Transformations

2D Rotation

\[
\begin{align*}
(x', y') &= (x \cos \theta - y \sin \theta, \quad x \sin \theta + y \cos \theta) \\
\theta &\text{ counterclockwise} \\
\text{RHS}
\end{align*}
\]

Matrix Representation

- represent 2D transformation with matrix
- apply transformation to point
- transformations combined by multiplication
- 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:

\[
\begin{align*}
\begin{bmatrix}
    x' \\
    y'
\end{bmatrix} &= \begin{bmatrix}
    a & b \\
    c & d
\end{bmatrix} \begin{bmatrix}
    x \\
    y
\end{bmatrix} \\
% &= \begin{bmatrix}
    ax + by \\
    cx + dy
\end{bmatrix}
\end{align*}
\]

2D Rotation From Trig Identities

- Trig Identity: 
- Substitute:

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 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(\theta) and cos(\theta) are nonlinear functions of \(\theta\),
  - \(x'\) 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

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

Reflection

- reflect across x axis
  - mirror

2D Translation

- vector addition

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

Linear Transformations

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

- properties of linear transformations
  - satisfies T(ax + by) = aT(x) + bT(y)
  - origin maps to origin
  - lines map to lines
  - parallel lines remain parallel
  - ratios are preserved
  - closed under composition

Homogeneous Coordinates

- our 2D transformation matrices are now 3x3:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} A \\ B \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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 Geometrically

- point in 2D cartesian

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

- point in 3D 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 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
  - general OpenGL command
  - rotate in z
    - glRotate(angle, 0, 0, 1);

- 3D Rotation in X, Y
  - around x axis: glRotate(angle, 1,0,0);
  - around y axis: glRotate(angle, 0,1,0);

3D Scaling

- general shear
  - to avoid ambiguity, always say "shear along <axis> in direction of <vector>"

3D Translation

Summary: Transformations

- translate(a,b,c)
- scale(a,b,c)
- rotate around x axis:
- rotate around y axis:
- rotate around z axis:

Undoing Transformations: Inverses

$$\mathbf{T}(x,y,z)^{-1} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

$$\mathbf{T}_0(x,y,z) = \mathbf{T}^{-1}(x',y',z')$$

- R is orthogonal
- \( R(\theta) R(-\theta) = \mathbf{I} \)

- so rotations add
- so translations add
- so scales multiply

Composing Transformations

- translation
- scaling
- rotation
Composing Transformations

Transformations commute:
- Rotations around same axis commute
- Rotations around different axes do not commute
- Translations do not commute

Rotation: Changing Coordinate Systems

- Suppose we want to change coordinate system
- Right to left:
  - Interpret operations wrt fixed coordinates
  - Moving object
  - Left to right:
    - Interpret operations wrt local coordinates
    - Changing coordinate system

OpenGL pipeline ordering!

Matrix Composition

- Matrices are convenient, efficient way to represent series of transformations
- General purpose representation
- Hardware matrix multiply
- Matrix multiplication is associative
- \( p' = (T \circ \{R(z, -90)\}) \circ p \)
- \( p' = (T \circ \{R(z, -90)\}) \circ p \)
- 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: Changing Coordinate Systems

- Same example: rotation around arbitrary center
- Step 1: Translate coordinate system to rotation center
- Step 2: Perform rotation

Rotation About a Point: Moving Object

- Rotate about \( p \) by \( \theta \):
- Translate \( p \) to origin
- Rotate about origin
- Translate \( p \) back

Rotation: Changing Coordinate Systems

- Same example: rotation around arbitrary center
- Step 1: Translate coordinate system to rotation center
- Step 2: Perform rotation

Interpreting Transformations

- Translate by \( (1,0) \)
- Moving object
- Intuitive?
- Changing coordinate system
- OpenGL
Rotation: Changing Coordinate Systems

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

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
  - it's location in the XYZ coordinate system is \((b_x, b_y, b_z, 1)\), ...

Arbitrary Rotation

- arbitrary rotation: change of basis
  - given two orthonormal coordinate systems XYZ and ABC
  - transformation from one to the other is matrix \(R\) whose columns are \(A, B, C\):
    \[
    R = \begin{bmatrix}
    a_x & a_y & a_z & 0 \\
    b_x & b_y & b_z & 0 \\
    0 & 0 & 0 & 1 \\
    \end{bmatrix}
    \]

Transformation Hierarchies

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

Transformation Hierarchies Demo

- transforms apply to graph nodes beneath

Transformation Hierarchy Example 2

- draw same 3D data with different transformations: instancing

Transformation Hierarchy Example 3

- scene graph

Matrix Stacks

- challenge of avoiding unnecessary computation
- using inverse to return to origin
- computing incremental \(T_1 \rightarrow T_2\)

Transformation Hierarchy Example 1

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

Matrix Stacks

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

Modularization

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

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

Transformation Hierarchies Demo

- transforms apply to graph nodes beneath

Matrix Stacks

- challenge of avoiding unnecessary computation
- using inverse to return to origin
- computing incremental \(T_1 \rightarrow T_2\)
Transformation Hierarchy Example 4

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
- 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 parameterizable
- 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

Instantiate Many Snowmen

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

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)

Planes and Normals
- plane is all points perpendicular to normal
  - \( N \times P = 0 \) (with dot product)
  - \( N^T \times P = 0 \) (matrix multiply requires transpose)
- explicit form: plane = \( ax + by + cz + d \)

Finding Correct Normal Transform
- transform a plane
  \[ \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b & c \\ 0 & 0 & 0 \\ d \end{bmatrix} \begin{bmatrix} P \end{bmatrix} \]
  \[ N = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \quad P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \text{given } M, \ \text{what should } Q \text{ be?} \]
  - stay perpendicular
    - substitute from above
      - translations OK: w=0 means unaffected
      - rotations OK
      - uniform scaling OK
  - these all maintain direction

Display Lists
- translate 3f(x,y,0);
- glRotatef(,0,0,1);
- DrawBody();
- glPushMatrix();
- glTranslatef();
- glScalef(2,1,1);
- DrawHead();
- glPopMatrix();
  ...
  (draw other arm)
  
  
Transforming Normals
- making display lists
  - createDL();
  - createDL();
  - createDL();
  - createDL();
  - createDL();
  - Call the function to draw a snowman
  - drawSnowMan();
  - glPopMatrix();
}

36K polygons, 55 FPS

Transforming Normals
- planes and normals
  - plane is all points perpendicular to normal
    - \( N^T \times P = 0 \) (matrix multiply requires transpose)

Finding Correct Normal Transform
- transform a plane
  \[ \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b & c \\ 0 & 0 & 0 \\ d \end{bmatrix} \begin{bmatrix} P \end{bmatrix} \]
  \[ N = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \quad P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \text{given } M, \ \text{what should } Q \text{ be?} \]
  - stay perpendicular
    - substitute from above
      - translations OK: w=0 means unaffected
      - rotations OK
      - uniform scaling OK
  - these all maintain direction

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

Transforming Normals
- planes and normals
  - plane is all points perpendicular to normal
    - \( N^T \times P = 0 \) (matrix multiply requires transpose)

Finding Correct Normal Transform
- transform a plane
  \[ \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b & c \\ 0 & 0 & 0 \\ d \end{bmatrix} \begin{bmatrix} P \end{bmatrix} \]
  \[ N = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \quad P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \text{given } M, \ \text{what should } Q \text{ be?} \]
  - stay perpendicular
    - substitute from above
      - translations OK: w=0 means unaffected
      - rotations OK
      - uniform scaling OK
  - these all maintain direction

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

Reading for Next Topic: Viewing
- FCG Chapter 7 Viewing