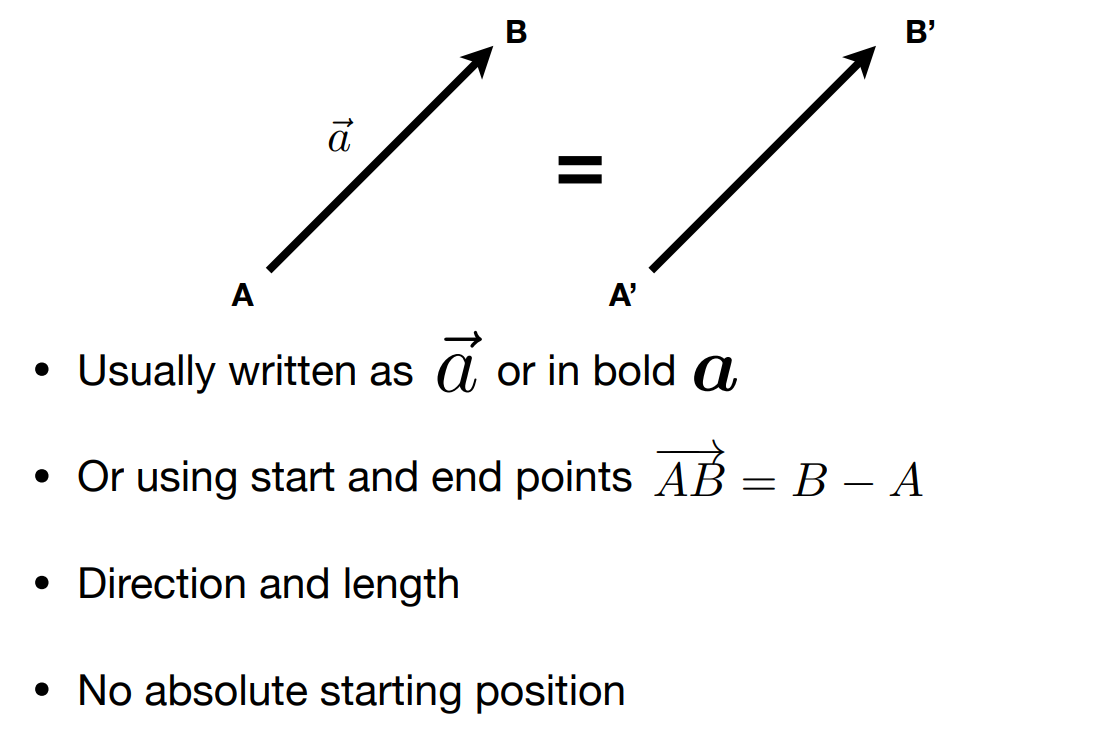
Graphics Notes

The Quick Brown Fox Jumps Over The Lazy Dog

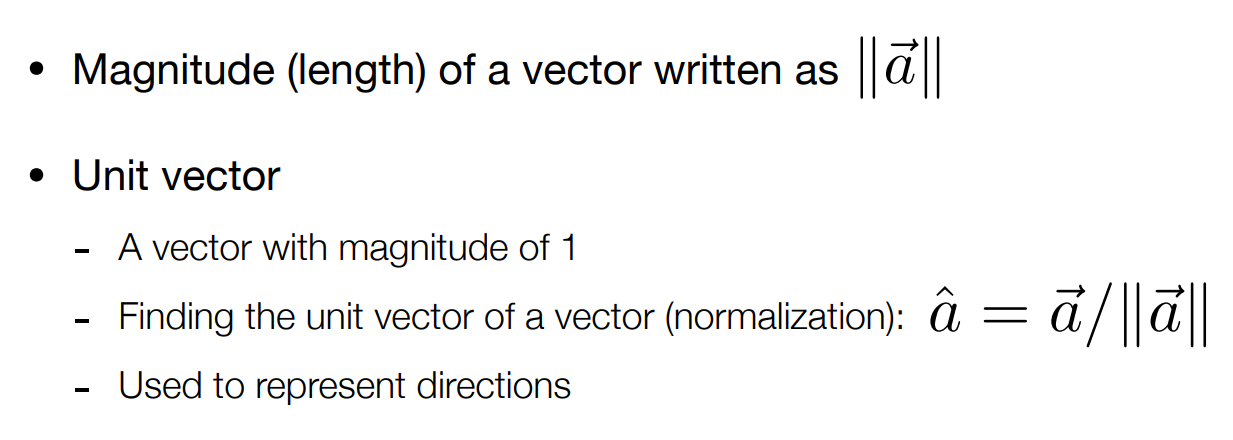
道理我都懂，鸽子为什么那么大

# Linear Algebra

## Vectors

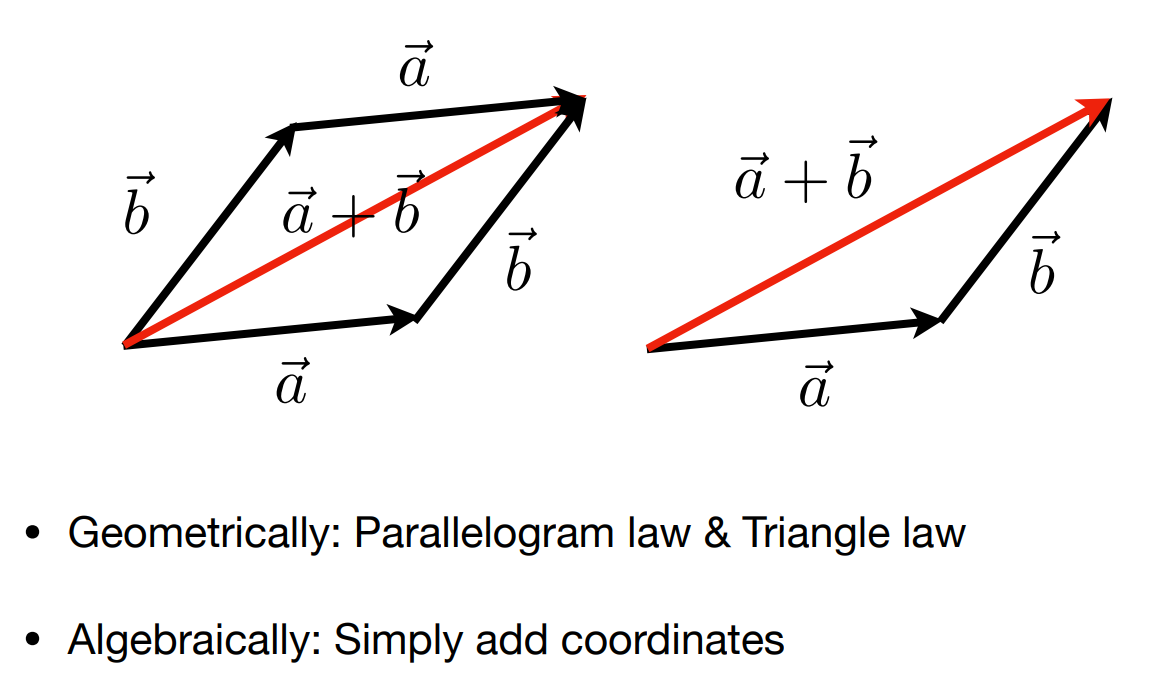


## Vector Normalization

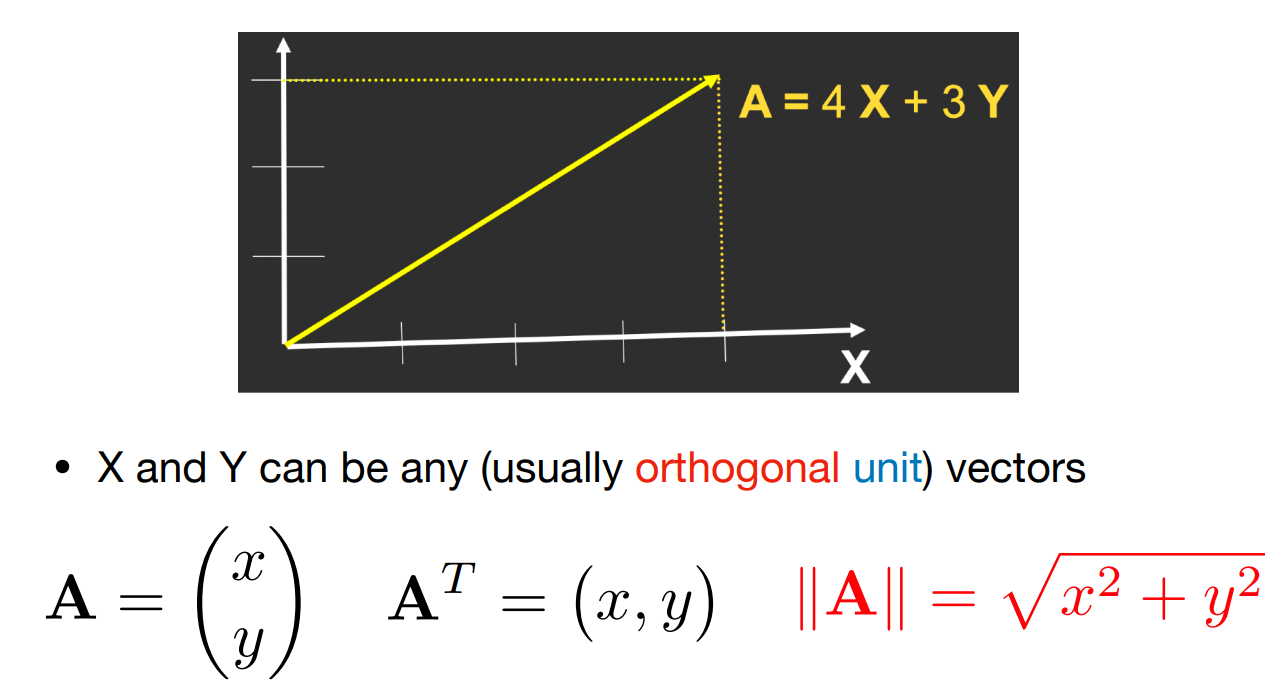


单位向量一般读作aHat（a上面的小尖角像是一顶帽子）

## Vector Addition



## Cartesian Coordinates

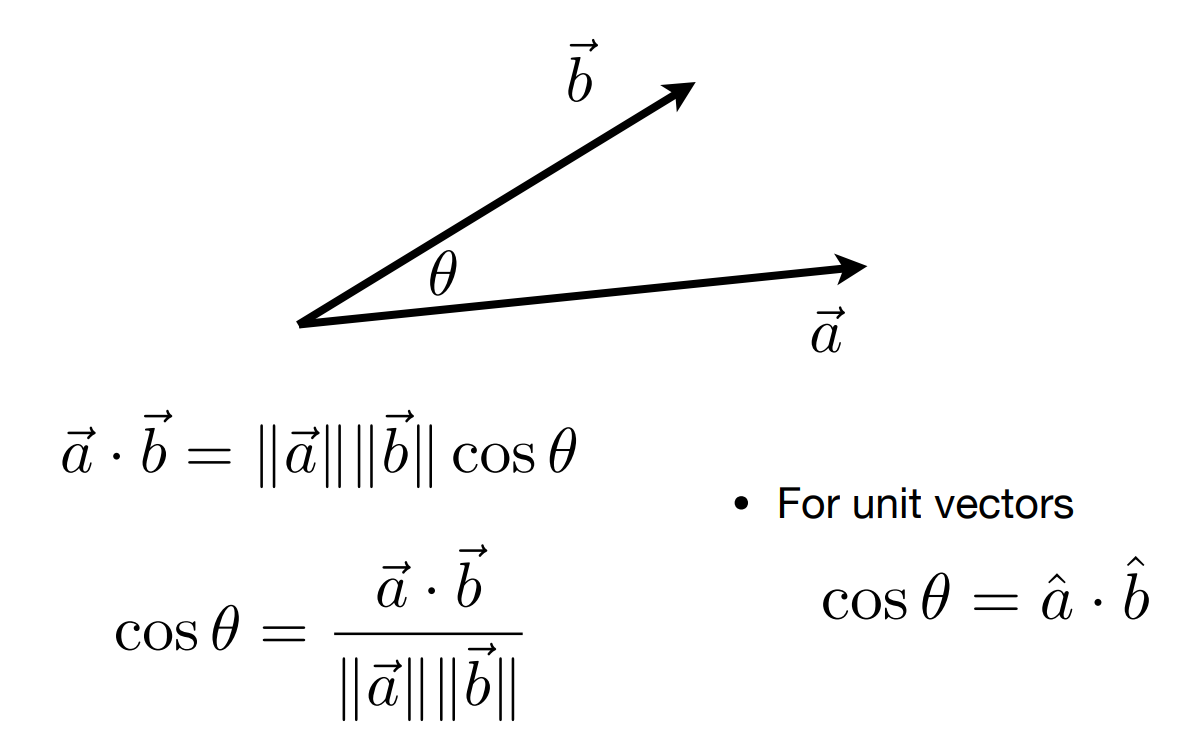


如果不做特别说明的话，默认是列向量（竖着写），转置（行列互换）后为横向量，使用列向量矩阵可以左乘

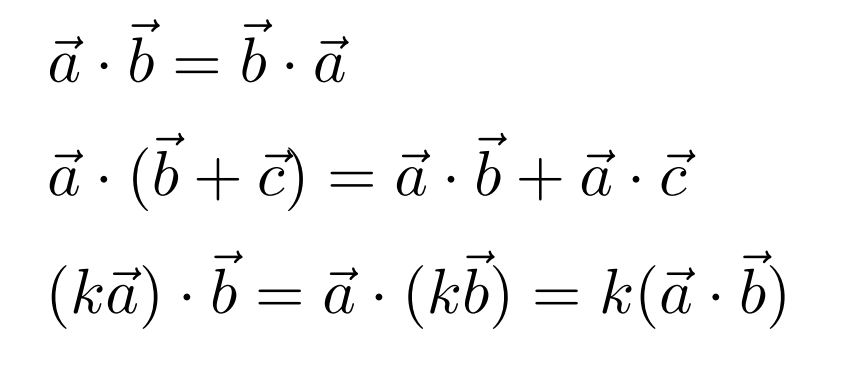
## Vector Multiplication

### Dot product

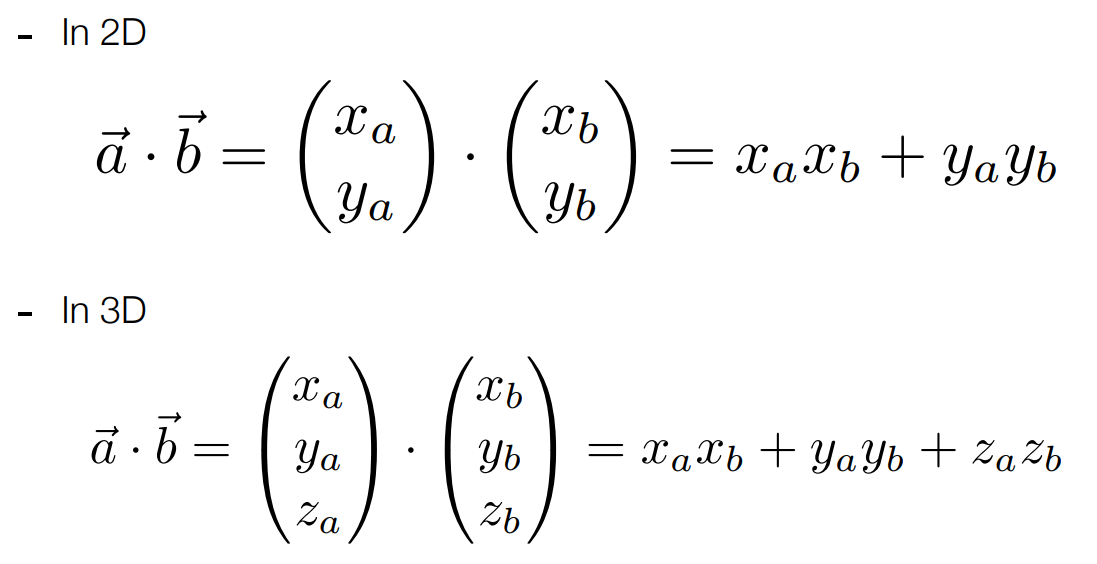
用于求两向量之间的夹角（两个向量点乘，结果是一个标量）



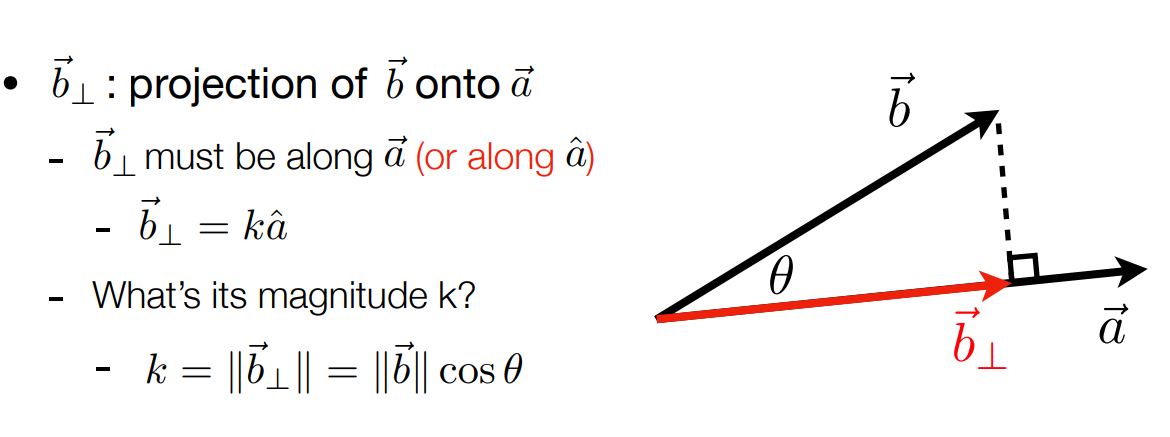
点乘的性质满足交换律，结合律和分配律：



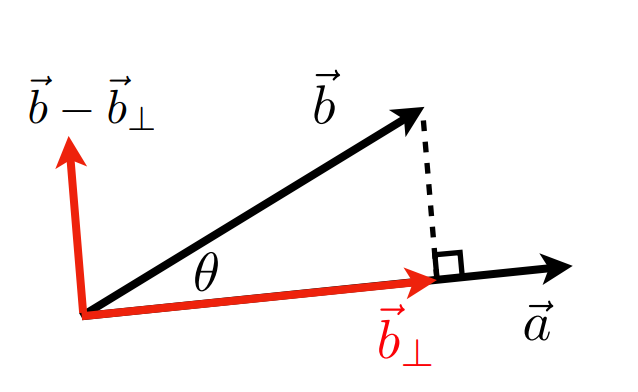
在笛卡尔坐标系中，先将对应部分相乘，然后相加

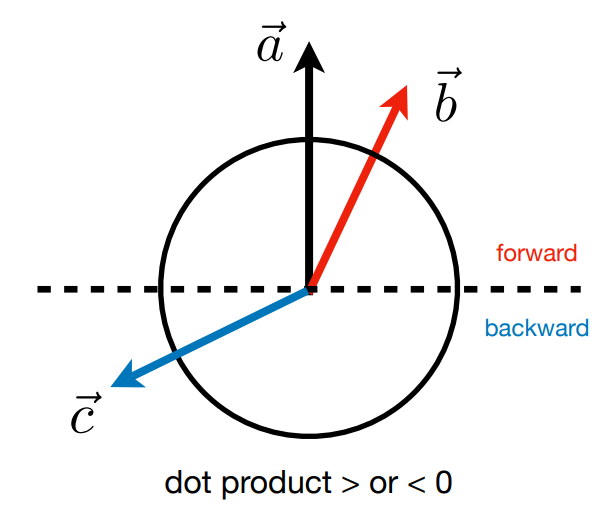


用于计算一个向量在另一个向量上的投影，投影算出来后还可以把这个向量进行平行与垂直方向的分解：

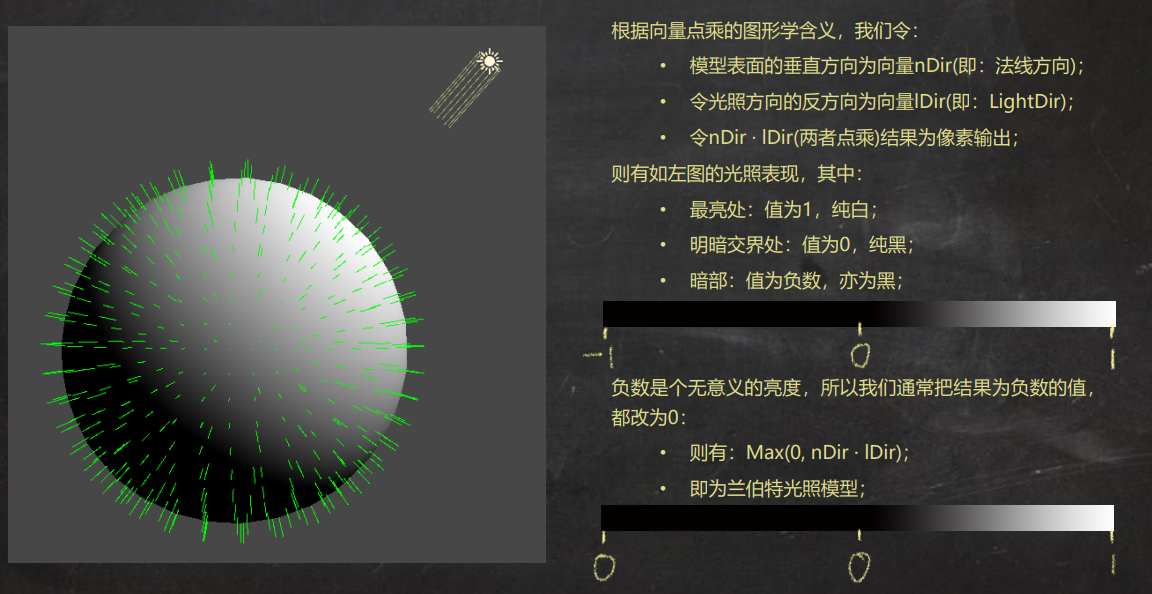


向量b在向量a上的投影读作b Perp(perpendicular)

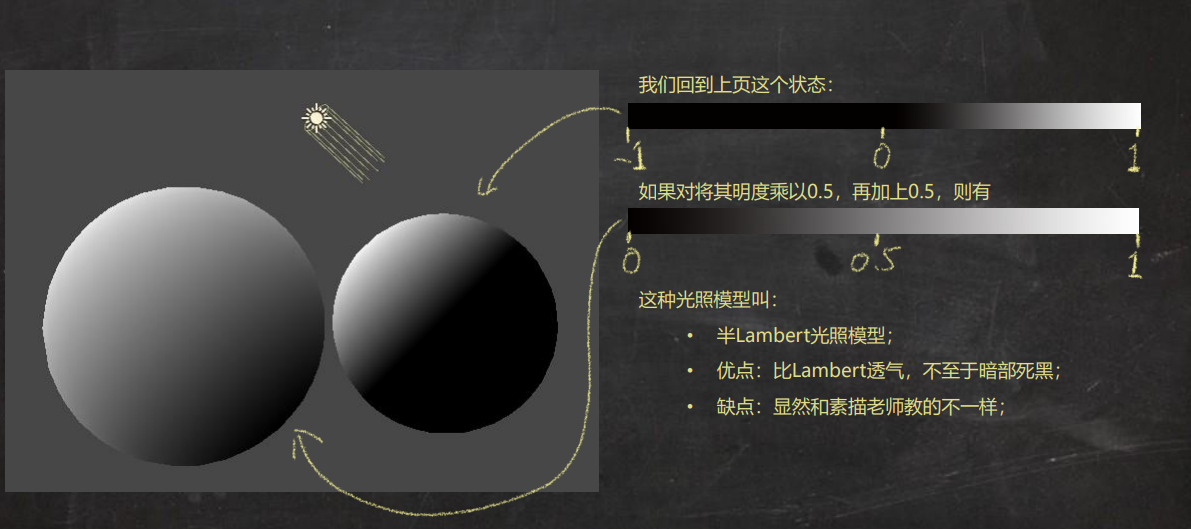




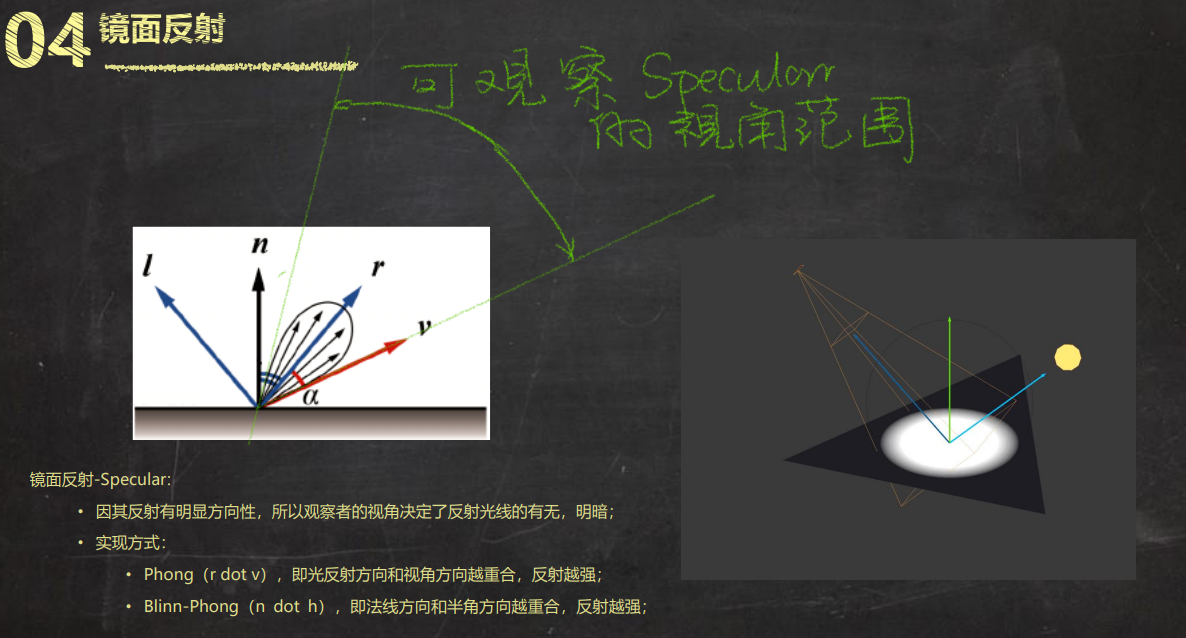
用两个向量点乘的结果与0比较来判断方向，如果大于0则说明方向接近一致，等于零说明互相垂直，小于零则方向接近相反，下面是一些光照模型中的应用（nDotl,rDotv,ndoth）



Lambert



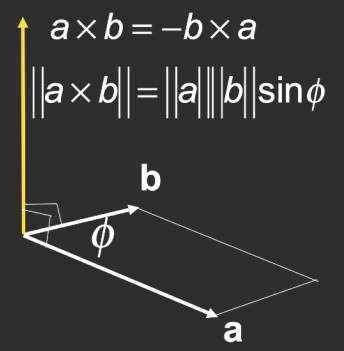
Half-Lambert



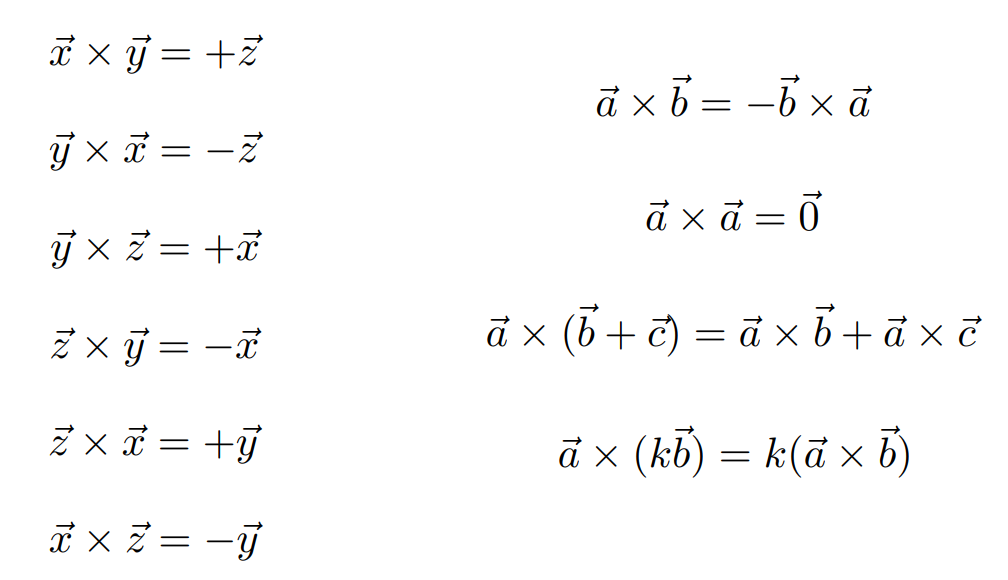
Phong and Blinn-Phong

### Cross product

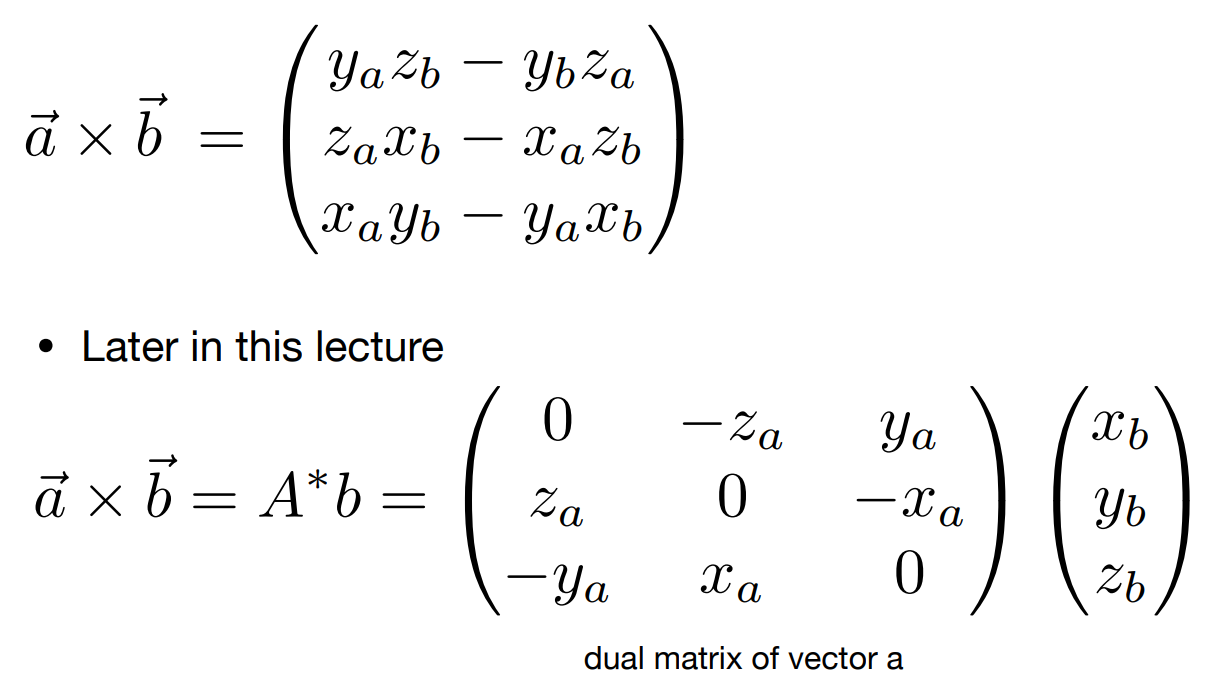
两个向量叉乘，结果也是一个向量，且结果的向量垂直于叉乘的两个向量组成的平面，叉乘结果的方向可以用右手螺旋定则判断（a叉乘b，手掌伸直，四指由a握向b，叉乘结果的方向为大拇指的方向，可以看出b 叉乘a得到的方向是相反的，叉乘并不满足交换律）



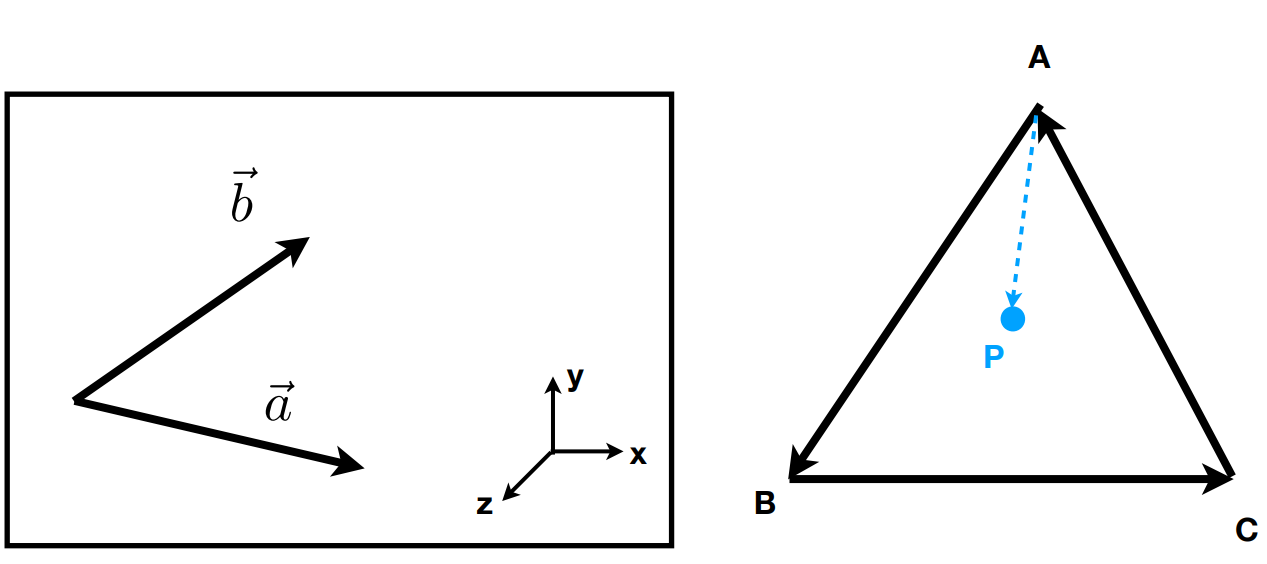
叉乘的性质：



在笛卡尔坐标系中的计算规则：

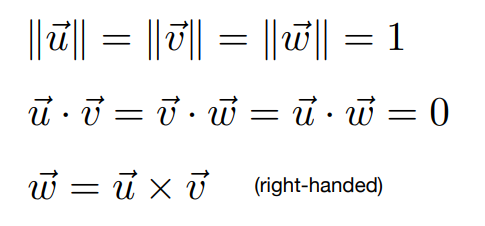


用于判断两个向量间的左右关系，点在多边形的内外关系（判断在多边形内外时会出现叉乘为0向量，点正好在边上的情况Corner Case，这时一般由自己决定边界情况算是内还是外）；判断点P是否在三角形ABC内可以通过计算AB AP,BC BP,CA CP的叉乘结果是否同大于（小于）0来得出：

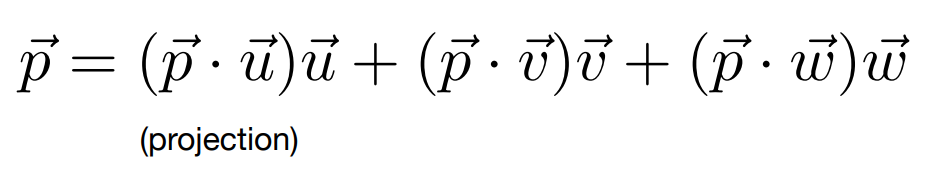


### Orthonormal bases and coordinate frames

三个向量同时满足以下条件时即可构成一个右手坐标系：

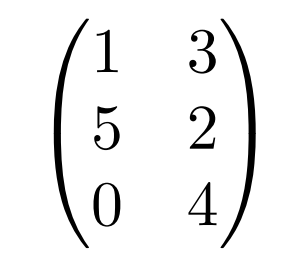


并且可以将投影进行分解：



## Matrices

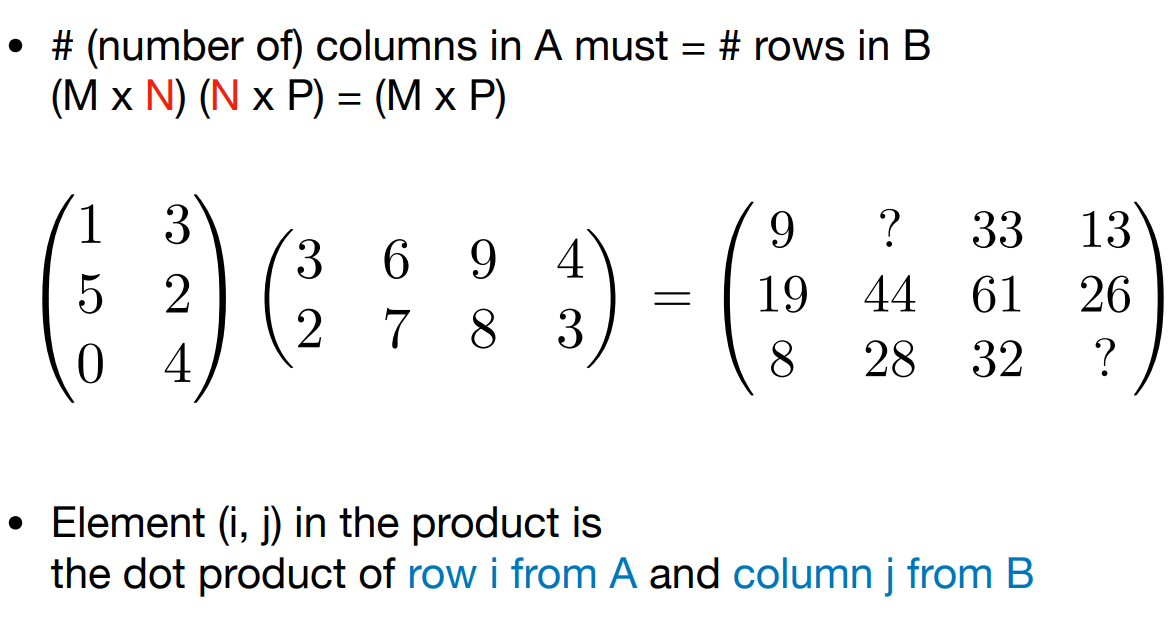
Array of numbers (m × n = m rows, n columns)：



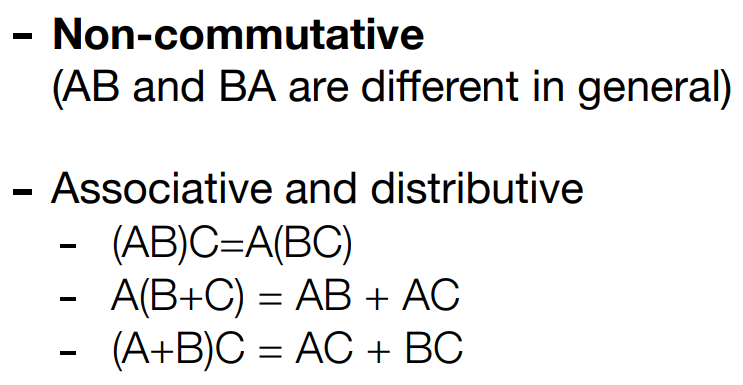
### Matrix-Matrix Multiplication

两个AB矩阵相乘的规则是A矩阵的行数要等于B矩阵的列数，相乘的结果是一个有着A矩阵行数，B矩阵列数的新矩阵，例如一个M行N列矩阵A，与一个N行P列的矩阵相乘，得到的是一个M行P列的矩阵

这边闫老师提出了一个很好的计算方式，结果矩阵的第i行j列的数可以由A矩阵的第i行点乘B矩阵的第j行得来，如下图第1行第2列的？，计算结果为A矩阵的第1行点乘B矩阵的第2列，也就是1\*3+3\*2=9；第3行第4列的？，计算结果为A矩阵的第3行点乘B矩阵的第4列，也就是0\*4+4\*3=12

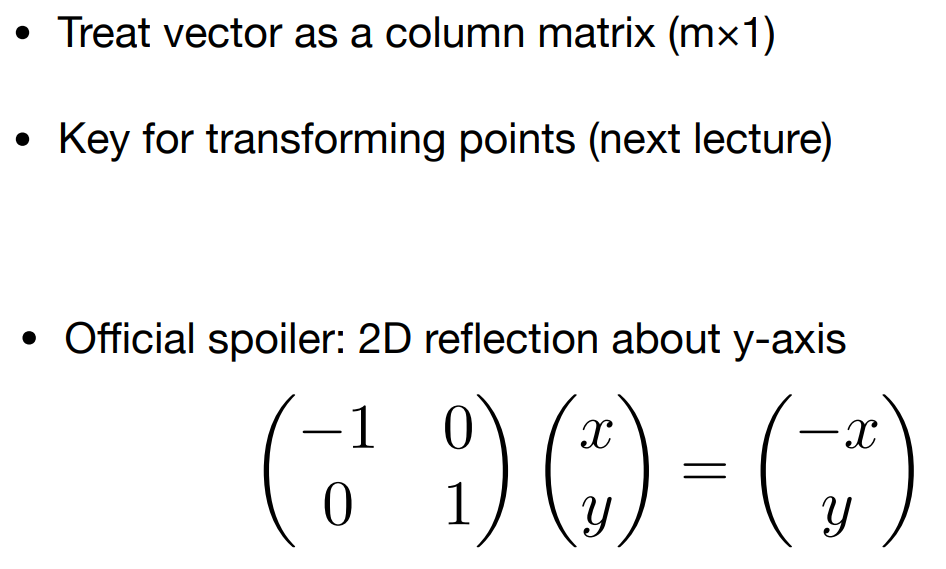


性质：大部分不满足交换律（只有一些特殊的矩阵在交换顺序相乘时结果也相同），满足结合律和分配律：



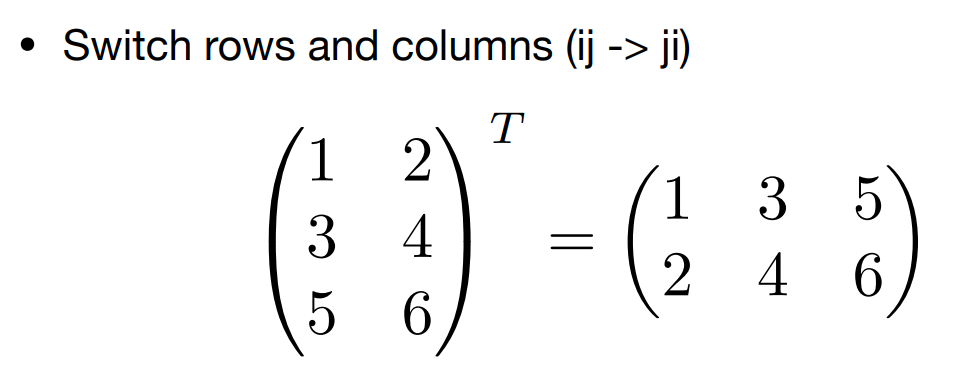
### Matrix-Vector Multiplication

把向量看成是一个m行一列的矩阵，一个向量乘矩阵可以完成旋转、平移、缩放等操作，下面是一个二位向量乘矩阵得到与y轴对称的向量：

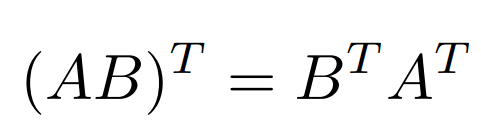


### Transpose of a Matrix

矩阵转置：矩阵的行列互换：



性质：穿脱原则，类似于栈，先入后出

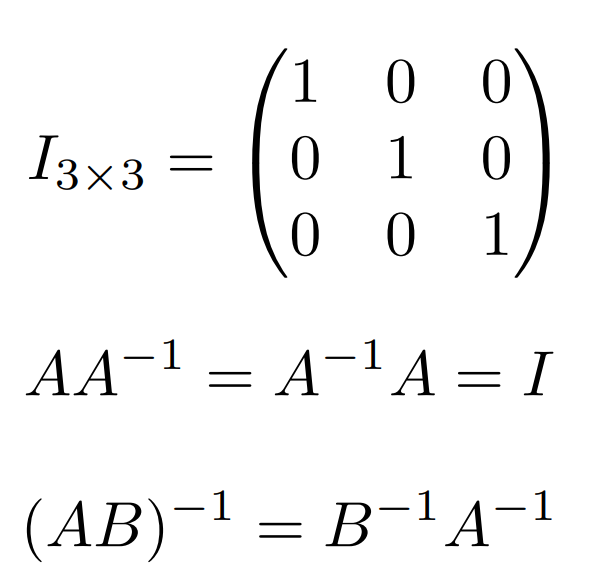


### Identity Matrix and Inverses

只有对角线上有非0元素的是单位矩阵

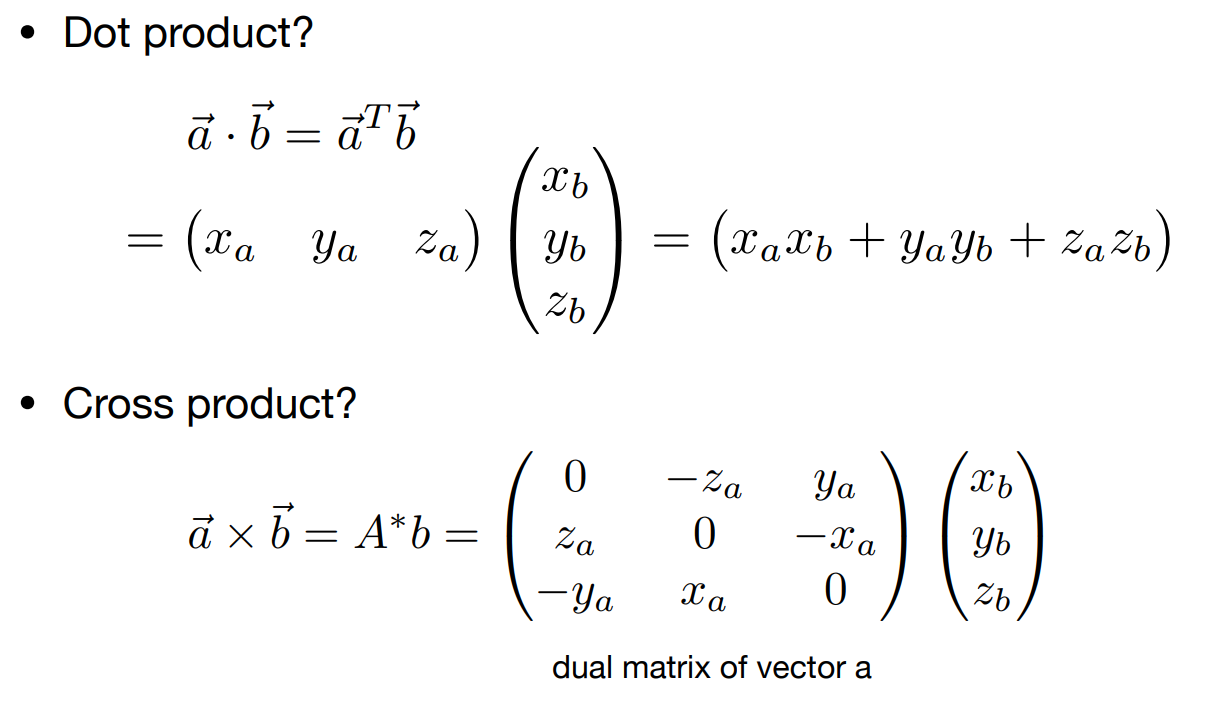
如果一个矩阵A乘另一个矩阵得到的结果是单位矩阵，那么这两个矩阵是互逆的

与矩阵的转置性质相同，穿脱原则，类似于栈，先入后出



### Vector multiplication in Matrix form

点乘和叉乘都可以写成矩阵的形式：

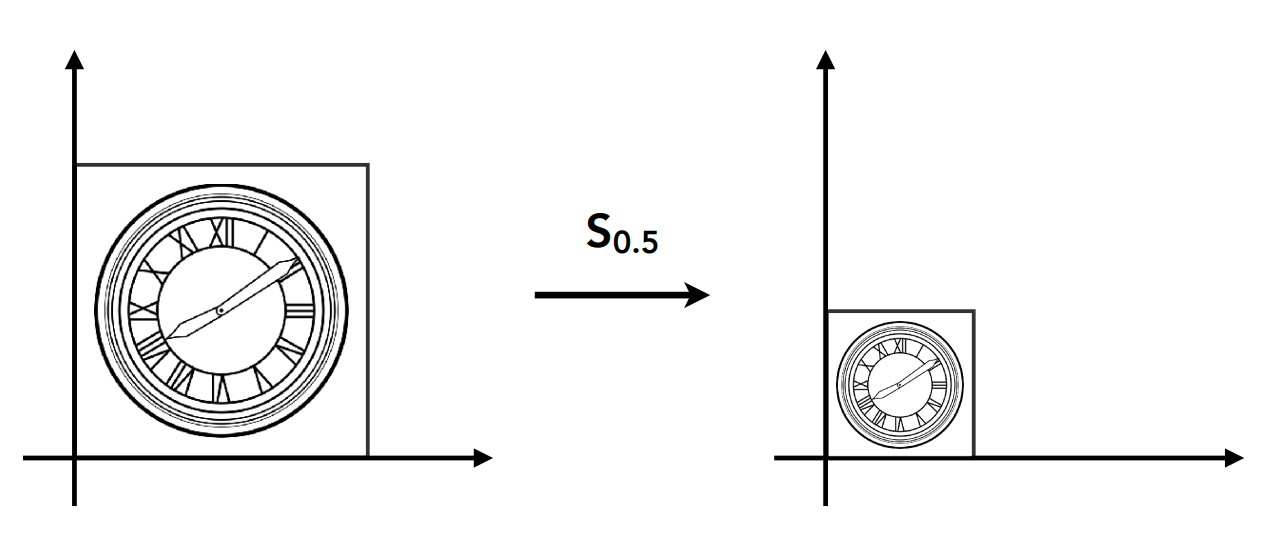


# Transformation

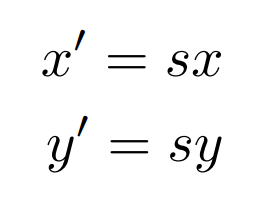
在光栅化成像的过程中大量用到了变换，常见的2维空间的变换有旋转，缩放和切变：

2D transformations: rotation, scale, shear

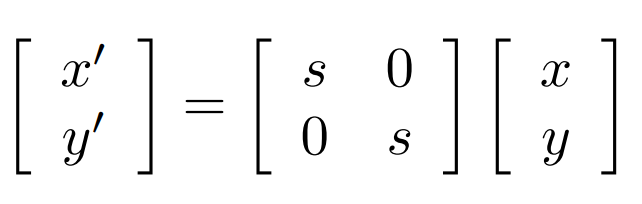
## Scale



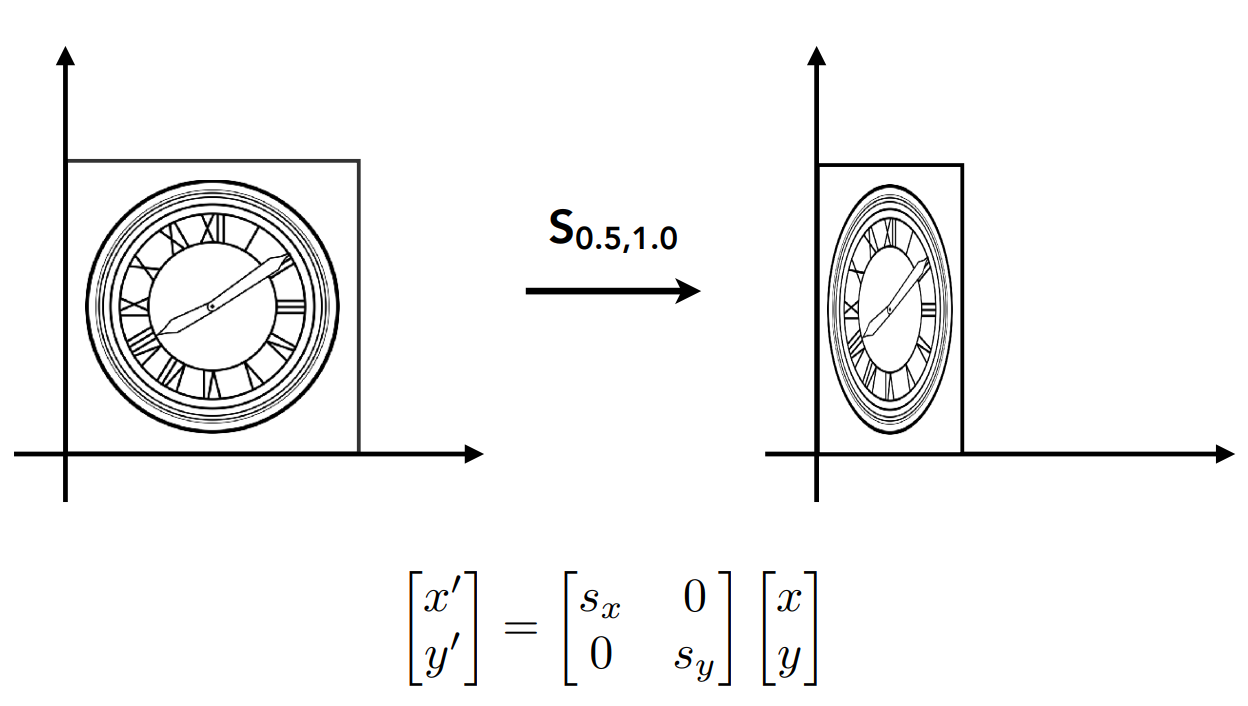
这边由左图缩放到右图，假设左图的坐标为x,y，变换后的右图为xPrime,yPrime，缩放的比例为s，则有：



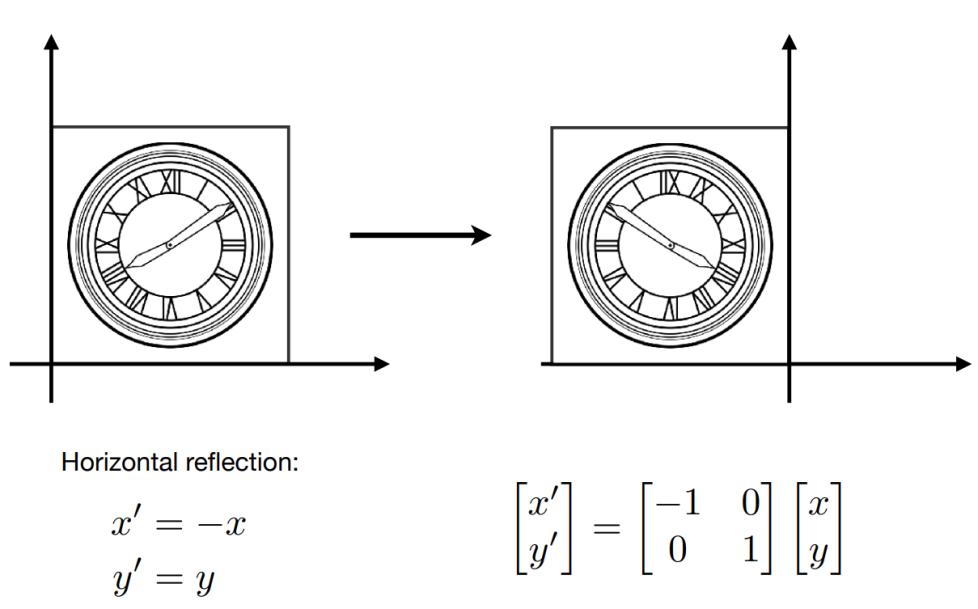
写成矩阵的形式，可以得到缩放矩阵：



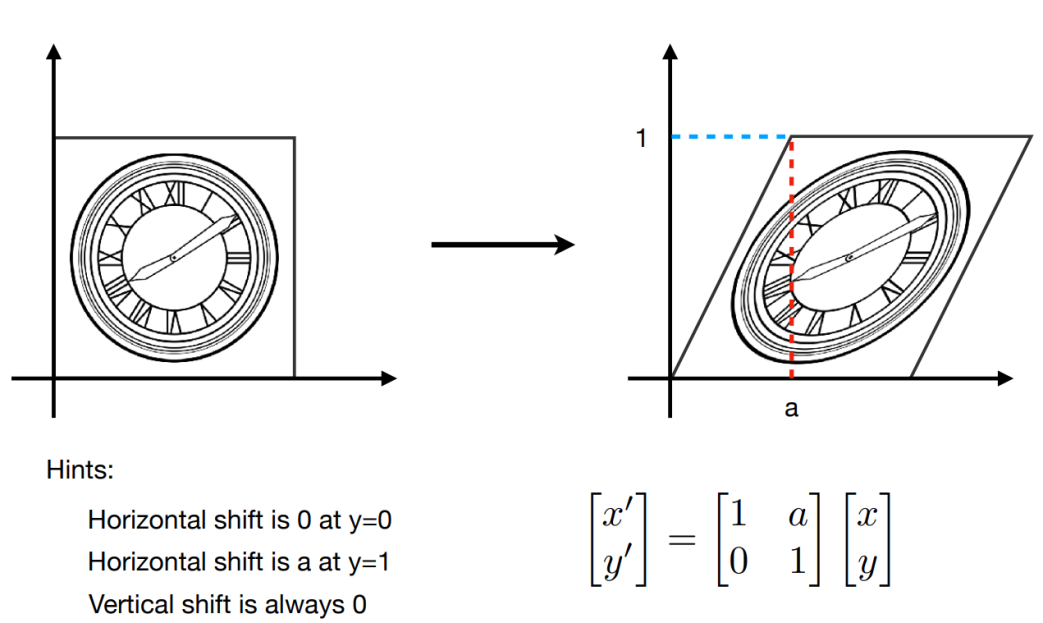
xy轴的缩放比例各不相同也是同理，仍然可以写成这样的矩阵：

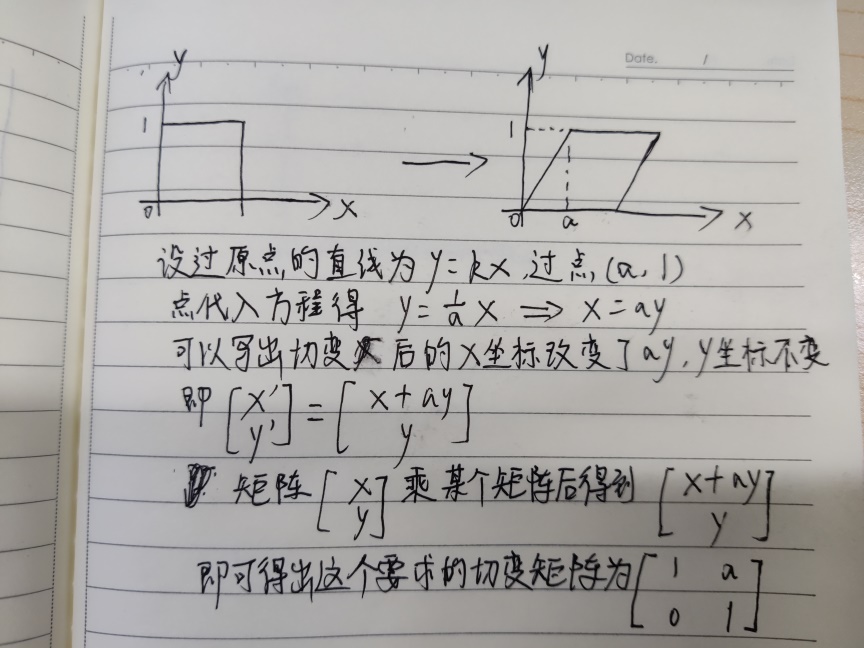


将原本的图片做一个对y轴的反射操作（镜像），可以看出翻转的y坐标不变，x坐标相反，可以写出变换的矩阵：



将左边的图看作是有弹性可以拉伸的，形成右侧的图形，将之称为切变（Shear），可以看出y坐标是不变的，水平方向变换之后可以用x+ay来表示，即可算出对应的切变矩阵（下面附上推算过程）



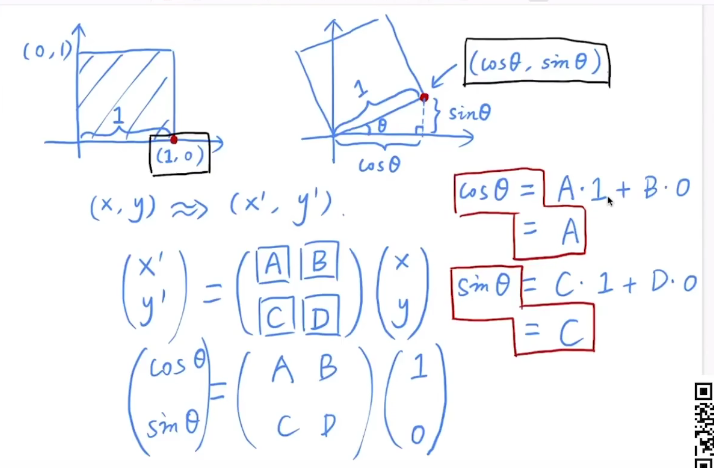


## Rotate (about the origin (0, 0), CCW by default)

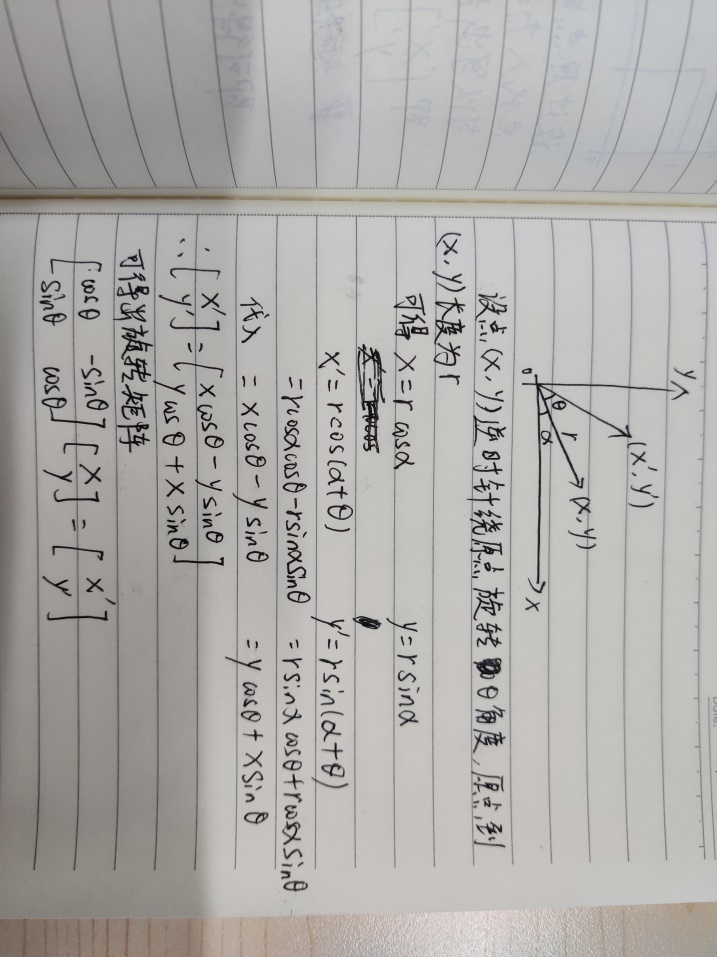
不做特别说明的话默认围绕（0，0）点逆时针旋转：下面附上闫大神的矩阵推导与自己的另一种矩阵推导方式



闫大神通过两个特殊的点推导出来（特殊值法yyds）：



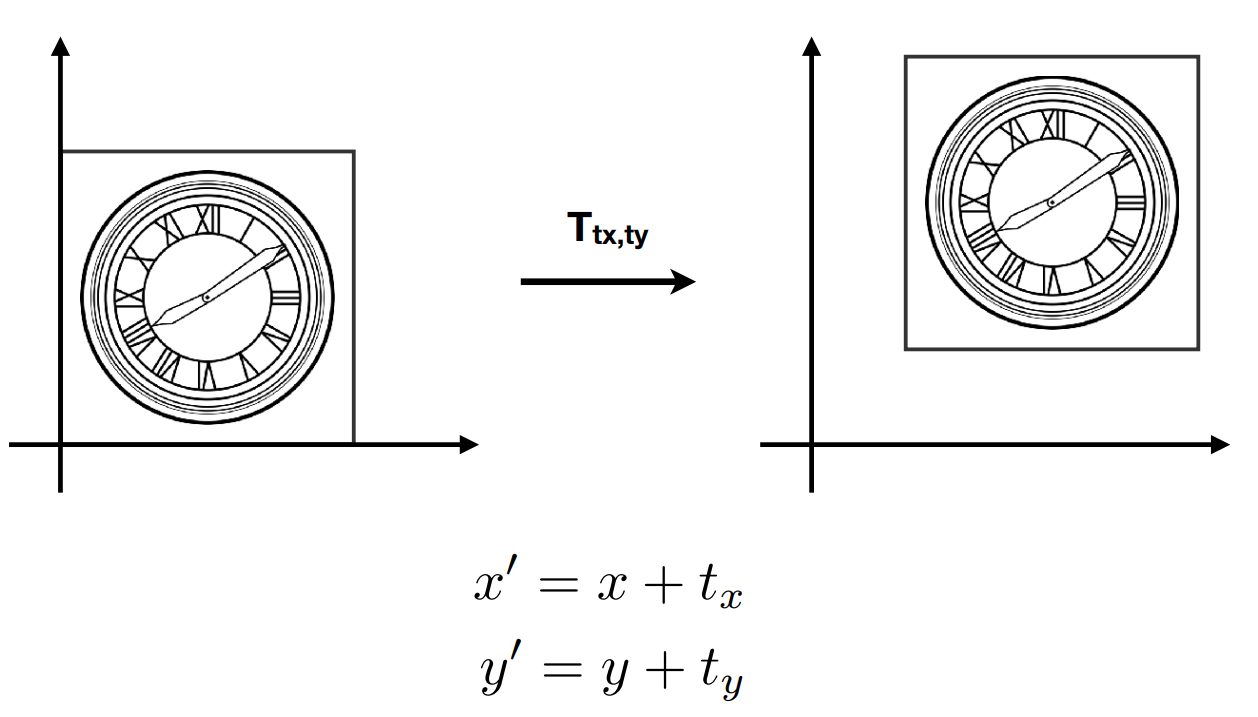
通过三角函数公式推导出来：



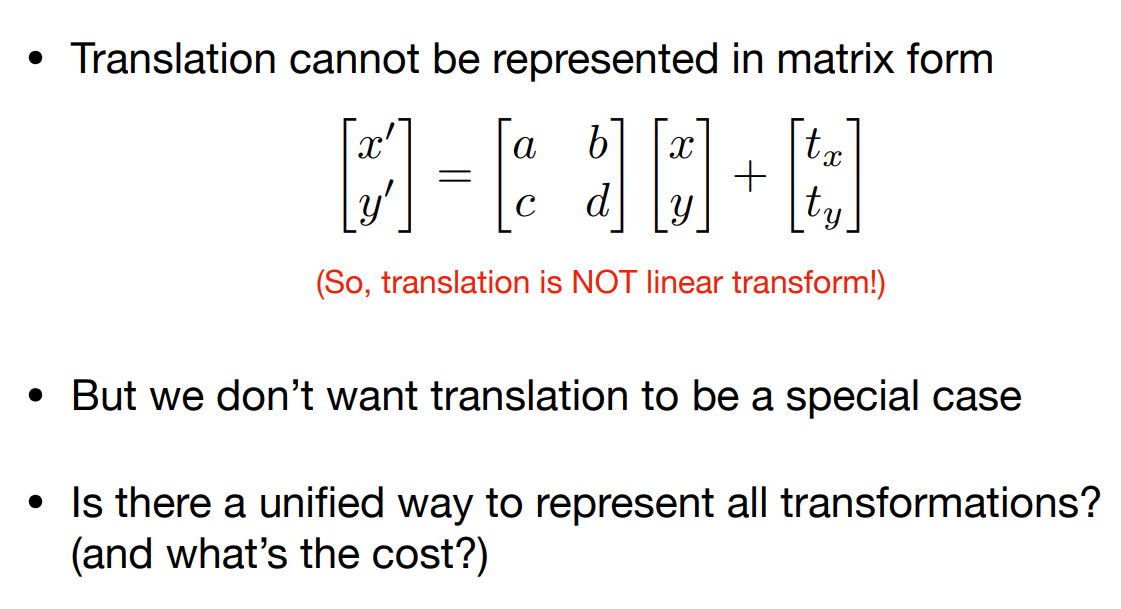
可以用矩阵相乘输入的点得到变换后的点，这种变换叫做线性变换



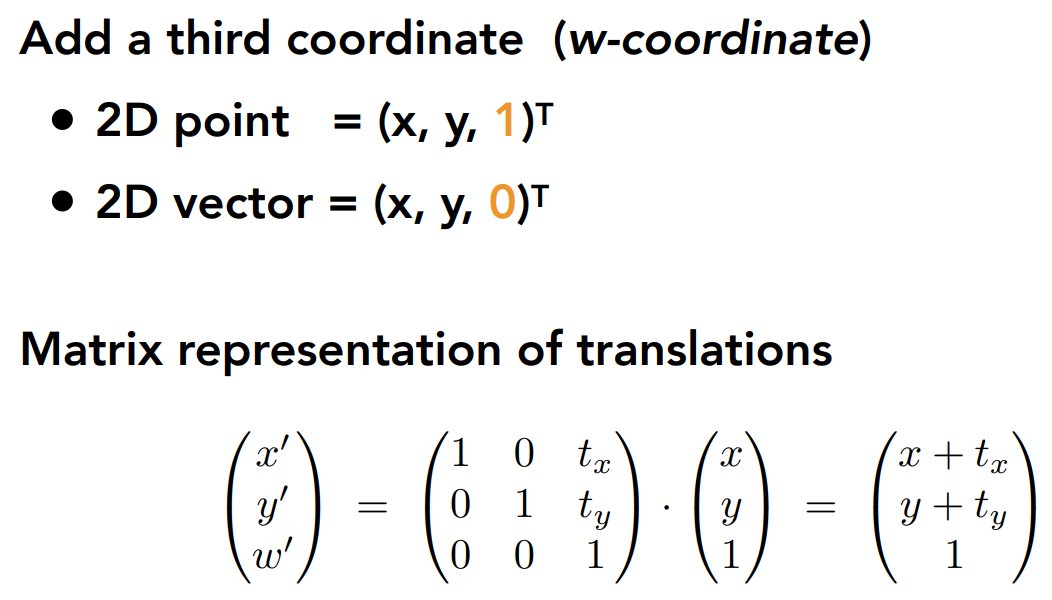
旋转和缩放变换都可以用相同维度的矩阵相乘来表示，平移变换为了便于用矩阵相乘表示引入其次坐标的概念，如下我们把左图平移tx,ty，平移后得到xprime,yprime



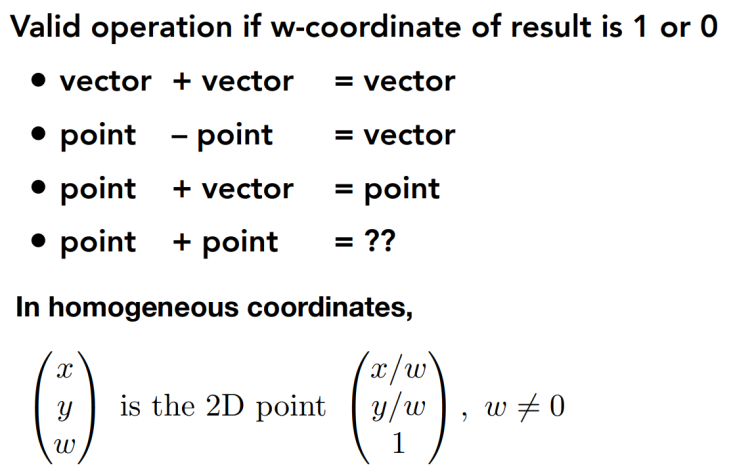
但是想要把变换后的图形写成矩阵相乘的形式（维度不变），就只能写成下面这样，先进行线性变换，然后再平移：



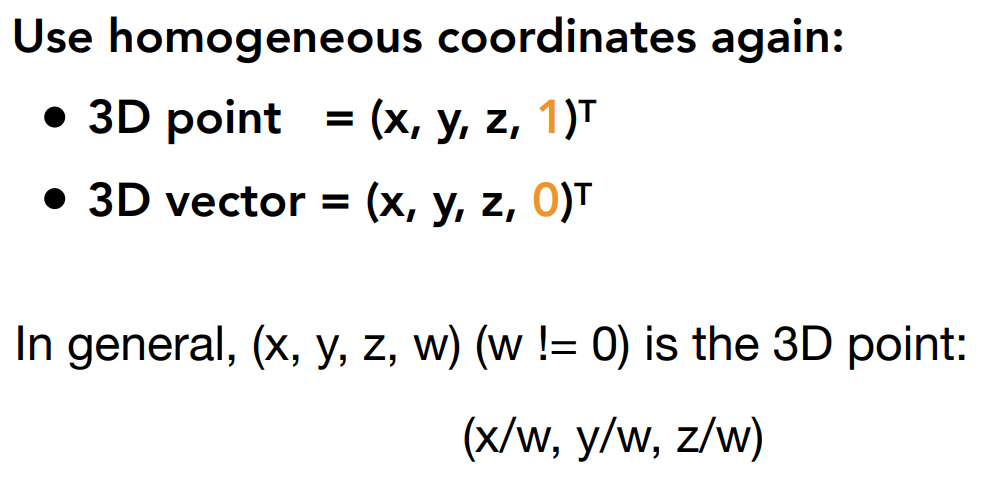
由于不想把平移变换当成特例来处理，那么有没有一个方法能够统一表示所有的变换，统一表示的代价如何（tradeOff）有人提出了解决方法，引入齐次坐标，增加一个维度，同时代价是数据量增加了：



为什么表示点引入的w坐标为1，而表示向量引入的w坐标为0？因为这样表示恰好能够表示向量与点的一些性质，向量加向量结果是向量，点减去另一个点结果是向量，点加向量结果是点而点与点相加其实是没什么的意义的（1+1=2），在其次坐标中的意义是两个点的中点：

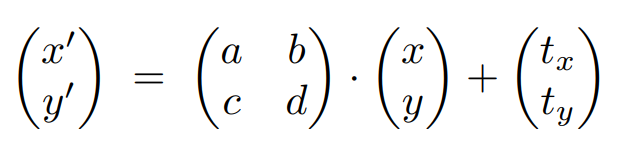


三维变换中也是同理：

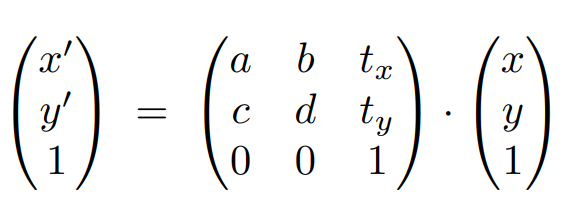


## Affine Transformations

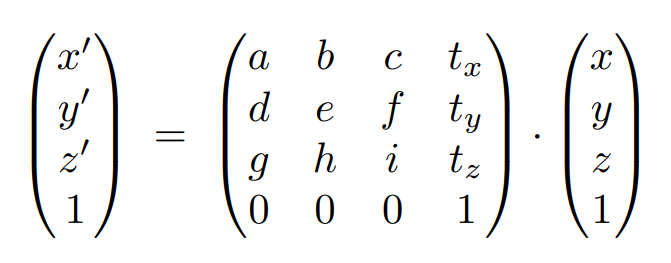
Affine map = linear map + translation



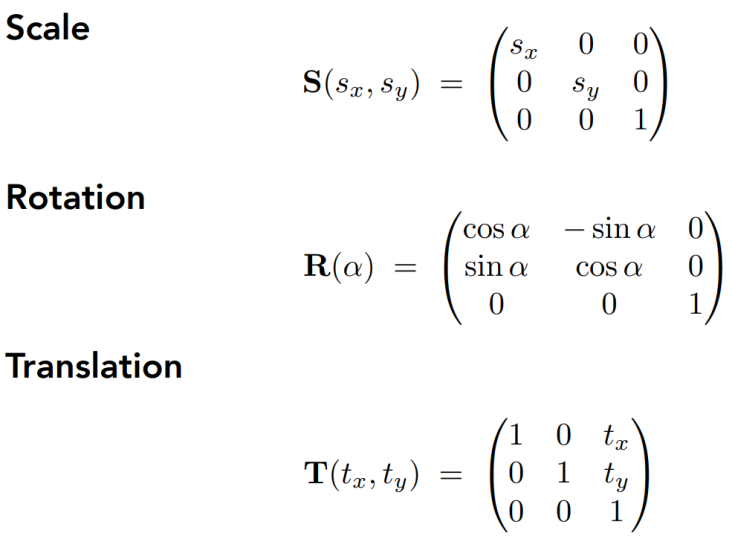
Using homogenous coordinates:



三维空间中的仿射变换也是如此，增加一个维度使用4\*4的矩阵：



在表示二维下的仿射变换矩阵时，最后一行是0，0，1；以下是一些二维变换的矩阵：

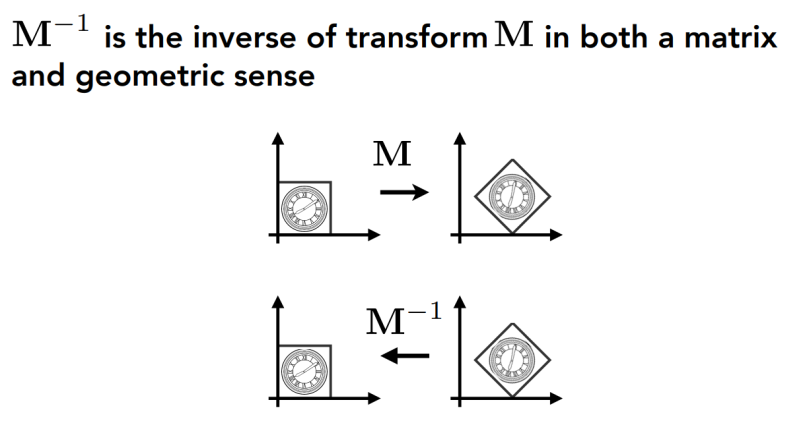


以下是一些三维变换的矩阵：



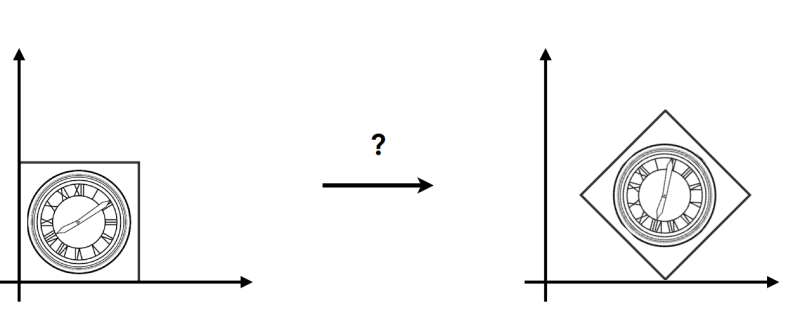
## Inverse Transform

逆矩阵用于表示一个图形的逆变换，假如输入图形A乘矩阵M变换后得到矩阵B,那么矩阵B乘矩阵M的逆矩阵后还原到变换前的矩阵A(做了一个操作之后再做一个反的操作，相当于什么操作都没做)

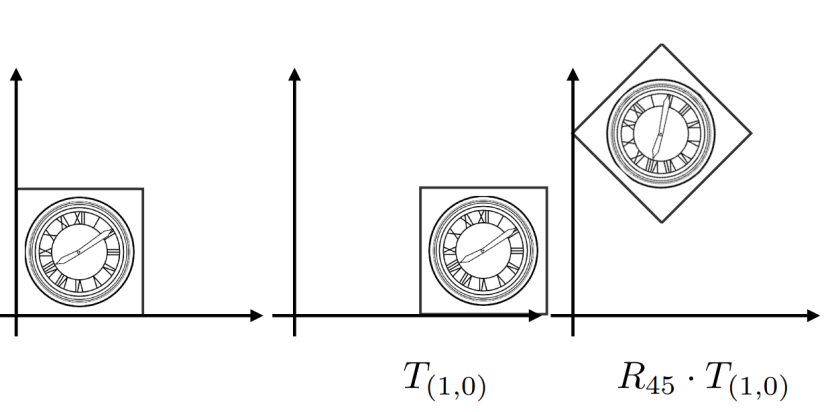


## Composing Transforms

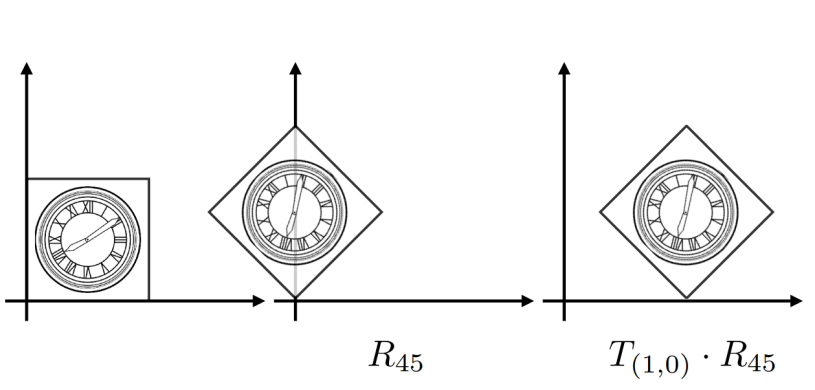
复杂的变换可以通过简单的变换组合而成，在过程中可以发现变换的顺序是不能改变的，如下图，将左边的图形变换到右边的图形，先平移后旋转与先旋转后平移得到的结果不相同：



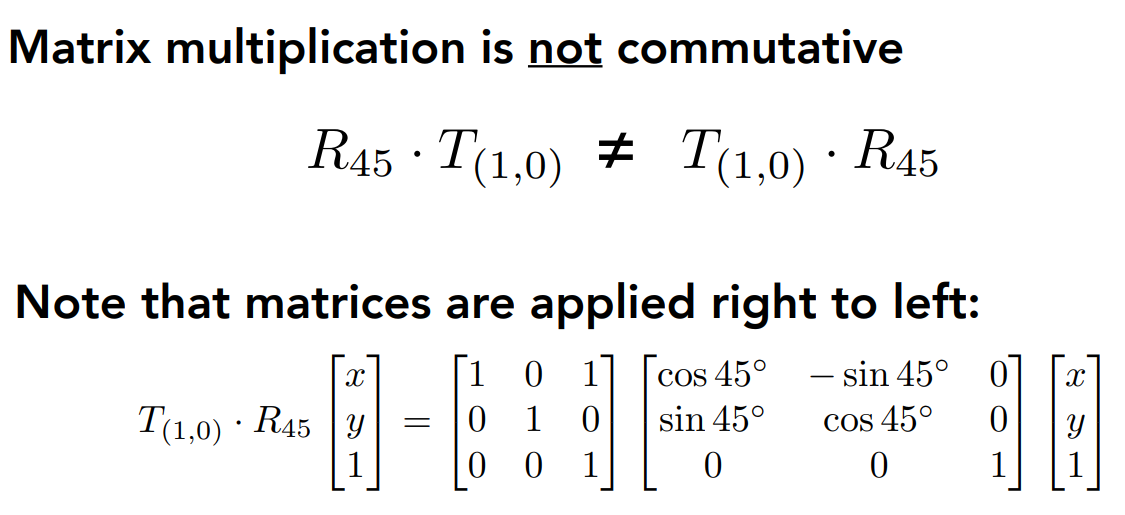
先平移后旋转：



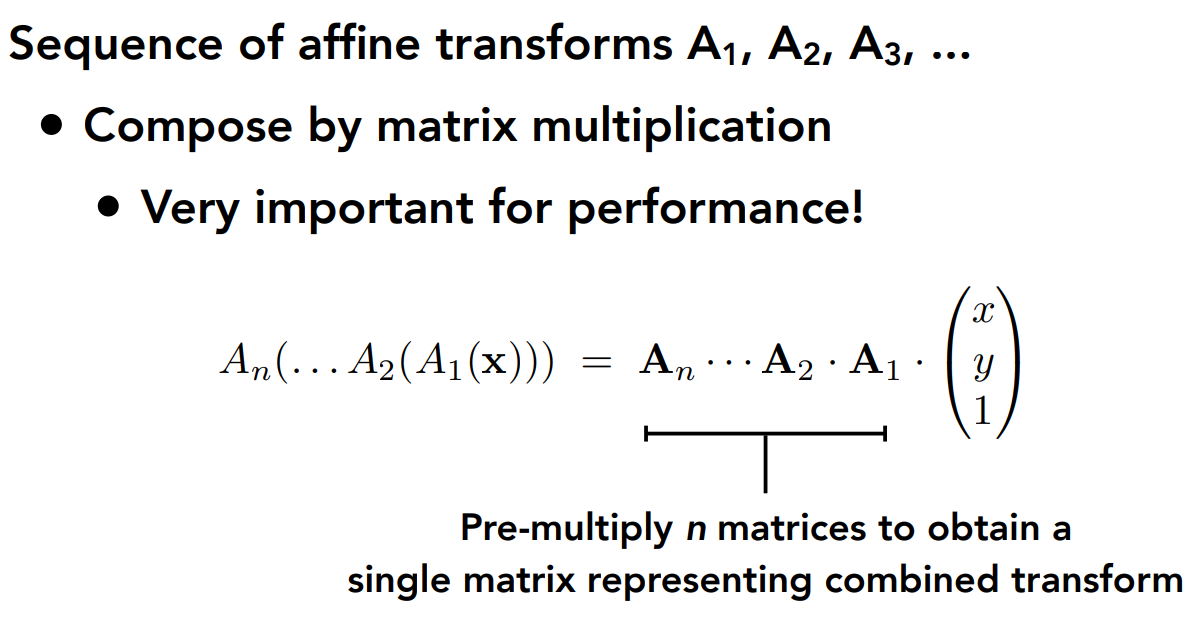
先旋转后平移：



印证了矩阵乘法是不满足交换律的

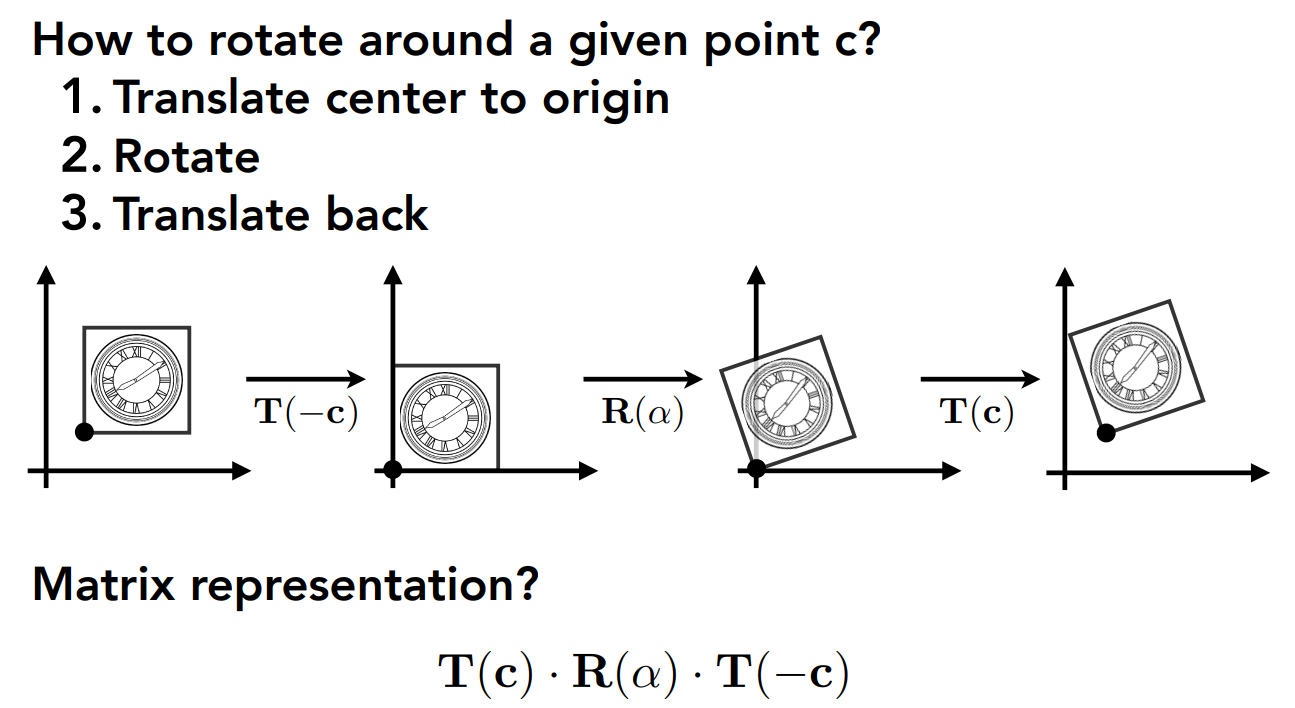


同时组合变换可以使用矩乘法的结合律，可以用一个组合变换的矩阵来表示复杂的变换，这是一个非常好用的性质：

：

## Decomposing Complex Transforms

一般不做特别说明的话，做旋转是围绕（0，0）点进行旋转的，如果想围绕任意一点C进行旋转，我们先把图形平移-C到原点的位置，旋转后再平移C移回去，写成矩阵的话要注意矩阵乘法是从右到左进行应用的：



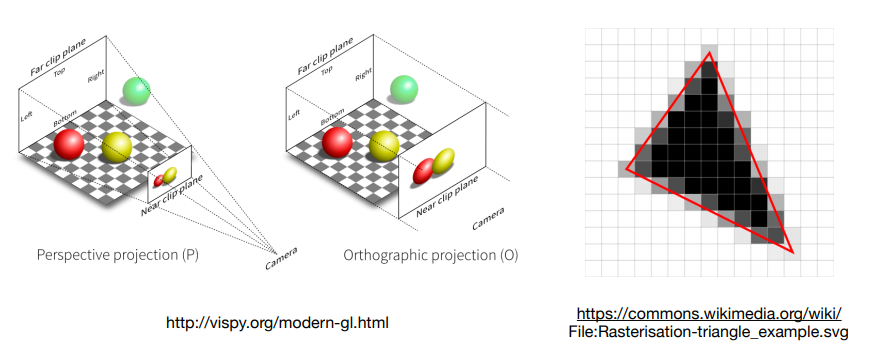
# Rasterization

## OverView

* **把三维空间中的几何形体显示到屏幕中**

光栅化是实时的计算机图形学的主要应用，在计算机图形学中将实时定义为每秒能够生成30幅画面，也叫30帧（30fps）,如果不能达到的话则叫做离线

* **将原物体投影映射到像素**
* **游戏的黄金标准**



光栅化过程中三角形遍历的时候可以应用叉乘来判断像素的中心点是否在三角形内，例如图1中的红点（在三角形内）和白点（在三角形外），

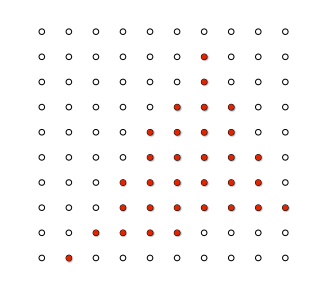


图1

然后把对应的像素进行着色，就会得到如图2这样的类似三角形的图案，有很多的锯齿，和期望得到的图3

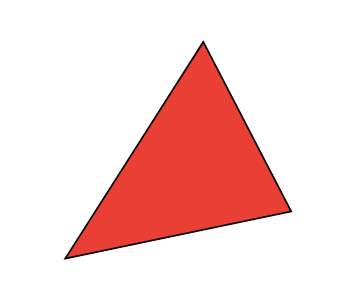
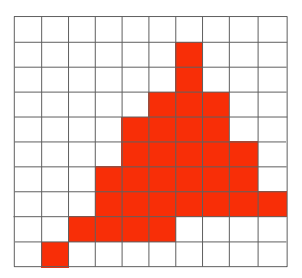
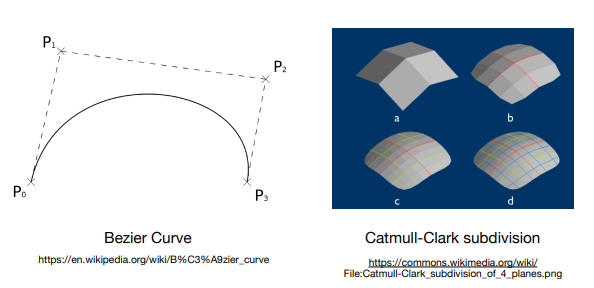


图2 图3

# Curves and Meshes

## OverView

如何表示一条光滑的曲线、如何表示曲面、如何用简单的曲面通过细分来得到一些更复杂的曲面，在形状发生变化的时候这些面要如何变化，如何保持住物体的拓扑结构



# Ray Tracing

## OverView

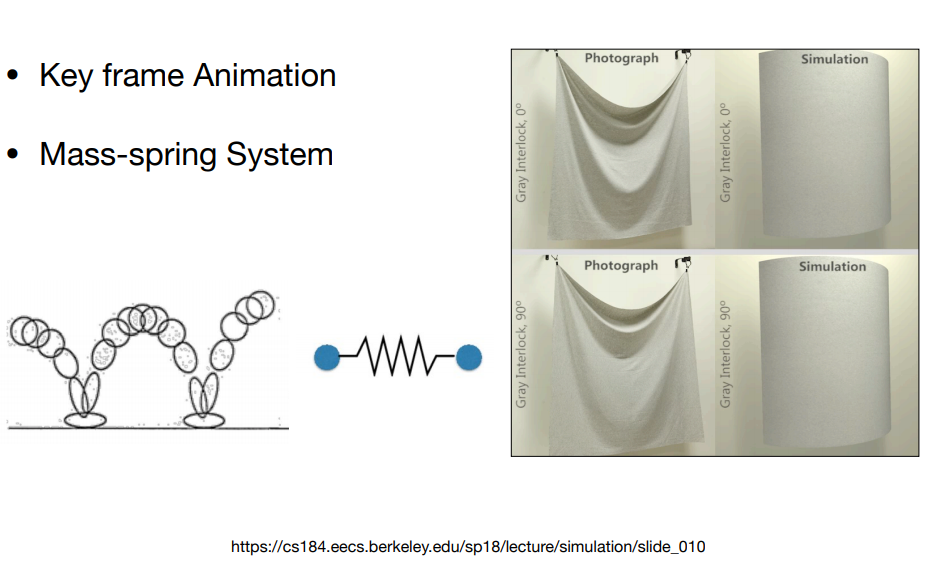
图形学乃至是生活中有很多事情其实可以看作tradeoff，在为了达成某一个目标不得不牺牲一些其他的目标，有一个取舍与权衡的过程，光线追踪能够生产非常真实的画面，如下右图，但是速度慢，在动画和电影中使用较多。



# Animation / Simulation

## OverView

动画与模拟仿真，一个弹性小球落在地上，与地面进行挤压发生形变弹上去再下来，布料自然下垂等



# HDR&LDR