3D Math

学计算机图形学的过程，必然有无数的人，不断BB数学有多重要，有多难，我们需要掌握\*\*\*，我们还需要掌握\*\*\*，还有\*\*\*。然后，整个人可能就蒙圈了，崩溃了，大受打击，退出图形学届，这也是另外一种形式的劝退。

我从不说数学不重要。但是对于图形学而言，有没有可怕到那个地步，需要掌握大量的数学，否则寸步难行？

客观来说，如果非要掌握各种图形学算法的实现原理，从学术研究方向来推导、实现各种效果，那么需要的数学是不可限量的。说穿了，做研究，需要用到大量数学，但是做应用，我并不认为需要掌握那么多。有人可能会反驳：你不懂那么多，怎么看懂别人的论文？首先，看懂别人的论文，其实未必需要懂那么多，很多时候我们不需要推导，而只需要直接用结论。其次，绝大部分人，抄这类代码能666就过得去了，原创这类代码，只有极少数人可以达到。

所以，本着这一个原则，本章只打算讲最常用，最基础的3D数学：向量，四元数，矩阵。更基础的，例如三角函数，平面解释几何，角度/弧度……，这类是必须掌握的基础知识，如果都没有掌握，真的该放弃了，人生苦短，我用python！

开始之前，我先讲一些我认为很重要的数学概念。由于我出身于十八线农村，从小几乎都是填鸭式教育，对数学缺乏一个深刻的理解，所以很多很基础很重要的数学概念，我都是大学毕业之后才慢慢领悟的，如果你早早领悟到了，那么恭喜你，你可以不用看以下部分了。

我们来认真思考一下这些问题：

数学到底是发明还是发现？

这个问题，现在网上能看到大量的讨论。但是，在十几二十年前，互联网还不发达的年代，我觉得绝大部分像我这种出身的，都没能理解或者探讨过这类问题，我认为这是个很重要的问题。我的解释是：两者都是。首先，数学是宇宙的一些规律，我们属于发现了这些规律。然而，这还不够，我们还需要发明各种各样的数学工具，来处理这些规律。举例：3D数学里，有个很重要的概念就是旋转，而为了处理这个旋转，可以分别用欧拉角、四元数、矩阵来处理。而这三种，就是不同的数学工具，而处理的，是同一个宇宙规律。再如，处理平面的一些问题，可以用笛卡尔直角坐标系，但是也可以用极坐标系，这也是两种不同的数学工具，处理同样的数学问题。

第二个问题：“为什么？”

从小我们就被教导，要多问为什么，碰到什么事情，都要问为什么。而在数学的学习中，我同样一如既往。但是，后来我终于发现，答案往往是：“不为什么”，因为，这就是宇宙的规律。

这个就好比，为什么第一种永动机造不出来？很简单，因为能量守恒定律。那么，为什么有能量守恒定律？不为什么，这是宇宙规律。

早期，学数学的时候，我很喜欢问为什么，后来终于懂得，很多都没有为什么，公理，定理，推论，都没有为什么，这就是宇宙规律。公理就是公认的宇宙规律，就比如两点间线段最短，这就是宇宙规律。所以，在学习中，经常会碰到例如：矩阵乘法为什么要这样算？向量的叉乘为什么这样算？这个时候，不要问为什么，这就是宇宙规律，使用这些数学工具，这样子就能算出来。

第三点，高等数学，很多已经颠覆了你的认知，我们不能以普通的认知，来学习高等数学。例如，我们常常潜意识里，会对加减乘除，有一个定义。例如除法跟乘法是逆运算，加法，减法，就是增加了，减去了。然后还有一些深入骨髓的理解，例如加法交换律，乘法交换律，结合律等等。这些都是一些常识性的存在。

但是，在很多高等数学的世界里，例如矩阵，是不存在交换律的，甚至连除法都不存在！再比如矩阵的加法，我搞那么多年的3D数学，写过那么多的矩阵计算，但是矩阵的加法只存在于线性代数的练习题里。矩阵的转置，求逆，正交，乘法，都是非常常用和常见的性质，但是加法，减法，我就没用过，也不知道几何意义。因此，学习高等数学的时候，先要摒弃常识，不要以为除法就一定是乘法的逆运算，不要想当然的就必然有交换律。

以上是我个人的一些粗浅理解。我不是数学专业的，很多全凭自己的理解，错漏难免，有知道哪里有错漏的，欢迎直接留言喷我，让我长点知识。

向量

计算机图形学里面，有三个常用的向量：二维向量，三维向量，四维向量。一般的定义为：

Struct Vector2

{

Float x, y;

}

Struct Vector3

{

Float x,y,z;

}

Struct Vector4

{

Float x,y,z,w;

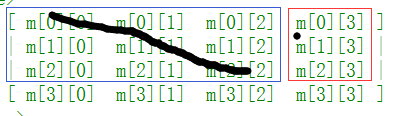
}

开始之前先讲个笑话，好像听说有人在知乎上说，u3d代码写得比UE好，因为U3d的vector3里面，用float v[3];这样来记录的，而UE4里面是float x,y,z;

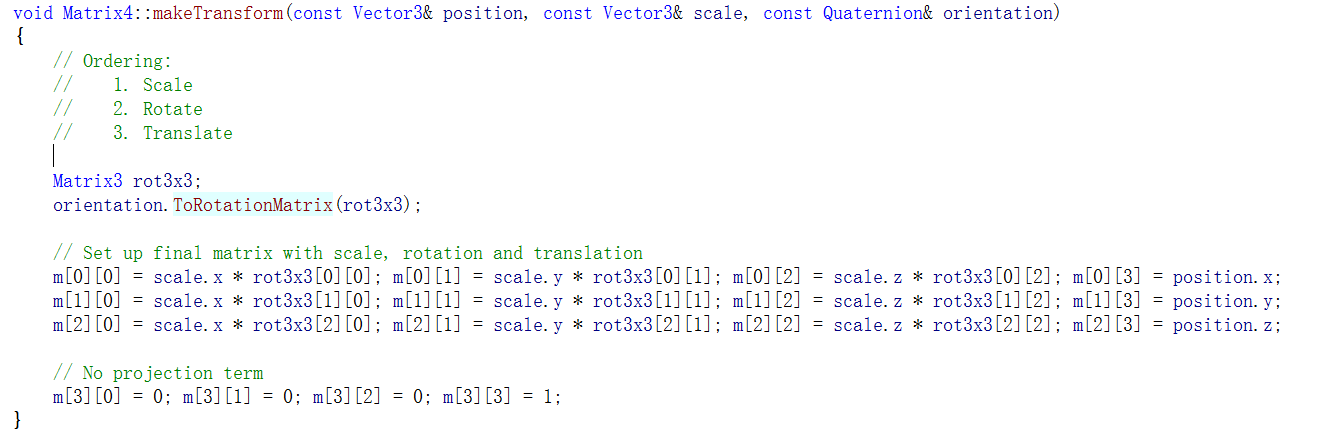
这个笑话不知真假，也无意去求证。如果是真的，只能说这哥们太天真了。不过我记得我的确在网上看到过列举种种喷UE4的FVector写得不好的。

言归正传，先简单介绍一下。Vector2顾名思义，主要是二维向量，用得比较少，主要用在涉及到屏幕坐标的计算。Verctor3，用得最多，主要用于绝大部分的3D计算，例如坐标，例如方向。绝大部分的接口，用到相关的，都是Vector3。那么，Vector4是干嘛的呢？做四维空间的计算的吗？我们为什么需要他？这个，当然不是用来计算四维空间的，这里，统一都叫齐次坐标。十几年前，网络不是那么发达，百度连百科都还没有，而我数学基础也差，为了搞明白这个齐次坐标，我花了大量的时间。这里要稍稍解释一下。

首先，为什么需要Vector4来做投影计算而不是Vector3？因为我们的投影矩阵是4x4矩阵而不是3x3矩阵。为什么是4x4？因为一个空间物体的变换，至少需要考虑到三块：平移、旋转、缩放。英文分别是：Translate, Rotate, Scale。我们可以分开一步一步计算这三块。但是为了效率考虑，最好一次计算完毕，这就需要用到矩阵。而一个3x3的矩阵，最多只能是包含旋转跟缩放的信息，如果需要再包含平移，最低只能需要一个4x4的矩阵。如图：



红色部分描述的是平移，蓝色部分描述的是旋转，而黑色部分描述的是缩放。按道理，如果你只想描述一个缩放，其实仅仅只需要一个三维向量即可。但是这里你需要做一个流水线作业，需要最快的处理速度，那么问题就变得更加复杂了。由于旋转跟缩放是在同一部分数据来描述，你需要确定一个顺序。所以，这部分代码大概率差不多这样：

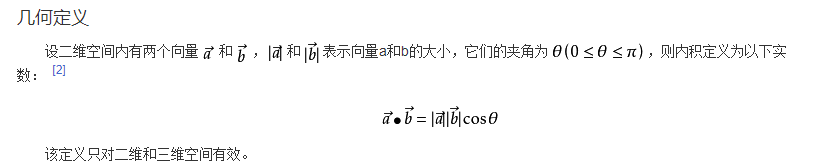


这里，你的scale影响到了整个rotate，而不仅仅是中间一行。

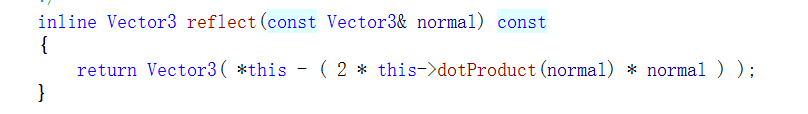
所以，使用齐次坐标的原因是我们需要用一个4x4的矩阵，而齐次坐标仅仅是把x，y，z的坐标变成：x，y，z，1。然后用矩阵乘以这个坐标即可。

Vector3，既可以描述成一个三维坐标，也可以描述为一个方向，这是基本功，应该都懂。向量我记得是高中数学，这里没有太多可以说的，例如向量的加减乘除，normalize，两个坐标的距离计算，都太简单了，随便找找就有，这里不打算详细讲。需要注意的我认为只有两个，一个是向量的点乘，一个是叉乘。

点乘：

点乘就是：return x1 \* x2 + y1 \* y2 + z1 \* z2;几何意义很重要，计算光照必须。

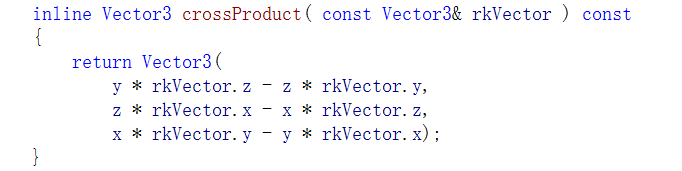
什么时候用到点乘？最常见的求向量的夹角的时候用到，计算光照的时候用到。应该说，一切跟向量夹角相关的时候，大概率都用到。光照计算为什么用到？高中物理有讲，入射，反射，折射都涉及到了角度，这里就举一个简单的例子：假设求一个平面的反射向量，就可以用点乘。例如，游戏中常见的，给水面加一个反射，给镜子加一个反射等等，就可能用到这块的计算，大概的代码如下：



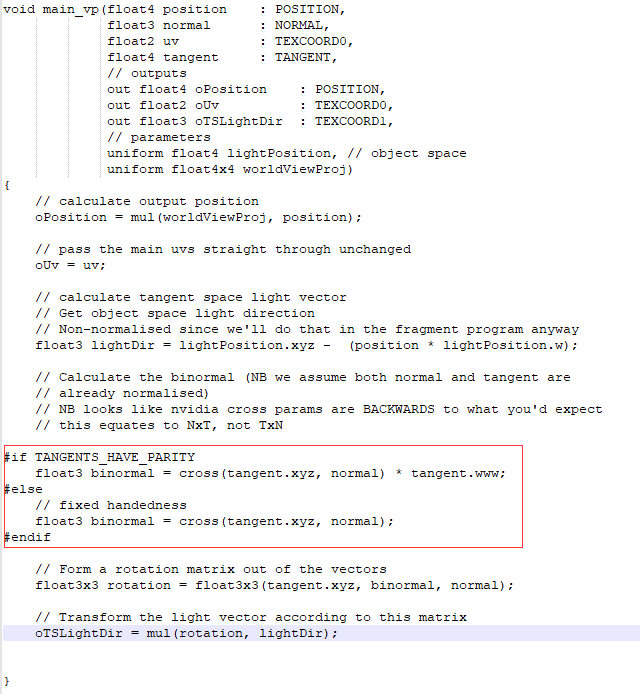
有兴趣的可以自己研究一下。我觉得向量的点乘是必须掌握的知识。

叉乘：

叉乘就是已知一个平面内的两个向量，求垂直于这个平面的向量。如图：



什么时候用到叉乘？大部分求垂直向量的时候，都可能用到。例如已知两个坐标轴求另外一个坐标轴（这个典型的就是法线贴图计算切线坐标系）。这个大概的计算是：已知一个三角形的法线，然后已知三角形的切线，就可以用叉乘计算副法线，构建一个坐标系。这个就是所谓的切线坐标系。这个在很多shader里面都能看到，大概代码如下：



老实说，如果连个向量都搞不明白的，搞图形学是没有什么前途的。点乘跟叉乘，是两个重点，必考，必会，所以这里提一下，其他感觉没什么好说的。

四元数。

描述物体方向的三种方案：欧拉角，四元数，矩阵。

欧拉角很好理解，分别可以定义为Pitch，Yaw，Roll。然而，欧拉角有个万向节死锁的问题，正规的3d引擎直接用欧拉角的，基本没有。

四元数，一个当初几乎没多少应用的数学工具，随着计算机图形学的发展，爆发了第二春。几乎所有3d引擎，都用四元数来描述物体的旋转。

矩阵，主要在最终shader计算里面用到，一个4x4矩阵包括了平移，旋转，缩放，一个矩阵乘法，完成一次华丽的空间变换，优秀。

那么，为什么旋转主要用四元数？优点在哪里？

主要有以下几个方面：

1. 四元数计算量比同样的矩阵计算量更小，速度也就更快。
2. 四元数比同样目的的矩阵占用的内存空间更小，四元数只需要4个浮点数，而3\*3矩阵，至少9个。
3. 四元数可以很方便做插值。例如一段时间内，物体从一个方向旋转到另外一个方向，那么slerp插值是个非常好的选择，而这个计算，在动画计算里面几乎是必备的。

四元数的定义大概是这样：

Struct Quaternion

{

Float x,y,z,w;

}

这里，虽然都是x，y，z，w。但是跟Vector4有本质的区别，所以绝对不能混用。

注意，也有定义为w，x，y，z的，所以要注意构造函数，不同引擎可能是不同的。

使用四元数的时候，有很多需要注意的地方。

首先，不要认为x，y，z，w跟欧拉角有一毛钱关系，他们毫无关系。当然了，你已知四元数，要求欧拉角，或者已知欧拉角，要求四元数，都是可以的。这需要一个比较复杂的计算。而不能简单的通过计算一个sin，cos之类的三角函数，就能够计算出四元数的几个分量。

其次，当你用欧拉角构造了四元数，然后做了旋转，如果你还想获得你现有的欧拉角，这是可行的吗？我刚入行的时候，做过这类尝试，后来发现结果都是不对的，当时我用的引擎是OGRE。后来，我解决这个问题的方案，是同时记录了欧拉角和四元数。这是为什么呢？我反思过这个问题。我是这样理解的：当你要得到某个旋转的方向，你可以通过不同的Pitch，Yaw，Roll来得到。那么，既然可以通过不同的旋转来实现，那么可能计算得出一个符合你要求的唯一的Pitch，Yaw，Roll吗？我的理解是不能的。后来我又看了其他引擎，发现不同引擎的四元数到欧拉角的求解，算法居然是完全不同的。例如UE4，引用的是这里的算法：<http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToEuler/>

我不能确定其他引擎，例如UE4，多次旋转之后，能不能得到你想要的欧拉角，我的猜测是不能的，我没能找到这方面的证明，对这块很熟悉的朋友也可以解释一下或者发一下资料链接。

四元数常用计算。

理论上，四元数有很多很多的计算方式，有很多很多的性质，例如四元数有加法，减法，有点积等等，但是，常用的其实没有那么多。下面，我列举一些常见的用法，一些不常用的用法，例如四元数的加法，减法，我就从来只在练习题里用过。

1. 四元数的乘法。这个非常关键，最常用，用于做基本的旋转。例如，最常见的Roll，Yaw，Pitch，都用乘法。再如沿着某个坐标轴旋转一个角度，也是用的乘法。有一点要切记：四元数不满足乘法交换律！因此，一切的四元数乘法绝对不能想当然的交换一下。计算方式大概是两步：
2. 、构建一个旋转四元数。
3. 、原四元数乘以构建的四元数。

举例：计算一个Yaw，也就是绕着Y轴旋转一个角度，那么可以先通过Axis，angle两个参数，构建出来一个四元数。由于是Axis，那么坐标系就是(0, 1, 0)。然后用原来的四元数，乘以构建的四元数即可（本地坐标系）。

1. 已知两个空间的向量，求一个四元数，让方向从向量1转到向量2。这也是一个比较常用到的计算。各大群里，关于四元数的问题这个估计能排进前三。这个的计算大概如下：
2. 、把两个向量叉乘，得到一个垂直于这两个向量平面的向量。
3. 、计算两个向量的夹角。这个可以通过点乘来计算，不会计算的话，一般引擎有接口。实在不会可以去查查点乘是怎么算的，或者上网去百度。
4. 、那么已知旋转向量以及旋转角度，可以通过接口算出来旋转四元数。
5. 、原四元数乘以新的四元数，可以得到新的旋转四元数。

3、四元数的插值。这点一般在镜头动画或者旋转动画的时候，都会用到。大概是：已知镜头的方向四元数q1，假设经过时间t\_delta之后，会到达另外一个旋转四元数q2，那么，现在求t\_cur的四元数（0 < t\_cur < t\_delta）。这种计算，一般引擎都有接口，接口名字一般是Slerp。

好了，四元数我也不能讲出花来，其他的四元数操作，例如转换成矩阵，点乘等等，随便一个引擎都有源码，看看源码或者接口都很方便，也没什么好讲的。

矩阵。

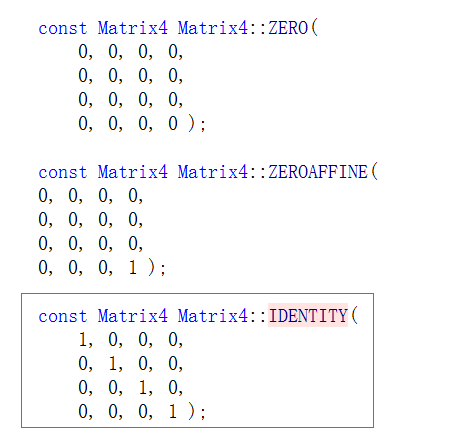
矩阵，应该是整个线性代数里面，最重要的组成部分。

矩阵最初的应用，我记得是解方程。例如一个N元一次方程组，用消元法来解，如果未知数不是那么多，那么不断消元，是没问题的。但是如果未知数的数量达到了几千个甚至更多，一次一次消元，就很坑了。我记得在哪里看过如何用矩阵来处理这类计算的。

矩阵有非常多的性质，例如有加法，减法，乘法，正交，转置，逆等等。而矩阵的加法和减法，我只在线性代数的习题里面做过，在3D计算里面就没用过。

下面，大概讲解一下矩阵的一些常见用法，性质，坑。

1. 矩阵的初始化。

一般的矩阵初始化，分为三种形式，而不是直接全部清零。但是大部分的矩阵初始化，都是初始化为单位矩阵。主要原因是矩阵一般用乘法，而单位矩阵最适合乘法。

上图中，红色部分就是单位矩阵。单位矩阵很重要，比0矩阵还要重要得多。0矩阵很少用，大部分时候都是用的单位矩阵。

2、矩阵的转置。在各种英文或者接口里，一般命名为：transpose

本来，我一直都觉得矩阵的转置没什么用，而且超级简单。无非就是行变成列，列变成行。

确实，如果你一直用DirectX，一直用hlsl，转置用到的地方就更少了。但是这些还是必须掌握的技能，因为早几年前我写HTC Vive openGL程序的时候，发现居然跟DX有很大差别。人家sample里面直接用的列矩阵。可怜，我一直都是用的行矩阵，完全忽略了这个，导致排查了半天。由此可见，说不定有些引擎，sample，计算等等，都是用的列矩阵。如果想当然的认为所有矩阵都是行矩阵，可能被坑。一旦发现别人用的列矩阵，那么转置一下即可，或者计算的时候需要注意矩阵是左乘还是右乘。

1. 矩阵的乘法。

这个是最常用的性质。矩阵的乘法规则这里就不抄了，网上大把。矩阵乘法有一些规则，必须非常熟悉，这里大概列一下：

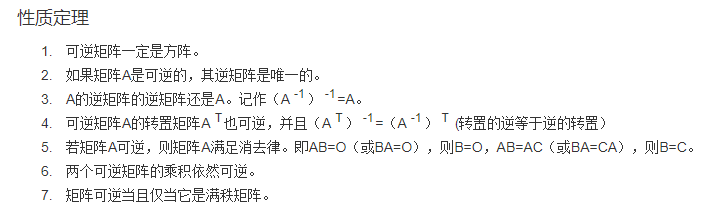
1. 、左乘、右乘。这个必须要搞清楚，尤其是搞openGL的，或者自己写GLSL等等shader的时候，不搞清楚有时候会自己坑自己。
2. 、矩阵乘法大概率不满足交换律，这点切记。
3. 、矩阵乘法有结合律，所以透视变换最重要的矩阵，WVP（world view projection）矩阵的时候，只要顺序对了就没事，一般会三个乘在一起传进shader，或者需要的时候三个分别传，也可以把world跟viewprojection分开传，很灵活。
4. 、矩阵乘法可以跟转置一起用，这点主要在OPENGL的时候用到，我用DX的时候很少用转置。

3、矩阵的逆。在英文里，一般命名为：inverse

由于矩阵不存在除法，所以，逆可以大体上理解为除法。其实就是一个矩阵乘以它的逆矩阵，等于单位矩阵。由于矩阵满足结合律，那么乘以一个矩阵的逆矩阵，可以大体上理解为除以这个矩阵。

那么问题来了，既然逆矩阵跟除法差不多的，为什么不直接搞矩阵的除法，而搞出来一套逆矩阵这类东东？我的理解是：首先，一般实数来说，只要除数不为0，你是可以任意除的。但是矩阵不同。绝大部分矩阵，都不存在逆矩阵！所以，必须搞出来一套东西，并且必须证明并且可以算出来逆矩阵，才可以这么干。所以，矩阵的逆矩阵，只能大体上理解为除法，而不能直接说就是除法。其次，矩阵只存在乘法的计算规则，而不存在除法的，至于为什么？我认为还是那句：这是宇宙规律。至于能不能证明，我就不知道了，估计需要专门研究数学的大佬才能给出答案，而我不能。

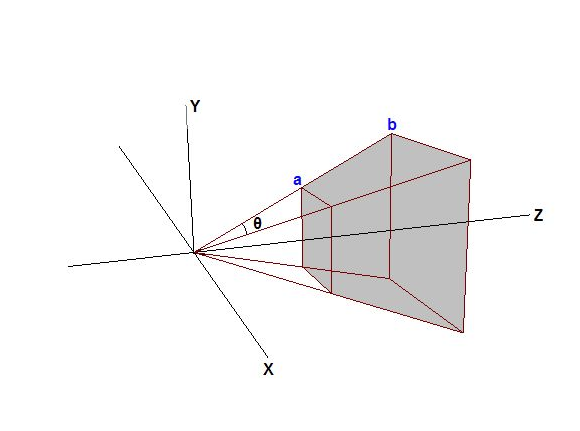
矩阵的逆矩阵有以下规则：（这部分是抄的百科，我觉得很重要）



以上是矩阵计算的一些常见性质和用法。还有一些必须掌握的，例如如何构建一个平移矩阵，构建一个旋转矩阵，构建一个缩放矩阵。如何构建一个投影矩阵，就算不会，至少可以抄代码，有一定的理解。

还有一些可能用到，但是不那么常用的性质，例如，倾斜矩阵。

在一般的3D游戏里，矩阵都是必然是一个正矩阵。正矩阵的意思是，视锥体就是一个正梯形，例如这种：

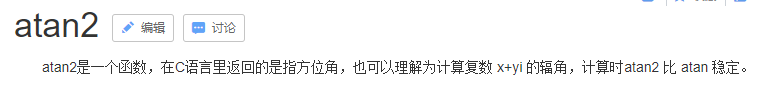


但是，有些时候，我们不能用正的梯形，可能需要用到倾斜梯形。什么时候会用到？例如做一个投影的拼接融合的时候。一些大型的展览展示项目，可能会做一些L型屏幕的投影，弧形屏幕的投影等等，这里，就可能要用到倾斜矩阵。还有VR，由于人的视角，上下左右的角度是不一样的，这里的矩阵计算就更加复杂了，我在VR的专门章节里有详细的讲述。如果有这方面的疑问，也可以直接留言问我。由于这部分用得并不多，这里不打算花费大量的篇幅来讲这块。

好了，矩阵这块暂时就想到这么多。

除了向量，四元数，矩阵，其他常用的数学主要有：三角函数，四舍五入，floor，ceil，clamp等，随便找个引擎看看就知道了。而shader里面，常用的更多一些，有一些函数，是为了不用或者尽量少用if语句而设计的，例如saturate。

三角函数里面，有一个以前教科书上没有见过的，Atan2.反正切函数Atan是教科书里面就有的，为什么需要新增加一个Atan2？这个在百度百科里就有很详细的讲述，这个我认为是必须要搞明白的，因为这个常用，而Atan没那么常用。



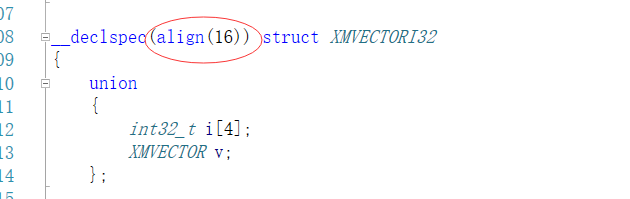
比Atan更稳定的原因，网上也有大量讲述，我就不抄了，这个是重点需要掌握的函数。

3D数学这么重要，有什么优化的方法吗？

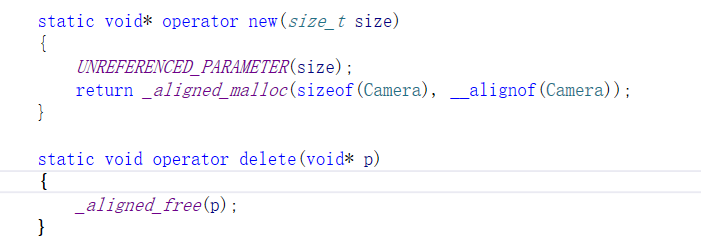
有的，早期的时候，3D数学库，还有很多是直接用汇编写的。但是，后来慢慢又都不用了，原因很简单，速度不会快多少，但是代码可读性，跨平台等等，都不那么好用。典型的例如OGRE，老版本的时候，数学库里一大堆汇编，后来都删掉了。再后来，后续又流行了SIMD优化。这个SIMD优化，其实就是单指令多数据，例如矩阵这种计算，就很适合做SIMD优化。DirectxX的数学库里，就清一色用了SIMD优化，有兴趣的，打开DirectXMath.h头文件就能看得到。那么，这个是必须的吗？我其实并没有那么认为。一般来说，一个引擎的数学计算，基本上不会是瓶颈，瓶颈一般会在于GPU渲染，或者其他大量的逻辑处理。当然了，如果你的引擎需要用到大量的CPU计算，那当我没说。

不过，这东西本来就见仁见智，做一下SIMD优化，一般来说，代码看起来至少逼格比较高一些。就好比曾经有个迅雷的大佬，就跟我吹嘘他们做了多少的SIMD优化，例如图片混合，用SIMD优化速度多快多快。我其实心里在默默吐槽：你那么要求效率，做图片混合，怎么不用GPU呢？不用CUDA呢？这绝壁秒杀你的SIMD。

所以，我的意见是，如果真有需要大规模的并行计算的时候，用CUDA，用compute shader，会是个更优的选择，而不是SIMD优化。

SIMD优化还要注意一下对齐，这个必须是16字节对齐。这点切记。

而且，使用的时候，还需要重载一下new跟delete，大概这样：



这里我记得还有其他写法，需要用到的可以查查。但是，如果你的类成员变量并没有用到这类带SIMD优化的结构体，是不需要这么干的。我这里用到了D3D的。

好了，关于3D数学的部分，先讲到这里，可能有很多没想到的，想到了再补充。有什么不对的，欢迎留言喷，让我提升一下。时间有限，就不回去检查了。