计算机图形学课程作业

Assignment1

3D-Transformation

姓 名: 李 昊 伟
学 号: 20337056
班 号: 20 级计科一班

计算机图形学
(秋季, 2022)

中山大学
计算机学院
SYSUCSE

2022 年 10 月 9 日

目 录

1 概述

本次作业报告主要是对 8 个 task 的完成过程的介绍以及知识点的解析。作业程序代码可访问本人 github 代码仓库：https://github.com/distancemay5/learn_cg。

2 作业任务解答

2.1 关于坐标系转化的知识点自我梳理

一个 n 维空间的坐标系，代表着一组基底向量，基底向量的个数为 n。

比如说三维空间中的一个向量可以用一组基底 (e_1, e_2, e_3) 的线性组合来表示，即 $v = ae_1 + be_2 + ce_3$ ，也可以用另一组基底 (e'_1, e'_2, e'_3) 来表示，即 $v = a'e'_1 + b'e'_2 + c'e'_3$，其中 $e_1, e_2, e_3, e'_1, e'_2, e'_3 \in R^3$。

如果基底两两正交，那么成为一组正交基底。如果要把用一组基底 (e_1, e_2, e_3) 表示的向量用另外一组基底 (e'_1, e'_2, e'_3) 表示，那么我们可以认为是要做这样一件事情：$P = a_1\vec{e_1} + a_2\vec{e_2} + a_3\vec{e_3} = a'_1\vec{e_1}' + a'_2\vec{e_2}' + a'_3\vec{e_3}'$

已知 $e_1, e_2, e_3, e'_1, e'_2, e'_3, a_1, a_2, a_3$，求 a'_1, a'_2, a'_3

2.1.1 坐标变换的一般形式

用矩阵的形式表示:

$$\begin{bmatrix} \vec{e'_1} & \vec{e'_2} & \vec{e'_3} \end{bmatrix} \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} = \begin{bmatrix} \vec{e_1} & \vec{e_2} & \vec{e_3} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (1)$$

两边同时左乘，解方程得:

$$\begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} = \begin{bmatrix} \vec{e'_1} & \vec{e'_2} & \vec{e'_3} \end{bmatrix}^{-1} \begin{bmatrix} \vec{e_1} & \vec{e_2} & \vec{e_3} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (2)$$

2.1.2 对位姿概念的引入

本部分参考中山大学计算机学院专选课《机器人导论》部分PPT内容

我们已经知道，三维向量和三维点可以统一使用齐次坐标来进行表示，表示为一个四维向量。

形如 $[x, y, z, 0]$ 这样的向量是一个三维向量，因为最后一位恒为 0，所以若用表示平移的变换矩阵乘之，则每次向量内积加法最后一个数字恒为 0，换句话说平移操作对其没有意义。

形如 $[x, y, z, 1]$ 这样的向量是一个三维点，因为最后一位恒为 2，所以若用表示平移的变换矩阵乘之，则每次向量内积加法最后一个数字为偏移量 bias，可以达到平移的目的。

三维齐次坐标系是由四个四维向量组成的坐标系：

$\begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \vec{p} \end{bmatrix}$

前四个向量表示的是三维向量，也就是 x , y , z 三个轴的方向，最后一位是 0，最后一个向量表示的是一个三维点，即该坐标系原点的位置，最后一位是 1。在机器人学中，该四维矩阵表示的是一个机械臂的位置 (position) 和姿态 (pose)。

则 1.1 中的坐标变换可以表示为:

$$\begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \\ 1 \end{bmatrix} = \begin{bmatrix} \vec{e}'_1 & \vec{e}'_2 & \vec{e}'_3 & \vec{p}' \end{bmatrix}^{-1} \begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \vec{p} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} \quad (2)$$

也就是从一种位置和姿态 (位姿) 的坐标系转化到另外一种位姿的坐标系。

2.2 Task1 实现观察矩阵（View Matrix）的计算

2.2.1 问题分析

我们可以把摄像机看作一个” 机械臂”，然后求它的位置和姿态，假设一开始在世界坐标系中摄像机的位置为 (0,0,0)，姿态为三个方向和世界坐标系的三个方向重合，那么从世界坐标系到摄像机坐标系的转化过程，可以认识为摄像机位姿的转变过程，就可以用上 (2) 式中的变换，求 $\begin{bmatrix} \vec{e}'_1 & \vec{e}'_2 & \vec{e}'_3 & \vec{p}' \end{bmatrix}^{-1} \times \begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \vec{p} \end{bmatrix}$ 即可。

第一步我们要找到摄像机坐标系的位姿矩阵，首先找三个基底向量：首先是摄像机坐标系的 z 轴，也就是 f 轴，由摄像机的位置和目标物体的位置确定，是由目标物体指向摄像机的向量 (因为摄像机看向 f 轴的负方向)，这个向量首先确定:

```
glm::vec3 f = glm::vec3(glm::normalize(target - camera)); //
    这里的f等下会取反
```

然后是摄像机坐标轴的 x 轴，也就是 s 轴，由 WordUp 和 f 轴确定，s 轴与 WorfUp 垂直，可以用二者的叉乘得到:

```
glm::vec3 s = glm::vec3(glm::normalize(cross(worldUp,f)));
```

最后是摄像机坐标轴的 y 轴，也就是 u 轴，和 f，s 垂直即可：

```
glm::vec3 u = glm::vec3(glm::normalize(cross(s, f)));
```

得到这一组基底之后,再加上位置点向量 $[camera_x, camera_y, camera_z, 1]^T$,就可以得到 $\begin{bmatrix}\vec{e}_1' & \vec{e}_2' & \vec{e}_3' & \vec{p}'\end{bmatrix}$

由于原来的摄像机坐标系同世界坐标系重合:

所以 $\begin{bmatrix}\vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \vec{p}\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$

所以只需要求: $\begin{bmatrix}\vec{e}_1' & \vec{e}_2' & \vec{e}_3' & \vec{p}'\end{bmatrix}^{-1}$

观察易得, $\begin{bmatrix}\vec{e}_1' & \vec{e}_2' & \vec{e}_3' & \vec{p}'\end{bmatrix}^{-1} = (\begin{bmatrix}1 & 0 & 0 & p_1\\ 0 & 1 & 0 & p_2\\ 0 & 0 & 1 & p_3\\ 0 & 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}e_{11} & e_{21} & e_{31} & 0\\ e_{12} & e_{22} & e_{32} & 0\\ e_{13} & e_{23} & e_{33} & 0\\ 0 & 0 & 0 & 1\end{bmatrix})^{-1} =$

$$\begin{bmatrix}e_{11} & e_{12} & e_{13} & 0\\ e_{21} & e_{22} & e_{23} & 0\\ e_{31} & e_{32} & e_{33} & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}1 & 0 & 0 & -p_1\\ 0 & 1 & 0 & -p_2\\ 0 & 0 & 1 & -p_3\\ 0 & 0 & 0 & 1\end{bmatrix}$$

可以看出，如果把这个矩阵看作变换矩阵的话，其实是先做旋转变换再做平移变换。求逆的时候，旋转变换都是正交矩阵，可以直接转置的到逆，平移矩阵直接把平移量取反即可。

2.2.2 完整代码

```
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);

    //Task 1: 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
    glm::vec3 f = glm::vec3(glm::normalize(target - camera));
    glm::vec3 s = glm::vec3(glm::normalize(glm::cross(worldUp,f)));
    glm::vec3 u = glm::vec3(glm::normalize(glm:: cross(s, f)));
    vMat[0][0] = s.x;
    vMat[1][0] = s.y;
    vMat[2][0] = s.z;
    vMat[0][1] = u.x;
    vMat[1][1] = u.y;
```

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

2.3 Task2 实现透视投影矩阵（Project Matrix）的计算

2.3.1 问题分析

对于透视投影矩阵，如果不从整体上认识它，就会觉得很麻烦。我们先要认识到，透视投影矩阵做了哪些事情。其实透视投影矩阵无非两个功能，一个是把平截头体中的点投影到 near 这个平面上，为了后续的计算，我们保留了深度 z 的信息；另外一个就是把所有点的所有坐标归一化到 [-1,1] 这个区间。我们可以分开来思考这两个功能是如何实现的。

首先是投影。与正交投影不同，透视投影的平截头体是一个棱台，如下图所示。我们把棱台的前后左右上下六个面分别命名为：n,f,l,r,t,b。我们的第一个目标是把后面棱台里面所有的点都投影到 n 这个平面上。我们先不考虑保留深度信息的问题，也就是我们先考虑投影后的点的 x,y 的值。

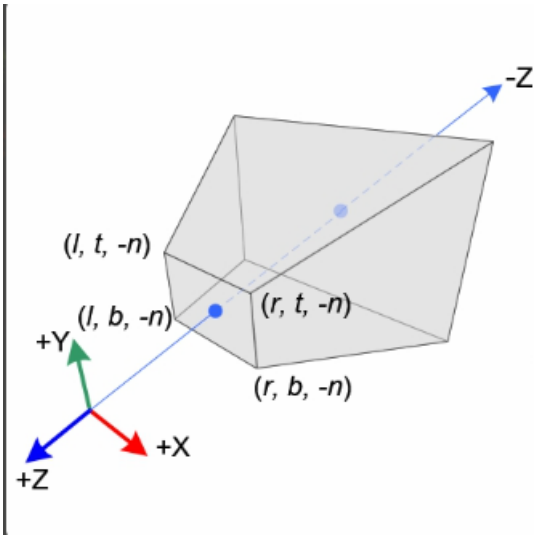


图 1: 透视投影的平截头体

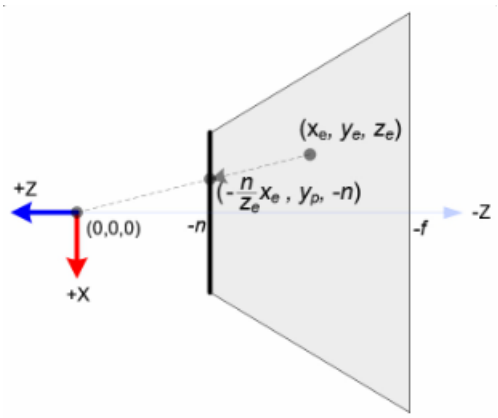


图 2: 向-x 方向看到的平截头体

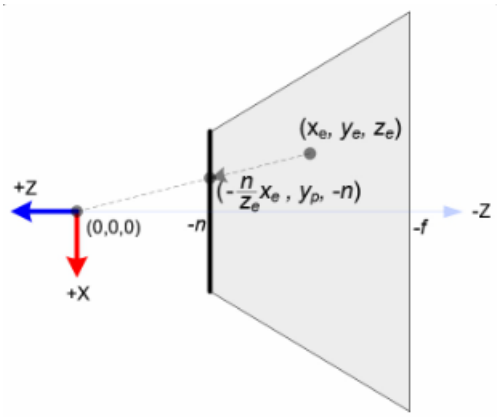


图 3: 向-y 方向看到的平截头体

由上面两张图片可以看出，利用相似三角形的知识可以轻易地求出 x 和 y 的值：在视锥体中的顶点 (x_e, y_e, z_e) 被投影到视锥体的近平面，近平面上的点我们记为 (x_p, y_p, z_p) 。如图 2 和图 3 所示，根据三角形相似的原理，我们有：

$\frac{x_p}{x_e} = \frac{n}{-z_e}$ -> $x_p = -\frac{nx_e}{z_e}$

$\frac{x_p}{x_e} = \frac{n}{-z_e}$ -> $y_p = -\frac{ny_e}{z_e}$

想要将 x_e, y_e, z_e 和 x_p 以及 x_e, y_e, z_e 和 y_p 的对应关系用矩阵表示，有一个问题: 就是关系式中都存在除以 $-z_e$ 这一个操作，我们把这个操作留给透视除法来做，也就是对于一个三维向量的四维表示 (x, y, z, w)，等价于 $(x/w, y/w, z/w, 1)$，如果我们的变换能够使得输入的向量 $w = -z_e$，那么透视除法就会帮我们完成除以 $-z_e$ 这个操作。这时候，我们至少知道变换矩阵的最后一行了，变换矩阵最起码是这样的，如果用点来代替未知值的话。

$$\begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

不要忘了，透视投影矩阵另外一个功能就是把所有点的所有坐标归一化到 [-1,1] 这个区间。首先把坐标归一化到 [0,1]。直观地想，可以把坐标减去最小值，得到宽/长度，再把这个值除以投影面的宽/长度，这个值一定在 [0,1]；将这个值乘上一个 2 再减去 1(仿射变换) 就映射到了 $[-1,1]$，得:

$x_p = -\frac{nx_e}{z_e}$

$x_c = 2 \times \frac{x_p - l}{r-l} - 1$

$x_c = (\frac{2n}{r-l}x_e + \frac{r+l}{r-l}z_e)/ - z_e$

同理：$y_c = (\frac{2n}{t-b}y_e + \frac{t+b}{t-b}z_e)/ - z_e$

于是，又可以填充矩阵：$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ . & . & . & . \\ 0 & 0 & -1 & 0 \end{bmatrix}$

最后，将深度信息保留，将 z_e 直接归一化到 [-1,1] 即可，可以看出第三行前两个数字一定是 0。后两个数字的求解就是这样一个等价的问题：已知 $x_e \in [-f, -n]$，由于原点坐标系是左手坐标，摄像机坐标系是右手坐标，所以求一仿射变换 $x_c = (kz_e + b)/ - z_e$, 使得 $x_c \in [-1, 1]$

直接带入特殊值:

$-1 = (-kn + b)/n$

$1 = (-kf + b)/f$

解得:

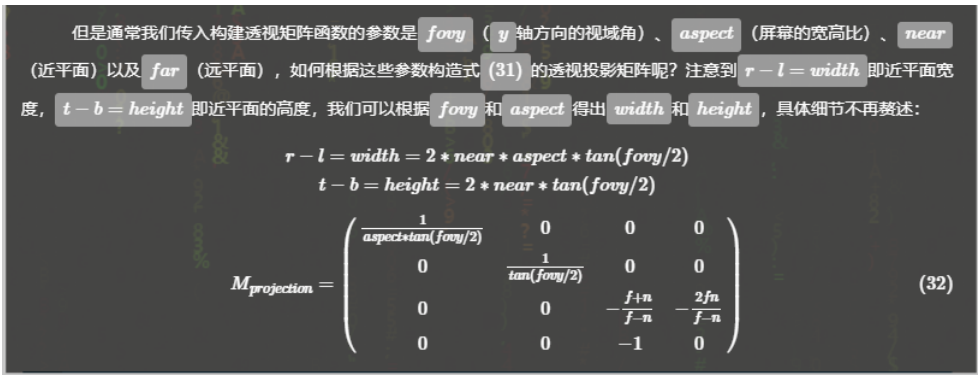
$\mathrm{k} = \frac{n-f}{n+f}$

$\mathrm{b} = \frac{2fn}{n-f}$

所以，得到最终填充的透视投影矩阵为:

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

如果传入的参数是 fov 等等，经过简单的代数运算即可得到：



但是通常我们传入构建透视矩阵函数的参数是 $fovy$（y 轴方向的视域角）、$aspect$（屏幕的宽高比）、$near$（近平面）以及 far（远平面），如何根据这些参数构造式 (31) 的透视投影矩阵呢？注意到 $r-l=width$ 即近平面宽度，$t-b=height$ 即近平面的高度，我们可以根据 $fovy$ 和 $aspect$ 得出 $width$ 和 $height$，具体细节不再赘述：

$$r-l=width=2*near*aspect*tan(fovy/2)$$
$$t-b=height=2*near*tan(fovy/2)$$
$$M_{projection}=\begin{pmatrix}\frac{1}{aspect*tan(fovy/2)} & 0 & 0 & 0\\ 0 & \frac{1}{tan(fovy/2)} & 0 & 0\\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n}\\ 0 & 0 & -1 & 0\end{pmatrix} \tag{32}$$

图 4: 透视投影变换矩阵运算公式

2.3.2 完整代码

```
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);

  //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));
  pMat[0][0] = 1.0f / (aspect * tanHalfFovy);
  pMat[1][1] = 1.0f / (tanHalfFovy);
  pMat[2][2] = -(far + near) / (far - near);
  pMat[2][3] = -1.f;
  pMat[3][2] = (-2.0f * near * far) / (far - near);
  return pMat;
}
```

2.4 Task3 实现视口变换矩阵（Viewport Matrix）的计算

2.4.1 问题分析

视口变换矩阵将归一化的坐标转化为与窗口适配的坐标。

我们现在有模型都在 [-1,1]*[-1,1]*[-1,1] 正方体中，我们想把它映射到位置 [x,x+w]*[y,y+h]*[0,d] 中，我们的操作是：

平移：先把 [-1,1]*[-1,1]*[-1,1] 平移到 [0,2]*[0,2]*[0,2]

缩放：[0,2]*[0,2]*[0,2] 缩放到 [0,1]*[0,1]*[0,1]
缩放：[0,1]*[0,1]*[0,1] 缩放到 [0,w]*[0,h]*[0,d]
平移：[0,w]*[0,h]*[0,d] 移动到 [x,x+w]*[y,y+h]*[0,d]

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

图 5: 变换矩阵组

$$\begin{pmatrix} w/2 & 0 & 0 & x + w/2 \\ 0 & h/2 & 0 & y + h/2 \\ 0 & 0 & d/2 & d/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

图 6: 变换矩阵

取 x=y=d = 0，且由于屏幕 y 轴与之前空间 y 轴反向，所以取反 y，得到平面上的坐标变换矩阵:

$$\begin{bmatrix} w/2 & 0 & 0 & w/2 \\ 0 & -h/2 & 0 & w/2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.4.2 完整代码

```
glm::mat4 TRRenderer::calcViewPortMatrix(int width, int height)
{
  //Setup viewport matrix (ndc space -> screen space)
  //Task 3: Implement the calculation of viewport matrix, and then set it
      to vpMat
  glm::mat4 vpMat = glm::mat4(1.0f);
  vpMat[0][0] = float(width/2);
  vpMat[1][0] = 0;
  vpMat[2][0] = 0;
```

```
    vpMat[3][0] = float(width / 2);
    vpMat[0][1] = 0;
    vpMat[1][1] = -float(height / 2);
    vpMat[2][1] = 0;
    vpMat[3][1] = float(height / 2);
    vpMat[0][2] = 0;
    vpMat[1][2] = 0;
    vpMat[2][2] = 0;
    vpMat[3][2] = 0;
    vpMat[0][3] = 0;
    vpMat[1][3] = 0;
    vpMat[2][3] = 0;
    vpMat[3][3] = 1;
    return vpMat;
}
```

2.5 Task1-3 结果

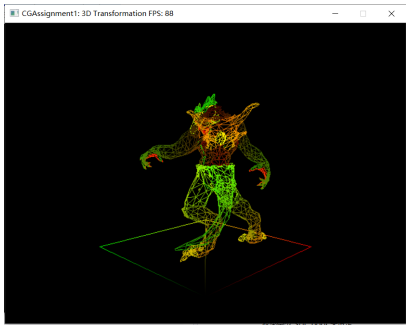


图 7: 正常大小

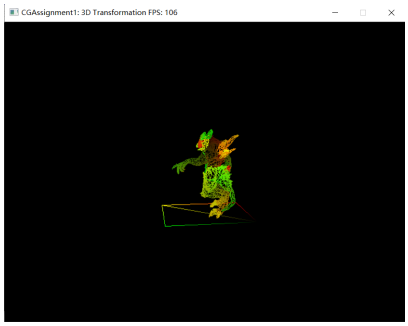


图 8: 滚轮缩小

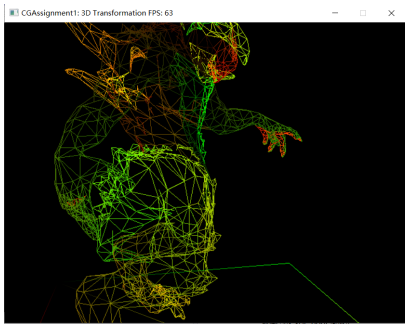


图 9: 滚轮放大

2.6 Task4 使得物体分别绕 x 轴和 z 轴旋转

2.6.1 问题分析

这个问题很简单，只需要在生成 rotate 矩阵的时候，把第三个参数改为 (1,0,0) 向量或者 (0,0,1) 向量即可。输出结果视频见附件。

2.6.2 完整代码

```
model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f,
    glm::vec3(0, 0, 1));
```

2.7 Task5 实现物体不停地放大、缩小、放大的循环动画

2.7.1 问题分析

使用 detaTime 的累加作为当前时间，为了避免数据溢出，当到达 sin 函数的周期 2π 时，将当前时间清零。

```
  double totalTime = 0;
while (!winApp->shouldWindowClose())
{
  //Process event
  if (abs(totalTime - M_PI) == 0.0001) {
    totalTime = 0;
  }
\\剩余渲染循环代码
  }
```

然后，将缩放倍率设置为 $1 + abs(sin((float)totalTime * 0.001f))$，使得这个倍率从 1 变到 2 再变回 1。然后根据这个倍率设置 scale 矩阵:

```
    //Scale
```

```
    {
        glm::mat4 tmp = glm::mat4(1.0f);
        float size = 1 + abs(sin((float)totalTime * 0.001f));
        printf("%f", size);
        scale_mat = glm::scale(tmp , glm::vec3(size, size, size));
    }
```

2.7.2 完整代码

```
    //Rendering loop
  double totalTime = 0;
  while (!winApp->shouldWindowClose())
  {
    //Process event
    if (abs(totalTime - M_PI) == 0.0001) {
      totalTime = 0;
    }
    winApp->processEvent();


    //Clear frame buffer (both color buffer and depth buffer)
    renderer->clearColor(glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));

    //Draw call
    renderer->renderAllDrawableMeshes();

    //Display to screen
    double deltaTime =
        winApp->updateScreenSurface(renderer->commitRenderedColorBuffer(),
        width, height, 4);
    totalTime += deltaTime;
    // printf("%llf\n", deltaTime);
    //Model transformation
    {
      static glm::mat4 rotate_mat = glm::mat4(1.0f);
      static glm::mat4 scale_mat = glm::mat4(1.0f);
      //Rotation
      {
        //Task 4: Make the model rotate around x axis and z axis,
            respectively
        rotate_mat = glm::rotate(rotate_mat, (float)deltaTime * 0.001f,
            glm::vec3(0, 1, 0));
      }
      //Scale
     {
        glm::mat4 tmp = glm::mat4(1.0f);
        float size = 1 + abs(sin((float)totalTime * 0.001f));
        printf("%f", size);
        scale_mat = glm::scale(tmp , glm::vec3(size, size, size));
```

```
    }
     glm::mat4 model_mat = rotate_mat * scale_mat;
     renderer->setModelMatrix(model_mat);
   }
```

2.8 Task6、task7 实现正交投影矩阵的计算

2.8.1 问题分析

由于正交投影只是一个简单的归一化过程，同透视投影的归一化分析过程完全一样，所以直接贴示推导结果，输出结果视频见附件：

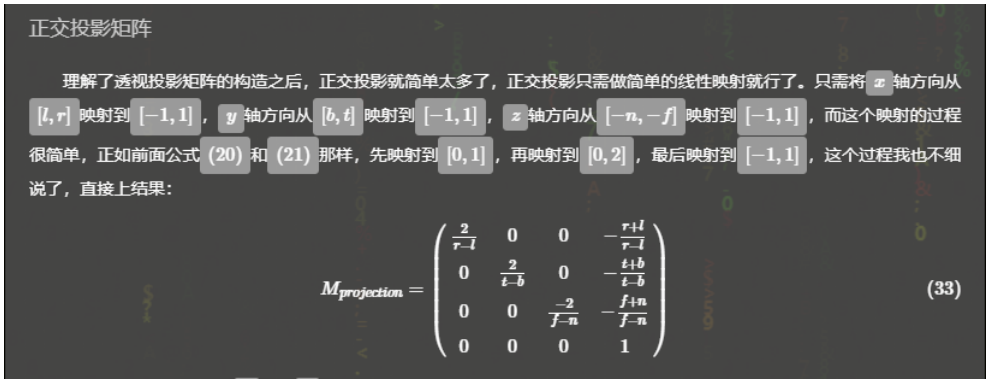


图 10: 正交投影公式推导

2.8.2 完整代码

```
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);

  //Task 6: Implement the calculation of orthogonal projection, and then
      set it to pMat
  pMat[0][0] = 2.0f / (right - left);
  pMat[1][1] = 2.0f / (top - bottom);
  pMat[2][2] = -2.0f / (far - near);
  pMat[3][0] = -(right + left) / (right - left);
  pMat[3][1] = -(top + bottom) / (top - bottom);
  pMat[3][2] = -(far + near) / (far - near);
  return pMat;
}
```

2.9 task8

2.9.1 Q1

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（远些的物体在图像平面上要小一些），因为它保证平行线在变换后仍然保持平行，也就使得物体之间的相对距离在变换后保持不变。简单的说，正交投影变换忽略物体远近时的大小缩放变化，将物体以原比例投影到截面（如显示屏幕）。

透视投影变换跟正交投影一样，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。然而，它却有透视收缩效果：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。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。换言之，透视投影变换能够实现一个物体在玩家近距离比较大，远距离比较小，那么实现这样的效果的照相机就叫做远景照相机。远景照相机常用来开发 3D 游戏，它的工作原理是根据照相机和物体之间的距离缩放投影的比例（也就是截面的大小）。透视投影跟人的眼睛或相机镜头产生三维世界的图像的原理还是很接近的。

2.9.2 Q2

需要经过以下几个坐标空间：局部空间 (Local Space，或者称为物体空间 (Object Space))，这是三维模型最初被建立的空间。

世界空间 (World Space)，这是分别建模的三维模型被放置到一起，具有了相对位置关系的空间。

观察空间 (View Space，或者称为视觉空间 (Eye Space))，这是摄影机被移动到指定位置，观察指定物体后，以摄影机的局部坐标系为坐标系的空间。

裁剪空间 (Clip Space)，这是平截头体被建立，物体被投影到 near 范围内的空间。

屏幕空间 (Screen Space)，这是最终归一化的坐标经过视口变换，变化为可以投影到电脑屏幕上的坐标的空间。

裁剪空间下的顶点的 w 值是观察空间 View Space 坐标下的 z 值，如同 2.3.1 中的分析推导：

$\frac{x_p}{x_e} = \frac{n}{-z_e}$ -> $x_p = -\frac{nx_e}{z_e}$

$\frac{x_p}{x_e} = \frac{n}{-z_e}$ -> $y_p = -\frac{ny_e}{z_e}$

想要将 x_e, y_e, z_e 和 x_p 以及 x_e, y_e, z_e 和 y_p 的对应关系用矩阵表示，有一个问题: 就是关系式中都存在除以 $-z_e$ 这一个操作，但是矩阵变换只能表

示加权和这样的操作。我们把这个操作留给透视除法来做，也就是对于一个三维向量的四维表示 (x, y, z, w)，等价于 $(x/w, y/w, z/w, 1)$，如果我们的变换能够使得输入的向量 $w = -z_e$，那么透视除法就会帮我们完成除以 $-z_e$ 这个操作。z 代表的是一个点距离摄像机的距离，由于观察空间的 z 轴指向摄像机，所以-z 就是点到摄像机的距离。

2.9.3 Q3

还没有到 ndc 空间，还有一个 w 也就是 z_e 没有除。在投影矩阵作用后的向量中，第四个分量 w 来自于视空间的 z，透视除法消除了齐次坐标的影响，同时把 z 归一化，经过这一步，才能把坐标转到 ndc 空间。透视除法将 Clip Space 顶点的 4 个分量都除以 w 分量，就从 Clip Space 转换到了 NDC 了。

透视除法直接的来讲，就是把向量 (x,y,z,w) 变成 (x/w,y/w,z/w,1)，将 2.3.1 中所说的不适合用矩阵处理的操作部分完成，将坐标转到 ndc 空间。