四元数

http://www.qiujiawei.com/understanding-quaternions/ （重要）

http://blog.csdn.net/candycat1992/article/details/41254799

**四元数介绍**

旋转，应该是三种坐标变换——缩放、旋转和平移，中最复杂的一种了。大家应该都听过，有一种旋转的表示方法叫四元数。按照我们的习惯，我们更加熟悉的是另外两种旋转的表示方法——矩阵旋转和[欧拉旋转](http://zh.wikipedia.org/wiki/%E6%AC%A7%E6%8B%89%E8%A7%92)。矩阵旋转使用了一个4\*4大小的矩阵来表示绕任意轴旋转的变换矩阵，而欧拉选择则是按照一定的坐标轴顺序（例如先x、再y、最后z）、每个轴旋转一定角度来变换坐标或向量，它实际上是一系列坐标轴旋转的组合。

那么，[四元数](http://zh.wikipedia.org/wiki/%E5%9B%9B%E5%85%83%E6%95%B8)又是什么呢？简单来说，四元数本质上是一种高阶复数（听不懂了吧。。。），是一个四维空间，相对于复数的二维空间。我们高中的时候应该都学过复数，一个复数由实部和虚部组成，即x = a + bi，i是虚数单位，如果你还记得的话应该知道i^2 = -1。而四元数其实和我们学到的这种是类似的，不同的是，它的虚部包含了三个虚数单位，i、j、k，即一个四元数可以表示为x = a + bi + cj + dk。那么，它和旋转为什么会有关系呢？

在Unity里，tranform组件有一个变量名为rotation，它的类型就是四元数。很多初学者会直接取rotation的x、y、z，认为它们分别对应了Transform面板里R的各个分量。当然很快我们就会发现这是完全不对的。实际上，四元数的x、y、z和R的那三个值从直观上来讲没什么关系，当然会存在一个表达式可以转换，在后面会讲。

大家应该和我一样都有很多疑问，既然已经存在了这两种旋转表示方式，为什么还要使用四元数这种听起来很难懂的东西呢？我们先要了解这三种旋转方式的优缺点：

* **矩阵旋转**
  + 优点：
    - 旋转轴可以是任意向量；
  + 缺点：
    - 旋转其实只需要知道一个向量+一个角度，一共4个值的信息，但矩阵法却使用了16个元素；
    - 而且在做乘法操作时也会增加计算量，造成了空间和时间上的一些浪费；
* **欧拉旋转**
  + 优点：
    - 很容易理解，形象直观；
    - 表示更方便，只需要3个值（分别对应x、y、z轴的旋转角度）；但按我的理解，它还是转换到了3个3\*3的矩阵做变换，效率不如四元数；
  + 缺点：
    - 之前提到过这种方法是要按照一个固定的坐标轴的顺序旋转的，因此不同的顺序会造成不同的结果；
    - 会造成[万向节锁](http://zh.wikipedia.org/wiki/%E8%90%AC%E5%90%91%E9%8E%96)（Gimbal Lock）的现象。这种现象的发生就是由于上述固定坐标轴旋转顺序造成的。理论上，欧拉旋转可以靠这种顺序让一个物体指到任何一个想要的方向，但如果在旋转中不幸让某些坐标轴重合了就会发生万向节锁，这时就会丢失一个方向上的旋转能力，也就是说在这种状态下我们无论怎么旋转（当然还是要原先的顺序）都不可能得到某些想要的旋转效果，除非我们打破原先的旋转顺序或者同时旋转3个坐标轴。这里有个[视频](http://v.youku.com/v_show/id_XNzkyOTIyMTI=.html)可以直观的理解下；
    - 由于万向节锁的存在，欧拉旋转无法实现球面平滑插值；
* **四元数旋转**
  + 优点：
    - 可以避免万向节锁现象；
    - 只需要一个4维的四元数就可以执行绕任意过原点的向量的旋转，方便快捷，在某些实现下比旋转矩阵效率更高；
    - 可以提供平滑插值；
  + 缺点：
    - 比欧拉旋转稍微复杂了一点点，因为多了一个维度；
    - 理解更困难，不直观；

**四元数和欧拉角**

**基础知识**

前面说过，一个四元数可以表示为q = w + xi + yj + zk，现在就来回答这样一个简单的式子是怎么和三维旋转结合在一起的。为了方便，我们下面使用q = ((x, y, z)，w) = (v, w)，其中v是向量，w是实数，这样的式子来表示一个四元数。

我们先来看问题的答案。我们可以使用一个四元数q=((x,y,z)sinθ2, cosθ2) 来执行一个旋转。具体来说，如果我们想要把空间的一个点P绕着单位向量轴u = (x, y, z)表示的旋转轴旋转θ角度，我们首先把点P扩展到四元数空间，即四元数p = (P, 0)。那么，旋转后新的点对应的四元数（当然这个计算而得的四元数的实部为0，虚部系数就是新的坐标）为：

p′=qpq−1

其中，q=(cosθ2, (x,y,z)sinθ2) ，q−1=q∗N(q)，由于u是单位向量，因此  
N(q)=1，即q−1=q∗。右边表达式包含了四元数乘法。相关的定义如下：

* **四元数乘法：q1q2=(v1→×v2→+w1v2→+w2v1→,w1w2−v1→⋅v2→)**

* **共轭四元数：q∗=(−v⃗ ,w)**
* **四元数的模：N(q) = √(x^2 + y^2 + z^2 +w^2)，即四元数到原点的距离**
* **四元数的逆：q−1=q∗N(q)**

http://www.cnblogs.com/yiyezhai/p/3176725.html

它的证明这里不再赘述，有兴趣的可以参见[这篇文章](http://www.cnblogs.com/yiyezhai/p/3176725.html)。主要思想是构建了一个辅助向量k，它是将p绕旋转轴旋转θ/2得到的。证明过程尝试证明wk∗=kv∗，以此证明w与v、k在同一平面内，且与v夹角为θ。

我们举个最简单的例子：把点P(1, 0, 1)绕旋转轴u = (0, 1, 0)旋转90°，求旋转后的顶点坐标。首先将P扩充到四元数，即p = (P, 0)。而q = (u\*sin45°, cos45°)。求p′=qpq−1的值。建议大家一定要在纸上计算一边，这样才能加深印象，连笔都懒得动的人还是不要往下看了。最后的结果p` = ((1, 0, -1), 0)，即旋转后的顶点位置是(1, 0, -1)。

如果想要得到复合旋转，只需类似复合矩阵那样左乘新的四元数，再进行运算即可。

我们来总结下四元数旋转的**几个需要注意的地方**：

* 用于旋转的四元数，每个分量的范围都在（-1，1）；
* 每一次旋转实际上需要两个四元数的参与，即q和q\*；
* 所有用于旋转的四元数都是单位四元数，即它们的模是1；

下面是几点建议：

* 实际上，在Unity里即便你不知道上述公式和变换也丝毫不妨碍我们使用四元数，但是有一点要提醒你，**除非你对四元数非常了解，那么不要直接对它们进行赋值**。
* 如果你不想知道原理，只想在Unity里找到对应的函数来进行四元数变换，那么你可以使用这两个函数：[Quaternion.Euler](http://docs.unity3d.com/ScriptReference/Quaternion.Euler.html)和[Quaternion.eulerAngles](http://docs.unity3d.com/ScriptReference/Quaternion-eulerAngles.html)。它们基本可以满足绝大多数的四元数旋转变换。

**和其他类型的转换**

首先是**轴角到四元数**：

给定一个单位长度的旋转轴(x, y, z)和一个角度θ。对应的四元数为：

q=((x,y,z)sinθ2, cosθ2)

这个公式的推导过程上面已经给出。

**欧拉角到四元数**：

给定一个欧拉旋转(X, Y, Z)（即分别绕x轴、y轴和z轴旋转X、Y、Z度），则对应的四元数为：

x = sin(Y/2)sin(Z/2)cos(X/2)+cos(Y/2)cos(Z/2)sin(X/2)  
y = sin(Y/2)cos(Z/2)cos(X/2)+cos(Y/2)sin(Z/2)sin(X/2)  
z = cos(Y/2)sin(Z/2)cos(X/2)-sin(Y/2)cos(Z/2)sin(X/2)  
w = cos(Y/2)cos(Z/2)cos(X/2)-sin(Y/2)sin(Z/2)sin(X/2)  
q = ((x, y, z), w)

它的证明过程可以依靠轴角到四元数的公式进行推导。

其他**参考链接**：

1. [**Euler to Quaternion**](http://www.euclideanspace.com/maths/geometry/rotations/conversions/eulerToQuaternion/index.htm)  
2. [**Quaternion To Euler**](http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToEuler/index.htm)  
3. [**AngleAxis to Quaternion**](http://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToQuaternion/index.htm)  
4. [**Quaternion to AngleAxis**](http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToAngle/index.htm)

**四元数的插值**

这里的插值指的是球面线性插值。

设t是一个在0到1之间的变量。我们想要基于t求Q1到Q2之间插值后四元数Q。它的公式是：

Q3  = (sin((1-t)A)/sin(A))\*Q1 + (sin((tA)/sin(A))\*Q2)

Q = Q3/|Q3|，即单位化

**四元数的创建**

在了解了上述知识后，我们就不需要那么惧怕四元数了，实际上它和矩阵类似，不同的只是它的表示方式以及运算方式。那么在Unity里如何利用四元数进行旋转呢？Unity里提供了非常多的方式来创建一个四元数。例如Quaternion.AngleAxis(float angle, Vector3 axis)，它可以返回一个绕轴线axis旋转angle角度的四元数变换。我们可以一个Vector3和它进行左乘，就将得到旋转后的Vector3。在Unity里只需要用一个“ \* ”操作符就可以进行四元数对向量的变换操作，相当于我们上述讲到的p′=qpq−1操作。如果我们想要进行多个旋转变换，只需要左乘其他四元数变换即可。例如下面这样：

**[csharp]** [view plain](http://blog.csdn.net/candycat1992/article/details/41254799) [copy](http://blog.csdn.net/candycat1992/article/details/41254799)

[print?](http://blog.csdn.net/candycat1992/article/details/41254799)[在CODE上查看代码片](https://code.csdn.net/snippets/529322)

1. Vector3 newVector = Quaternion.AngleAxis(90, Vector3.up) \* Quaternion.LookRotation(someDirection) \* someVector;

尽管欧拉角更容易我们理解，但四元数比欧拉角要强大很多。Unity提供了这两种方式供我们选择，我们可以选择最合适的变换。

例如，如果我们需要对旋转进行插值，我们可以首先使用Quaternion.eulerAngles来得到欧拉角度，然后使用Mathf.Clamp对其进行插值运算。

最后更新Quaternion.eulerAngles或者使用Quaternion.Euler(yourAngles)来创建一个新的四元数。

又例如，如果你想要组合旋转，比如让人物的脑袋向下看或者旋转身体，两种方法其实都可以，但一旦这些旋转不是以世界坐标轴为旋转轴，比如人物扭动脖子向下看等，那么四元数是一个更合适的选择。Unity还提供了transform.forward, transform.right and transform.up 这些非常有用的轴，这些轴可以和Quaternion.AngleAxis组合起来，来创建非常有用的旋转组合。例如，下面的代码让物体执行低头的动作：

**[csharp]** [view plain](http://blog.csdn.net/candycat1992/article/details/41254799) [copy](http://blog.csdn.net/candycat1992/article/details/41254799)

[print?](http://blog.csdn.net/candycat1992/article/details/41254799)[在CODE上查看代码片](https://code.csdn.net/snippets/529322)

1. transform.rotation = Quaternion.AngleAxis(degrees, transform.right) \* transform.rotation;

关于Quaternion的其他函数，后面再补充吧，原理类似~

**补充：欧拉旋转**

在文章开头关于欧拉旋转的细节没有解释的太清楚，而又有不少人询问相关问题，我尽量把自己的理解写到这里，如有不对还望指出。

**欧拉旋转是怎么运作的**

欧拉旋转是我们最容易理解的一种旋转方式。以我们生活中为例，一个舞蹈老师告诉我们，完成某个舞蹈动作需要先向你的左边转30°，再向左侧弯腰60°，再起身向后弯腰90°（如果你能办到的话）。上面这样一个旋转的过程其实和我们在三维中进行欧拉旋转很类似，即我们是通过指明绕三个轴旋转的角度来进行旋转的，不同的是，日常生活中我们更愿意叫这些轴为前后左右上下。而这也意味着我们需要指明一个旋转顺序。这是因为，先绕X轴旋转90°、再绕Y轴30°和先绕Y轴旋转90°、再绕X轴30°得到的是不同的结果。

在Unity里，欧拉旋转的旋转顺序是Z、X、Y，这在相关的API文档中都有说明，例如[Transform.Rotate](http://docs.unity3d.com/ScriptReference/Transform.Rotate.html)。其实文档中说得不是非常详细，还有一个细节我们需要明白。如果你仔细想想，就会发现有一个非常重要的东西我们没有说明白，那就是旋转时使用的坐标系。给定一个旋转顺序（例如这里的Z、X、Y），以及它们对应的旋转角度（α，β，r），有两种坐标系可以选择：

1. 绕坐标系E下的Z轴旋转α，绕坐标系E下的Y轴旋转β，绕坐标系E下的X轴旋转r，即进行一次旋转时不一起旋转当前坐标系；
2. 绕坐标系E下的Z轴旋转α，绕坐标系E在绕Z轴旋转α后的新坐标系E'下的Y轴旋转β，绕坐标系E'在绕Y轴旋转β后的新坐标系E''下的X轴旋转r， 即在旋转时，把坐标系一起转动；

很容易知道，这两种选择的结果是不一样的。但如果把它们的旋转顺序颠倒一下，其实结果就会一样。说得明白点，在第一种情况下、按ZXY顺序旋转和在第二种情况下、按YXZ顺序旋转是一样的。证明方法可以看下[这篇文章](http://www.cnitblog.com/luckydmz/archive/2010/09/07/68674.html)。而Unity文档中说明的旋转顺序指的是在第一种情况下的顺序。

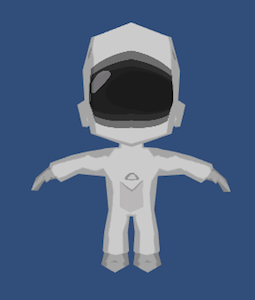
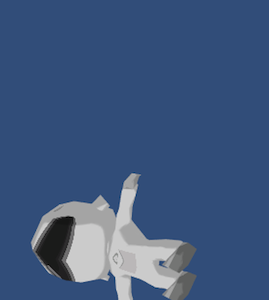
如果你还是不懂这意味着什么，可以试着调用下这个函数。例如，你认为下面代码的结果是什么：

**[csharp]** [view plain](http://blog.csdn.net/candycat1992/article/details/41254799) [copy](http://blog.csdn.net/candycat1992/article/details/41254799)

[print?](http://blog.csdn.net/candycat1992/article/details/41254799)[在CODE上查看代码片](https://code.csdn.net/snippets/529322)

1. transform.Rotate(**new** Vector3(0, 30, 90));

原模型的方向和执行结果如下：

而我们可以再分别执行下面的代码：

**[csharp]** [view plain](http://blog.csdn.net/candycat1992/article/details/41254799) [copy](http://blog.csdn.net/candycat1992/article/details/41254799)

[print?](http://blog.csdn.net/candycat1992/article/details/41254799)[在CODE上查看代码片](https://code.csdn.net/snippets/529322)

1. // First case
2. transform.Rotate(**new** Vector3(0, 30, 0));
3. transform.Rotate(**new** Vector3(0, 0, 90));
5. // Second case
6. //      transform.Rotate(new Vector3(0, 0, 90));
7. //      transform.Rotate(new Vector3(0, 30, 0));

两种情况的结果分别是：

可以发现，调用transform.Rotate(new Vector3(0, 30, 90));是和第一种情况中的代码是一样的结果，即先旋转Y、再旋转Z。进一步实验，我们会发现transform.Rotate(new Vector3(30, 90, -40));的结果是和transform.Rotate(new Vector3(0, 90, 0));transform.Rotate(new Vector3(30, 0, 0));transform.Rotate(new Vector3(0, 0, -40));的结果一样的。你会问了，文档中不是明明说了旋转顺序是Z、X、Y吗？怎么现在完全反过来了呢？原因就是我们之前说的两种坐标系的选择。在一次调用transform.Rotate的过程中，坐标轴是不随每次单个坐标轴的旋转而旋转的。而在调用transform.Rotate后，这个旋转坐标系才会变化。也就是说，transform.Rotate(new Vector3(30, 90, -40));执行时使用的是第一种情况，而transform.Rotate(new Vector3(0, 90, 0));transform.Rotate(new Vector3(30, 0, 0));transform.Rotate(new Vector3(0, 0, -40));每一句则是分别使用了上一句执行后的坐标系，即第二种坐标系情况。因此，我们看起来顺序好像是完全是反了，但结果是一样的。

上面只是说了一些容易混淆的地方，更多的内容大家可以搜搜wiki之类的。

**数学模型**

欧拉旋转的数学实现就是使用矩阵。而最常见的表示方法就是3\*3的矩阵。在[Wiki](http://en.wikipedia.org/wiki/Gimbal_lock)里我们可以找到这种矩阵的表示形式，以下以按XYZ的旋转顺序为例，三个矩阵分别表示了：

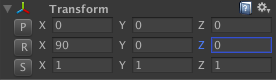


在计算时，我们将原来的旋转矩阵右乘（这里使用的是列向量）上面的矩阵。从这里我们也可以证明上面所说的两种坐标系选择是一样的结果，它们之间的不同从这里来看其实就是矩阵相乘时的顺序不同。第一种坐标系情况，指的是在计算时，先从左到右直接计算R中3个矩阵的结果矩阵，最后再和原旋转矩阵相乘，因此顺序是XYZ；而第二种坐标系情况，指的是在计算时，从右往左依次相乘，因此顺序是反过来的，ZYX。你可以验证R左乘和右乘的结果表达式，就可以相信这个结论了！

**万向节锁**

虽然欧拉旋转非常容易理解，但它会造成臭名昭著的万向节锁问题。我之前给出了链接大家可能都看了，但还是不明白这是怎么回事。这里[有一篇文章](http://blog.csdn.net/huazai434/article/details/6458257)是我目前找到说得最容易懂的中文文章，大家可以看看。

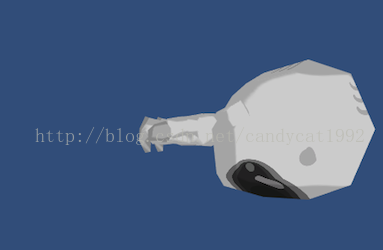
如果你还是不明白，我们来做个试验。还是使用之前的模型，这次我们直接在面板中把它的欧拉角中的X值设为90°，其他先保持不变：



此时模型是脸朝下（下图你看到的只是一个头顶）：



现在，如果我让你不动X轴，只设置Y和Z的值，把这个模型的脸转上来，让它向侧面看，你可以办到吗？你可以发现，这时候无论你怎么设置Y和Z的值，模型始终是脸朝下、在同一平面旋转，看起来就是Y和Z控制的是同一个轴的旋转，下面是我截取的任意两种情况：

这就是一种万向节锁的情况。这里我们先设置X轴为90°也是有原因的，这是因为Unity中欧拉角的旋转顺序是ZXY，即X轴是第二个旋转轴。当我们在面板中设置任意旋转值时，Unity实际是按照固定的ZXY顺序依次旋转特定角度的。

在代码里，我们同样可以重现万向节锁现象。

**[csharp]** [view plain](http://blog.csdn.net/candycat1992/article/details/41254799) [copy](http://blog.csdn.net/candycat1992/article/details/41254799)

[print?](http://blog.csdn.net/candycat1992/article/details/41254799)[在CODE上查看代码片](https://code.csdn.net/snippets/529322)

1. transform.Rotate(**new** Vector3(0, 0, 40));
2. transform.Rotate(**new** Vector3(0, 90, 0));
3. transform.Rotate(**new** Vector3(80, 0, 0));

我们只需要固定中间一句代码，即使Y轴的旋转角度始终为90°，那么你会发现无论你怎么调整第一句和最后一句中的X或Z值，它会像一个钟表的表针一样总是在同一个平面上运动。

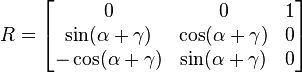
万向节锁中的“锁”，其实是给人一种误导，这可能也是让很多人觉得难以理解的一个原因。实际上，实际上它并没有锁住任何一个旋转轴，只是说我们会在这种旋转情况下会感觉丧失了一个维度。以上面的例子来说，尽管固定了第二个旋转轴的角度为90°，但我们原以为依靠改变其他两个轴的旋转角度是可以得到任意旋转位置的（因为按我们理解，两个轴应该控制的是两个空间维度），而事实是它被“锁”在了一个平面，即只有一个维度了，缺失了一个维度。而只要第二个旋转轴不是±90°，我们就可以依靠改变其他两个轴的旋转角度来得到任意旋转位置。

**数学解释**

我们从最简单的矩阵来理解。还是使用XYZ的旋转顺序。当Y轴的旋转角度为90°时，我们会得到下面的旋转矩阵：



我们对上述矩阵进行左乘可以得到下面的结果：



可以发现，此时当我们改变第一次和第三次的旋转角度时，是同样的效果，而不会改变第一行和第三列的任何数值，从而缺失了一个维度。

我们再尝试着理解下它的本质。[Wiki](http://en.wikipedia.org/wiki/Gimbal_lock)上写，万向节锁出现的本质原因，是因为从欧拉角到旋转的映射并不是一个覆盖映射，即它并不是在每个点处都是局部同胚的。不懂吧。。。恩，我们再来通俗一下解释，这意味着，从欧拉角到旋转是一个多对一的映射（即不同的欧拉角可以表示同一个旋转方向），而且并不是每一个旋转变化都可以用欧拉角来表示。其他更多的大家去参考wiki吧。

建议还是多看看视频，尤其是后面的部分。当然，如果还是觉得懵懵懂懂的话，在《3D数学基础：图形与游戏开发》一书中有一话说的很有道理，“如果您从来没有遇到过万向锁情况，你可能会对此感到困惑，而且不幸的是，很难在本书中讲清楚这个问题，你需要亲身经历才能明白。”因此，大家也不要纠结啦，等到遇到的时候可以想到是因为万向节锁的原因就好。