SOS - Computer Graphics
Cushing Lecture05 - Spring 2014

1. Tool Tips – *WebGL?*
2. Geometry, Geometric objects, & Transformations
   Vectors, Matrices

<BREAK>

3. Tomorrow's Lab… Back to shaders….
4. Problem sets 5 and 6 …. For tomorrow (and/or next Wednesday)
5. Recap Last Week's Lab & Assignment
   - Ray Tracing Code Review – Isaac or Dani?
   - Comments about stretching the 3D Sierpinski?
6. The rest of the quarter….

Acknowledgements: Ed Angel, Jenny Orr, Ron Metoyer, Mike Bailey

Geometry

- Introduce the elements of geometry
  - Scalars
  - Vectors
  - Points
- Develop mathematical operations among them in a coordinate-free manner
- Define basic primitives
  - Line segments
  - Polygons

Basic Elements

- Geometry is the study of the relationships among objects in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in three dimensions
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements
  - Scalars
  - Vectors
  - Points

Coordinate-Free Geometry

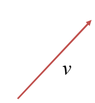
- When we learned simple geometry, most of us started with a Cartesian approach
  - Points were at locations in space **p**=(x,y,z)
  - We derived results by algebraic manipulations involving these coordinates
- This approach was nonphysical
  - Physically, points exist regardless of the location of an arbitrary coordinate system
  - Most geometric results are independent of the coordinate system
  - Example Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical

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

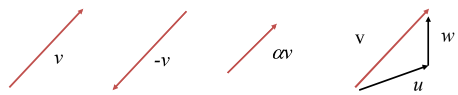
Vectors

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



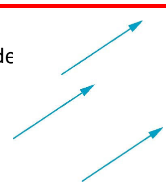
Linear Vector Spaces

- 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

Vectors Lack Position

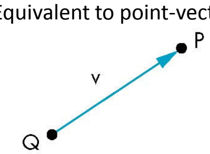
- These vectors are identical
  - Same length and magnitude



- 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



$v = P - Q$

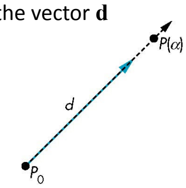
$P = v + Q$

Affine Spaces

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

Lines

- Consider all points of the form
  - $P(\alpha)=P_0 + \alpha\, \mathbf{d}$
  - Set of all points that pass through P_0 in the direction of the vector $\mathbf{d}$



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

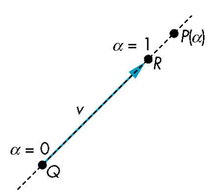
Rays and Line Segments

If α >= 0, then $P(\alpha)$ is the *ray* leaving P_0 in the direction $\mathbf{d}$

If we use two points to define v, then

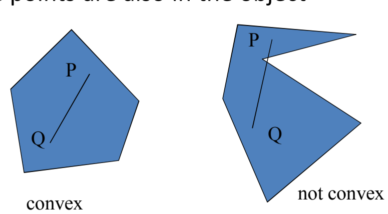
$P(\alpha) = Q + \alpha v$
$= Q + \alpha (R-Q)$
$= \alpha R + (1-\alpha)Q$

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



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



Affine Sums

Consider the "sum" $P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$

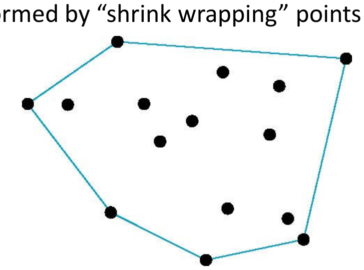
Can show by induction that this sum makes sense iff $\alpha_1 + \alpha_2 + \ldots \alpha_n = 1$

in which case we have the *affine sum* of the points $P_1, P_2, \ldots P_n$

If, in addition, $\alpha_i >= 0$, we have the *convex hull* of $P_1, P_2, \ldots P_n$

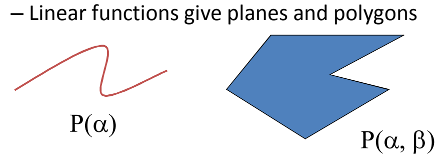
Convex Hull

- Smallest convex object containing $P_1, P_2, \ldots P_n$
- Formed by "shrink wrapping" points



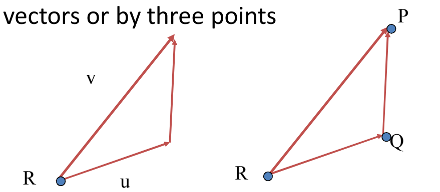
Curves and Surfaces

- Curves are one parameter entities of the form $P(\alpha)$ where the function is nonlinear
- Surfaces are formed from two-parameter functions $P(\alpha, \beta)$
  - Linear functions give planes and polygons



Planes

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



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

Representation

- New concepts : dimension and basis
- Introduce coordinate systems for representing vectors, spaces, and frames for representing affine spaces
- Discuss change of frames and bases
- Introduce homogeneous coordinates

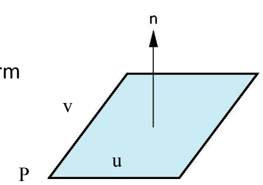
Normals

- Every plane has a vector n normal (perpendicular, orthogonal) to it
- From point-two vector form $P(\alpha,\beta)=R+\alpha u+\beta v$, we know we can use the cross product to find

  $n = u \times v$

  and the equivalent form

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



Linear Independence

- A set of vectors $v_1, v_2, \ldots, v_n$ is *linearly independent* if

  $\alpha_1 v_1+\alpha_2 v_2+.. \alpha_n v_n=0$ iff $\alpha_1=\alpha_2=\ldots=0$
- If a set of vectors is linearly independent, we cannot represent one in terms of the others
- If a set of vectors is linearly dependent, at least one can be written in terms of the others

BREAK

Dimension

- In a vector space, the maximum number of linearly independent vectors is fixed and is called the *dimension* of the space
- In an *n*-dimensional space, any set of n linearly independent vectors form a *basis* * for the space
- Given a basis $v_1, v_2, \ldots, v_n$, any vector v can be written as

  $v=\alpha_1 v_1+ \alpha_2 v_2 +\ldots+\alpha_n v_n$

  where the $\{\alpha_i\}$ are unique

*See text, pp 129-133

Evergreen

Representation

- 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

25

Evergreen

Coordinate Systems

- Consider a basis $v_1, v_2, \ldots, v_n$
- 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

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

26

Evergreen

Example

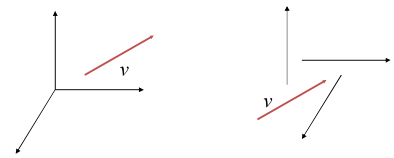
- $v = 2v_1 + 3v_2 - 4v_3$
- $\mathbf{a} = [2 \; 3 \; -4]^T$
- Note that this representation is with respect to a particular basis
- For example, in OpenGL we start by representing vectors using the object basis but later the system needs a representation in terms of the camera or eye basis

27

Evergreen

Coordinate Systems

- Which is correct?



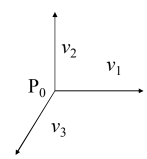
- Both - vectors have no fixed location

28

Evergreen

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*



29

Evergreen

Representation in 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

  $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$

30

Confusing Points and Vectors

Consider the point and the vector

$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$

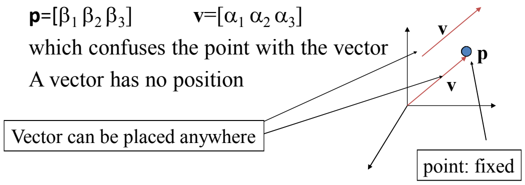
$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$

They appear to have the similar representations

$\mathbf{p}=[\beta_1\ \beta_2\ \beta_3]$ $\quad$ $\mathbf{v}=[\alpha_1\ \alpha_2\ \alpha_3]$

which confuses the point with the vector

A vector has no position



Homogeneous Coordinates

The homogeneous coordinates form for a three dimensional point [x y z] 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

$x \leftarrow x'/w$

$y \leftarrow y'/w$

$z \leftarrow z'/w$

What is w?

If w=0, the representation is that of a vector

Note : homogeneous coordinates replace 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]

A Single Representation

If we define $0 \bullet P = \mathbf{0}$ and $1 \bullet P = P$ then we can write

$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0]\ [v_1\ v_2\ v_3\ P_0]^T$

$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1\ \beta_2\ \beta_3\ 1]\ [v_1\ v_2\ v_3\ P_0]^T$

Thus we obtain the four-dimensional *homogeneous coordinate* representation

$\mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3\ 0]^T$

$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3\ 1]^T$

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 we need a *perspective division*

Change of Coordinate Systems

- Consider two representations of the same vector with respect to two different bases. The representations are

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

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

where

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

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

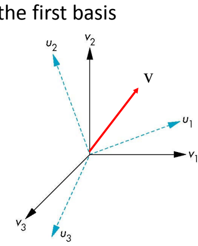
Representing second basis in terms of first

Each of the basis vectors, u1,u2, u3, are vectors that can be represented in terms of the first basis

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

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

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



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

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

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

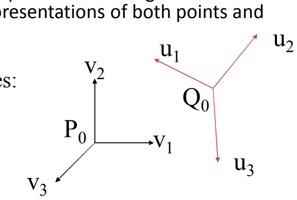
$$\mathbf{a}=\mathbf{M}^T\mathbf{b}$$

see text for numerical examples

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
  e.g., 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

$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$
$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$
$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$
$Q_0 = \gamma_{41}v_1+\gamma_{42}v_2+\gamma_{43}v_3 +\gamma_{44}P_0$

defining a 4 x 4 matrix

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

Working with Representations

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

$\mathbf{a}=[\alpha_1\ \alpha_2\ \ \alpha_3\ \alpha_4]$ in the first frame
$\mathbf{b}=[\beta_1\ \beta_2\ \ \beta_3\ \beta_4]$ in the second frame

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

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

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

Affine Transformations

- Every linear transformation is equivalent to a change in frames
- Every affine transformation preserves lines
- However, an affine transformation has only 12 *degrees of freedom* because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations

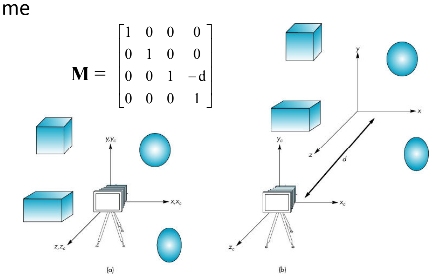
The World and Camera Frames

- 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 ($\mathbf{M}=\mathbf{I}$)

Moving the Camera

If objects are on both sides of z=0, we must move camera frame

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



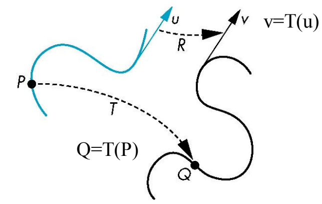
Time-Check….

Transformations

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

General Transformations

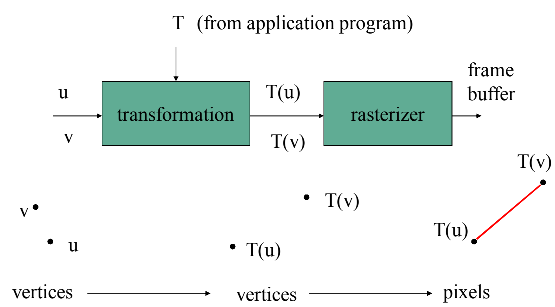
A *transformation* maps points to other points and/or vectors to other vectors



Affine Transformations

- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- In graphics : we need only transform endpoints of line segments. Then, let the implementation draw line segment between the transformed endpoints.

Pipeline Implementation



Notation

We will be working with both coordinate-free representations of transformations and representations within a particular frame

P, Q, R: points in an affine space

u, v, w: vectors in an affine space

α, β, γ: scalars

p, **q**, **r**: representations of points

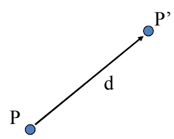
- array of 4 scalars in homogeneous coordinates

u, **v**, **w**: representations of points

- array of 4 scalars in homogeneous coordinates

Translation

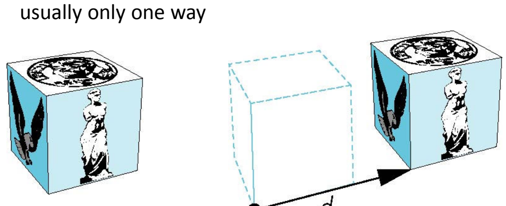
- Move (translate, displace) a point to a new location



- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

How many ways?

Although we can move a point to a new location in infinite ways, when we move many points there is usually only one way



object

translation: every point displaced by same vector

Translation Using Representations

Using the homogeneous coordinate representation in some frame

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

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

$\mathbf{d}=[dx\ dy\ dz\ 0]^T$

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 = vector + point

Translation Matrix

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

$\mathbf{p}'=\mathbf{Tp}$ where

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

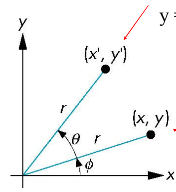
This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated

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

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

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

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

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

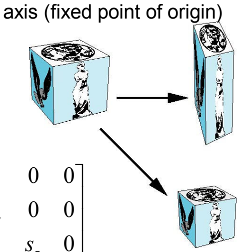
Scaling

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

$$x' = s_x x$$
$$y' = s_y x$$
$$z' = s_z x$$

$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

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

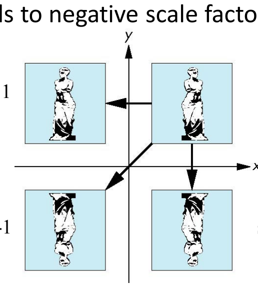


Reflection

corresponds to negative scale factors

$s_x = -1$ $s_y = 1$

original



$s_x = -1$ $s_y = -1$

$s_x = 1$ $s_y = -1$

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 $\sin(-\theta) = -\sin(\theta)$ $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(\theta)$
  - 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

Is this because order matters?

Order of Transformations

- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

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

- Note many references use column matrices to represent points. In terms of column matrices

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

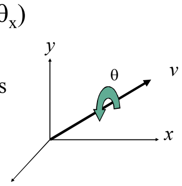
General Rotation About the Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes

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

$\theta_x\ \theta_y\ \theta_z$ are called the Euler angles

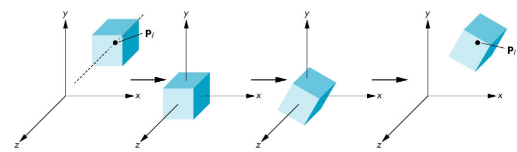
Note that rotations do not commute
We can use rotations in another order but with different angles



Rotation About a Fixed Point other than the Origin

Move fixed point to origin
Rotate
Move fixed point back

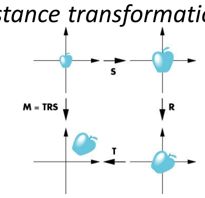
$\mathbf{M} = \mathbf{T}(p_f)\ \mathbf{R}(\theta)\ \mathbf{T}(-p_f)$



Instancing

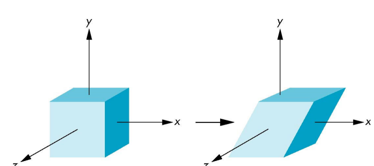
- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an *instance transformation* to its vertices to

  Scale
  Orient
  Locate



Shear

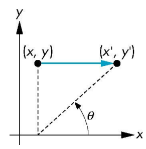
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



Shear Matrix

Consider simple shear along *x* axis

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



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

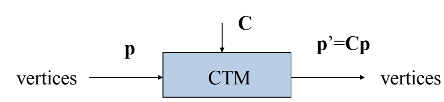
Pre 3.1 OpenGL Matrices

- Matrices were part of the state
- Multiple types
  - Model-View (`GL_MODELVIEW`)
  - Projection (`GL_PROJECTION`)
  - Texture (`GL_TEXTURE`)
  - Color(`GL_COLOR`)
- Single set of functions for manipulation
- Select which to manipulate by
  - `glMatrixMode(GL_MODELVIEW);`
  - `glMatrixMode(GL_PROJECTION);`

Time-Check….

Current Transformation Matrix (CTM)

- Conceptually 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*



OpenGL Transformations

- Learn how to carry out OpenGL transformations
  - Rotation
  - Translation
  - Scaling
- Introduce mat.h & vec.h transformations
  - Model-view
  - Projection

CTM operations

- The CTM can be altered either by loading a new CTM or by postmutiplication

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: $\mathbf{C} \leftarrow \mathbf{C}\ \mathbf{S}$

Rotation about a Fixed Point

Start with identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$
Move fixed point to origin: $\mathbf{C} \leftarrow \mathbf{CT}$
Rotate: $\mathbf{C} \leftarrow \mathbf{CR}$
Move fixed point back: $\mathbf{C} \leftarrow \mathbf{CT}^{-1}$

Result: $\mathbf{C} = \mathbf{TR}\,\mathbf{T}^{-1}$ which is **backwards**.

This result is a consequence of doing postmultiplications.

Let's try again.

Reversing the Order

We want $\mathbf{C} = \mathbf{T}^{-1}\,\mathbf{R}\,\mathbf{T}$
so we must do the operations in the following order

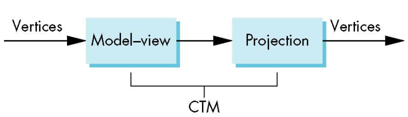
$\mathbf{C} \leftarrow \mathbf{I}$
$\mathbf{C} \leftarrow \mathbf{CT}^{-1}$
$\mathbf{C} \leftarrow \mathbf{CR}$
$\mathbf{C} \leftarrow \mathbf{CT}$

Each operation corresponds to one function call in the program.

Note :
the last operation specified is the first executed in the program

CTM in OpenGL

- OpenGL had a model-view and a projection matrix in the pipeline which were concatenated together to form the CTM
- We will emulate this process



Rotation, Translation, Scaling

Create an identity matrix:

```
mat4 m = Identity();
```

Multiply on right by rotation matrix of `theta` in degrees where (`vx, vy, vz`) define axis of rotation

```
mat4 r = Rotate(theta, vx, vy, vz)
m = m*r;
```

Do same with translation and scaling:

```
mat4 s = Scale( sx, sy, sz)
mat4 t = Transalate(dx, dy, dz);
m = m*s*t;
```


Example

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

```
mat 4 m = Identity();
m = Translate(1.0, 2.0, 3.0)*
  Rotate(30.0, 0.0, 0.0, 1.0)*
  Translate(-1.0, -2.0, -3.0);
```

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

Arbitrary Matrices

- Can load and multiply by matrices defined in the application program
- Matrices are stored as one dimensional array of 16 elements which are the components of the desired 4 x 4 matrix stored by <u>columns</u>
- OpenGL functions that have matrices as parameters allow the application to send the matrix or its transpose

Evergreen

Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - Traversing hierarchical data structures (Chapter 8)
  - Avoiding state changes when executing display lists
- Pre 3.1 OpenGL maintained stacks for each type of matrix
- Easy to create the same functionality with a simple stack class

Evergreen

Reading Back State

- Can also access OpenGL variables (and other parts of the state) by *query* functions

```
glGetIntegerv
glGetFloatv
glGetBooleanv
glGetDoublev
glIsEnabled
```


Evergreen

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 in a standard way
  - Centered at origin
  - Sides aligned with axes

(Will discuss modeling next week)

Evergreen

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();
}
```


Evergreen

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;
 }
```


Evergreen

Display callback

- can form a matrix in the application and
  send it to the shader and
  let shader do the rotation

or

- can send the angle and axis to the shader and
  let the shader form the transformation matrix and
  then do the rotation

More efficient than transforming data in application & resending the data

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glUniform(…); //or glUniformMatrix
     glDrawArrays(…);
   glutSwapBuffers();
}
```

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 a LookAt function
  - Build models of objects
- The projection matrix is used to define the view volume and to select a camera lens
- Although these matrices are no longer part of the OpenGL state, it is usually a good strategy to create them in our own applications

Smooth Rotation

- From a practical standpoint, we are often want to use transformations to move and reorient an object smoothly
  - Problem: find a sequence of model-view matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_n$ so that when they are applied successively to one or more objects we see a smooth transition
- For orientating an object, we can use the fact that every rotation corresponds to part of a great circle on a sphere
  - Find the axis of rotation and angle
  - Virtual trackball (see text)

Incremental Rotation

- Consider the two approaches
  - For a sequence of rotation matrices $\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_n$, find the Euler angles for each and use $\mathbf{R}_i = \mathbf{R}_{iz} \mathbf{R}_{iy} \mathbf{R}_{ix}$
    - Not very efficient
  - Use the final positions to determine the axis and angle of rotation, then increment only the angle
- *Quaternions** can be more efficient than either

*p. 186

Interfaces

- One of the major problems in interactive computer graphics is how to use two-dimensional devices such as a mouse to interface with three dimensional objects
- Example: how to form an *instance matrix**?
- Some alternatives
  - Virtual trackball
  - 3D input devices such as the spaceball
  - Use areas of the screen
    - Distance from center controls angle, position, scale depending on mouse button depressed

*p. 168-169?

Lab/Asst 05: A Simple Program (?)

Generate a square on a solid background