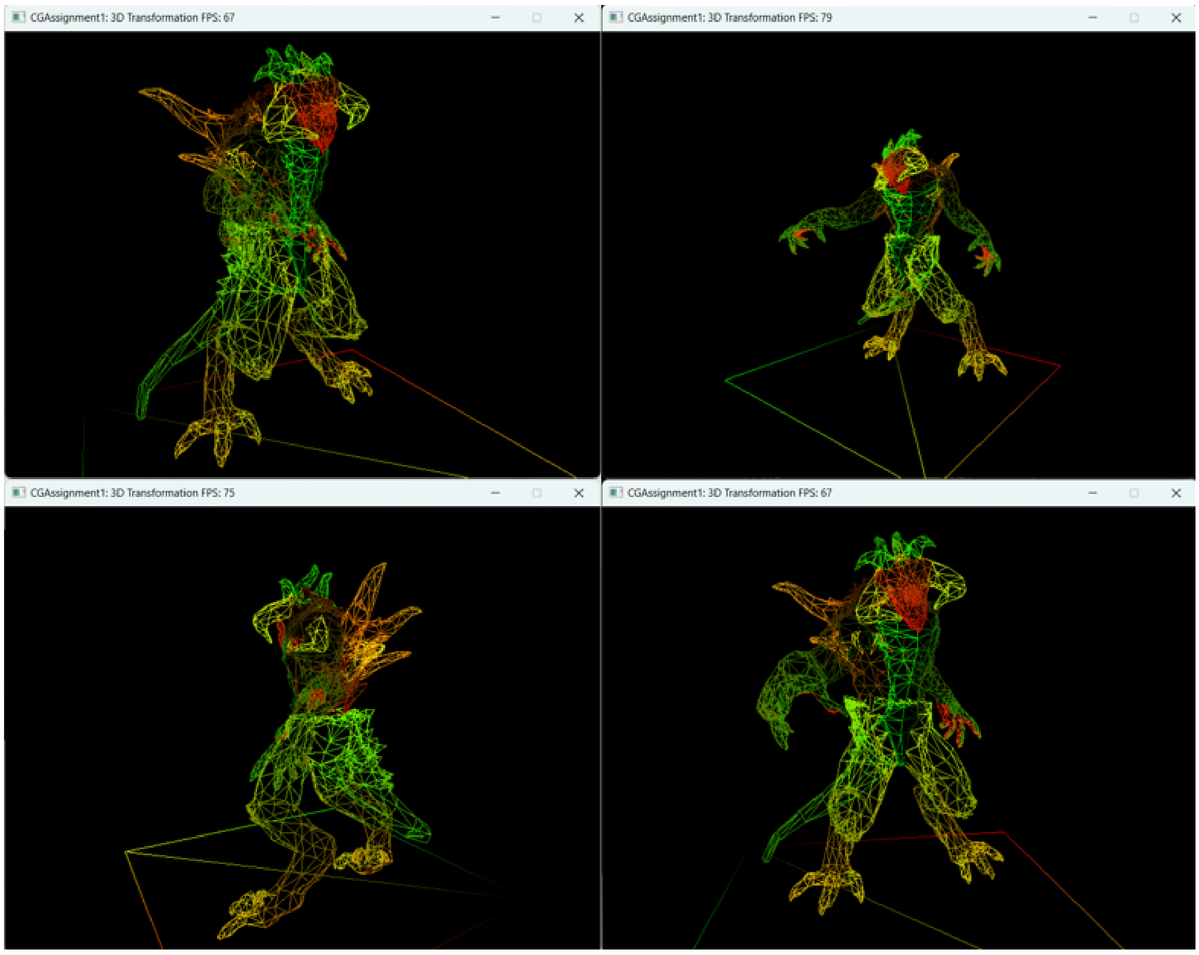
```
glm::mat4 TRRenderer::calcViewPortMatrix(int width, int height)
{
    //Setup viewport matrix (ndc space -> screen space)
    glm::mat4 vpMat = glm::mat4(1.0f);

    //Implement the calculation of viewport matrix, and then set it to vpMat
    vpMat[0][0] = static_cast<float>(width) / 2.0f;
    vpMat[1][0] = 0;
    vpMat[2][0] = 0;
    vpMat[3][0] = static_cast<float>(width) / 2.0f;

    vpMat[0][1] = 0;
    vpMat[1][1] = static_cast<float>(-height) / 2.0f;
    vpMat[2][1] = 0;
    vpMat[3][1] = static_cast<float>(height) / 2.0f;

    vpMat[0][2] = 0;
    vpMat[1][2] = 0;
    vpMat[2][2] = 0.5f;
    vpMat[3][2] = 0.5f;

    vpMat[0][3] = 0;
    vpMat[1][3] = 0;
    vpMat[2][3] = 0;
    vpMat[3][3] = 1.0f;

    return vpMat;
}
```

Parameter Analysis

将$[-1,1]\times[-1,1]$的物体转换到$[0,width]\times[0,height]$的屏幕坐标系上

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

4. 基础任务实现效果：


CGAssignment1: 3D Transformation FPS: 67
CGAssignment1: 3D Transformation FPS: 79
CGAssignment1: 3D Transformation FPS: 75
CGAssignment1: 3D Transformation FPS: 67

提升任务

1. 缩放（Scaling)

Source Code:

```
//Model transformation
{
    glm::mat4 model_mat = glm::mat4(1.0f);

    //Rotation
    {
        model_mat = glm::rotate(model_mat, (float)deltaTime * 0.0001f,
glm::vec3(0, 1, 0));
```

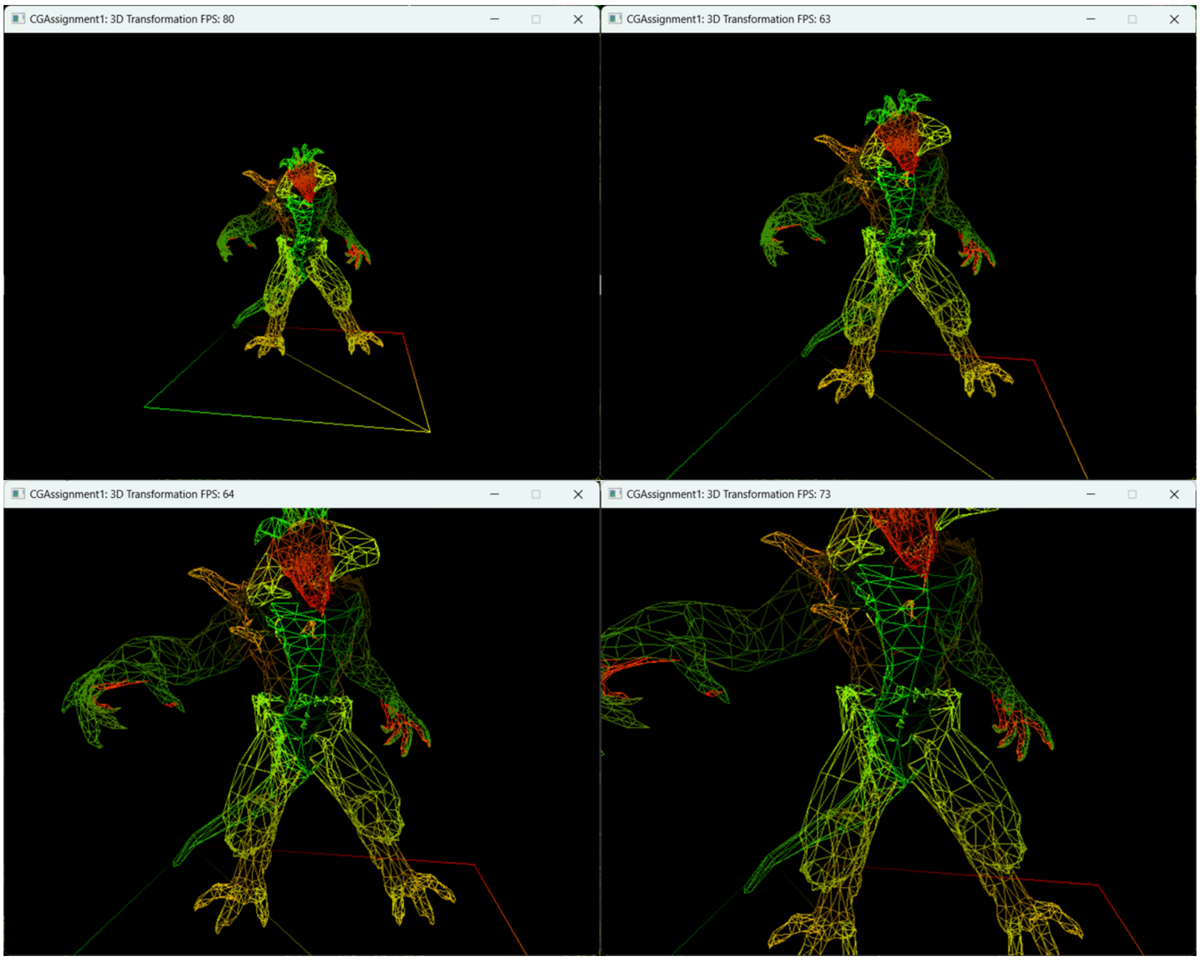
```
    }

    //Scale
    {
        //Improvement: Implement the scale up and down animation using glm::scale
function
        static float time = 0.0f;
        time += (float)deltaTime * 0.001f;

        float scaleValue = (sin(time) * 1.0f) + 2.0f;
        model_mat = glm::scale(model_mat,glm::vec3(scaleValue, scaleValue,
scaleValue));
    }

    renderer->setModelMatrix(model_mat);
}
```

Parameter Analysis

`time`：仿照rotation获取的时间

`scaleValue`：缩放比例，由sin函数实现$[1.0, 3.0]$循环

2. 提升任务实现效果


CGAssignment1: 3D Transformation FPS: 80
CGAssignment1: 3D Transformation FPS: 63
CGAssignment1: 3D Transformation FPS: 64
CGAssignment1: 3D Transformation FPS: 73

挑战任务

1. 正交投影矩阵（Orthogonal Projection Matrix）实现

Source Code:

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

        //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;



        return pMat;
    }
```

Parameter Analysis

`left`：取景长方体左边界；

`right`：取景长方体右边界；

`top`：取景长方体上边界；

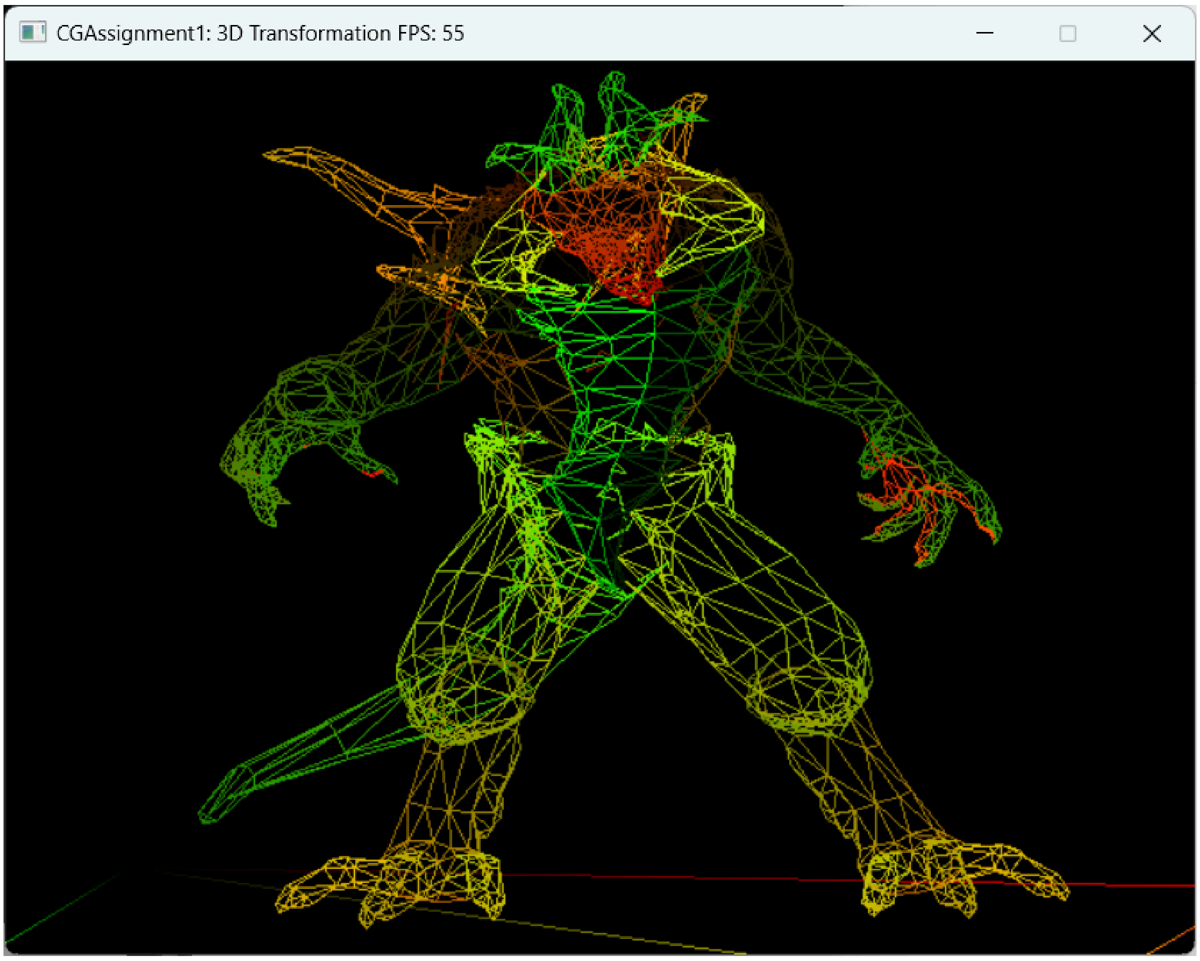
`bottom`：取景长方体下边界；

`far`：远裁剪面；

`near`：近裁剪面

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

2. 挑战任务实现效果



使用正交投影时移动滚轮不会对物体的大小产生影响，而使用透视投影时会产生影响，即照相机的位置会对透视投影方式产生影响，而不会对正交投影方式产生影响。

选做

（1）请简述正交投影和透视投影的区别。

正交投影变换用一个取景长方体取景，并把场景投影到这个长方体前面，投影没有透视收缩效果；透视投影变换是把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，有透视收缩效果。

（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？

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x、y、z进行缩放。

（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，是先输出在裁剪空间，然后由GPU做透视除法变到NDC，透视除法是将裁剪空间顶点的4个分量都除以w分量，将x、y映射到$[-1, 1]$，z映射到$[0, 1]$的操作。