CS3241
Computer Graphics

Semester 1, 2019/2020

Lecture 4

Geometric Objects & Transformations

School *of* Computing
National University *of* Singapore

Geometry

Outline

- Introduce the elements of geometry
  - Scalars
  - Vectors
  - Points
- Define basic primitives
  - Line segments
  - Polygons

Basic Elements

- **Geometry** is the study of the **spatial relationships** among objects in an n-dimensional space
  - In computer graphics, we are interested in objects in **3D**

- Want a minimum set of **primitives** from which we can build more sophisticated objects

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

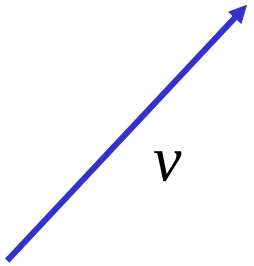
Scalars

- Need three basic elements in geometry
  - **Scalars**, **Vectors**, **Points**

- **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:** The **real** and **complex** number systems under the ordinary rules with which we are familiar
  - Scalars alone have no geometric properties

Vectors

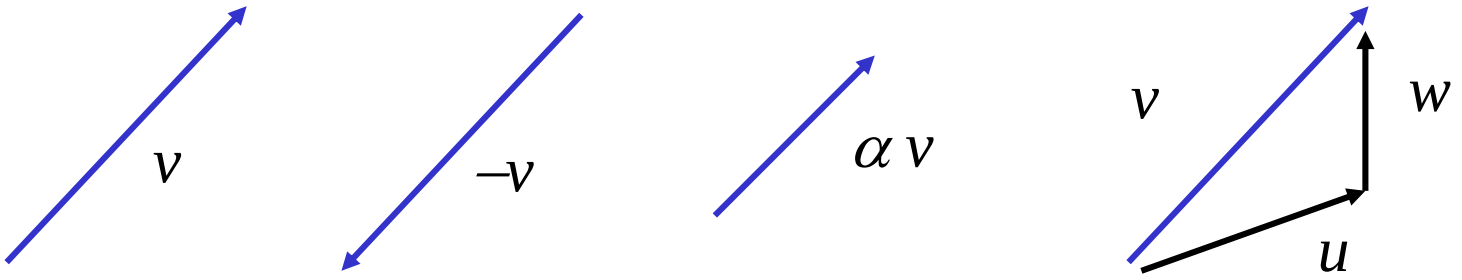
- Physical definition: a **vector** is a quantity with two attributes
  - **Direction**
  - **Magnitude**



- Examples
  - Force
  - Velocity
  - Directed line segments

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 direction
- 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: $w = u + v$

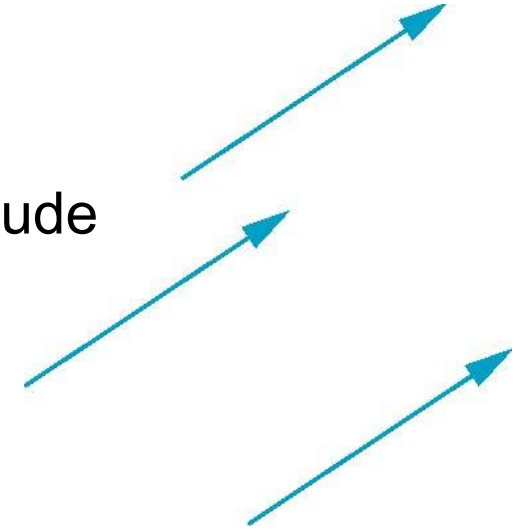
- Expressions such as

  $v = u + 2w - 3r$

  make sense in a **vector space**

Vectors Lack Position

- These vectors are identical
  - Same direction and magnitude

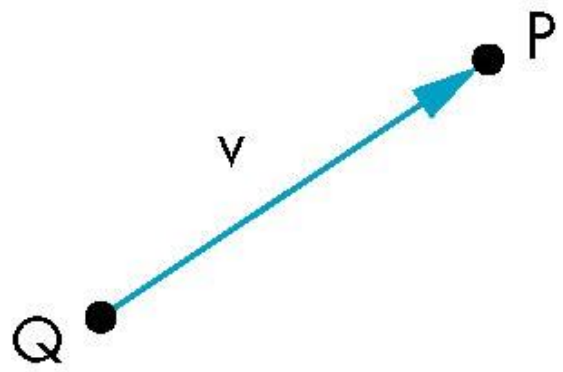


- Vectors spaces insufficient for geometry
  - Need points

Points

- **Location** in space

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



$$v = P - Q$$

$$P = Q + v$$

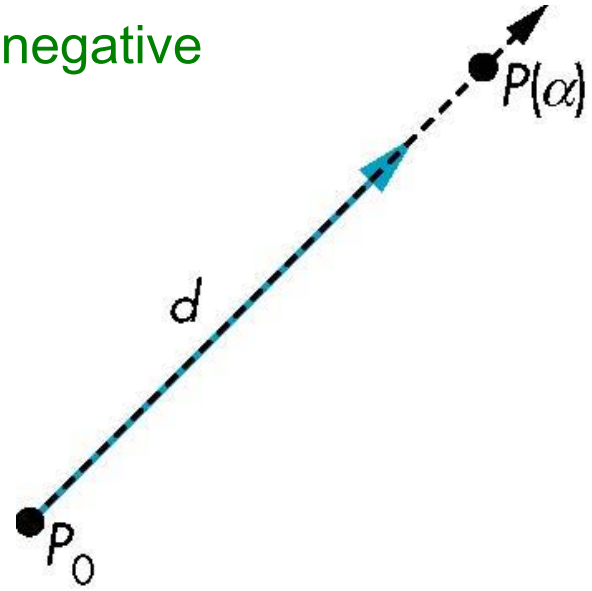
Self-Study

Affine Spaces

- Point + a vector space
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
- For any point P, define
  - $1 \bullet P = P$
  - $0 \bullet P = \mathbf{0}$ (zero vector)

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 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

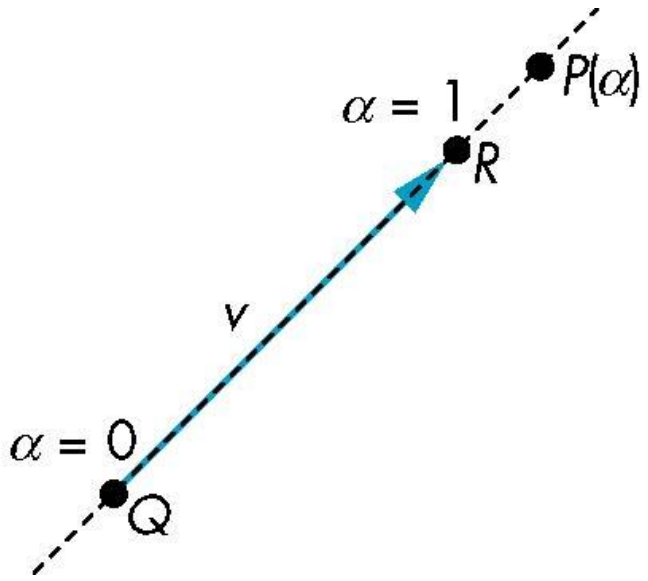
- If $\alpha \geq 0$, then $P(\alpha)$ is the **ray** leaving P_0 in the direction d

$$P(\alpha) = P_0 + \alpha\, d$$

- If we use two points to define v, then

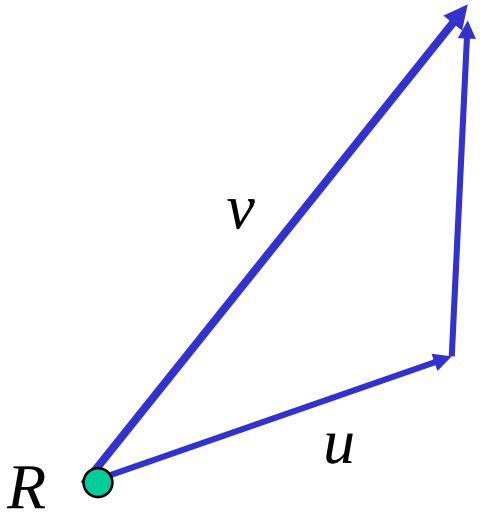
$$\begin{aligned} P(\alpha) &= Q + \alpha\,(R - Q) \\ &= Q + \alpha\, v \\ &= \alpha R + (1 - \alpha)Q \end{aligned}$$

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

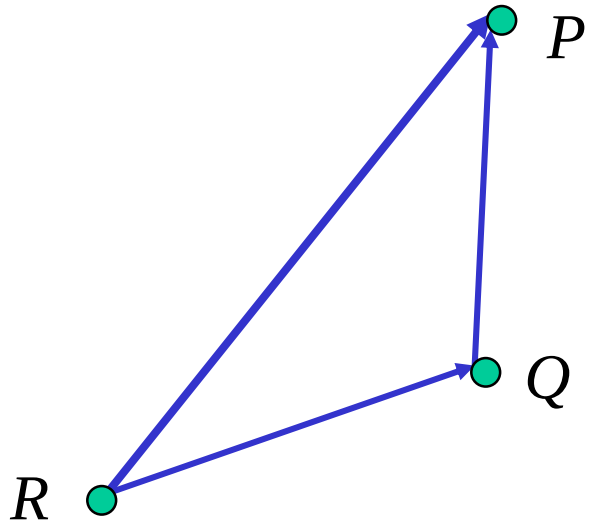


Planes

- A **plane** can be defined by a point and two vectors or by three points



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



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

Normals

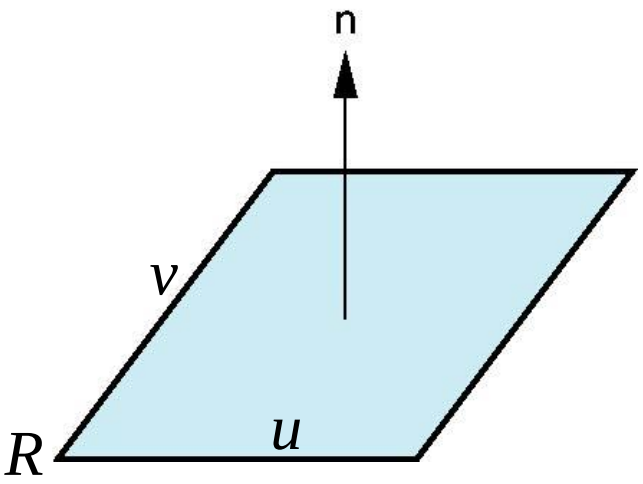
- The one-point-two-vector form,

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

- 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 is any point on the plane



  - How is it related to the form $ax + by + cz + d = 0$?

Representation

Outline

- Introduce
  - **coordinate systems** for representing **vector spaces**
  - **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

- 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

Coordinate Systems

- Consider a **basis** $v_1, v_2, \ldots, v_n$
  - A basis is a set of **linearly independent vectors**
- A **vector** is written $v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$
- The list of scalars $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is the **representation** of v **with respect to the given basis**
- We can write the representation as a row or column array of scalars

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

matrix transpose

Example

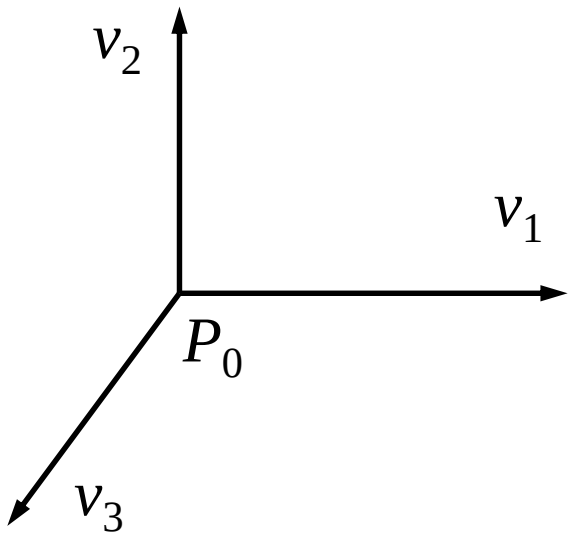
- $v = 2v_1 + 3v_2 - 4v_3$
- $\mathbf{a} = [2 \;\; 3 \; -4]^T$

- Note that this representation is with respect to a particular basis

- For example, in OpenGL we start by representing vectors using the **object basis** but later the system needs a representation in terms of the **camera** or **eye basis**

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
  $v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$
- Every **point** can be written as
  $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$



Confusing Points and Vectors

- Consider the point P and the vector v
  - $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + ... + \beta_n v_n$
  - $v = \alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$

- They appear to have similar representations
  - $\mathbf{p} = [\beta_1 \ \ \beta_2 \ \ \beta_3]^T \qquad \mathbf{v} = [\alpha_1 \ \ \alpha_2 \ \ \alpha_3]^T$

  which confuses the point with the vector

- A vector has no position

A Single Representation

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

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

Homogeneous Coordinates

- The **homogeneous coordinates** for a three dimensional point $[x \;\; y \;\; z]^T$ is given as
  - $\mathbf{p} = [x' \; y' \; z' \; w]^T = [wx \;\; wy \;\; wz \;\; w]^T$
- 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 $[x \;\; y \;\; z \;\; 1]^T$

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 pipeline** 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 a **perspective division**

Self-Study

Change of Coordinate Systems

- Consider **two representations** of the **same vector** v 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

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

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

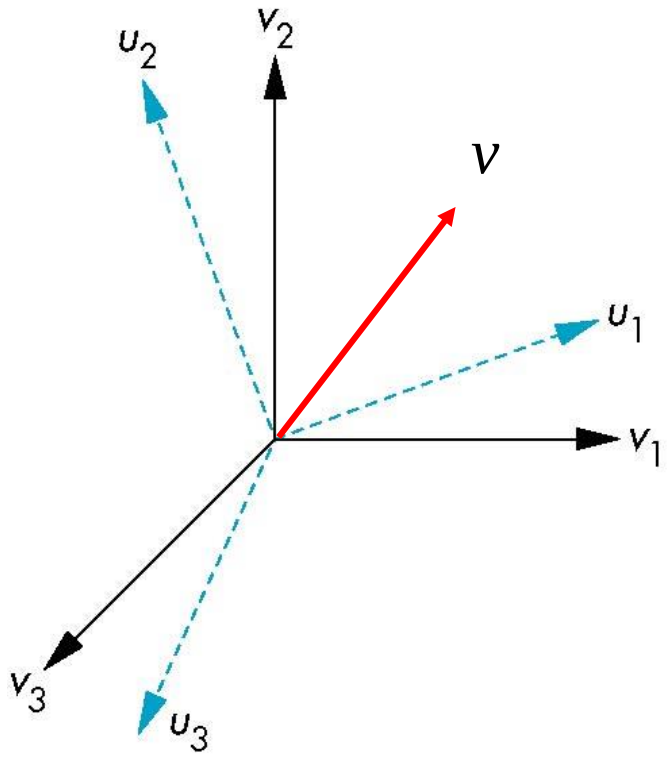
Representing Second Basis in Terms of First

- Each of the basis vectors, u_1, u_2, u_3, are vectors that can be represented in terms of the first basis

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

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

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



Matrix Form

- The coefficients define a 3 x 3 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the bases can be related by

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

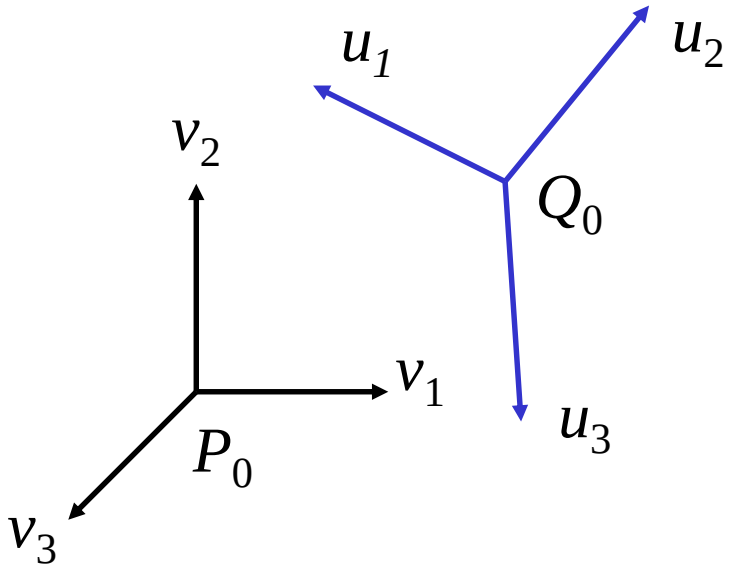
where

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

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

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

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

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

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

$$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + \gamma_{44}P_0$$

- The coefficients define a 4 x 4 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

Self-Study

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

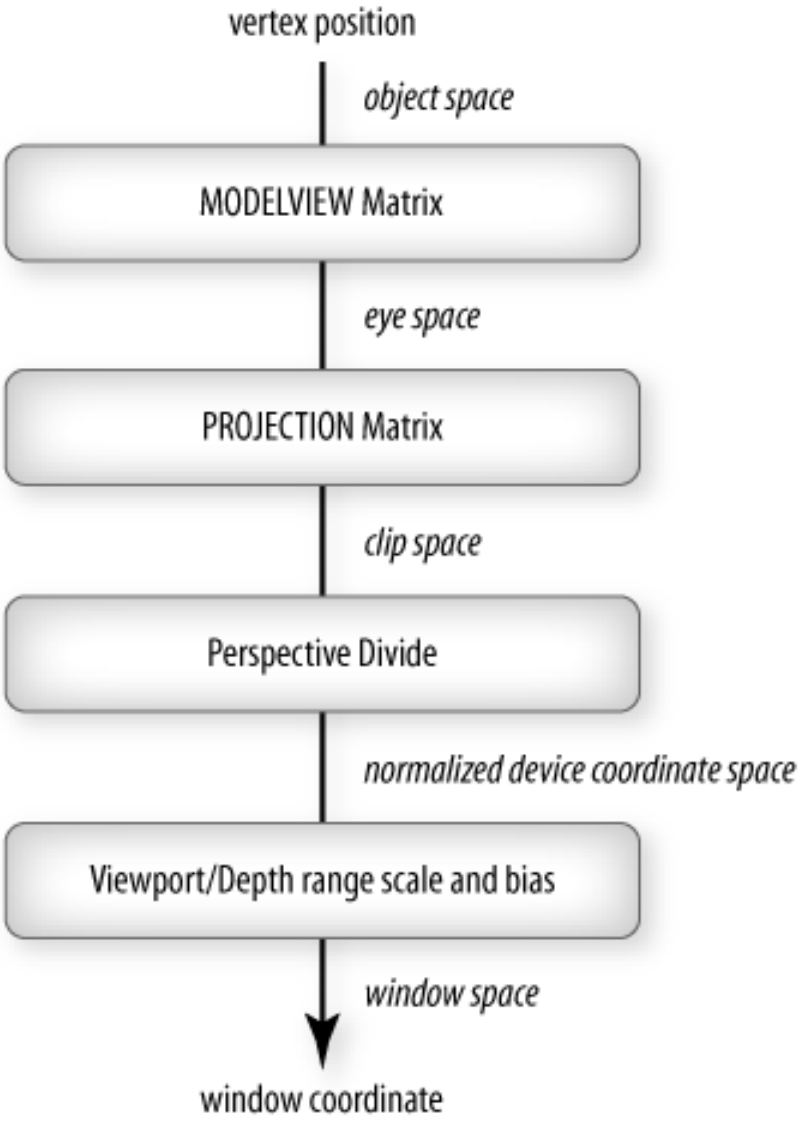
  $\mathbf{a} = \mathbf{M}^T \mathbf{b}$

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

Object, 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 vertices are specified in the **object frame**
- The vertices are then represented in the **world frame**
- Eventually we represent entities in the **camera frame** by changing the object representation using the **model-view matrix**
- Initially these frames are the same (i.e. $\mathbf{M} = \mathbf{I}$)

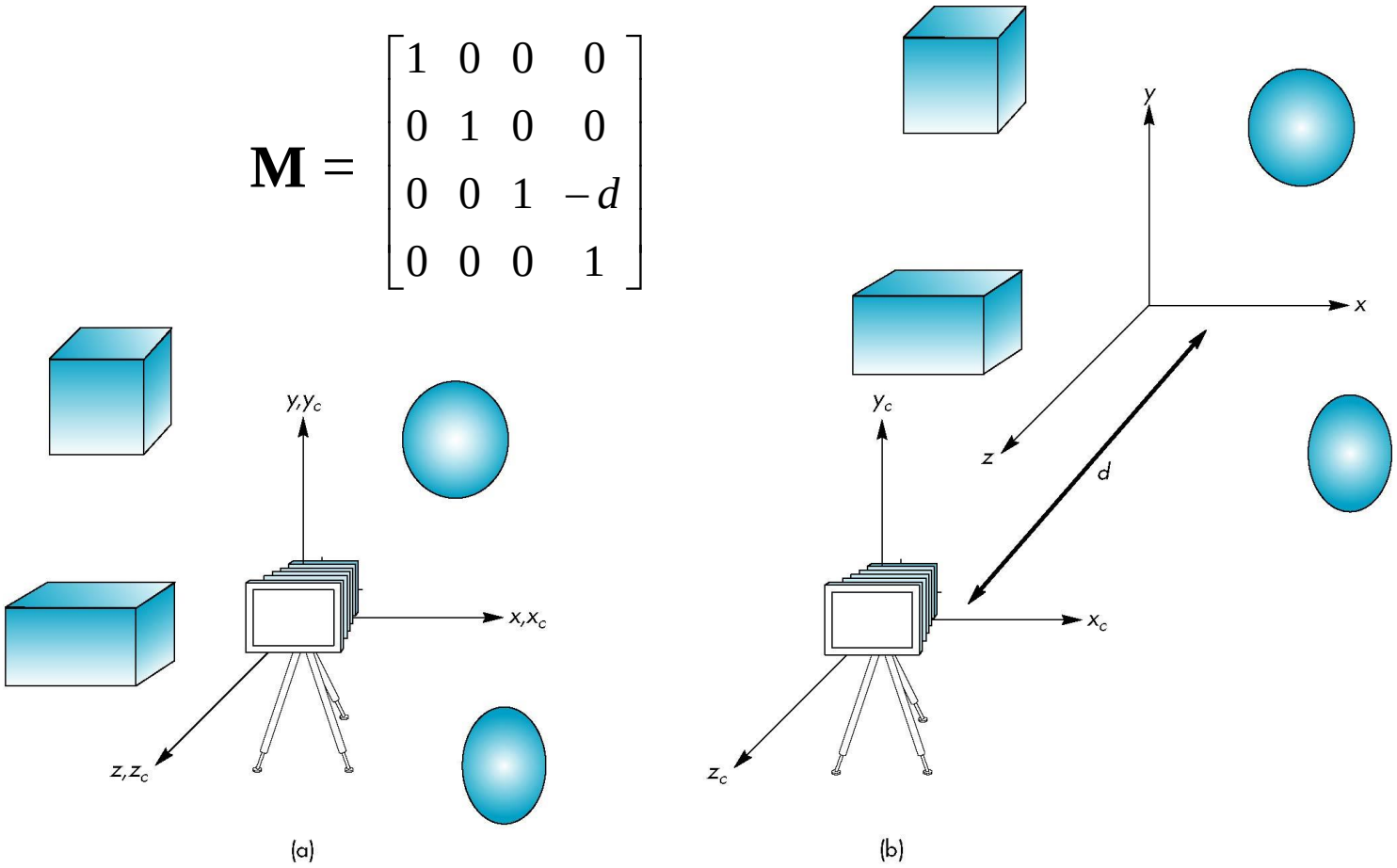
OpenGL Geometric Transformations



Moving the Camera

- If objects are on both sides of $z = 0$, and if we want the camera to see all objects, we must move camera frame

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Transformations

Outline

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

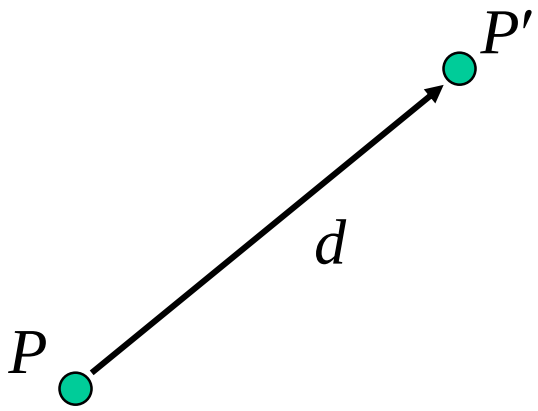
Affine Transformations

- Line preserving
- Characteristic of many physically important transformations
  - **Rigid body transformations**: rotation, translation
  - Scaling, shear
- 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

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

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

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

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

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

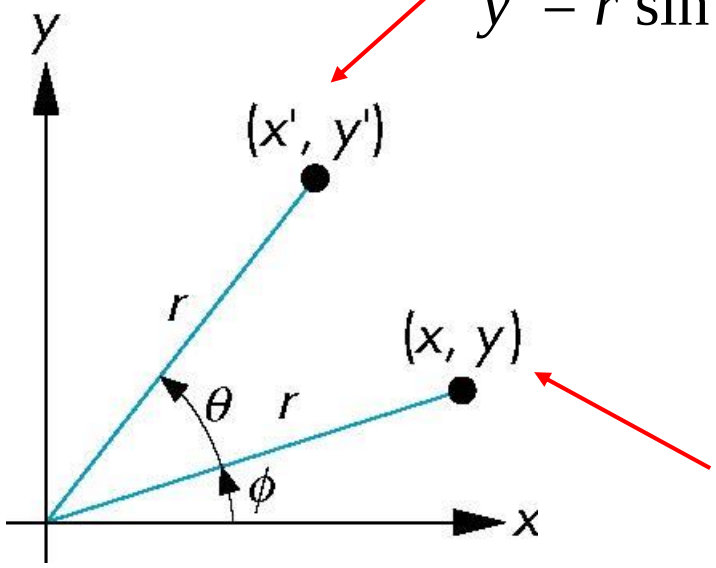
where $\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- 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

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



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

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

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)

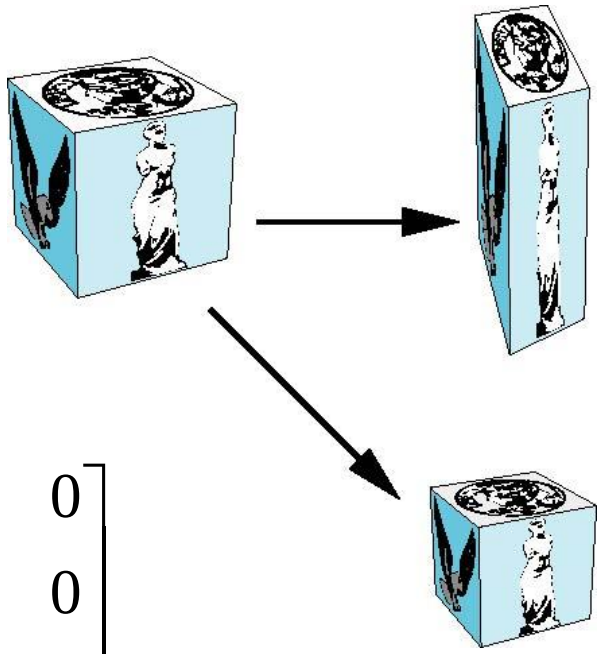
$$x' = s_x\, x$$

$$y' = s_y\, y$$

$$z' = s_z\, z$$

  - Or in homogeneous coordinates

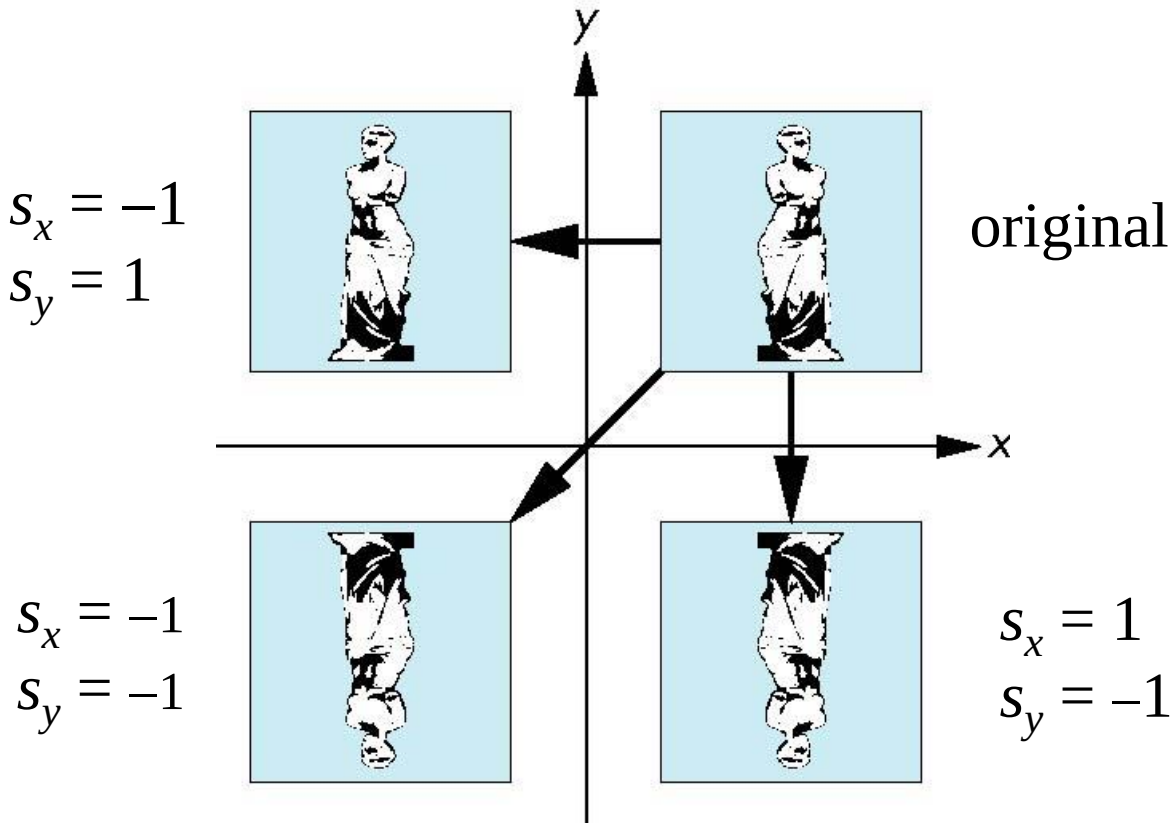
$$\mathbf{p}' = \mathbf{S}(s_x, s_y, s_z)\, \mathbf{p}$$



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

Reflection

- Corresponds to negative scale factors



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 many 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
- Mathematically, the following are equivalent

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

- Note many references 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
  - $(\mathbf{ABC})^{\mathrm{T}} = \mathbf{C}^{\mathrm{T}}\,\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$
  - $(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\,\mathbf{B}^{-1}\mathbf{A}^{-1}$

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
  - $\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\,\mathbf{R}_y(\theta_y)\,\mathbf{R}_x(\theta_x)$

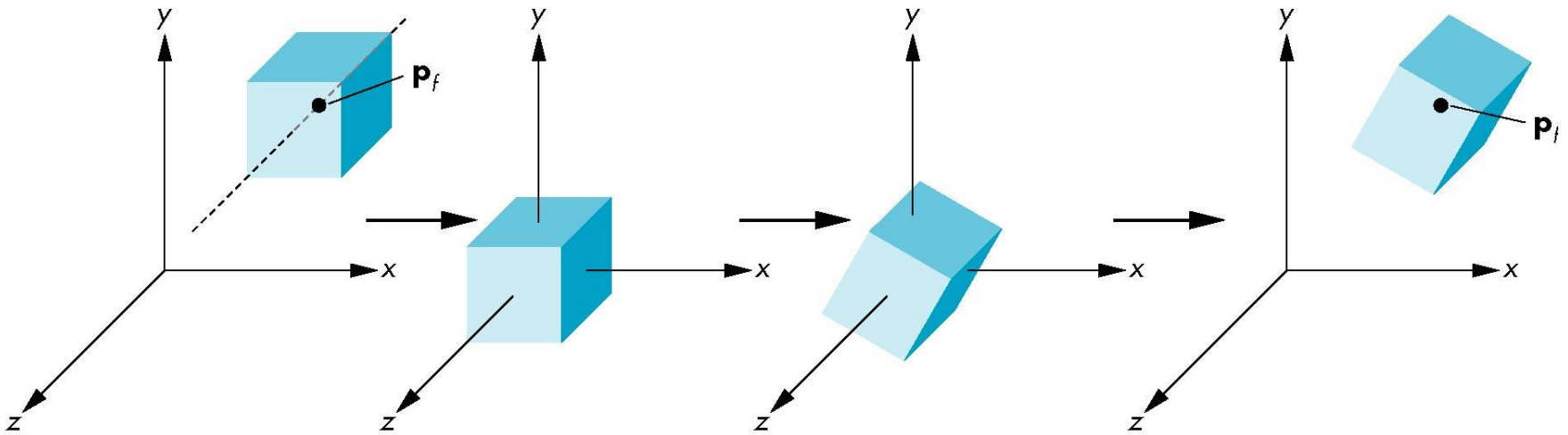
  where θ_x, θ_y, θ_z, are caller the **Euler Angles**

- Note that rotations **do not commute**

Rotation About a Fixed Point Other than the Origin

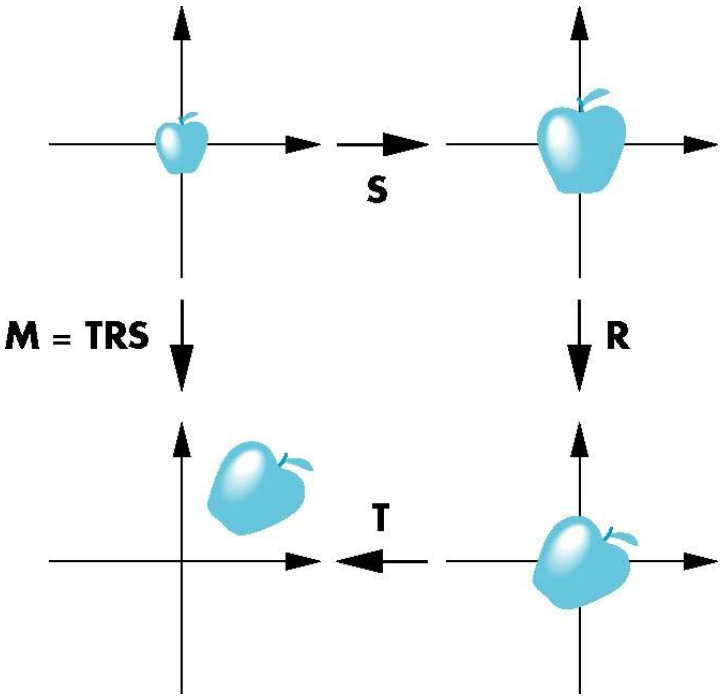
1. Move fixed point to origin
2. Rotate
3. Move fixed point back

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



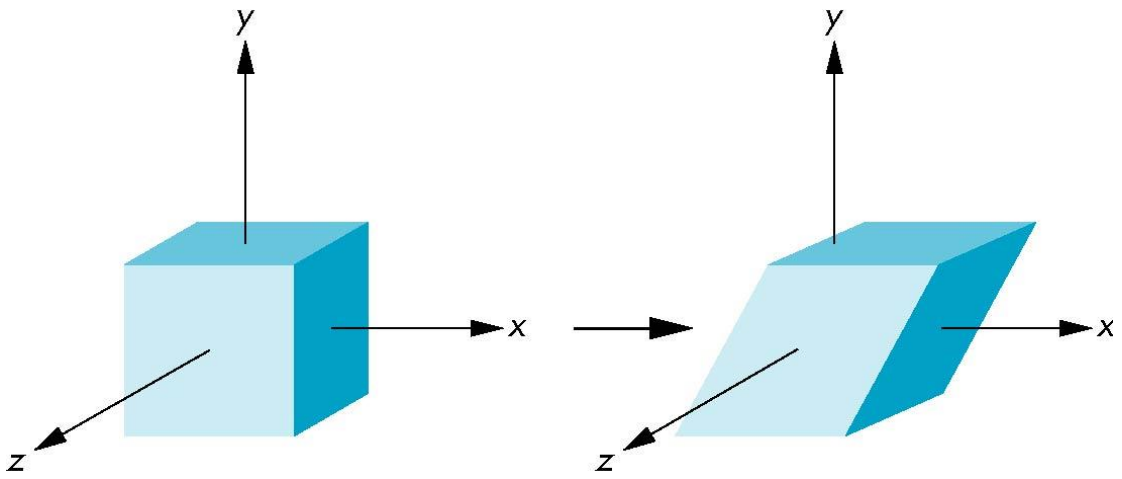
Instancing

- 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



Shear

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



Shear Matrix

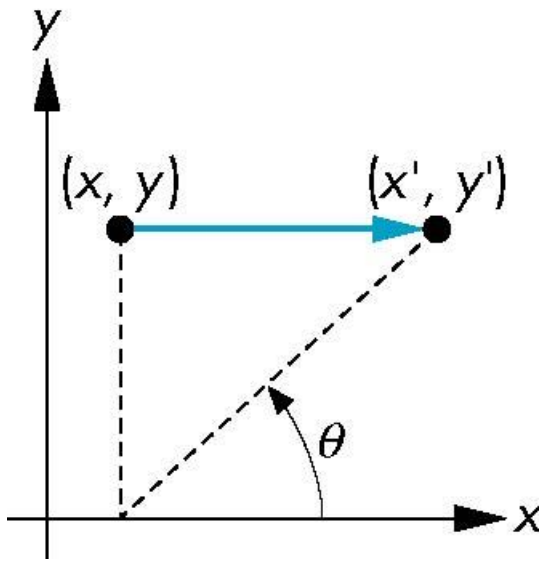
- Consider a simple shear along the x axis

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

$$y' = y$$

$$z' = z$$

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



OpenGL Transformations

Outline

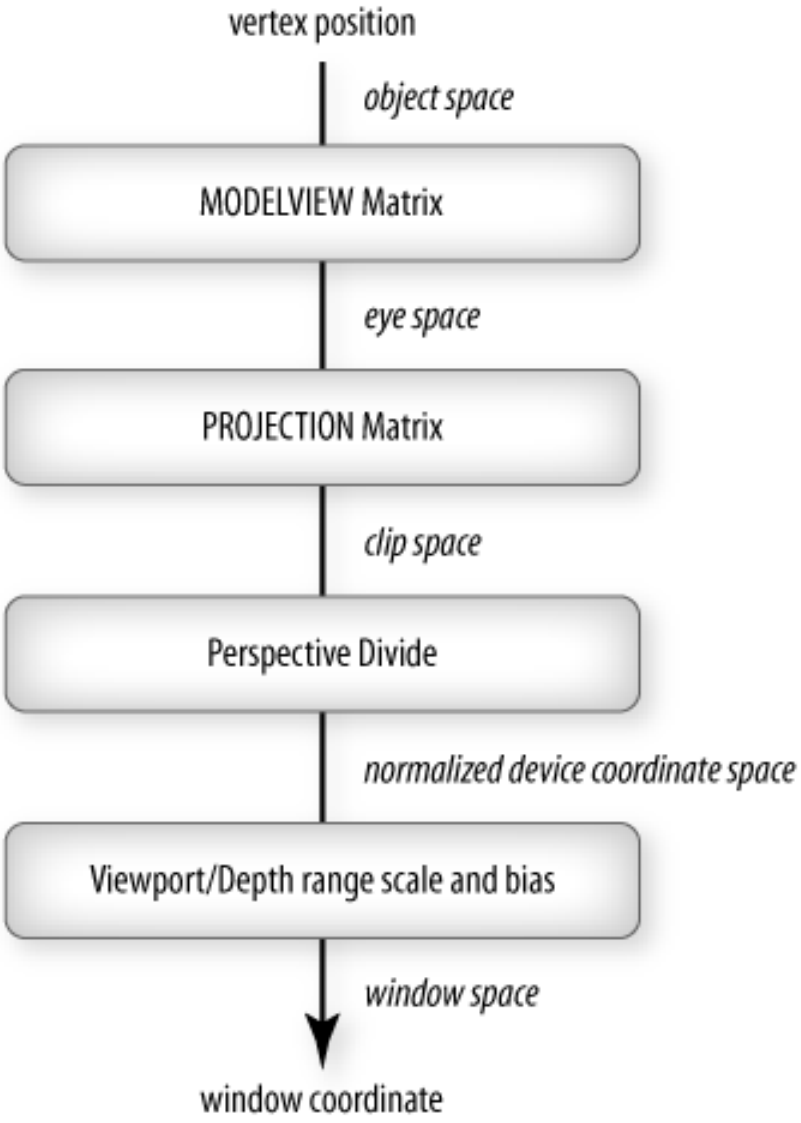
- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling

- 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 this)
  - Color (`GL_COLOR`) (ignore this)
- Single set of functions for manipulation
- Select which to be manipulated by
  - `glMatrixMode(GL_MODELVIEW);`
  - `glMatrixMode(GL_PROJECTION);`
- Note that OpenGL matrix mode is also a state

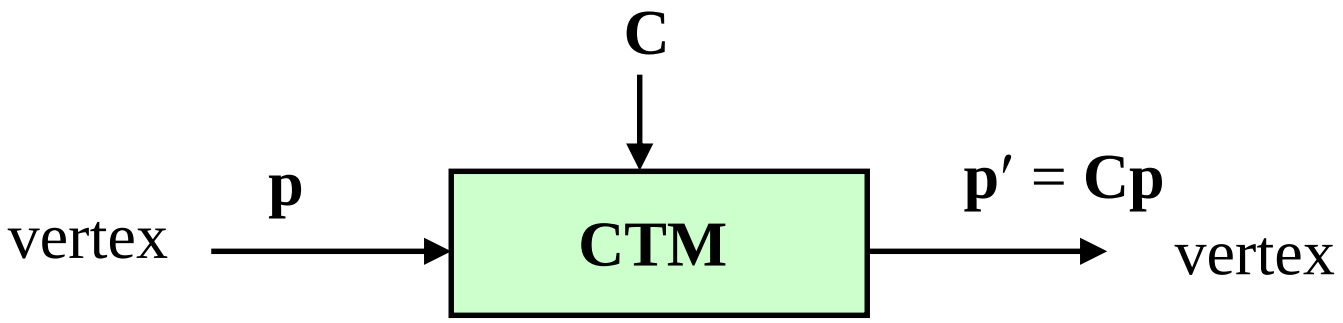
OpenGL Geometric Transformations



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: $\mathbf{C} \leftarrow \mathbf{I}$
  - 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: $\mathbf{C} \leftarrow \mathbf{CM}$
  - Postmultiply by a translation matrix: $\mathbf{C} \leftarrow \mathbf{CT}$
  - 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: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

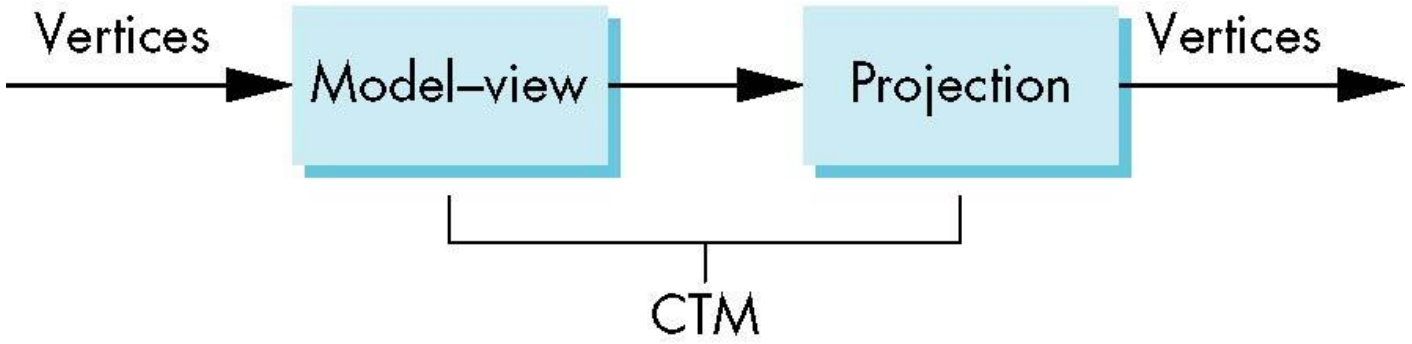
- Result: $\mathbf{C} = \mathbf{T}^{-1}\mathbf{R}\mathbf{T}$
  - Which is **backwards**
  - A consequence of doing post-multiplications

- Each operation corresponds to one function call in the program
- Note that 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



OpenGL Rotation, Translation, Scaling

- Load an **identity matrix** (overwrite CTM)

```
glLoadIdentity();
```

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

```
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 CTM)

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

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

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

- Each has (`f`)`loat` and (`d`)`ouble` format (e.g. `glScaled`)

Example

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

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glTranslatef( 1.0, 2.0, 3.0 );
glRotatef( 30.0, 0.0, 0.0, 1.0 );
glTranslatef( -1.0, -2.0, -3.0 );
...
glBegin( GL_POLYGON );
  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 matrx 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 transformation matrices for use later
  - Traversing **hierarchical** data structures
  - Avoiding state changes when executing display lists

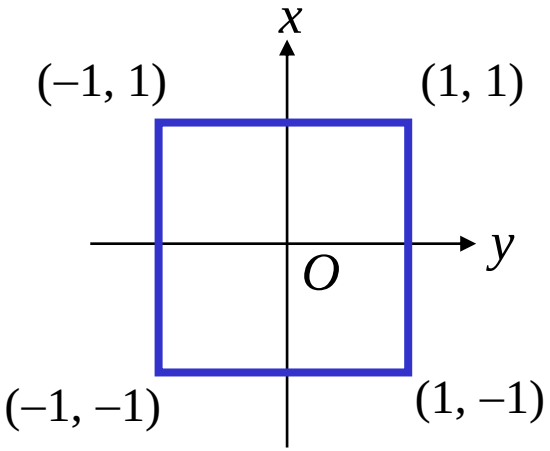
- OpenGL maintains **stack** for **each matrix mode** (as set by **`glMatrixMode`**)

- Operations on matrix stack
  - **`glPushMatrix();`**
  - **`glPopMatrix();`**

Example: Saving and Restoring Matrix

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

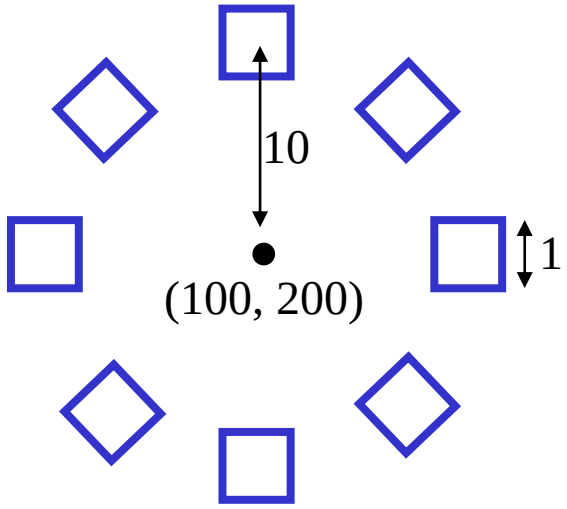
```
void DrawSquare() {
  glBegin( GL_POLYGON );
    glVertex3d( 1.0, 1.0, 0.0 );
    glVertex3d( -1.0, 1.0, 0.0 );
    glVertex3d( -1.0, -1.0, 0.0 );
    glVertex3d( 1.0, -1.0, 0.0 );
  glEnd();
}
```



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();
}
...
```



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

```
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.cpp`**) in a standard way
  - Centered at origin
  - Sides aligned with axes
  - Will discuss modeling in next lecture

main()

```
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

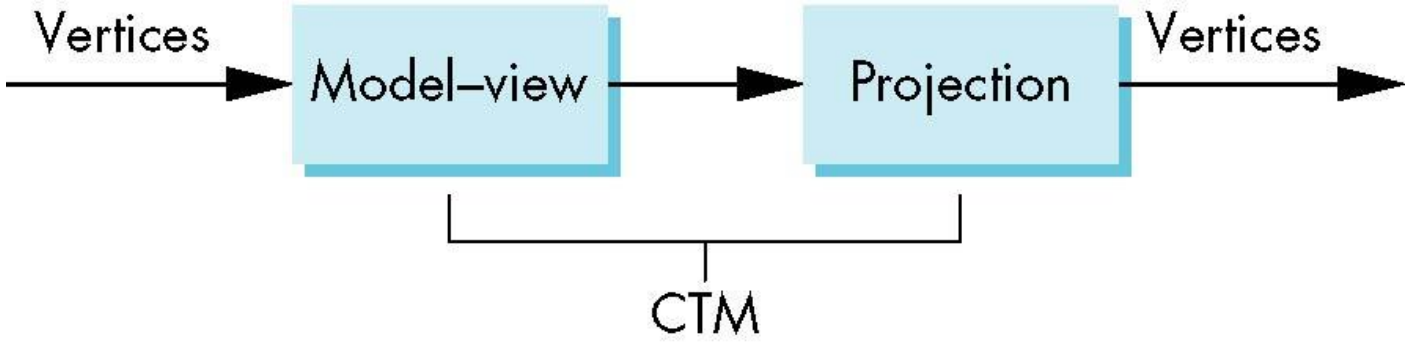
Using the Model-View Matrix

- In OpenGL the **model-view matrix** is used to
  - Position the **camera**
    - Can be done by rotations and translations but is often easier to use `gluLookAt()`
  - Build models of objects

- The **projection matrix** is used to define the **view volume**

Model-View and Projection Matrices

- Although both are manipulated by the same functions, we have to be careful because incremental changes are always made by post-multiplication
  - For example, rotating model-view and projection matrices by the same matrix are not equivalent operations. **Post-multiplication** of the **model-view matrix** is equivalent to **pre-multiplication** of the **projection matrix**



End of Lecture 4