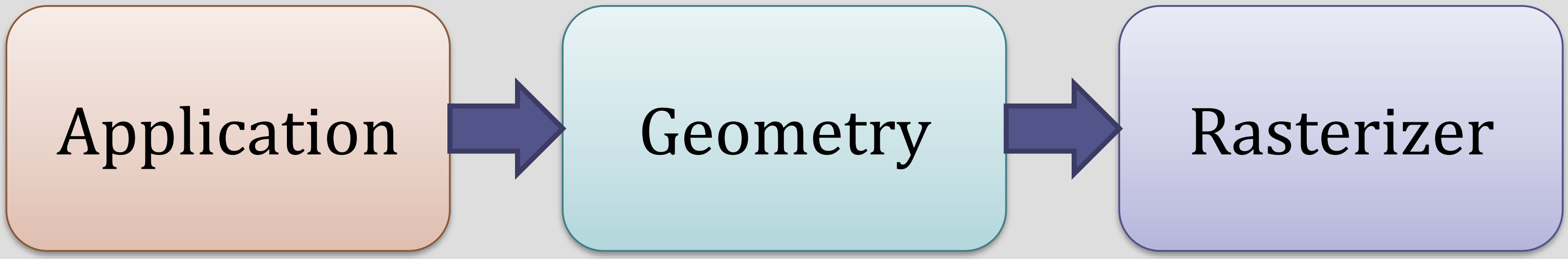
# Coordinate Spaces

Jan Ondrej
Senior Research Fellow

# Graphics Pipeline Overview

- Coarse Division
- Each stage is a pipeline in itself

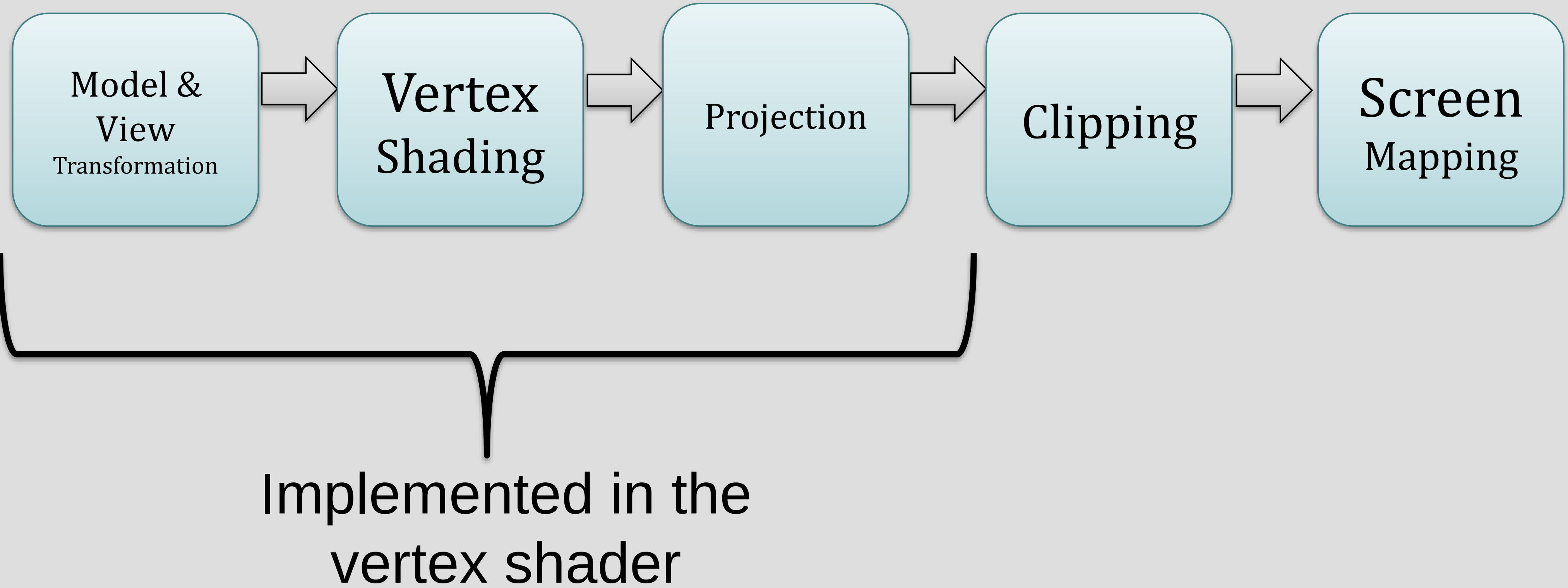



- The slowest pipeline stage determines the *rendering speed (fps)*

*Real-Time Rendering, 3rd Edition, Akenine-Moller*

# The Geometry Stage

- Responsible for the per-polygon and per-vertex operations



# Model Space

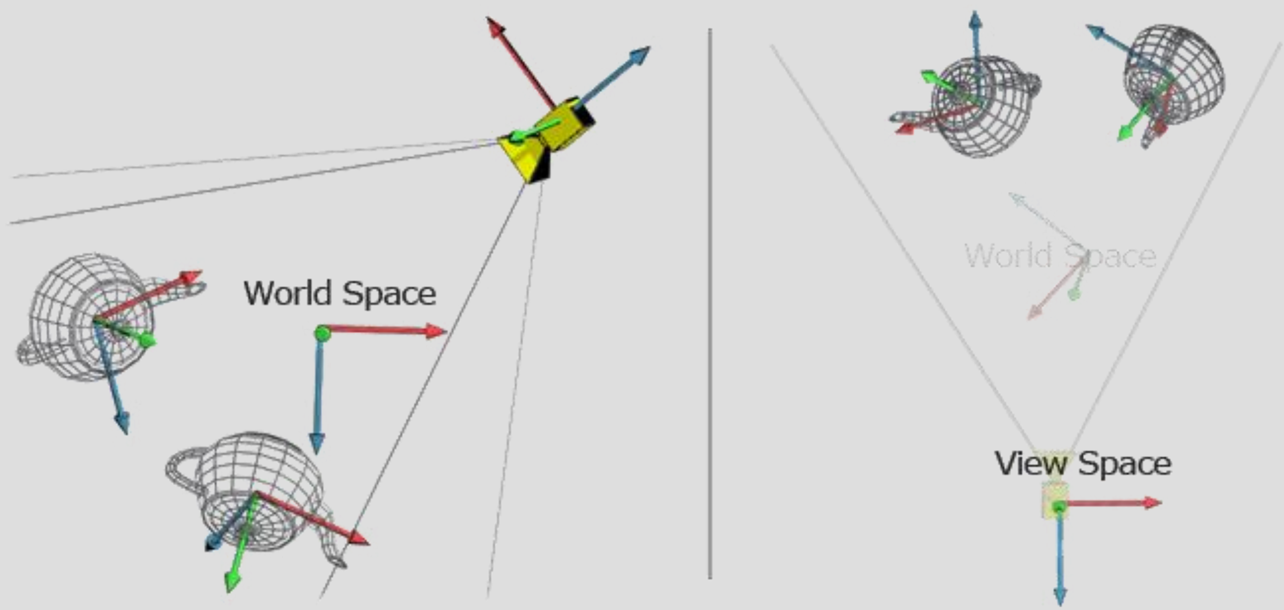
- 3ds Max, Maya, Softimage, Blender, Auto CAD etc.
- Vertices specified relative to a Cartesian coordinate system called model space
- Origin usually in centre or at feet of the character

# World Space

- *World space* is a fixed coordinate space, in which positions, orientations, scales of all objects in the game world are expressed
- Ties all objects together into a cohesive virtual world
- Right hand system in OpenGL
- Origin at the centre of the screen

# View Space

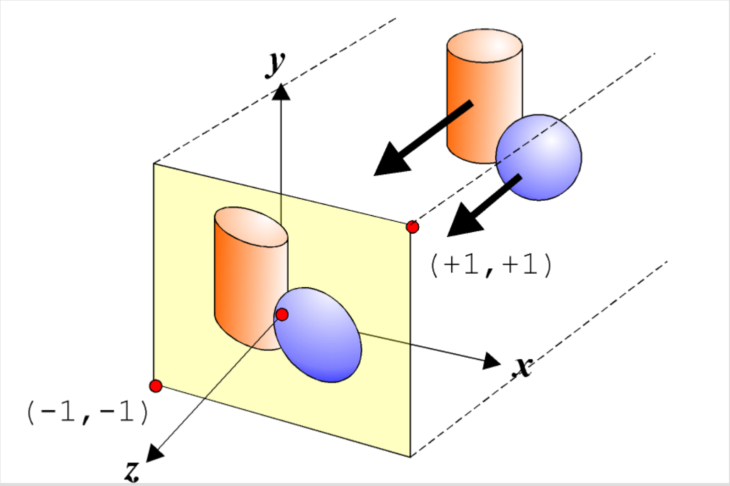
- A coordinate frame fixed to the camera
- Origin is placed at the focal point o the camera
- Right hand system in OpenGL



Projection

# Projection

- After shading, rendering systems perform *projection*
- Models are projected from three to two dimensions
- *Perspective* or *orthographic* viewing



# The Old Vertex Shader

```
in vec4 vPosition;

void main () {
  // The value of vPosition should be between -1.0 and +1.0
  gl_Position = vPosition;
}
----------------------------------------
out vec4 fColor ;

void main () {
  // No matter what, color the pixel red!
  fColor = vec4 (1.0, 0.0, 0.0, 1.0);
}
```

# A Better Vertex Shader

```
in vec4 vPosition; // the vertex in local coordinate system
uniform mat4 mM; // the matrix for the pose of the model
uniform mat4 mV; // The matrix for the pose of the camera
uniform mat4 mP; // The projection matrix (perspective)


void main () {
   // The value of vPosition should be between -1.0 and +1.0
   gl_Position = mP * mV * mM * vPosition;
}
```

New position in NDC

Original (local) position

# Geometric Transformations

# Objectives

- Learn how to carry out transformations
  - Rotation
  - Translation
  - Scaling
  - Combinations!

# Computer Graphics Problems

- Much of graphics concerns itself with the problem of **displaying** 3D objects in 2D screen
- We want to be able to:
  - rotate, translate, scale our objects
  - view them from arbitrary points of view
  - view them in perspective
- Want to display objects in coordinate systems that are convenient for us and to be able to reuse object descriptions

# Matrices

- Matrix addition
  - $\begin{bmatrix} a & c \\ b & d \end{bmatrix} + \begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} a+e & c+g \\ b+f & d+h \end{bmatrix}$
- Matrix multiplication
  - $\begin{bmatrix} a & c \\ b & d \end{bmatrix} \times \begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} ae+cf & ag+ch \\ be+df & bg+dh \end{bmatrix}$
  - Not commutative in most cases
    - **AB** ≠ **BA**
    - If **AB** = **AC**, it does not necessarily follow that **B** = **C**
  - It is associative and distributive
    - (AB)C = A(BC)
    - A(B+C) = AB + AC
    - (A+B)C = AC + BC
  - Transpose $A^T$ of a matrix A is one whose rows are switched with its columns

# Question

- $A = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$
- $B = \begin{bmatrix} 1 & -4 \\ 3 & 2 \end{bmatrix}$

- What is $A+B^T$?

# Answer

- $A = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$
- $B = \begin{bmatrix} 1 & -4 \\ 3 & 2 \end{bmatrix}$

- What is $A+B^T$?
  - $AB^T = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ -4 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 5 \\ -2 & 4 \end{bmatrix}$

# Matrices

- If you need to rotate a million vertices representing a dinosaur object about some axis, you don't need to multiply each point by 5 different matrices

- You simply multiply the 5 matrices together once and multiply each dinosaur point by that one matrix. Huge saving!

# Homogeneous Coordinates

- Basis of the homogeneous co-ordinate system is the set of *n* basis vectors and the origin position:

$$\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \text{ and } P_o$$

- All points and vectors are therefore compactly represented using their ordinates:

$$\begin{bmatrix} a_1 \\ \vdots \\ a_n \\ a_o \end{bmatrix} \text{or more usually} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

# Homogeneous Coordinates

- Vectors have no positional information and are represented using $a_o = 0$ whereas points are represented with $a_o = 1$:

$$\vec{v} = a_1\mathbf{v}_1 + \cdots + a_n\mathbf{v}_n + 0$$

$$P = a_1\mathbf{v}_1 + \cdots + a_n\mathbf{v}_n + P_o$$

- <u>Examples</u>:

$$\begin{bmatrix} 0.2 \\ 1.3 \\ 2.2 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 1 \end{bmatrix} \qquad\qquad \begin{bmatrix} 0.2 \\ 1.3 \\ 2.2 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 0 \end{bmatrix}$$

Points

Associated vectors

# Homogeneous Coordinates

Using this scheme,
every rotation, translation, and scaling operation can be represented by a matrix multiplication, and any combination of the operations corresponds to the products of the corresponding matrices

- Using homogeneous co-ordinates allows us to treat translation in the same way as rotation and scaling

# Translation

- Simplest of the operations
  - Add a positive number – moves to the right
  - Add a negative number – moves to the left
- Addition of constant values, causes uniform translations in those directions
- Translations are independent and can be performed in any order (including all at once)
  - Object moved one unit to the right then up
  - Same as if moved one unit up and to the right
  - Net result is motion of sqrt(2) units to the upper-right

# Translation

**Definition (Translation)**

A translation is a displacement in a particular direction

- A translation is defined by specifying the displacements a, b, and c

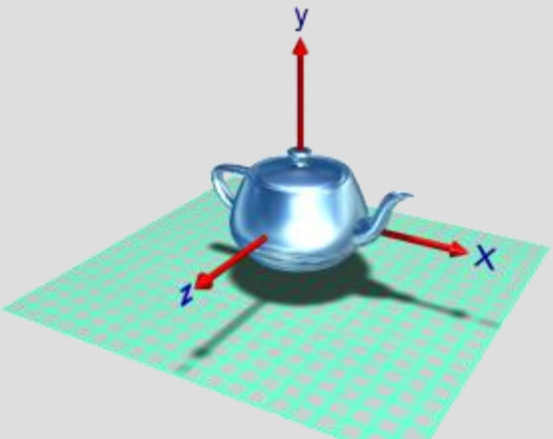
$$x' = x + a$$
$$y' = y + b$$
$$z' = z + c$$

# Translation

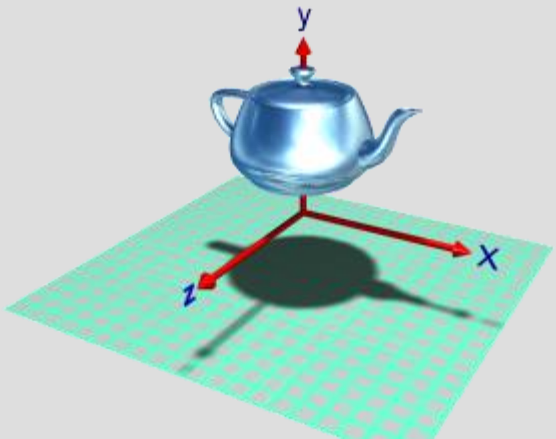
- Translation *only applies to points*, we never translate vectors.
- <u>Remember</u>: points have homogeneous co-ordinate w = *1*

$$x' = x + a$$
$$y' = y + b$$
$$z' = z + c$$

$$\Rightarrow \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$




translate along ***y***



# Scaling

- What if we want to make things larger or smaller?
- Have a car model
  - Want one 3 times smaller!

# Scaling

**Definition (Scaling)**

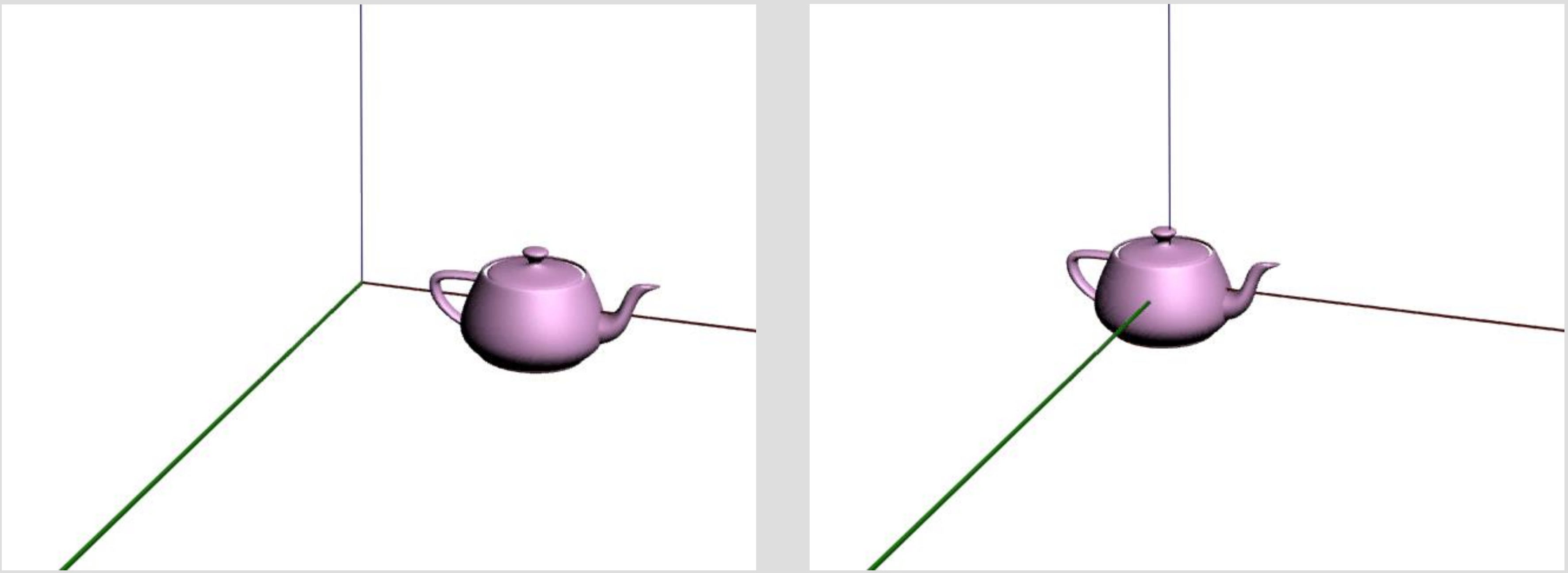
A scaling is an expansion or contraction in the x, y, and z directions by scale factors sx, sy, sz and centred at the point (a,b, c)

- Generally we centre the scaling at the origin

$$x' = s_x x$$

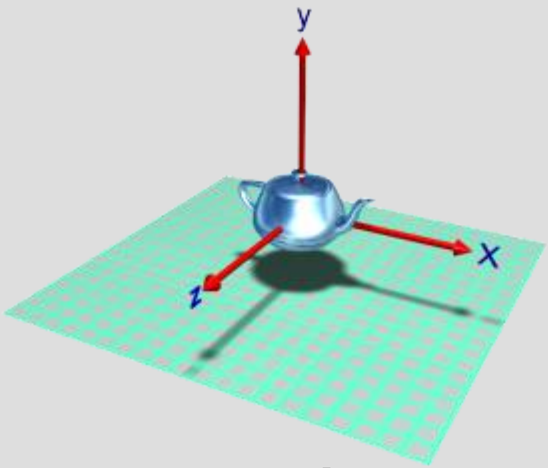
$$y' = s_y y$$

$$z' = s_z z$$

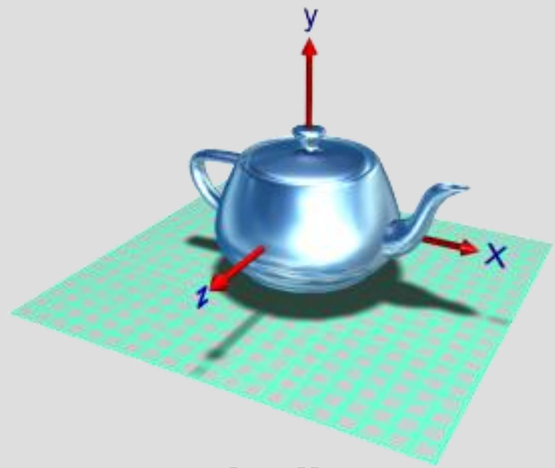


# Scale

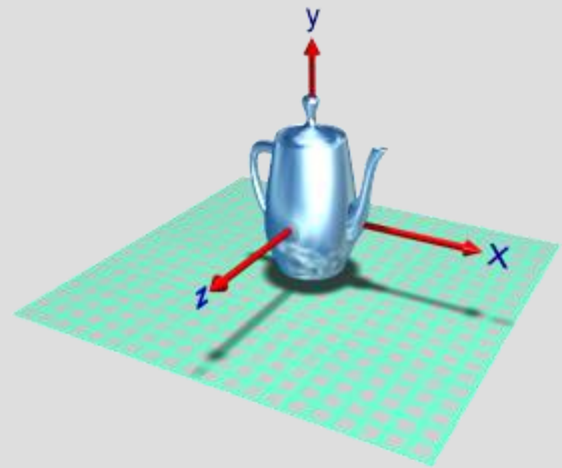
- all vectors are scaled from the origin:



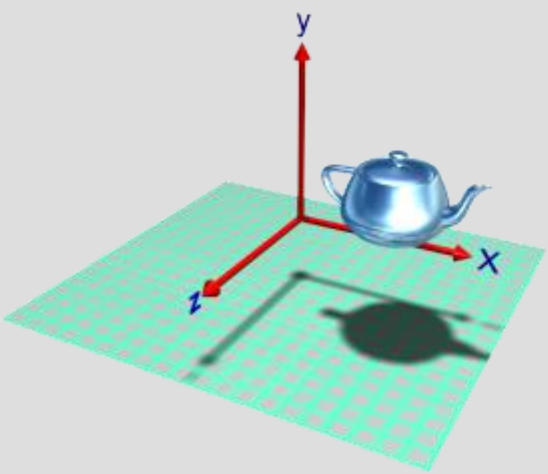
Original



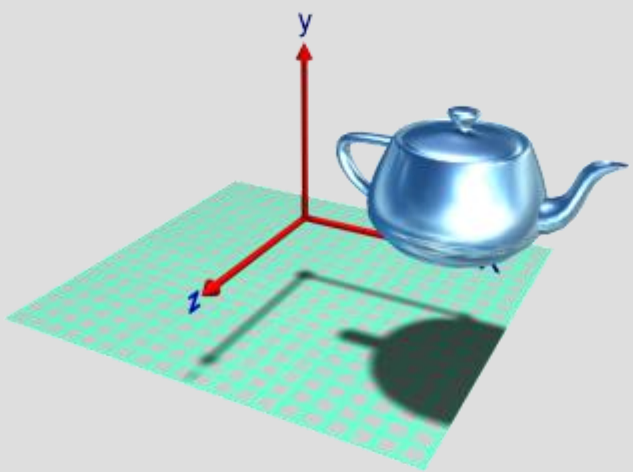
scale all axes



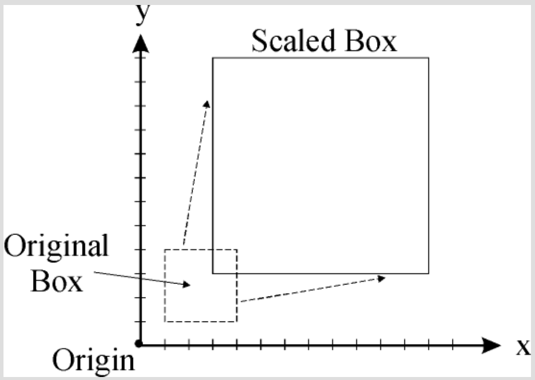
scale Y axis



offset from origin



distance from origin also scales



# Scale

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \Rightarrow \quad \mathbf{v}' = \mathbf{S}\mathbf{v}$$

We would also like to scale points thus we need a *homogeneous transformation* for consistency:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ w \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

$$\mathbf{S}^{-1} = \begin{bmatrix} {}^{1}\!/_{s_x} & 0 & 0 & 0 \\ 0 & {}^{1}\!/_{s_y} & 0 & 0 \\ 0 & 0 & {}^{1}\!/_{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Non-Uniform Scaling

- Make an object twice as big in the x-direction
  - Multiply all x-coordinates by 2, leave y&z unchanged
- 3 times as large in the y-direction
  - Multiply all y-coordinates by 3, leave z&x unchanged

# Rotation

**Definition (Rotation)**

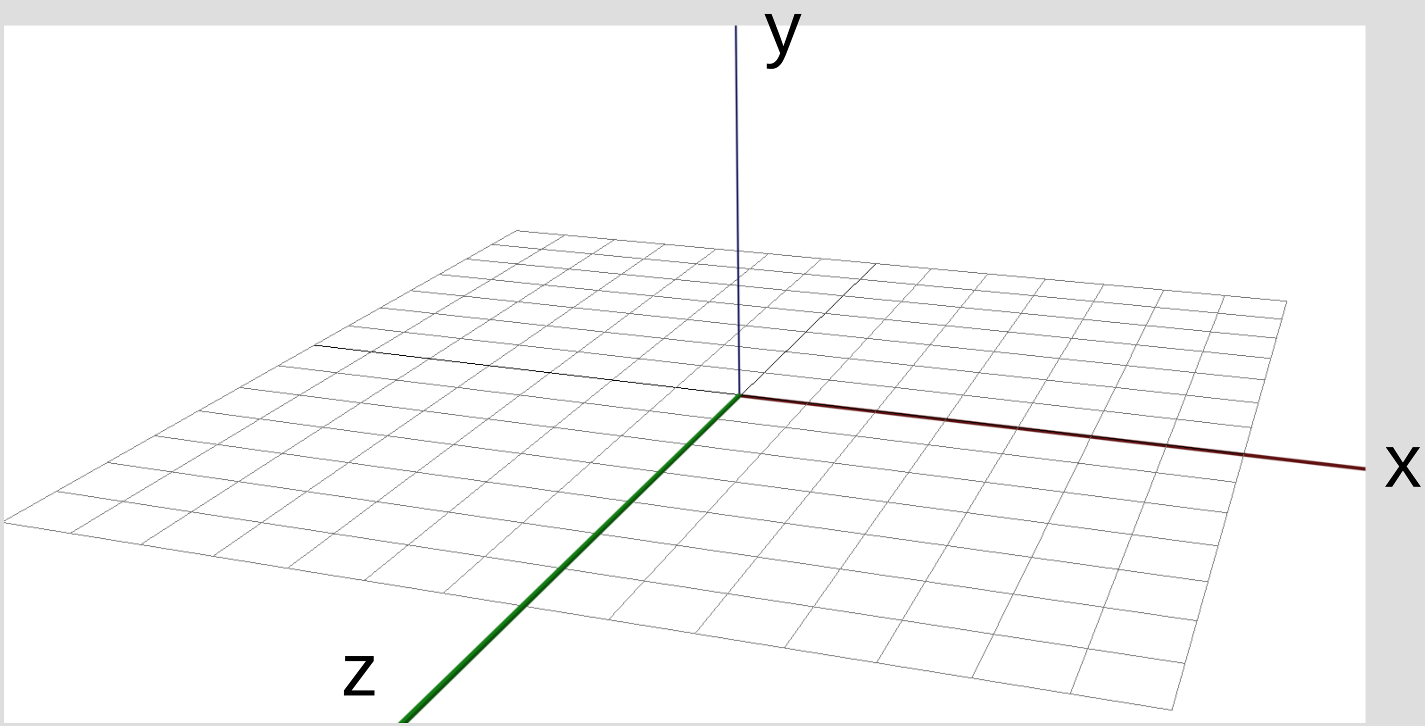
A rotation turns about a point (a,b) through an angle θ

- Generally, we rotate about the origin
- Using the z-axis as the axis of rotation, the equations are:

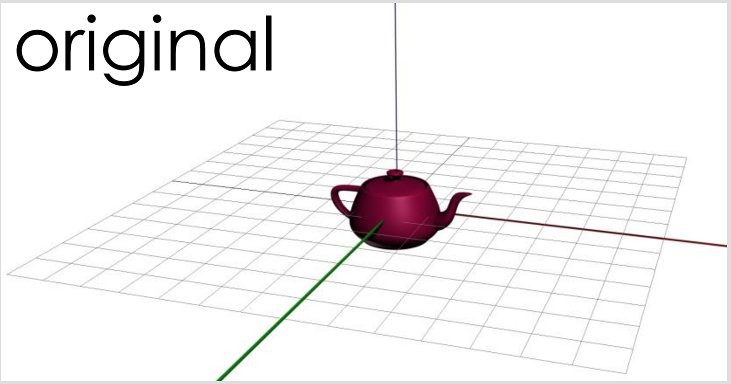
$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$
$$z' = z$$

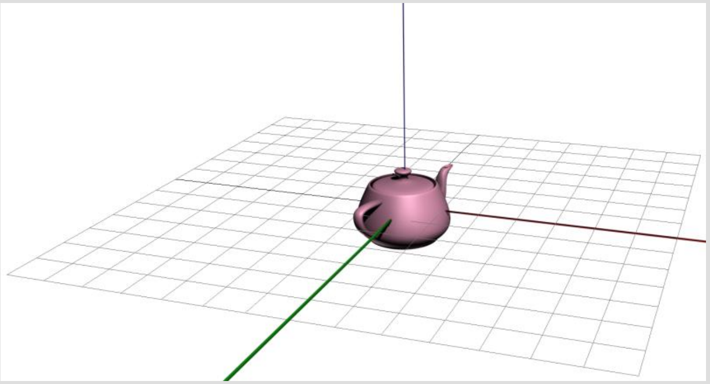
# Rotation - idea

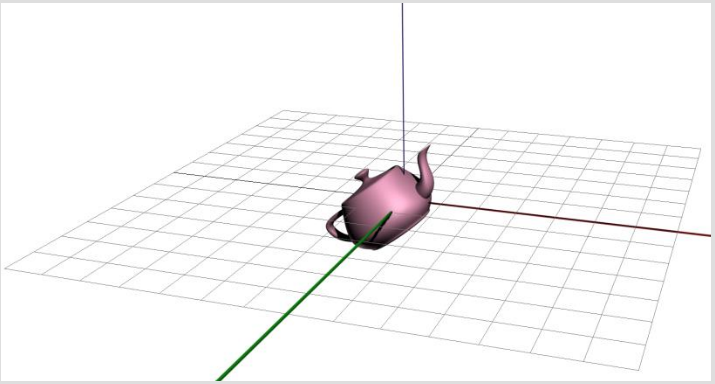
- Visualise rotation about an axis:
  - Put your eye on that axis in the positive direction and look towards the origin
  - Then, a positive rotation corresponds to a counter-clockwise rotation

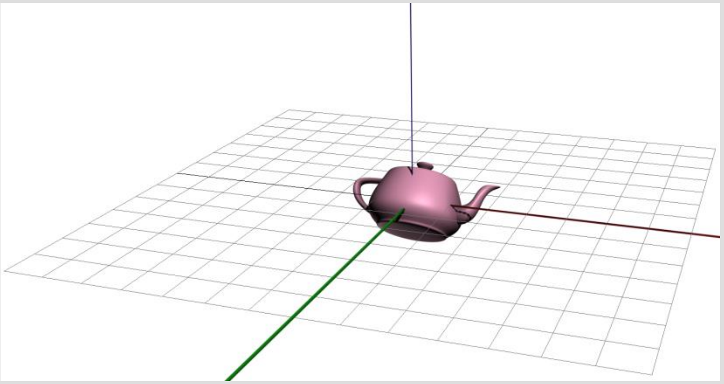


# Which Axis?



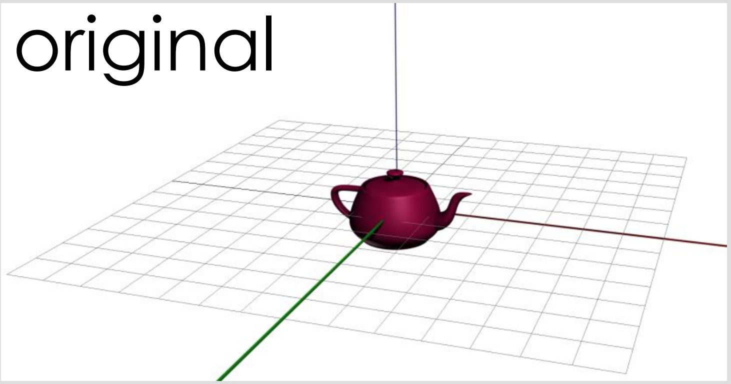





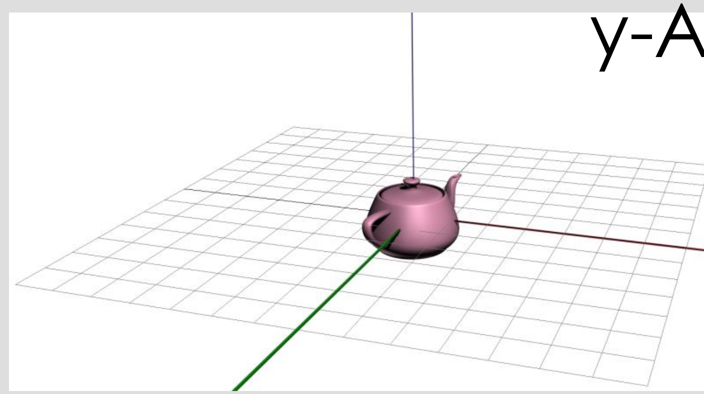


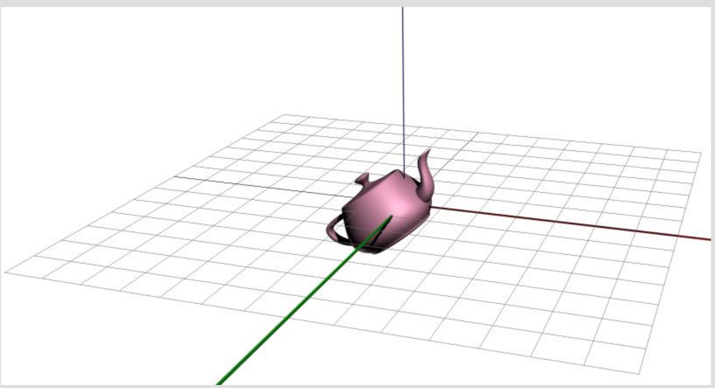
# Which Axis?

original

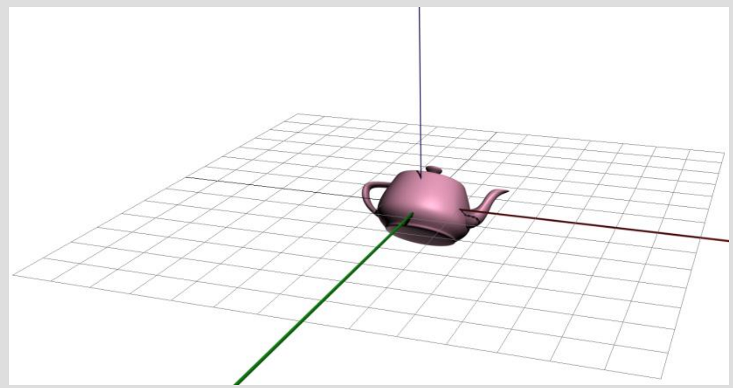


y-Axis





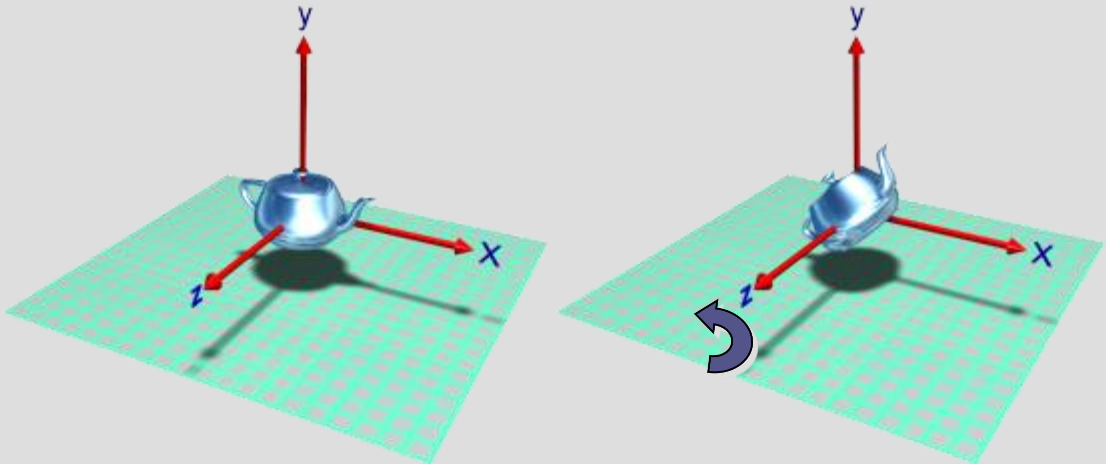
z-axis



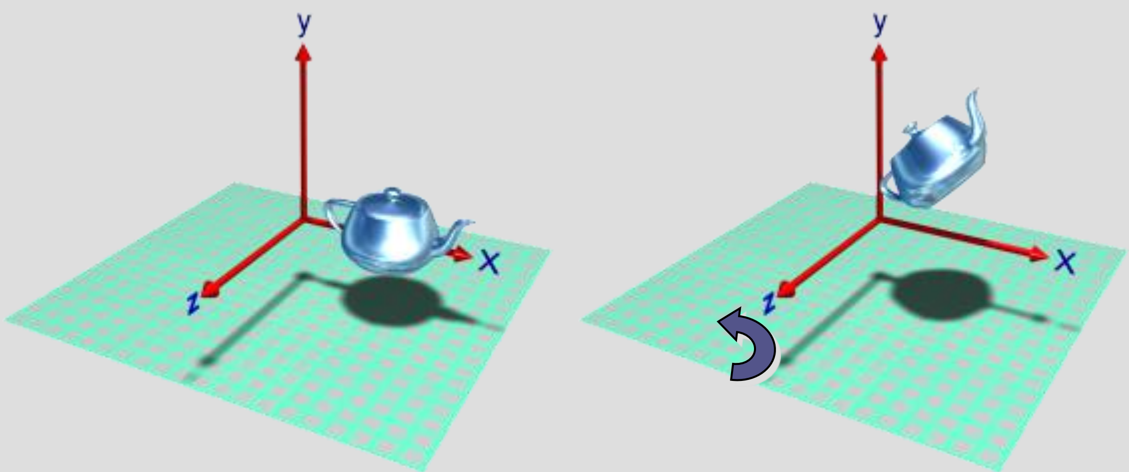
(-) x-axis

# Rotation

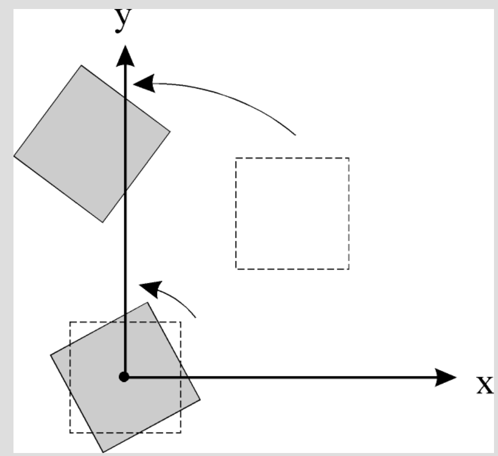
- Rotations are *anti-clockwise* about the *origin:*




rotation of 45$^o$ about the Z axis

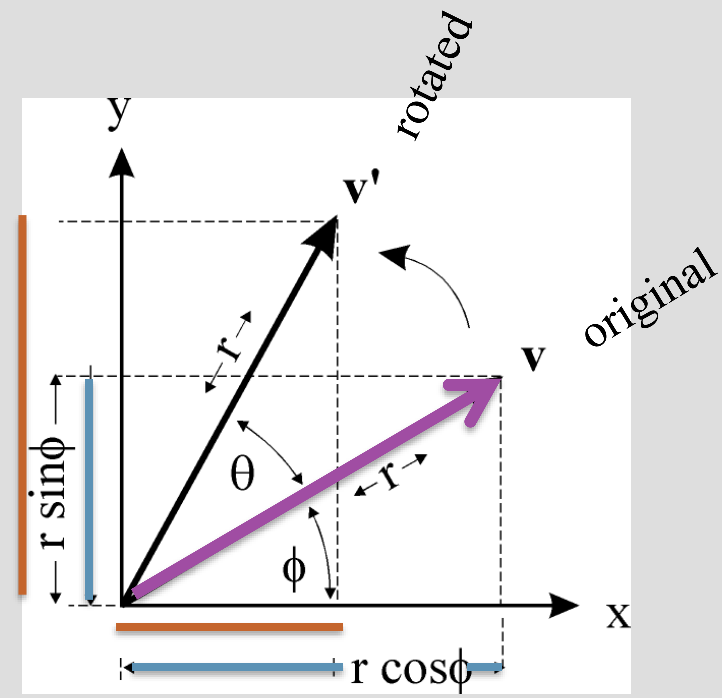



offset from origin rotation



# Rotation about the z-axis

$$\mathbf{v} = \begin{bmatrix} r\cos\phi \\ r\sin\phi \end{bmatrix} \qquad \mathbf{v}' =$$




$$\text{expand}\,(\phi+\theta) \Rightarrow \begin{cases} x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ y' = r\cos\phi\sin\theta + r\sin\phi\cos\theta \end{cases}$$

$$\text{but} \quad \begin{matrix} \underline{x = r\cos\phi} \\ \underline{y = r\sin\phi} \end{matrix} \Rightarrow \begin{matrix} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{matrix}$$

# Rotation about the z-axis

- Rotation in the clockwise direction is the inverse of rotation in the counter-clockwise direction and vice versa

# Rotation

- 2D rotation of θ about origin: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

- 3D homogeneous rotations:

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

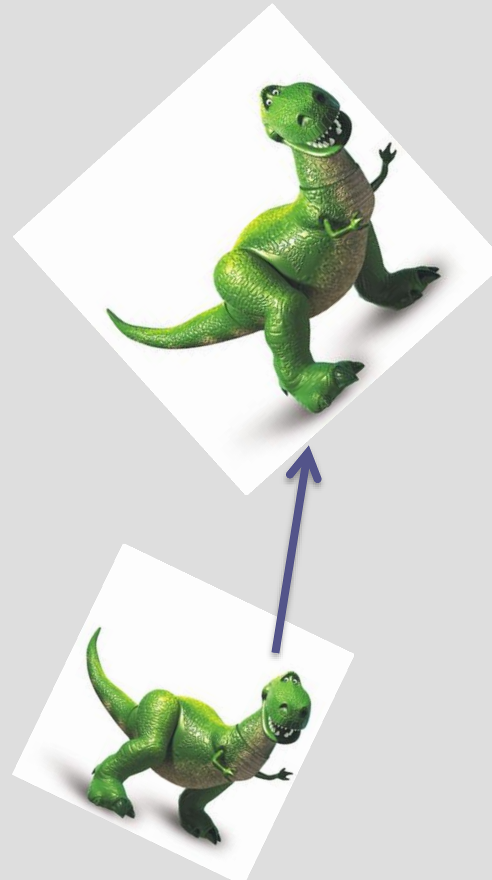
- If $\mathbf{M}^{-1} = \mathbf{M}^{T}$ then **M** is *orthonormal*. All orthonormal matrices are rotations about the origin.

# Combining Rotation, Translation, & Scaling

- Often advantageous to **combine** various transformations to form a more complex transformation

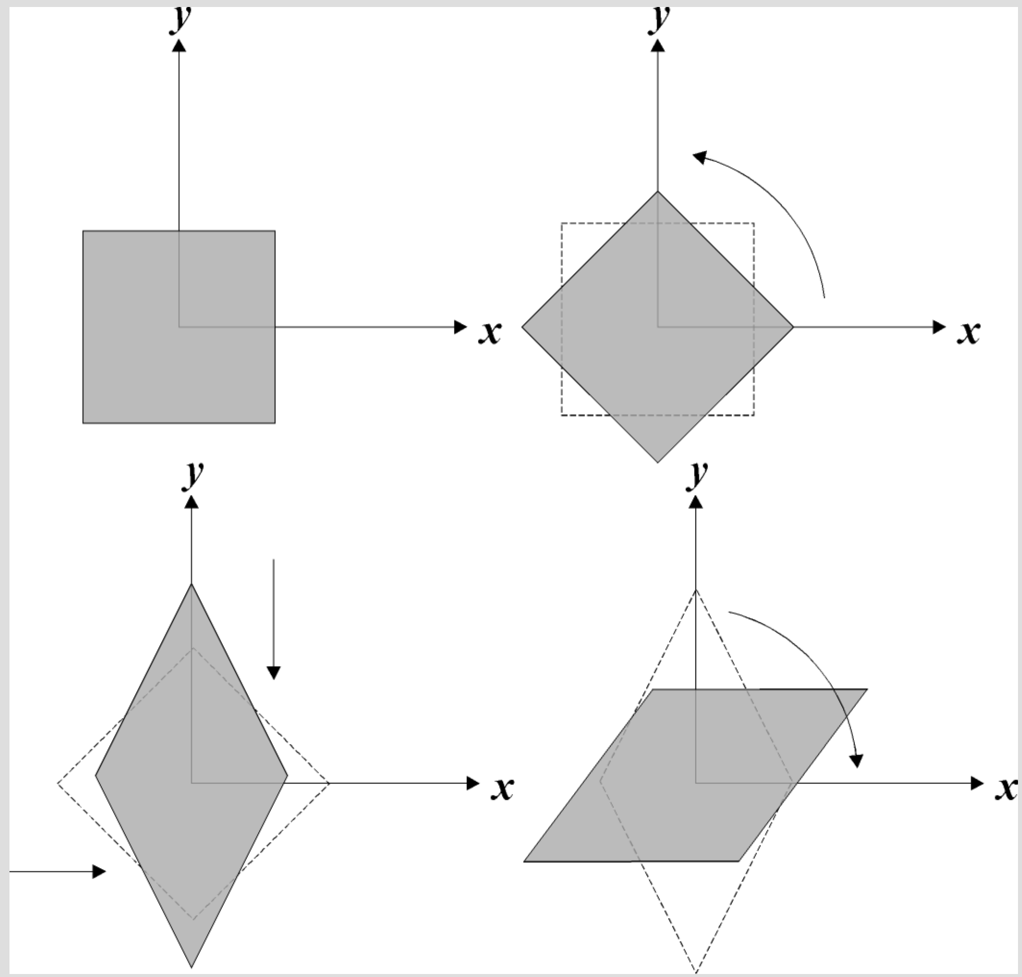
- If we do the algebra – things get complicated quickly

- Easier method – **matrices**



# Affine Transformations

- All *affine transformations* are combinations of rotations, scaling and translations.



# Homogenous Coordinates

Using this scheme,
every rotation, translation, and scaling operation can be represented by a matrix multiplication,
and **any combination** of the operations corresponds to the products of the corresponding matrices

# Transformation Composition

- It is common for graphics programs to apply more than one transformation to an object
  - Take vector $v_1$, Scale it (**S**), then rotate it (**R**)
  - First, $v_2 = \mathbf{S}v_1$, then, $v_3 = \mathbf{R}v_2$
  - $v_3 = \mathbf{R}(\mathbf{S}v_1)$
  - Since matrix multiplication is associative: $v_3 = (\mathbf{RS})v_1$
- In other words, we can represent the effects of transforms by two matrices in a single matrix of the same size by multiplying the two matrices: $\mathbf{M} = \mathbf{RS}$
- **NB:** transforms are applied from the right side first
  - Matrix multiplication is not commutative
  - Order matters!
    - Scaling then rotating is usually different than rotating then scaling

# Transformation Composition

- More complex transformations can be created by *concatenating* or *composing* individual transformations together.

$$\mathbf{M} = \mathbf{T} \circ \mathbf{R} \circ \mathbf{S} \circ \mathbf{T} = \mathbf{TRST} \qquad \mathbf{v}' = \mathbf{T}[\mathbf{R}[\mathbf{S}[\mathbf{T}\mathbf{v}]]] = \mathbf{M}\mathbf{v}$$

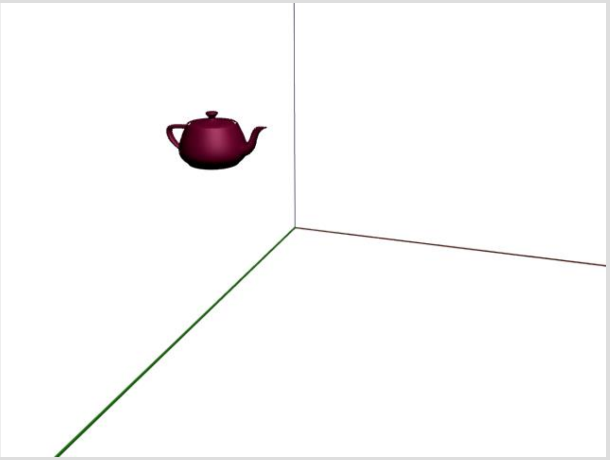
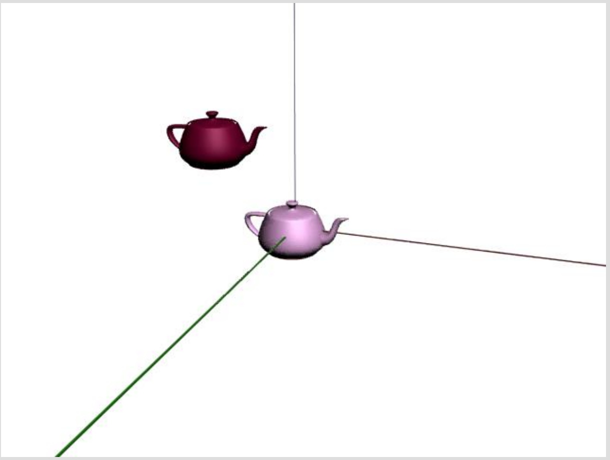
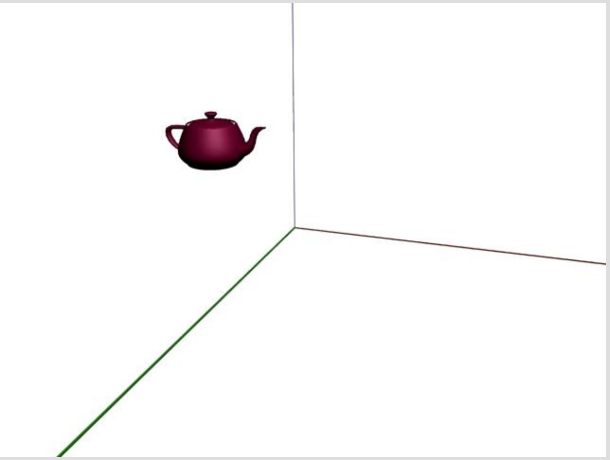
- Matrix multiplication is *non-commutative* $\Rightarrow$ **order is vital**
- We can create an affine transformation representing rotation about a point $P_R$:

$$\mathbf{M} = \mathbf{T}(P_R)\mathbf{R}(\theta)\mathbf{T}(-P_R)$$

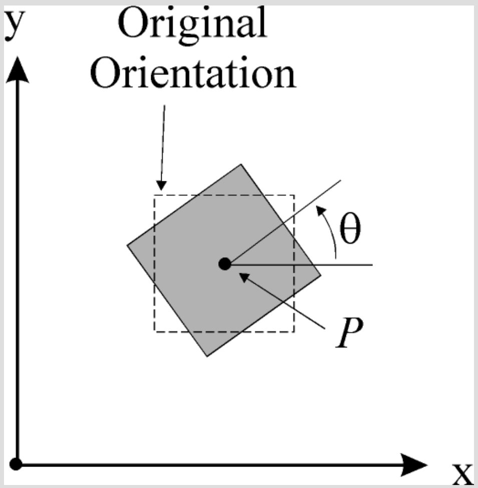
= *translate to origin, rotate about origin, translate back to original location*

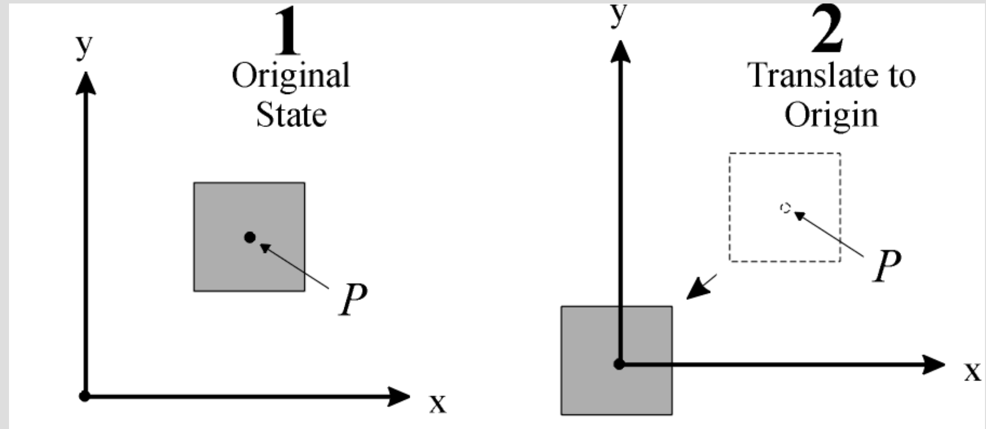
# Rotation about a point

- What if rotation is not about the origin?
  - Translate the centre of rotation to the origin,
  - Perform the rotation
  - Translate back

# Transformation Composition



# Transformation Composition

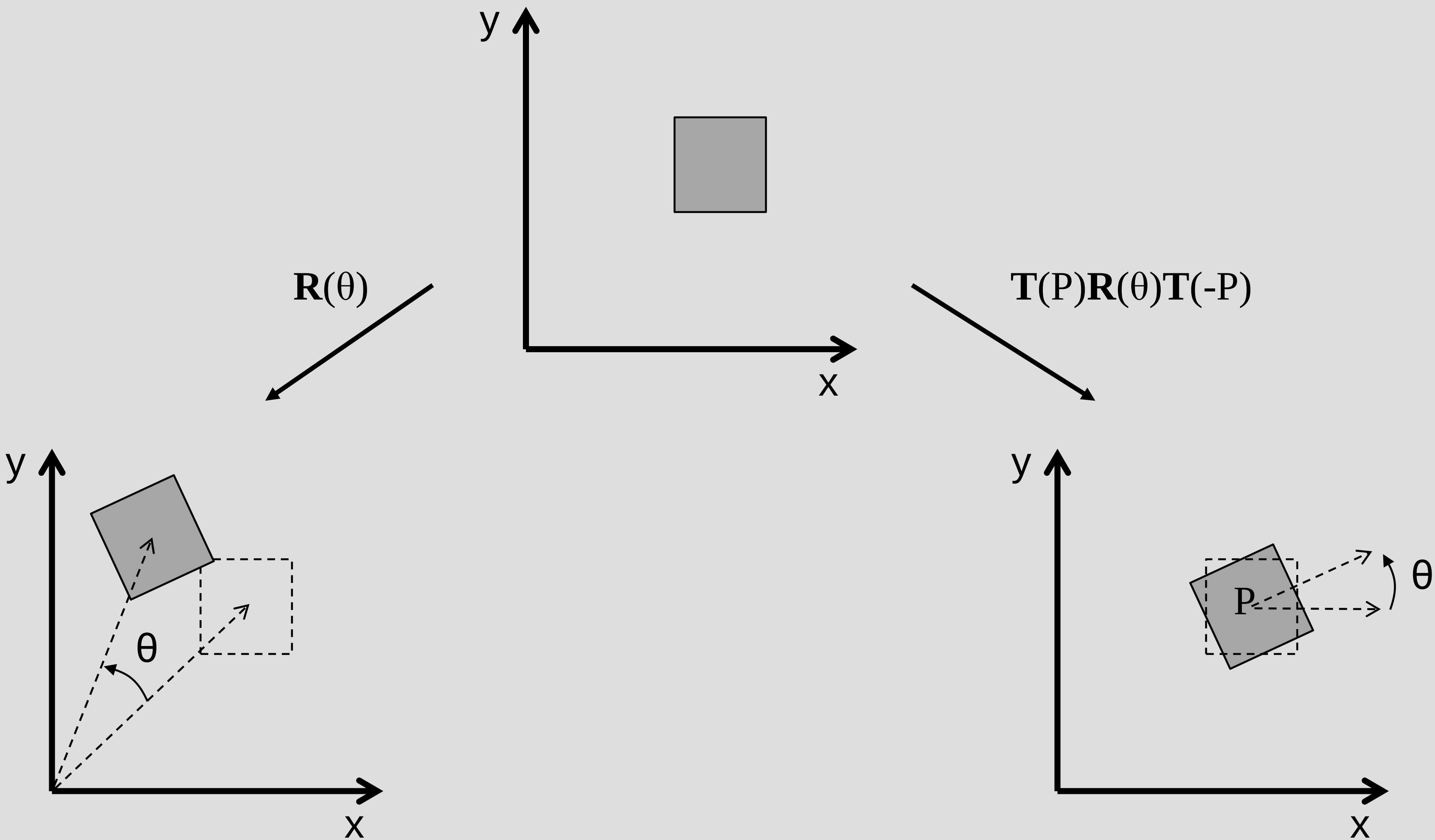
Rotation in **XY** plane by *q* degrees anti-clockwise about point *P*

$$\mathbf{M} = \mathbf{T}(P)\mathbf{R}(\theta)\mathbf{T}(-P)$$

$$= \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & P_x - P_x\cos\theta + P_y\sin\theta \\ \sin\theta & \cos\theta & 0 & P_y - P_x\sin\theta - P_y\cos\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Transformation Composition



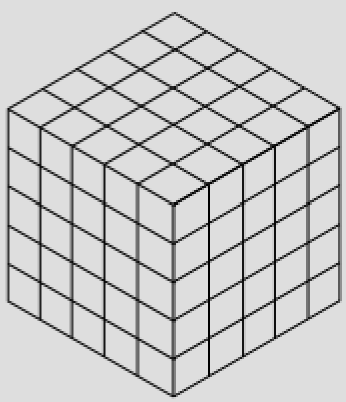
# Orientation Representations

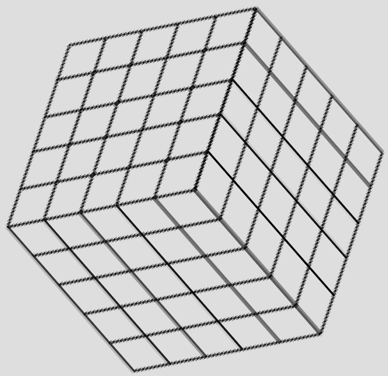
# Overview

- Orientation Representation
  - Matrices
  - Fixed Axis Angles
  - Euler Angles
  - Axis + Angle
  - Quaternions

# Orientation Representation

- What is the best way to represent the orientation of an object in space?
- Typical Scenarios:
  - User specifies an object in 2 transformed states and computer used to interpolate to produce animated keyframes
  - Object is to undergo 2 or more successive transformations
- Strengths and Weaknesses of each approach
  - Storage
  - Concatenation
  - Interpolation
  - Application





# 4 x 4 Transformation Matrix

- 4 x 4 matrix great for concatenation
- Fast to compute
- Storage issue for bone-animation
- Not suitable for keyframe interpolation
  - Intermediate matrices not correct

$$M(t0)=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

$$\mathbf{R}x(0)$$

$$M(t1)=\begin{bmatrix}1&0&0&0\\0&0.5&\frac{-\sqrt{3}}{2}&0\\0&\frac{\sqrt{3}}{2}&0.5&0\\0&0&0&1\end{bmatrix}$$

$$\mathbf{R}x(60)$$

$$0.5*M(t0)+0.5*M(t1)=\begin{bmatrix}1&0&0&0\\0&0.75&\frac{-\sqrt{3}}{4}&0\\0&\frac{\sqrt{3}}{4}&0.75&0\\0&0&0&1\end{bmatrix}$$

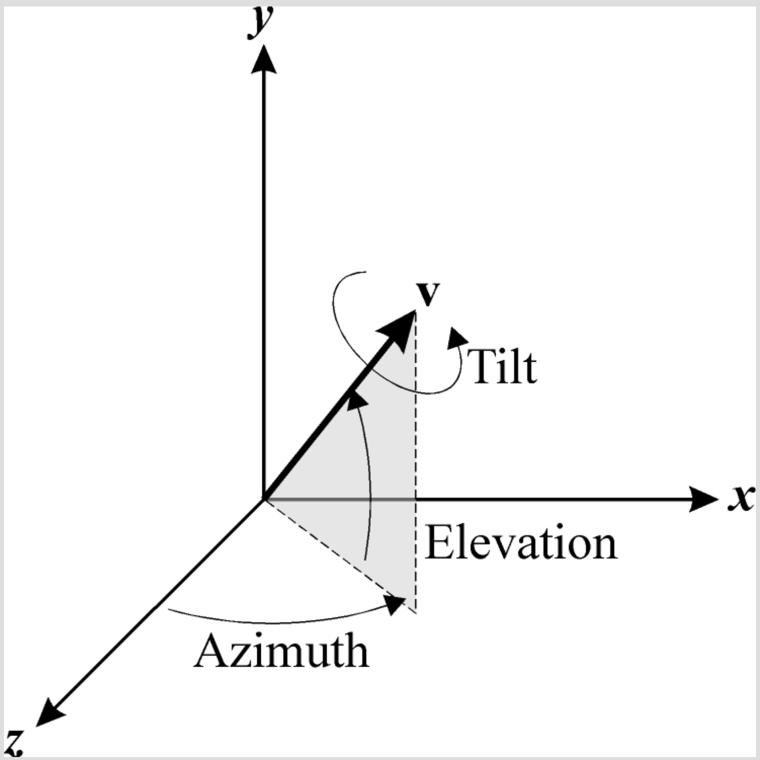
$$\cos^{-1}(0.75)\neq\sin^{-1}(\sqrt{3}/4)$$

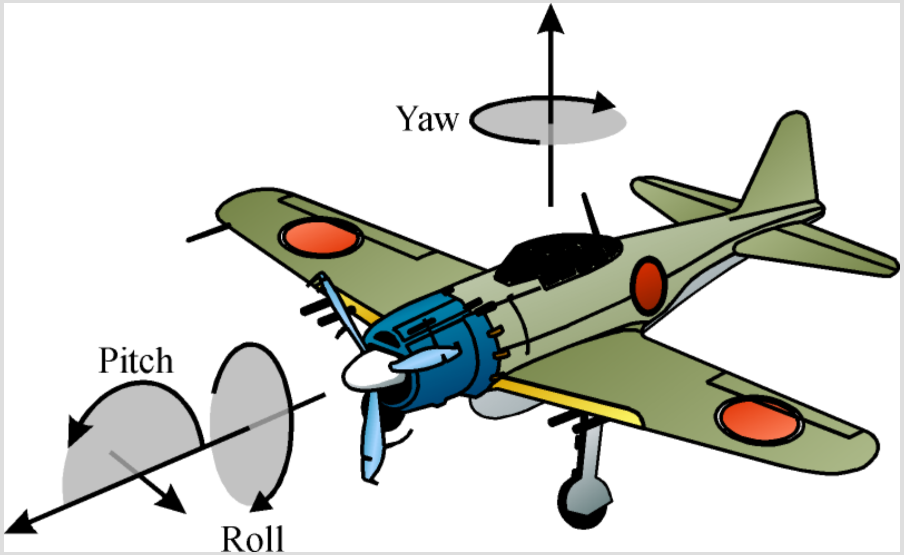
# Fixed-angle Representation

- Angles used to rotate about world axes
- A fixed order of three rotations is implied
  - e.g., x-y-z
- Many possible orderings of rotations
  - x-y-x
  - y-x-z
  - etc.
- Object orientation given by 3 angles: (10, 45, 90)

$$\mathbf{M} = \mathbf{R}_z(90)\mathbf{R}_y(45)\mathbf{R}_x(10)$$

# Euler Angles



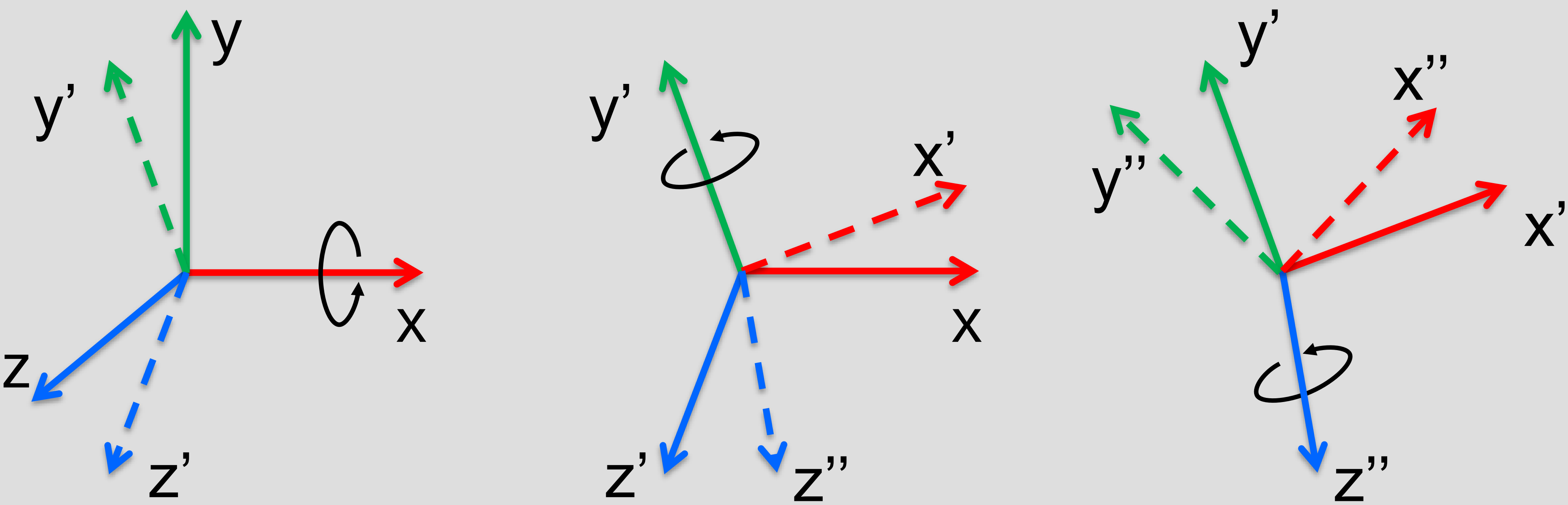

Sometimes known as *roll*, *pitch* and *yaw*

# Euler Angles

- In a *Euler angle representation,* the axes of rotation are the axes of the local coordinate system that rotate with the object, as opposed to the fixed global axes
- Can use any of various ordering of three axes of rotation as its representation scheme
- Example: Want rotation x-y-z by (10, 45, 90)
- Use prime symbol to represent rotation about a rotated frame

$$\mathbf{M} = \mathbf{R}_{z''}(90)\,\mathbf{R}_{y'}(45)\,\mathbf{R}_{x}(10)$$

# Euler Angles: xyz



$$\mathbf{M} = \mathbf{R}_z''(90)\mathbf{R}_y'(45)\mathbf{R}_x(10)$$

# Euler Angles

$$\mathbf{M} = \mathbf{R}_y'(45)\mathbf{R}_x(10)$$

- Using global axis rotation matrices to implement the transformations, the y-axis rotation can be achieved by: $\mathbf{R}_x(10)\mathbf{R}_y(45)\mathbf{R}_x(-10)$

- Thus result of two rotations is:

$$\mathbf{R}_y'(45)\mathbf{R}_x(10) = \mathbf{R}_x(10)\mathbf{R}_y(45)\mathbf{R}_x(-10)\boxed{\mathbf{R}_x(10)} = \mathbf{R}_x(10)\mathbf{R}_y(45)$$

$$\mathbf{M} = \mathbf{R}_z''(90)\mathbf{R}_y'(45)\mathbf{R}_x(10)$$

# Euler Angles

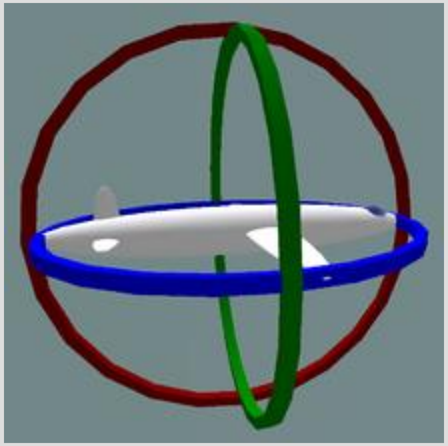
- The third rotation $\mathbf{R}_z(90)$ is around the now twice rotated frame
- This rotation can be achieved by undoing the previous rotations $\mathbf{R}_x(-10)$ followed by $\mathbf{R}_y(-45)$
- Then rotating around the global z-axis by $\mathbf{R}_z(90)$ & then re-applying the previous rotations

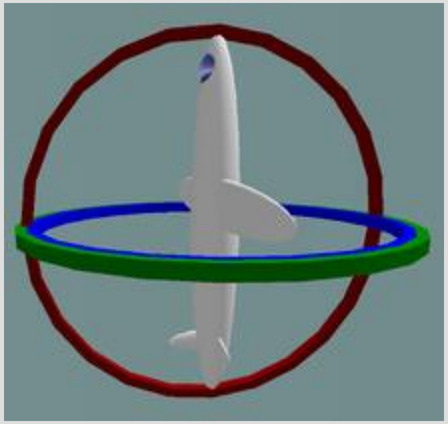
$$\mathbf{R}_z''(90)\mathbf{R}_y'(45)\mathbf{R}_x(10) = \overbrace{\mathbf{R}_x(10)\mathbf{R}_y(45)}\mathbf{R}_z(90)\overbrace{\mathbf{R}_y(-45)\mathbf{R}_x(-10)}\boxed{\mathbf{R}_x(10)\mathbf{R}_y(45)}$$

$$= \mathbf{R}_x(10)\mathbf{R}_y(45)\mathbf{R}_z(90)$$

# Gimbal Lock

- Problem: two axes of rotation can effectively line up on top of each other when an object can rotate freely in space
- Example
  - If the aircraft pitches up 90 degrees, the aircraft and platform's Yaw axis gimbal becomes parallel to the Roll axis gimbal
  - changes about yaw can no longer be compensated for.





http://en.wikipedia.org/wiki/Gimbal_lock

# Euler Angle Concatenation

- Can't just add or multiply components
- Best way:
  - Convert to matrices
  - Multiply matrices
  - Extract euler angles from resulting matrix
- Not cheap

# Euler Angle Interpolation

- •Example:
  - Halfway between (0, 90, 0) & (90, 45, 90)
  - Lerp directly, get (45, 67.5, 45)
  - Desired result is (90, 22.5, 90)
- Can use Hermite curves to interpolate
- Assumes you have correct tangents
- AFAIK, slerp not even possible

# Euler Angles

- Thus, system of Euler angles is precisely equivalent to the fixed-angle system in reverse order
- Has exactly the same advantages and disadvantages as those of the fixed-angle representation

# Advantages

- Compact
- Fairly intuitive
- Good for GUI
- Easy to work with
  - Similar to what we know how to do in mathematics
- Disadvantage
  - Gimbal lock
  - Interpolation
  - Expensive for concatenation

# Axis + Angle

- Euler Rotation Theorem: One orientation can be derived from another by a single rotation about an arbitrary axis

- Angle-axis
  - Any orientation can be represented by two parameters
    - The axis of rotation (vector)
    - The angle of rotation (scalar)

# Axis + Angle

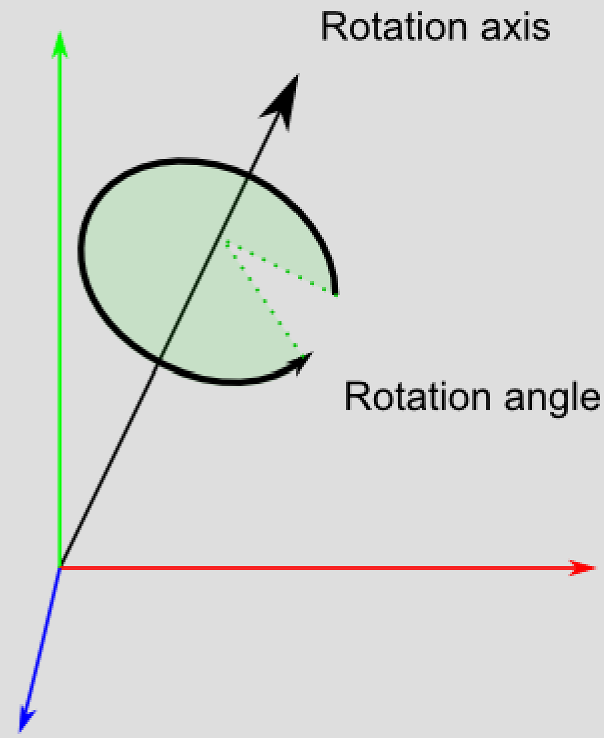
- Specify vector, rotate counter clockwise around it
- Intuitive
- Compact (only requires 3 floats)
- Cannot apply directly to points or vectors
- Can interpolate
  - Interpolate the axes of rotation and angles separately
- Not good for concatenation
  - Convert to matrix, concatenate, convert back
  - Same issue as Euler angles

# Axis + Angle

- Rotation

$$R(\mathbf{p},\hat{\mathbf{r}},\theta) = \cos\theta\cdot\,\mathbf{p} + (1-\cos\theta)(\mathbf{p}\bullet\hat{\mathbf{r}})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}}\times\mathbf{p})$$

- Convenient at times to do this
- Not the simplest operation
- More of a transitional format
  - Convert to axis-angle, manipulate angle or axis, convert back




https://www.youtube.com/watch?v=dttFiVn0rvc

# Axis + Angle - General

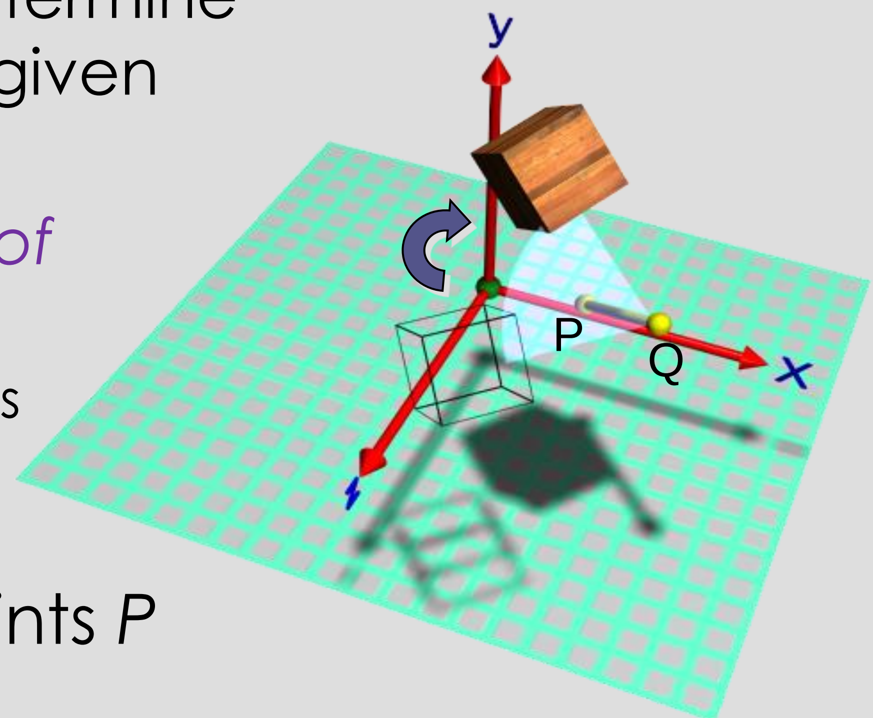
- What if you want to rotate about an axis that does not happen to be one of the 3 principle axes?
  - Can do this using operations we already have

- Strategy:
  - Do one or two rotations about the principal axes to get the axis we want aligned with the z-axis
  - Then, rotate about the z-axis
  - Undo the rotations we did to align your axis with the z-axis

# Axis + Angle - Matrices

- A frequent requirement is to determine the matrix for rotation about a given axis.
- Such rotations have 3 *degrees of freedom* (DOF):
  - 2 for spherical angles specifying axis orientation
  - 1 for twist about the rotation axis
- Assume axis is defined by points *P* and Q
- Pivot point is C and rotation axis vector is:

$$\mathbf{v} = \frac{P - Q}{|P - Q|}$$



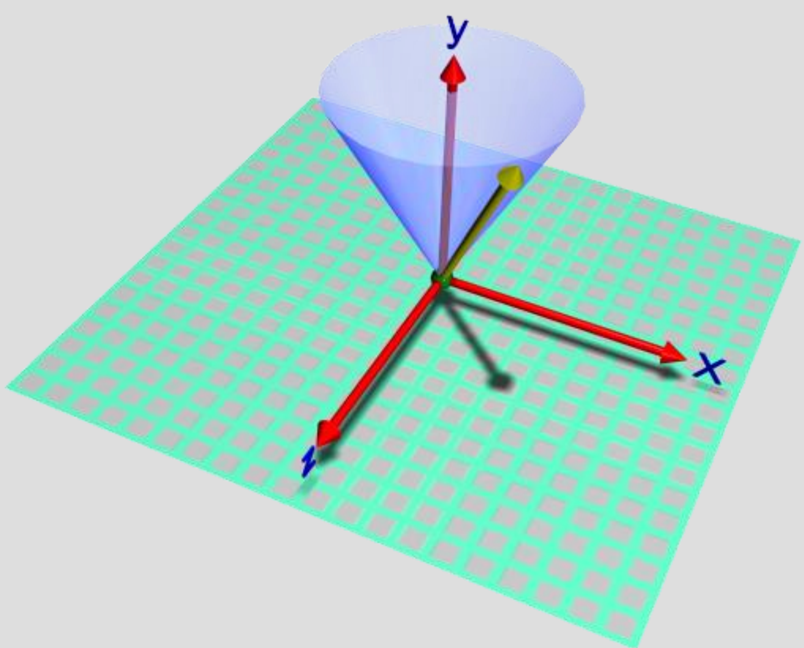
# Axis + Angle - Matrices

1. Translate the pivot point to the origin $\Rightarrow$ **T**$(-P)$
2. Align the axis of rotation (P) so that the axis lines up with **z** say $\Rightarrow$ **R**$(\theta_y)$**R**$(\theta_x)$
3. Rotate about **z** by the required angle $\theta \Rightarrow$ **R**$(\theta)$
4. Undo the first 2 rotations to bring us back to the original orientation $\Rightarrow$ **R**$(-\theta_x)$**R**$(-\theta_y)$
5. Translate back to the original position $\Rightarrow$ **T**$(P)$
6. The final rotation matrix is:

$$\mathbf{M} = \mathbf{T}(P)\mathbf{R}(-\theta_y)\mathbf{R}(-\theta_x)\mathbf{R}(\theta)\mathbf{R}(\theta_x)\mathbf{R}(\theta_y)\mathbf{T}(-P)$$

# Axis + Angle - Matrices

- We need the Euler angles $\theta_x$ and $\theta_y$ which will orient the rotation axis along the **z** axis.
- We determine these using simple trigonometry.

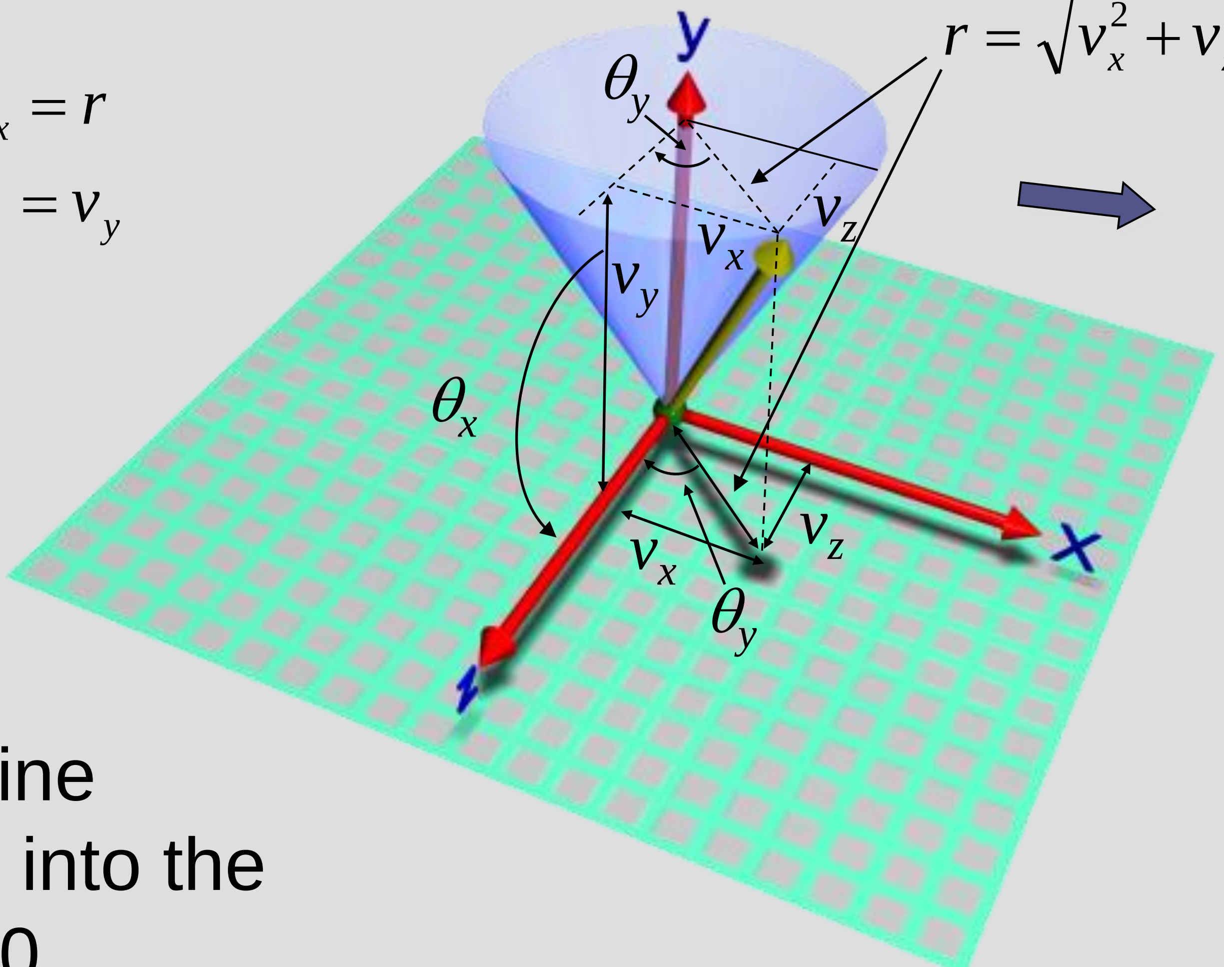


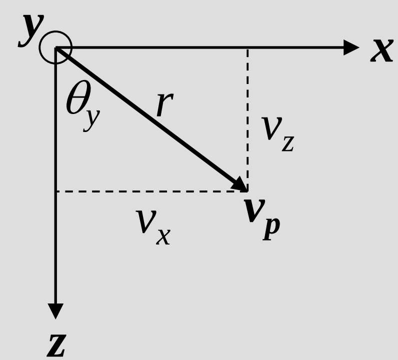
# Aligning axis with z

$$\cos\theta_x = r$$

$$\sin\theta_x = v_y$$

$$r = \sqrt{v_x^2 + v_z^2}$$




$$\cos\theta_y = \frac{v_z}{r}$$

$$\sin\theta_y = \frac{v_x}{r}$$

-Rotate line segment into the plane y=0

# Aligning axis with z

- Note that as shown the rotation about the **x** axis is anti-clockwise but the **y** axis rotation is *clockwise*.
- Therefore the required **y** axis rotation is $-\theta_y \Rightarrow$

$$\mathbf{R}(\theta_x)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & r & -v_y & 0\\ 0 & v_y & r & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\Rightarrow \mathbf{R}(-\theta_x)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & r & v_y & 0\\ 0 & -v_y & r & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$\mathbf{R}(\theta_y)=\begin{bmatrix}v_z/r & 0 & v_x/r & 0\\ 0 & 1 & 0 & 0\\ -v_x/r & 0 & v_z/r & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\Rightarrow \mathbf{R}(-\theta_y)=\begin{bmatrix}v_z/r & 0 & -v_x/r & 0\\ 0 & 1 & 0 & 0\\ v_x/r & 0 & v_z/r & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$\mathbf{M}=\mathbf{T}(P)\mathbf{R}(\theta_y)\mathbf{R}(-\theta_x)\mathbf{R}(\theta)\mathbf{R}(\theta_x)\mathbf{R}(-\theta_y)\mathbf{T}(-P)$$

# Quaternion Representation

- Developed by Sir William Rowan Hamilton in 1843
- Irish physicist, astronomer, and mathematician
- Was looking for ways of extending complex numbers to higher spatial dimensions
- Carved into side of Broom Bridge, along the Royal Canal
- Extension of complex numbers that gives us an elegant way of representing 3d rotations

# Quaternion Representation

- A quaternion is a 3-dimensional vector plus a fourth scalar coordinate
  - Axis of rotation is scaled by the sine of the half angle of rotation
  - Angle is stored as cosine of the half angle

$$\mathbf{q} = [\boldsymbol{a} sin\frac{\theta}{2} \; \cos\frac{\theta}{2}]$$

- Big benefits over axis+angle:
  - Rotations can be **concatenated** and directly applied to points and vectors via quaternion multiplication
  - Good for **interpolation** via LERP or SLERP
- No Gimbal Lock
- Small size

# Quaternion Operations

- Support some of the familiar operations
- Quaternion Multiplication
  - Given 2 quaternions: p, q representing rotations **P** and **Q**
  - pq represents the composite rotation (rotation **Q** followed by rotation **P**)
  - Grassman product

$$pq = (pq)_s + (\vec{pq})_v = (p_s q_s - \vec{p}_v \cdot \vec{q}_v) + (p_s \vec{q}_v + \vec{p}_v q_s + \vec{p}_v \times \vec{q}_v).$$

# Rotating Vectors

- Re-write vector in quaternion form
  - $v = [\mathbf{v}\ 0]$
  - $v' = \text{rotate}(q, \mathbf{v}) = qvq^{-1}$
- Since quaternions are always unit length:
  - $v' = \text{rotate}(q, \mathbf{v}) = qvq^{*}$

# Quaternion Concatenation

- Same as for matrices
- Just multiply quaternions together
- $q_{net} = q_3 q_2 q_1$;
- $v' = q_3 q_2 q_1 v {q_1}^{-1} {q_2}^{-1} {q_3}^{-1}$
- Note how quaternion rotations always multiply on *both* sides of the vector

# Quaternion-Matrix

- We can convert any 3D rotation freely between a 3 x 3 matrix and a quaternion
- Let $q = [\mathbf{q}\mathrm{v}\ \mathit{qs}] = [\mathbf{q}\mathrm{v}_x\ \mathbf{q}\mathrm{v}_y\ \mathbf{q}\mathrm{v}_z\ \mathit{qs}] = [x\ y\ z\ w]$, then:

$$\begin{bmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{bmatrix}$$

**eq. 1: Quaternion to matrix**

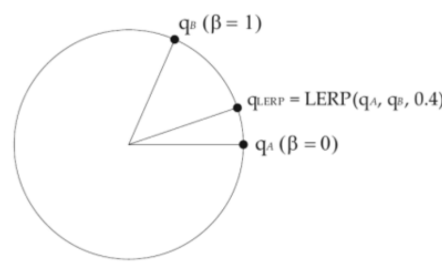
- (assumes using row vectors)
- Faster conversion & Discussion
  - http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm

# Linear Interpolation

- LERP is the easiest and least computationally intensive approach
- Given 2 quaternions, p & q, representing rotations A and B, we can find an intermediate rotation $q_{lerp}$ that is β percent of the way from A to B

$$\mathbf{q}_{LERP} = \mathrm{LERP}(\mathbf{q}_A, \mathbf{q}_B, \beta) = \frac{(1-\beta)\mathbf{q}_A + \beta\mathbf{q}_B}{|(1-\beta)\mathbf{q}_A + \beta\mathbf{q}_B|}$$

$$= \mathrm{normalize}\left(\begin{bmatrix} (1-\beta)q_{Ax} + \beta q_{Bx} \\ (1-\beta)q_{Ay} + \beta q_{By} \\ (1-\beta)q_{Az} + \beta q_{Bz} \\ (1-\beta)q_{Aw} + \beta q_{Bw} \end{bmatrix}^T\right).$$



# Linear Interpolation

- Quaternions are points on a 4-D hypersphere
- LERP interpolates along a chord of the hypersphere
  - Rotation animations do not have a constant angular speed when parameter is changing at a constant rate
  - Rotation will appear slower at the end points and faster in the middle

# Spherical Linear Interpolation

- SLERP uses sines and cosines to interpolate along a great circle of the 4D hypersphere
- Results in constant angular speed

$$\mathrm{SLERP}(\mathrm{p},\mathrm{q},\beta) = w_p\,\mathrm{p} + w_q\,\mathrm{q},$$

$$w_p = \frac{\sin((1-\beta)\theta)}{\sin(\theta)},$$

$$w_q = \frac{\sin(\beta\theta)}{\sin(\theta)}.$$

# SQT Transformations

- Quaternion only represents rotation
- Use SQT transform to combine with scale and translation
- Widely used in computer animation because of the smaller size

# Comparison

- Euler Angles
  - Simplicity
  - Small size
  - Intuitive – pitch, roll, yaw easy to visualize
  - Easily interpolate rotations about a single axis
  - Not good for more complex interpolations
  - Gimbal lock
  - Order of rotations matters (PYR, YPR..)

# Comparison

- Matrices
  - No gimbal lock
  - Can represent arbitrary rotations uniquely
  - Straightforward to apply to vectors and points
  - Built-in support
  - Transpose, inverse
  - Arbitrary affine transformations
  - Not intuitive to look at
  - Not easily interpolated
  - Storage

# Comparison

- Axis + Angle
  - Reasonably intuitive
  - Compact
  - Rotations not easily interpolated
  - Cannot be applied in a straightforward way to vectors

# Comparison

- Quaternions
  - Compact
  - Concatenation of rotations
  - Easily applied to points and vectors
  - Easy and smooth interpolation
  - Use SQT for combinations
  - No Gimbal lock
  - Not as intuitive to work with

# Degrees of Freedom

- Euler Angles
  - 3 parameters – 0 constraints = 3 DOF
- Axis+Angle
  - 4 parameters – 1 constraint = 3 DOF
  - Constraint: Axis is constrained to be unit length
- Quaternion
  - 4 parameters – 1 constraint = 3 DOF
  - Constraint: Quaternion is constrained to be unit length
- 3 x 3 matrix
  - 9 parameters – 6 constraints = 3 DOF
  - Constraints: All 3 rows and all 3 columns must be of unit length

# Recommended Reading

- Computer Animation: Algorithms & Techniques, 3rd Edition, Rick Parent
- Game Engine Architecture, 2nd Edition, Jason Gregory
- GDC vaults – Understanding Rotations, Jim Van Verth
- http://en.wikipedia.org/wiki/Gimbal_lock
- http://antongerdelan.net/teaching/guest_lectures/morph_targets/
- "Homogeneous Coordinates and Computer Graphics" by Tom Davis
- http://www.geometer.org/mathcircles/cghomogen.pdf
- Interactive Computer Graphics: A Top-down approach with OpenGL, by Edward Angel