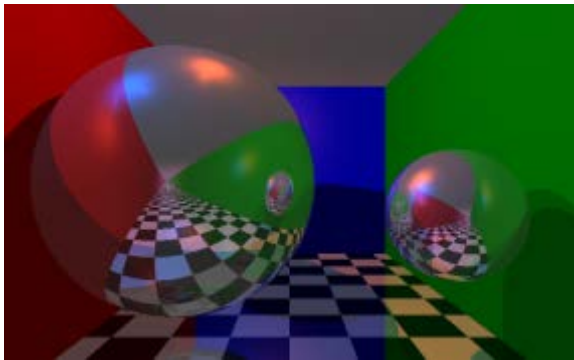
CSI 4105 Computer Graphics

Spring 2017

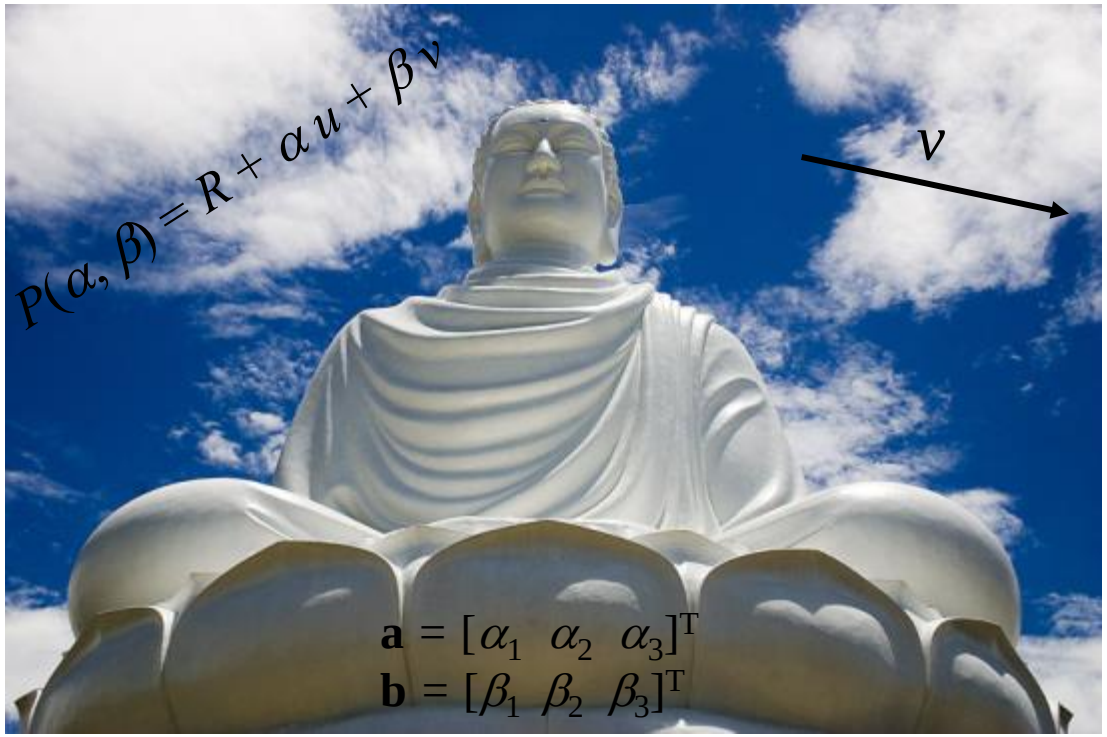
Lecture 4:
Geometry and Transformation

Seon Joo Kim
Yonsei University



Clear Your Mind

- The first part of this lecture will seem a bit abstract, but just clear your head and start thinking in terms of higher mathematics



Our Basic Elements

- We will need three basic elements
  - Scalars
  - Vectors
  - Points

Remember naming convention:
Points will be in uppercase Roman Letters (e.g. P, Q, R)
Vectors will be in lowercase Roman Letters (e.g. u, v, w) sometimes with arrows over them.
Scalars will be in Greek letters (α,β,γ)

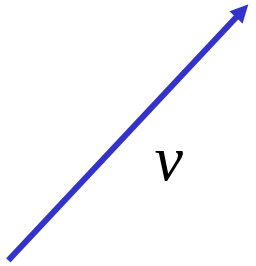
Scalars

- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutativity, inverses)
- Examples include the real and complex number systems under the ordinary rules with which we are familiar
- **Scalars alone** have no geometric properties

http://en.wikipedia.org/wiki/Scalar_(mathematics)

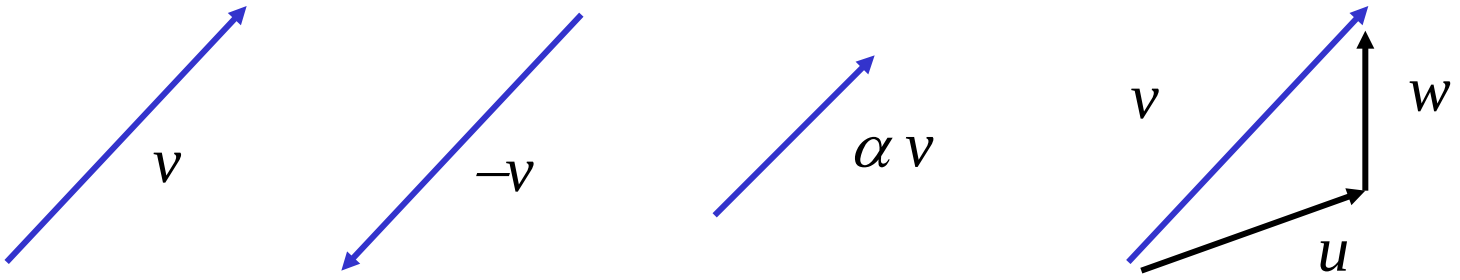
Vectors

- Physical definition: a vector is a quantity with two attributes
  - Direction
  - Magnitude
- Examples include
  - Force
  - Velocity
  - Directed line segments
    - Most important example for graphics
    - Can map to other types



Vector Operations

- Every vector has an “inverse”
  - Same magnitude but points in opposite direction
- Every vector can be multiplied by a scalar
- There is a zero vector
  - Zero magnitude, undefined orientation
- The sum of any two vectors is a vector
  - Use head-to-tail axiom



Linear Vector Spaces

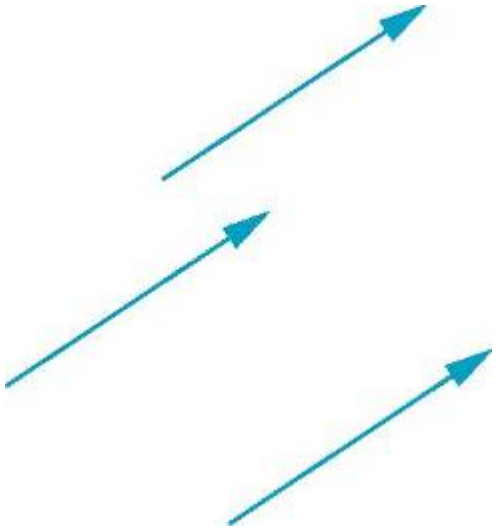
- Mathematical system for manipulating vectors
- Operations
  - Scalar-vector multiplication: $u = \alpha v$
  - Vector-vector addition: $w = u + v$
- Expressions such as

  $v = u + 2w - 3r$

  make sense in a vector space.

Remember Vectors Lack Position!

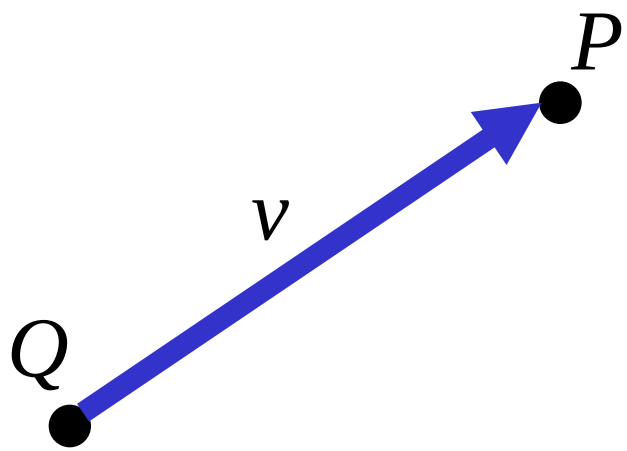
- These vectors are identical
  - Same length and magnitude



- Vectors spaces **insufficient** to express geometry
  - Need points

Points

- Location in space
- Operations allowed between points and vectors
  - Point-point subtraction yields a vector
  - Equivalent to point-vector addition



$$v = P - Q$$

$$P = v + Q$$

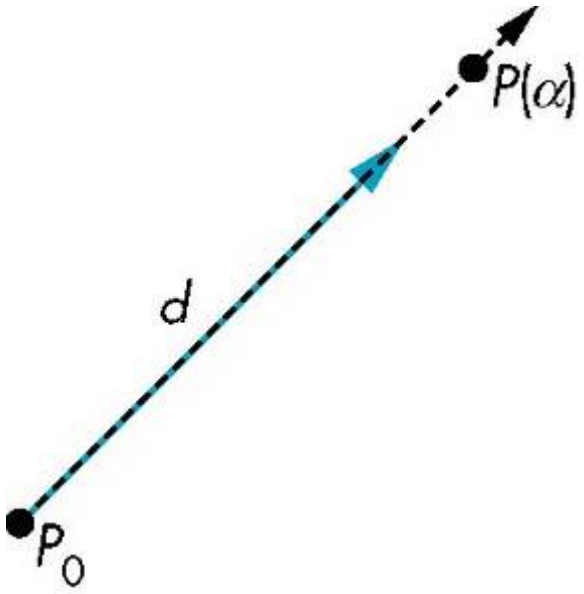
Affine Spaces

- Point + a vector space
  - Imagine a space where the origin is arbitrary (i.e. some point, not necessarily 0,0,0)
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
- For any point, we must define
  - $1 \bullet P = P$
  - $0 \bullet P = \mathbf{0}$ (zero vector)

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

Lines

- Consider all points of the form
  - $P(\alpha) = P_0 + \alpha d$
  - The set of all points that pass through P_0 in the direction of the vector d
    - Note that α can be negative



Parametric Line Form

- This form is known as the parametric form of the line
  - More robust and general than other forms
  - Extends to curves and surfaces
- Two-dimensional forms
  - Explicit: $y = mx + h$
  - Implicit: $ax + by + c = 0$
  - Parametric:

$$x(\alpha) = \alpha\, x_0 + (1 - \alpha)\, x_1$$
$$y(\alpha) = \alpha\, y_0 + (1 - \alpha)\, y_1$$

same as
$$P(\alpha) = \alpha\, P_0 + (1 - \alpha)\, P_1$$

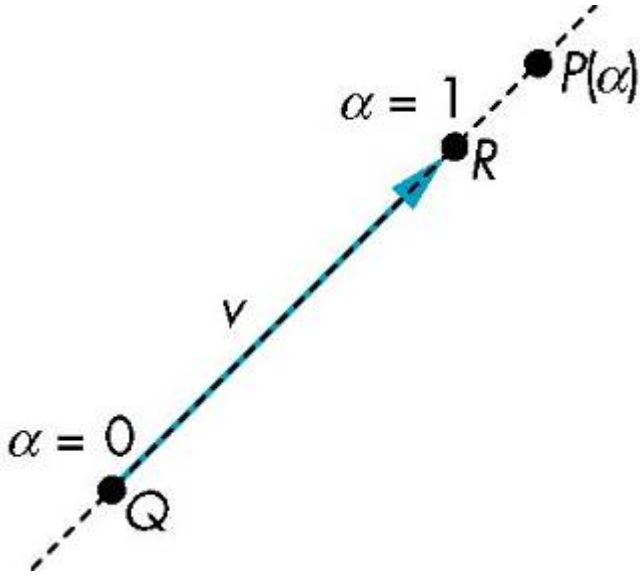
Rays and Line Segments

- If $\alpha \geq 0$, then $P(\alpha)$ is the *ray* leaving P_0 in the direction d

- If we use two points to define v, then

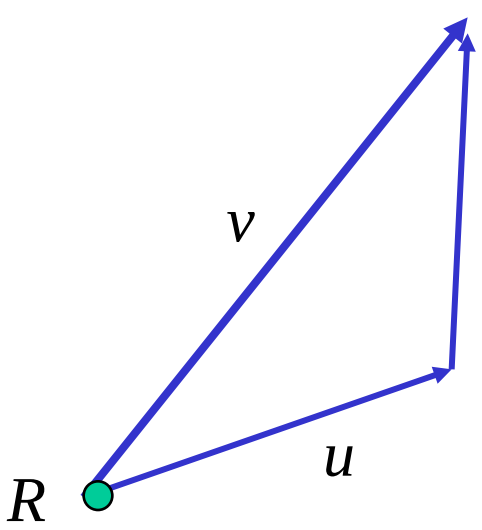
$$\begin{aligned} P(\alpha) &= Q + \alpha(R - Q) \\ &= Q + \alpha v \\ &= \alpha R + (1 - \alpha)Q \end{aligned}$$

For $0 \leq \alpha \leq 1$, we get all the points on the *line segment* joining R and Q

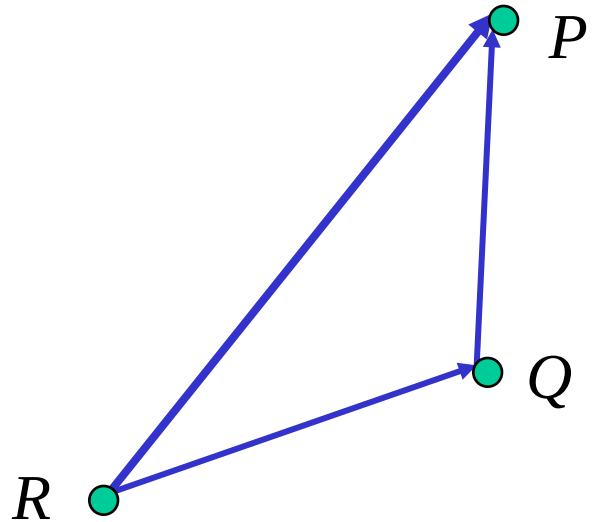


Planes

- A plane can be defined by 1) a point and two vectors; or 2) by three points



$$P(\alpha, \beta) = R + \alpha u + \beta v$$



$$P(\alpha, \beta) = R + \alpha (Q - R) + \beta (P - Q)$$

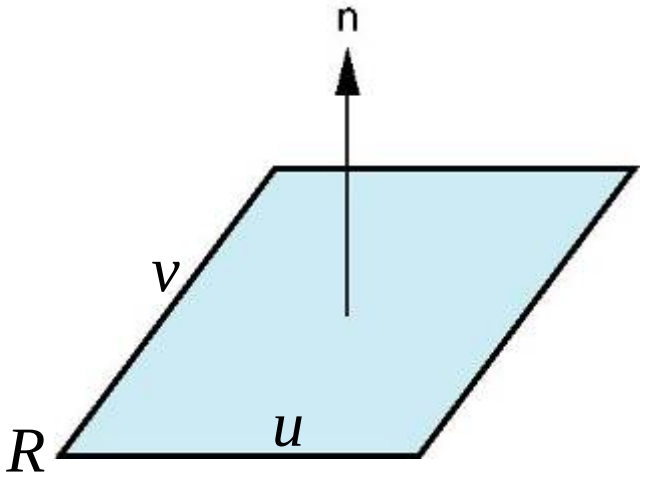
Normals

- The one-point-two-vector form plane,

$$P(\alpha, \beta) = R + \alpha u + \beta v$$

- Every plane has a vector n normal (perpendicular, orthogonal) to it
- We can use the cross product to find $n = u \times v$, and the plane has the equivalent implicit form

$$(P - R) \cdot n = 0$$



where P,R are any points on the plane

  - How is it related to the form $ax + by + cz + d = 0$?
    (Think about it, I won't answer in class, we can talk about it later)

Representation

Objectives

- Introduce coordinate systems for representing vectors spaces and frames for representing affine spaces
- Discuss change of frames and bases
- Introduce homogeneous coordinates

Representation

- Until now we have been able to work with geometric entities without using any frame of reference, such as a coordinate system
- **We need** a frame of reference to relate points and objects to our physical world
  - For example, where is a point?
    - Can't answer without a reference system
  - World coordinates
  - Camera coordinates

Coordinate Systems

- Consider a *basis* $v_1, v_2, \ldots, v_n$
  - A basis is a set of linearly independent vectors
  - Linear independence means none of the vectors can be expressed by a combination of the other vectors

Coordinate Systems

- Given a *basis* $v_1, v_2, \ldots, v_n$
- A vector in this basis can be expressed as linear combination of the basis vectors:

  $w = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$
- The list of scalars $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is the *representation* of w with respect to the given basis
- We can write the representation as a row or column array of scalars

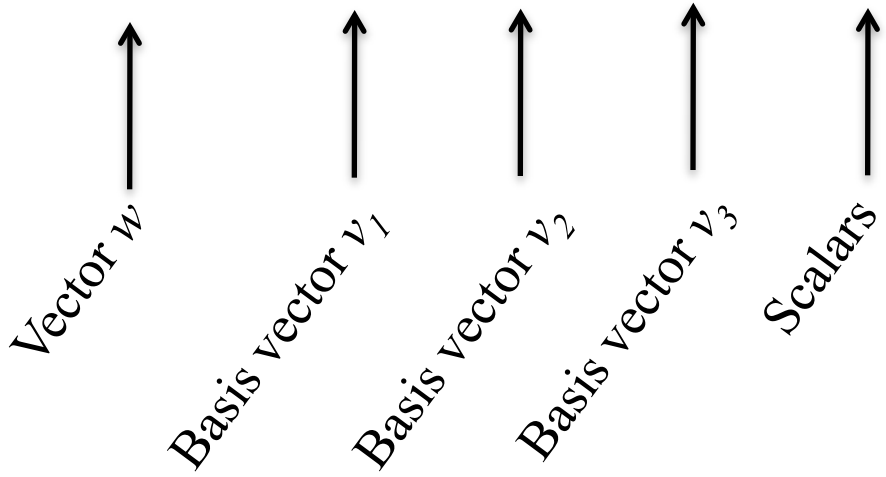
$$\mathbf{a} = [\alpha_1 \ \alpha_2 \ \ldots \ \alpha_n]^{\mathrm{T}} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

transpose operator

Writing it in Matrix Format

- $w = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$
- Expressed in matrix multiplication format (a 3x3 example):

$$\begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = \begin{bmatrix} v_{1x} & v_{2x} & v_{3x} \\ v_{1y} & v_{2y} & v_{3y} \\ v_{1z} & v_{2z} & v_{3z} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$



Alternative Notation

- $w = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$

$$w = \mathrm{a}^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix} \begin{bmatrix} v_{1x} & v_{1y} & v_{1z} \\ v_{2x} & v_{2y} & v_{2z} \\ v_{3x} & v_{3y} & v_{3z} \end{bmatrix}$$

← Basis vector v_1

← Basis vector v_2

← Basis vector v_3

Example

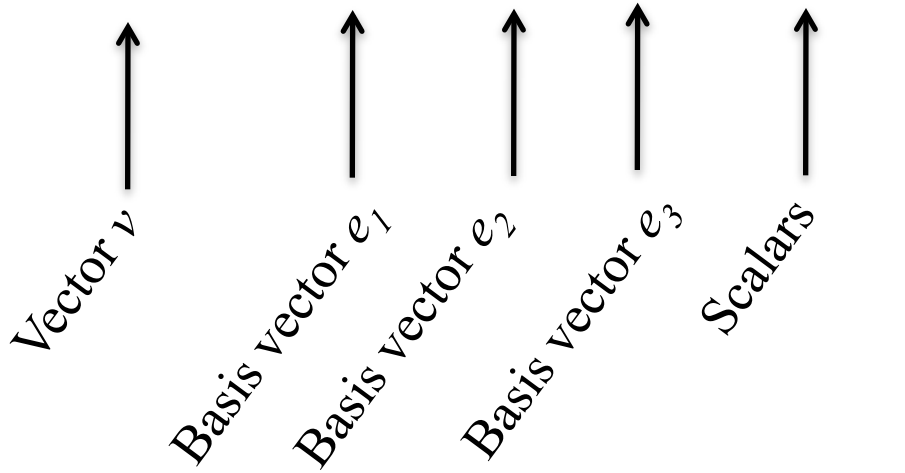
- $w = 2v_1 + 3v_2 - 4v_3$
- $\mathbf{a} = [2 \;\; 3 \; -4]^T$
- Note that this representation is with respect to a particular basis
- For example, in OpenGL we start by representing vectors using the *object or model basis* but later the system needs a representation in terms of the *camera or eye basis*

This formulation applies for all vectors. When we don't explicitly give a basis, then we assume it is the *natural basis* (or *standard basis*); i.e. $[1\ 0\ 0]^T$, $[0\ 1\ 0]^T$, $[0\ 0\ 1]^T$ basis. (sometimes we call these $e_1, e_2, e_3, \ldots, e_n$)
http://en.wikipedia.org/wiki/Standard_basis

Standard (Natural) basis

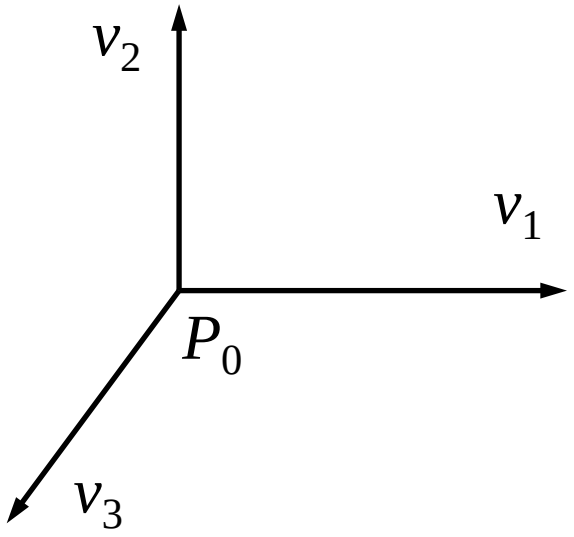
- $w = 2v_1 + 3v_2 - 4v_3$
- $\mathbf{a} = [2 \;\; 3 \;\; -4]^T$

$$\begin{bmatrix} 2 \\ 3 \\ -4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -4 \end{bmatrix}$$



Frames

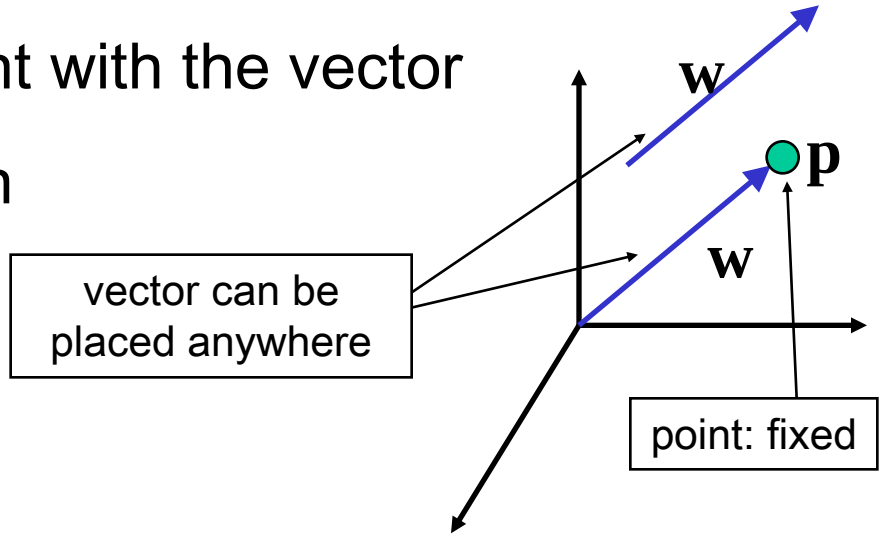
- A coordinate system is insufficient to represent points
- If we work in an affine space we can add a single point, the *origin*, to the basis vectors to form a *frame*
- Frame determined by (P_0, v_1, v_2, v_3)
- Within this frame, every vector can be written as
  $w = \alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$
- Every point can be written as
  $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + ... + \beta_n v_n$



Confusing Points and Vectors

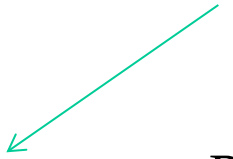
- Consider the point and the vector
  - $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + ... + \beta_n v_n$
  - $v = \alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$
- They appear to have the similar representations
  - $\mathbf{p} = [\beta_1 \ \ \beta_2 \ \ \beta_3]^T \qquad \mathbf{w} = [\alpha_1 \ \ \alpha_2 \ \ \alpha_3]^T$

  which confuses the point with the vector
- A vector has no position



A Single Representation

These are columns in the matrix.



- We can write them as:
  - $w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [v_1 \;\; v_2 \;\; v_3 \;\; P_0]\,[\alpha_1 \;\; \alpha_2 \;\; \alpha_3 \;\; 0\,]^{\mathrm{T}}$
  - $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [v_1 \;\; v_2 \;\; v_3 \;\; P_0]\,[\beta_1 \;\; \beta_2 \;\; \beta_3 \;\; 1\,]^{\mathrm{T}}$

- Thus we obtain the four-dimensional *homogeneous coordinate* representation
  - $\mathbf{w} = [\alpha_1 \;\; \alpha_2 \;\; \alpha_3 \;\; 0\,]^{\mathrm{T}}$
  - $\mathbf{p} = [\beta_1 \;\; \beta_2 \;\; \beta_3 \;\; 1\,]^{\mathrm{T}}$

Example

For vector v: $\boldsymbol{v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [v_1\ v_2\ v_3\ P_0]\ [\alpha_1\ \alpha_2\ \alpha_3\ 0]^T}$

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix} = \begin{bmatrix} v_{1x} & v_{2x} & v_{3x} & P_{0x} \\ v_{1y} & v_{2y} & v_{3y} & P_{0y} \\ v_{1z} & v_{2z} & v_{3z} & P_{0z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{bmatrix}$$

For point P: $\boldsymbol{P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [v_1\ v_2\ v_3\ P_0]\ [\beta_1\ \beta_2\ \beta_3\ 1]^T}$

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ 1 \end{bmatrix} = \begin{bmatrix} v_{1x} & v_{2x} & v_{3x} & P_{0x} \\ v_{1y} & v_{2y} & v_{3y} & P_{0y} \\ v_{1z} & v_{2z} & v_{3z} & P_{0z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ 1 \end{bmatrix}$$

Note, the matrix does not change in either formulation.

Awesome Factor (1): This might not seem very impressive, but it really is, because we now have a mathematic way to handle both points and vectors into a single representation.

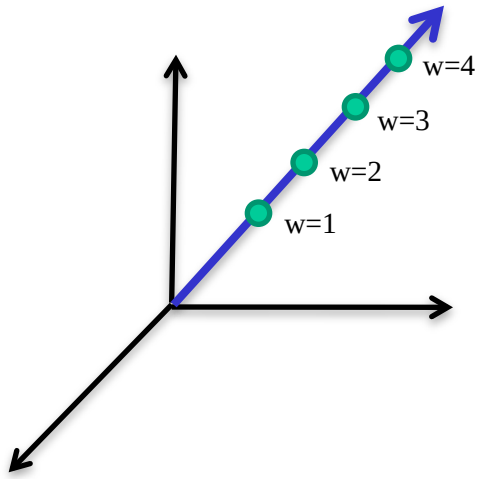
Homogeneous Coordinates

- The homogeneous coordinates for a three dimensional point $[x \;\; y \;\; z]^{\mathrm{T}}$ is given as
  - $\mathbf{p} = [x' \; y' \; z' \; w]^{\mathrm{T}} = [wx \;\; wy \;\; wz \;\; w]^{\mathrm{T}}$
- We return to a three dimensional point (for $w \neq 0$) by
  - $x \leftarrow x' / w \qquad y \leftarrow y' / w \qquad z \leftarrow z' / w$
- If $w = 0$, the representation is that of a vector
- Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions
- For $w = 1$, the representation of a point is $[x \;\; y \;\; z \;\; 1]^{\mathrm{T}}$

Awesome Factor (2): In homogenous coordinates, not only can we represent points and vectors, but a point can actually be thought of as a ray passing through the origin (by adjusting w).

Homogenous Coordinates

$$\begin{bmatrix} wP_1 \\ wP_2 \\ wP_3 \\ w \end{bmatrix} = \begin{bmatrix} v_{1x} & v_{2x} & v_{3x} & P_{0x} \\ v_{1y} & v_{2y} & v_{3y} & P_{0y} \\ v_{1z} & v_{2z} & v_{3z} & P_{0z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w\beta_1 \\ w\beta_2 \\ w\beta_3 \\ w \end{bmatrix}$$



We will talk about this more later, but in a way this implies that the mathematics can handle uniform scale changes in a point's location without changing any of the mathematical machinery of the 4x4 matrix. (see, the matrix is still the same).

Homogeneous Coordinates and Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
  - Hardware pipelines (e.g. nVidia, ATI) works with 4 dimensional representations
  - For orthographic viewing, we can maintain $w = 0$ for vectors and $w = 1$ for points
  - For perspective viewing, we need an additional *perspective division,* but everything else is done by the 4x4 matrix
    (we will talk about viewing in the next lecture)

Change of Coordinate Systems

- Consider two representations of the same vector w with respect to two different bases

$$\mathbf{a} = [\alpha_1 \;\; \alpha_2 \;\; \alpha_3]^T$$

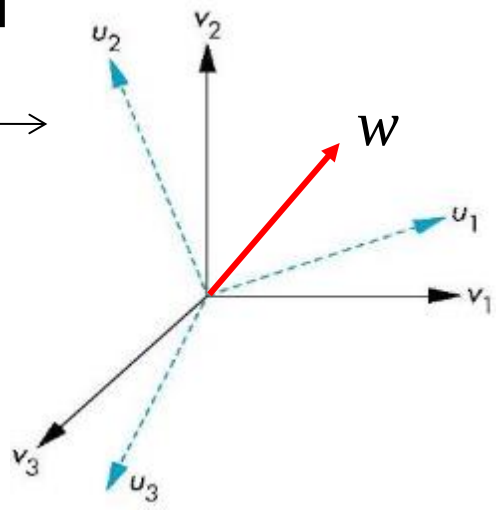
$$\mathbf{b} = [\beta_1 \;\; \beta_2 \;\; \beta_3]^T$$

where

$$w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [v_1 \;\; v_2 \;\; v_3]\,[\alpha_1 \;\; \alpha_2 \;\; \alpha_3]^T$$

$$= \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3 = [u_1 \;\; u_2 \;\; u_3]\,[\beta_1 \;\; \beta_2 \;\; \beta_3]^T$$

Think about this carefully - The vector w can be expressed using either the basis v_1, v_2, v_3 or u_1, u_2, u_3. Of course the associate coefficients (a_1, a_2, a_3) and (b_1, b_2, b_3) will be different in the different basis.



Representing Second Basis in Terms of First

- Each of the basis vectors, u_1, u_2, u_3, are vectors that can be represented in terms of the other base $\{v_1, v_2, v_3\}$

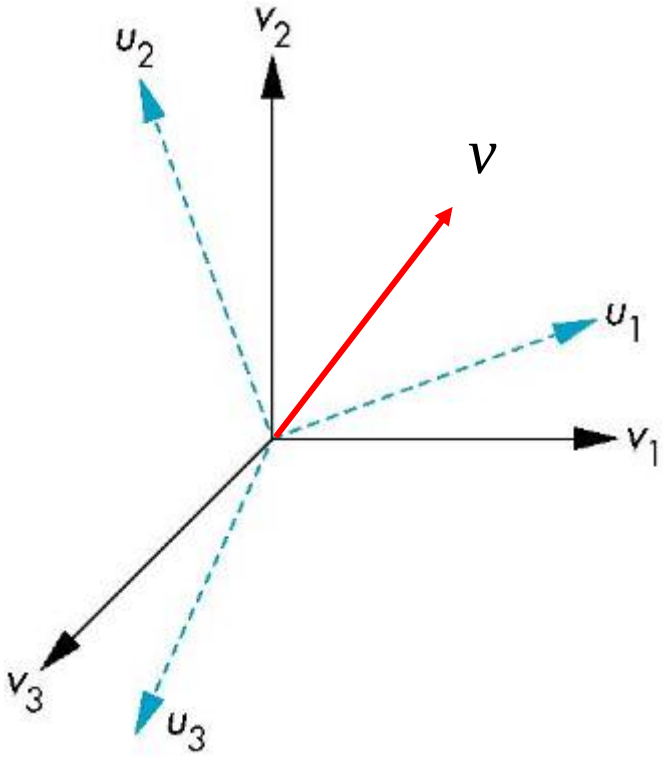
$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$

$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$

$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$

We can find a transformation that expresses the basis vectors u_1, u_2, u_3 as a combination of the other basis v_1, v_2, v_3. Apply this type of transform is known as a "change of basis".

We will talk about how to find this transform later . .now, just think abstractly.



Matrix Form

- The coefficients define a 3 x 3 matrix

$$\begin{bmatrix} - & u1 & - \\ - & u2 & - \\ - & u3 & - \end{bmatrix} = \begin{bmatrix} \gamma_{11} & \gamma_{21} & \gamma_{31} \\ \gamma_{12} & \gamma_{22} & \gamma_{32} \\ \gamma_{13} & \gamma_{23} & \gamma_{33} \end{bmatrix} \begin{bmatrix} - & v1 & - \\ - & v2 & - \\ - & v3 & - \end{bmatrix}$$

and the bases can be related by

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

We need to add a transpose because our notion is changing from row notion to column.

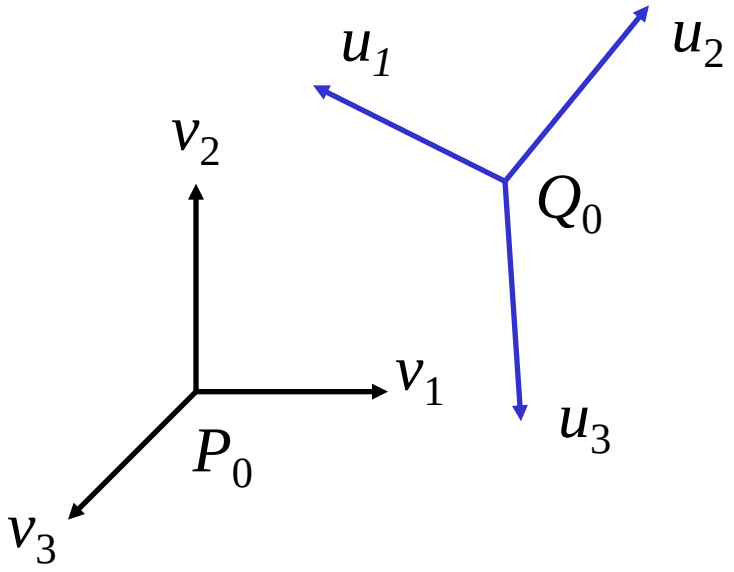
where

$$\mathbf{a} = [\alpha_1 \;\; \alpha_2 \;\; \alpha_3]^{\mathrm{T}}$$

$$\mathbf{b} = [\beta_1 \;\; \beta_2 \;\; \beta_3]^{\mathrm{T}}$$

Change of Frames

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors
- Consider two frames
  - (P_0, v_1, v_2, v_3)
  - (Q_0, u_1, u_2, u_3)
- Any point or vector can be represented in either frame
- We can represent Q_0, u_1, u_2, u_3 in terms of P_0, v_1, v_2, v_3



Representing One Frame in Terms of the Other

- Extending what we did with change of bases, we have

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$

$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$

$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$

$$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + P_0$$

- The coefficients define a 4 x 4 matrix

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

*I've already transposed the matrix. Just trust me (I'm following the book's notation). What is important is not the transpose, but the format of the matrix – a 4x4). We will often just call it "M".

Working with Representations

- Within the two frames any point or vector has a representation of the same form

  $\mathbf{a} = [\alpha_1 \ \alpha_2 \ \alpha_3 \ \alpha_4]^T$ in the first frame

  $\mathbf{b} = [\beta_1 \ \beta_2 \ \beta_3 \ \beta_4]^T$ in the second frame

  where $\alpha_4 = \beta_4 = 1$ for points and $\alpha_4 = \beta_4 = 0$ for vectors

- They are related by

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

- The matrix $\mathbf{M}$ is 4 x 4 and specifies an affine transformation in homogeneous coordinates

The World and Camera Frames

- When we work with representations, we work with n-tuples or arrays of scalars
- Changes in frame are then defined by 4 x 4 matrices
- In OpenGL, the base frame that we start with is the world frame
- Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix
- Initially these frames are the same ($\mathbf{M} = \mathbf{I}$)

Transformations

Objectives

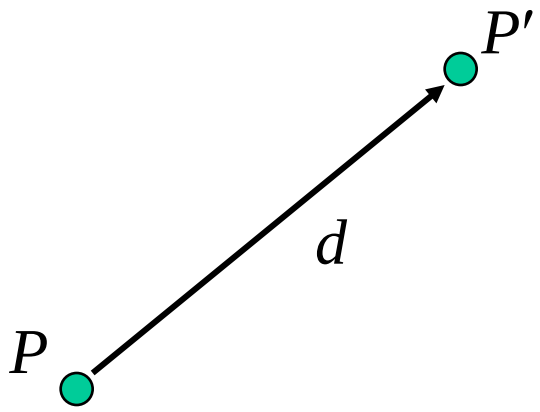
- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive homogeneous coordinate transformation matrices
- Learn to build arbitrary transformation matrices from simple transformations

Affine Transformations

- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear (we don’t use shear too often)
- Importance in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints

Translation

- Move (translate, displace) a point to a new location
- Displacement determined by a vector d
  - Three degrees of freedom
  - $P' = P + d$



Translation Using Representations

- Using the homogeneous coordinate representation in some frame

$$\mathbf{p} = [x \;\; y \;\; z \;\; 1]^T$$

$$\mathbf{p}' = [x' \; y' \; z' \; 1]^T$$

$$\mathbf{d} = [d_x \;\; d_y \;\; d_z \;\; 0]^T$$

We will use the prime ′ symbol to refer to the result of a transform. So p →p′.

Hence $\mathbf{p}' = \mathbf{p} + \mathbf{d}$ or

$$x' = x + d_x$$

$$y' = y + d_y$$

$$z' = z + d_z$$

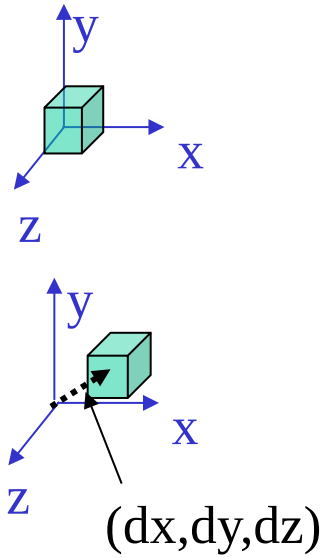
note that this expression is in four dimensions and expresses point = point + *vector*

Translation Matrix

- We can also express translation using a 4 x 4 matrix **T** in homogeneous coordinates

$$\mathbf{p}' = \mathbf{T}\mathbf{p}$$

where $\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

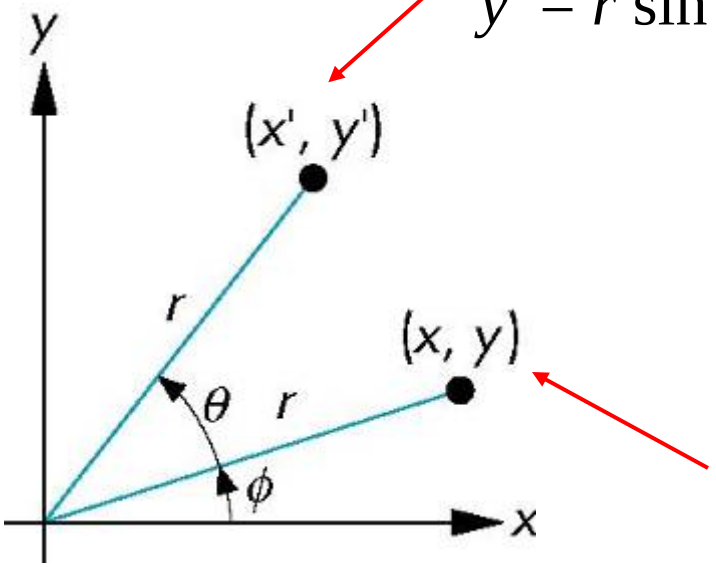


- This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

Rotation (2D)

- Consider rotation about the origin by θ degrees
  - Radius stays the same, angle increases by θ

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



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

$$x = r\cos\phi$$
$$y = r\sin\phi$$

We obtain the final format by some simple manipulation of trig identities – try it yourself, a nice exercise..

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

Rotation About the z-Axis

- Rotation about z-axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

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

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

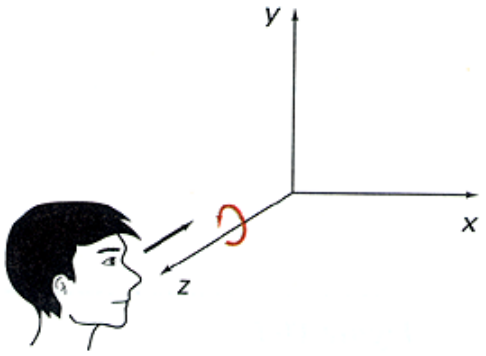
$$z' = z$$

  - Or in homogeneous coordinates

$$\mathbf{p}' = \mathbf{R}_z(\theta)\mathbf{p}$$

Rotation Matrix

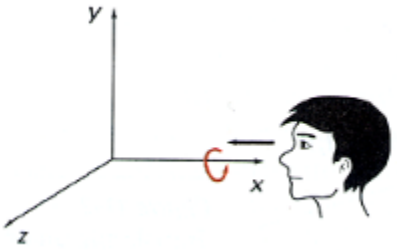
$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



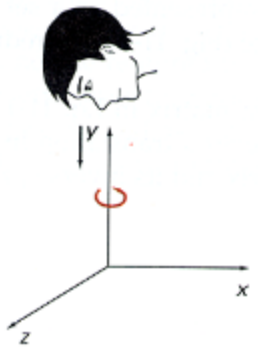
Rotation About x- and y-Axes

- Same argument as for rotation about z-axis
  - For rotation about x-axis, x is unchanged
  - For rotation about y-axis, y is unchanged

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



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



Scaling

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

$$x' = s_x\, x$$

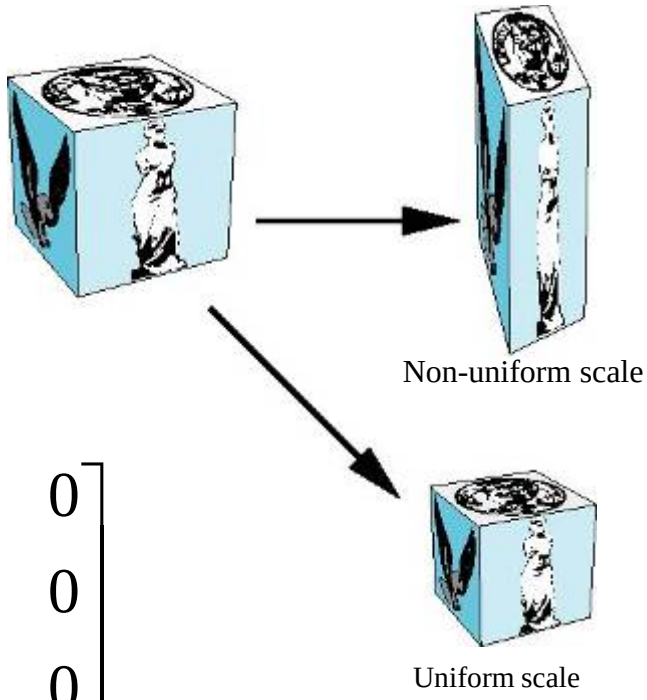
$$y' = s_y\, y$$

$$z' = s_z\, z$$

  - Or in homogeneous coordinates

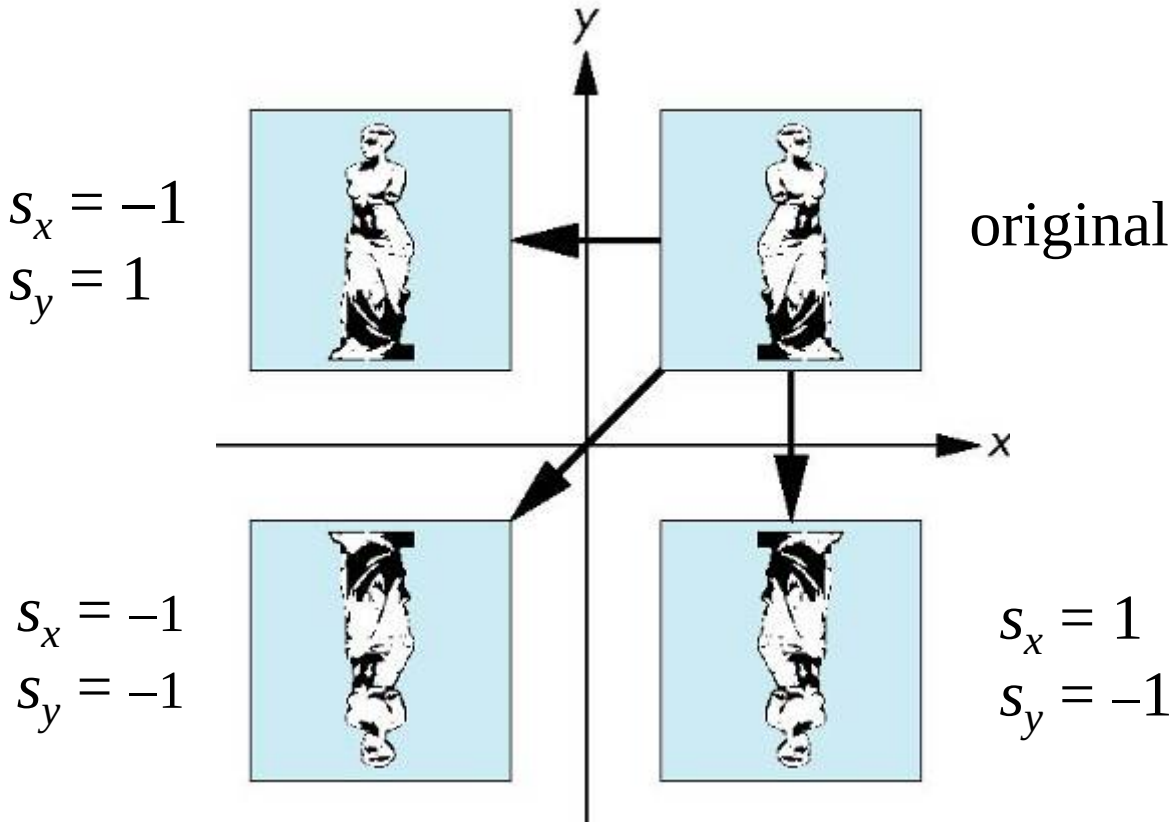
$$\mathbf{p}' = \mathbf{S}(s_x, s_y, s_z)\, \mathbf{p}$$

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



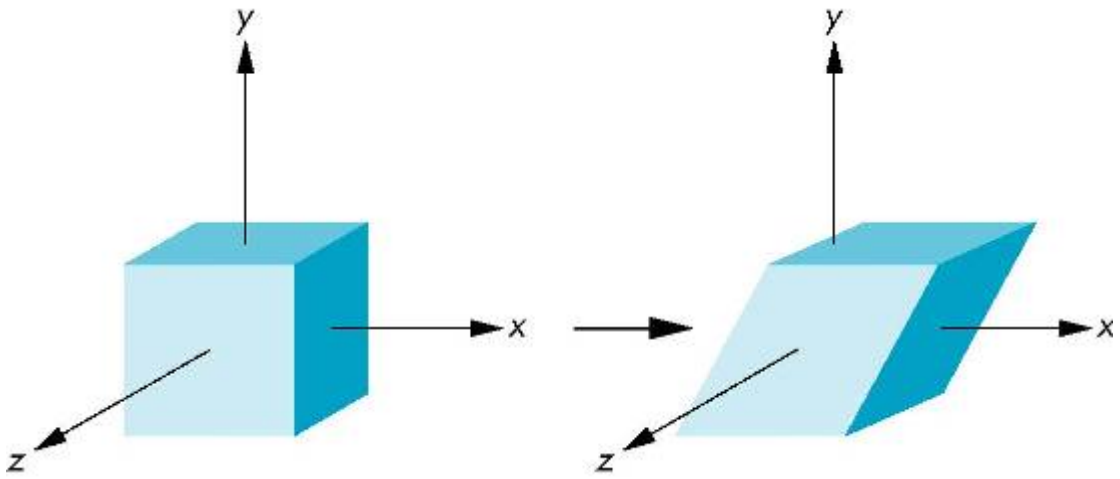
Reflection (using scaling)

- Corresponds to negative scale factors



Shear

- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



Shear is added here for completeness, but honestly speaking, we rarely use shears (I can't think of a single time I've used a shear).

Shear Matrix

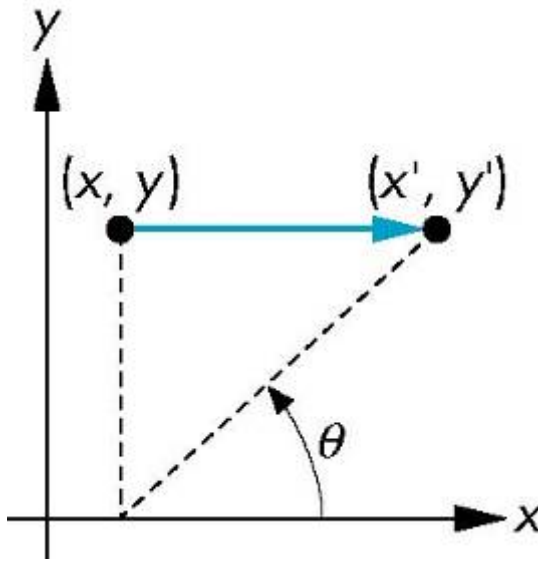
- Consider a simple shear along the x axis

$$x' = x + y \cot \theta$$

$$y' = y$$

$$z' = z$$

$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(-\theta)$ and $\sin(-\theta) = -\sin(\theta)$, $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{\mathrm{T}}(\theta)$
    - For any general rotation matrix $\mathbf{R}$, $\mathbf{R}^{-1} = \mathbf{R}^{\mathrm{T}}$
  - Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices
- Because the same transformation is applied to many vertices, the cost of forming a matrix $\mathbf{M} = \mathbf{ABCD}$ is not significant compared to the cost of computing $\mathbf{Mp}$ for hundreds or even thousands vertices $\mathbf{p}$
- The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

- Note that matrix on the **right** is the first applied

$$\mathbf{p}' = \mathbf{STRp}$$ ← Order of operation: First rotate by R, second translate by T, last scale by S

- Mathematically, the following are equivalent

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

- Note, a few references (not many) use row vectors to represent points. In this case:

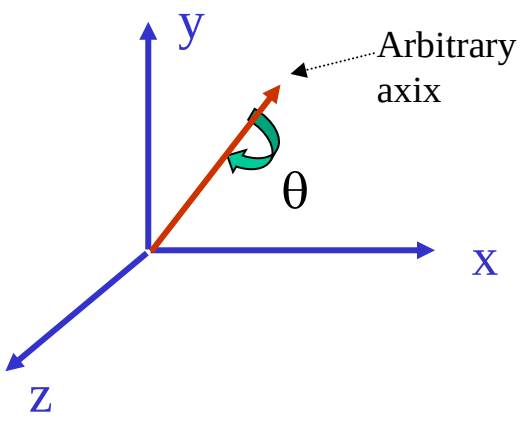
$$\mathbf{p}'^{\mathrm{T}} = \mathbf{p}^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$$

- Note that
  - For transpose operator: $(\mathbf{ABC})^{\mathrm{T}} = \mathbf{C}^{\mathrm{T}}\,\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$
  - For matrix inverse: $(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\,\mathbf{B}^{-1}\mathbf{A}^{-1}$

General Rotation About the Origin

- A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the x, y, and z axes
  - $\mathbf{R}(\theta, axis_{xyz}) = \mathbf{R}_z(\theta_z)\,\mathbf{R}_y(\theta_y)\,\mathbf{R}_x(\theta_x)$

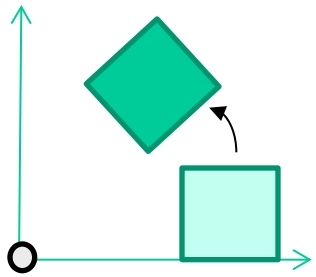
  where θ_x, θ_y, θ_z, are called the *Euler Angles*



- Note that rotations do not commute (that is, order matters!)

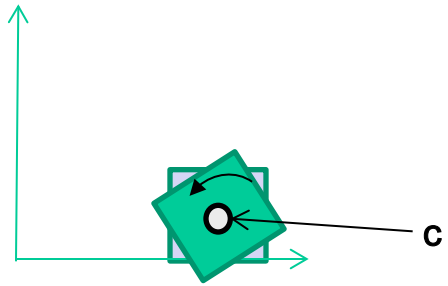
Thinking about transformations

- Before we apply a transformation, we need to consider what we want to achieve
- For example:



Rotating an object by 15° about the origin.

$p' = R(\theta)p$



Rotating an object by 15° about its own center **c**.

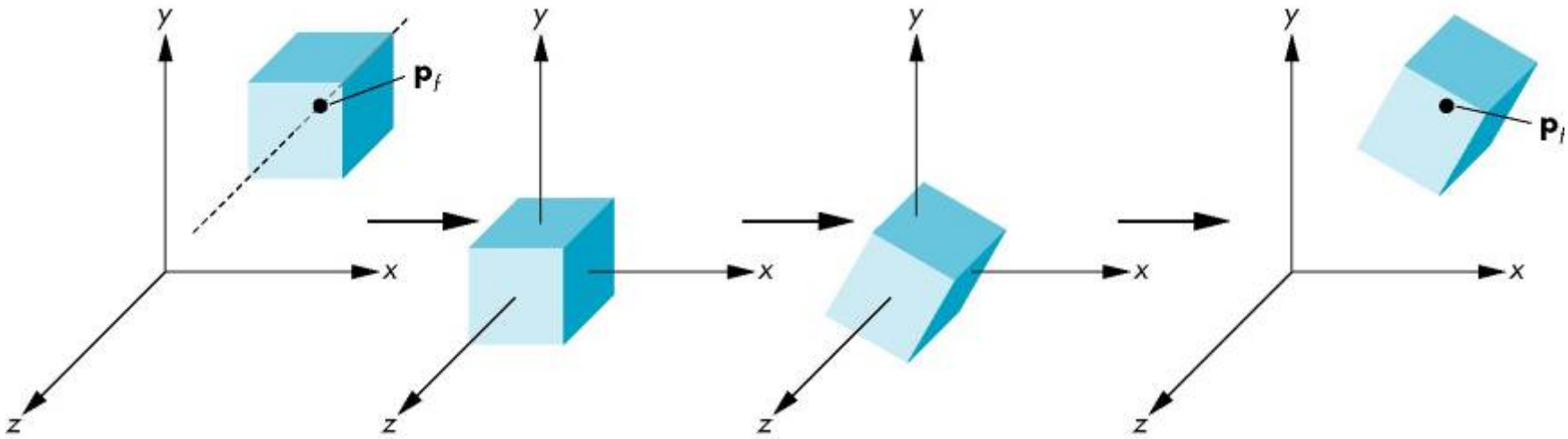
$p' = T(c)R(\theta)T(-c)p$

1st Translate object by center to origin
2nd Rotate
3rd Translate object back

Example in 3D

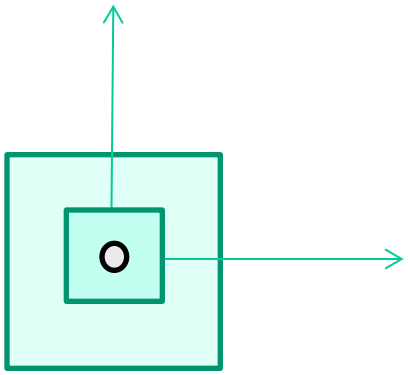
1. Move fixed point to origin
2. Rotate
3. Move fixed point back

$$\mathbf{M} = \mathbf{T}(\mathbf{p}_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-\mathbf{p}_f)$$



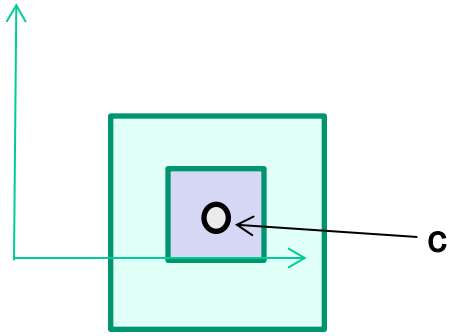
Scaling an object

- Say we want to scale an object that is not at the origin.
- For example:



Scaling an object
by 2 at the origin.

$p' = S(2)p$



Scaling an object by
2 about its own center **c**.

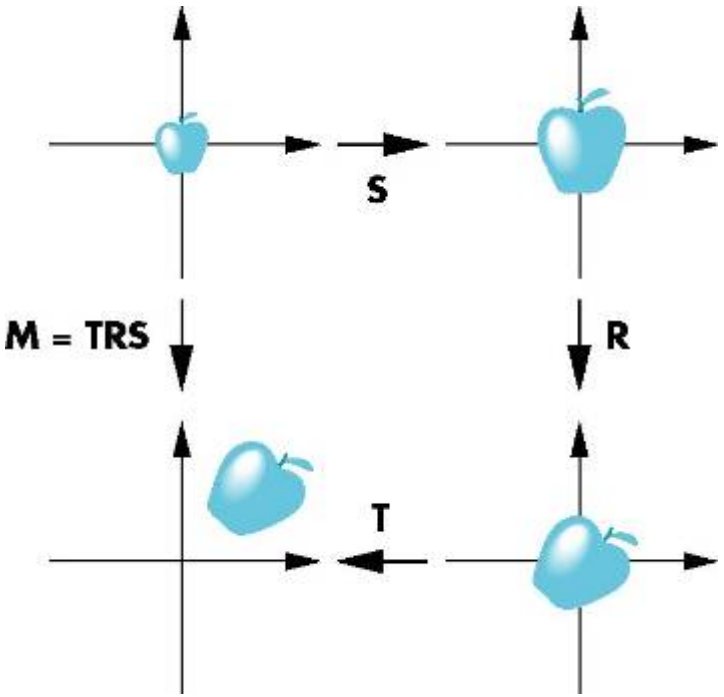
$p' = T(c)S(2)T(-c)p$

1st Translate object by center to origin
2nd Scale
3rd Translate object back

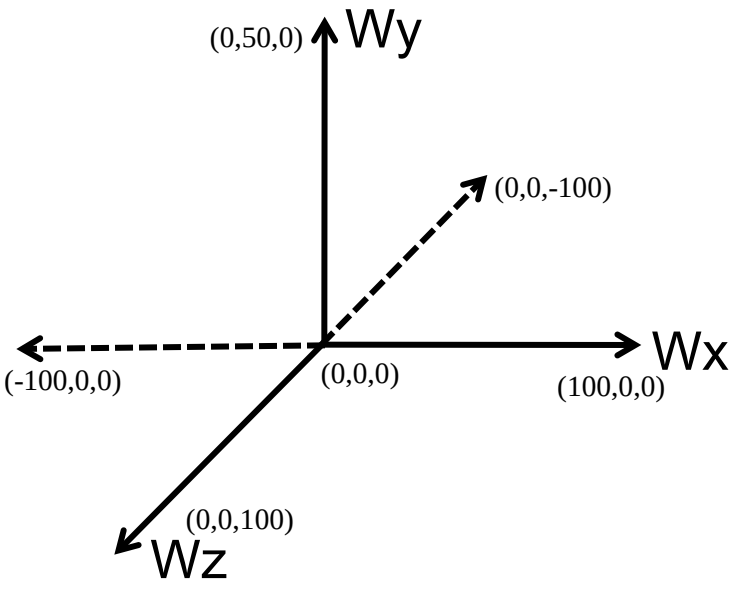
What would happen if you **didn't** translate by –c first?

Instancing

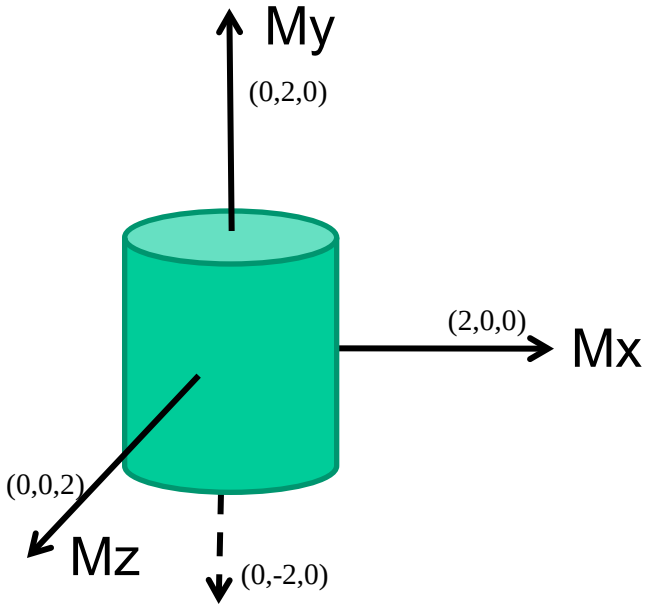
- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an *instance transformation* to its vertices to
  - scale
  - orientate
  - locate



Typical Situation with Instancing (1/3)



Establish a world

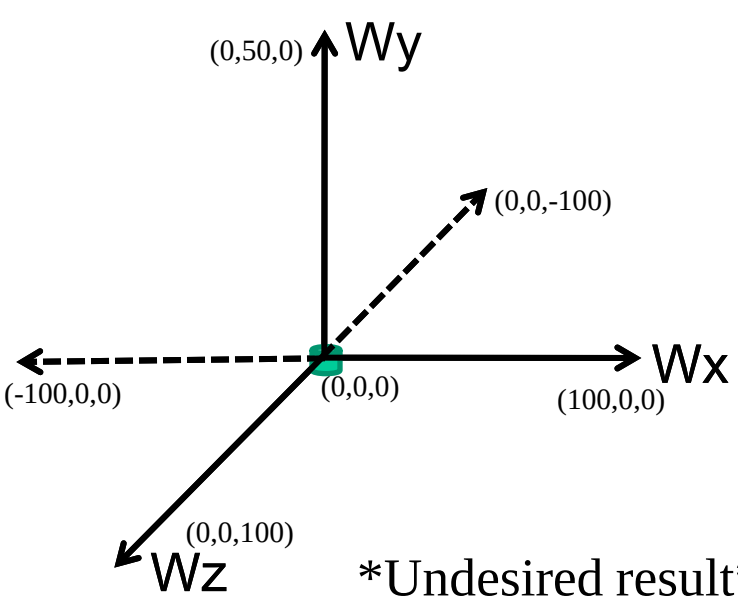


Create/load a model

World Coordinates defined by the programmer Lets say our world is Wx [-100,100], Wy = [0, 50], Wz = [-100,100]

We create an instance of a model, it's coordinates are centered about (0,0,0) and size Mx=[-1,1], My=[-1,1], Mz=[-1,-1].

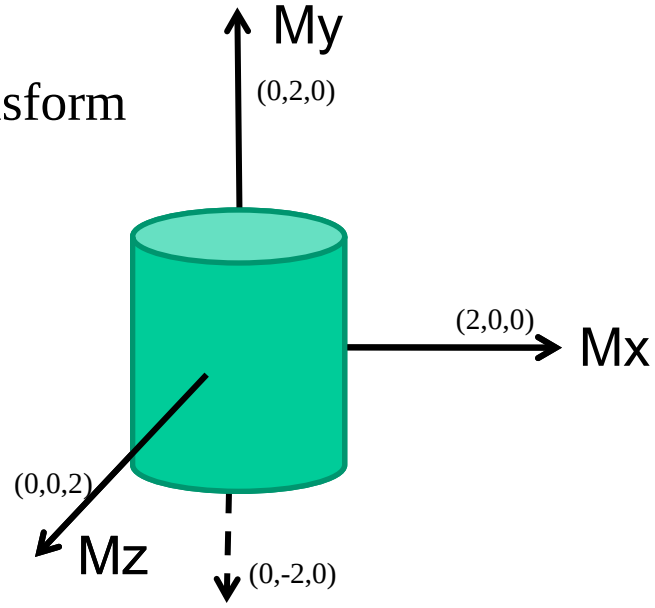
Typical Situation with Instancing (2/3)



Undesired result

Establish a world

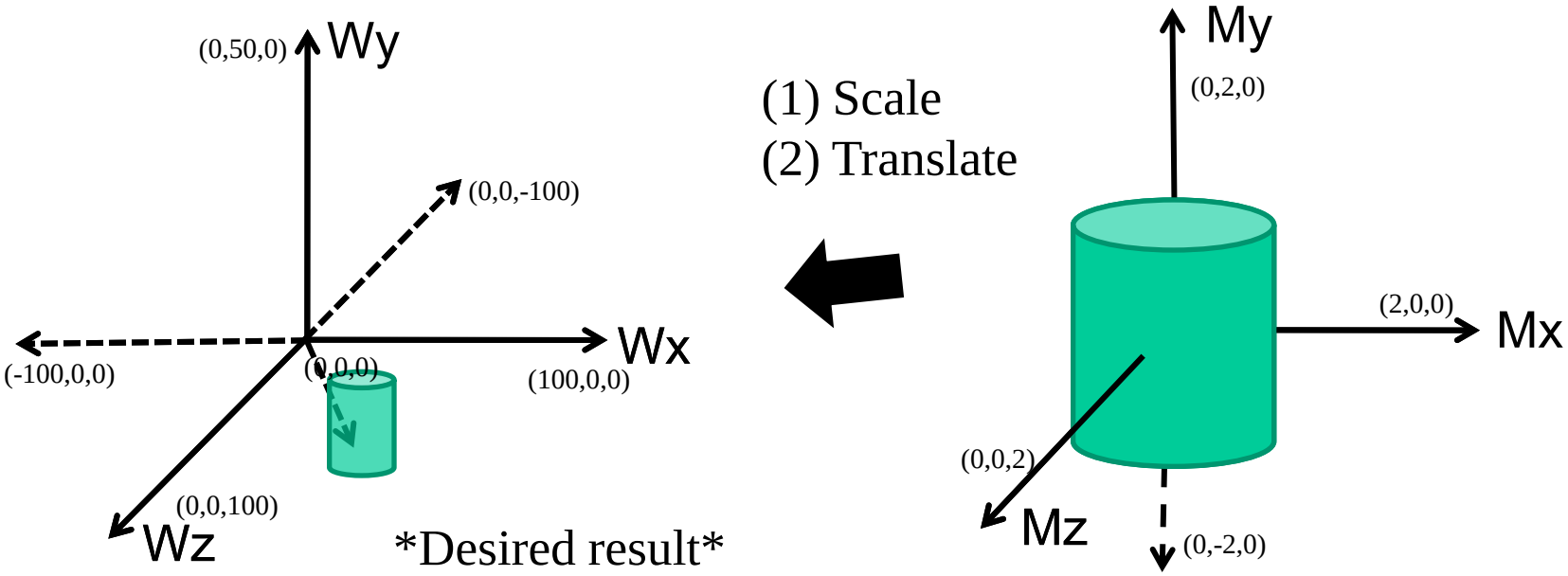
If no transform applied.



Create/load a model

If we do not transform our model, it will appear at very small at the center of our world. In our case, part of the model will be cut, since the model contains Y points < 0, but our world is only Y > 0. Thus have of the model is outside our world coordinate frame.

Typical Situation with Instancing (3/3)



Establish a world

Create/load a model

We need to scale and translate the instance model to appear in the right place in the 3D world coordinate frame. Note, generally we don't update the instance models vertices, so we will need to do this scale and translate every time we draw the model.

OpenGL Transformations

Objectives

- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling
- Introduce OpenGL matrix modes
  - Model-view
  - Projection

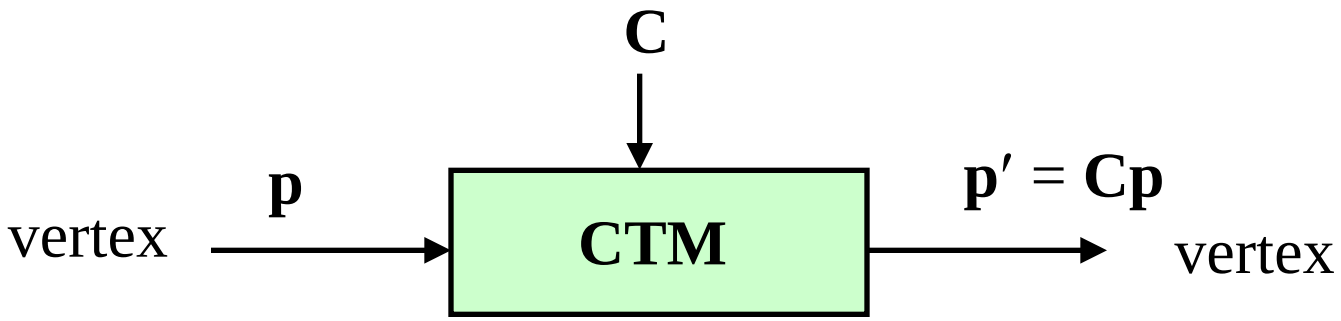
OpenGL Matrices

- In OpenGL, matrices are part of the state
- Multiple types
  - Model-View (**`GL_MODELVIEW`**)
  - Projection (**`GL_PROJECTION`**)
  - Texture (**`GL_TEXTURE`**) (ignore for now)
  - Color (**`GL_COLOR`**) (ignore for now)
- Single set of functions for manipulation
- Select which to manipulated by
  - **`glMatrixMode(GL_MODELVIEW);`**
  - **`glMatrixMode(GL_PROJECTION);`**
- Note that OpenGL matrix mode is also a state

glMatrixMode — specify which matrix is the current matrix

Current Transformation Matrix (CTM)

- Conceptually, in each matrix mode, there is a 4 x 4 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
- The CTM is defined in the user program and loaded into a transformation unit



CTM Operations

- The CTM can be altered either by loading a new CTM or by post-mutiplication
  - Load an identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$ **(very common)**
  - Load an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{M}$
  - Load a translation matrix: $\mathbf{C} \leftarrow \mathbf{T}$
  - Load a rotation matrix: $\mathbf{C} \leftarrow \mathbf{R}$
  - Load a scaling matrix: $\mathbf{C} \leftarrow \mathbf{S}$
  - Postmultiply by an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{M}$
  - Postmultiply by a translation matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{T}$
  - Postmultiply by a rotation matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{R}$
  - Postmultiply by a scaling matrix: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{S}$

Example: Rotation About a Fixed Point

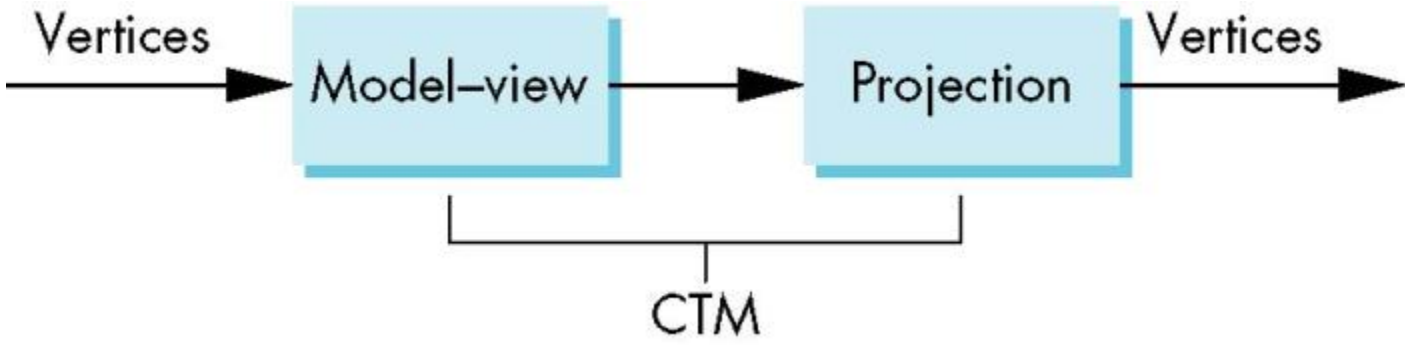
1. Start with identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$
2. Move fixed point back: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$
3. Rotate: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
4. Move fixed point to origin: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

- Result: $\mathbf{C} = \mathbf{T}^{-1}\mathbf{R}\mathbf{T}$
  - Which is **backwards**
  - A consequence of doing post-multiplications

- Each operation corresponds to one function call in the program
- The last operation specified is the first executed in the program

CTM in OpenGL*

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM
- Can manipulate each by first setting the correct matrix mode



*Book calls this **CTM**, I'm just going to call it **M** – this is more standard notation, even used by OpenGL.
Also, we often think of there being two matrices: **M** – for the model-view, and **P** for the projection.
CTM = P M. In reality, there is only one matrix – **CTM**.

OpenGL Rotation, Translation, Scaling

- Load an identity matrix (overwrite M)

```
glLoadIdentity();
```

- Specify a **rotation** (post-multiply to M)

```
glRotatef(theta, vx, vy, yz);
```

  - **`theta`** is the angle of rotation in degrees
  - **`(vx, vy, vz)`** is the axis of rotation

- Specify a **translation** (post-multiply to M)

```
glTranslatef(dx, dy, dz);
```

- Specify a **scale** (post-multiply to M)

```
glScalef(sx, sy, sz);
```

- Each has a **`float`** (**`f`**) and **`double`** (**`d`**) format (**`glScaled`**)

Example

- Rotation about z axis by 30 degrees at a fixed point of (1.0, 2.0, 3.0)

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();                       // M = I
glTranslatef( 1.0, 2.0, 3.0 );          // M = T-1
glRotatef( 30.0, 0.0, 0.0, 1.0 );       // M = TR
glTranslatef( -1.0, -2.0, -3.0 );       // M = T-1RT
...
glBegin( GL_POLYGON );                  // Apply
  glVertex3d( x0, y0, z0 );
  ...
glEnd();
```

- Remember that last matrix specified in the program is the first applied to vertices

Arbitrary Matrices

- Can load and post-multiply by any 4x4 matrix defined in the application program
  - **`glLoadMatrixf(m);`** or **`glLoadMatrixd(m);`**
    - Load **`m`** as CTM
  - **`glMultMatrixf(m);`** or **`glMultMatrixd(m);`**
    - Post-multiply **`m`** to CTM
- The matrix **`m`** is a one-dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns

```
double m[16];
```

$$\begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix}$$

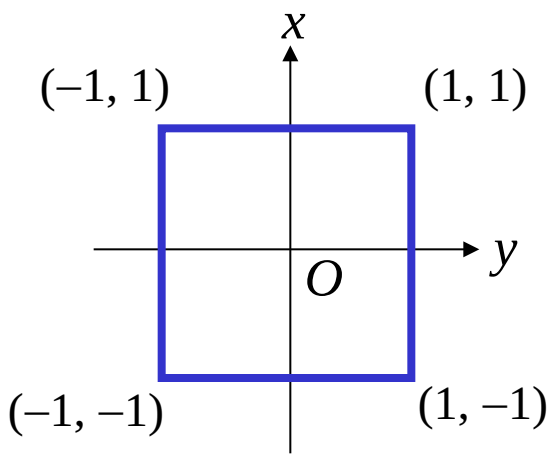
Matrix Stacks

- In many situations we want to save away the current transformation matrices for use later
  - Traversing hierarchical data structures
  - Avoiding state changes when executing display lists
  - Avoid state changes when drawing 2D primitives (e.g. text) on top of our 3D model
- OpenGL maintains stack for each matrix mode (as set by **`glMatrixMode`**)
- Operations on matrix stack
  - **`glPushMatrix();`** **`// Note current CMT does not change`**
    `// set where to start the current transformation`
  - **`glPopMatrix();`** **`// Current CMT is overridden`**
    `// end the current object transformation`

Example: Saving and Restoring Matrix

- **`DrawSquare()`** draws a 2-by-2 square centered at the origin

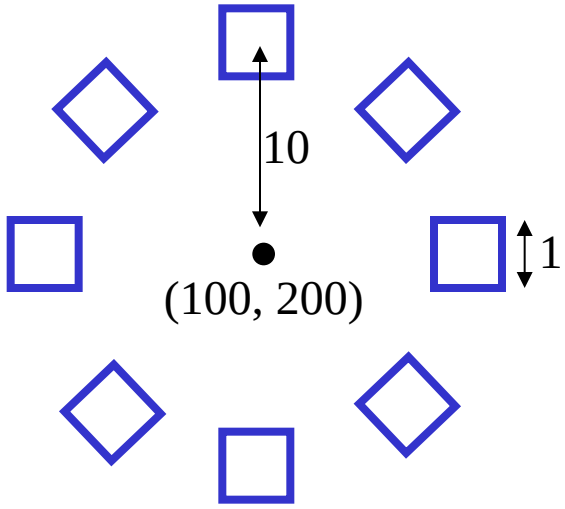
```
void DrawSquare() {
  glBegin( GL_POLYGON );
    glVertex3d( 1.0, 1.0, 0.0 );
    glVertex3d( -1.0, 1.0, 0.0 );
    glVertex3d( -1.0, -1.0, 0.0 );
    glVertex3d( 1.0, -1.0, 0.0 );
  glEnd();
}
```



Example: Saving and Restoring Matrix

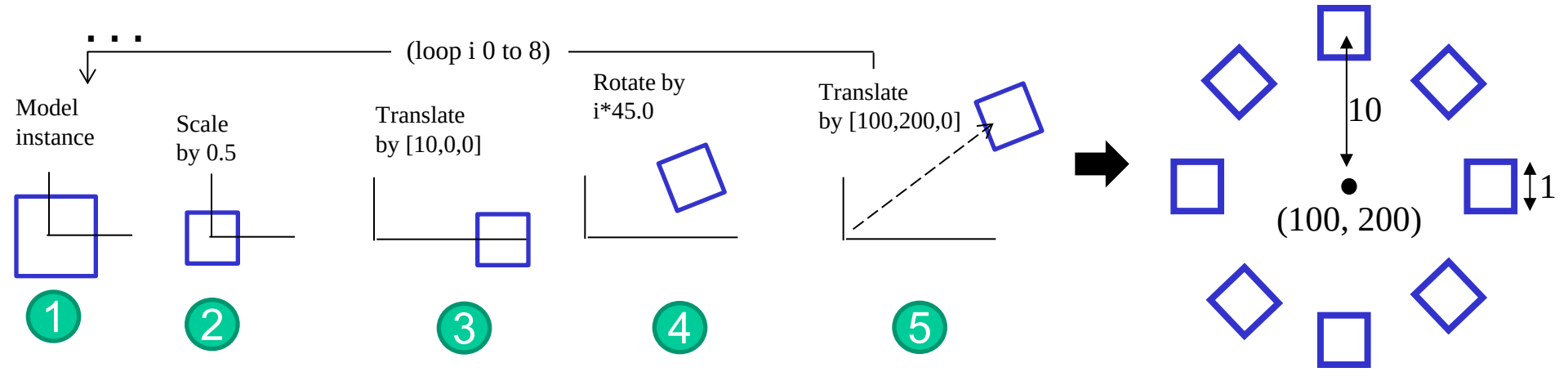
- Suppose we want to use **`DrawSquare()`** to draw a complex pattern, centered at (100, 200) with 8 oriented unit squares

```
...
glTranslated(100.0, 200.0, 0.0);
for ( int i = 0; i < 8; i++ ) {
  glPushMatrix();
     glRotated(i*45.0, 0.0, 0.0, 1.0);
     glTranslated(10.0, 0.0, 0.0);
     glScaled(0.5, 0.5, 0.5);
     DrawSquare();
  glPopMatrix();
}
...
```



Dissecting this example

```
5 glTranslated(100.0, 200.0, 0.0);      // M = T
  for ( int i = 0; i < 8; i++ ) {
    glPushMatrix();                     // M = T (save on stack)
4     glRotated(i*45.0, 0.0, 0.0, 1.0); // M = T R
3     glTranslated(10.0, 0.0, 0.0);     // M = T R T(10,0,0)
2     glScaled(0.5, 0.5, 0.5);          // M = T R T(10,0,0) S
1     DrawSquare();                     // Draw (see from 75)
    glPopMatrix();                      // M = T (from stack)
  }
  ...
```



Reading Back Current Matrices

- Can also access matrices (and other parts of the state) by *query* functions
  - **glGetIntegerv**
  - **glGetFloatv**
  - **glGetBooleanv**
  - **glGetDoublev**
  - **glIsEnabled**
- For matrices, we use as

```
double m[16];
glGetDoublev( GL_MODELVIEW, m );
```

Sample Code: Using Transformations

- Example: use idle function to rotate a cube and mouse function to change direction of rotation
- Start with a program that draws a cube (**`cube.c`**) in a standard way
  - Centered at origin
  - Sides aligned with axes
  - Will discuss modeling in next lecture

main.c

```
void main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
                      GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutCreateWindow("colorcube");
  glutReshapeFunc(myReshape);
  glutDisplayFunc(display);
  glutIdleFunc(spinCube);
  glutMouseFunc(mouse);
  glEnable(GL_DEPTH_TEST);
  glutMainLoop();
}
```

Idle and Mouse Callbacks

```
void spinCube()
{
  theta[axis] += 2.0;
   if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
   glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
   if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
           axis = 0;
   if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
           axis = 1;
   if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
           axis = 2;
}
```

Display Callback

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();
   glRotatef(theta[0], 1.0, 0.0, 0.0);
   glRotatef(theta[1], 0.0, 1.0, 0.0);
   glRotatef(theta[2], 0.0, 0.0, 1.0);
   colorcube();
   glutSwapBuffers();
}
```

- Note that because of the fixed form of callbacks, variables such as **`theta`** and **`axis`** must be defined as globals
- Camera information is in standard reshape callback

Summary

- Vectors + Points can be manipulated using a 4x4 matrix
  - This is because of the power of the homogenous coordinate formulation
  - This makes hardware implementation incredible fast
    - Optimize 4x4 transforms!
  - The key is knowing how to create the proper 4x4 matrices
    - We have seen: Translation, Scale, Rotation
- Concatenation
  - Matrix can be concatenated to perform multiple operations at once
  - Need to keep track of the order of our operation
- OpenGL
  - Need to understand how commands affect the matrix
  - Some support to manage matrices (glPushMatrix()/glPopMatrix)
  - Two different matrix modes (Model View and Projection (viewing))

End of Lecture 4
next -- projections