# 第三章 数学基础

3D图形学中几乎每个方面、每种效果——移动、缩放、透视、纹理、光照、阴影等——都在很大程度上以数学方式实现。

## 3.1 3D坐标系统

3D空间通常用3个坐标轴X、Y和Z来表示。这3个轴可以以两种方式

来布置：左手或右手（它们是以轴的朝向来命名的，通过左手或右手大拇指与食指、中指成直角来进行构造）。如图3.1所示

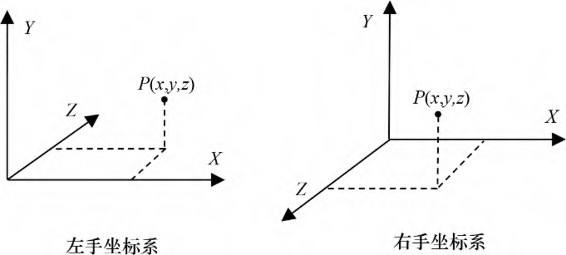


图3.1 3D坐标系统

OpenGL中的坐标系大体是右手坐标系，而Direct3D中大体是左手坐标系。在本书中，除非特别说明，否则我们都是用右手坐标系。

大拇指代表Z轴，食指代表X轴，中指代表Y轴。图3.1中，左手大拇指向外，手心向上；右手大拇指向内，手心向上。 ——译者注

## 3.2 点

3D空间中的点可以通过使用形如（2, 8, −3）的符号，列出X、Y、Z的值来表示。不过，如果用齐次坐标表示法来表示点会更有用。在每个点的齐次坐标有4个值。前3个值表示X、Y和Z，第四个值W总是非零值，通常为1。因此，我们会将之前的点表示为（2, 8, −3, 1）。齐次坐标将会使我们的图形学计算更高效。用来存储齐次3D坐标的GLSL数据类型是vec4（“vec”代表向量，同时也可以用来表示点）。GLM库包含适合在C++/OpenGL应用中创建和存储3元和4元（齐次）点的类，分别叫作vec3和vec4

## 3.3 矩阵

矩阵是矩形的值阵列，它的元素通常使用下标访问。第一个下标表示行号，第二个下标表示列号，下标从0开始。我们在3D图形计算中要用到的矩阵大多数大小为4×4，如图3.2所示。

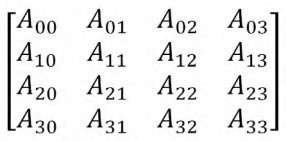
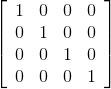


图3.2 4×4矩阵

GLSL语言中的mat4数据类型用来存储4×4矩阵。同样，GLM中有 mat4类用以实例化并存储4×4矩阵。

•单位矩阵：

单位矩阵中一条对角线的值为1，其余值全为0：

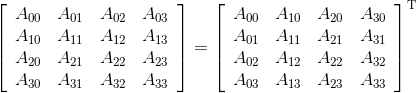


任何值乘以单位矩阵都不会改变。在GLM中，调用构造函数

glm::mat4 m(1.0f)以在变量m中生成单位矩阵。

### •矩阵转置：

矩阵转置的计算是通过交换矩阵的行和列完成的。例如：

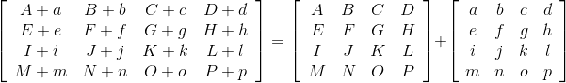


GLM库和GLSL库都有转置函数，分别是glm::transpose(mat4)和

transpose(mat4)。

•矩阵相加：

矩阵加法简单明了

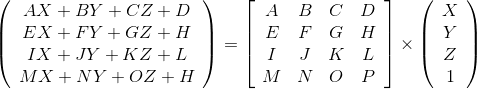


在GLSL中，+运算符在mat4上进行了重载，以支持矩阵加法。

•矩阵相乘(合并)：

3D图形学中有很多有用的矩阵乘法操作。矩阵乘法一般可以从左向右或从右向左处理（注意，由于左乘和右乘是不同的，所以矩阵乘法不满足交换律）。

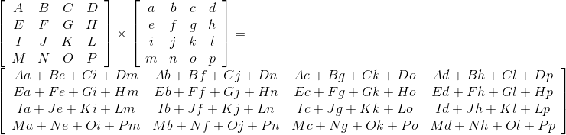
在3D图形学中，点与矩阵相乘通常从右向左，得到点，如：



注意，我们用齐次坐标将点(X, Y, Z)表示为列数为1的矩阵。

GLSL和GLM都支持点（确切地说是vec4）与矩阵使用\*操作符相乘。

4×4矩阵与4×4矩阵相乘如下：



矩阵相乘也经常叫作合并，稍后我们会看到，它可以用于将一系列矩阵变换合并成一个矩阵。这种合并矩阵变换的能力来自矩阵乘法 的结合律。

考虑如下运算序列：



我们将一个点与Matrix3相乘，之后将结果与Matrix2相乘，最后将 结果与Matrix1相乘。其结果是一个新的点。结合律确保了之前的计算 与如下计算相同：



我们先将3个矩阵相乘，建立Matrix1、Matrix2、Matrix3的连接。 如果我们称其为MatrixC，我们就可以将之前的运算写作：



我们稍后在第4章会看到这么做的好处是，我们需要经常将相同的一系列矩阵变换应用到场景中的每个点上。通过预先一次计算好这些矩阵的合并，就可以成倍减少总的矩阵运算量。

GLSL和GLM都支持使用重载后的\*运算符进行矩阵乘法。

•逆矩阵：

一个4×4矩阵的逆矩阵是另一个4×4矩阵，用表示，在矩阵乘法中有如下性质：



在此我们就不展示计算逆矩阵的细节了。但是，需要知道的是计算矩阵的逆矩阵的运算量很大。幸运的是我们只有很少情况下需要用到它。在这些极少的情况下，GLSL和GLM都提供了mat4.inverse()函数。

## 3.4 变换矩阵

在图形学中，矩阵通常用来进行物体的变换。例如矩阵可以用来将点从一处移动到另一处。在本章中，我们将会学习5个有用的变换矩 阵：

平移矩阵； 缩放矩阵； 旋转矩阵； 投影矩阵； LookAt矩阵。

变换矩阵的重要特性之一就是它们都是4×4矩阵。这是因为我们决定使用齐次坐标系。否则，各变换矩阵可能会有不同的维度并且无法相乘。

### 3.4.1 平移矩阵

平移矩阵用于将物体从一个位置移至另一位置。它包含一个单位矩阵，同时

X、Y和Z的移动量在 。图3.3展示了平移矩阵和它与齐次坐标点相乘的效果。其结果是一个以平移值“移动过”的点。

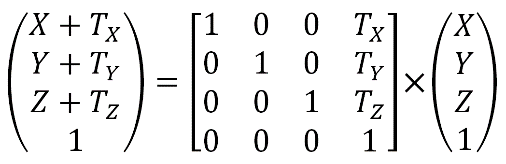


图3.3 平移矩阵变换

注意，作为与平移矩阵相乘的结果，点(X,Y,Z)平移(或移动)到位置(X+, Y+,, Z+,)。同样需要注意的是这个乘法是从右向左相乘。

GLM中有一些函数是用于构建与点相乘的平移矩阵的。其中相关的操作有：

• glm::translate(x, y, z)构建平移(X,Y,Z)的矩阵

• mat4 \* vec4。

### 3.4.2 缩放矩阵

缩放矩阵用于改变物体的大小或者将点向原点相反方向移动。虽然缩放点这个操作乍一看有点奇怪，不过OpenGL中的物体都是用一组或多组的点定义的。因此，缩放物体涉及缩放它的点的集合。

缩放矩阵变换由单位矩阵和位于 的XYZ缩放因子组成。图3.4中展示了缩放矩阵的形式和当它与齐次坐标点相乘的效果：所得的结果是经过缩放值修改后的新点。

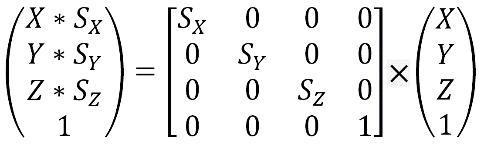


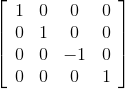
图3.4 缩放矩阵变换

GLM中有一些函数是用于构建与点相乘的缩放矩阵的。其中相关的操作有：

• glm::scale(x, y, z) 构建缩放(X,Y,Z)的矩阵；

• mat4 × vec4

缩放还可以用来切换坐标系。例如，我们可以用缩放来在给定右手坐标系的情况下确定左手坐标。从图3.1中我们可以看到通过反转Z坐标就可以在右手坐标系和左手坐标系中切换，因此，用来切换坐标系的缩放矩阵变换是：



### 3.4.3 旋转矩阵

旋转会稍微复杂一些，因为在3D空间中旋转物体需要指定旋转轴和旋转的角度或弧度。

在16世纪中叶，数学家莱昂哈德· 欧拉表明，围绕任何轴的旋转都可以表示为绕 X,Y,Z轴旋转的组合[EU76]。围绕这3个轴的旋转角度被称为欧拉角。这个被称为欧拉定理的发现，对我们很有用，因为对于每个坐标轴的旋转可以用矩阵变换来表示。

旋转变换有3种，分别是绕X Y和Z轴旋转，见图3.5。同时GLM中也有一些用于构建旋转矩阵的函数。

• glm::rotate(mat4, θ, x, y, z)构建绕X Y Z轴旋转度的缩放矩阵。

• mat4 × vec4。

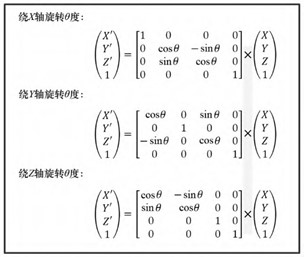


图3.5 旋转变换矩阵

实践中，当3D空间中旋转轴不穿过原点时，物体使用欧拉角进行旋转需要几个额外的步骤。一般有：

1. 平移旋转轴以使它经过原点；
2. 绕XYZ轴旋转适当的欧拉角；
3. 复原步骤（1）中的平移。

图3.5中所示的3个旋转变换都有自己有趣的特性，即反向旋转的矩阵恰等于其转置矩阵。后面将会用到这个特性。

欧拉角在某些3D图形应用中会导致一些瑕疵。因此，通常在计算旋转时推荐使用四元数。有兴趣探索四元数的读者可以寻求很多已有的资源（如[KU98]）。欧拉角足以满足我们的大部分需求。

## 3.5 向量

向量表示大小和方向。它们没有特定位置。“移动”向量并不改变它所代表的含义。

记录向量的方法各式各样，如：一端带箭头的线段、二元组（幅度+方向）或两点之差。在3D图形学中，向量一般用空间中的单个点表示，向量的大小是原点到该点的距离，方向则是原点到该点的方向。在图3.6中，向量V可以用点之间的差表示，也可以等价地用原点到的距离来表示。在我们的所有应用中，我们都简单地将V表示为(X,Y,Z)，即我们用来表示的符号。

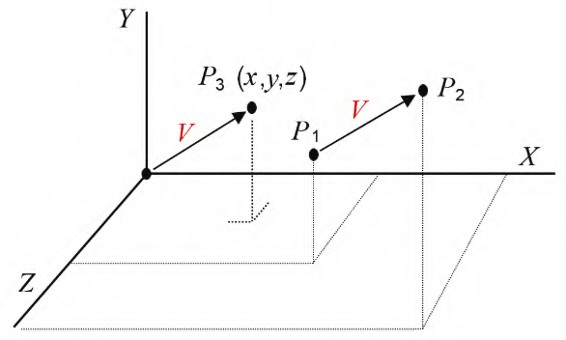


图3.6 向量V的两种表示

用与表示点相同的方式来表示向量很方便，因为我们对点和向量用同样的矩阵变换。不过这也会使人困惑。因此，我们有时候会在向量上加一个小箭头（如）。许多图形系统并不区分点和向量，如GLSL 和GLM，它们所提供的vec3/vec4类型既能用来存储点，又能用来存储向量。

在GLM和GLSL中有许多3D图形学中经常用到的向量操作。如假设有

向量A (u,v,w)和B(x,y,z)

### 加减法：

A±B=(u±x,v±y,w±z)

glm: vec3 ± vec3 GLSL: vec3 ± vec3

GLSL: vec3 ± vec3

### 归一化（变为长度 =1）：

glm: normalize(vec3) 或normalize(vec4)

GLSL: normalize(vec3) 或normalize(vec4)

### 点积：

A•B=ux+vy\_wz

glm: dot(vec3,vec3) 或dot(vec4,vec4)

GLSL: dot(vec3,vec3) 或dot(vec4,vec4)

### 叉积：

A◊B=(vz-wy,wx-uz,uy-vx)

glm: cross(vec3,vec3)

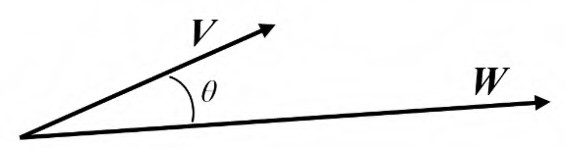
GLSL: cross(vec3,vec3)

其他有用的向量函数如magnitude（在GLSL和GLM中是 length()）、reflection和refraction（在GLSL和GLM中都有）。

我们现在仔细看一下点积和叉积函数。

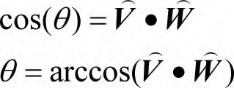
### 3.5.1 点积的应用

在本书中的程序大量使用了点积。点积最重要也最基本的应用是求解两向量夹角。设有向量V和W ，计算其夹角为θ





因此，如果V和 W是正规化向量（有着单位长度的向量——这里用“^”标记正规化），则有：



有趣的是，我们后面会看到通常用到的是cos(θ)，而非θ。因此，这两个推导出的公式都很有用。 点积同时还有许多其他用途。

求解向量的大小： 

求解两向量是否正交，若正交，则

求解两向量是否平行，若平行，则 

求解两向量是否平行但指向相反方向，若满足，则



求解两向量夹角是否在（−90°～+90°）： 

求解点P=(X,Y,Z)到平面S=(a,b,c,d)的最小有符号距离:

首先，求垂直于S的单位法向量：

以及从原点到平面的最短距离





之后从P到S的最小有符号距离为()+D，其符号由P在S的哪边决定

### 3.5.2 叉积的应用

两向量叉积的一个重要特性是，它会生成一个新的向量，新的向量正交（垂直）于之前两个向量所定义的平面。我们会在本书中大量使用到这一特性。

任意两个不共线向量都定义了一个平面。例如考虑两个任意向量

V和W。由于向量可以在不改变含义的情况下进行平移，因此，可以将它们移动到起点相交的位置。图3.7展示了V和W定义的平面，以及其叉积所得法向量。其所得法向量的方向遵循右手定则，即将右手手指从 V向W卷曲会使得大拇指指向法向量R

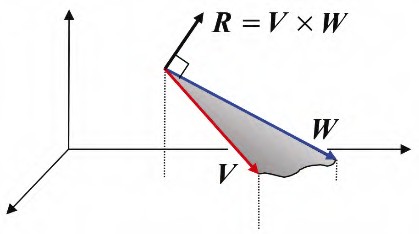


图3.7 叉积得到法向量

注意，这里顺序很重要。W×V将会得到与R方向相反的向量。

通过叉积来获得法向量的能力对我们后面要学习的光照部分非常重要。为了确定光照效果，我们需要知道所渲染模型的外向法向量。图3.8中展示了一个例子，其中有一个6个点（顶点）构成的简单模 型，使用叉积计算来获得其中一面的外向法向量。

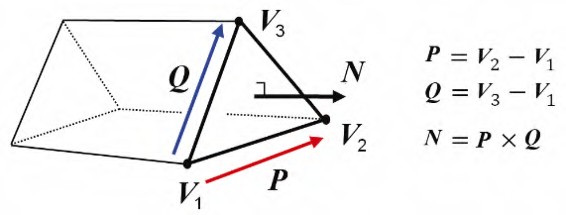


图3.8 计算外向法向量

## 3.6 局部和世界空间

3D图形学（用OpenGL或其他框架）常见的应用是模拟三维世界、 在其中放入物体并在显示器上观看它。放在3D世界的物体通常用三角形的集合来进行建模。稍后我们会在第6章中详细讲解建模。但是我们可以先了解一下大致的处理过程。

当建立物体的3D模型时，我们通常以最方便的定位方式描述模型。如果模型是个球形，那么我们很可能将球心定位于原点（0,0,0）并赋予它一个方便的半径，比如1。模型定义的空间叫作局部空间（local space）或模型空间（model space）。OpenGL文档使用的术语是物体空间（object space）。

之后这个球形可能用于一个大模型的部分，如成为机器人的头部。这个机器人，当然，定义在它自己的局部/模型空间。我们可以用图3.9所示的矩阵变换通过缩放、旋转和平移，将球形模型放在机器人模型的空间。通过这种方式，可以分层次地构建复杂模型（在4.8节中 会进一步通过使用一堆矩阵讲解这个主题）。

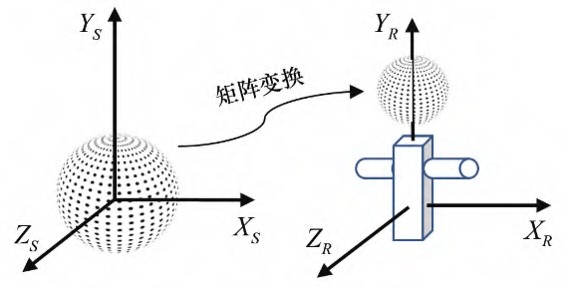


图3.9 球形和机器人的模型空间

使用同样的方式，通过设定物体在模拟世界中的朝向和大小，将 物体放在模拟这个世界的空间中，这个空间叫作世界空间。将对象定位及定向在世界空间的矩阵称为模型矩阵或M

## 3.7 视觉空间和合成相机

到此为止，我们所接触的变换矩阵全都在3D空间中操作。但是， 我们最终需要将3D空间——或它的一部分——展示在2D显示器上。为了达成这个目标，我们需要找到一个有利点。正如我们在现实世界通过眼睛从一点观察一样，我们也必须找到一点并确立观察方向作为我们观察虚拟世界的窗口。这个点叫作“视图”或“视觉”空间，或 “合成相机”。

如图3.10和图3.12所示，观察3D世界需要：（a）将相机放入世界 的某个位置；（b）调整相机的角度，通常需要一套它自己的直角坐标UVN（c）定义一个视体（view volume）；（d）将视体内的对象投影到投影平面（projection plane）上。

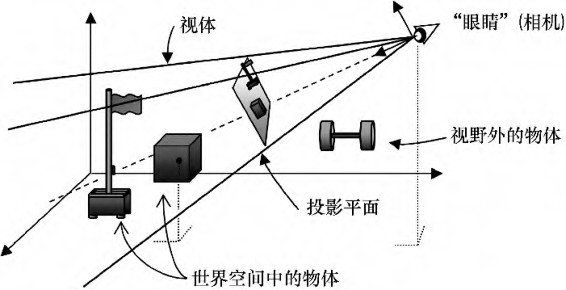


图3.10 将相机放入3D世界中

OpenGL有一个固定在原点(0,0,0)并向下看向Z轴负方向的相机，如图3.11所示

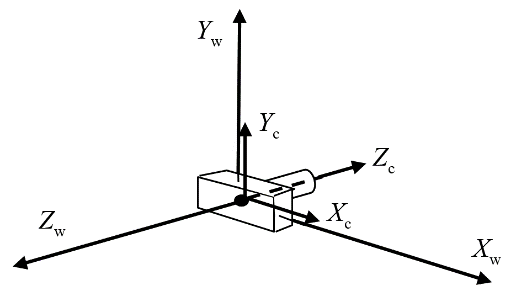


图3.11 OpenGL固定相机

为了应用OpenGL相机，我们需要做的是将它模拟移动到适合的位置和方向。我们需要先找出在世界中的物体与我们期望的相机位置的相对位置（如物体应该在由图3.12所示相机UVN轴定义的“相机空间”中的何处）。给定世界空间中的点，我们需要通过变换将它转换成相应相机空间中的点，从而让它看起来好像是我们从期望的相机位置进行观看的。我们通过计算它在相机空间中的位置实现。已知OpenGL相机位置永远固定在点(0,0,0)，问：我们如何变换来实现上述功能？

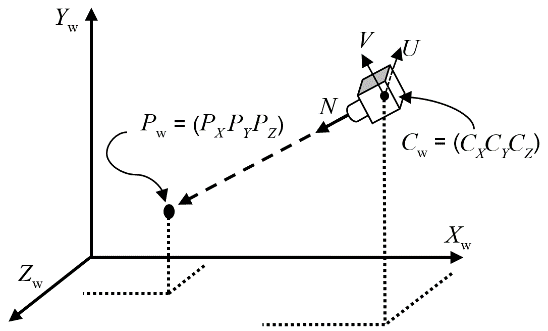


图3.12 相机方向

需要做的变换如下：

（1）将平移，其向量为负的期望相机位置

（2）将旋转，其角度为负的期望相机旋转的欧拉角。

我们可以构建一个单一变换矩阵以完成旋转和平移，这个矩阵叫作视图变换（viewing transform）矩阵V。矩阵V通过合并矩阵T（包含负相机期望位置的平移矩阵）和R（包含负相机期望旋转的旋转矩阵）。在本例中，从右向左，我们先平移世界空间中的点，之后旋转：



如前所见，通过结合律我们得到如下运算：



如果我们将合并后的R\*T存入矩阵V，运算成为



完整的计算以及T和R的准确内容在图3.13中（我们忽略了对矩阵R的推导——推导过程见参考资料[FV95]）。

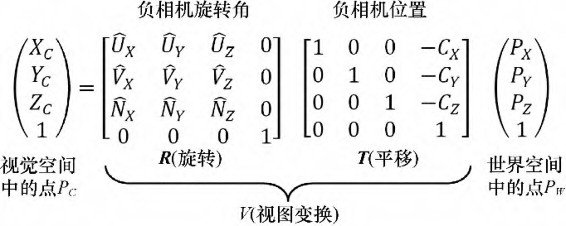


图3.13 推导视图矩阵

通常V矩阵与模型矩阵M合并成为一个模型-视图（model-view MV矩阵）



之后，点在从自己的模型空间通过如下一个步骤就可以直接转换至相机空间：



在复杂场景中，当我们需要对每个顶点，而非只是一个点做这个变换的时候，这种方法的好处就很明显了。通过预先计算MV，对于空间中每个点的变换只需要我们进行一次矩阵乘法计算。之后，我们将会看到，我们可以将这个过程延伸到与计算更多的合并矩阵，以大量减少每个顶点的计算量。

## 3.8 投影矩阵

当我们设置好相机之后，就可以学习投影矩阵了。我们需要学习的两个重要的投影矩阵是：透视投影和正射投影。

### 3.8.1 透视投影矩阵

透视投影通过使用透视概念模仿我们看真实世界的方式，尝试让2D图像看起来像是3D的。物体近大远小，3D空间中有的平行线用透视法画出来就不再平行。

透视法是15世纪初～16世纪初文艺复兴时期的伟大发现之一，当 时的画家开始绘制比前人更加真实的画作。



图3.14中是一个绝好的例子

图3.14 《圣母领报》（卡洛·克里韦利，1486年）

我们通过使用变换矩阵将平行线变为恰当的不平行线来实现这个效果。这个矩阵叫作透视矩阵或者透视变换，通过定义4个参数来进行视体（view volume）的构建。其中4个参数是纵横比、视场、投影平面或近剪裁平面、远剪裁平面。

只有在远近剪裁平面间的物体才会被渲染。近剪裁平面同时也是物体所投影到的平面，通常放在离眼睛或相机较近的位置（如图3.15 左侧所示）。我们会在第4章中讨论如何为远剪裁平面选择合适的值。 视场是可视空间的纵向角度。纵横比是远近剪裁平面的宽度比高度。 通过这些元素所形成的形状叫作视锥（frustum），如图3.15所示。

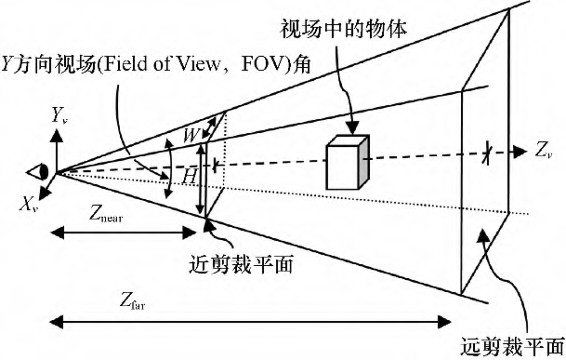


图3.15 透视视体或视锥

透视矩阵用于将3D空间中的点变换至近剪裁平面上合适的位置，它的构建需要先计算 q A B C的值，之后用这些值来构建透视矩阵，如图3.16所示（推导过程见参考资料[FV95]）。

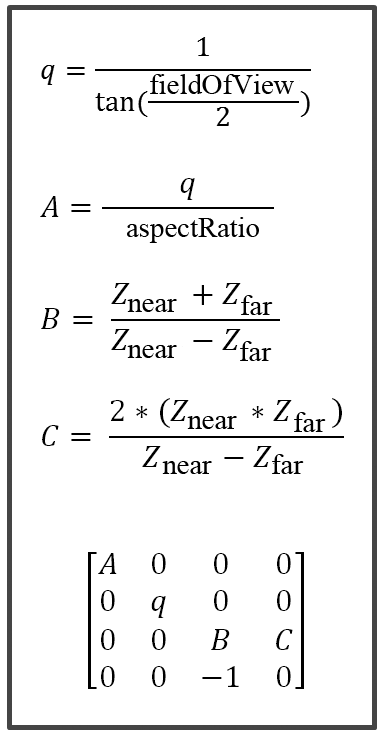


图3.16 构建透视矩阵

生成透视变换矩阵很容易，只需要将所描述的公式插入一个4×4 矩阵。GLM库也包含了一个用于构建透视矩阵的函数 glm::perspective()。

### 3.8.2 正射投影矩阵

在正射投影中，平行线仍然是平行的，即不使用透视，如图3.17 所示。正射与透视相反，在视体中的物体不因其距相机距离做任何调 整，而直接进行投影。

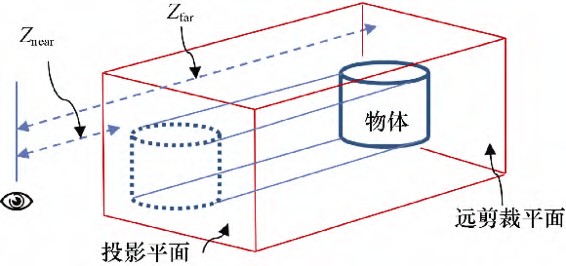


图3.17 正射投影

正射投影是一种平行投影，其中所有的投影都与投影平面垂直。正射矩阵通过如下参数构建：（a）从相机到投影平面的距离（b）从相机到远剪裁平面的距离（c）LRT和B的值，其中L和R分别是投影平面左右边界的X坐标.T和B分别是投影平面上下边界的Y坐标，如图3.18所示。

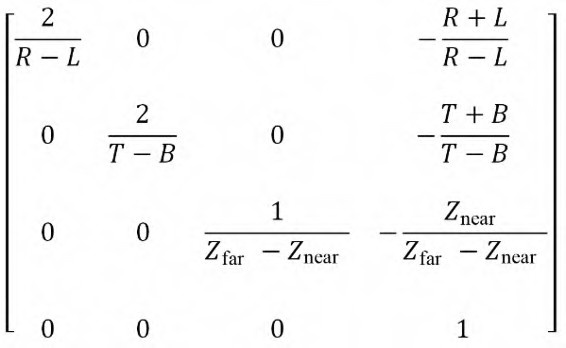


图3.18 正射投影矩阵

## 3.9 LookAt矩阵

我们最后要学习的变换是LookAt矩阵。当你想要把相机放在某处并看向一个特定的位置时，就需要用到它了，如图3.19所示。当然，用我们已经学到的方法也可以做到，但是这个操作非常频繁，因此为它专门构建一个矩阵通常比较有用。

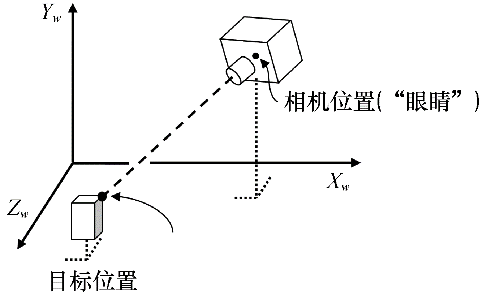


图3.19 LookAt的元素

LookAt变换依然由相机旋转决定。我们通过指定大致旋转朝向的向量（如世界Y轴）。通常，可以通过一系列叉积获得相机旋转的正面、侧面以及上面。图3.20展示了计算过程，从相机位置（眼睛）、目标位置以及初始向上向量Y来构建LookAt矩阵，其推导过程在参考资料[FV95]中。

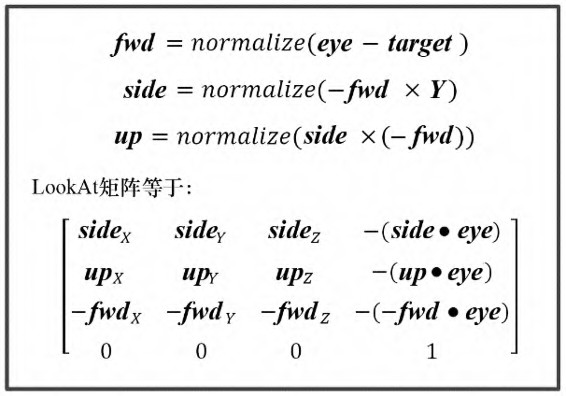


图3.20 LookAt矩阵

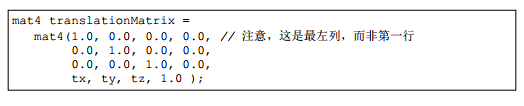
我们可以将这个过程构建为一个C++/OpenGL实用函数，通过指定相机位置、目标位置以及初始“向上”向量Y，构建一个LookAt矩阵。

由于GLM中已经有一个用来构建LookAt矩阵的函数glm::lookAt()，我 们用它就可以了。稍后本书第8章生成阴影的时候会用到这个函数。

## 3.10 用来构建矩阵变换的GLSL函数

虽然GLM包含了许多预定义的3D变换函数，本章也已经涵盖了其中如平移、旋转和缩放，但GLSL只包含了基础的矩阵运算，如加法、合 并等。因此，有时我们需要自己为GLSL写一些实用函数来构建3D变换矩阵，以在着色器中进行特定3D运算。用于存储这些矩阵的GLSL数据类型是mat4。

GLSL中用于初始化mat4矩阵的语法以列为单位读入值。前4个参数会放入第一列，接下来4个参数放入下一列，直到第四列，如下所示：



构建如图3.3所示的平移矩阵。

程序3.1中包含了5个用于构建4×4平移、旋转和缩放矩阵的GLSL 函数，每个函数对应于本章之前给出的一个公式。我们稍后在书中将会用到这些函数。

程序3.1 在GLSL中构建变换矩阵

