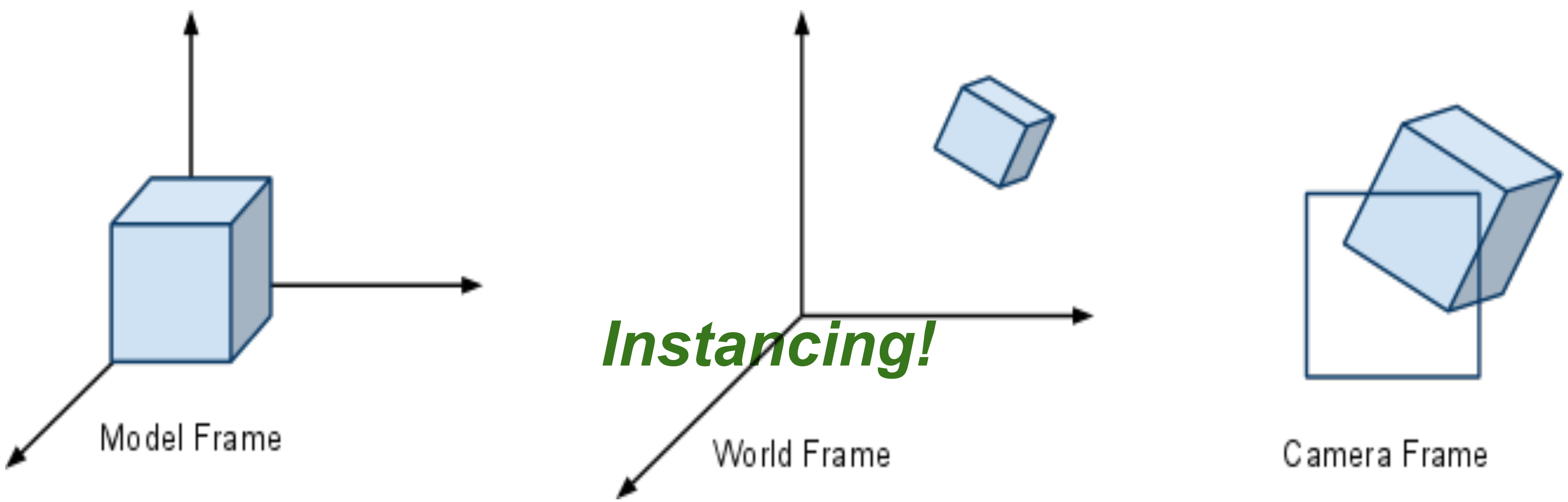
# Lecture 4: Transformations and Matrices

*CSE 40166 Computer Graphics (Fall 2010)*

# Overall Objective

- Define object in ***object frame***
- Move object to ***world/scene frame***
- Bring object into ***camera/eye frame***

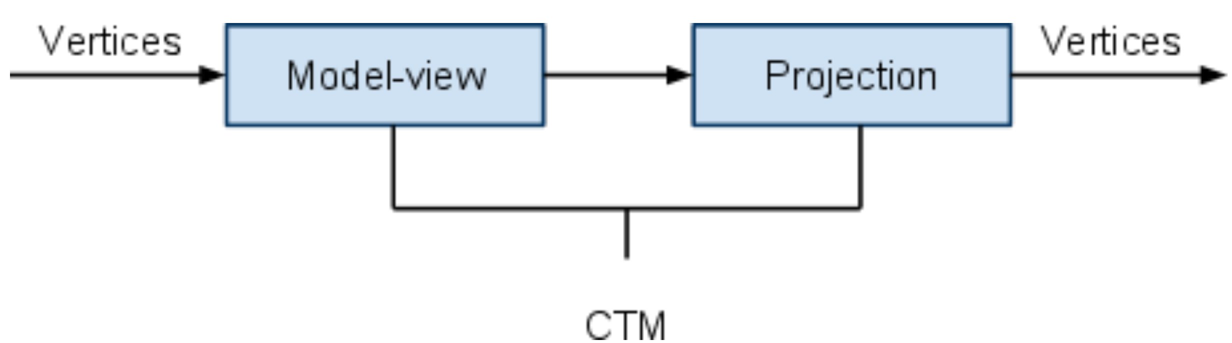


# Graphics... how does it work?

- **Linear Algebra and geometry** *(magical math)*

  Frames are represented by ***tuples*** and we change frames (representations) through the use of ***matrices***.
- **In OpenGL, vertices are modified by the Current Transformation Matrix (CTM)**

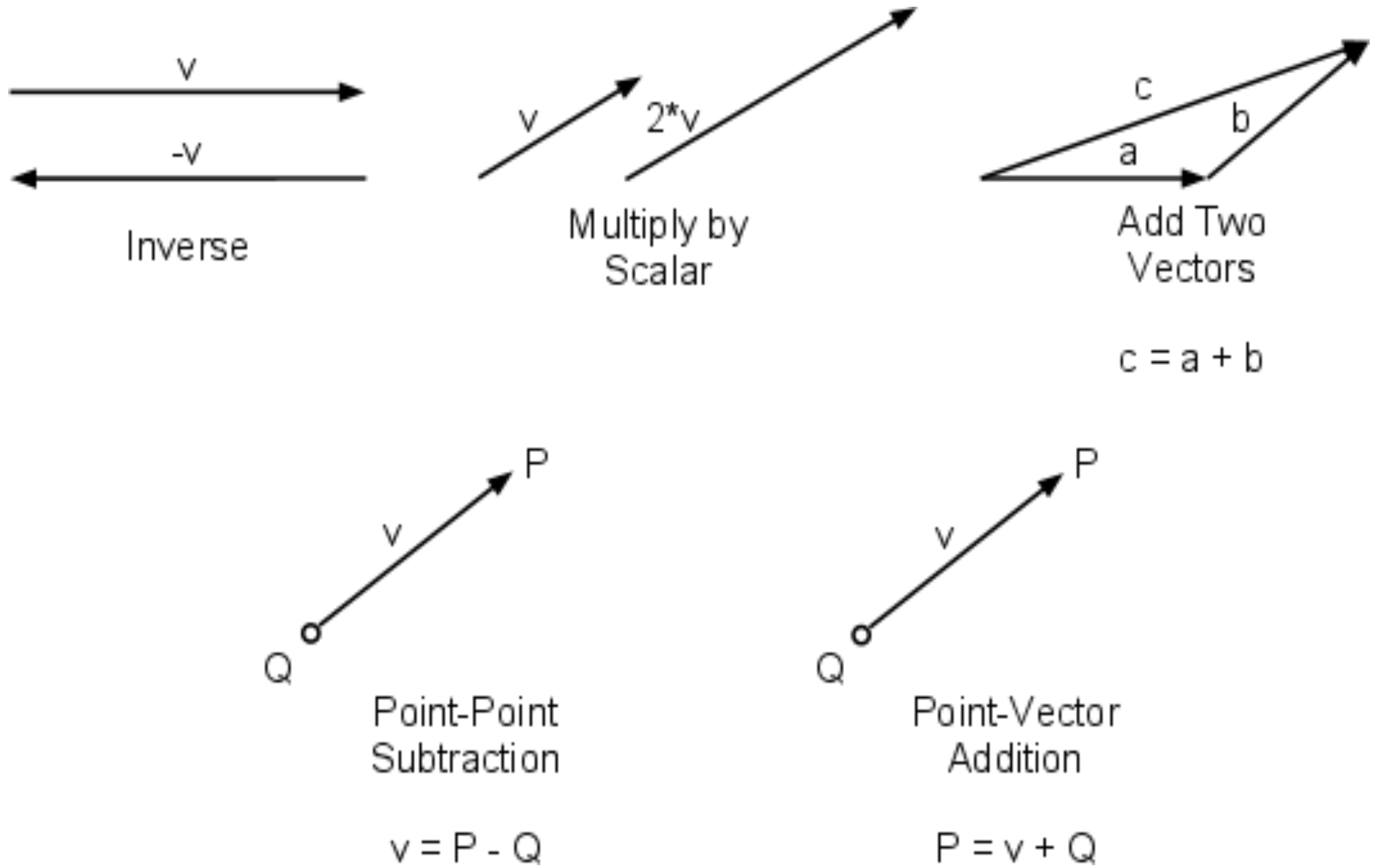
  4x4 homogeneous coordinate matrix that is part of the state and applied to all vertices that pass down the pipeline.



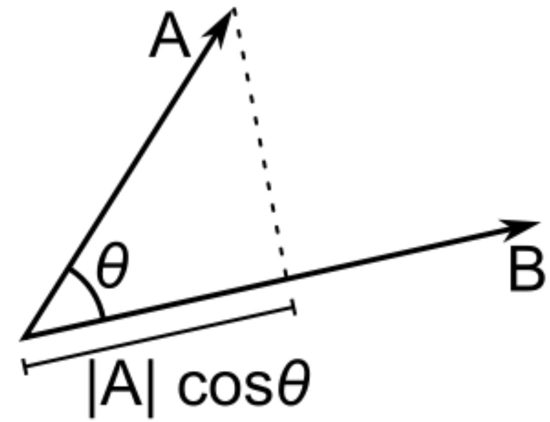
# Basic Geometric Elements

- **Scalars:** members of sets which can be combined by two operations (addition, multiplication).
  - Real numbers.
  - No geometric properties.
- **Vectors:** a quantity with both direction and magnitude.
  - Forces, velocity
  - Synonymous with *directed line segment*
  - Has no fixed location in space
- **Points:** location in space. (neither size nor shape).

# Basic Geometric Operations



# Vector Operations




## Dot Product

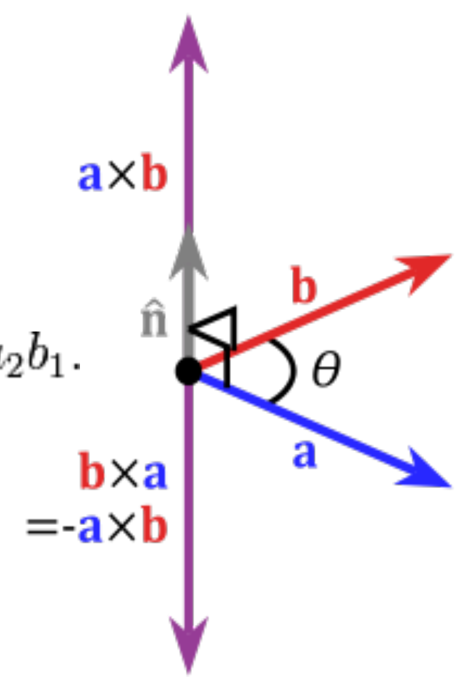
- Viewed as projection of one vector on another

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

## Cross Product

- Result is vector perpendicular to originals

$$\mathbf{a} \times \mathbf{b} = \mathbf{i}a_2b_3 + \mathbf{j}a_3b_1 + \mathbf{k}a_1b_2 - \mathbf{i}a_3b_2 - \mathbf{j}a_1b_3 - \mathbf{k}a_2b_1.$$




(images from wikipedia)

# Affine Space

**Vectors and points exist without a reference point**

- Manipulate vectors and points as abstract geometric entities

**Linear Vector Space**

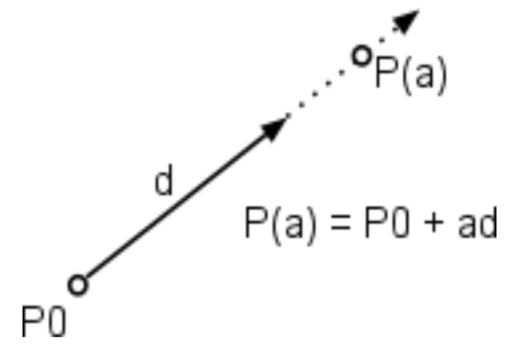
- Mathematical system for manipulating vectors

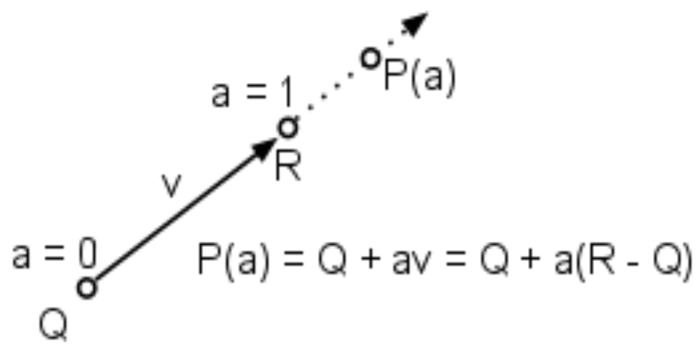
**Affine Space**

- Vector space + points

# Lines, Rays, Segments

- **Line:** Set of all points that pass through $\mathbf{P_0}$ in the direction of **d**
- **Ray: a** >= 0
- **Segments:** 0 <= **a** <= 1



# Curves and Surfaces

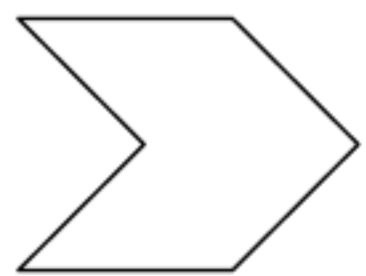
## Curves

- One parameter entities of the form **P(a)** where the function is nonlinear

## Surfaces

- Entities are formed from two-parameter functions **P(a, b)**

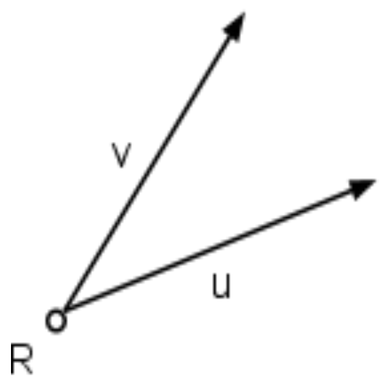
P(a)



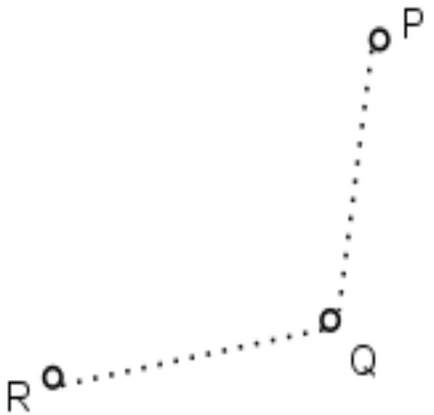
P(a, b)

# Planes

A **plane** can be defined by either a **point and two vectors**, or by **three non-collinear points**.




$P(a, b) = R + au + Bv$

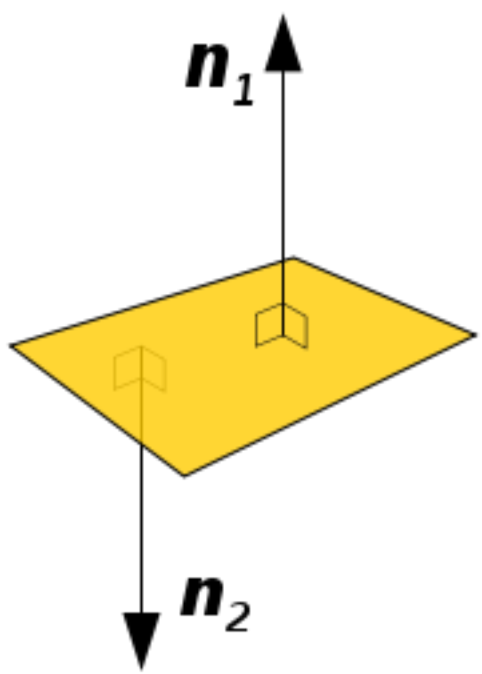


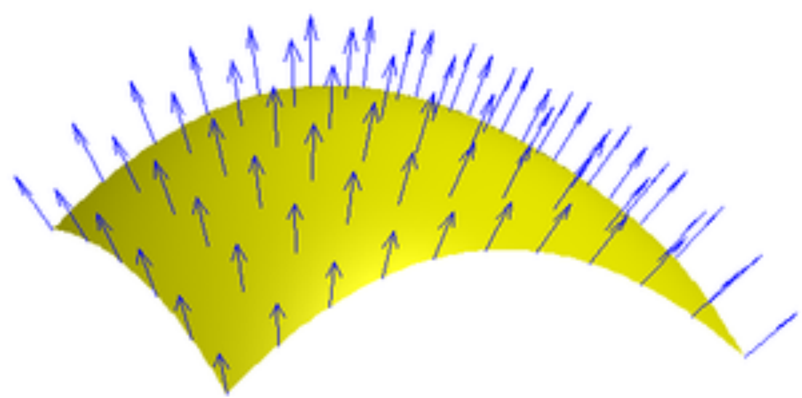

$P(a, b) = R + a(Q - R) + b(P - Q)$

# Normals

Every plane has a vector **n** normal (perpendicular, orthogonal) to it.

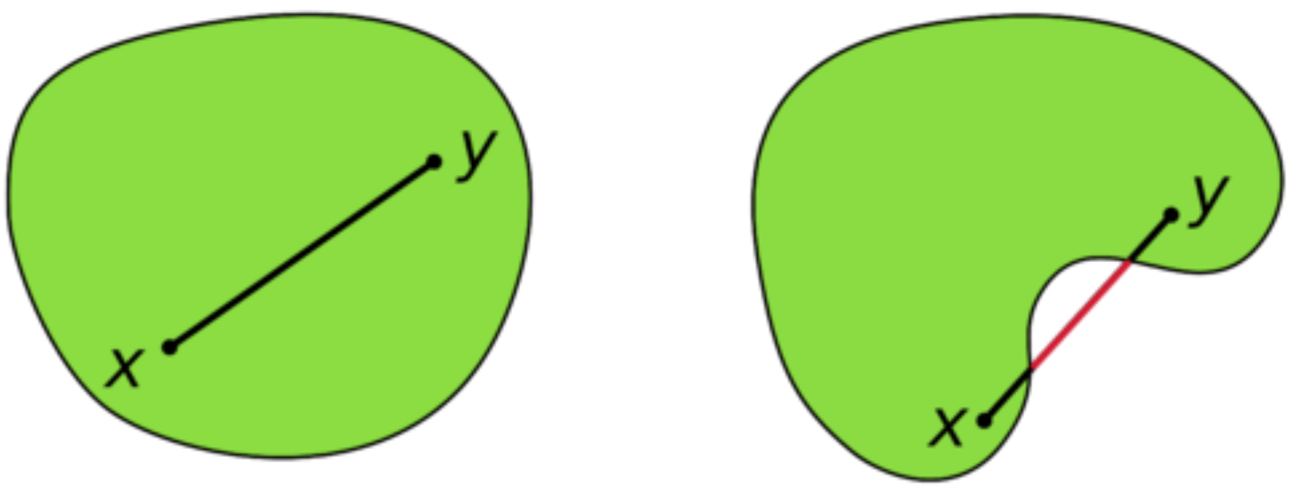
Surfaces have multiple normals.



# Convexity

An object is **convex** iff for any two points in the object, all points on the line segment between these points are also in the object.

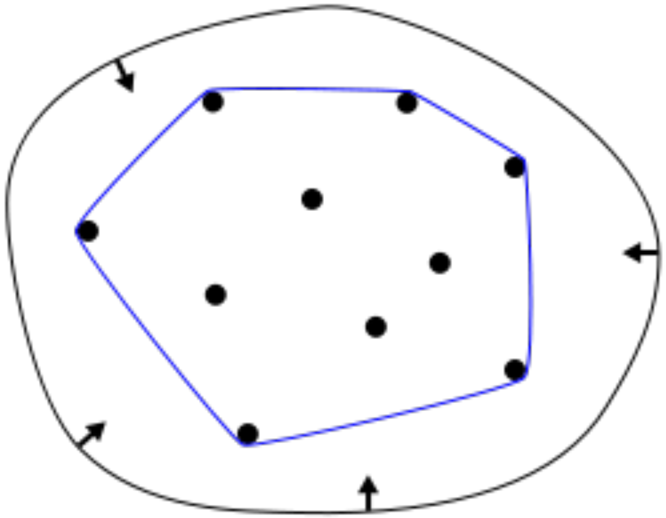



convex non-convex

# Convex Hull

Smallest convext object containing all points **Pi** in

$$P = a_1P_1 + a_2P_2 + ... + a_nP_n$$



*Formed by "shrink wrapping" points*

# Linear Independence and Dimension

## Linear Independence

If a set of vectors is **linearly independent**, we cannot represent one in terms of the others:

## Dimension

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = \mathbf{0}.$$

In a vector space, the maximum number of linearly independent vectors is fixed and is called the **dimension**.

In an **n**-dimensional space, any set of **n** linearly independent vectors form a **basis** for the space.

Given a basis $\mathbf{v_1}$, $\mathbf{v_2}$, **...** $\mathbf{v_n}$, any vector **v** can be written: v = $\mathbf{a_1 v_1} + \mathbf{a_2 v_2} + \ldots + \mathbf{a_n v_n}$

# Coordinate Systems

- Thus far, we have been able to work with geometric entities without using any frame of reference or coordinate system
- However, we need a frame of reference to relate points and objects in our abstract mathematical space to our physical world
    - Where is a point?
    - How does object map to world coordinates?
    - How does object map to camera coordinates?

# Representation

- Consider a basis $\mathbf{v}_1$, $\mathbf{v}_2$, ..., $\mathbf{v}_n$, a vector **v** is written as
  - $\mathbf{v} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + ... + a_n\mathbf{v}_n$
-
- The list of scalars $\{a_1, a_2, ..., a_n\}$ is the **representation** of **v** with respect to the given **basis:**
  - $\mathbf{v}_1 = \mathbf{e}_1 = (1, 0, 0)^T$
  - $\mathbf{v}_2 = \mathbf{e}_2 = (0, 1, 0)^T$
  - $\mathbf{v}_3 = \mathbf{e}_3 = (0, 0, 1)^T$
  - $\mathbf{a} = [a1, a2, a3]^T$

# Homogeneous Coordinates

- Using 3-tuples, it is not possible to distinguish between points and vectors:
  - $\mathbf{v} = [a_1, a_2, a_3]$
  - $\mathbf{p} = [b_1, b_2, b_3]$
- By adding a 4th coordinate component, we can use the same representation for both:
  - $\mathbf{v} = [a_1, a_2, a_3, 0]^T$
  - $\mathbf{p} = [b_1, b_2, b_3, 1]^T$

# Change of Representation

We can represent one frame in terms of another by applying a transformation matrix **C**:

$$\mathbf{a} = \mathbf{C}\mathbf{b} = \mathbf{M}^T\mathbf{b}$$

**where**

$$\mathbf{M}^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Matrices in Computer Graphics

- In OpenGL, we have multiple **frames**: model, world, camera frame
- To change frames or representation, we use **transformation matrices**
  - All standard transformations (rotation, translation, scaling) can be implemented as matrix multiplications using 4x4 matrices (concatenation)
  - Hardware pipeline optimized to work with 4-dimensional representations

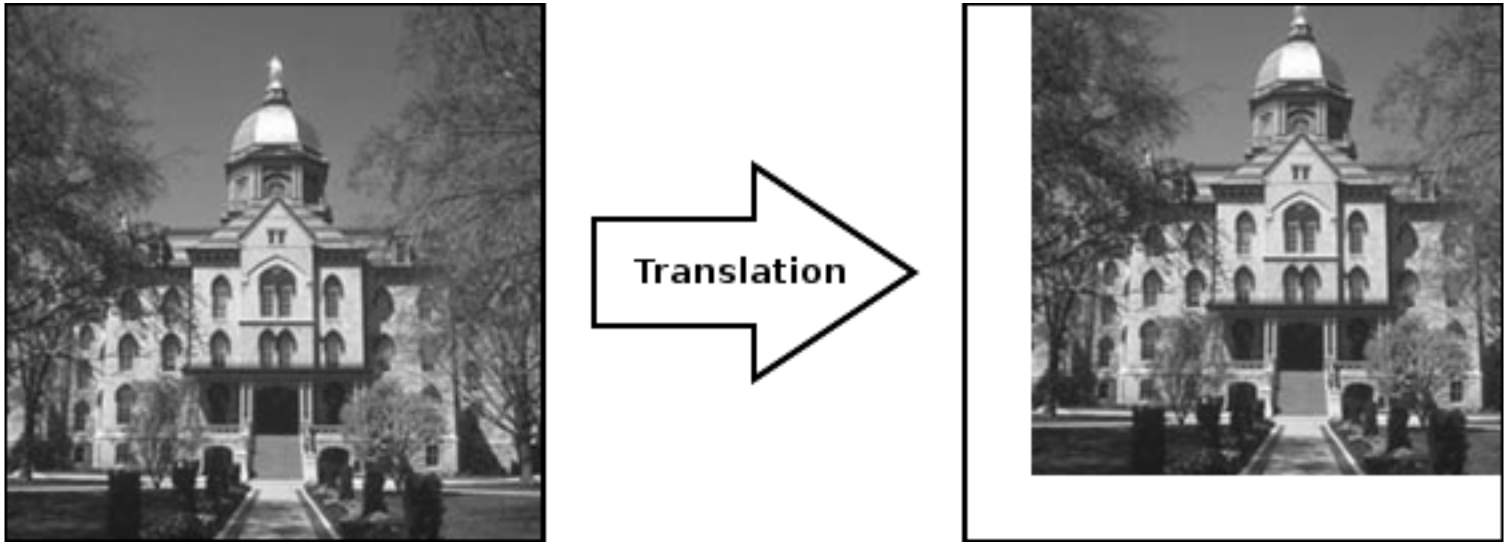
# Affine Transformations

- Tranformation maps points/vectors to other points/vectors
- Every affine transformation *preserves lines*
  - Preserve collinearity
  - Preserve ratio of distances on a line
- Only have *12 degrees of freedom* because 4 elements of the matrix are fixed **[0 0 0 1]**
- Only comprise a *subset* of possible linear transformations
  - **Rigid body:** translation, rotation
  - **Non-rigid:** scaling, shearing

# Translation

Move (translate, displace) a point to a new location:

$$P' = P + d$$



# Translation Matrix

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

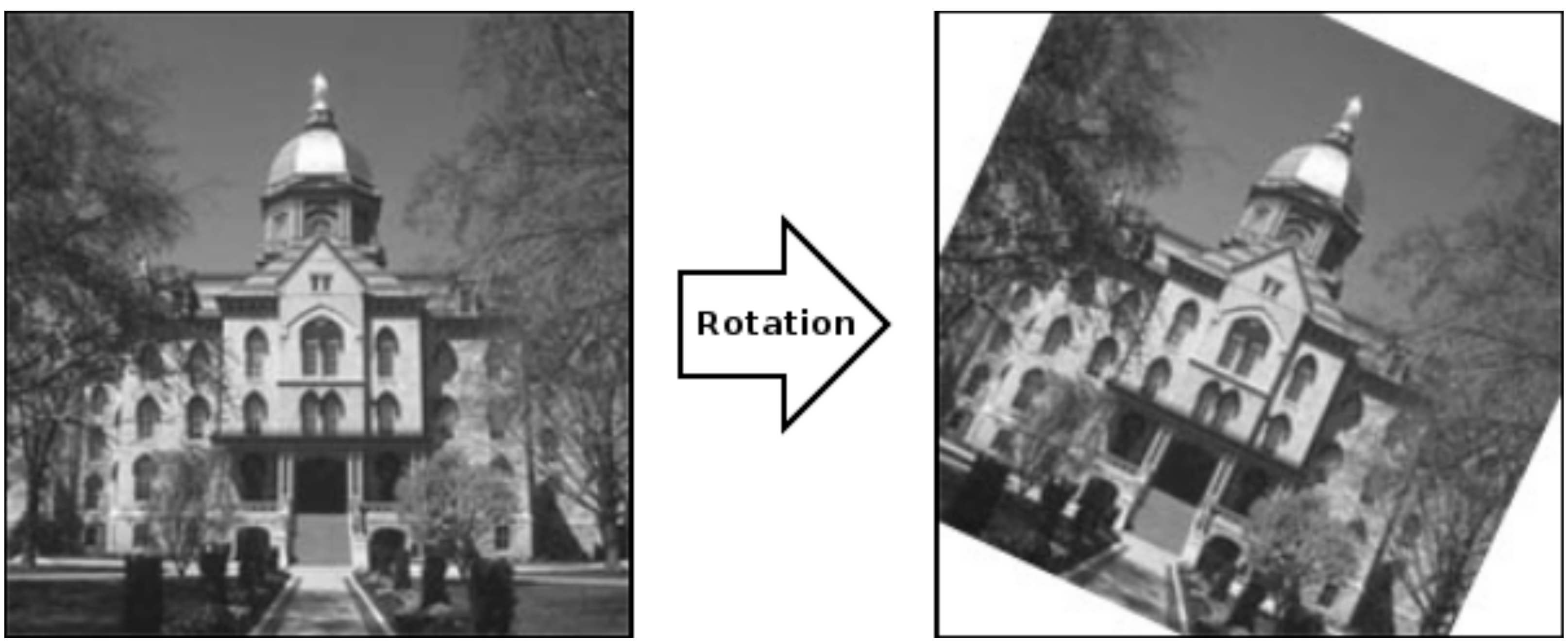
resulting coordinate — 3d translation matrix — original coordinate

# Rotation (about an axis)

Rotation about **z** axis leaves all points with the same **z**:

- **$x' = x \cos(t) - y \sin(t)$**
- **$y' = x \sin(t) + y \cos(t)$**
- **$z' = z$**

$$P' = R_z(t)P$$



# Rotation About Z Axis Matrix

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

resulting coordinate — 3d rotation matrix in Z — original coordinate

# Rotation About X Axis Matrix

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

resulting coordinate

3d rotation matrix in X

original coordinate

# Rotation About Y Axis Matrix

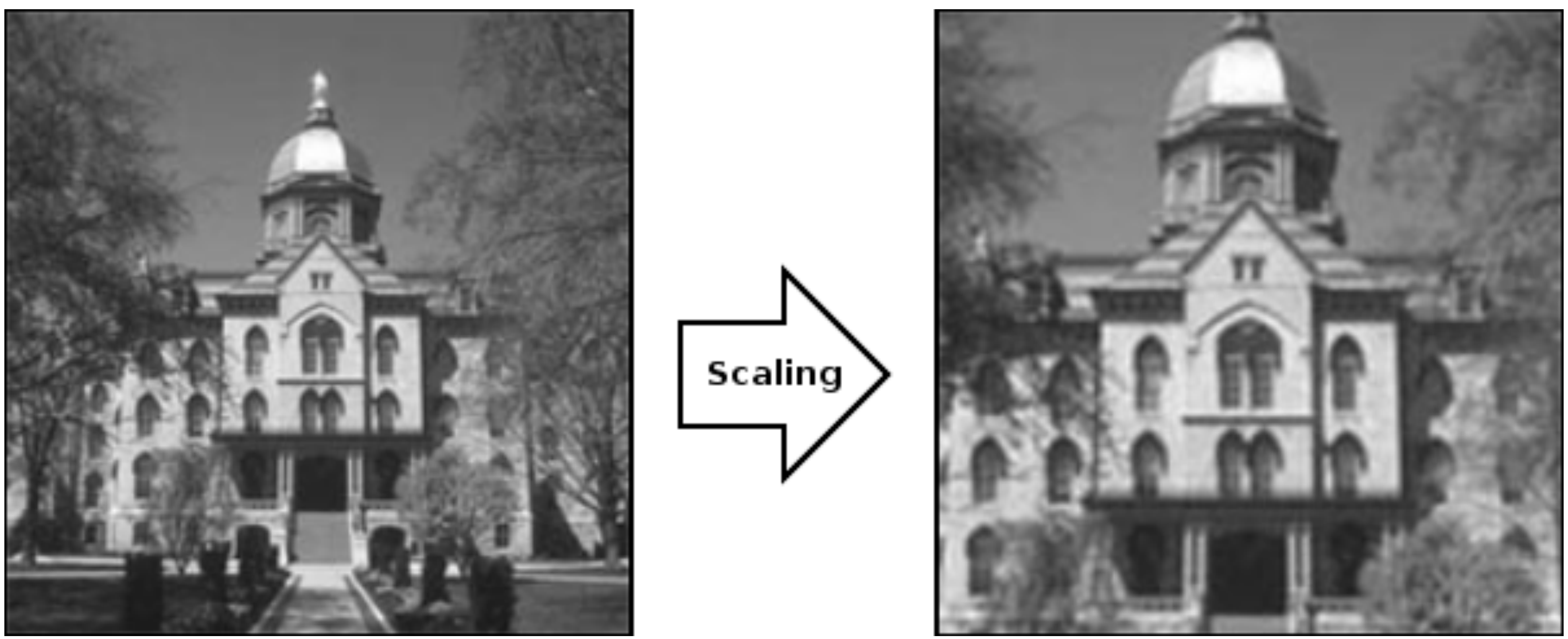
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

resulting coordinate — 3d rotation matrix in Y — original coordinate

# Scaling

Expand or contract along each axis (fixed point of origin)

**P' = SP**



# Scaling Matrix

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

resulting coordinate — 3d scaling matrix — original coordinate

***If sx, sy, sz are negative, then we will perform reflection.***

# Concatenation

To form arbitrary affine transformation matrices we can ***multiply together*** translation, rotation, and scaling matrices:

**p' = ABCDp**

To optimize the computation, we group the transformation matrices:

**p' = Mp where M = ABCD**

This saves us the cost of multiplying every vertex by multiple matrices; instead we multiply by just one.

# Order of Transformations

The ***right matrix*** is the first applied to the vertex:

$$\mathbf{p' = ABCp = A(B(Cp))}$$

Sometimes we may use column matrices to represent points, so this equation becomes:

$$\mathbf{p'^T = p^T C^T B^T A^T}$$

# OpenGL Matrices

**In OpenGL matrices are part of the state**

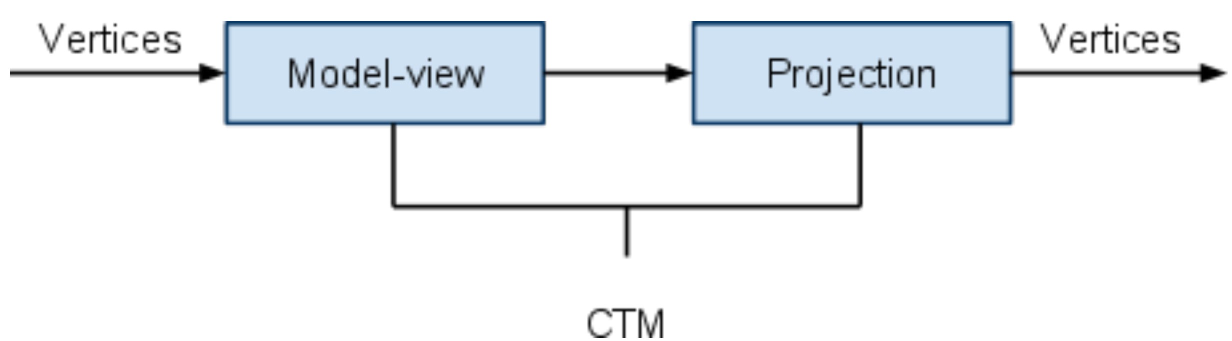
- `GL_MODELVIEW`
- `GL_PROJECTION`
- `GL_TEXTURE`
- `GL_COLOR`

Select which matrix to manipulate by using **glMatrixMode**:

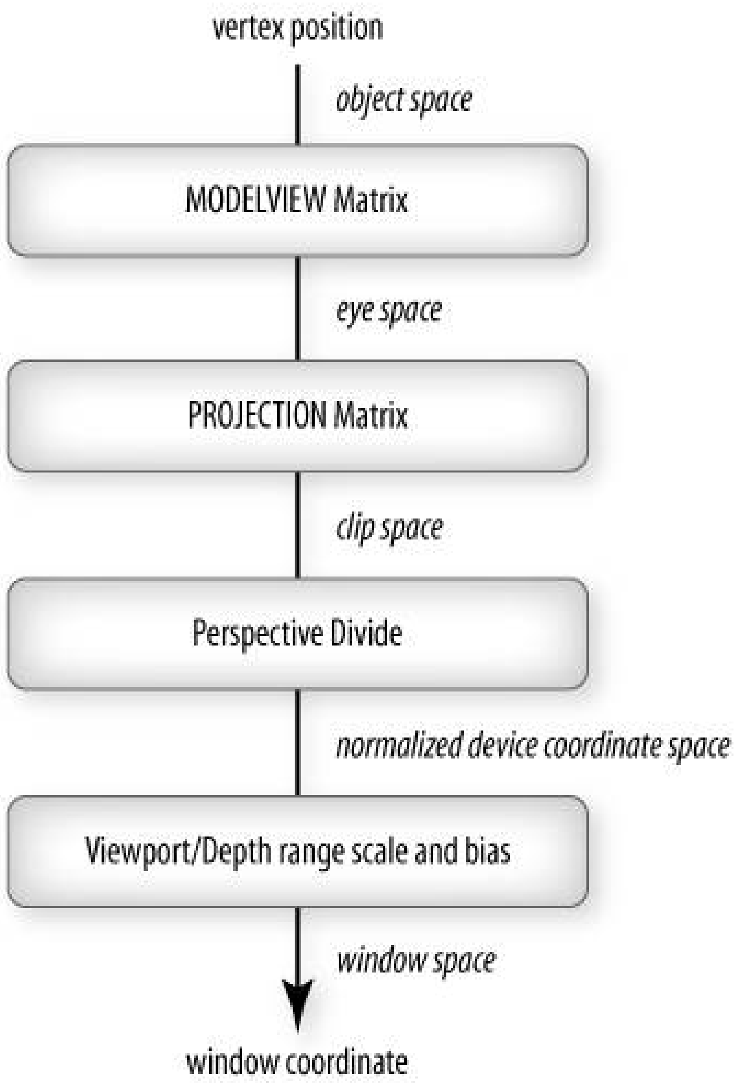
```
glMatrixMode(GL_MODELVIEW);
```

# Current Transformation Matrix (CTM)

Conceptually there is a 4x4 homogeneous coordinate matrix, the **current transformation matrix (CTM)**, that is part of the state and is applied to all vertices that pass down the pipeline.



# Transformation Pipeline



# CTM Operations

## Loading a 4x4 Matrix:

- `glLoadIdentity()  C <- I`
- `glLoadMatrix(M)   C <- M`

## Postmultiplying by another 4x4 Matrix:

- `glTranslatef(dx, dy, dz)          C <- MT`
- `glRotatef(theta, vx, vy, vz)      C <- MTR`
- `glScalef(sx, sy, sz)              C <- MTRS`

## Saving and Restoring Matrix:

- `glPushMatrix()`
- `glPopMatrix()`

# Instancing

In modeling, we start with a simple object centered at the origin, oriented with some axis, and at a standard size.

To instantiate an object, we apply an instance transformation:

- Scale
- Orient
- Locate

***Remember the last matrix specified in the program is the first applied!***

# Translate, Rotate, Scale (TRS)

***Remember the last matrix specified in the program is the first applied!***

For instancing, you want to scale, rotate, and then translate:

```
glPushMatrix();
glTranslatef(i->x, i->y, 0.0);
glRotatef(i->angle, 0.0, 0.0, 1.0);
glScalef(10.0, 10.0, 1.0);
glCallList(DisplayListsBase + MissileType);
glPopMatrix();
```