Transformations

CS 432/537 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive homogeneous coordinate transformation matrices
- Learn to build arbitrary transformation matrices from simple transformations
A transformation maps points to other points and/or vectors to other vectors

\[ Q = T(P) \]

\[ v = T(u) \]
Affine Transformations

• Line preserving
• Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
• Importance in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints
Pipeline Implementation

T (from application program)

transformation → T(u) → rasterizer → frame buffer

vertices → T(u) vertices → pixels

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

\( \alpha, \beta, \gamma \): scalars

\( p, q, r \): representations of points
  - array of 4 scalars in homogeneous coordinates

\( u, v, w \): representations of vectors
  - 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

d
Translation Using Representations

Using the homogeneous coordinate representation in some frame

\[ \mathbf{p} = [x \ y \ z \ 1]^T \]
\[ \mathbf{p}' = [x' \ y' \ z' \ 1]^T \]
\[ \mathbf{d} = [dx \ dy \ dz \ 0]^T \]

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 p’ = Tp where

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

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

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & d_x \\
0 & 1 & 0 & d_y \\
0 & 0 & 1 & d_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

\[
x' = x + d_x
\]
\[
y' = y + d_y
\]
\[
z' = z + d_z
\]
Rotation (2D)

Consider rotation about the origin by $\theta$ degrees
- radius stays the same, angle increases by $\theta$

\[
x' = r \cdot \cos (\phi + \theta) = r \cdot \cos \phi \cdot \cos \theta - r \cdot \sin \phi \cdot \sin \theta
\]
\[
y' = r \cdot \sin (\phi + \theta) = r \cdot \cos \phi \cdot \sin \theta + r \cdot \sin \phi \cdot \cos \theta
\]

\[
x = r \cos \phi
\]
\[
y = r \sin \phi
\]
\[
x' = x \cos \theta - y \sin \theta
\]
\[
y' = x \sin \theta + y \cos \theta
\]
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 $z$
    \[
    \begin{align*}
    x' &= x \cos \theta - y \sin \theta \\
    y' &= x \sin \theta + y \cos \theta \\
    z' &= z
    \end{align*}
    \]
  - or in homogeneous coordinates
    \[
    p' = R_z(\theta)p
    \]
Rotation Matrix

Rotation Matrix

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

\[x' = x \cos \theta - y \sin \theta\]
\[y' = x \sin \theta + y \cos \theta\]
\[z' = z\]
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

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

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

Rotation Around an Arbitrary Axis

• Rotate a point \( P \) around axis \( \mathbf{n} = (x, y, z) \) by angle \( \theta \)

\[
R = \begin{bmatrix}
tx^2 + c & txy + sz & txz - sy & 0 \\
txy - sz & ty^2 + c & tyz + sx & 0 \\
txz + sy & tyz - sx & tz^2 + c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

• \( c = \cos(\theta) \)
• \( s = \sin(\theta) \)
• \( t = (1 - c) \)

Graphics Gems I, p. 466 & 498
Rotation Around an Arbitrary Axis

• Also can be expressed as the Rodrigues Formula

\[ P_{rot} = P \cos(\vartheta) + (\mathbf{n} \times P) \sin(\vartheta) + \mathbf{n}(\mathbf{n} \cdot P)(1 - \cos(\vartheta)) \]
Scaling

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

\[ x' = s_x x \]
\[ y' = s_y y \]
\[ z' = s_z z \]

\[ p' = Sp \]

\[ S = S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Reflection corresponds to negative scale factors

\[
\begin{align*}
s_x &= -1 \quad s_y = 1 \\
s_x &= -1 \quad s_y = -1 \\
s_x &= 1 \quad s_y = -1
\end{align*}
\]
Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: $T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z)$
  - Rotation: $R^{-1}(\theta) = R(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$
      $R^{-1}(\theta) = R^T(\theta)$
  - Scaling: $S^{-1}(s_x, s_y, s_z) = S(1/s_x, 1/s_y, 1/s_z)$
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 $M = ABCD$ is not significant compared to the cost of computing $Mp$ for many vertices $p$
• The difficult part is how to form a desired transformation from the specifications in the application
• Note that matrix on the right is the first applied
• Mathematically, the following are equivalent
  \[ p' = ABCp = A(B(Cp)) \]  // pre-multiply
• Note many references use column matrices to represent points. In terms of column matrices
  \[ p'^T = p^T C^T B^T A^T \]  // post-multiply
Properties of Transformation Matrices

• Note that matrix multiplication is not commutative
• i.e. in general \( M_1M_2 \neq M_2M_1 \)

- \( T \) – reflection around y axis
- \( T' \) – rotation in the plane
A rotation by $\theta$ about an arbitrary axis can be decomposed into the concatenation of rotations about the $x$, $y$, and $z$ axes

$$R(\theta) = R_z(\theta_z) \ R_y(\theta_y) \ R_x(\theta_x)$$

$\theta_x \ \theta_y \ \theta_z$ 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

\[ M = T(p_f) \cdot R(\theta) \cdot T(-p_f) \]
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

$$\begin{align*}
x' &= x + y \cot \theta \\
y' &= y \\
z' &= z
\end{align*}$$

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

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

• Introduce mat.h and vec.h transformations
  - Model-view
  - Projection
Pre 3.1 OpenGL Matrices

- In OpenGL 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 manipulated by
  - `glMatrixMode(GL_MODELVIEW);`
  - `glMatrixMode(GL_PROJECTION);`
Current Transformation Matrix (CTM)

- Conceptually there was 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 was defined in the user program and loaded into a transformation unit.

\[ p' = C p \]

\[ \text{vertices} \xrightarrow{\text{CTM}} \text{vertices} \]
Rotation about a Fixed Point

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

Result: \( C = TR T^{-1} \) which is backwards.

This result is a consequence of doing postmultiplications. Let’s try again.
Reversing the Order

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

\[
\begin{align*}
C & \leftarrow I \\
C & \leftarrow C T^{-1} \\
C & \leftarrow C R \\
C & \leftarrow C T
\end{align*}
\]

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

Recall \( p' = ABCp = A(B(Cp)) \quad // \text{pre-multiply} \)
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
vec.h and mat.h

Define basic types

• vec2, vec3, vec4
  - 2, 3 & 4 element arrays

• mat2, mat3, mat4
  - 2x2, 3x3 & 4x4 matrices

• Overloaded operators
  - +, -, *, /, +=, -=, *=, /=, []

• Generators
  - RotateX, RotateY, RotateZ, Translate, Scale
  - Ortho, Ortho2D, Frustum, Perspective, LookAt
Rotation, Translation, Scaling

Create an identity matrix:

```cpp
mat4 m = identity();
```

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

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

Do same with translation and scaling:

```cpp
mat4 s = Scale(sx, sy, sz)
mat4 t = Translate(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)

```cpp
mat4 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 columns
• OpenGL functions that have matrices as parameters allow the application to send the matrix or its transpose
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
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 in next lecture
```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();
}
```
Idle and Mouse callbacks

void spinCube()
{
    theta[axis] += 2.0;
    if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
    glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        axis = 0;
    if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
        axis = 1;
    if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        axis = 2;
}
Display callback

We can form matrix in application and send to shader and let shader do the rotation or we 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 and resending the data

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

Using the Model-view Matrix

• In OpenGL the model-view matrix is used to
  - Position the camera
    • Can be done by rotations and translations but is often easier to use 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 often want to use transformations to move and reorient an object smoothly
  - Problem: find a sequence of model-view matrices $M_0, M_1, \ldots, M_n$ so that when they are applied successively to one or more objects we see a smooth transition
• For orienting 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 $R_0, R_1, \ldots, R_n$, find the Euler angles for each and use $R_i = R_{iz} R_{iy} R_{ix}$
    • 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/stable than either
Quaternions

• Extension of imaginary numbers from two to three dimensions
• Requires one real and three imaginary components $i, j, k$

\[ q = q_0 + q_1 i + q_2 j + q_3 k \]

• Quaternions can express rotations on sphere smoothly and efficiently. Process:
  - Model-view matrix $\rightarrow$ quaternion
  - Carry out operations with quaternions
  - Quaternion $\rightarrow$ Model-view matrix
Improved Rotations

• Euler Angles have problems
  - How to interpolate keyframes?
  - Angles aren’t independent
  - Interpolation can create Gimble Lock, i.e. loss of a degree of freedom when axes align

• Solution: Quaternions!
Quaternions

Matrices are not the only (or best) way of representing rotations. For one thing, they are redundant (9 numbers instead of 3) and, for another, they are difficult to interpolate.
An alternative representation was developed by Hamilton in the early 19th century (and forgotten until relatively recently). The quaternion is a 4-tuple of reals with the operations of addition and multiplication defined. Just as complex numbers allow us to multiply and divide two-dimensional vectors, quaternions enable us to multiply and divide four dimensional vectors.

\[ q = q_0 + q_1 i + q_2 j + q_3 k \]
\[ i^2 = j^2 = k^2 = -1 \quad ij = k, \quad jk = i, \quad ki = j \]

A quaternion can also be interpreted as having a scalar part and a vector part. This will give us a more convenient notation.

\[ q = (s, \vec{a}) \quad \text{pure quaternion: } p = (0, \vec{a}) \]

Quaternion addition is just the usual vector addition, the quaternion product is defined as:

\[ q_1 q_2 = (s_1 s_2 - (\vec{a}_1 \cdot \vec{a}_2), s_1 \vec{a}_2 + s_2 \vec{a}_1 + \vec{a}_1 \times \vec{a}_2) \]
Quatertion Facts

conjugate: \( q^* = (s, -\bar{a}) \)

magnitude: \( |q| = \sqrt{qq^*} = \sqrt{s^2 + \bar{a} \cdot \bar{a}} \)

unit quaternion: \( |q| = 1 \)

inverse: \( q^{-1} = \frac{1}{|q|^2} q^* \)

\( q^{-1} = q^* \), for unit quaternions

It turns out that we will be able to represent rotations with a unit quaternion. Before looking at why this is so, there are a few important properties to keep in mind:

- The unit quaternions form a three-dimensional sphere in the 4-dimensional space of quaternions.
- Any quaternion can be interpreted as a rotation simply by normalizing it (dividing it by its length).
- Both \( q \) and \(-q\) represent the same rotation (corresponding to angles of \( q \) and \( 2\pi - q \)
Rotation by Quaternion

\[ R_q(p) = qpq^{-1} \]

\[ p = (0, \bar{x}) \]

\[ q = (\cos(\theta/2), \sin(\theta/2)\bar{a}), \quad \text{where } |\bar{a}| = 1 \]

\[
\begin{align*}
R_q(p) &= (0, (s^2 - \bar{a} \cdot \bar{a})\bar{x} \\
&\quad + 2\bar{a}(\bar{a} \cdot \bar{x}) \\
&\quad + 2s(\bar{a} \times \bar{x}) \quad )
\end{align*}
\]

\[
\begin{align*}
R_q(p) &= (0, (\cos^2(\theta/2) - \sin^2(\theta/2))\bar{x} \\
&\quad + (2\sin^2(\theta/2))\bar{a}(\bar{a} \cdot \bar{x}) \\
&\quad + (2\cos(\theta/2)\sin(\theta/2))(\bar{a} \times \bar{x}) \quad )
\end{align*}
\]

\[
\begin{align*}
R_q(p) &= (0, (\cos \theta)\bar{x} \\
&\quad + (1 - \cos \theta)\bar{a}(\bar{a} \cdot \bar{x}) \\
&\quad + (\sin \theta)(\bar{a} \times \bar{x}) \quad )
\end{align*}
\]

Recognize this? It is the Rodrigues formula!
Quatertion Composition

Since a quaternion basically stores the axis vector and angle of rotation, it is not surprising that we can write the components of a rotation matrix given the quaternion components.

$$q = (\cos(\theta/2), \sin(\theta/2)\vec{a}) = (w, (x, y, z))$$

$$R_q = \begin{pmatrix}
1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy & 0 \\
2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx & 0 \\
2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}$$

Crucially, the composition of two rotations given by quaternions is simply their quaternion product.

$$R_{q'}(R_q(p)) = R_{q''}(p) \quad \text{where } q'' = q'q$$

- Note that the product of two unit quaternions is another unit quaternion.
- Note that quaternion multiplication, like matrix multiplication, is not commutative.
From rotation matrix to quaternion

Given $R = (r_{ij})$, solve expression on previous page for quaternion elements $q_i$

Linear combinations of diagonal elements seem to solve the problem:

\[
q_0^2 = \frac{1}{4}(1 + r_{11} + r_{22} + r_{33})
\]

\[
q_1^2 = \frac{1}{4}(1 + r_{11} - r_{22} - r_{33})
\]

\[
q_2^2 = \frac{1}{4}(1 - r_{11} + r_{22} - r_{33})
\]

\[
q_3^2 = \frac{1}{4}(1 - r_{11} - r_{22} + r_{33})
\]

so take four square roots and you’re done? You have to figure the signs out. There is a better way . . .
Quatertion Interpolation

One of the main motivations for using quaternions in Graphics is the ease with which we can define interpolation between two orientations. Think, for example, about moving a camera smoothly between two views.

\[ \cos \Omega = A \cdot B \]

\[ C(t) = \text{slerp}(A, B, t) = A \frac{\sin(\Omega(1-t))}{\sin \Omega} + B \frac{\sin(\Omega t)}{\sin \Omega} \]

slerp – Spherical linear interpolation

Need to take equals steps on the sphere

A & B are quaternions
What about interpolating multiple keyframes?

• Shoemake suggests using Bezier curves on the sphere
• Offers a variation of the De Casteljau algorithm using slerp and quaternion control points
• See K. Shoemake, “Animating rotation with quaternion curves”, Proc. SIGGRAPH ’85