# 游戏开发数学基础

数学是游戏引擎的基础。

游戏有各种各样的类型，有非常抽象的游戏， 也有偏向模拟现实世界的游戏。对于抽象类的游戏来讲，游戏规则本身还是要有一定的数学基础，例如纸牌和麻将；某些小游戏看似画面抽象，但是还是会构建于完整的物理系统之上，比如流行的“割绳子”；模拟现实的游戏自然更无需多言，对于拟真的3D游戏来说，无论摄像机移动、角色的移动、阴影的显示、输入操作的转换还是粒子的播放，每一帧背后都是大量数学运算在支持。游戏引擎和游戏逻辑的实现必须基于坚实的数学基础，才能带给玩家真实感人的体验，否则可能会让玩家感觉到虚假或者困惑，从而无法沉浸到游戏世界中去。

在3D游戏的开荒时代，以《DOOM》、《QUAKE》为代表的3D游戏将当时最前沿的图形学技术进行改造和变革，实现了在一秒之内渲染数十次的目标，奠定了现代3D游戏的基础。到了今天，游戏引擎已经得到了充分的发展，我们已经不需要再从计算顶点、渲染像素开始从头制作游戏了。虽然如此，当需要制作一款具有独特性的游戏时，依然有很多至关重要的问题需要考虑：玩家的输入如何转化为角色的行动？角色的行动和动画如何拟合？摄像机如何配合角色的移动？等等，这些问题关系到了游戏的手感、表现力和游戏性，是游戏成败的关键。而完美地实现我们最初想要的效果，也需要有完备的数学算法的支持。

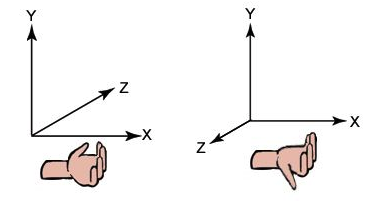
因此数学也是游戏机制的基础。

游戏数学的内容非常庞杂，市面上有不少书籍整本书都在讨论相关的问题。本章只讨论游戏中最普遍、最基本的一些数学问题，目标是帮助初学游戏开发的读者们能够解决一部分游戏开发中的计算相关问题。

## 坐标系

### 左手、右手坐标系

3D坐标系是3D数学的基本概念。3D软件一般都采用笛卡尔坐标系来描述物体的坐标信息。笛卡尔坐标系可以是左手坐标系也可以是右手坐标系，先看一下两种坐标系的图示：



（左图：左手坐标系。右图：右手坐标系）

3D空间中的朝向问题总是复杂的，两种坐标系看似一样，实际上是镜像对称的，就像左右手一样。为什么要叫左手、右手坐标系呢？如图，手掌沿X轴放平，向Y轴握拳，如果左手拇指指向Z轴方向，就是左手坐标系；如果左手拇指方向和Z轴相反，那么就是右手坐标系。这时如果换成右手，右手拇指就会和Z轴方向相同了，因为我们的手也是镜像对称的。

通常坐标值用放在小括号中的三个数表示，分别是(x, y, z)，比如坐标(1, 0, 2)。

Unity采用左手坐标系，且X、Y、Z轴的默认方向与上图中的左边完全一致。X、Y、Z默认为右、上、前。

### 全局坐标系

无论生活中还是游戏开发中，我们总是在使用不同的坐标系在指代方位，以回答问路为例：

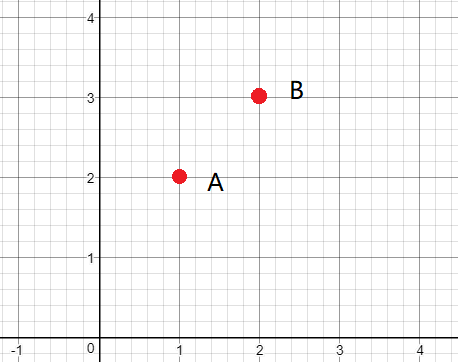
1. 超市就在我的左手边。这是以回答者本人的坐标系来说的。
2. 往前走，第一个路口左拐直走再右拐就到了。这里的左、右是以行人的坐标系来说的。
3. 超市在南边200米的位置。这是以全局坐标系来说的，东南西北是常用的指示全局坐标系的方法。

全局坐标系是场景内所有物体和方向的基准，也称为世界坐标系。在全局坐标系中原点(0, 0, 0)是所有物体位置的基准，且全局坐标系指定了统一的X、Y、Z朝向。例如新建一个物体坐标为(1, 2, 1)。那么它在X方向离原点1米，Y方向离原点2米，Z方向离原点1米。

### 局部坐标系

每个物体都有其独立的坐标系，并且随着物体进行相同的移动或者旋转，也称本地坐标系。模型Mesh保存的顶点坐标均为局部坐标系下的坐标。

Unity中，局部坐标系与“父子关系”这一概念有很强的相关性，子物会以父物体的坐标系作为定位自身的坐标系。



如图，A点全局坐标(1,2)，B点全局坐标(2,3)，如果B是A的子物体，那么B点的局部坐标为(1,1)；如果A点是B点的子物体，A点的坐标为(-1, -1)。

局部坐标系有一个特例，是相机坐标系。相机作为一个组件，也是挂载在某个物体上的，那个物体的坐标系就是相机的坐标系。相机坐标系有很多用途，用它可以方便地判断物体是否在相机前方，以及物体之间的前后遮挡关系。

### 屏幕坐标系

2D游戏和3D游戏的世界坐标系不尽相同，但是屏幕坐标系一定是2D的，因为显示设备是平面的（就算是VR设备，也是由两个屏幕组成的）。

Unity中，可以使用transform.TransformPoint方法将局部坐标转换为全局的，transform.InverseTransformPoint可以将全局坐标转换为局部的。transform.TransformDirection和transform.InverseTransformDirection则用于对向量在全局和局部坐标系之间转换。

以下示例将讲解如何通过改变物体的全局和局部坐标系来改变物体的运动方向。

1. 在Unity场景中，先新建一个立方体，并将旋转的Y值改为300，也就是在Y轴旋转300度。
2. 新建脚本CoordinateLocal.cs，内容如下：

|  |
| --- |
| using UnityEngine;  public class CoordinateLocal : MonoBehaviour {  void Start() {  }  void Update() {  transform.Translate(Vector3.forward \* Time.deltaTime);  }  } |

将本脚本挂载到立方体上，运行游戏，会看到立方体沿着自身的Z轴方向慢慢移动。

1. 再新建脚本CoorindateWorld.cs，内容如下：

|  |
| --- |
| using UnityEngine;  public class CoordinateLocal : MonoBehaviour {  void Start() {  }  void Update() {  Vector3 v = transform.InverseTransformDirection(Vector3.forward);  transform.Translate(v \* Time.deltaTime);  }  } |

将新建的脚本挂载到立方体上，并取消勾选原来的脚本CoordinateLocal，就只有新的脚本发挥作用了。可以通过取消/勾选的方式在两个脚本中切换。

这时运行游戏，就发现立方体会沿着世界坐标系的Z轴方向移动了。

解释一下以上结果：

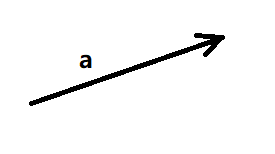
transform.Translate函数，默认是以局部坐标系为基准的，所以第一个脚本中，虽然Translate函数参数为Vector3.forward，依然会以局部坐标系的前方为准。Vector3.forward的值是常数(0, 0, 1)，它在不同的坐标系下代表不同的“前方”。

第二个脚本就比较复杂了，由于Translate函数会以局部坐标为准，我们就要把世界坐标的“前方”转化为局部坐标的向量v，这里我们认为Vector3.forward是世界的“前方”，用InverseTransformDirection方法将这个向量(0, 0, 1)以局部坐标系表示（也就是v），然后以v作为参数来执行Translate方法，就会朝世界的前方移动了。

## 向量

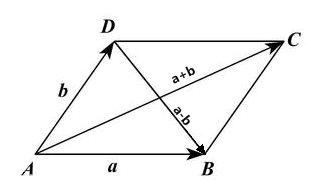
向量的重要性不言而喻。在游戏开发中会反复用到向量的概念，以及Vector3这个数据类型。不仅在空间中，位置和方向可以用向量表示，而且物理系统中有向量有更多用途，它可以表示力、速度、加速度等等概念。

在数学中，既有大小又有方向的量就是向量。在几何中，向量可以用一段有方向的线段来表示：



### 向量的加法

向量的加法为x、y、z分量分别相加，在几何上表示方法如下图：



从A点到C点的向量，就是**a**+**b**的结果。注意由于从D到C的向量与向量**a**完全相等，所以**a**+**b**既可以看做是平行四边形的对角线，也可以看做加上得到的。这就得到了向量加法在几何中的两种表示方法：首尾相接法和平行四边形法。

首尾相接法就是先平移一个向量，让它的起点与另一个向量的终点重合，然后连接另一个向量的起点和该向量的终点。

平行四边形法是将两个向量的起点放在一起，然后作一个平行四边形，对角线向量即两个向量的和。

在物理上，向量相加用来计算两个力的合力，或者几个速度分量的叠加。

在日常生活中也有向量加法的例子，比如从家里走到学校，再走到商店，将这两个位移相加，就得到了从家里直接到商店的位移。

### 向量的减法

先解释什么是负向量，向量-**a**就是大小和**a**相同，方向相反的一个向量，称之为向量**a**的负向量。

那么运算**a**-**b**就可以看做是**a**+(-**b**)，前面加法的例图中已经标明了**a**-**b**的几何表示方法。具体做法：将两个向量的起点放在一起，以**b**的终点开始，到**a**的终点结束，这个向量就是**a**-**b**。

### 点乘

两个向量的点积是一个标量，其数值为二者长度相乘，再乘以二者夹角的余弦：



通过两个向量的点积可以快速判断两个向量的夹角：

若点积等于0，则二者垂直；

若点积大于0，则二者夹角小于90度。

若点积小于0，则二者夹角大于90度。

### 叉乘

两个向量的叉积是一个新的向量，新向量垂直于原来的两个向量，且长度为二者长度相乘，再乘以夹角的正弦值。

叉积的方向也可以用左手判断（用左手还是右手和坐标系有关），手掌沿第一个向量放平，向第二个向量握拳，拇指指向的即叉积方向。

可以看出，两个向量叉乘的顺序不同，手掌转向会不同，所以叉乘不满足交换律。

### Vector3结构体

提醒：C#中有类（class）和结构体（struct）的区别，虽然它们都具有字段、属性、方法，但是前者是引用类型，后者是值类型，在使用时区别很大。Vector2和Vector3属于结构体，详情可参考C#语法相关资料。

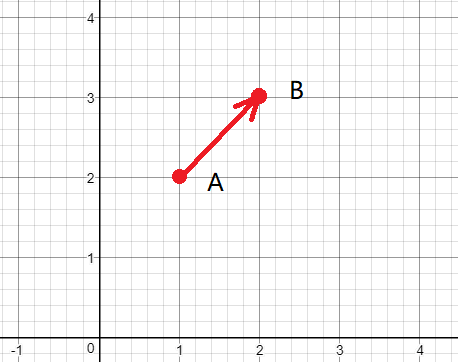
在Unity中，和向量有关的结构体有Vector2、Vector3，分别用来表示二维和三维向量，其中Vector3最为常用，下面列举Vector3的属性和方法：

|  |  |
| --- | --- |
| 字段或属性 | 说明 |
| x | 向量的x分量 |
| y | 向量的y分量 |
| z | 向量的z分量 |
| normalized | 得到单位化向量（方向相同，长度为1） |
| magnitude | 得到向量长度，长度是标量 |
| sqrMagnitude | 得到向量长度的平方，运算速度比得到长度要快 |

|  |  |
| --- | --- |
| 方法 | 说明 |
| Cross | 向量叉乘 |
| Dot | 向量点乘 |
| Project | 计算向量在另一个向量上的投影 |
| Angle | 返回两个向量的夹角 |
| Distance | 返回两个向量的距离 |
| + | 用于向量相加 |
| - | 用于向量减法 |
| \* | 用于向量乘以标量，（会改变向量长度） |
| / | 用于向量除以标量，（会改变向量长度） |
| == | 判断两个向量是否相等 |
| != | 判断两个向量是否不相等 |

### 位置与向量的区别和联系

在数学中，点的坐标与向量有着严格的概念区分，如下图：



点A坐标为(1, 2)，点B坐标为(2, 3)，向量可以记为(1, 1)。

在数学符号系统中，点用大写字母表示，向量会以两个点名称再加上顶上箭头表示，或者用小写字母加顶上箭头表示，印刷品也会以黑体小写字母表示。总之在表示向量时，总是有特殊的符号和记法，能够比较清楚地区分向量和点。而在程序中，向量和坐标位置的记法就模糊了，给脚本编写的新手带来一些容易混乱的地方。

向量和坐标可以混合计算的原因是，它们是有联系的。A点的坐标，既是一个位置坐标，又是向量的值，原点记作O，坐标为(0, 0)，所以用A的坐标减去O的坐标就得到了向量。换句话说，任何一个点X的坐标都可以看成是从原点到X点的向量。向量和坐标显然是有联系的。而且要注意，由于我们说的向量都是自由向量，它的位置可以任意平移，它更多地表示了一个相对关系。

在脚本中，向量和位置都是以Vector3表示，那如何区分坐标和向量呢？仅从字面上其实无法区分，关键是代码的意图决定了Vector3的意义。

|  |
| --- |
| // 这是当前物体的坐标  Vector3 p1 = transform.position;  // 这是物体B的坐标  Vector3 p2 = gameObjectB.transform.position;  // 这是从当前物体到B的向量  Vector3 diff = p2 - p1;  // 获得了一个新的坐标，相当于B再远离当前物体一倍的位置  Vector3 p3= p2 + diff;  // 从物体C的位置出发，发生从A到B的位移，得到新的坐标  Vector3 p3 = gameObjectC.transform.position + diff;  // 调整向量的长度为一半  Vector3 diffHalf = diff \* 0.5f; |

如以上示例代码所示，坐标和向量不仅都用Vector3表示，而且它们之间还会发生运算，常用的情况列举如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **元素1** | **运算符** | **元素2** | **一般意义** |
| 坐标 | + | 坐标 | 一般无意义。特殊的比如(A+B)/2可以得到线段的中点 |
| 坐标 | + | 向量 | 从某个坐标位移，得到新的坐标。 |
| 向量 | + | 向量 | 向量叠加为一个新的向量，详见向量加法的几何意义 |
| 坐标 | - | 坐标 | 得到一个向量，被减数是终点 |
| 坐标 | - | 向量 | 得到新的坐标，相当于加上负向量 |
| 向量 | - | 向量 | 得到一个向量，不同情景下有不同意义，详见向量减法 |

向量的加减法看似非常基本，但是在游戏开发中非常基本且常用。例如以下问题：

1. 在玩家角色头顶1米处添加粒子。
2. 在玩家角色前方1米处生成炮塔。
3. 敌人瞄准玩家前方0.5米处（射击的提前量）。
4. 敌人一边朝玩家移动，一边躲开危险区域（AI躲避障碍，用到向量的叠加）。

前3个问题用坐标+向量偏移的方法即可解决。第4个问题代表了一类游戏AI问题，详见游戏AI开发有关的资料。

### Vector3用法实例

用两个例子来演示Vector3的使用方法：

#### 例子：获得两个物体的距离

在场景中创建一个立方体，在主摄像机上挂载以下脚本Distance.cs：

|  |
| --- |
| using UnityEngine;  public class Distance : MonoBehaviour {  Transform otherCube;  void Start()  {  otherCube = GameObject.Find(“Cube”).transform;  float dist = Vector3.Distance(otherCube.position, transform.position);  Debug.Log(“Distance:” + dist);  }  } |

执行后就可以在控制台窗口（Console）中看到立方体和摄像机之间的距离了。

#### 例子：缓动效果

接下来实现一种比较特别的缓动效果：物体一开始快速向终点移动，但是离终点越近速度就越慢，最终达到目标位置。

首先新建一个球体，在检视窗口的Transform组建菜单中选择Reset，重置坐标为(0, 0, 0)，然后挂载以下脚本MoveToTarget.cs：

|  |
| --- |
| using UnityEngine;  public class Distance : MonoBehaviour {  public Vector3 target = new Vector3(10, 0, 10);  void Update()  {  transform.position = Vector3.Lerp(transform.position, target, 0.3f);  }  } |

准备完毕后运行游戏，会发现物体快速向坐标(10, 0, 10)移动，距离越近移动越慢。

简单解释一下，Vector3.Lerp被称为差值函数，它的用法可以表示为

**插值结果 = Lerp(起点，终点，比例)**

起点、终点、差值结果都是Vector3类型，比例是一个float数字。插值的结果在起点和终点之间，根据比例而定。如果比例为0，则返回起点；如果比例为1，则返回终点；如果比例为0.5，则返回二者中间的位置。比例小于0等价于0，比例大于1等价于1。

本例子中，起点的位置取的是物体当前位置，比例固定为0.3，所以物体每一帧移动的距离都是剩余距离的30%，所以一开始移动很快，之后越来越慢，实现了一种非常特别的移动效果。

## 矩阵

和向量一样，矩阵也是3D数学的基础。矩阵就像一个表格，具有几行、几列。要想正确进行物体的位移、旋转、缩放变换，就必须要用到矩阵。

3D游戏中的向量一般只有3个维度，但矩阵要使用4x4矩阵，4x4矩阵是能够正常进行所有线性变换的最小矩阵。

介绍3D游戏中矩阵算法的问题，远远超过了本书介绍的范围，以下通过展示单独的平移、旋转、缩放矩阵，让读者对矩阵有一个直观的认识，消除陌生感。

1、平移矩阵：



向量v乘以T(p)，相当于让向量v的x、y、z分量分别变化Px、Py、Pz。

1. 旋转矩阵：



X(θ)矩阵可以让向量沿着X轴旋转θ角。

1. 缩放矩阵：



缩放矩阵可以对向量的各个分量进行缩放，向量v与S(q)相乘后，v的三个分量分别缩放Px、Py、Pz倍。

矩阵变换最强大的地方在于，它可以通过矩阵乘法进行组合，组合以后通过一个矩阵就可以表示连续一组变换操作，假设有三个矩阵S、R、T分别进行缩放、旋转、位移操作，三者相乘就得到了**M**矩阵。那么：

**vSRT = vM**

对**v**依次乘以**S**、**R**、**T**矩阵进行变换，得到的结果向量和**v**直接乘以**M**矩阵得到的结果是一致的。

虽然矩阵的作用很大，但是由于其使用有一定门槛，所以Unity已经封装了一些矩阵和变换函数，用户可以直接使用。而且旋转相关的问题还可以用四元数Quaternion来进行，进一步减少了直接操作矩阵的必要性。

## 齐次坐标

在3D数学中，齐次坐标就是将原本的三维向量(x, y, z)用四维向量(x, y, z, w)来表示。

引入齐次坐标有如下目的：

1. 更好的区分坐标点和向量。在三维空间中，(x, y, z)既可以表示一个点，也可以表示一个向量。如果采用齐次坐标，则可以使用(x, y, z, 1)来代表坐标点，而用(x, y, z, 0)来代表向量。在进行一些错误操作时，例如将两个坐标点相加的操作，会立即得到一个错误的结果（w值为2，既不是点也不是向量）。
2. 统一用矩阵乘法表示平移、旋转、缩放变换。3x3的矩阵可以用来表示旋转和缩放矩阵，但是无法表示平移，会带来很多问题。用4x4的矩阵就可以统一三种线性变换。

齐次坐标是计算机图形学中一个非常重要的概念，但是Unity中很少会考虑齐次坐标的问题。这个概念只在某些编写Shader的时候用到，游戏逻辑中大部分情况下还是使用Vector3。Unity引擎内部会使用齐次坐标的概念，但是对用户是隐藏的。

## 四元数

### 概念

四元数包含一个标量分量和一个三维向量分量，四元数Q可以记作：



在3D数学中使用单位四元数来表示旋转，对于三维空间中旋转轴为n，旋转角度为α的旋转，如果用四元数表示，四个分量分别为：



其中三个余弦的角度，分别为旋转轴的x、y、z分量。

从上面的描述中可以看到四元数表示的旋转并不直观。在3D数学中，旋转还可以用欧拉角和矩阵表示，但是每一种表示方法都有其各自的优缺点，下表对这三种方式进行了对比：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 欧拉角 | 矩阵 | 四元数 |
| 旋转一个位置点 | 不支持 | 支持 | 不支持 |
| 增量旋转 | 不支持 | 支持，运算量大 | 支持，运算量小 |
| 平滑差值 | 支持（潜在问题） | 基本不支持 | 支持 |
| 内存占用 | 3个浮点数 | 16个数值 | 4个浮点数 |
| 表达式是否唯一 | 无数种组合 | 唯一 | 互为负的两种 |
| 潜在问题 | 万向锁 | 矩阵蠕变 | 误差累计 |

由于三种表示旋转的方法都有各自的优势和缺点，所以实际中会根据具体需求进行考虑。另一方面，正因为旋转的表示方法非常重要，所以应当尽可能在引擎层面进行统一，这样才能尽可能减少开发游戏时的问题。

本书在前面的章节中介绍过，Unity内部旋转是用四元数，即结构体Quaternion表示的，但是在界面上很多地方会用到更直观的欧拉角。

### 结构体Quaternion简介

下面详细介绍Quaternion结构体的属性和方法。

|  |  |
| --- | --- |
| x | 四元数的x分量，不应直接修改 |
| y | 四元数的y分量，不应直接修改 |
| z | 四元数的z分量，不应直接修改 |
| w | 四元数的w分量，不应直接修改 |
| this[int index] | 允许通过下标运算符访问x, y, z, w分量。例如[1]可以访问y |
| eulerAngles | 获得对应的欧拉角 |
| identity | 获得无旋转的四元数 |

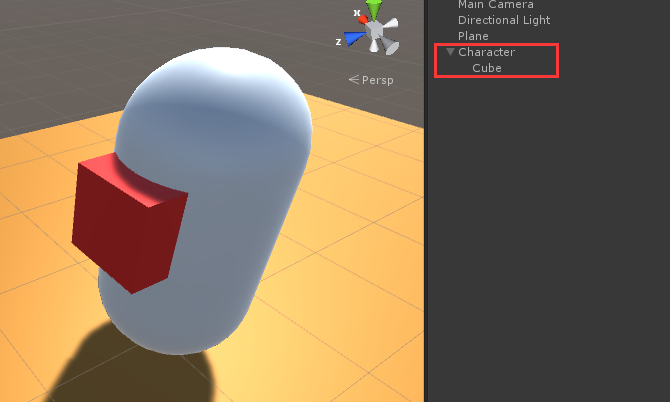
|  |  |
| --- | --- |
| ToAngleAxis | 将旋转转换为一个轴和一个角度的形式 |
| SetFromToRotation | 与FromToRotation类似，但是直接修改当前四元数对象 |
| SetLookRotation | 与LookRotation类似，但是直接修改当前四元数对象 |
| \* | 四元数相乘，代表依次旋转的操作 |
| == | 判断四元数是否相等 |
| != | 判断四元数是否不相等 |
| Dot | 两个旋转点乘 |
| AngleAxis | 根据一个轴和一个角度获得一个四元数 |
| FromToRotation | 获得一个四元数，代表从from到to向量的旋转 |
| LookRotation | 给定前方和上方向量，获得一个旋转 |
| Slerp | 插值，根据比例在两个四元数之间进行球面插值 |
| Lerp | 插值，根据比例在两个四元数之间插值并将结果规范化 |
| RotateTowards | 将旋转from变到旋转to |
| Inverse | 返回旋转的逆 |
| Angle | 返回a和b两个旋转之间的夹角 |
| Euler | 转换为对应的欧拉角 |

### 四元数操作示例

在游戏对象的变换组件中，transform.rotation为对象在全局坐标系下的旋转，transform.localRotation是对象在父对象坐标系下的旋转，两个变量的类型均为四元数。因此只要通过改变transform.rotation或者transform.localRotation就可以设置游戏对象的旋转。

#### **示例——输入控制旋转**

为了更好地解释Unity中四元数的使用方法，作者特别设计了一个有趣的控制角色旋转的例子。如图：



场景中有一个胶囊体外形的角色，具有一个红色立方体表示它的正面。我们的目标是通过键盘输入能够任意控制角色的旋转，而且我们准备一步一步加入更多功能，体会四元数的强大与灵活。

首先在场景中搭建上图中的角色，然后在角色物体上挂载如下脚本SimpleRotate：

|  |
| --- |
| using UnityEngine;  public class SimpleRotate : MonoBehaviour {  void Update () {  float v = Input.GetAxis("Vertical");  float h = Input.GetAxis("Horizontal");  // 将横向输入转化为左右旋转，将纵向输入转化为俯仰旋转。得到一个很小的旋转四元数  Quaternion smallRotate = Quaternion.Euler(v, h, 0);  // 将这个小的旋转叠加到当前旋转位置上  transform.rotation = smallRotate \* transform.rotation;  }  } |

以上代码将键盘输入，转化为一个小的旋转动作，然后使用四元数相乘的方法，把小的旋转应用在当前的transform.rotation上面，每帧调用一次，形成了连续的旋转动画。

运行游戏，使用键盘的A和D键可以控制物体的左右旋转，使用W键和S键可以控制物体的俯仰。

在有俯仰角度以后，再使用A和D键，会发现左右旋转依然是以角色本身为准的。这点是四元数运算的一个特性，可以通过测试进行体会。

#### 示例——全局坐标系旋转

可能有读者会好奇：那么如果必须让物体以世界坐标系为准进行左右旋转呢？这个问题对初学者来说有点复杂，但是如果采用四元数提供的“轴、角”方式，也很容易解决。代码修改如下：

|  |
| --- |
| using UnityEngine;  public class SimpleRotate : MonoBehaviour {  void Update () {  float v = Input.GetAxis("Vertical");  float h = Input.GetAxis("Horizontal");  // 将纵向输入转化为第一个旋转  Quaternion smallRotate = Quaternion.Euler(v, 0, 0);  // 用轴、角方式构造横向的旋转  // 全局坐标系中的Vector3.up，其实就是Y轴的方向  Quaternion smallRotate2 = Quaternion.AngleAxis(h, Vector3.up);  // 对物体连续应用两个旋转  transform.rotation = smallRotate2 \* smallRotate \* transform.rotation;  }  } |

测试即可发现，现在当角色倒下的时候，左右旋转就是以全局坐标为准了。

#### 示例——旋转到指定位置

很多时候，我们需要让角色快速旋转到指定位置，比如当敌人角色发现主角的时候，无论他的当前朝向是哪里，都会快速转向主角。但是毕竟转身是有速度限制的，如果瞬间转向，效果就会很虚假。以下的代码能够让角色以插值的方式转向(0, 0, 0)的位置：

|  |
| --- |
| using UnityEngine;  public class SimpleRotate : MonoBehaviour {  void Update () {  float v = Input.GetAxis("Vertical");  float h = Input.GetAxis("Horizontal");  // 将横向输入转化为左右旋转，将纵向输入转化为俯仰旋转。得到一个很小的旋转四元数  Quaternion smallRotate = Quaternion.Euler(v, h, 0);  // 将这个小的旋转叠加到当前旋转位置上  transform.rotation = smallRotate \* transform.rotation;  if (Input.GetButton("Jump"))  {  // 目标位置是没有旋转的位置，相当于欧拉角(0, 0, 0)  Quaternion target = Quaternion.identity;  // 插值旋转，制造先快后慢的效果  Quaternion temp = Quaternion.Slerp(transform.rotation, target, 0.1f);  transform.rotation = temp;  }  }  } |

这时如果按住空格键（默认的Jump跳跃键），物体就会向着零位旋转。

这里采用插值的效果是物体旋转先慢后快，这是Slerp函数的常用方法，每次都在物体当前位置和目标位置之间取值，那么当距离越短，取到的差值就越小。

能否匀速转动呢？答案是可以的，只需要将temp的计算改为如下代码即可：

|  |
| --- |
| // 匀速旋转，每次最大转动1度  Quaternion temp = Quaternion.RotateTowards(transform.rotation, target, 1.0f); |

RotateTowards方法也同样是接收了两个起点、终点四元数，和Slerp几乎一样，不同的是，RotateTorwards的含义是生成一个从起点到终点的旋转，但是角度大小不超过第三个参数的范围。这个函数用于匀速转动非常方便。

#### 示例——角色快速起身

四元数的操作非常灵活，一旦熟悉了也会非常有趣。考虑这样的需求：由于各种原因，角色倒地了，但是角色需要快速爬起来，也就是回到直立的状态。应当如何旋转呢？

下面给出了示例的代码：

|  |
| --- |
| using UnityEngine;  public class SimpleRotate : MonoBehaviour {  void Update () {  float v = Input.GetAxis("Vertical");  float h = Input.GetAxis("Horizontal");  // 将横向输入转化为左右旋转，将纵向输入转化为俯仰旋转。得到一个很小的旋转四元数  Quaternion smallRotate = Quaternion.Euler(v, h, 0);  // 将这个小的旋转叠加到当前旋转位置上  transform.rotation = smallRotate \* transform.rotation;  if (Input.GetButton("Jump"))  {  // 同前文，略 ……  }  if (Input.GetButton("Fire1"))  {  // 要让物体直立，就是要让物体的上方与世界上方一致  // 也就是 transform.up 和 Vector3.up 一致  Quaternion target = Quaternion.FromToRotation(transform.up, Vector3.up);  target = target \* transform.rotation;  // 匀速旋转，每次最大转动1度  Quaternion temp = Quaternion.RotateTowards(transform.rotation, target, 1.0f);  transform.rotation = temp;  }  }  } |

简单来说，要让角色起身，就是要考虑让角色自身的上方，和世界的上方一致。那么就要创建一个从角色上方到世界上方的四元数。这个四元数是一个旋转的动作，不是目标旋转位置。将它乘以当前旋转位置，就得到了目标旋转位置。

然后再使用前面的RotateTowards方法匀速转过去即可。

## 本章小结

本章侧重最基本的坐标系、向量等概念的讲解，而且在最后附上了几个简单但又有代表性的例子，希望能给读者打一个良好的数学基础。

每当谈论到数学，总是有些人感觉到非常有趣，而另一些人觉得非常头疼。好消息是——越是基础的东西在日常开发中用到的越多。比如向量的加减法、归一化就比向量的点乘、叉乘用到的多得多。

所以无论你是否喜欢学习游戏开发中的数学，能掌握并灵活运用最基本的概念总是最重要的，而且也不是难事。