

三维几何学基础

MILO YIP

2015/9/21



大纲

1. 三维坐标系
2. 点与矢量
3. 矩阵与几何变换
4. 四元数与三维旋转
5. 参考资料

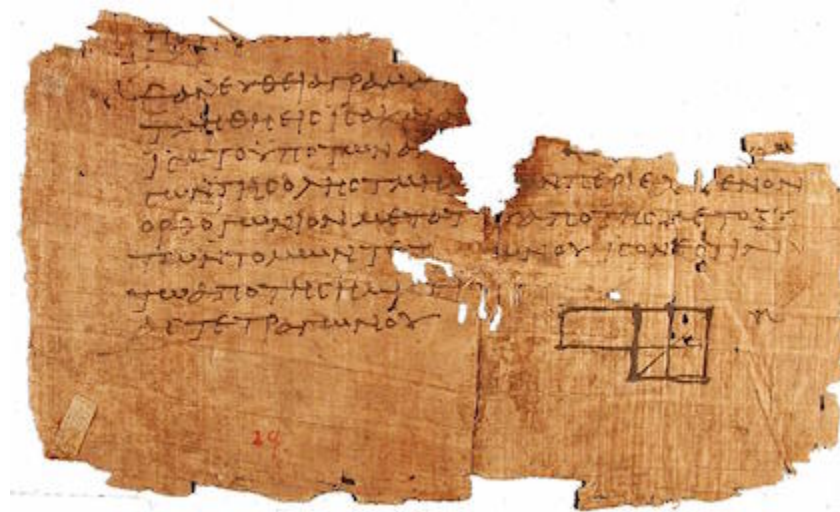
游戏中的应用

- 任何与游戏中三维空间相关的地方
 - 游戏逻辑 (gameplay logic)
 - 计算几何 (computational geometry)
 - 计算机图形 (computer graphics)
 - 计算机动画 (computer animation)
 - 计算物理 (computational physics)
 - 声音与音乐计算 (sound and music computing)
 - 人工智能 (artificial intelligence)

1. 三维坐标系

几何学 GEOMETRY

古希腊数学家欧几里得（Euclid）约于300BC著成13卷的
《Elements》



1607年意大利传教士利玛窦（Matteo Ricci）和明朝科学家
徐光启（Paul Xu）合译了前6卷，定名为《几何原本》

欧几里得几何

- 从欧几里得几何，通过公理推导出
 - 等腰三角形底角相等
 - 三角形内角和是 180°
 - 勾股定理
 -

解析几何

- 解析几何 ([Analytic geometry](#)) 由法国哲学家笛卡儿 (Descartes) 于1637年著的《方法论》开创
 - 使用坐标系
 - 把几何问题转换为代数问题

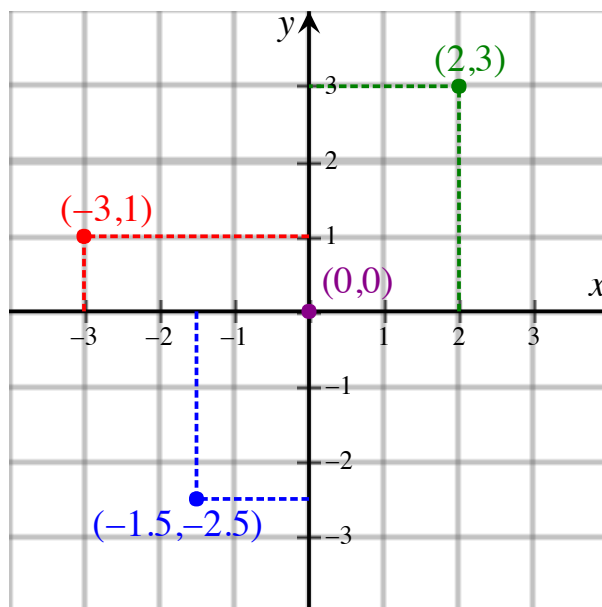


“Cogito ergo sum (I think, therefore I am / 我思故我在).”

笛卡儿坐标系

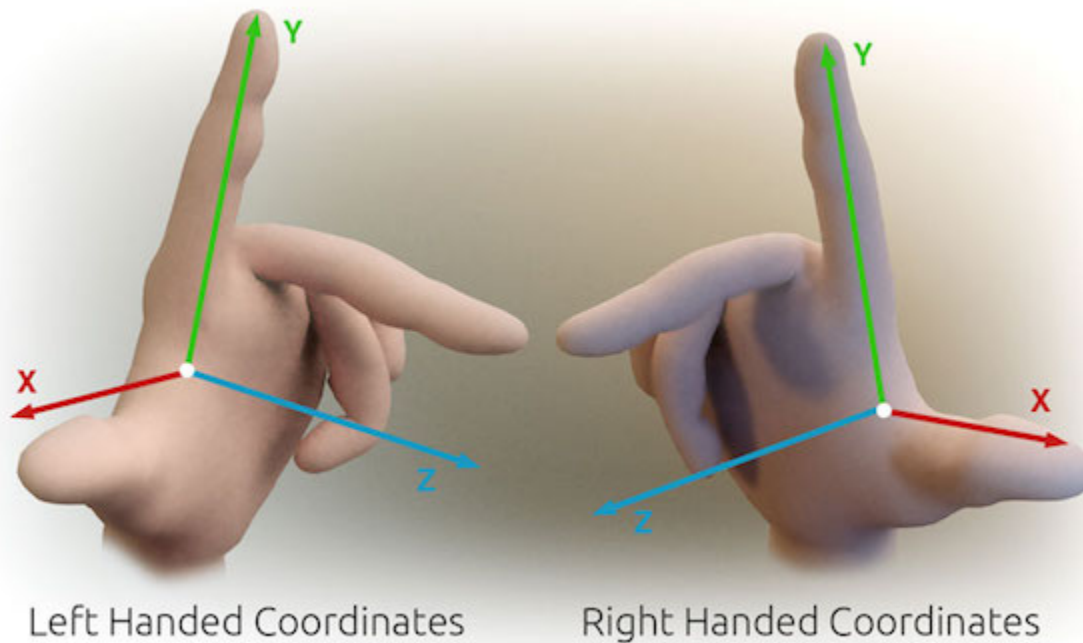
- Cartesian coordinate system
- 在 n 维空间中，用 n 个数值表示一点
 - 一维：用 x 表示数线上的一个点
 - 二维：用 (x, y) 表示平面上的一个点
 - 三维：用 (x, y, z) 表示立体空间中的一个点
- 原点 (origin) : 0 、 $(0, 0)$ 、 $(0, 0, 0)$
- 轴 (axis) : n 个互相垂直的直线 \rightarrow 直角坐标系

二维笛卡尔坐标系



- 扩展至三维时， z 轴应该指向前，还是后？

三维左手 / 右手坐标系

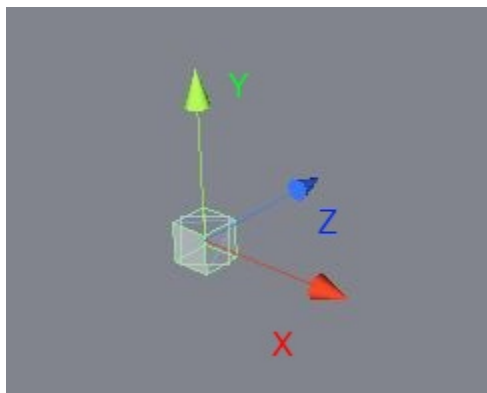


CC

- 不影响数学运算
- 把任意一个轴反转就能互换
- Unity 的世界坐标是左手的

轴与方向的映射

- x 、 y 、 z 是数学中，轴的常用名字
- 需要映射至现实的概念
- 例：Unity 中的世界坐标 $+x$ 右、 $+y$ 上、 $+z$ 前

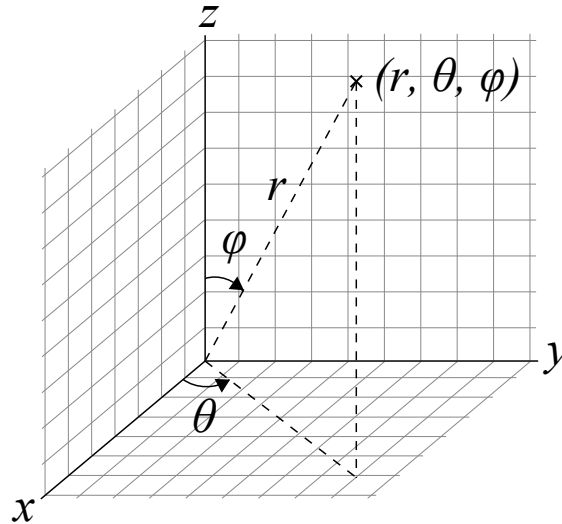


常用软件的世界坐标

软件	利手	+x	+y	+z
Unity	左手	右	上	前
Unreal	左手	前	右	上
3ds Max	右手	右	前	上
Maya	右手	右	上或前	后或上

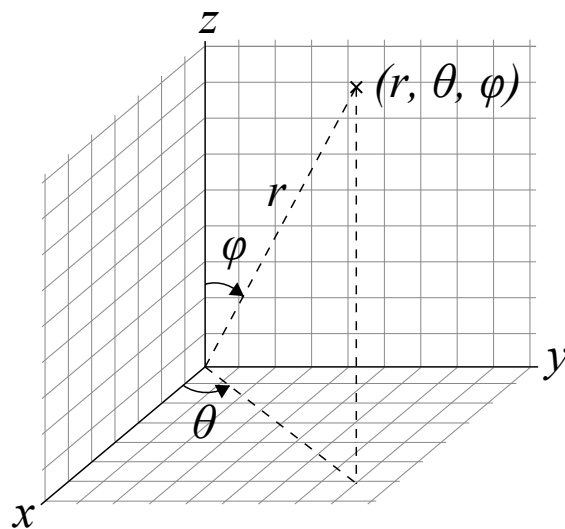
球坐标系

- spherical coordinate system
- 二维极坐标 (r, θ) 扩展至三维
- 增加一个角度 φ （英文记为 phi）
- 有不同的约定，数学上常用：



- http://mathinsight.org/spherical_coordinates

球坐标 \rightarrow 笛卡尔坐标

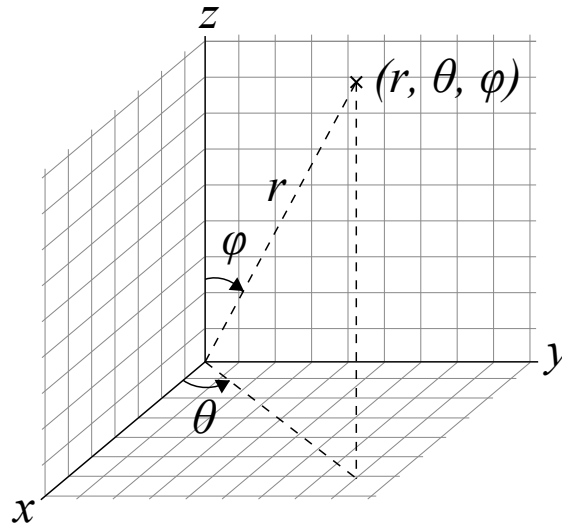


$$x = r \sin \varphi \cos \theta$$

$$y = r \sin \varphi \sin \theta$$

$$z = r \cos \varphi$$

笛卡尔坐标 \rightarrow 球坐标



$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\varphi = \cos^{-1} \frac{z}{r}$$

$$\theta = \text{atan2}(y, x)$$

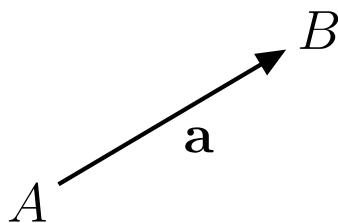
练习：三维坐标系

1. 设一个坐标映射中， $+x$ 为东方， $+z$ 为北方， $+y$ 为上方，1个单位为1米。一只小鸟从 $(-200, 0, 400)$ 起飞，往南直飞3公里，途中爬升了300米，然后往西直飞4公里。
 - 它现在的坐标是？
 - 它在地图上与出发地的距离是？
2. 写出一个绕 y 轴螺旋移动点在时间 t 的笛卡尔坐标。该点的初始位置为 $(r, 0, 0)$ ， xz 平面上的圆形路径半径为 r ， xz 平面上的角速度为 ω ， y 轴上的移动速度为 v 。

2. 点与矢量

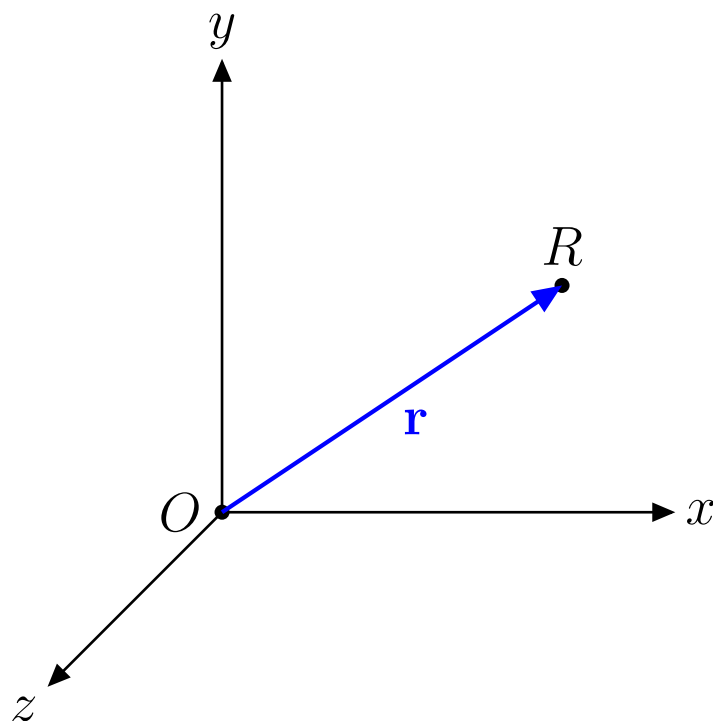
矢量

- 矢量 (vector) 又称作向量
- 具有方向和大小
- 可写作 \mathbf{a} 、 \vec{a} 或 \overrightarrow{AB}



位置矢量

- 点可表示为位置矢量（position vector） / 矢径（radius vector）



游戏常见的标量／矢量

标量

矢量

时间、质量

面积、体积

长度、距离

位置、位移

速率、加速率

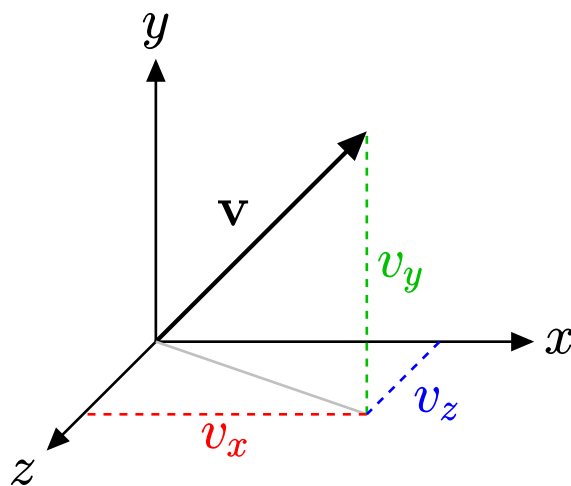
速度、加速度

力

颜色

矢量分解

- 三维矢量可分解成 3 个分量

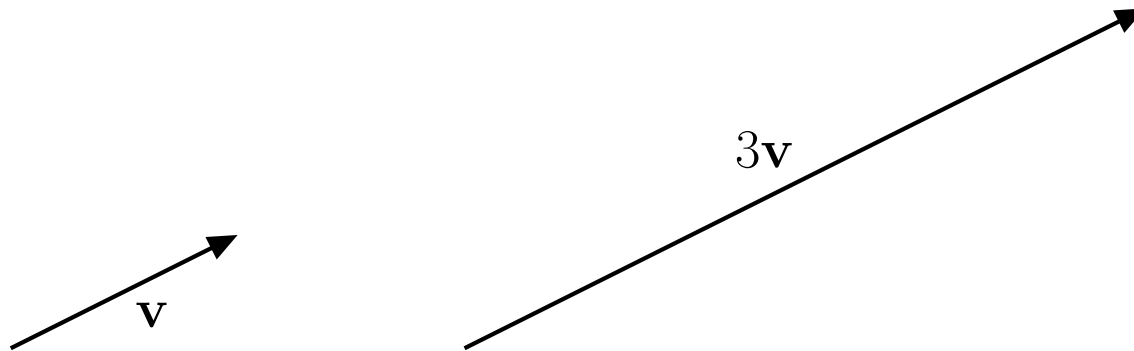


$$\mathbf{v} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}$$

- Unity : `Vector3 v = new Vector3(x, y, z)`

矢量与标量的乘法

- 矢量乘以标量（scalar）形成等比缩放（uniform scaling）

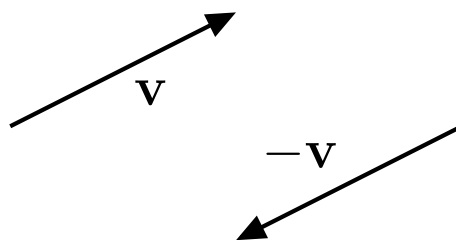


$$\begin{aligned}\mathbf{v}' &= s\mathbf{v} \\ &= \begin{bmatrix} sv_x & sv_y & sv_z \end{bmatrix}\end{aligned}$$

- Unity : $\mathbf{v} * 2.0f$ 或 $2.0f * \mathbf{v}$

反方向矢量

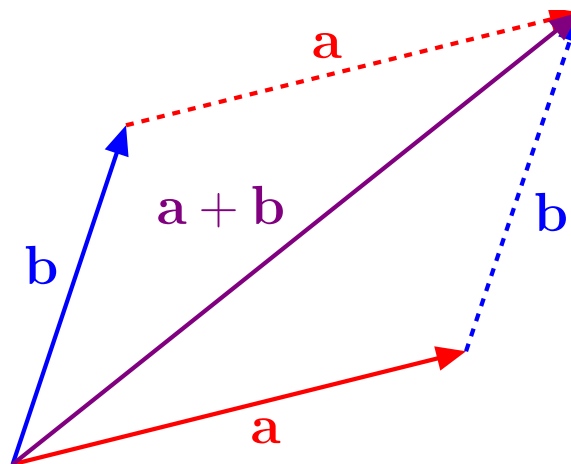
- 以 -1 缩放等于把方向反转



$$\begin{aligned}\mathbf{v}' &= -\mathbf{v} \\ &= \begin{bmatrix} -v_x & -v_y & -v_z \end{bmatrix}\end{aligned}$$

- Unity : $-\mathbf{v}$

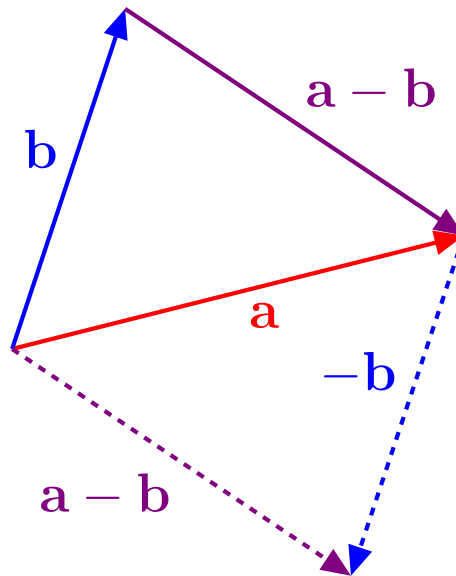
矢量加法



$$\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_x + b_x & a_y + b_y & a_z + b_z \end{bmatrix}$$

- Unity : $\mathbf{a} + \mathbf{b}$

矢量减法



$$\begin{aligned}\mathbf{a} - \mathbf{b} &= \mathbf{a} + (-\mathbf{b}) \\ &= \begin{bmatrix} a_x - b_x & a_y - b_y & a_z - b_z \end{bmatrix}\end{aligned}$$

- Unity : $\mathbf{a} - \mathbf{b}$

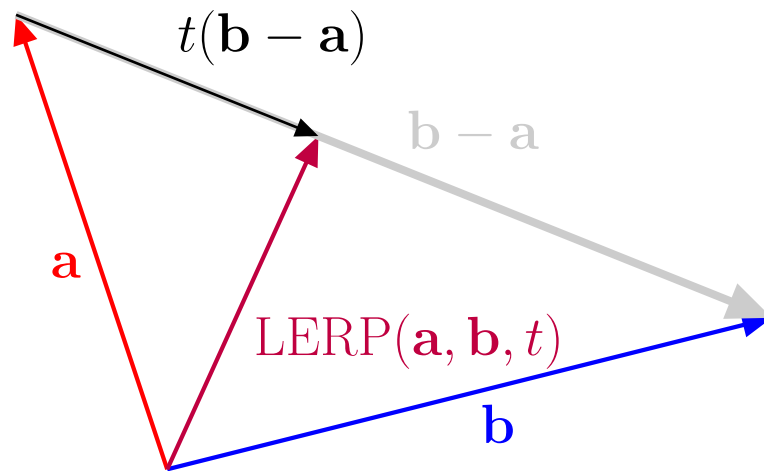
点和矢量的加减

运算	结果	意义
矢量 + 矢量	矢量	叠加
矢量 - 矢量	矢量	叠加反向矢量
点 + 矢量	点	把点平移
点 - 矢量	点	把点平移
点 - 点	矢量	两点间的距离 / 方向
点 + 点	?	无几何意义

- 大部分引擎不区分点和矢量，但我们要了解个别对象表示点还是矢量

用例：线性插值

- 线性插值 (linear interpolation, LERP)
- $t \in [0, 1]$

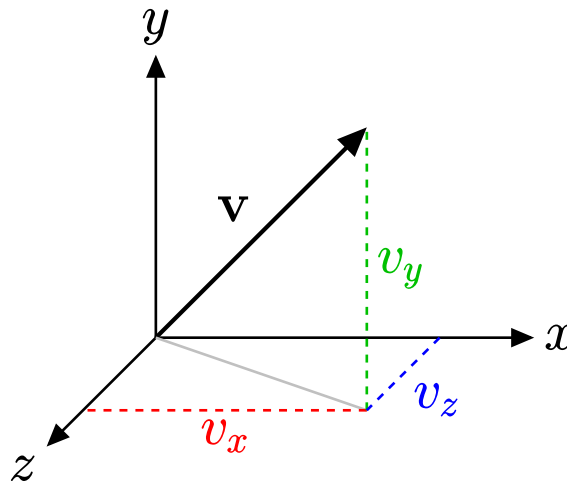


$$\begin{aligned}\text{LERP}(\mathbf{a}, \mathbf{b}, t) &= \mathbf{a} + t(\mathbf{b} - \mathbf{a}) \\ &= (1 - t)\mathbf{a} + t\mathbf{b}\end{aligned}$$

- Unity : `Vector3.Lerp(a, b, t)`

模

- 模 (magnitude) 即矢量的大小 / 长度
- 利用勾股定理 / 畢氏定理 (Pythagorean theorem)



$$\|\mathbf{v}\| = \sqrt{\left(\sqrt{v_x^2 + v_z^2}\right)^2 + v_y^2} = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

- Unity : `v.magnitude`

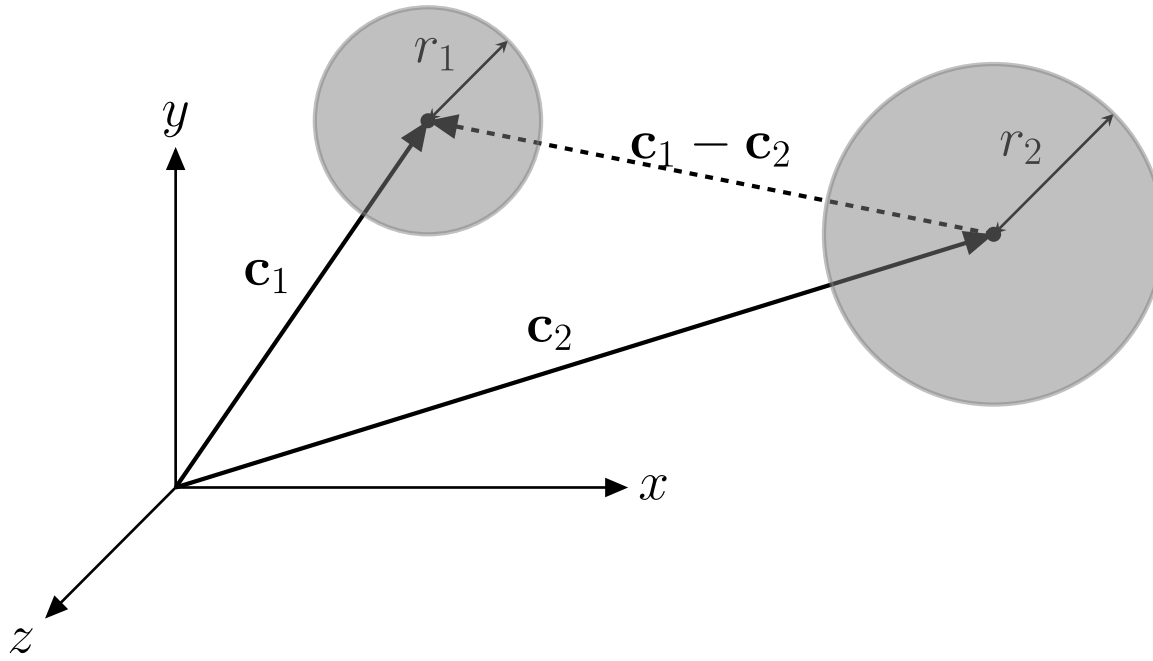
归一化和单位矢量

- 归一化 (normalization) 求相同方向但模为1的矢量

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \frac{1}{\sqrt{v_x^2 + v_y^2 + v_z^2}} \mathbf{v}$$

- 模为1的矢量称为单位矢量 (unit vector) , 记为 $\hat{\mathbf{v}}$
- 注意当 $\mathbf{v} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$, $\hat{\mathbf{v}}$ 无定义
- Unity : `v.normalized` 或 `v.Normalize()`

用例：球体相交测试



- 若球心距离小于半径之和，两球体便相交

$$\|\mathbf{c}_1 - \mathbf{c}_2\| \leq r_1 + r_2$$

- Unity : `(c1 - c2).magnitude <= r1 + r2`

优化

- 计算模需要开方运算
- 但由于模必然为非负数，可以把不等式左右侧平方
- 设 $\mathbf{v} = \mathbf{c}_1 - \mathbf{c}_2$

$$\|\mathbf{v}\| \leq r_1 + r_2$$

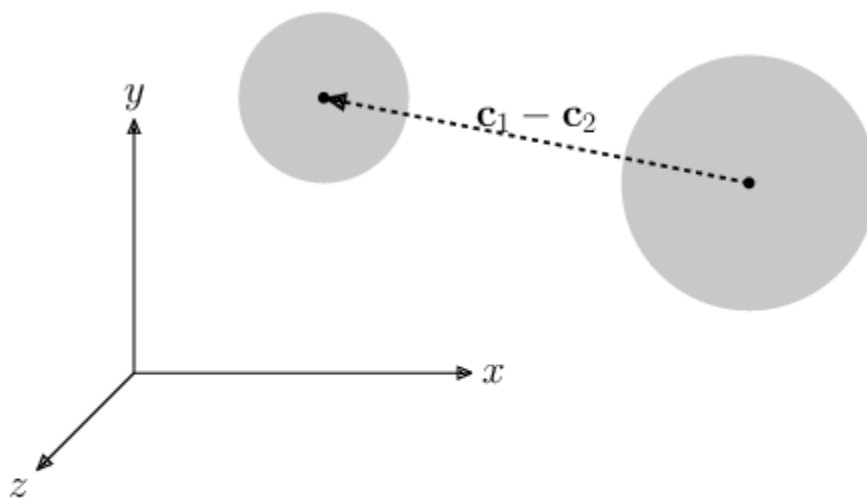
$$\|\mathbf{v}\|^2 \leq (r_1 + r_2)^2$$

$$v_x^2 + v_y^2 + v_z^2 \leq (r_1 + r_2)^2$$

- Unity：左侧用 `Vector3.sqrMagnitude`

有趣发现

测试时，只考虑 $\mathbf{c}_1 - \mathbf{c}_2$ 和 $r_1 + r_2$



两个球体相交 \Leftrightarrow 一个放大并平移后的球体与原点相交

点积

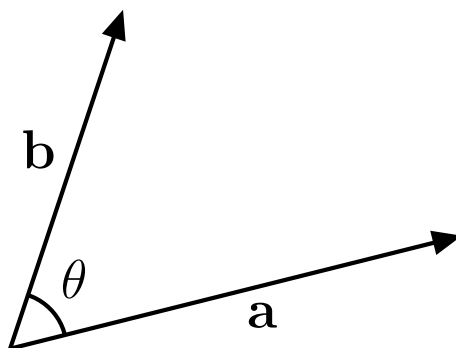
- 点积 (dot product) / 内积 (inner product)
- 和矢量夹角相关的标量
- n 维适用

$$\begin{aligned}\mathbf{a} \cdot \mathbf{b} &= \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta \\ &= a_x b_x + a_y b_y + a_z b_z\end{aligned}$$

- Unity : `Vector3.Dot(a, b)`
- 模可以用点积来定义

$$\|\mathbf{v}\|^2 = \mathbf{v} \cdot \mathbf{v}$$

用例：求夹角

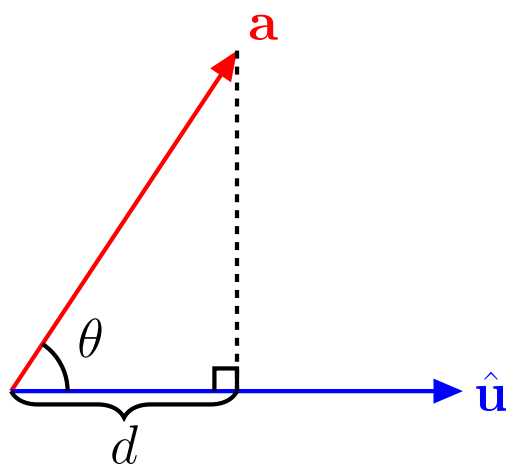


$$\begin{aligned}\theta &= \cos^{-1} \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \\ &= \cos^{-1} (\hat{\mathbf{a}} \cdot \hat{\mathbf{b}})\end{aligned}$$

- 夹角必然为正数： $0 \leq \theta \leq \pi$
- 也可用于判断两矢量是否平行或垂直

用例：求方向上的投影

- 求 \mathbf{a} 在 $\hat{\mathbf{u}}$ 方向上的投影

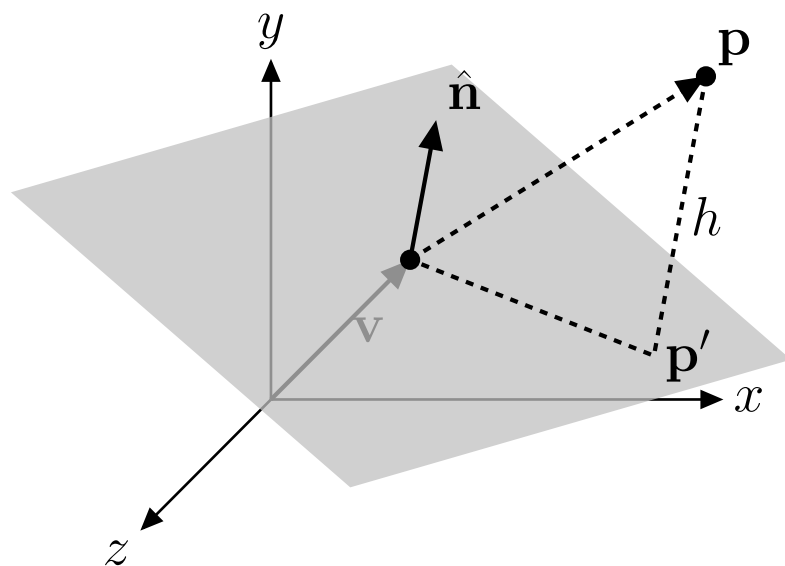


$$d = \|\mathbf{a}\| \cos \theta$$

$$d = \|\mathbf{a}\| \|\hat{\mathbf{u}}\| \cos \theta$$

$$= \mathbf{a} \cdot \hat{\mathbf{u}}$$

用例：求平面上的投影

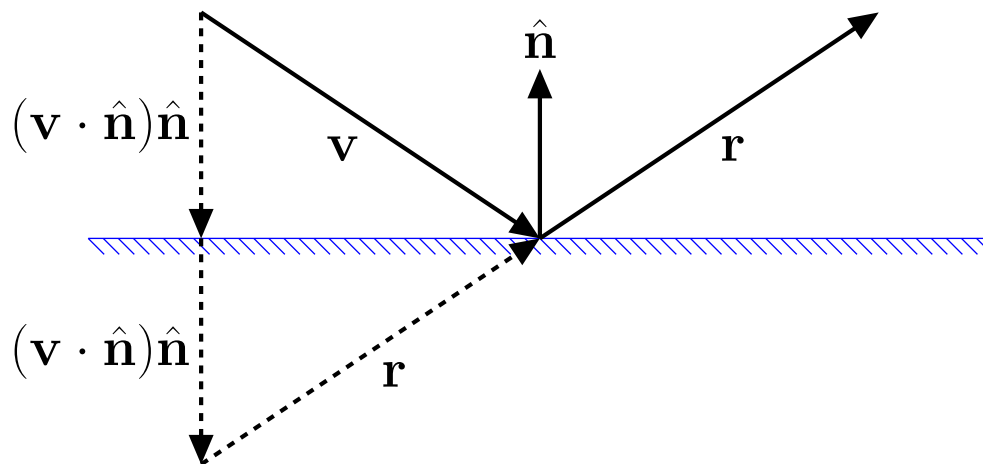


$$h = (\mathbf{p} - \mathbf{v}) \cdot \hat{\mathbf{n}}$$

$$\mathbf{p}' = \mathbf{p} - h\hat{\mathbf{n}}$$

用例：反射矢量

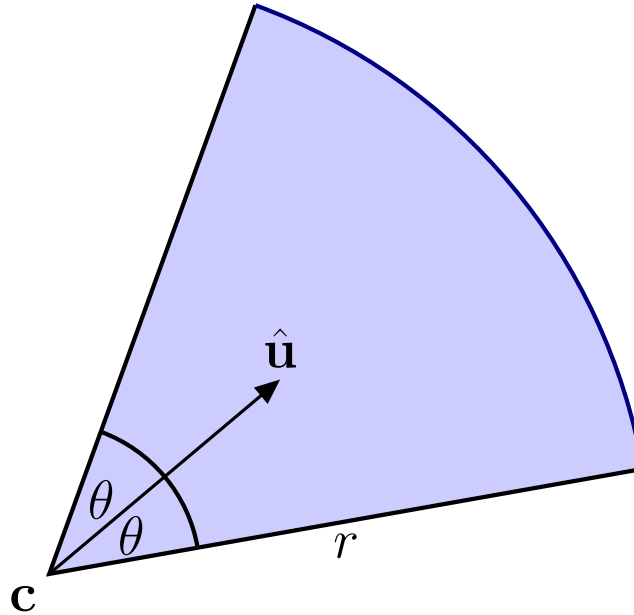
- 把矢量 \mathbf{v} 于一个平面上反射
- 平面的法矢量（normal vector）为 $\hat{\mathbf{n}}$



$$\mathbf{r} = \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

- Unity : [Vector3.Reflect](#)

用例：点于扇形内测试



$$\cos^{-1} \left(\frac{\mathbf{p} - \mathbf{c}}{\|\mathbf{p} - \mathbf{c}\|} \cdot \hat{\mathbf{u}} \right) < \theta \quad \text{及} \quad \|\mathbf{p} - \mathbf{c}\| < r$$

优化：检测点是否在扇形之内

叉积

- 叉积 (cross product) 是垂直于两个矢量的矢量

$$\mathbf{n} = \mathbf{a} \times \mathbf{b}$$

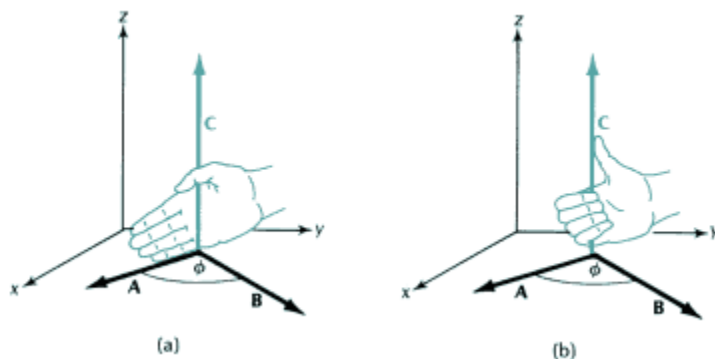
$$= \|\mathbf{a}\| \|\mathbf{b}\| \sin(\theta) \hat{\mathbf{n}}$$

$$= \begin{bmatrix} a_y b_z - a_z b_y & a_z b_x - a_x b_z & a_x b_y - a_y b_x \end{bmatrix}$$

- 有别于点积，叉积一般只于三维空间中有定义
- Unity : `Vector3.Cross(a, b)`

叉积的左右手法则

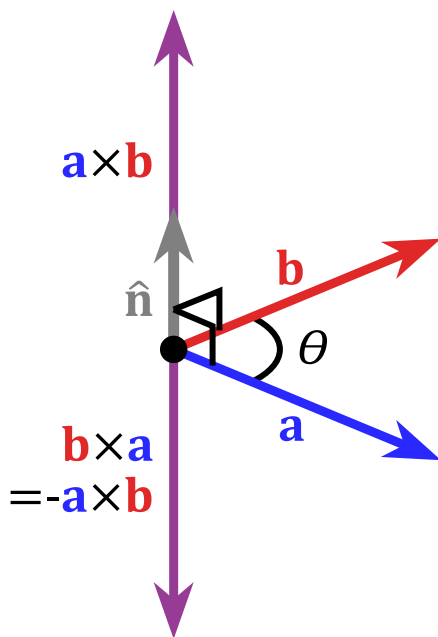
- 在右手坐标系，用右手法则定义积方向



- 在左手坐标系，则使用左手法则

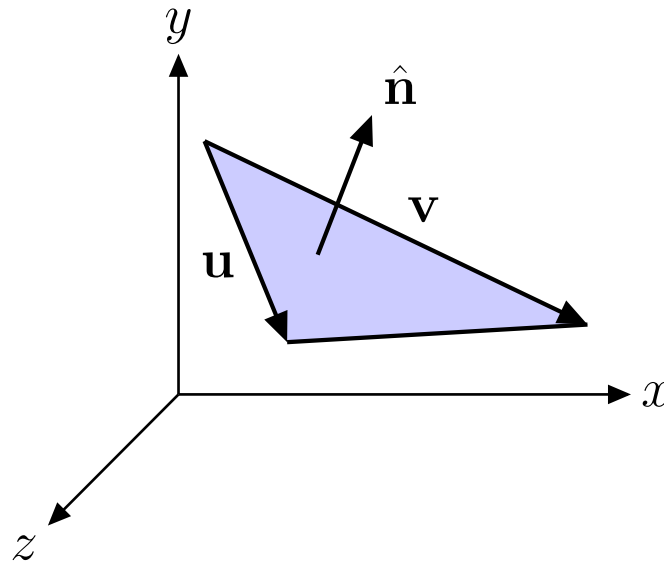
叉积的反交换律

- $\mathbf{a} \times \mathbf{b}$ 和 $\mathbf{b} \times \mathbf{a}$ 的结果是不同的



- 实际上，它满足反交换律 $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$

用例：求三角形法线



$$\hat{\mathbf{n}} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|}$$

- 因反交换律，交换 \mathbf{u} 和 \mathbf{v} 会反转法线
- 需要顶点的缠绕顺序（winding order）来定义正反面

阿达马积

- 阿达马积 (Hadamard product) 是各分量相乘的矢量积
- 数学文献中常记作 $\mathbf{a} \circ \mathbf{b}$ ，在《RTR》中记作：

$$\mathbf{a} \otimes \mathbf{b} = \begin{bmatrix} a_x b_x & a_y b_y & a_z b_z \end{bmatrix}$$

- 常用于颜色的运算，把RGB颜色当作三维矢量，如：

$$\text{出射光颜色} = \text{入射光颜色} \otimes \text{反照率}$$

- 着色器语言中，通常 $\mathbf{a} * \mathbf{b}$ 就代表阿达马积

总结：点和矢量

- 矢量缩放、加法、减法
- 模、归一化、单位矢量
- 点积、叉积、阿达马积
- LERP、投影、求法线

矢量运算一覽

运算	记法	结果	维度	交换	反交换	结合
乘以标量	$s\mathbf{v}$	矢量	n 维	-	-	-
加法	$\mathbf{a} + \mathbf{b}$	矢量	n 维	✓	✗	✓
减法	$\mathbf{a} - \mathbf{b}$	矢量	n 维	✗	✓	✗
点积	$\mathbf{a} \cdot \mathbf{b}$	标量	n 维	✓	✗	-
叉积	$\mathbf{a} \times \mathbf{b}$	矢量	3维	✗	✓	✗
阿达马积	$\mathbf{a} \otimes \mathbf{b}$	矢量	n 维	✓	✗	✓

练习：点和矢量

1. 写出检测矢量 \mathbf{a} 、 \mathbf{b} 是否接近垂直（夹角在 1° 以内）的不等式。
2. 设两个球体($i = \{A, B\}$)的球心初始位置为 \mathbf{c}_i ，以匀速 \mathbf{v}_i 移动。
 - 写出球心在时间 t 的位置函数 $\mathbf{x}_i(t)$ ；
 - 写出判断两球体在时间 t 相交的不等式；
 - 写出判断两移动球体碰撞的不等式。如两移动球体将会碰撞，求碰撞时间 t_0 。
3. 给定一个三维单位矢量 $\hat{\mathbf{u}}$ （3个分量皆不为0），生成两个垂直于 $\hat{\mathbf{u}}$ 的单位矢量 $\hat{\mathbf{v}}$ 和 $\hat{\mathbf{w}}$ ，且 $\hat{\mathbf{v}}$ 垂直于 $\hat{\mathbf{w}}$ 。

3. 矩阵与几何变换

矩阵与图形学

- 1963年 Timothy E. Johnson 开发了史上首个三维计算机软件 Sketchpad III



- 当中使用到矩阵变换，他把此工作归功于一名 PhD 学生 Larry E. Roberts¹

矩阵

- 矩阵（matrix）是 $m \times n$ 个元素所组成的长方形数组：

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

- 矩阵及其乘法最常用作表示线性变换（linear transformation）
- Unity的 4×4 矩阵：`Matrix4x4`

矩阵乘法

- 把 $n \times \underline{m}$ 的矩阵 \mathbf{A} ，乘以 $\underline{m} \times p$ 的矩阵 \mathbf{B}
- 得 $n \times p$ 的矩阵 \mathbf{C}

$$C_{ij} = \sum_{k=1}^m A_{ik} B_{kj}$$

- 满足结合律 $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$
- 一般情况下不满足交换律 $\mathbf{AB} \neq \mathbf{BA}$
- 直接实现 $O(n^3)$
- Coppersmith–Winograd算法 $O(n^{2.376})$ ，但 n 需极大
- Unity : $a * b$

矩阵乘法例子

$$\begin{array}{c} 4 \times 2 \text{ matrix} \\ \begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix} \end{array} \begin{array}{c} 2 \times 3 \text{ matrix} \\ \begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix} \end{array} = \begin{array}{c} 4 \times 3 \text{ matrix} \\ \begin{bmatrix} \cdot & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ \cdot & \cdot & \cdot \\ \cdot & a_{31}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \\ \cdot & \cdot & \cdot \end{bmatrix} \end{array}$$

单位矩阵

- 单位矩阵 (identity matrix) 是方形的 ($n \times n$) , 记作 \mathbf{I}_n
$$\mathbf{I}_1 = \begin{bmatrix} 1 \end{bmatrix}, \mathbf{I}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \dots, \mathbf{I}_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$
- 是矩阵乘法的单位元, 对于 $m \times n$ 的 \mathbf{A} :
$$\mathbf{A}\mathbf{I}_n = \mathbf{I}_m\mathbf{A} = \mathbf{A}$$
- Unity : `Matrix44.identity`

逆矩阵

- 一些方形矩阵能求出它在乘法上的逆
- 这称为逆矩阵 (inverse matrix)
- 记作 \mathbf{A}^{-1}

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

$$(\mathbf{A}^{-1})^{-1} = \mathbf{A}$$

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

- 求逆是一个比较耗时的运算
- Unity : `a.inverse`

转置

- 把一个矩阵转置 (transpose) ，即把行和列互换
- 记作 \mathbf{A}^T
- 若 $\mathbf{B} = \mathbf{A}^T$ 则 $b_{ji} = a_{ij}$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & 7 \end{bmatrix}^T = \begin{bmatrix} 1 & 0 \\ 2 & -6 \\ 3 & 7 \end{bmatrix}$$

- $(\mathbf{A}^T)^T = \mathbf{A}$, $(\mathbf{A}^T)^{-1} = (\mathbf{A}^{-1})^T$, $(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$
- Unity : `a.transpose`

以矩阵表示线性变换

- 变换 (transformation) 是指把矢量从一个空间映射至另一空间

$$\mathbf{v}' = f(\mathbf{v})$$

- 线性变换要满足

1. 可加性：

$$f(\mathbf{v} + \mathbf{u}) = f(\mathbf{v}) + f(\mathbf{u})$$

2. 齐次性：

$$f(a\mathbf{v}) = af(\mathbf{v})$$

矩阵和三维空间变换

- 矩阵可以表示多种三维空间变换
 - 平移 (translation)
 - 旋转 (rotation)
 - 缩放 (scaling)
 - 切变 (shearing)
 - 反射 (reflection)
 - 投影 (projection)

以矩阵表示矢量

- 为了以矩阵来变换矢量，须把矢量表示为矩阵

- 行矩阵（row matrix）即 $1 \times n$ 矩阵

$$\mathbf{v}_1 = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}$$

- 列矩阵（column matrix）即 $n \times 1$ 矩阵

$$\mathbf{v}_2 = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \mathbf{v}_1^T$$

- Unity 和《RTR》使用列矩阵，《GEA》使用行矩阵

对矢量进行线性变换

- 使用列矩阵表示矢量时，用这样的矩阵乘法表示变换：

$$\mathbf{v}' = \mathbf{M}\mathbf{v}$$
$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

- 为什么不直接展开运算，而要引入矩阵？

变换串接

- 如果要对矢量进行多次线性变换：

$$\mathbf{v}' = \mathbf{M}_3(\mathbf{M}_2(\mathbf{M}_1 \mathbf{v}))$$

- 由于矩阵乘法具有结合律，可以改写成：

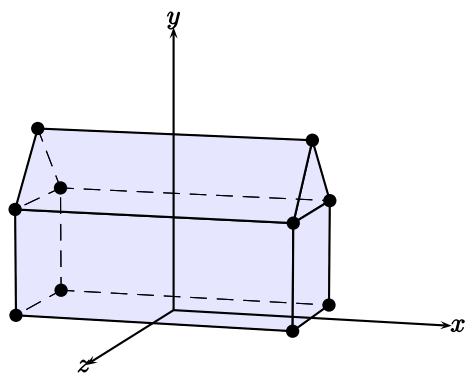
$$\mathbf{v}' = (\mathbf{M}_3 \mathbf{M}_2 \mathbf{M}_1) \mathbf{v}$$

- 要把 $\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3$ 施于大量矢量时，先计算

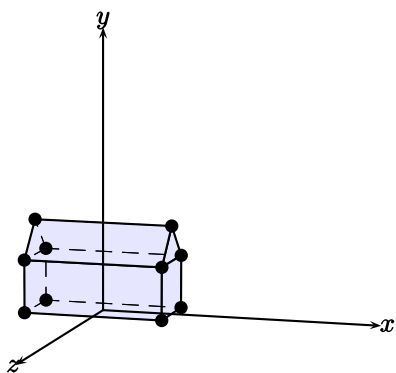
$$\mathbf{M} = \mathbf{M}_3 \mathbf{M}_2 \mathbf{M}_1$$

- 无论线性变换有多复杂，最后施于矢量时都是花费相同的运算量！

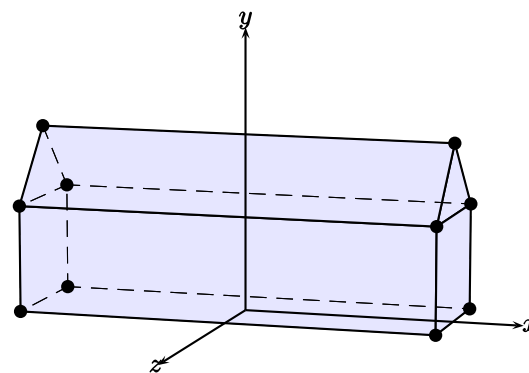
缩放变换



原始



等比缩放 $s = [0.5 \ 0.5 \ 0.5]$



非等比缩放 $s = [1.5 \ 1 \ 1]$

等比与非等比缩放

- 给定各轴的缩放比 $\mathbf{s} = \begin{bmatrix} s_x & s_y & s_z \end{bmatrix}^T$ ，把矢量缩放：

$$\mathbf{v}' = \mathbf{s} \otimes \mathbf{v}$$

$$= \begin{bmatrix} s_x v_x & s_y v_y & s_z v_z \end{bmatrix}^T$$

- 以矩阵表示：

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

- 若 $s_x = s_y = s_z$ ，则称为等比缩放 (uniform scaling)

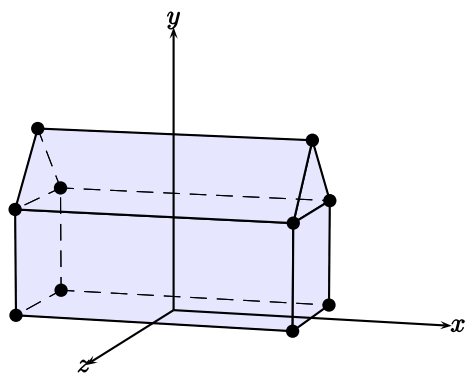
缩放的逆变换

- 使用缩放比的倒数：

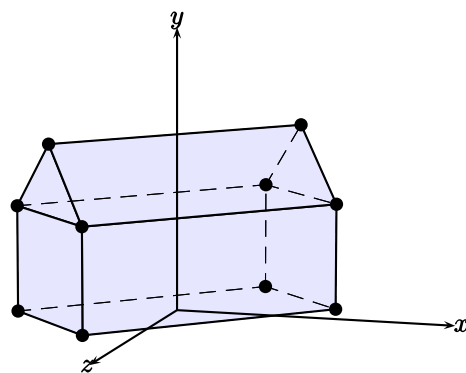
$$\mathbf{S}\mathbf{S}^{-1} = \mathbf{I}$$

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & \frac{1}{s_z} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

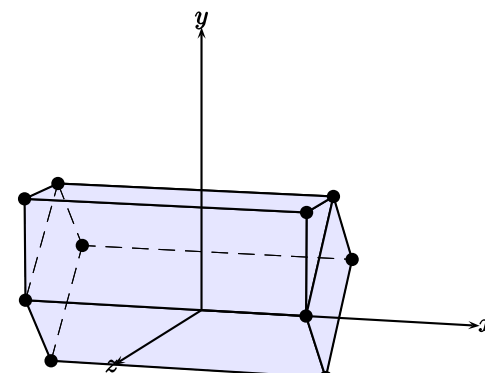
旋转变换



原始



绕 y 轴旋转 $\theta = 45^\circ$



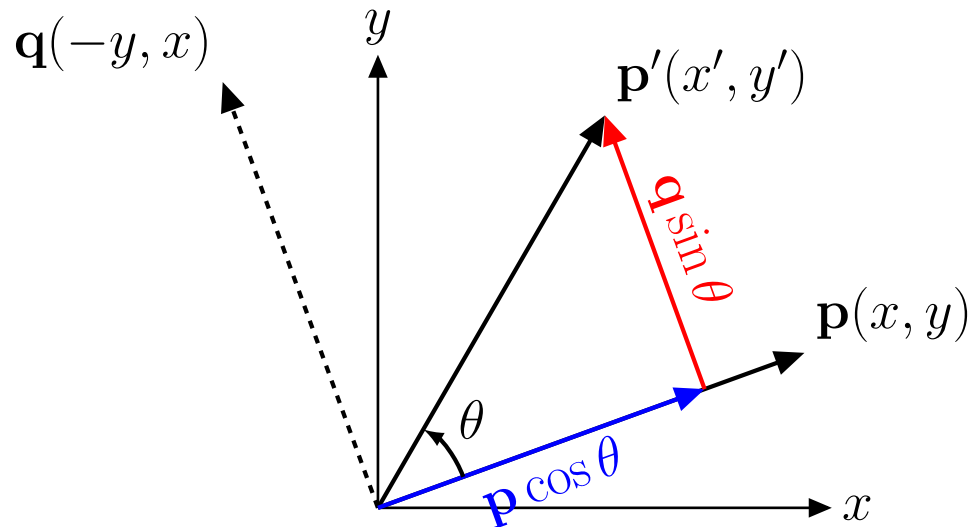
绕 x 轴旋转 $\theta = 45^\circ$

旋转特性

- 当把个物体旋转时
 1. 在某线上的点保持不变，该线称为旋转轴
 2. 任意两点旋转后的距离保持不变
 3. 所有的点绕旋转轴旋转某个角度
 4. 对任何整数 k ，一个点旋转 $2\pi k + \theta$ 角度后不变
- 两个旋转变换串接后也是一个旋转变换（旋转集合与乘法形成旋转群）

二维旋转

- 把 \mathbf{p} 绕原点逆时针旋转 θ 度至 \mathbf{p}'



$$\mathbf{p}' = \mathbf{p} \cos \theta + \mathbf{q} \sin \theta$$

$$x' = x \cos \theta + (-y \sin \theta)$$

$$y' = y \cos \theta + x \sin \theta$$

二维旋转矩阵

- 重新排列：

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

- 写成矩阵形式：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

三维旋转

- 绕各主轴的三维旋转等价于二维旋转

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

旋转的逆变换

- 旋转的逆变换只需反方向旋转相同角度：

$$\begin{aligned}\mathbf{R}_z^{-1}(\theta) &= \mathbf{R}_z(-\theta) \\ &= \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \mathbf{R}_z^T\end{aligned}$$

- 实际上，任何旋转矩阵的逆都是其转置矩阵：

$$\mathbf{R}^{-1} = \mathbf{R}^T$$

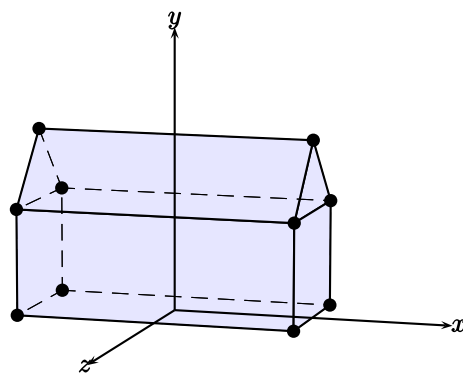
- 因为旋转矩阵是一种正交矩阵（orthogonal matrix）

欧拉角

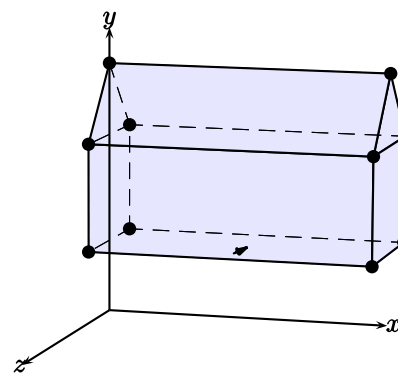
- 任何三维旋转都可以表示为3个欧拉角
- Unity 中使用的旋转次序是 $z - x - y$:
$$\mathbf{v}' = [\mathbf{R}_y(yaw)\mathbf{R}_x(pitch)\mathbf{R}_z(roll)]\mathbf{v}$$
- Unity 没有直接用欧拉角生成矩阵的函数

```
Matrix4x4 m = Matrix4x4.TRS(  
    Vector3.zero,  
    Quaternion.Euler(pitch, yaw, roll),  
    Vector3.one);
```

平移变换



原始



平移 $\mathbf{t} = [4 \ 2 \ 0]$

平移的问题

- 把点平移是很简单的：

$$\mathbf{p}' = \mathbf{p} + \mathbf{t}$$

- 然而，除非 $\mathbf{t} = \mathbf{0}$ ，这种变换并不能用 3×3 矩阵表示：

$$\mathbf{p} + \mathbf{t} \neq \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \mathbf{p}$$

齐次坐标

- 解决方法是齐次坐标 (homogeneous coordinates)
- 把点或矢量表示为四维矢量

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ p_w \end{bmatrix}$$

- 点的 $p_w = 1$ ，矢量的 $p_w = 0$ （不受平移影响）
- 后话：在图形学中还会利用齐次坐标实现透示投影

平移与齐次坐标

- 使用齐次坐标后，点的平移可以用矩阵表示：

$$\mathbf{p}' = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{bmatrix}$$

- 扩展缩放和旋转变换为 4×4 矩阵
 - 除 $m_{44} = 1$ 外其他新元素都设为0

刚体变换

- 刚体变换 (rigid body transformation) 只含旋转和平移

$$\mathbf{v}' = \mathbf{TRv}$$

$$= \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}$$

- 可从矩阵轻易因式分解为 \mathbf{T} 和 \mathbf{R}
- 逆变换也容易求得

非等比缩放旋转平移

- 在刚体变换上加入非等比缩放

$$\mathbf{v}' = \mathbf{TRS}\mathbf{v}$$

$$= \begin{bmatrix} s_x r_{11} & s_y r_{12} & s_z r_{13} & t_x \\ s_x r_{21} & s_y r_{22} & s_z r_{23} & t_y \\ s_x r_{31} & s_y r_{32} & s_z r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}$$

- 仅一次变换后仍可因式分解，两次以上无法分解
- 逆变换需采用矩阵求逆

模型空间

- 模型空间（model space）、物体空间（object space）、局部空间（local space）
- 用建模工具（如3ds Max）制作模型时的空间
- 原点通常是质心，或是中心点的底部

世界空间

- 世界空间 (world space)
- 各个游戏中物体的位置 / 定向和缩放在这空间表示
- 每个物体的变换可表示为 $\mathbf{M}_{\text{model} \rightarrow \text{world}}$
 - 常简称为世界矩阵 (world matrix)

- 把网格顶点从模型空间变换至世界空间：

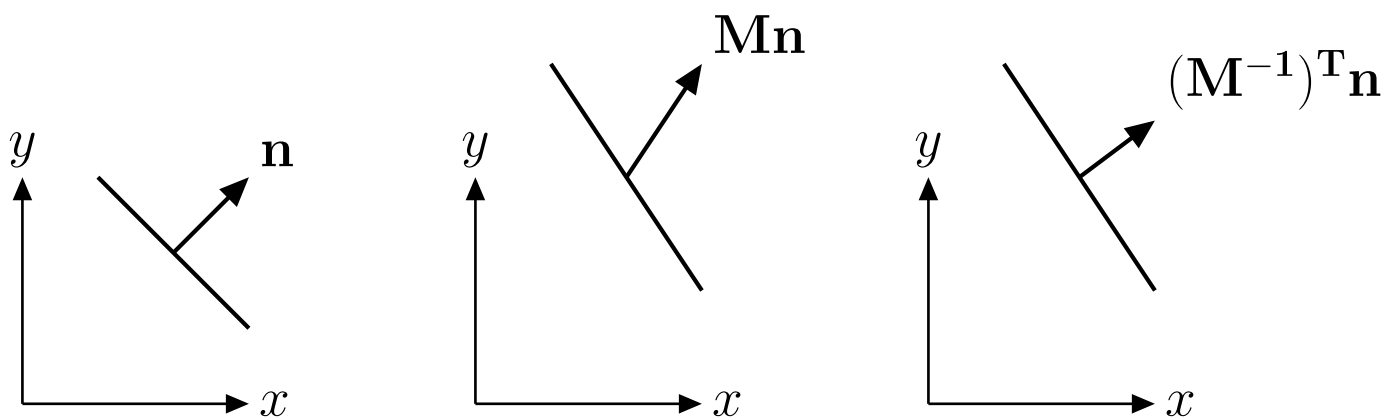
$$\mathbf{V}_{\text{world}} = \mathbf{M}_{\text{model} \rightarrow \text{world}} \mathbf{V}_{\text{model}}$$

- 如有逆矩阵 $\mathbf{M}_{\text{world} \rightarrow \text{model}} = \mathbf{M}_{\text{model} \rightarrow \text{world}}^{-1}$ 可反向变换：

$$\mathbf{V}_{\text{model}} = \mathbf{M}_{\text{world} \rightarrow \text{model}} \mathbf{V}_{\text{world}}$$

变换法线

- 法线在非等比缩放后，便不会垂直于表面
- 这种情况需要用 $(\mathbf{M}^{-1})^T$ 来变换法线



变换法线：证明

- 设 $\mathbf{p}_1, \mathbf{p}_2$ 为表面上接近的两点，切线 $\mathbf{t} = \mathbf{p}_2 - \mathbf{p}_1$
- 变换后的切线 \mathbf{t}' 等于两点分别变换后的差

$$\mathbf{t}' = \mathbf{M}\mathbf{p}_2 - \mathbf{M}\mathbf{p}_1 = \mathbf{M}(\mathbf{p}_2 - \mathbf{p}_1) = \mathbf{M}\mathbf{t}$$

- 变换前的法线须与切线垂直：

$$\mathbf{n} \cdot \mathbf{t} = 0 \Leftrightarrow \mathbf{n}^T \mathbf{t} = 0 \Leftrightarrow \mathbf{n}^T \mathbf{M}^{-1} \mathbf{M} \mathbf{t} = 0 \Leftrightarrow (\mathbf{n}^T \mathbf{M}^{-1}) \mathbf{t}' =$$

- 变换后的法线须与切线垂直： $(\mathbf{n}'^T) \mathbf{t}' = 0$

- 结合两式的括号部分：

$$\mathbf{n}'^T = \mathbf{n}^T \mathbf{M}^{-1}$$

$$\mathbf{n}' = (\mathbf{n}^T \mathbf{M}^{-1})^T$$

$$= (\mathbf{M}^{-1})^T \mathbf{n}$$

总结：矩阵与几何变换

- 矩阵乘法可表示线性变换
- 矩阵乘法可串接变换
- 三维缩放、旋转、平移可表示成 4×4 矩阵
- 以齐次坐标表示点

矩阵概念一览

概念	符号	意义
矩阵	\mathbf{M}	$m \times n$ 个元素的数组
矩阵乘法	$\mathbf{M}\mathbf{v}$	线性变换
转置矩阵	\mathbf{M}^T	行与列互换
逆矩阵	\mathbf{M}^{-1}	$\mathbf{M}\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$
正交矩阵	\mathbf{Q}	$\mathbf{Q}^{-1} = \mathbf{Q}^T$

常用线性变换一览

变换	符号	意义
缩放	$\mathbf{S}(s)$	以 s 的比率作（等比／非等比）缩放
旋转	$\mathbf{R}_z(\theta)$	绕 z 轴旋转 θ 角度
平移	$\mathbf{T}(\mathbf{t})$	平移 \mathbf{t}
刚体	\mathbf{TR}	先旋转后平移（两点距离不变）
通用	\mathbf{TRS}	非等比缩放、旋转、平移
法线	$(\mathbf{M}^{-1})^T$	\mathbf{M} 含非等比缩放时的法线变换矩阵

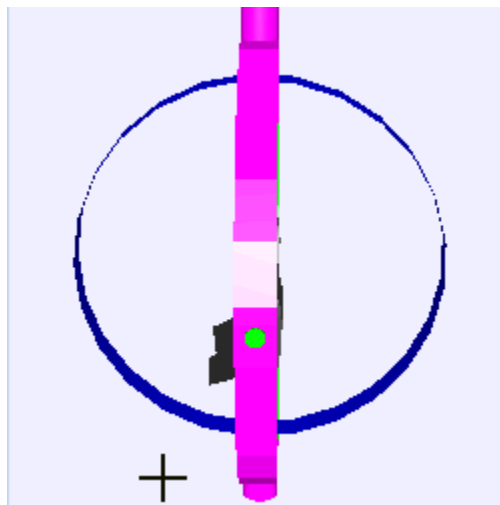
练习：矩阵和变性变换

1. 利用 $\mathbf{A}\mathbf{I} = \mathbf{I}\mathbf{A} = \mathbf{I}$ 及 $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ 证明 $(\mathbf{A}^{-1})^{-1} = \mathbf{A}$ 。
2. 以 $\mathbf{R}_z(\theta)$ 和 $\mathbf{T}(\mathbf{t})$ 表示在 xy 平面上绕点 $(3, 4)$ 旋转 $\frac{\pi}{3}$ 的变换。

4. 四元数与三维旋转

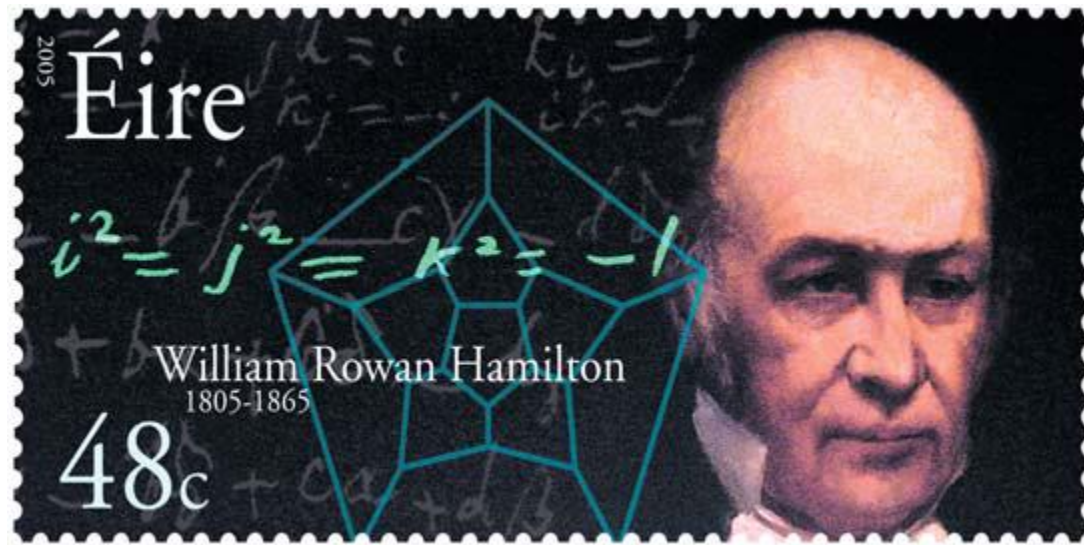
旋转矩阵的问题

- 花了 9 个实数去表示 3 个自由度
- 难以对两个旋转矩阵插值
- 用欧拉角会有万向节死锁问题 (gimbal lock)



四元数

- 四元数（[quaternion](#)）由威廉·哈密顿（William Hamilton）于1843年发明



- Shoemake在1985年SIGGRAPH把四元数引进图形学²

四元数与复数

- 类似于复数 $z = a + ib$ ，四元数可写成

$$\begin{aligned} \mathbf{q} &= \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{q}_v & q_w \end{bmatrix} \\ &= iq_x + jq_y + kq_z + q_w \end{aligned}$$

- 类似于复数中 $i^2 = -1$ ，四元数的分量有这些特性：

$$i^2 = j^2 = k^2 = ijk = -1,$$

$$jk = -kj = i, \quad ki = -ik = j, \quad ij = -ji = k,$$

- Unity : `new Quatenrion(x, y, z, w)`

四元数乘法

$$\begin{aligned}
 \mathbf{qr} &= (iq_x + jq_y + kq_z + q_w)(ir_x + jr_y + kr_z + r_w) \\
 &= i(q_yr_z - q_zr_y + q_xr_w + q_wr_x) \\
 &\quad + j(q_zr_x - q_xr_z + q_yr_w + q_wr_y) \\
 &\quad + k(q_xr_y - q_yr_x + q_zr_w + q_wr_z) \\
 &\quad + q_wr_w - q_xr_x - q_yr_y - q_zr_z \\
 &= \left[\mathbf{q}_v \times \mathbf{r}_v + r_w\mathbf{q}_v + q_w\mathbf{r}_v \quad q_wr_w - \mathbf{q}_v \cdot \mathbf{r}_v \right]
 \end{aligned}$$

这是点积和叉积的起源！

- Unity : $\mathbf{q} * \mathbf{r}$

模与归一化

- 四元数的模和四维矢量相似：

$$\|q\| = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$$

- 模为1的四元数称为单位四元数
- 归一化

$$u = \frac{1}{\|q\|} q$$

以单位四元数旋转矢量

- 要令矢量 \mathbf{v} 绕 $\hat{\mathbf{u}}$ 轴旋转 θ 角度

- 设

$$\mathbf{q} = \left[\hat{\mathbf{u}} \sin \frac{\theta}{2} \quad \cos \frac{\theta}{2} \right]$$

$$\mathbf{q}^* = \left[-\hat{\mathbf{u}} \sin \frac{\theta}{2} \quad \cos \frac{\theta}{2} \right]$$

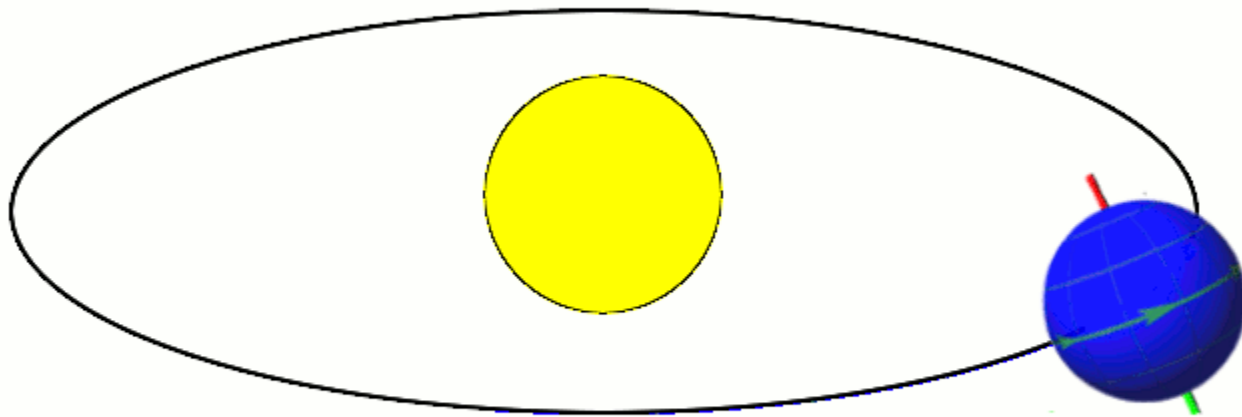
$$\mathbf{v} = \left[\mathbf{v} \quad 0 \right]$$

- 则旋转后的矢量 \mathbf{v}' 在 \mathbf{v}' 的矢量部分

$$\mathbf{v}' = \mathbf{q}\mathbf{v}\mathbf{q}^*$$

用例：地球自转与公转

- 地球的自转轴与轨道平面有约 23° 的倾角
- 模拟地球自转和公转，轨道平面为XZ
- 直接生成单位四元数



自转

```
public class EarthMovement : MonoBehaviour {  
    public float t = 0.0f; // day  
    public float earthRotationSpeed = 360.0f * Mathf.Deg2Rad;  
    public float tilt = 23.0f * Mathf.Deg2Rad;  
    void Update () {  
        float halfTheta = t * earthRotationSpeed * 0.5f;  
        Quaternion earthRotation = new Quaternion(  
            0, Mathf.Sin(halfTheta), 0, Mathf.Cos(halfTheta));  
  
        float halfTilt = tilt * 0.5f;  
        Quaternion earthTilt = new Quaternion(  
            0, 0, -Mathf.Sin(halfTilt), Mathf.Cos(halfTilt));  
  
        transform.rotation = earthTilt * earthRotation;  
    }  
}
```

公转

```
public class EarthMovement : MonoBehaviour {  
    // ...  
    public float orbitRadius = 2.0f;  
    public float orbitSpeed = 360.0f / 365.24f * Mathf.Deg2Rad;  
  
    void Update () {  
        // ...  
        float phi = t * orbitSpeed;  
        transform.position = new Vector3(orbitRadius * Mathf.Cos(phi)  
    }  
}
```

单位四元数与旋转矩阵

	旋转矩阵	单位四元数
旋转矢量	$\mathbf{R}\mathbf{v}$	$\mathbf{q}\mathbf{v}\mathbf{q}^*$
串接变换	$\mathbf{R}_2\mathbf{R}_1\mathbf{v}$	$\mathbf{q}_2\mathbf{q}_1\mathbf{v}\mathbf{q}_1^*\mathbf{q}_2^*$
逆变换	$\mathbf{R}^{-1} = \mathbf{R}^T$	$\mathbf{q}^{-1} = \mathbf{q}^*$

单位四元数 \rightarrow 旋转矩阵

$$\mathbf{R} = \begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_xq_y - 2q_zq_w & 2q_xq_z + 2q_yq_w \\ 2q_xq_y + 2q_zq_w & 1 - 2q_x^2 - 2q_z^2 & 2q_yq_z - 2q_xq_w \\ 2q_xq_z - 2q_yq_w & 2q_yq_z + 2q_xq_w & 1 - 2q_x^2 - 2q_y^2 \end{bmatrix}$$

深圳—旧金山最短路径

大圆弧



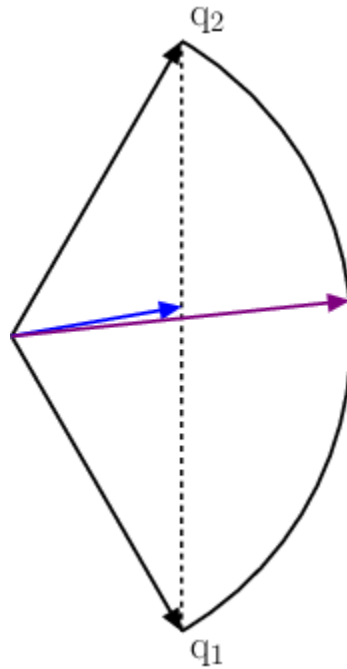
旋转插值

- 单位四元数等价于单位四维球面 S^3 上的点
- 两个旋转间的插值等价于两点在四维球面上插值
- 可使用线性插值再归一化

$$\text{LERP}(q_1, q_2, t) = \frac{(1 - t)q_1 + tq_2}{\|(1 - t)q_1 + tq_2\|}$$

LERP旋转不均匀速

- 蓝色 LERP (未归一化前)，紫色是匀速目标



球面插值

- 球面插值 (spherical linear interpolation, SLERP)

$$SLERP(q_1, q_2, t) = \frac{\sin((1 - t)\theta)}{\sin \theta} q_1 + \frac{\sin t\theta}{\sin \theta} q_2$$

- 当中 $\theta = \cos^{-1}(q_1 \cdot q_2)$
- 匀速 (t 与角度变化成正比)

压缩单位四元数

- 三维旋转只有3个DOF，但四元数需储了4个数
- 方法
 - 利用单位四元数的模为1
 - 忽略其中一个分量，如 q_w
 - 用其他三个分量还原

$$q_w = \pm \sqrt{1 - q_x^2 - q_y^2 - q_z^2}$$

- 问题：怎样决定正负号？

解决方法

- 假设： q 和 $-q$ 产生相同的旋转变换

$$\begin{aligned} \mathbf{v}' &= (-q)\mathbf{v}(-q)^* \\ &= (-1q)\mathbf{v}(-1q)^* \\ &= (q)(-1)\mathbf{v}(-1)(q^*) \\ &= q(-1)(-1)\mathbf{v}q^* \\ &= q\mathbf{v}q^* \end{aligned}$$

- 若来源的 $q_w \geq 0$ ，储存 q_x, q_y, q_z
- 若来源的 $q_w < 0$ ，储存 $-q_x, -q_y, -q_z$
- 那么可确保还原时 q_w 必为正

各种旋转表示法的比较

表示	储存量	串接	变换矢量	插值
3×3 矩阵	9	45	15	×
四元数	4	28	30	✓
欧拉角	3	×	×	×
轴角	4	×	×	×
旋转矢量	3	×	×	×

总结：四元数


- 四元数可以容易表示绕任意轴的旋转
- 非常紧凑
- 串接较矩阵快，变换较慢
- 避免欧拉角的万向节死锁
- 能实现球面线性插值（SLERP）

练习：四元数

1. 相对于矩阵乘法的单位元是 \mathbf{I}_n ，四元数乘法的单位元是什么？
2. 证明单位四元数 q 的乘法逆元是 q^* 。
3. 表示绕 x 轴旋转 a 弧度，然后绕 y 轴旋转 b 弧度的四元数。
4. 写出一个矢量 \mathbf{v} 经 s 倍的等比缩放，以单位四元数 q 旋转，再平移 \mathbf{t} 的变换公式。
 - 这种变换称为SQT变换。推导两个SQT变换 (s_1, q_1, \mathbf{t}_1) 和 (s_2, q_2, \mathbf{t}_2) 的串接方式。
 - 推导SQT变换 (s, q, \mathbf{t}) 的逆变换。
 - 比较SQT变换与 4×4 矩阵变换的优缺点。

5. 参考资料

- Gregory, Jason. Game engine architecture. Chapter 4. CRC Press, 2009. 中译本：《游戏引擎架构》第4章，叶劲峰译，电子工业出版社，2014.
- Lengyel, Eric. Mathematics for 3D game programming and computer graphics. Cengage Learning, 2012.
- Dunn, Fletcher, and Ian Parberry. 3D math primer for graphics and game development, 2nd Edition. CRC Press, 2011. 中译本：《3D数学基础：图形与游戏开发》，史银雪／陈洪／王荣静译，清华大学出版社，2005.

-
1. Johnson, Timothy E. "Sketchpad III: a computer program for drawing in three dimensions." Proceedings of the May 21-23, 1963, spring joint computer conference. ACM, 1963. 
 2. Shoemake, Ken. "Animating rotation with quaternion curves." ACM SIGGRAPH computer graphics. Vol. 19. No. 3. ACM, 1985. 