Transforms 1

Christian Miller

CS 354 - Fall 2011
Transformations

\[ \mathbf{v'} = \mathbf{Mv} \]

- What happens if you multiply a square matrix and a vector together?
- You get a different vector with the same number of coordinates
- These are linear transformations
Scaling

\[ S(\alpha) = \begin{bmatrix} \alpha_x & 0 & 0 \\ 0 & \alpha_y & 0 \\ 0 & 0 & \alpha_z \end{bmatrix} \]

- Stretches each of the coordinates by a factor alpha
- Origin stays unchanged
Reflection

- Just scaling with negative values
- Flips over the origin

\[ \alpha = \langle -1, 1 \rangle \quad \alpha = \langle 1, 1 \rangle \]
\[ \alpha = \langle -1, -1 \rangle \quad \alpha = \langle 1, -1 \rangle \]
Rotation

- Rotates CCW around the chosen axis
- Also through the origin
Rotation

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

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

$$R_z(\theta) = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 \\
\sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 1
\end{bmatrix}$$
Shear

- Stretches one dimension at an angle
- Not very useful

\[ H(\theta) = \begin{bmatrix} 1 & \cot \theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \]
Translation

• Everything you can do with 3x3 matrices happens around the origin

• How do we just move something around?
Homogeneous coordinates

- We go up to 4D
- Add a $w$ coordinate to each point / vector
- Add an additional row and column to matrices
- (It’s actually 3D projective space, but uses 4 coordinates)
Homogeneous points and vectors

• Points have $w = 1$
• Vectors have $w = 0$

\[
\begin{align*}
\text{point} &= \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\
\text{vector} &= \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}
\end{align*}
\]
Translation

\[ T(v) = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- Use the extra coordinates to perform translation
- Vectors are unaffected by translation
Homogeneous matrices

\[ S(\alpha) = \begin{bmatrix}
\alpha_x & 0 & 0 & 0 \\
0 & \alpha_y & 0 & 0 \\
0 & 0 & \alpha_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

\[ R_x(\theta) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\theta) & -\sin(\theta) & 0 \\
0 & \sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

- Can take existing transformation matrices, and extend them with the identity to get 4x4 versions
Rotation through a point

- Translate to origin, rotate, translate back
- Multiply the transform matrices together
Inverse shortcuts

• Scale: replace $\alpha$ with $1/\alpha$

• Rotate: inverse is the transpose

• Shear: I don’t even remember

• Translation: replace $\mathbf{v}$ with $-\mathbf{v}$

• Composite transforms: recall $(\mathbf{AB})^{-1} = \mathbf{B}^{-1} \mathbf{A}^{-1}$
General transforms

\[ M = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \]

- Multiplying transforms makes them harder to read
- This is the general form of a 3D affine transform
  - \( R \): rotational / scaling part (3x3)
  - \( t \): translational part (3x1)
Transforming normals

• Normals get distorted when scaled
• Must use the inverse transpose of a transform

\[ \mathbf{M} \rightarrow (\mathbf{M}^T)^{-1} = (\mathbf{M}^{-1})^T \]
Note on coordinates

- We’ll use 4x1 vectors and 4x4 matrices from here on.
- What about that last row? (always 0, 0, 0, 1)
  - Required to keep \( w \) coordinate intact
  - Changes when perspective is brought in
Change of coordinates

• A matrix transforms coordinates from one frame to another
• The final coordinate frame is called the world or spatial frame
• A coordinate frame attached to an object is called a local frame
Change of coordinates

Point in world frame $v'$ = $Mv$
Point in local frame

Positioning of local frame relative to world frame
Change of coordinates

\( \mathbf{v} = <0.6, 0.6> \)

Local frame

World frame
Change of coordinates

Local frame

World frame
**Change of coordinates**

Local frame

World frame
**Change of coordinates**

Local frame

World frame
**Change of coordinates**

Local frame

\[ \mathbf{v} = \langle 0.6, 0.6 \rangle \]

World frame

\[ \mathbf{v}' = \langle 1.1, 2.1 \rangle \]
• A coordinate frame is just a point and 3 vectors

• Write the coordinates of a new frame relative to the world frame into a matrix, and you have a transformation
The inverse of a transform does the opposite

- Converts world frame coordinates to local
Composing transforms

\[ ABCv = Mv \]

- Matrices can be multiplied together
- If you multiply several transform matrices together, the result has the same effect as doing them all separately
Composing Transforms

- Matrix multiplication is non-commutative $AB \neq BA$
- The order in which you apply transforms is crucial
Intuition

• There are two ways to interpret what sequences of transforms do: forwards and backwards

• Both methods yield the same results

• Use whichever one is more intuitive for a given problem

• We’ll use translate / rotate / scale as an example
What we want

- Use translate, rotate, scale to get this result
Forwards

• Multiply new transforms on the right of the previous one
• Every transform is specified in the local coordinate system of the previous one
• Imagine that you’re carrying the local frame along as you stack up transformations
Forwards: $M = T$

- Translate first to target location
Forwards: $M = TR$

- Rotate next, which happens relative to the already translated frame
Forwards: $M = TRS$

- Then scale, which happens with respect to the already translated and rotated frame
Backwards

• Multiply new transforms on the left of the previous one

• Every transform is specified with respect to the world frame, regardless of what has already been done

• Imagine that you’re discarding the local frame and reusing to the world frame after every transform
Backwards: $M = S$

- Scale first
Backwards: M = RS

- Rotate next
Backwards: $M = TRS$

- Finally, translate to the final location
Intuition

• We got the same result \((M = TRS)\) each time

• It’s important to be able to interpret transforms in both ways!

• If you’re having trouble coming up with a transform, try using the other method
Scene graph

- A scene graph is how anything with a position and orientation is stored in graphics software.
- A hierarchical tree of transformations, rooted at the world frame.
- Transforms for objects are multiplied together (on the right) traversing down the tree.
Scene Graph

World

Knight
  - Torso
  - Arm1
  - Arm2
  - Sword
  - Leg1
  - Leg2

Dragon
  - Leg1
  - Leg2
  - Leg3
  - Leg4
  - Damsel
Scene graph

- Rendering implemented with a matrix stack
- Do a depth-first traversal of the tree
  - Enter node: push matrix and apply transform
  - Render object
  - Traverse children
  - Exit node: pop matrix
OpenGL: ordering

• OpenGL uses the forward method

• Calling glRotate, glTranslate, etc. multiplies the new transform onto the right of the old

• E.g.: $M = MX$
OpenGL: matrix stack

- `glPushMatrix()` and `glPopMatrix()` will do the respective operation on the matrix stack
- Objects are rendered using the top matrix
- Push will copy the existing top matrix
- Use to implement scene graphs
OpenGL: matrix funcs

- `glLoadIdentity()`: replaces current $M$ with $I$
- `glLoadMatrix()`: loads a given matrix into $M$
- `glMultMatrix()`: multiplies a given matrix onto the right of $M$
But watch out

- OpenGL stores matrices in column-major order
- Normal C multidimensional arrays are row-major
- You may have to transpose matrices before handing them off to OpenGL
One last thing

- Look up gluLookAt()