

# Geometric Transformations

Lecturer:

Rachel McDonnell

Assistant Professor in Creative Technologies

Rachel.McDonnell@cs.tcd.ie

Course www: <https://www.scss.tcd.ie/~ramcdonn/>

Credit: Some slides taken from Robb T. Koether, Hampden-Sydney College

# Objectives

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

# Extra Reading

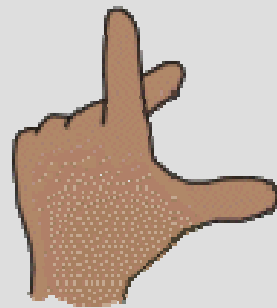
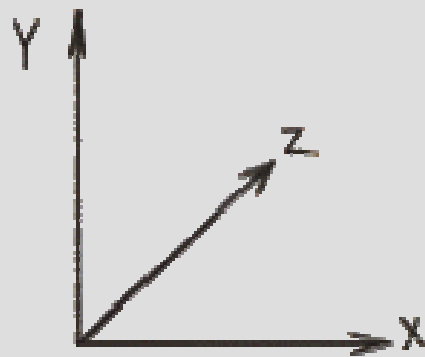
- **Chapter 3: Geometric Objects and Transformations**
- Interactive Computer Graphics: A Top Down Approach with OpenGL, 6<sup>th</sup> Edition (or other) Angel
- **Chapter 4: Math for 3D Graphics**
- OpenGL Superbible, 6<sup>th</sup> Edition
- **Elementary Linear Algebra**, Anton

# Linear Algebra

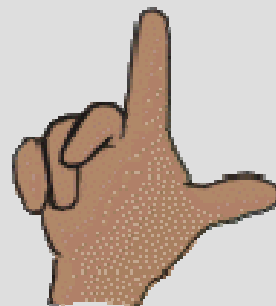
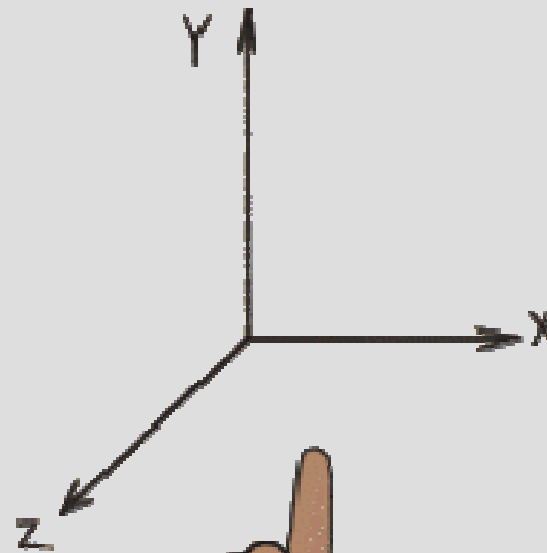
- Linear algebra is the cornerstone of computer graphics.
- Fundamentally, we need to be able to manipulate *points* and *vectors*.
  - these form the basis of all geometric objects & operations
- Geometric operations (*scale, rotate, translate, perspective projection*) are defined using matrix transformations.
- Optical effects (*reflect, refract*) defined using vector algebra.

# Co-ordinate Systems

- By convention we usually employ a *Cartesian basis*:
  - basis vectors are *mutually orthogonal* and *unit length*
  - basis vectors named **x**, **y** and **z**
- We need to define the relationship between the 3 vectors: there are 2 possibilities:
  - *right handed systems*: **z** comes out of page
  - *left handed systems*: **z** goes into page
  - (note: OpenGL uses a right handed system)
- This affects direction of rotations and specification of *normal vectors*

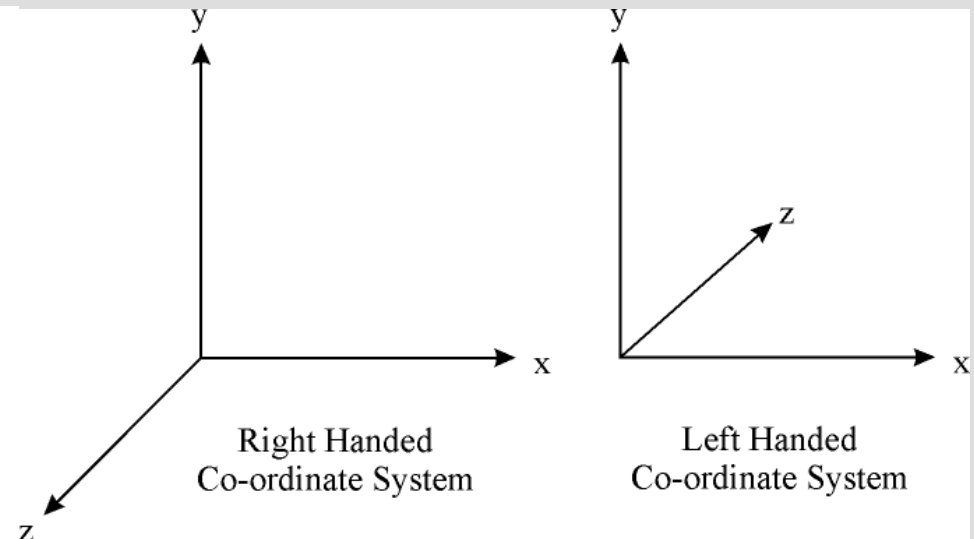
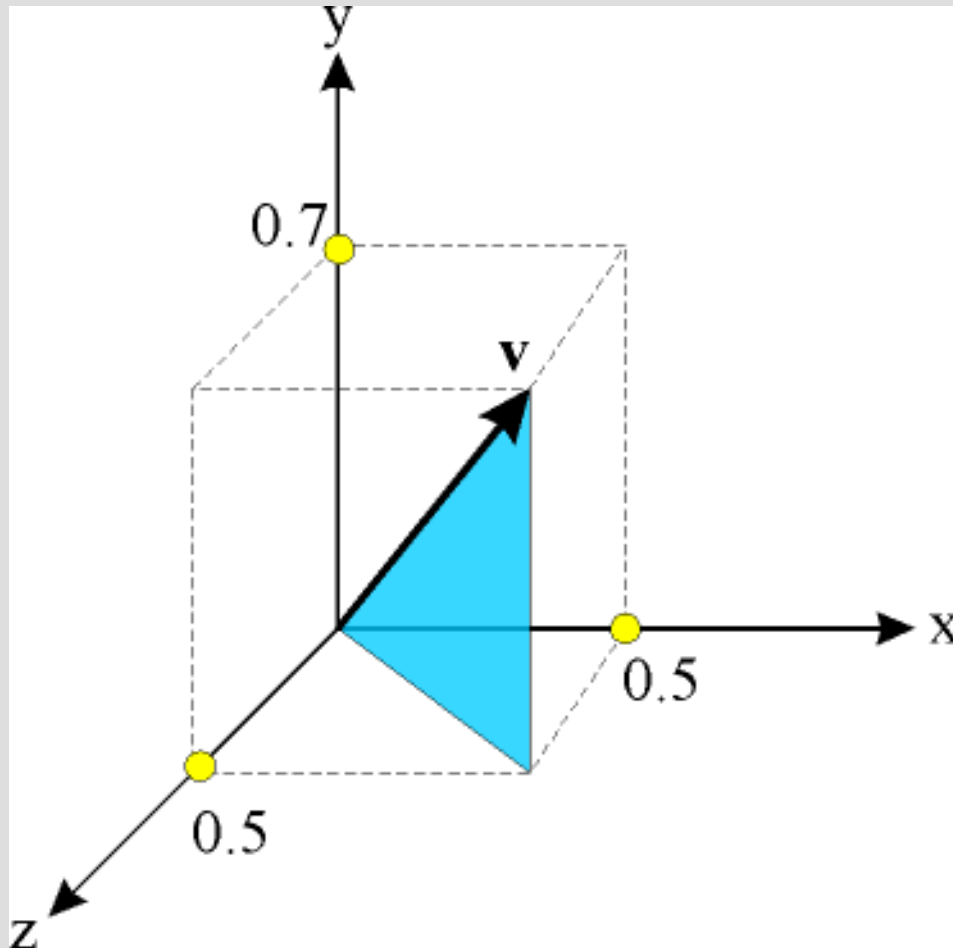


Left hand



Right hand

# Cartesian co-ordinate System



$$\mathbf{v} = \begin{bmatrix} 0.5 \\ 0.7 \\ 0.5 \end{bmatrix} =$$

RHS

LHS

# Conventions

- Vector quantities denoted as  $\mathbf{v}$  or  $\vec{v}$
- Each vector is defined with respect to a set of *basis vectors* (which define a co-ordinate system).
- We will use *column format* vectors:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \neq [v_1 \quad v_2 \quad v_3] \quad \left( = [v_1 \quad v_2 \quad v_3]^T \right)$$



# Row vs. Column Formats

- Both formats, though appearing equivalent, are in fact fundamentally different:
  - be wary of different formats used in textbooks

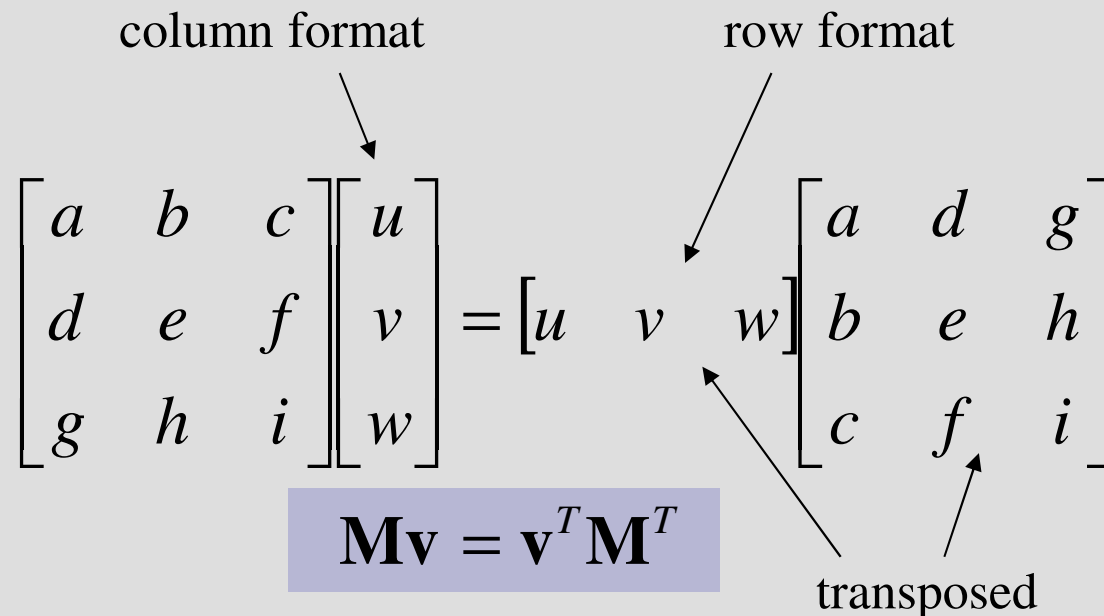
column format

row format

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} u & v & w \end{bmatrix} \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

$\mathbf{M}\mathbf{v} = \mathbf{v}^T \mathbf{M}^T$

transposed



$$\begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$$

Math Notation

Result	3x3	3d
	Matrix	Vector

---

mat3	M;
vec3	r;

vec3	u = M * r;
------	------------

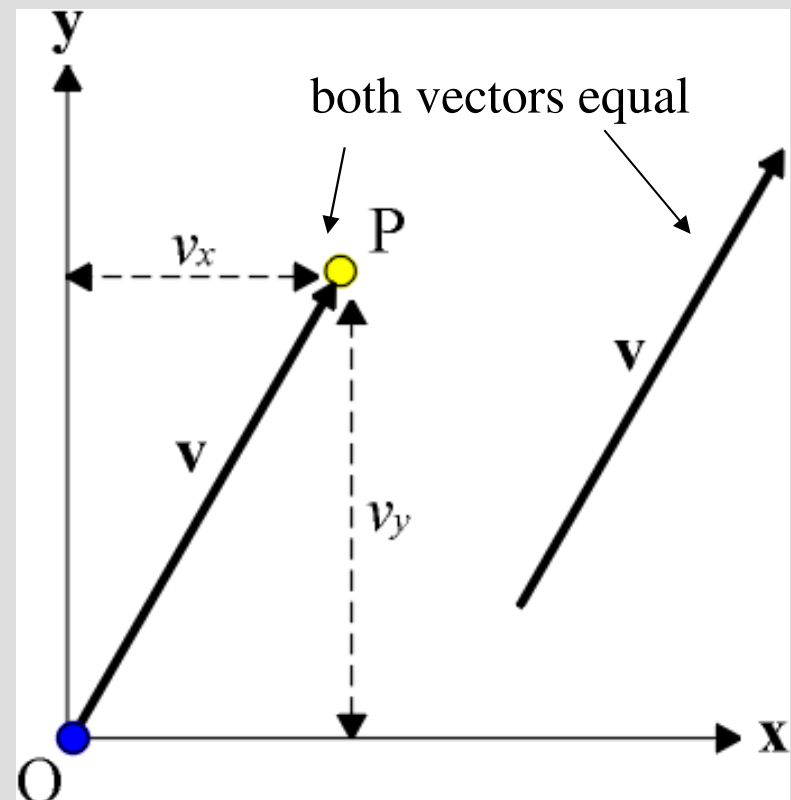
---

GLSL

# Vectors & Points

- Although *vectors* and *points* are often used inter-changeably in graphics texts, it is important to distinguish between them.
  - vectors represent directions
  - points represent positions
- Both are meaningless without reference to a *coordinate system*
  - vectors require a set of *basis vectors*
  - points require an *origin* and a *vector space*

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$



# 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
- Road example
  - Cars, tyres
  - View from a helicopter

# 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!



# 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

- $\mathbf{AB} \neq \mathbf{BA}$

- If  $\mathbf{AB} = \mathbf{AC}$ , it does not necessarily follow that  $\mathbf{B} = \mathbf{C}$

- It is associative and distributive

- $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$

- $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB} + \mathbf{AC}$

- $(\mathbf{A}+\mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$

- Transpose  $\mathbf{A}^T$  of a matrix  $\mathbf{A}$  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  $A+B^T$ ?
  - $AB^T = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ -4 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 5 \\ -2 & 4 \end{bmatrix}$



# Geometric Transformations

- Many geometric transformations are *linear* and can be represented as a matrix multiplication.

- Function  $f$  is linear iff:

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

- Implications:

- to transform a line we transform the *end-points*. Points between are *affine combinations* of the transformed endpoints.
- Given line defined by points  $P$  and  $Q$ , points along transformed line are affine combinations of transformed  $P'$  and  $Q'$

$$L(t) = P + t(Q - P)$$

$$L'(t) = P' + t(Q' - P')$$

# Homogeneous Co-ordinates

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

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

- 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 Co-ordinates

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

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

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

- Examples:

$\begin{bmatrix} 0.2 \\ 1.3 \\ 2.2 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0.2 \\ 1.3 \\ 2.2 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 0 \end{bmatrix}$
Points		Associated vectors	

# 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
- 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  $c$

$$x' = x + a$$

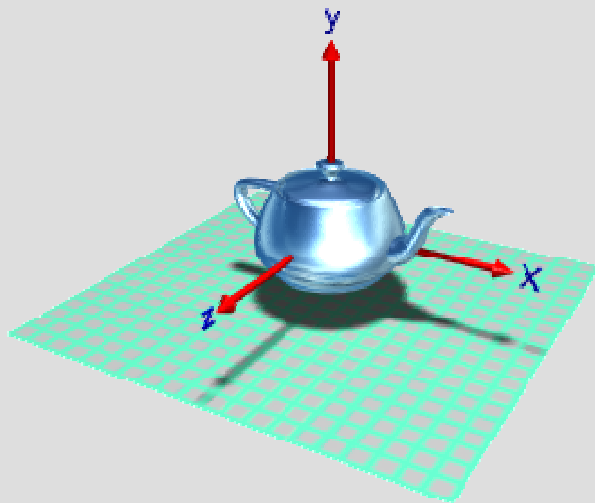
$$y' = y + b$$

$$z' = z + c$$

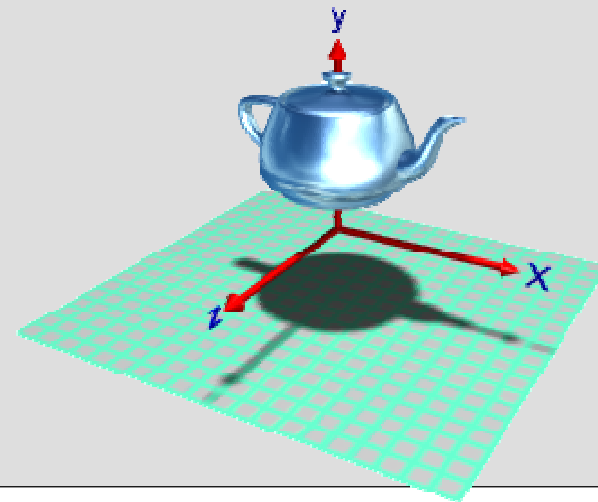
# Translation

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

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



translate along y



# Scaling

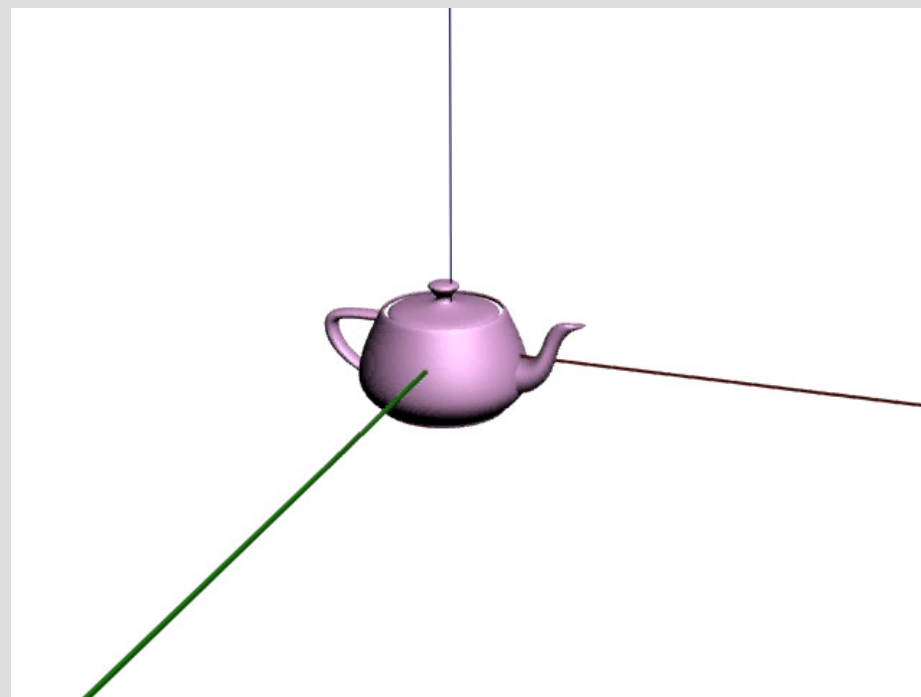
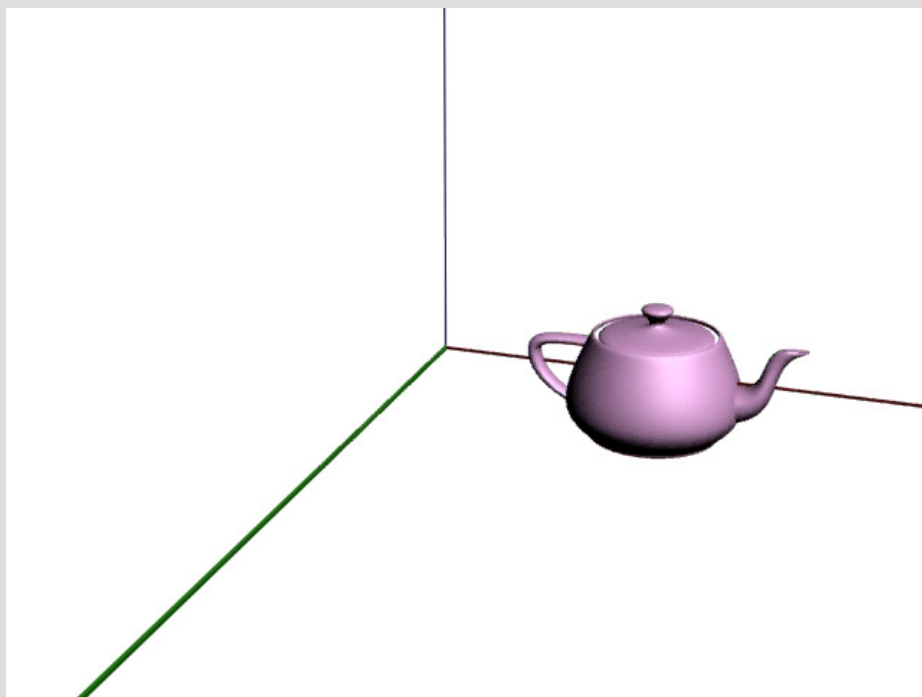
- What if we want to make things larger or smaller?
- Have a car model
  - Want one 3 times smaller!





# Scaling an object

- 3 times smaller
- Multiply all our coordinates by  $1/3$
- We get a model that is  $1/3$  of the size
- However
  - If original coordinates described a car 1 mile from the origin
  - Miniature car would only be  $1/3$  mile from the origin
- Solution – translation to origin, and then scale, then translate back



# Scaling

## Definition (Scaling)

A **scaling** is an expansion or contraction in the x, y, and z directions by scale factors  $s_x$ ,  $s_y$ ,  $s_z$  and centred at the point (a,b, c)

- Generally we centre the scaling at the origin

$$x' = s_x x$$

$$y' = s_y y$$

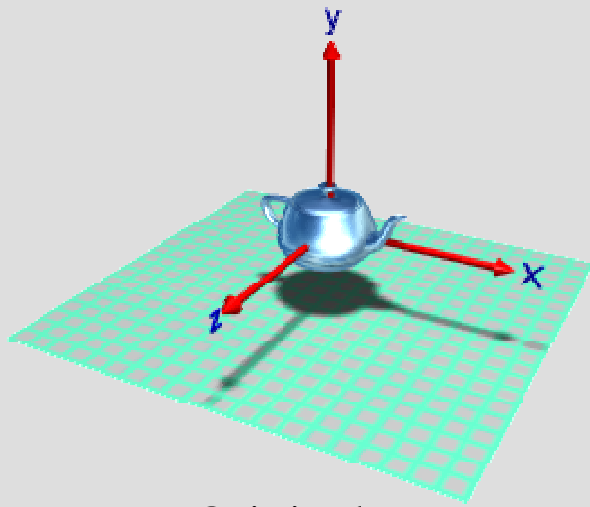
$$z' = s_z z$$

# Non-Uniform Scaling

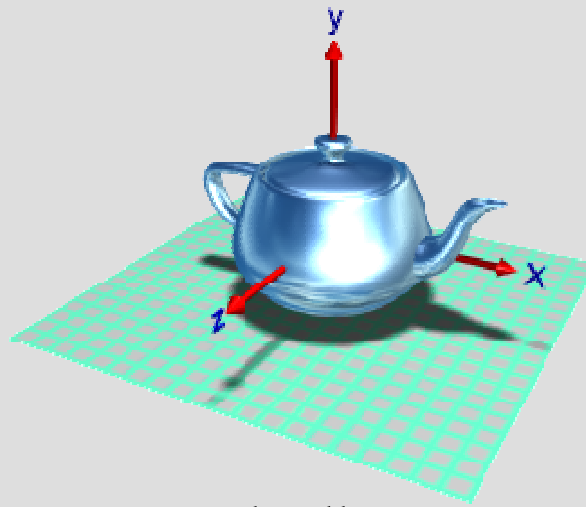
- Make an object twice as big in the x-direction
  - Multiply all x-coordinates by 2, leave y&z unchanged
- 3 times as large in the y-direction
  - Multiply all y-coordinates by 3, leave z&x unchanged

# Scale

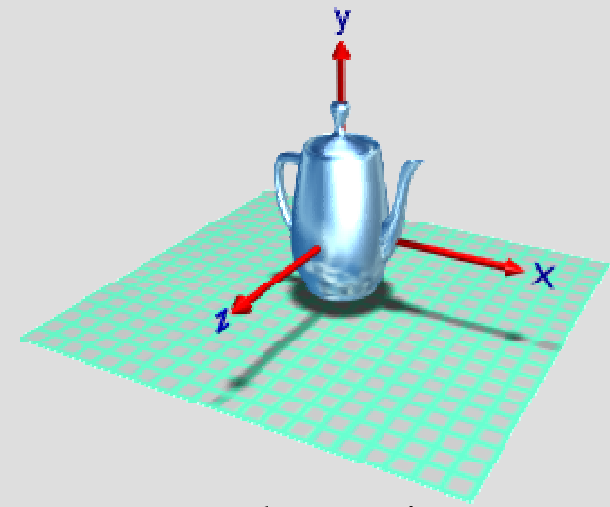
- all vectors are scaled from the origin:



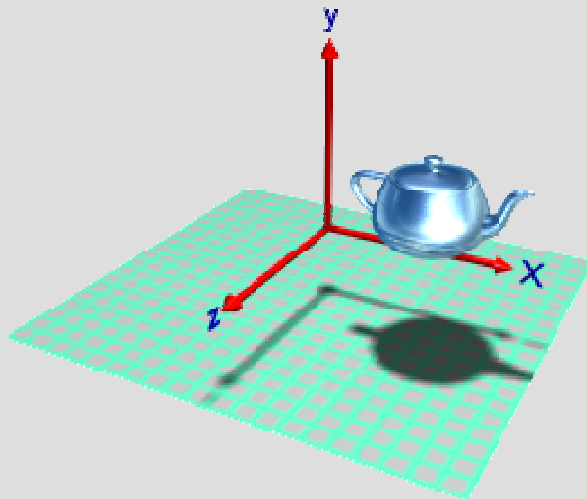
Original



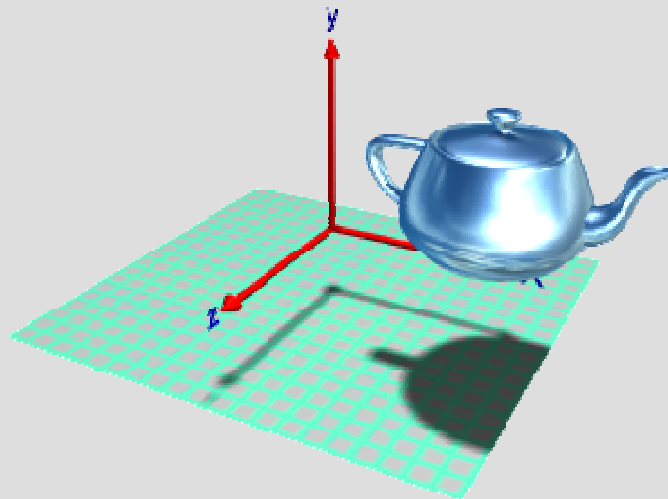
scale all axes



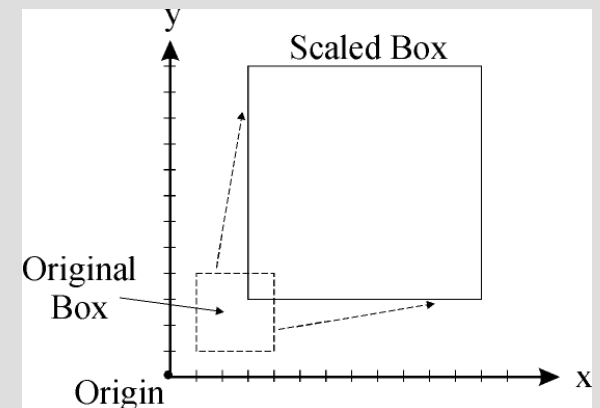
scale Y axis



offset from origin



distance from origin also scales



# Scale

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

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

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ w \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad \mathbf{S}^{-1} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Rotation

- Consider rotation in the x-y plane about the origin by an angle  $\theta$  in counter-clockwise direction
  - Same as rotation about the z-axis

# Rotation

## Definition (Rotation)

A rotation turns about a point  $(a,b)$  through an angle  $\theta$

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

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

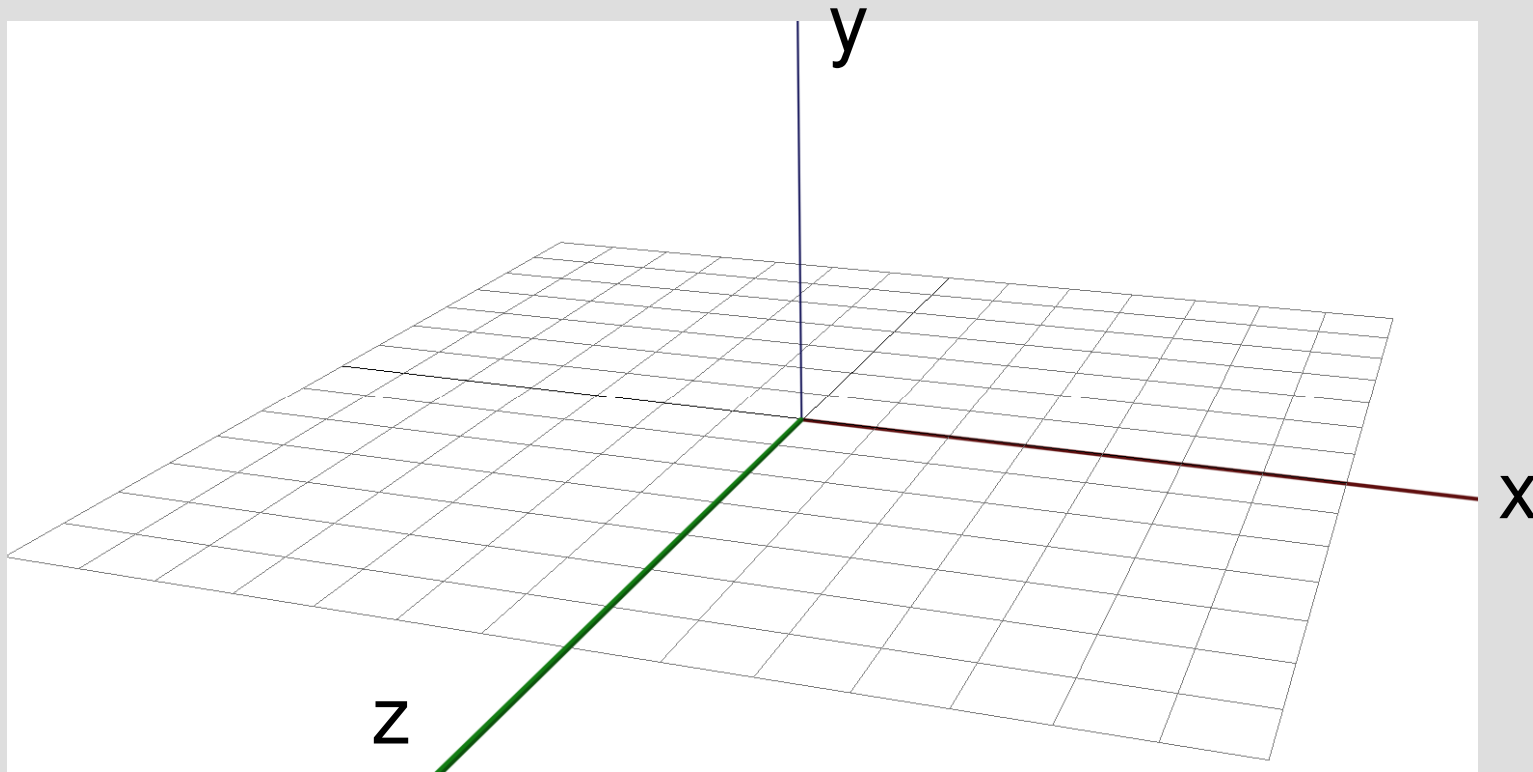
$$y' = x \sin \theta + y \cos \theta$$

$$z' = z$$



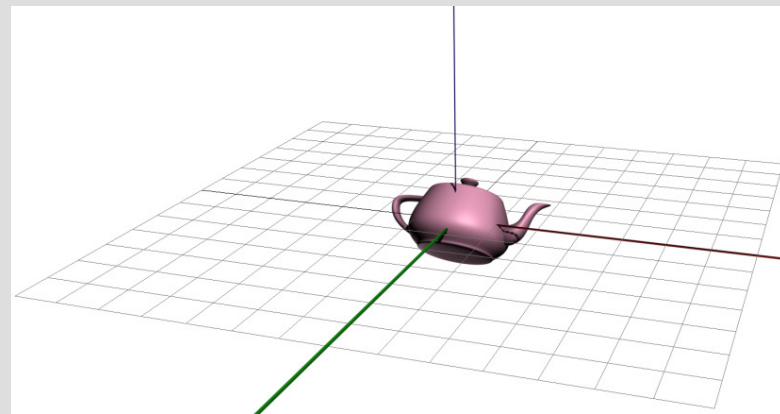
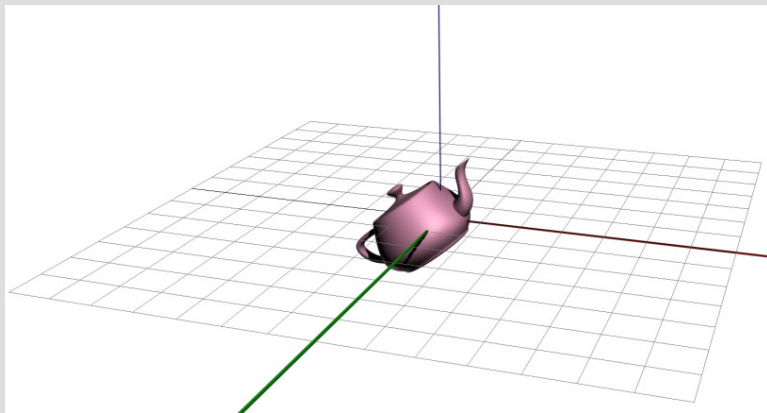
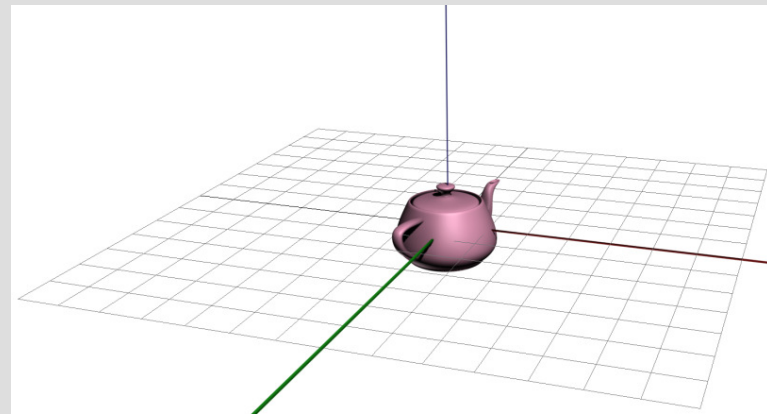
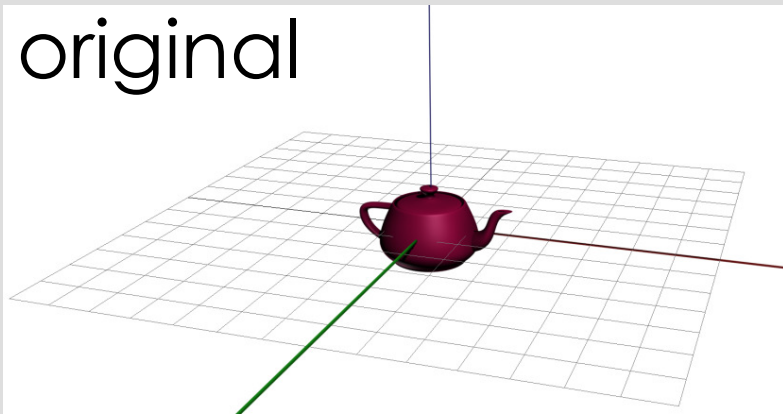
# Rotation - idea

- 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



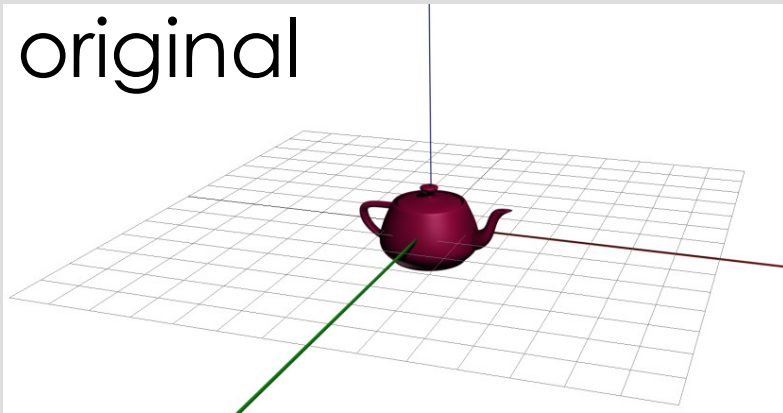
# Which Axis?

original

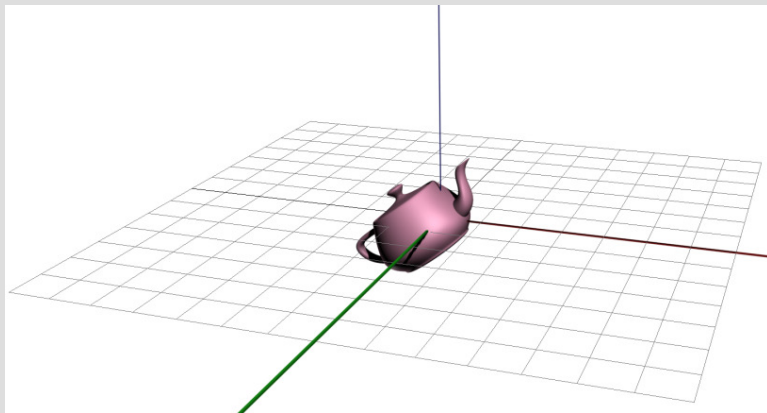
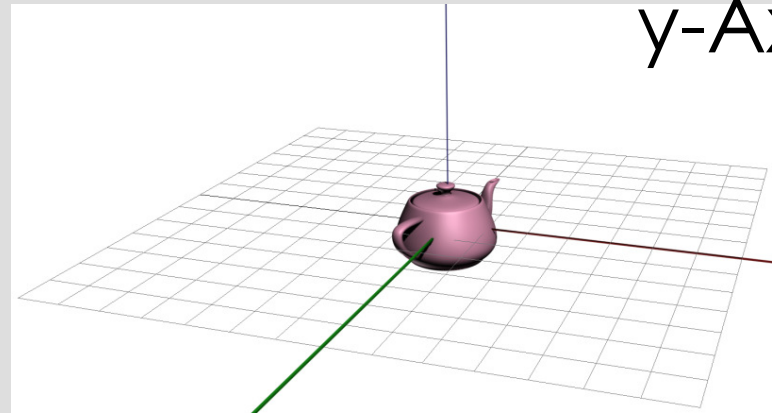


# Which Axis?

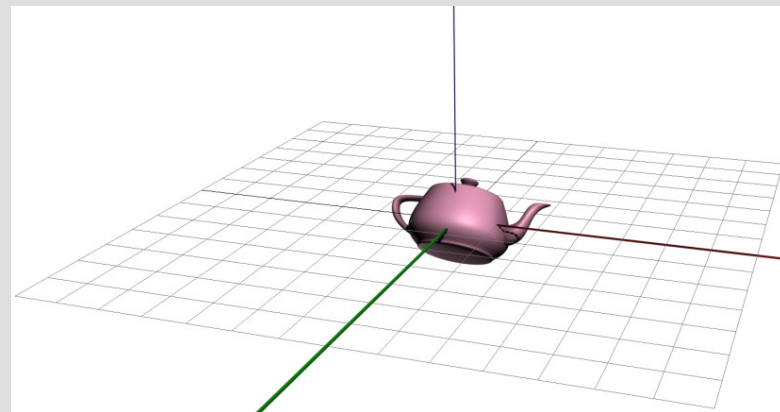
original



y-Axis



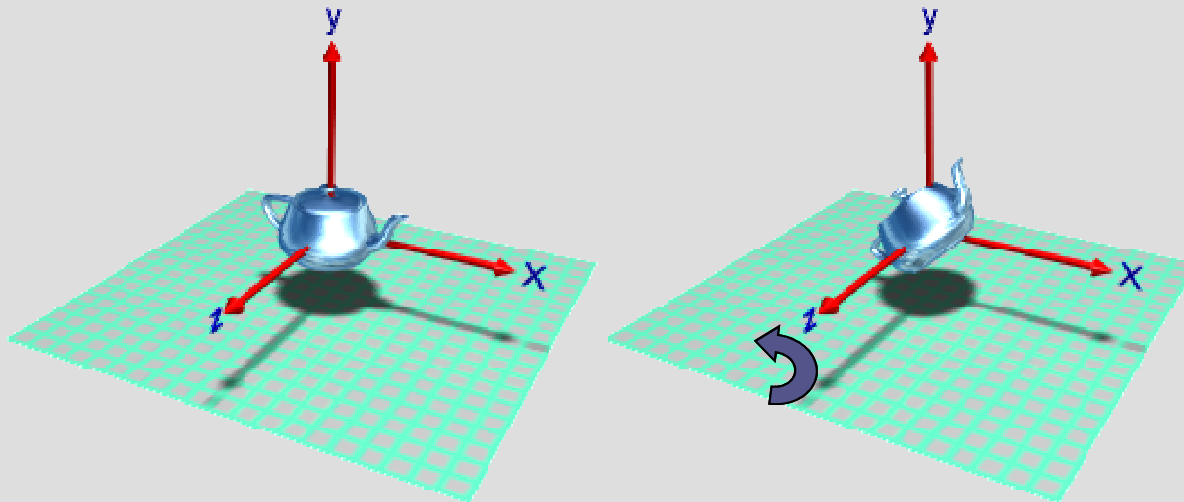
z-axis



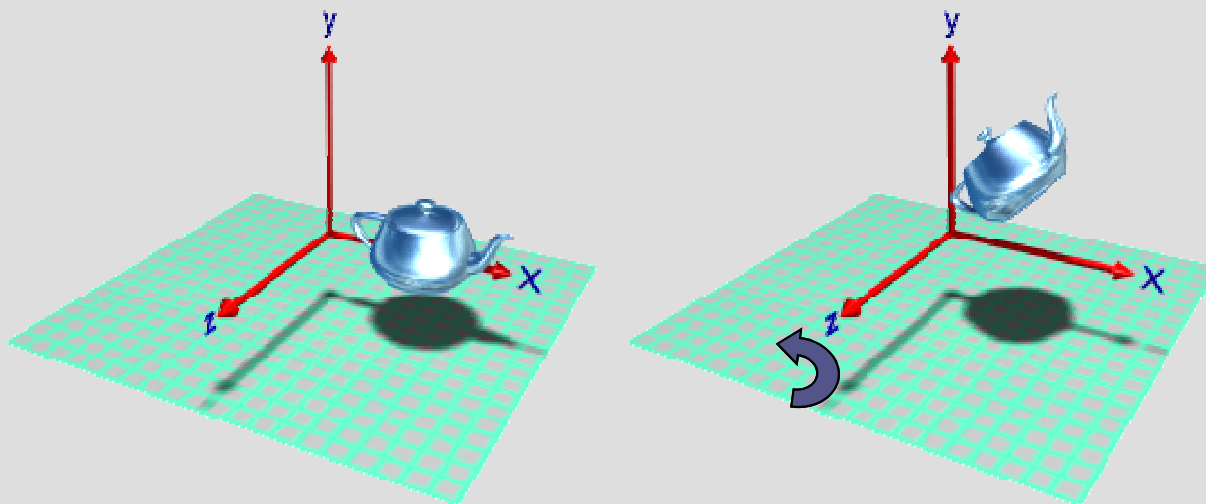
(-) x-axis

# Rotation

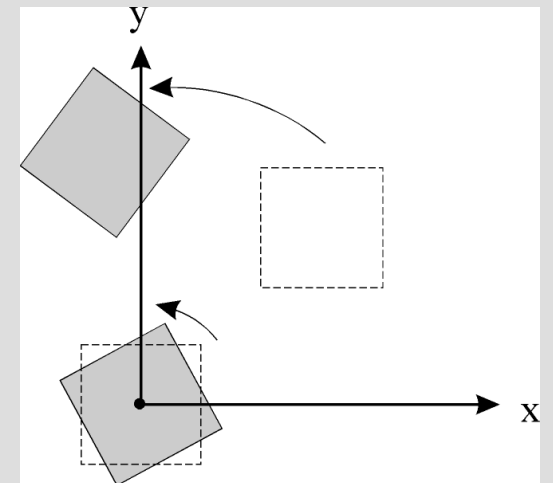
- Rotations are *anti-clockwise* about the *origin*:



rotation of  $45^\circ$  about the Z axis

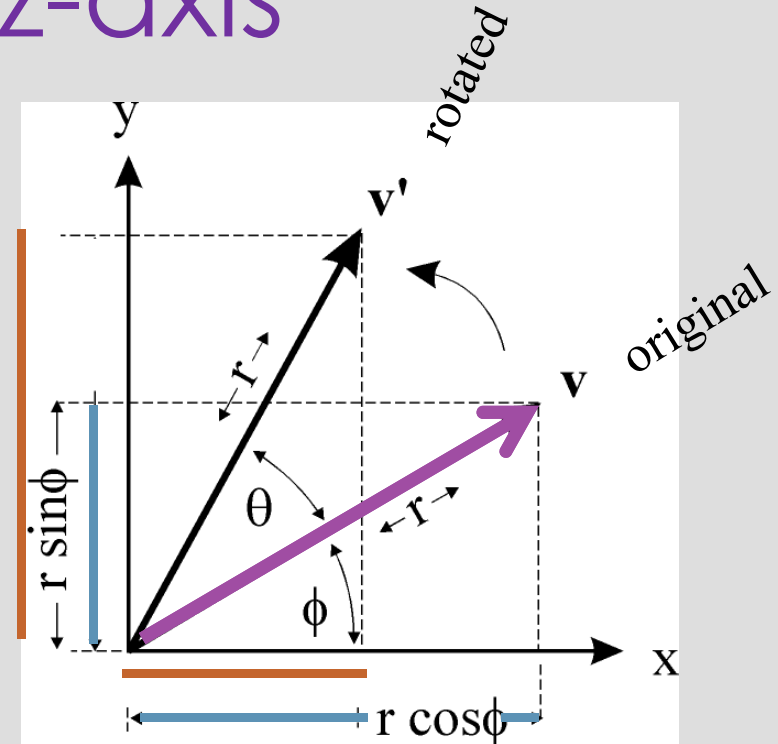


offset from origin rotation



# Rotation about the z-axis

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



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

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

# Rotation about the z-axis

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

# Rotation

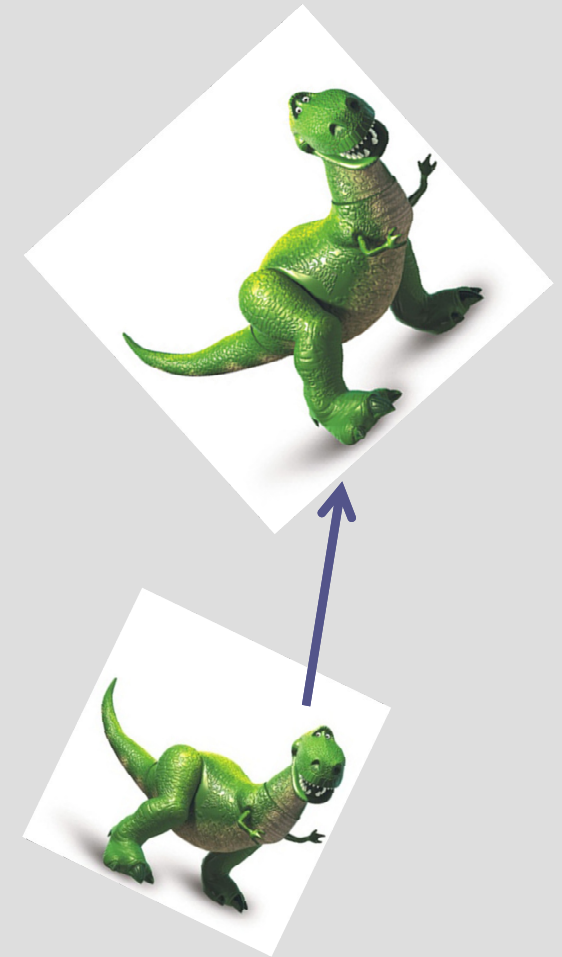
- 2D rotation of  $\theta$  about origin:  $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
- 3D homogeneous rotations:

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

- Note: difference for rotation about y, due to RHS
- Note:  $\cos(-\theta) = \cos \theta$   
 $\sin(-\theta) = -\sin \theta \Rightarrow \mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta) = \mathbf{R}^T(\theta)$
- If  $\mathbf{M}^{-1} = \mathbf{M}^T$  then  $\mathbf{M}$  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**



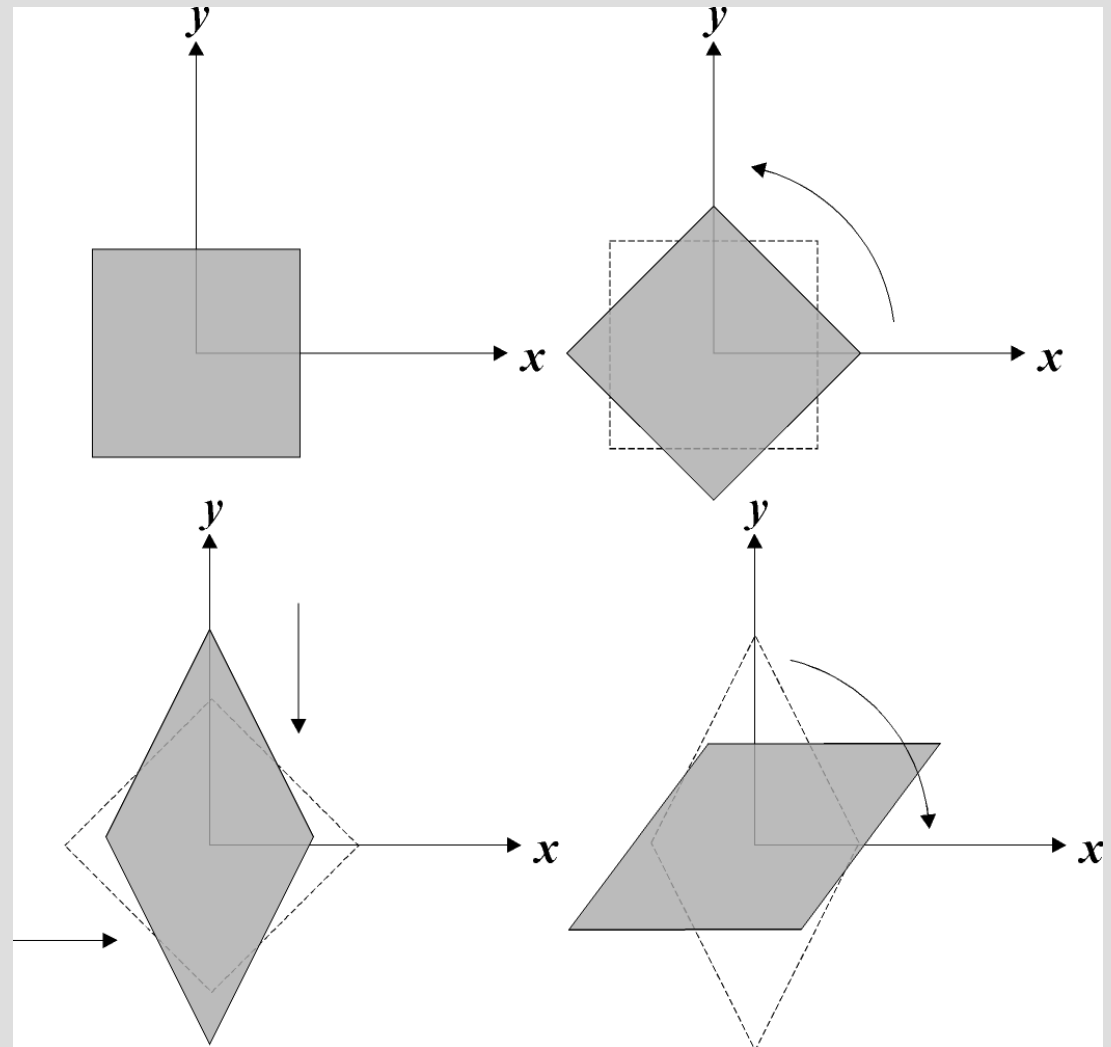


# Homogenous Coordinates

- 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

# Affine Transformations

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



# Transformation Composition

- It is common for graphics programs to apply **more than one** transformation to an object
  - Take vector  $\mathbf{v}_1$ , Scale it ( $\mathbf{S}$ ), then rotate it ( $\mathbf{R}$ )
  - First,  $\mathbf{v}_2 = \mathbf{S}\mathbf{v}_1$ , then,  $\mathbf{v}_3 = \mathbf{R}\mathbf{v}_2$
  - $\mathbf{v}_3 = \mathbf{R}(\mathbf{S}\mathbf{v}_1)$
  - Since matrix multiplication is **associative**:  $\mathbf{v}_3 = (\mathbf{RS})\mathbf{v}_1$
- 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:  $\mathbf{M} = \mathbf{RS}$

# Transformation Composition

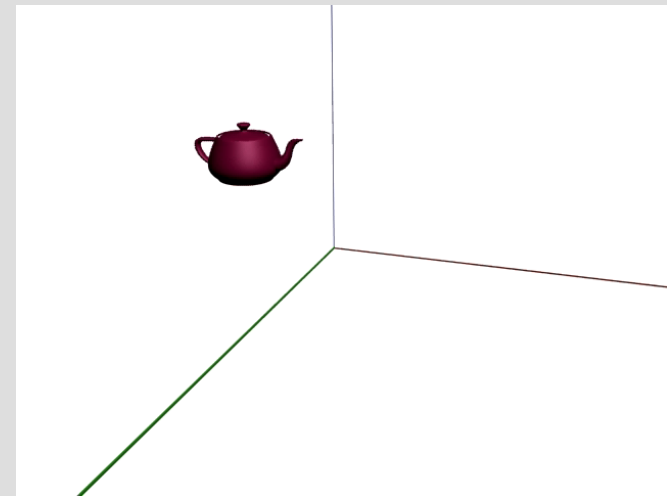
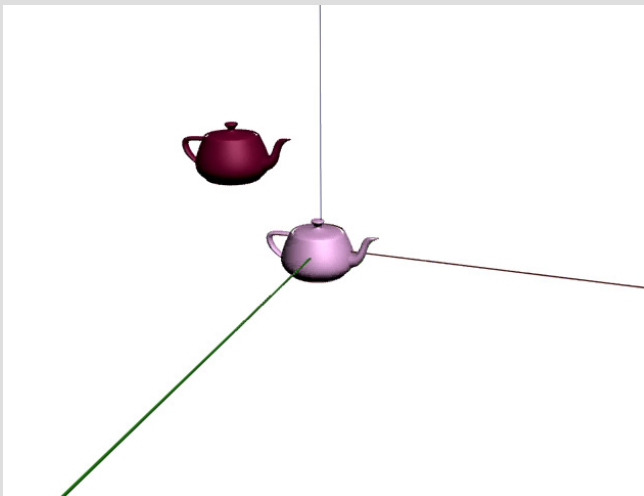
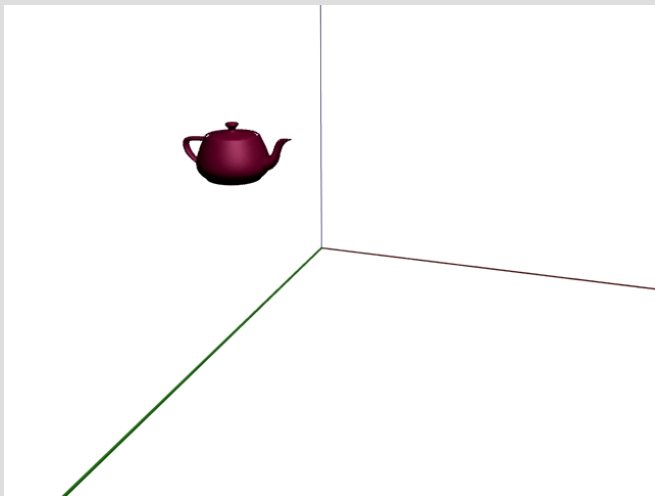
- More complex transformations can be created by *concatenating* or *composing* individual transformations together.

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

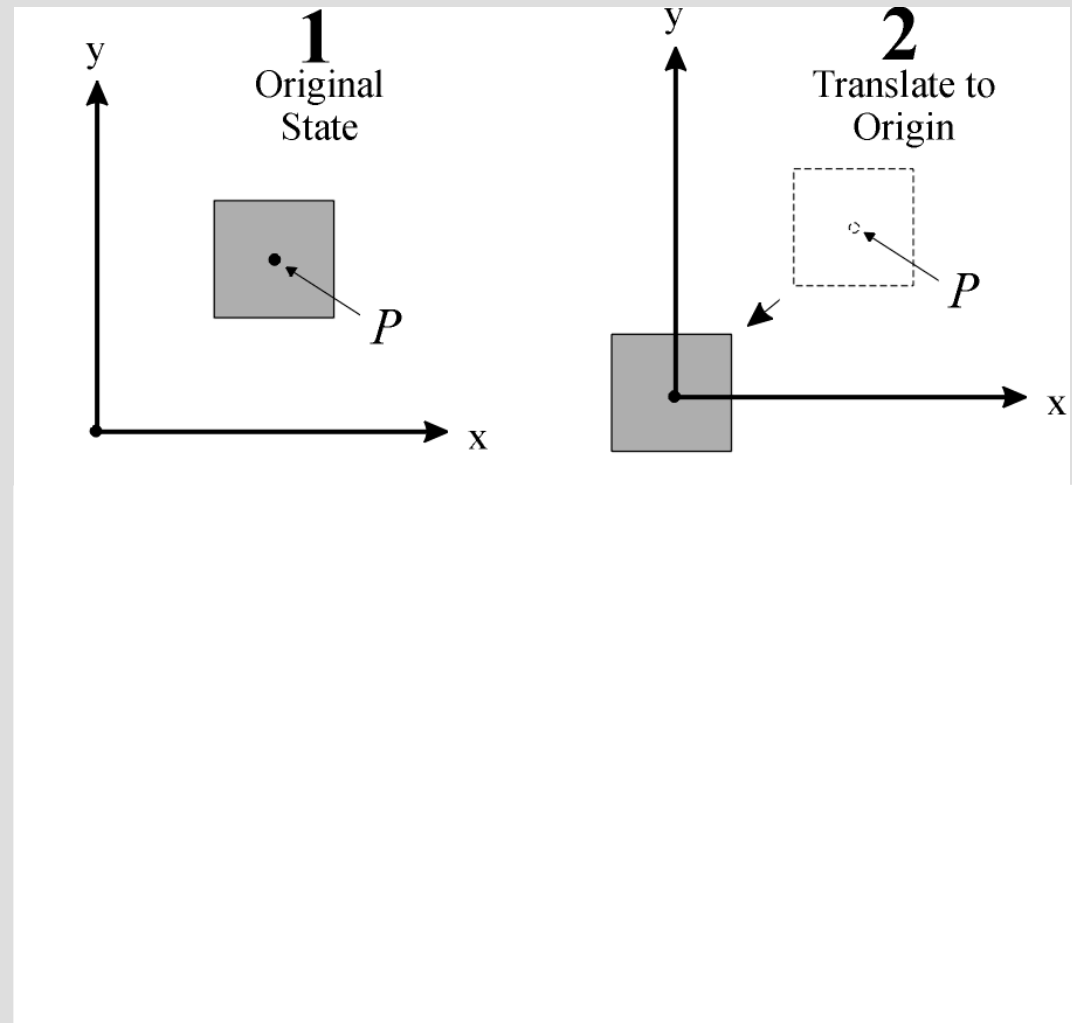
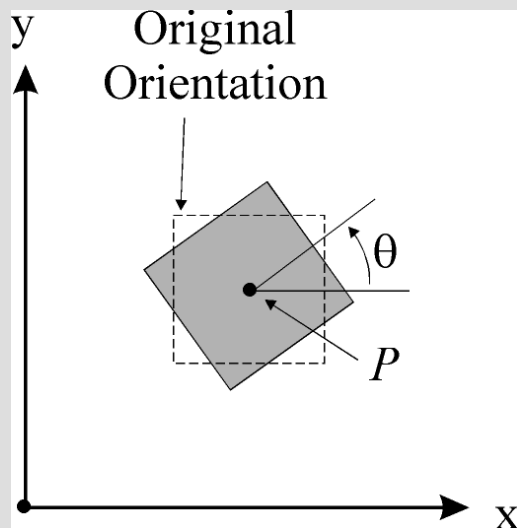
- Matrix multiplication is *non-commutative*  $\Rightarrow$  **order is vital**
  - We can create an affine transformation representing rotation about a point  $P_R$ :
$$\mathbf{M} = \mathbf{T}(P_R)\mathbf{R}(\theta)\mathbf{T}(-P_R)$$
- = *translate to origin, rotate about origin, translate back to original 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  $P$

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

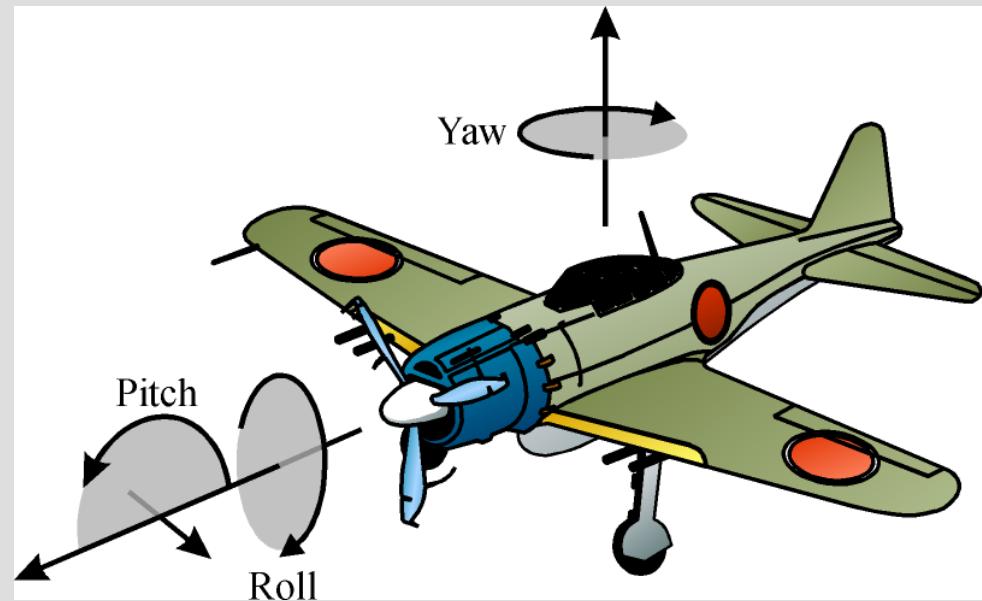
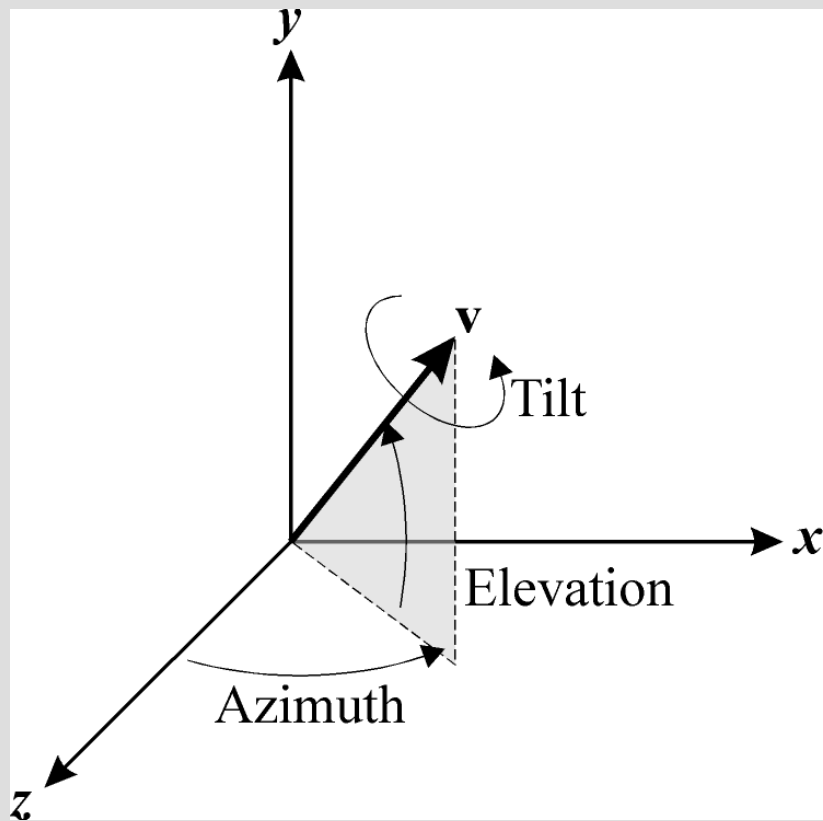
$$= \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0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

# Euler Angles

- *Euler angles* represent the angles of rotation about the co-ordinate axes required to achieve a given orientation  $(\theta_x, \theta_y, \theta_z)$
- The resulting matrix is:  $\mathbf{M} = \mathbf{R}(\theta_x)\mathbf{R}(\theta_y)\mathbf{R}(\theta_z)$
- Any required rotation may be described (*though not uniquely*) as a **composition** of 3 rotations about the coordinate axes.
- Remember rotation does **not commute**  $\Rightarrow$  order is important



# Rotational DOF



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

# OpenGL - Uniforms

- Pass data into a shader that stays the same – is uniform
  - e.g., transformation matrix
- Get data directly from application to shaders
- Simply place the keyword **uniform** at beginning of variable definition
  - uniform float fTime
  - uniform mat4 modelMatrix

# Using Uniforms to Transform Geometry

- Now it is time to put all our knowledge together and build a program that does a little more than pass vertices through un-transformed

# 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;
}
```

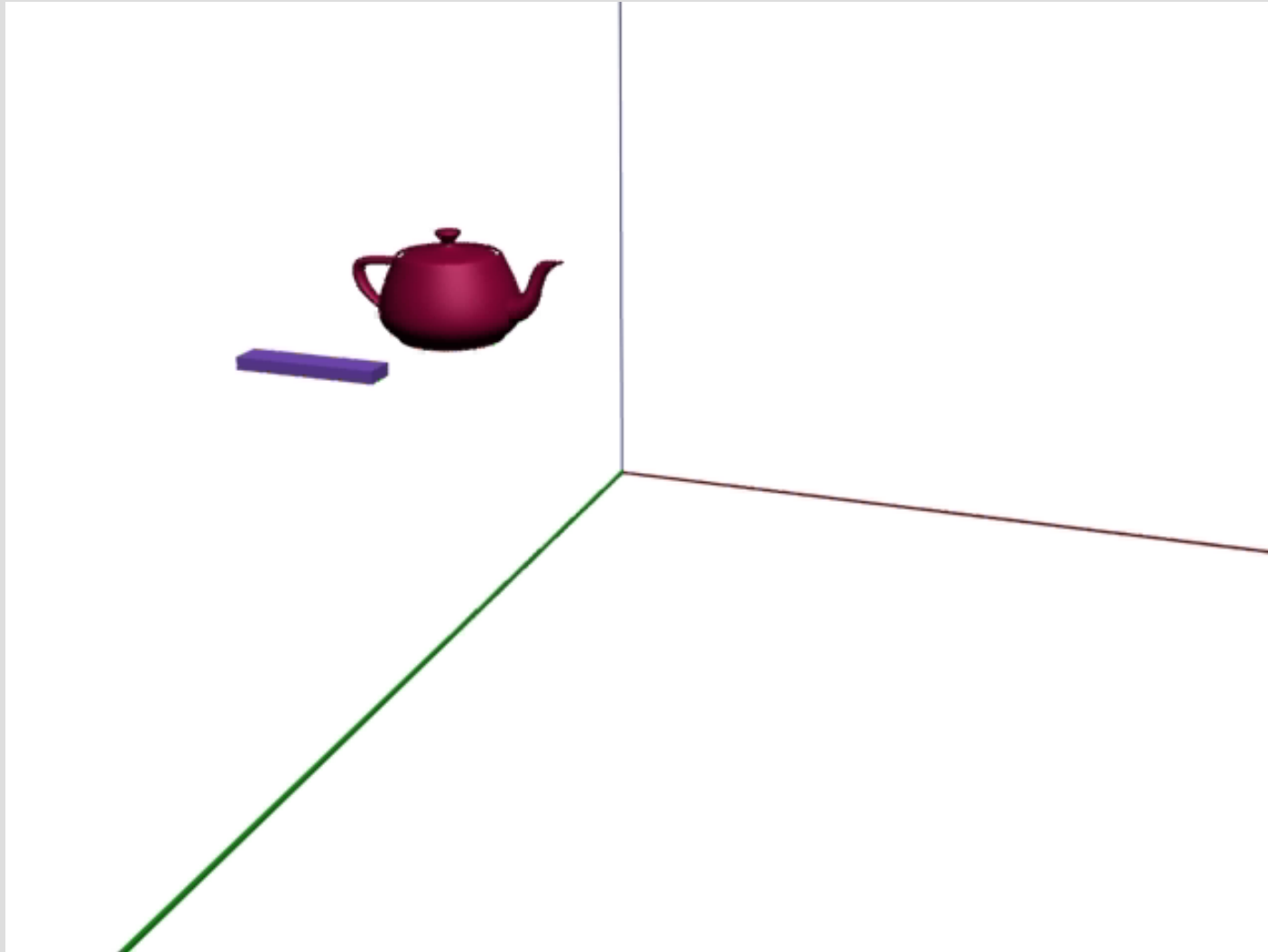
New position in NDC

Original (local) position

# General Rotation

- 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

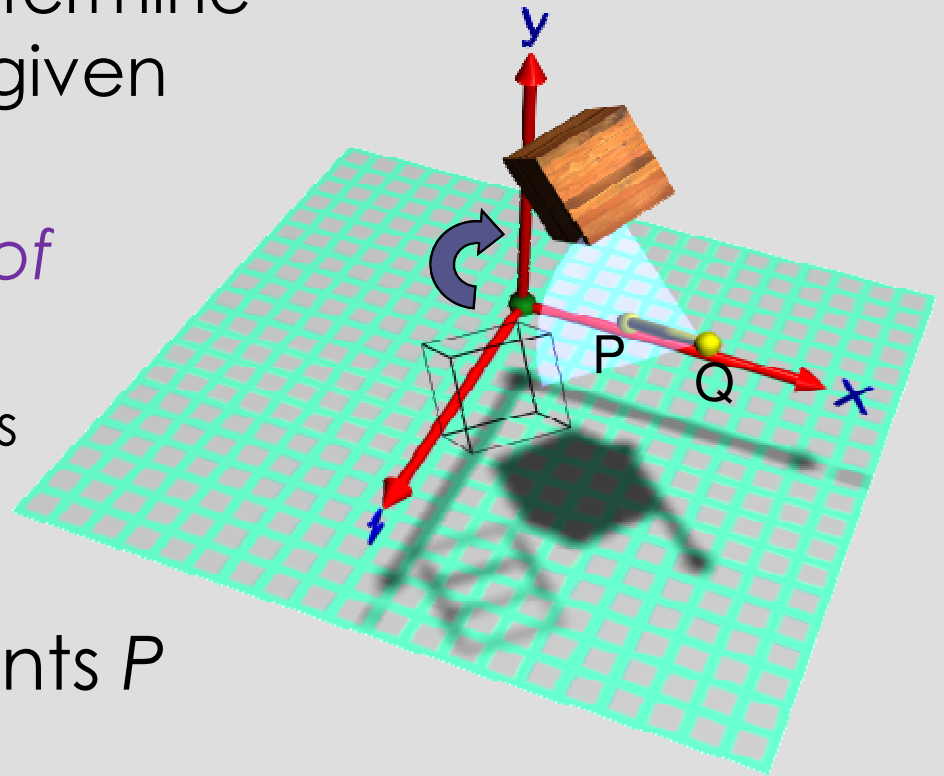
# Rotation about an arbitrary axis



# Rotation about an arbitrary axis

- 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  $P$  and rotation axis vector is:

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





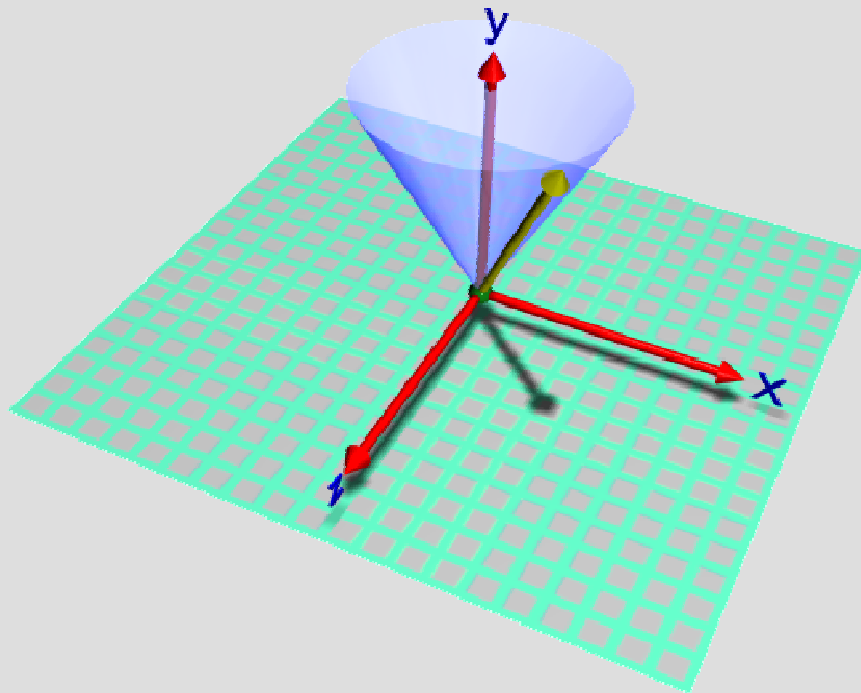
# Rotation about an arbitrary axis

1. Translate the pivot point of the axis to the origin  $\Rightarrow \mathbf{T}(-P)$
2. Determine a series of rotations to achieve the required rotation about the desired vector.
3. Rotate the axis and object so that the axis lines up with  $\mathbf{z}$  say  $\Rightarrow \mathbf{R}(\theta_y)\mathbf{R}(\theta_x)$
4. Rotate about  $\mathbf{z}$  by the required angle  $\theta \Rightarrow \mathbf{R}(\theta)$
5. Undo the first 2 rotations to bring us back to the original orientation  $\Rightarrow \mathbf{R}(-\theta_x)\mathbf{R}(-\theta_y)$
6. Translate back to the original position  $\Rightarrow \mathbf{T}(P)$
7. The final rotation matrix is:

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

# Rotation about an axis

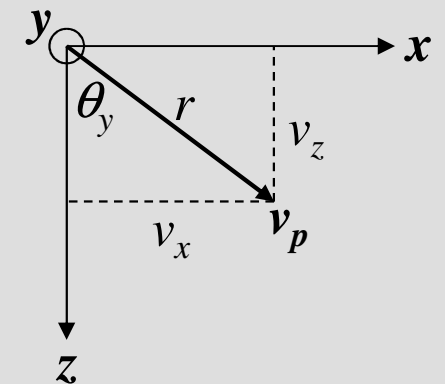
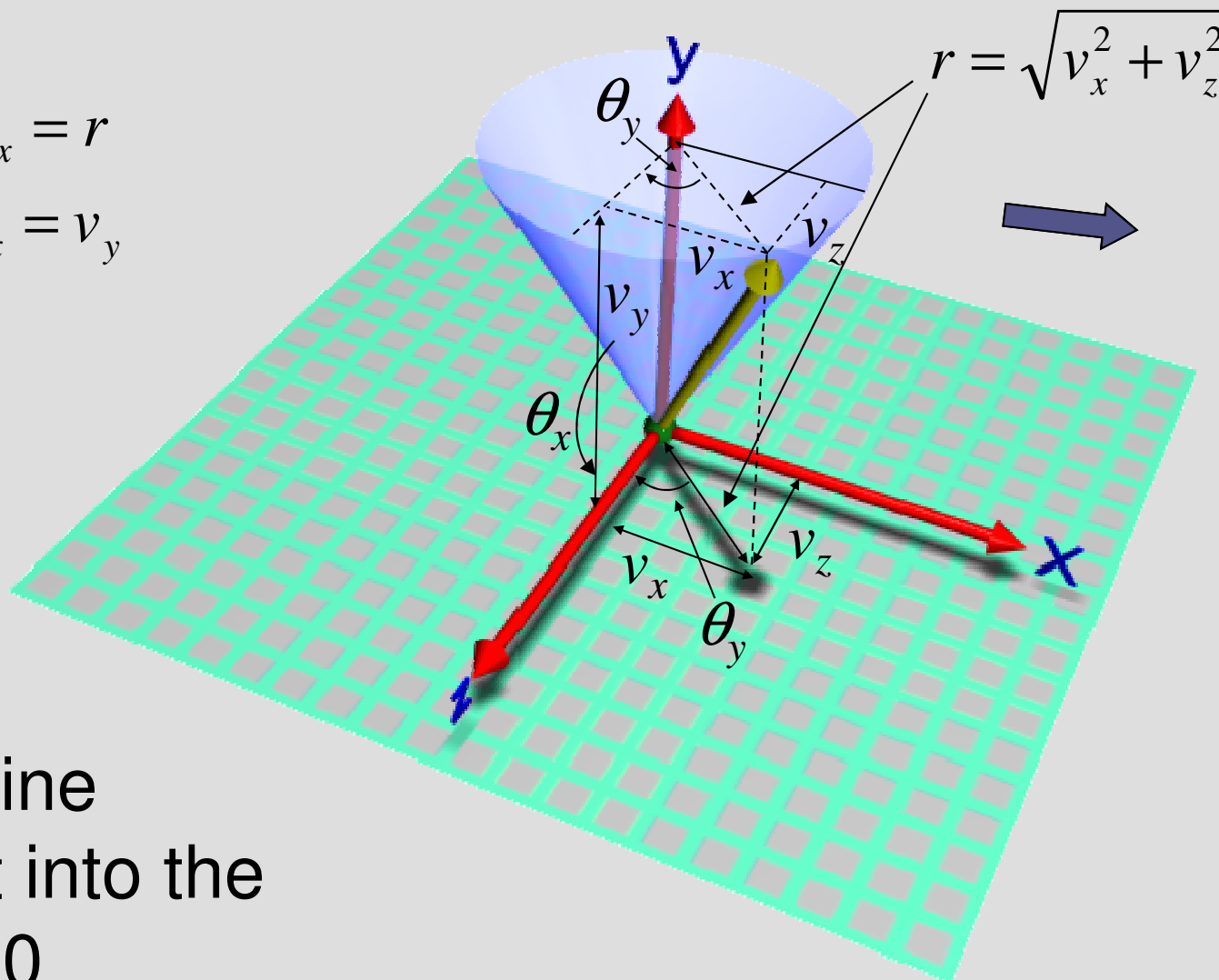
- We need the Euler angles  $\theta_x$  and  $\theta_y$  which will orient the rotation axis along the **z** axis.
- We determine these using simple trigonometry.



# Aligning axis with z

$$\cos \theta_x = r$$

$$\sin \theta_x = v_y$$



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

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

-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  $-\theta_y \Rightarrow$

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

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

# Recommended Reading

- “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