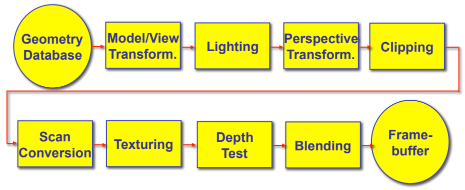
University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2013

Tamara Munzner

## Transformations

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2013

## Review: Rendering Pipeline






## Review: Graphics State

- set the state once, remains until overwritten
  - glColor3f(1.0, 1.0, 0.0) → set color to yellow
  - glSetClearColor(0.0, 0.0, 0.2) → dark blue bg
  - glEnable(LIGHT0) → turn on light
  - glEnable(GL_DEPTH_TEST) → hidden surf.



## Review: Geometry Pipeline

- tell it how to interpret geometry
  - glBegin(<*mode of geometric primitives*>)
  - *mode* = GL_TRIANGLE, GL_POLYGON, etc.
- feed it vertices
  - glVertex3f(-1.0, 0.0, -1.0)
  - glVertex3f(1.0, 0.0, -1.0)
  - glVertex3f(0.0, 1.0, -1.0)
- tell it you're done
  - glEnd()



## Review: GLUT: OpenGL Utility Toolkit

- simple, portable window manager
  - opening windows
    - handling graphics contexts
  - handling input with callbacks
    - keyboard, mouse, window reshape events
  - timing
    - idle processing, idle events
- designed for small/medium size applications



## Readings for Transformations I-IV

- FCG Chap 6 Transformation Matrices
  - *except* 6.1.6, 6.3.1
- FCG Sect 13.3 Scene Graphs (3rd ed: 12.2)
- RB Chap Viewing
  - Viewing and Modeling Transforms *until* Viewing Transformations
  - Examples of Composing Several Transformations *through* Building an Articulated Robot Arm
- RB Appendix Homogeneous Coordinates and Transformation Matrices
  - *until* Perspective Projection
- RB Chap Display Lists

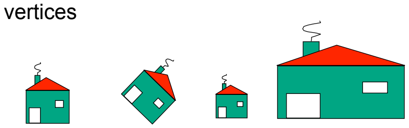


## 2D Transformations



## Transformations

- transforming an object = transforming all its points
- transforming a polygon = transforming its vertices





## Matrix Representation

- represent 2D transformation with matrix
  - multiply matrix by column vector $\Leftrightarrow$ apply transformation to point

$$\begin{bmatrix}x'\\y'\end{bmatrix}=\begin{bmatrix}a & b\\c & d\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} \qquad \begin{matrix}x'=ax+by\\y'=cx+dy\end{matrix}$$

- transformations combined by multiplication

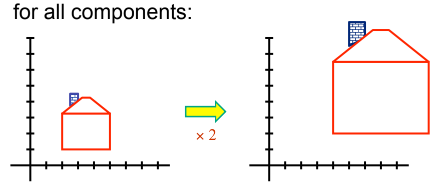
$$\begin{bmatrix}x'\\y'\end{bmatrix}=\begin{bmatrix}a & b\\c & d\end{bmatrix}\begin{bmatrix}d & e\\f & g\end{bmatrix}\begin{bmatrix}h & i\\j & k\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

  - matrices are efficient, convenient way to represent sequence of transformations!



## Scaling

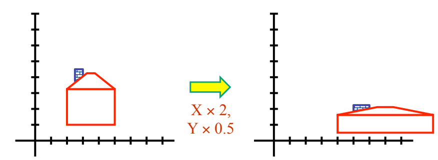
- scaling a coordinate means multiplying each of its components by a scalar
- uniform scaling means this scalar is the same for all components:






## Scaling

- non-uniform scaling: different scalars per component:




- how can we represent this in matrix form?

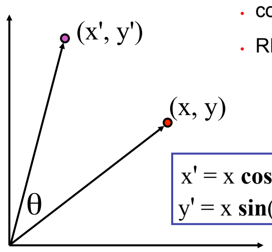


## Scaling

- scaling operation: $\begin{bmatrix}x'\\y'\end{bmatrix}=\begin{bmatrix}ax\\by\end{bmatrix}$
- or, in matrix form: $\begin{bmatrix}x'\\y'\end{bmatrix}=\underbrace{\begin{bmatrix}a & 0\\0 & b\end{bmatrix}}_{\textit{scaling matrix}}\begin{bmatrix}x\\y\end{bmatrix}$



## 2D Rotation

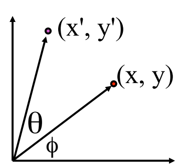



- counterclockwise
- RHS

$$\boxed{\begin{matrix}x' = x\,\mathbf{cos}(\theta) - y\,\mathbf{sin}(\theta)\\ y' = x\,\mathbf{sin}(\theta) + y\,\mathbf{cos}(\theta)\end{matrix}}$$



## 2D Rotation From Trig Identities



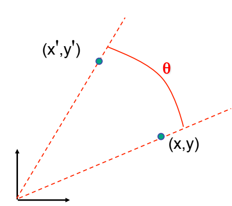

$x = r\cos(\phi)$
$y = r\sin(\phi)$
$x' = r\cos(\phi+\theta)$
$y' = r\sin(\phi+\theta)$

Trig Identity…
$x' = r\cos(\phi)\cos(\theta) - r\sin(\phi)\sin(\theta)$
$y' = r\sin(\phi)\cos(\theta) + r\cos(\phi)\sin(\theta)$

Substitute…
$x' = x\,\mathbf{cos}(\theta) - y\,\mathbf{sin}(\theta)$
$y' = x\,\mathbf{sin}(\theta) + y\,\mathbf{cos}(\theta)$



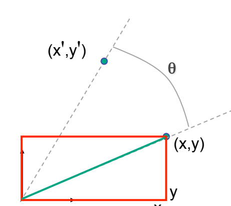
## 2D Rotation: Another Derivation




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

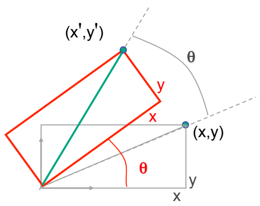


## 2D Rotation: Another Derivation




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

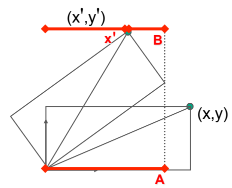
## 2D Rotation: Another Derivation




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



## 2D Rotation: Another Derivation

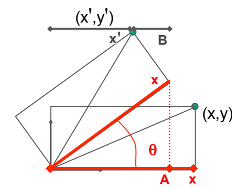



$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

$$x' = A - B$$



## 2D Rotation: Another Derivation

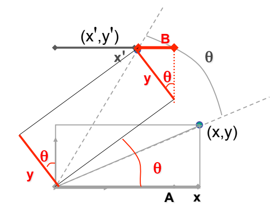



$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

$$x' = A - B$$
$$A = x\cos\theta$$



## 2D Rotation: Another Derivation




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

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



## 2D Rotation Matrix

- easy to capture in matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

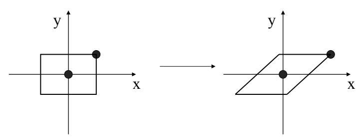
- even though sin(q) and cos(q) are nonlinear functions of q,
  - x' is a linear combination of x and y
  - y' is a linear combination of x and y



## Shear

- shear along x axis
  - push points to right in proportion to height

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} ? \\ ? \end{bmatrix}$$

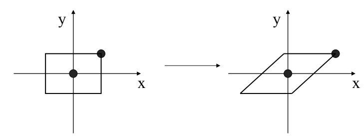





## Shear

- shear along x axis
  - push points to right in proportion to height

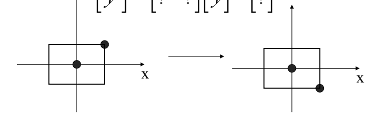
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & sh_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$






## Reflection

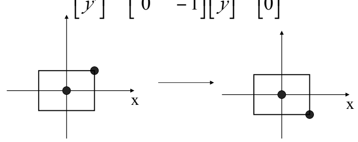
- reflect across x axis
  - mirror $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} ? \\ ? \end{bmatrix}$





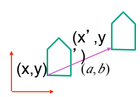

## Reflection

- reflect across x axis
  - mirror $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$





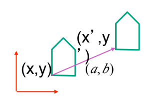

## 2D Translation




$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$



## 2D Translation

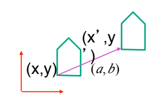



$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\textit{scaling matrix}} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}}_{\textit{rotation matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$



## 2D Translation




vector addition

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

matrix multiplication

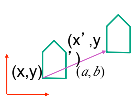
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\textit{scaling matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$

matrix multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}}_{\textit{rotation matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$



## 2D Translation




vector addition

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

matrix multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\textit{scaling matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$

matrix multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}}_{\textit{rotation matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{\textit{translation multiplication matrix??}} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$



## Linear Transformations

- linear transformations are combinations of
  - shear
  - scale
  - rotate
  - reflect

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned}$$

- properties of linear transformations
  - satisifes T(*s***x**+*t***y**) = *s* T(**x**) + *t* T(**y**)
  - origin maps to origin
  - lines map to lines
  - parallel lines remain parallel
  - ratios are preserved
  - closed under composition



## Challenge

- matrix multiplication
  - for everything except translation
  - how to do everything with multiplication?
    - then just do composition, no special cases
- homogeneous coordinates trick
  - represent 2D coordinates (x,y) with 3-vector (x,y,1)



## Homogeneous Coordinates

- our 2D transformation matrices are now 3x3:

$$\mathbf{R}otation = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{S}cale = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

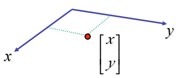
$$\mathbf{T}ranslation = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

- use rightmost column

$$\begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x*1 + a*1 \\ y*1 + b*1 \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ 1 \end{bmatrix}$$

## Homogeneous Coordinates Geometrically

- point in 2D cartesian

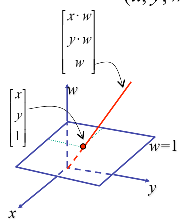





## Homogeneous Coordinates Geometrically

homogeneous → cartesian

$$(x, y, w) \xrightarrow{/w} \left(\frac{x}{w}, \frac{y}{w}\right)$$



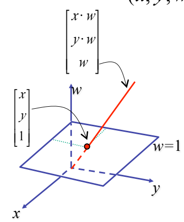

- point in 2D cartesian + weight w = point P in 3D homog. coords
- multiples of (x,y,w)
  - form a line L in 3D
  - all homogeneous points on L represent same 2D cartesian point
  - example: (2,2,1) = (4,4,2) = (1,1,0.5)



## Homogeneous Coordinates Geometrically

homogeneous → cartesian

$$(x, y, w) \xrightarrow{/w} \left(\frac{x}{w}, \frac{y}{w}\right)$$




- homogenize to convert homog. 3D point to cartesian 2D point:
  - divide by w to get (x/w, y/w, 1)
  - projects line to point onto w=1 plane
  - like normalizing, one dimension up
- when w=0, consider it as direction
  - points at infinity
  - these points cannot be homogenized
  - lies on x-y plane
- (0,0,0) is undefined



## Affine Transformations

- affine transforms are combinations of
  - linear transformations
  - translations

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- properties of affine transformations
  - origin does not necessarily map to origin
  - lines map to lines
  - parallel lines remain parallel
  - ratios are preserved
  - closed under composition



## Homogeneous Coordinates Summary

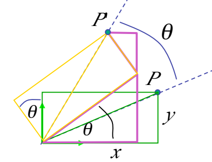
- may seem unintuitive, but they make graphics operations much easier
- allow all affine transformations to be expressed through matrix multiplication
  - we'll see even more later...
- use 3x3 matrices for 2D transformations
  - use 4x4 matrices for 3D transformations



## 3D Transformations



## 3D Rotation About Z Axis




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

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

- general OpenGL command
  **glRotatef(angle,x,y,z);**
- rotate in z
  **glRotatef(angle,0,0,1);**



## 3D Rotation in X, Y

around x axis: **glRotatef(angle,1,0,0);**

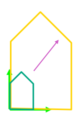
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

around y axis: **glRotatef(angle,0,1,0);**

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



## 3D Scaling

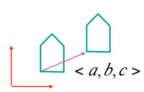


$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**glScalef(a,b,c);**



## 3D Translation




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

**glTranslatef(a,b,c);**



## 3D Shear

- general shear $shear(hxy, hxz, hyx, hyz, hzx, hzy) = \begin{bmatrix} 1 & hyx & hzx & 0 \\ hxy & 1 & hzy & 0 \\ hxz & hyz & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- to avoid ambiguity, always say "shear along <axis> in direction of <axis>"

$$shearAlongXinDirectionOfY(h) = \begin{bmatrix} 1 & h & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad shearAlongXinDirectionOfZ(h) = \begin{bmatrix} 1 & 0 & h & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$shearAlongYinDirectionOfX(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ h & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad shearAlongYinDirectionOfZ(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & h & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$shearAlongZinDirectionOfX(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ h & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad shearAlongZinDirectionOfY(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



## Summary: Transformations

**translate(a,b,c)**

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & & & a \\ & 1 & & b \\ & & 1 & c \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**scale(a,b,c)**

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & & & \\ & b & & \\ & & c & \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$\text{Rotate}(x, \theta)$

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

$\text{Rotate}(y, \theta)$

$$\begin{bmatrix} \cos\theta & & \sin\theta & \\ & 1 & & \\ -\sin\theta & & \cos\theta & \\ & & & 1 \end{bmatrix}$$

$\text{Rotate}(z, \theta)$

$$\begin{bmatrix} \cos\theta & -\sin\theta & & \\ \sin\theta & \cos\theta & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$



## Undoing Transformations: Inverses

$$\mathbf{T}(x,y,z)^{-1} = \mathbf{T}(-x,-y,-z)$$
$$\mathbf{T}(x,y,z)\,\mathbf{T}(-x,-y,-z) = \mathbf{I}$$

$$\mathbf{R}(z,\theta)^{-1} = \mathbf{R}(z,-\theta) = \mathbf{R}^{\mathbf{T}}(z,\theta)$$ **(R is orthogonal)**
$$\mathbf{R}(z,\theta)\,\mathbf{R}(z,-\theta) = \mathbf{I}$$

$$\mathbf{S}(sx,sy,sz)^{-1} = \mathbf{S}\left(\frac{1}{sx},\frac{1}{sy},\frac{1}{sz}\right)$$
$$\mathbf{S}(sx,sy,sz)\,\mathbf{S}\left(\frac{1}{sx},\frac{1}{sy},\frac{1}{sz}\right) = \mathbf{I}$$



## Composing Transformations



## Composing Transformations

- translation

$$T1 = T(dx_1, dy_1) = \begin{bmatrix} 1 & & & dx_1 \\ & 1 & & dy_1 \\ & & 1 & \\ & & & 1 \end{bmatrix} \quad T2 = T(dx_2, dy_2) = \begin{bmatrix} 1 & & & dx_2 \\ & 1 & & dy_2 \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

$$P'' = T2 \bullet P' = T2 \bullet [T1 \bullet P] = [T2 \bullet T1] \bullet P, \textit{where}$$

$$T2 \bullet T1 = \begin{bmatrix} 1 & & & dx_1 + dx_2 \\ & 1 & & dy_1 + dy_2 \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

**so translations add**



## Composing Transformations

- scaling

$$S2 \bullet S1 = \begin{bmatrix} sx_1 * dx_2 & & & \\ & sy_1 * sy_2 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

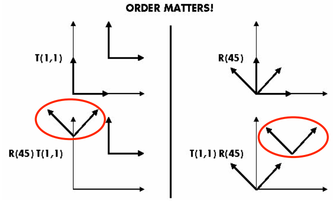
**so scales multiply**

- rotation

$$R2 \bullet R1 = \begin{bmatrix} \cos(\theta 1 + \theta 2) & -\sin(\theta 1 + \theta 2) & & \\ \sin(\theta 1 + \theta 2) & \cos(\theta 1 + \theta 2) & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

**so rotations add**

## Composing Transformations

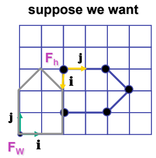



**Ta Tb = Tb Ta, but Ra Rb != Rb Ra and Ta Rb != Rb Ta**

- translations commute
- rotations around same axis commute
- rotations around different axes do not commute
- rotations and translations do not commute

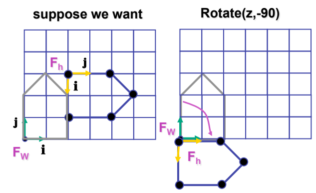


## Composing Transformations





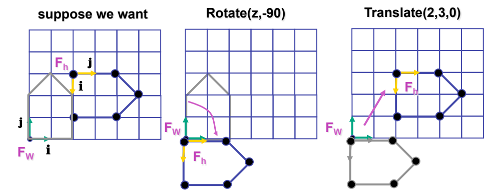

## Composing Transformations




$\mathbf{p'} = \mathbf{R}(z,-90)\,\mathbf{p}$



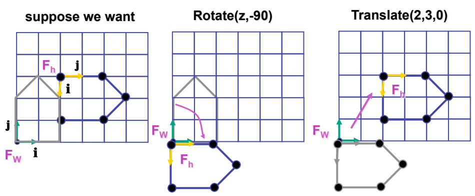
## Composing Transformations




$\mathbf{p'} = \mathbf{R}(z,-90)\,\mathbf{p}$ $\mathbf{p''} = \mathbf{T}(2,3,0)\,\mathbf{p'}$



## Composing Transformations




$\mathbf{p'} = \mathbf{R}(z,-90)\,\mathbf{p}$ $\mathbf{p''} = \mathbf{T}(2,3,0)\,\mathbf{p'}$

$\mathbf{p''} = \mathbf{T}(2,3,0)\,\mathbf{R}(z,-90)\,\mathbf{p} = \mathbf{TRp}$



## Composing Transformations

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

- which direction to read?



## Composing Transformations

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

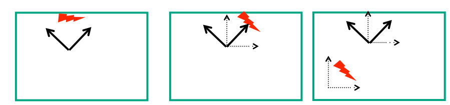
- which direction to read?
  - right to left
    - interpret operations wrt fixed coordinates
    - moving object
  - left to right
    - interpret operations wrt local coordinates
    - changing coordinate system
    - in OpenGL, cannot move object once it is drawn!!
      - object specified as set of coordinates wrt specific coord sys



## Composing Transformations

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

- which direction to read? ←
  - right to left
    - interpret operations wrt fixed coordinates
    - moving object
      - draw thing
      - rotate thing by -45 degrees wrt origin
      - translate it (-2, -3) over

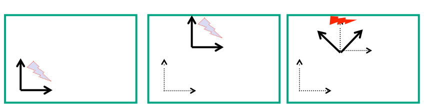




## Composing Transformations

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

- which direction to read? →
  - left to right
    - interpret operations wrt local coordinates
    - changing coordinate system
      - translate coordinate system (2, 3) over
      - rotate coordinate system 45 degrees wrt origin
      - draw object in current coordinate system





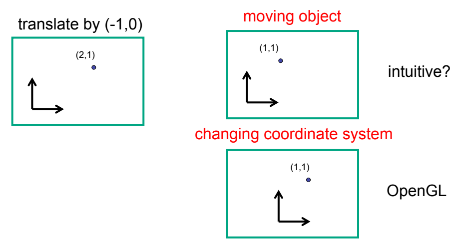
## Composing Transformations

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

- which direction to read?
  - right to left
    - interpret operations wrt fixed coordinates
    - moving object
  - left to right **OpenGL pipeline ordering!**
    - interpret operations wrt local coordinates
    - changing coordinate system
    - OpenGL updates current matrix with postmultiply
      - glTranslatef(2,3,0);
      - glRotatef(-90,0,0,1);
      - glVertexf(1,1,1);
    - specify vector last, in final coordinate system
    - first matrix to affect it is specified second-to-last



## Interpreting Transformations




- same relative position between object and basis vectors

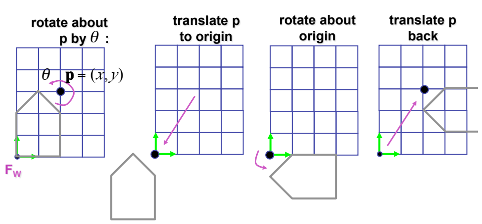


## Matrix Composition

- matrices are convenient, efficient way to represent series of transformations
  - general purpose representation
  - hardware matrix multiply
  - matrix multiplication is associative
    - **p′** = (T*(R*(S***p**)))
    - **p′** = (T*R*S)***p**
- procedure
  - correctly order your matrices!
  - multiply matrices together
  - result is one matrix, multiply vertices by this matrix
  - all vertices easily transformed with one matrix multiply



## Rotation About a Point: Moving Object

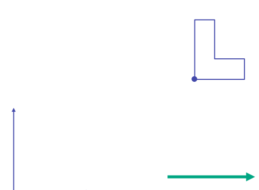



$\mathbf{T}(x,y,z)\,\mathbf{R}(z,\theta)\,\mathbf{T}(-x,-y,-z)$



## Rotation: Changing Coordinate Systems

- same example: rotation around arbitrary center

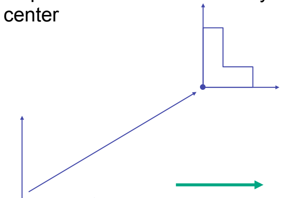


$\mathbf{T}(x,y,z)\,\mathbf{R}(z,\theta)\,\mathbf{T}(-x,-y,-z)$



## Rotation: Changing Coordinate Systems

- rotation around arbitrary center
  - step 1: translate coordinate system to rotation center

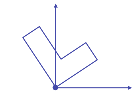


$\mathbf{T}(x,y,z)\,\mathbf{R}(z,\theta)\,\mathbf{T}(-x,-y,-z)$



## Rotation: Changing Coordinate Systems

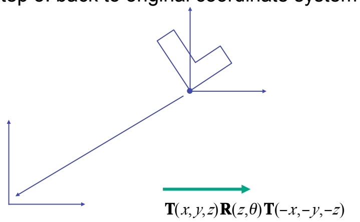
- rotation around arbitrary center
  - step 2: perform rotation



$\mathbf{T}(x,y,z)\,\mathbf{R}(z,\theta)\,\mathbf{T}(-x,-y,-z)$

## Rotation: Changing Coordinate Systems

- rotation around arbitrary center
  - step 3: back to original coordinate system



$$\mathbf{T}(x,y,z)\mathbf{R}(z,\theta)\mathbf{T}(-x,-y,-z)$$



## General Transform Composition

- transformation of geometry into coordinate system where operation becomes simpler
  - typically translate to origin
- perform operation
- transform geometry back to original coordinate system

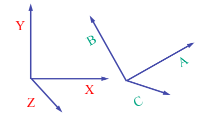


## Rotation About an Arbitrary Axis

- axis defined by two points
- translate point to the origin
- rotate to align axis with z-axis (or x or y)
- perform rotation
- undo aligning rotations
- undo translation

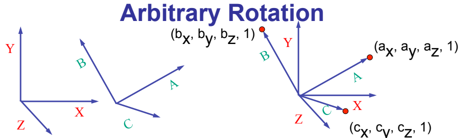


## Arbitrary Rotation



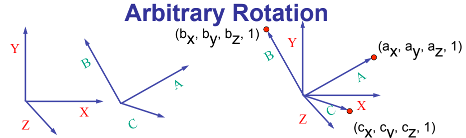

- arbitrary rotation: change of basis
  - given two orthonormal coordinate systems *XYZ* and *ABC*
    - *A*'s location in the XYZ coordinate system is ($a_x$, $a_y$, $a_z$, 1), ...

## Arbitrary Rotation




- arbitrary rotation: change of basis
  - given two orthonormal coordinate systems *XYZ* and *ABC*
    - *A*'s location in the XYZ coordinate system is ($a_x$, $a_y$, $a_z$, 1), ...

## Arbitrary Rotation




- arbitrary rotation: change of basis
  - given two orthonormal coordinate systems *XYZ* and *ABC*
    - *A*'s location in the XYZ coordinate system is ($a_x$, $a_y$, $a_z$, 1), ...
- transformation from one to the other is matrix R whose columns are *A,B,C*:

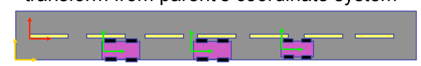
$$R(X) = \begin{bmatrix} a_x & b_x & c_x & 0 \\ a_y & b_y & c_y & 0 \\ a_z & b_z & c_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = (a_x, a_y, a_z, 1) = A$$



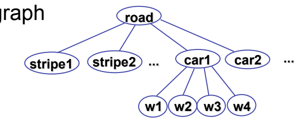
## Transformation Hierarchies

## Transformation Hierarchies

- scene may have a hierarchy of coordinate systems
  - stores matrix at each level with incremental transform from parent's coordinate system

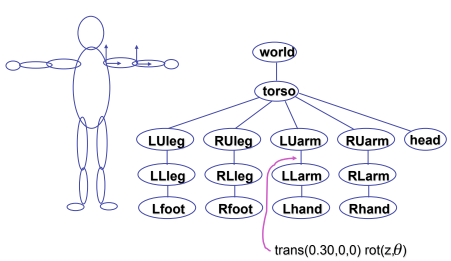


- scene graph





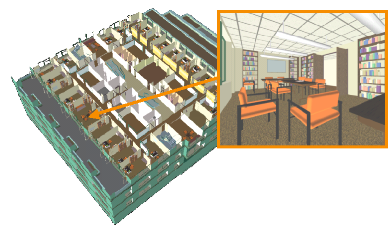

## Transformation Hierarchy Example 1






## Transformation Hierarchy Example 2

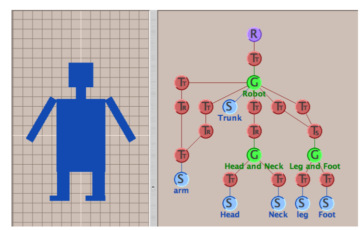
- draw same 3D data with different transformations: instancing





## Transformation Hierarchies Demo

- transforms apply to graph nodes beneath

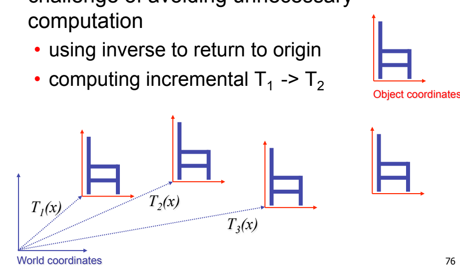



http://www.cs.brown.edu/exploratories/freeSoftware/catalogs/scenegraphs.html



## Matrix Stacks

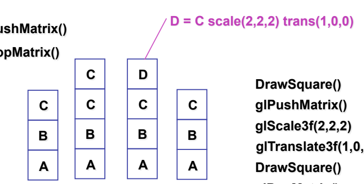
- challenge of avoiding unnecessary computation
  - using inverse to return to origin
  - computing incremental $T_1$ -> $T_2$






## Matrix Stacks

glPushMatrix()
glPopMatrix()



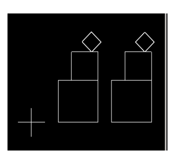

DrawSquare()
glPushMatrix()
glScale3f(2,2,2)
glTranslate3f(1,0,0)
DrawSquare()
glPopMatrix()



## Modularization

- drawing a scaled square
  - push/pop ensures no coord system change

```
void drawBlock(float k) {
  glPushMatrix();

  glScalef(k,k,k);
  glBegin(GL_LINE_LOOP);
  glVertex3f(0,0,0);
  glVertex3f(1,0,0);
  glVertex3f(1,1,0);
  glVertex3f(0,1,0);
  glEnd();

  glPopMatrix();
}
```



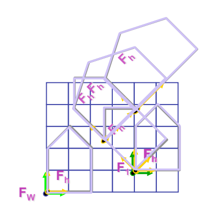


## Matrix Stacks

- advantages
  - no need to compute inverse matrices all the time
  - modularize changes to pipeline state
  - avoids incremental changes to coordinate systems
    - accumulation of numerical errors
- practical issues
  - in graphics hardware, depth of matrix stacks is limited
    - (typically 16 for model/view and about 4 for projective matrix)

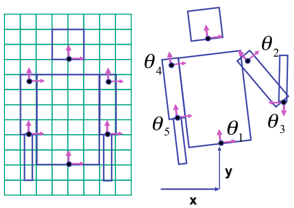


## Transformation Hierarchy Example 3




```
glLoadIdentity();
glTranslatef(4,1,0);
glPushMatrix();
glRotatef(45,0,0,1);
glTranslatef(0,2,0);
glScalef(2,1,1);
glTranslate(1,0,0);
glPopMatrix();
```

## Transformation Hierarchy Example 4




glTranslate3f(x,y,0);
glRotatef($\theta_1$,0,0,1);
DrawBody();
glPushMatrix();
  glTranslate3f(0,7,0);
  DrawHead();
glPopMatrix();
glPushMatrix();
  glTranslate(2.5,5.5,0);
  glRotatef($\theta_2$,0,0,1);
  DrawUArm();
  glTranslate(0,-3.5,0);
  glRotatef($\theta_3$,0,0,1);
  DrawLArm();
glPopMatrix();
... (draw other arm)



## Hierarchical Modelling

- advantages
  - define object once, instantiate multiple copies
  - transformation parameters often good control knobs
  - maintain structural constraints if well-designed
- limitations
  - expressivity: not always the best controls
  - can't do closed kinematic chains
    - keep hand on hip
  - can't do other constraints
    - collision detection
      - self-intersection
      - walk through walls



## Display Lists



## Display Lists

- precompile/cache block of OpenGL code for reuse
  - usually more efficient than immediate mode
    - exact optimizations depend on driver
  - good for multiple instances of same object
    - but cannot change contents, not parametrizable
  - good for static objects redrawn often
    - display lists persist across multiple frames
    - interactive graphics: objects redrawn every frame from new viewpoint from moving camera
  - can be nested hierarchically
- snowman example
  http://www.lighthouse3d.com/opengl/displaylists

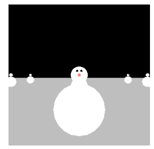


## One Snowman

```
void drawSnowMan() {

glColor3f(1.0f, 1.0f, 1.0f);

// Draw Body
glTranslatef(0.0f ,0.75f, 0.0f);
glutSolidSphere(0.75f,20,20);

// Draw Head
glTranslatef(0.0f, 1.0f, 0.0f);
glutSolidSphere(0.25f,20,20);

// Draw Eyes
glPushMatrix();
glColor3f(0.0f,0.0f,0.0f);
glTranslatef(0.05f, 0.10f, 0.18f);
glutSolidSphere(0.05f,10,10);
glTranslatef(-0.1f, 0.0f, 0.0f);
glutSolidSphere(0.05f,10,10);
glPopMatrix();

// Draw Nose
glColor3f(1.0f, 0.5f , 0.5f);
glRotatef(0.0f,1.0f, 0.0f, 0.0f);
glutSolidCone(0.08f,0.5f,10,2);
}
```



## Instantiate Many Snowmen



```
// Draw 36 Snowmen
for(int i = -3; i < 3; i++)
  for(int j=-3; j < 3; j++) {
  glPushMatrix();
  glTranslatef(i*10.0, 0, j * 10.0);
  // Call the function to draw a snowman
  drawSnowMan();
  glPopMatrix();
}
```

36K polygons, 55 FPS



## Making Display Lists

```
GLuint createDL() {
GLuint snowManDL;
// Create the id for the list
snowManDL = glGenLists(1);
glNewList(snowManDL,GL_COMPILE);
drawSnowMan();
glEndList();
return(snowManDL); }

snowmanDL = createDL();
for(int i = -3; i < 3; i++)
  for(int j=-3; j < 3; j++) {
    glPushMatrix();
    glTranslatef(i*10.0, 0, j * 10.0);
    glCallList(snowManDL);
    glPopMatrix(); }
```

36K polygons, 153 FPS



## Transforming Normals

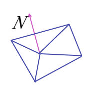


## Transforming Geometric Objects

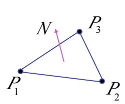
- lines, polygons made up of vertices
  - transform the vertices
  - interpolate between
- does this work for everything? no!
  - normals are trickier



## Computing Normals




- normal
  - direction specifying orientation of polygon
    - w=0 means direction with homogeneous coords
    - vs. w=1 for points/vectors of object vertices
  - used for lighting
    - must be normalized to unit length
  - can compute if not supplied with object




$$N = (P_2 - P_1) \times (P_3 - P_1)$$



## Transforming Normals

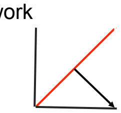
$$\begin{bmatrix} x' \\ y' \\ z' \\ 0 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & T_x \\ m_{21} & m_{22} & m_{23} & T_y \\ m_{31} & m_{32} & m_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

- so if points transformed by matrix **M**, can we just transform normal vector by **M** too?
  - translations OK: w=0 means unaffected
  - rotations OK
  - uniform scaling OK
- these all maintain direction



## Transforming Normals

- nonuniform scaling does not work
- x-y=0 plane
  - line x=y
  - normal: [1,-1,0]
    - direction of line x=-y
    - (ignore normalization for now)

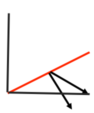




## Transforming Normals

- apply nonuniform scale: stretch along x by 2
  - new plane x = 2y
- transformed normal: [2,-1,0]

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



  - normal is direction of line x = -2y or x+2y=0
  - not perpendicular to plane!
  - should be direction of 2x = -y



## Planes and Normals

- plane is all points perpendicular to normal
  - $N \bullet P = 0$ (with dot product)
  - $N^T \bullet P = 0$ (matrix multiply requires transpose)

$$N = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- explicit form: plane = $ax + by + cz + d$



## Finding Correct Normal Transform

- transform a plane

$$\begin{matrix} P \\ N \end{matrix} \longrightarrow \begin{matrix} P' = MP \\ N' = QN \end{matrix}$$

given M, what should Q be?

$N'^T P' = 0$ — stay perpendicular

$(QN)^T (MP) = 0$ — substitute from above

$N^T \underbrace{Q^T M}P = 0$ — $(\mathbf{AB})^\mathbf{T} = \mathbf{B}^\mathbf{T}\mathbf{A}^\mathbf{T}$

$Q^T M = I$ — $\mathbf{N}^\mathbf{T}\mathbf{P} = 0$ if $\mathbf{Q}^\mathbf{T}\mathbf{M} = \mathbf{I}$

$$\boxed{\mathbf{Q} = \left(\mathbf{M}^{-1}\right)^\mathbf{T}}$$

thus the normal to any surface can be transformed by the inverse transpose of the modelling transformation



## Reading for Next Topic: Viewing

- FCG Chapter 7 Viewing
- FCG Section 6.3.1 Windowing Transforms
- RB rest of Chap Viewing
- RB rest of App Homogeneous Coords