三维视角投影变换与代码实战

Dezeming Family

Dezeming

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

0.1 本书前言 5

1 投影变换的原理 6

1.1 模型变换的流程 6

1.2 模型空间到世界空间 7

1.3 世界空间到相机空间 7

1.4 正交投影与透视投影 9

1.5 正规视体到屏幕空间 12

2 投影变换的 PBRT 源码实现 13

2.1 PBRT 中的透视投影实现 13

2.2 GLM 的矩阵投影变换 17

Literature 20

前言及简介

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列书，可以从我们的网站 *[https://dezeming.top/]* 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

0.1 本书前言

我还是觉得要结合代码来描述一下矩阵视角变换这种图形学基础入门技术。我会首先详细介绍生成视角变换矩阵的各种细节和原理（包括逆变换），然后再介绍代码中是怎么实现的。

这里我选择两个源码来解释（尽管一个就够了）。一个是大家常用的 glm 图形数学库，另一个就是我正在写的 PBRT 系列书的代码描述了。

本书的售价是 4 元（电子版），我们不直接收取任何费用，如果本书对大家学习有帮助，可以往我们的支付宝账户（17853140351）进行支持，您的赞助将是我们 Dezeming Family 继续创作各种计算机视觉、图形学、机器学习、以及数学原理小册子的动力！

2021-05-22：完成并发布第一版。

2021-06-23：更新第二版。修改了 lookAt 矩阵生成的一些错误；补充了 glm 中的列主矩阵带来的易错点；补充了一下最后的 glm::perspective 函数的原理。

1. 投影变换的原理

1.1 模型变换的流程 6

1.2 模型空间到世界空间 7

1.3 世界空间到相机空间 7

1.4 正交投影与透视投影 9

1.5 正规视体到屏幕空间 12

by WLOP

本节介绍模型视角变换的原理，包括空间变换的过程和模型视角矩阵生成的细节。

1.1 模型变换的流程

当我们在描述一个物体位置时，比如，你的面前有一台电脑，你的左手边有一杯水，这是以你的视角进行描述的。但我们还可以描述为，电脑在房间的北墙边上；再往大了说，我们甚至可以用经纬度来描述一个电脑的位置。甚至我们还可以以太阳为坐标原点，描述某时刻电脑在太阳系中的位置。

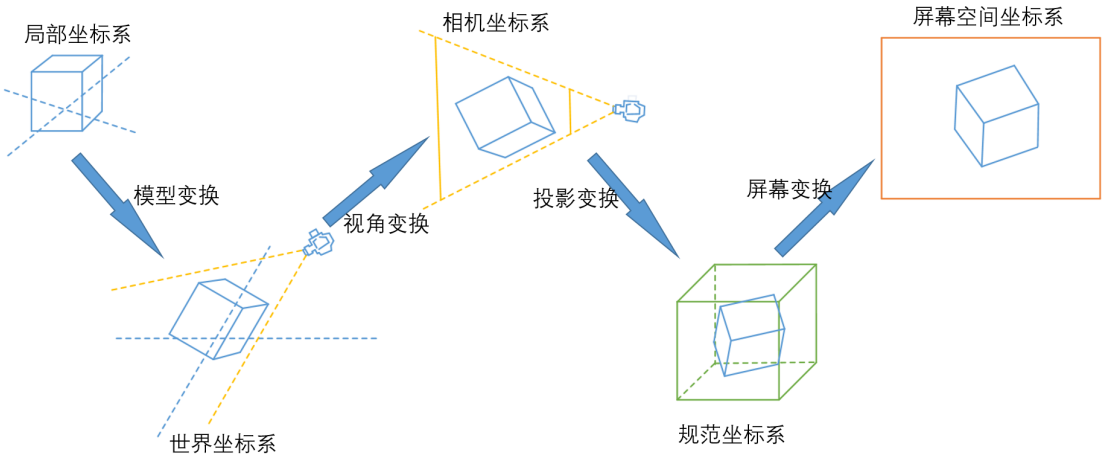
那么严格来说，描述物体位置时，我们都需要一个参考点作为坐标原点，描述某物体的位置时，就可以用该物体沿着坐标原点的偏移来进行描述。

在可视化中，物体一般都是作为一个模型来存储的，我们并不知道这个模型究竟放在什么地方，但我们知道对于一个不发生形变的模型，比如静止的人，他的头永远在肩膀上面，手永远在胳膊下端。因此，模型都有一个局部坐标系，来描述模型上的各个组件的局部位置。

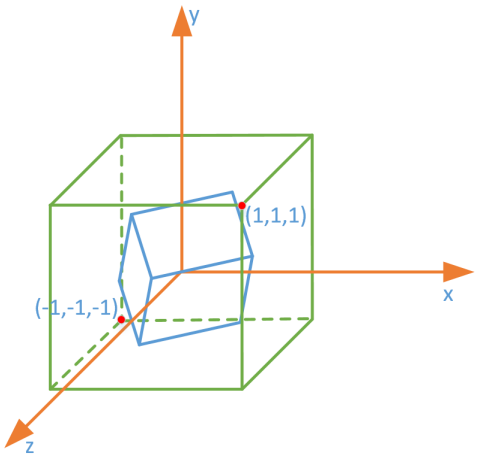
之后，模型在经过缩放、旋转和平移来移动到世界坐标系的位置。

要将物体成像，我们就需要一个观测点以及一个相机，然后将物体变换到相机视角下的坐标。

之后再将相机坐标进行一些后阶段的归一化之类的计算就可以了，具体图示如下。



其中，规范坐标系是三维坐标在 $(-1,-1,-1)$ 和 $(1,1,1)$ 之间的立方体：



1.2 模型空间到世界空间

通过缩放旋转平移矩阵将物体转移到世界坐标系下：

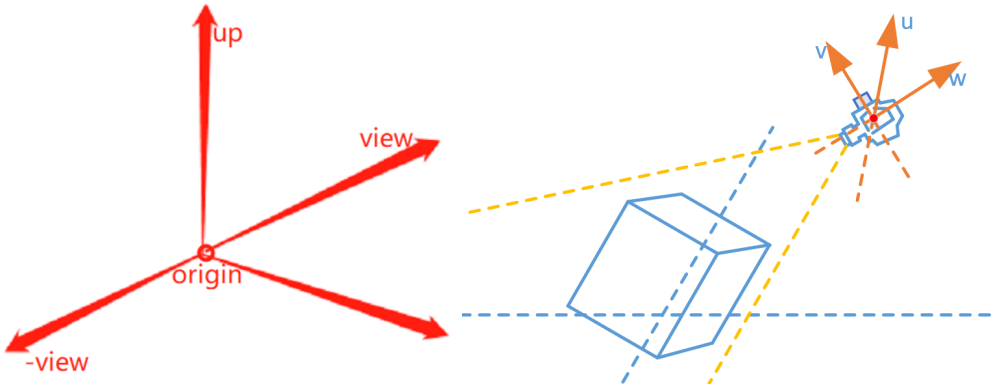
$$P_w = (M_T M_R M_S) P_o \tag{1.2.1}$$

$$= M_w P_o \tag{1.2.2}$$

其中 M_S 表示缩放矩阵变换，M_R 表示旋转变换，M_T 表示平移变换。我们将这三个变换的作用定义为 M_w，即从模型坐标系到世界坐标系的变换。

1.3 世界空间到相机空间

相机空间中，相机位置在原点，看向 $-z$ 轴，定义为 origin；以及相机看向的方向，定义为 $\widehat{view}$（默认为单位向量）；同时一个指向天空的方向，表示为 $\widehat{up}$（默认为单位向量）。



但相机也是在世界坐标空间里的，因此上述相机的向量也都会用世界坐标系来描述。我们可以定义出相机坐标空间的坐标基：

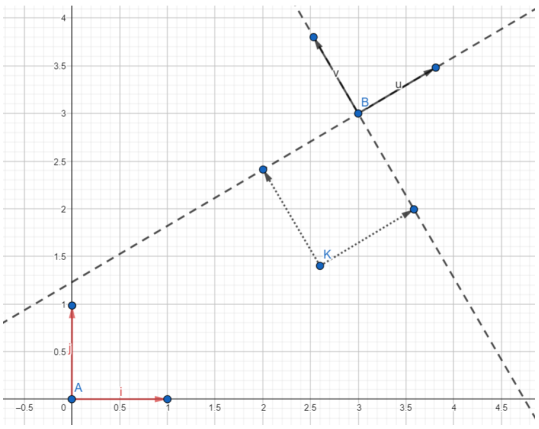
$$\widehat{w} = -\widehat{view} \tag{1.3.1}$$

$$\widehat{u} = cross(\widehat{up}, \widehat{w}) \tag{1.3.2}$$

$$\widehat{v} = cross(\widehat{w}, \widehat{u}) \tag{1.3.3}$$

得到的坐标基向量如上图右。

现在我们需要了解如何从一个坐标系转移到另外一个坐标系。我们以二维坐标系为示意图辅助理解，但实际还是用三维来进行介绍：



我们不需要考虑基变换的原理（可见 DezemingFamily 的《矩阵基变换》）。我们只需要如下考虑，我们假设在世界坐标系 A 和相机坐标系 B 下表示同一个点分别表示为：

$$P(a_1, a_2, a_3) = (0,0,0) + a_1\widehat{x} + a_2\widehat{y} + a_3\widehat{z} \tag{1.3.4}$$

$$P(b_1, b_2, b_3) = (0,0,0) + b_1\widehat{u} + b_2\widehat{v} + b_3\widehat{w} \tag{1.3.5}$$

用世界坐标系表示相机坐标系的点，就可以表示为：

$$P(a_1, a_2, a_3) = \widehat{b} + b_1\widehat{u} + b_2\widehat{v} + b_3\widehat{w} \tag{1.3.6}$$

其中 $\widehat{b} = (b_x, b_y, b_z)$ 是坐标系相对于世界坐标原点 $(0,0,0)$ 的偏移。

因此，$P(a_1,a_2,a_3)$ 转换为齐次坐标形式，可以用矩阵变换来表示：

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & b_x \\ 0 & 1 & 0 & b_y \\ 0 & 0 & 1 & b_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} \tag{1.3.7}$$

$$= \begin{bmatrix} 1 & \widehat{b} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \widehat{u} & \widehat{v} & \widehat{w} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} \tag{1.3.8}$$

而如果我们有了世界坐标系下的点 $P(a_1,a_2,a_3)$，我们通过逆变换就能得到相机坐标系下的点：

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -b_x \\ 0 & 1 & 0 & -b_y \\ 0 & 0 & 1 & -b_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} \tag{1.3.9}$$

$$= \begin{bmatrix} \widehat{u} & 0 \\ \widehat{v} & 0 \\ \widehat{w} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -\widehat{b} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} \tag{1.3.10}$$

需要注意的是，下面的两个基变换矩阵是正交矩阵（作用等同于旋转矩阵），即矩阵的转置等于矩阵的逆：

$$\begin{bmatrix} \widehat{u} & 0 \\ \widehat{v} & 0 \\ \widehat{w} & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} \widehat{u} & \widehat{v} & \widehat{w} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.3.11}$$

因此，当前从模型坐标系变换到相机坐标系的步骤我们定义为：

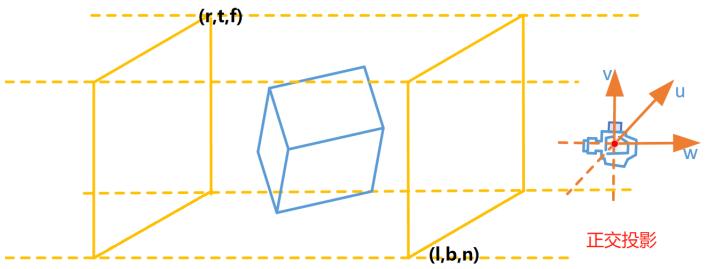
$$P_c = M_c M_w P_o \tag{1.3.12}$$

1.4 正交投影与透视投影

投影变换就是将视线可见区域压缩到规范视体中（$(-1,-1,-1)$ 和 $(1,1,1)$ 之间）我们定义的相机坐标系三轴是 $(\widehat{u},\widehat{v},\widehat{w})$，坐标原点即相机位置，为 $(0,0,0)$。

正交投影

对于正交投影来说：



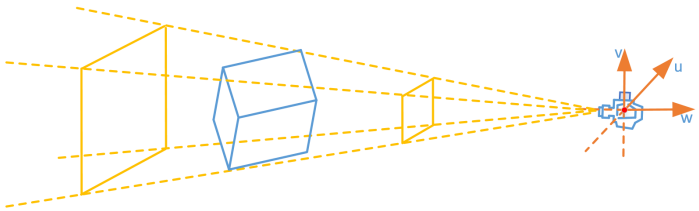
设立方体可见区中顶点坐标在相机坐标系中 $\widehat{u},\widehat{v},\widehat{w}$ 轴中最靠近相机的 w 值为 n，最远离相机的 w 的值为 f，立方体坐标表示为 (l,b,n)（near,bottom,near）和 (r,t,f)（right,top,far），可以得到正交变换矩阵：

$$M_{orth}=\begin{bmatrix}\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l}\\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b}\\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f}\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{1.4.1}$$

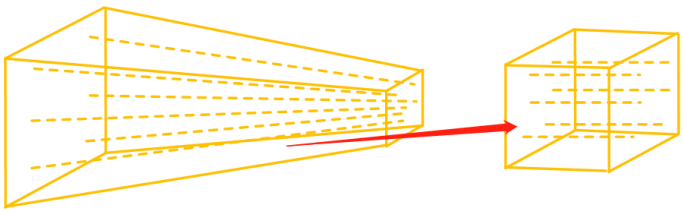
该变换将可见区的物体的三个坐标都压缩到坐标 $[-1,1]$ 内。

透视投影

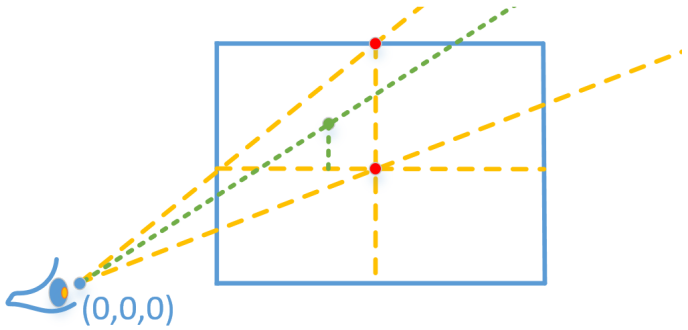
透视投影符合近大远小的特征。



将视锥体可见区变换到正规体下，直觉上看，这就是一个压缩的过程：



压缩可以分为两个过程，第一个过程是先“上下压缩”，把视锥体压缩到立方体，把视锥体粗的部分压缩为和细的部分一样粗；第二个过程是把立方体压缩为正规视体。我们先研究第一个过程，以下图中绿线上的点映射到屏幕上的 v 值为例：



其中上图显示的绿点在近平面上，w 值为 n。绿线上假设某点的 $(\widehat{u},\widehat{v},\widehat{w})$ 值为 (a,b,c)，则该点在屏幕上的 v 值 v_0 为：

$$\frac{v_0}{n}=\frac{a}{c} \tag{1.4.2}$$

$$v_0=\frac{an}{c} \tag{1.4.3}$$

同时，变换以后的近平面的 $\widehat{w}$ 值要变为-1，远平面的 $\widehat{w}$ 要变为 1。因此我们得到透视投影矩阵：

$$M_{p1}=\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.4.4}$$

至此，投影矩阵就构造出来了。我们验证一下：当该矩阵乘以某点的齐次坐标形式 $[a_0,a_1,n,1]^T$ 后，得到 $[na_0,na_1,n^2,n]$，转换为以 1 为底的坐标后为：$[a_0,a_1,n,1]^T$，近平面上的点变换到正规视体下的。

当然，“压缩”成立方体以后还得压缩到正规视体上，因此该过程和正交投影是一致的，我们得到完整的透视投影矩阵：

$$M_{pers}=\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.4.5}$$

$$=\begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & -\frac{2fn}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.4.6}$$

注意 l 和 r 分别指近平面的最左端和最右端坐标；t 和 b 也是分别指近平面的最上端和最下端坐标值。因为第一阶段压缩以后，立方体的 uv 截面是近平面的大小。

因为 l 一般是-r，b 一般是-t，因此其实上面的投影矩阵中，$M_{pers}[0][2]$ 和 $M_{pers}[1][2]$ 计算出来都是 0。

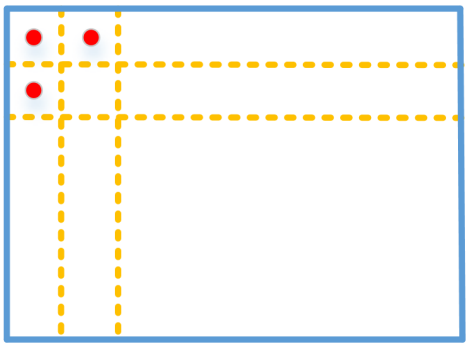
1.5 正规视体到屏幕空间

我们知道屏幕空间是有分辨率的，比如我们的屏幕是 400×200，但是屏幕空间是连续的，范围是：

$$width: \quad 0 \sim 400 \tag{1.5.1}$$

$$height: \quad 0 \sim 200 \tag{1.5.2}$$

例如正规视体中 x 坐标为 0.4（注意总范围是-1 到 1），映射到屏幕的坐标就是 140.0。注意这个没有特别严格的规定，屏幕显示的时候，像素位置一般是屏幕空间中两个整数点的中间位置：



由此我们得到的屏幕变换矩阵为：

$$M_{vp} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.5.3}$$

整体流程

整个变换管线的流程表示如下：

- 先对物体做世界坐标变换，即缩放旋转平移。
- 然后对物体做相机坐标变换，变换到相机坐标。
- 然后对物体做透视投影变换或正交投影变换，变换到正规视体。
- 然后做屏幕变换，变换到屏幕位置。

现在视角变换都已经介绍完了，我们下一章就介绍 glm[2][3] 和 PBRT[1] 中的源码实现。

2. 投影变换的 PBRT 源码实现

2.1 PBRT 中的透视投影实现 13
2.2 GLM 的矩阵投影变换 17

本章以 *PBRT* 中的投影矩阵生成和 *glm* 库的投影矩阵生成。因为正交投影过于简单，我们不再做介绍，而是重点介绍透视投影矩阵的实现。*PBRT* 的实现非常与众不同，本来我是不想讲它的，但是考虑到透视变换本身就有很多种变体，我们重点是学习变换的思路和矩阵的构造过程，而不仅仅是变换本身，因此 *PBRT* 也是一个很好的例子。

2.1 PBRT 中的透视投影实现

首先强调一点，对 PBRT 流程不太熟悉的人请不要轻易看本章节，可以看 glm 的源码实现，因为 PBRT 的流程是专门为光线追踪引擎设计的，有很多地方跟光栅引擎的实现都不太一样，特此强调一下。我一开始研究 PBRT 的相机空间变换代码时遇到了很多想不明白的地方，最终也是不断调试才渐渐学会的，因为这里面的操作和光栅器的操作差别很大!!!

为了和 PBRT[1] 一致从而方便描述，PBRT 中的相机坐标系的坐标轴用 x,y,z 值来描述。

整体流程如下：首先从图像光栅空间变换到 screen，screen 变换到相机空间，再把相机坐标系变换到世界坐标系（注意 PBRT 的实现并不和传统光栅器完全一样，而是从世界空间的相机发出光线去采样的）。

screen 上的范围并不是正规视体的范围，短边会被映射到 [-1,1]，长边则会超过这个范围。长宽比保持光栅成像的长宽比。同理，将相机空间映射到 screen 的时候也是要保证 screen 的长宽比（因此可以说 PBRT 的 screen 空间不是标准的 NDC 空间，同时 PBRT 也不需要进行空间裁剪）。

LookAt 矩阵的生成

首先是实现世界-相机空间变换的矩阵的源码（见 PBRT 源码 transform.cpp）：

```
1 Transform LookAt(const Point3f &pos, const Point3f &look, const
    Vector3f &up) {
2   Matrix4x4 cameraToWorld;
```

```
    // Initialize fourth column of viewing matrix
    cameraToWorld.m[0][3] = pos.x;
    cameraToWorld.m[1][3] = pos.y;
    cameraToWorld.m[2][3] = pos.z;
    cameraToWorld.m[3][3] = 1;
    // Initialize first three columns of viewing matrix
    Vector3f dir = Normalize(look - pos);
    Vector3f right = Normalize(Cross(Normalize(up), dir));
    Vector3f newUp = Cross(dir, right);
    cameraToWorld.m[0][0] = right.x;
    cameraToWorld.m[1][0] = right.y;
    cameraToWorld.m[2][0] = right.z;
    cameraToWorld.m[3][0] = 0.;
    cameraToWorld.m[0][1] = newUp.x;
    cameraToWorld.m[1][1] = newUp.y;
    cameraToWorld.m[2][1] = newUp.z;
    cameraToWorld.m[3][1] = 0.;
    cameraToWorld.m[0][2] = dir.x;
    cameraToWorld.m[1][2] = dir.y;
    cameraToWorld.m[2][2] = dir.z;
    cameraToWorld.m[3][2] = 0.;
    return Transform(Inverse(cameraToWorld), cameraToWorld);
}
```

该矩阵的实现与前面的内容完全一样，就是把新的基坐标变换实现到代码里。我们不再赘述。

透视投影矩阵的生成

然后是产生透视投影矩阵的源码（见 PBRT 源码 transform.cpp），该源码将相机空间的点变换到 screen 空间，该空间的短边范围是 [−1,1]，长边范围与短边范围成比例：

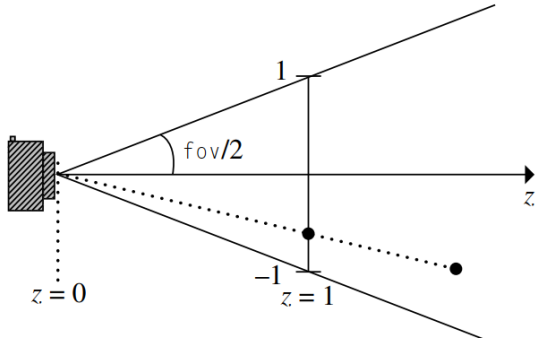
```
Transform Perspective(Float fov, Float n, Float f) {
  // Perform projective divide for perspective projection
  Matrix4x4 persp(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, f / (f - n), -f * n /
      (f - n),0, 0, 1, 0);
  // Scale canonical perspective view to specified field of view
  Float invTanAng = 1 / std::tan(Radians(fov) / 2);
  return Scale(invTanAng, invTanAng, 1) * Transform(persp);
}
```

我们需要注意的是，在 PBRT[1] 中的矩阵变换中利用了视场角 fov（field-of-view），近平面

映射到 $z=0$，远平面映射到 $z=1$。因此首先先生成了一个矩阵：

$$M_{persp}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&\frac{f}{f-n}&-\frac{fn}{f-n}\\0&0&1&0\end{bmatrix} \tag{2.1.1}$$

我们需要依赖视场角，得到可见区域的大小范围（注意映射到相机空间时，短边会被映射到 $[-near\cdot\tan(\frac{fov}{2}),near\cdot\tan(\frac{fov}{2})]$ 范围，而一般的光栅器则会将 screen 的宽（当宽的长度小于横边时与 PBRT 一样），下图表示在相机空间映射到这个范围）：



我们暂时先只考虑 screen 矩形的长大于宽的情况（横边长度大于宽边长度）：tan 运算符合 (x,y) 坐标增长的线性关系，即我们要构造一个运算，当处于视角上边缘时，能够映射到 1，下边缘也是同理。为了保证映射不畸变，需要令坐标映射呈现线性关系。本节一开始说过，要保证 screen 的长宽比，因此横坐标也是除以 tan（乘以它的倒数）。比如近平面的一个点 $(x,y,n,1)$（还是宽边小于横边长度）变换以后就成了 $(x,y,0,n)$，其实就是 $(x/n,y/n,0,1)$，因此再除以 tan 后，宽的范围就变到了 $[-1,1]$ 之间。

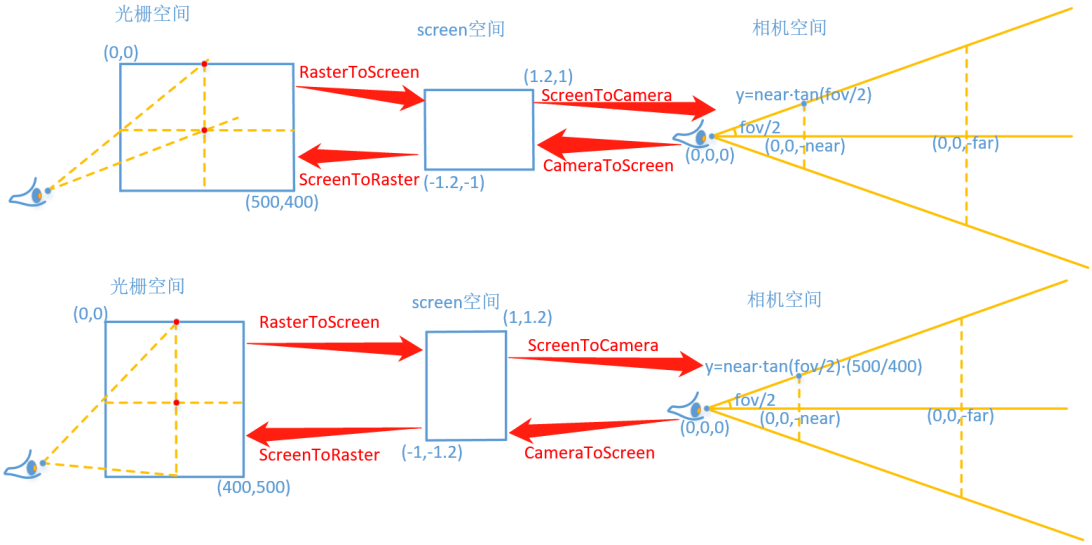
这样讲起来可能过于抽象，下一节我整体进行描述，然后从代码和实践操作的角度来更好的讲解。

整体流程描述

在 PBRT 中，从光栅空间变换到相机空间的变换矩阵叫做 RasterToCamera，它的作用就是将光栅空间的点变换到相机空间，其实这是两个变换的组合：ScreenToCamera*RasterToScreen（注意矩阵变换是从右到左的），即先从光栅空间变换到屏幕空间，再从屏幕空间变换到相机空间。在光栅空间的点要先变换到屏幕上，然后注意这个屏幕是近平面，因此齐次坐标 z 值为 0，设为 $(x',y',0,1)$，再经过 ScreenToCamera 就变换到了实际的相机位置。

ScreenToCamera 矩阵其实就是 CameraToScreen 的逆变换得到的，而 CameraToScreen 其实就是 Perspective() 函数生成的。

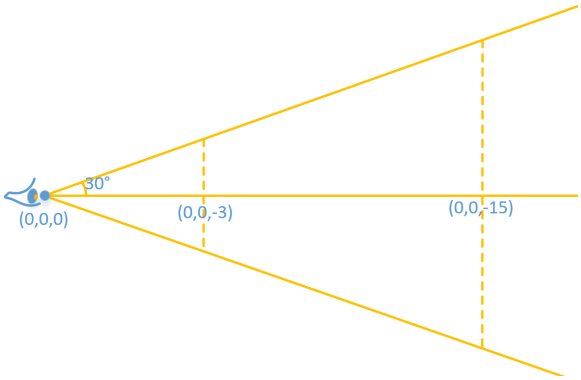
整体示意图表示如下，注意宽边是长边和宽边是短边变换出来的结果对于相机空间中的宽边长度是完全不一样的，也就是说当宽边大于横边长度时，fov 其实在变换到相机坐标系后就不再是我们设置的 fov 了：



我们接下来用程序来实际测试一下。

程序测试

假如我们的屏幕是 500×400 的，相机近平面距离原点为 3，远平面距离原点为 15，fov 为 60 度。



我们将 PerspectiveCamera::GenerateRay 函数里的下面这行代码屏蔽掉，因为我们不需要观察它变换到世界空间下的值，我们只想看从光栅空间变换到相机空间的对应关系；另外方向向量不要单位化，因为我们需要不单位化时，ray.d 向量的值其实就在相机空间里近平面上的值：

```
1 *ray = CameraToWorld(*ray);
2 //去掉下面代码的单位化
3 *ray = Ray(Point3f(0, 0, 0), Normalize(Vector3f(pCamera)));
```

pFilm 的值就是光栅上的点的位置，注意这里的位置不一定是整数，我们改变 pFilm 的值，然后得到结果：

```
1 CameraSample cameraSample;
2 cameraSample.pFilm = Point2f(0.f, 0.f);
3 Ray ray;
```

```
4 camera->GenerateRay(cameraSample, &ray);
```

我们得到的结果表示如下（注意光栅图像左下角在 screen 是左上角）：

$$pFilm \qquad ray.d \tag{2.1.2}$$

$$(0.0, 0.0) \qquad (-2.16506, 1.73205, 3) \tag{2.1.3}$$

$$(500, 400) \qquad (2.16506, -1.73205, 3) \tag{2.1.4}$$

可以看到 z 值变换到了 3，y 值正是 $\sqrt{3}$，而 z 值为 $\frac{5\sqrt{3}}{4}$，符合我们计算的值。
但当我们的屏幕是 400×500 的时候，情况就不一样了：

$$pFilm \qquad ray.d \tag{2.1.5}$$

$$(0.0, 0.0) \qquad (-1.73205, 2.16506, 3) \tag{2.1.6}$$

$$(400, 500) \qquad (1.73205, -2.16506, 3) \tag{2.1.7}$$

当屏幕是 100×500 的时候又变得不一样了：

$$pFilm \qquad ray.d \tag{2.1.8}$$

$$(0.0, 0.0) \qquad (-1.73205, 8.66026, 3) \tag{2.1.9}$$

$$(100, 500) \qquad (1.73205, -8.66026, 3) \tag{2.1.10}$$

按理来说，在光栅化引擎中（例如 OpenGL），无论屏幕分辨率怎么变，近平面的宽度值只会由近平面距离和 fov 决定，但是这里我们发现近平面大小随着宽高的分辨率比值的变化，其中短边变换到 screen 空间是值为 1。因此进行 tan 关系缩放后，短边在相机空间的值就为 $2 \cdot near \cdot \tan$，但是长边在相机坐标系的长度就是 $2 \cdot near \cdot \tan \frac{l}{s}$，其中 l 表示长边的长度，s 表示短边的长度，这正好印证了之前我们的代码：

```
1 //长边和短边都是使用的fov/2的tan值进行缩放
2 Scale(invTanAng, invTanAng, 1) * Transform(persp);
```

当短边是竖直边时，这样的结果是正好的（与光栅器变换的结果一样），但是当短边是横边时，这样的结果就跟光栅器的变换不一样了。

2.2 GLM 的矩阵投影变换

如果各位在 PBRT 的介绍中已经晕了，没有关系，我们可以看 glm 的实现过程。

视角变换

我们看到 glm 文件夹里的路径：*glm/glm/ext* 的 matrix_transform.inl 文件，lookAtLH 和 lookAtRH 分别是建立左手和右手坐标系的 LookAt 矩阵，默认是左手坐标系。

```
1 template<typename T, qualifier Q>
```

```
GLM_FUNC_QUALIFIER mat<4, 4, T, Q> lookAtLH(vec<3, T, Q> const& eye,
    vec<3, T, Q> const& center, vec<3, T, Q> const& up)
{
  vec<3, T, Q> const f(normalize(center - eye));
  vec<3, T, Q> const s(normalize(cross(up, f)));
  vec<3, T, Q> const u(cross(f, s));
  mat<4, 4, T, Q> Result(1);
  Result[0][0] = s.x;
  Result[1][0] = s.y;
  Result[2][0] = s.z;
  Result[0][1] = u.x;
  Result[1][1] = u.y;
  Result[2][1] = u.z;
  Result[0][2] = f.x;
  Result[1][2] = f.y;
  Result[2][2] = f.z;
  Result[3][0] = -dot(s, eye);
  Result[3][1] = -dot(u, eye);
  Result[3][2] = -dot(f, eye);
  return Result;
}
```

但当我们将第一章描述的 LookAt 矩阵中，我们发现上面返回的矩阵是我们计算的 LookAt 矩阵的转置，这是为什么呢？

这是因为 glm 定义的矩阵默认是列主矩阵，因此矩阵二维数组中，Mat[a][b]，a 表示第 a 列，b 表示第 b 行，我们在使用的时候一定要明确注意这一点，否则就有可能出现意想不到的错误！

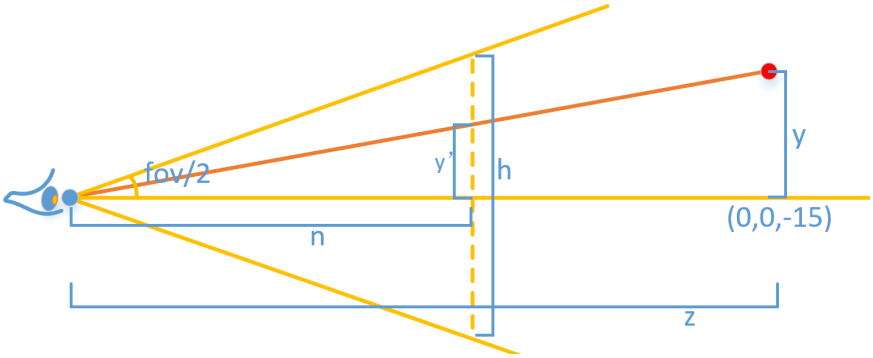
透视投影

透视投影的构建，该函数在 matrix_transform.ini 文件中：

```
template <typename T>
GLM_FUNC_QUALIFIER tmat4x4<T, defaultp> perspective(T fovy,T aspect,T
    zNear,T zFar)
{
  assert(abs(aspect - std::numeric_limits<T>::epsilon()) > static_cast
      <T>(0));
  T const tanHalfFovy = tan(fovy / static_cast<T>(2));
  tmat4x4<T, defaultp> Result(static_cast<T>(0));
  Result[0][0] = static_cast<T>(1) / (aspect * tanHalfFovy);
  Result[1][1] = static_cast<T>(1) / (tanHalfFovy);
```

```
9     Result[2][2] = -(zFar + zNear) / (zFar - zNear);
10    Result[2][3] = -static_cast<T>(1);
11    Result[3][2] = -(static_cast<T>(2) * zFar * zNear) / (zFar - zNear)
          ;
12    return Result;
13  }
```

这里的模板 T 一般是 float 或者 double 型的数据。我们可以看到 Result[0][0] 的值里有一个系数 aspect，该系数的值其实是屏幕分辨率宽除以高得到的：(float)width/(float)height。



这样变换到 screen 空间时，screen 的高就仅仅与近平面到相机原点的距离和 fov 角度有关了，且分辨率 width*height 内的像素点全都会变换到 $[-1,1]$ 坐标范围里。因此该变换就是将相机坐标下的点变换到正规视体内。

$$aspect = \frac{Width}{Height} \tag{2.2.1}$$

$$Height = 2n \times \tan(fov/2) \tag{2.2.2}$$

$$Width = 2n \times aspect \times \tan(fov/2) \tag{2.2.3}$$

$$\frac{y'}{y} = \frac{x'}{x} = \frac{n}{z}, \quad x' \in [-\frac{Width}{2}, \frac{Width}{2}] \quad y' \in [-\frac{Height}{2}, \frac{Height}{2}] \tag{2.2.4}$$

$$x' = xn/z \qquad y' = yn/z \tag{2.2.5}$$

$$\tag{2.2.6}$$

注意这里的 n 和 f 一般是我们设置的近平面和远平面距离，一般都设置的是正数，但 z 不一定是正数，比如当相机空间里，相机朝向相机坐标系的负轴，则可见区的物体 z 值就都是负数了，这个时候就要注意取反。但我们暂时先假设 z 是正数。

要令 x' 和 y' 收缩到 $[-1,1]$ 之间，需要分别除以 $Width/2$ 和 $Height/2$ 才行，因此就得到：

$$x' = \frac{x}{z \times aspect \times \tan(fov/2)} \tag{2.2.7}$$

$$y' = \frac{y}{z \times \tan(fov/2)} \tag{2.2.8}$$

我们设变换后的坐标为 $[x', y', z']$，其中这三个分量满足在 $[-1, 1]$ 之间，即：

$$\begin{bmatrix} \frac{1}{aspect \times \tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} & 0 & 0 \\ 0 & 0 & A_{2,2} & A_{2,3} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.2.9}$$

至于上面的矩阵中的 1 是正还是负，并没有严格的限制（上面说过，看变换后可见区在相机坐标系的 z 轴正半轴还是负半轴，这个是可以人为定义的，OpenGL 的 glm 库中是按照相机看向相机坐标系的负轴，所以说为-1）。$A_{2,2}$ 和 $A_{2,3}$ 的计算需要联立，把 $z = n$ 和 $z = f$ 分别代入上式去求解。我们设近平面压缩到-1. 远平面压缩到 1：

$$nA_{2,2} + A_{2,3} = -1 \times n \tag{2.2.10}$$

$$fA_{2,2} + A_{2,3} = f \tag{2.2.11}$$

联立就能得到 $A_{2,2}$ 和 $A_{2,3}$ 了。因为 glm 默认创建列主矩阵，因此需要注意上面的赋值中的 Mat 数组索引。

Bibliography

[1] Pharr M, Jakob W, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2016.

[2] https://github.com/g-truc/glm

[3] https://glm.g-truc.net/0.9.8/index.html

[4] https://www.cnblogs.com/bluebean/p/5276111.html