2D Affine Transformations

All represented as matrix operations on vectors!
Parallel lines preserved, angles/lengths not

- Scale
- Rotate
- Translate
- Reflect
- Shear

2D Transforms: Translation

- Rigid motion of points to new locations:
  \[ x' = x + d_x \]
  \[ y' = y + d_y \]
- Defined with column vectors:
  \[
  \begin{bmatrix}
  x' \\ y'
  \end{bmatrix}
  =
  \begin{bmatrix}
  x \\ y
  \end{bmatrix}
  +
  \begin{bmatrix}
  d_x \\ d_y
  \end{bmatrix}
  \]

as \( P' = P + T \)

2D Transforms: Scale

- Stretching of points along axes:
  \[ x' = s_x \cdot x \]
  \[ y' = s_y \cdot y \]
- Defined with column vectors:
  \[
  \begin{bmatrix}
  x' \\ y'
  \end{bmatrix}
  =
  \begin{bmatrix}
  s_x & 0 \\ 0 & s_y
  \end{bmatrix}
  \begin{bmatrix}
  x \\ y
  \end{bmatrix}
  \]

or just: \( P' = S \cdot P \)
2D Transforms: Rotation

- Rotation of points about the origin
  \[ x' = x \cos \theta - y \sin \theta \]
  \[ y' = x \sin \theta + y \cos \theta \]

Positive Angle: CCW
Negative Angle: CW

Matrix form:
\[
\begin{bmatrix}
x' \\
y'
\end{bmatrix} =
\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
\]
or just: \( P' = R \cdot P \)

Homogeneous Coordinates

- Observe: translation is treated differently from scaling and rotation
- Homogeneous coordinates: allows all transformations to be treated as matrix multiplications

Example: A 2D point \((x, y)\) is the line \((x, y, w)\), where \(w\) is any real number. In 3D homogeneous coordinates, to get the point, homogenize by dividing by \(w\) (i.e. \(w=1\)).

Translation

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

Recall our Affine Transformations

Matrix Representation of 2D Affine Transformations

- Translation: 
  \[
  \begin{bmatrix}
x' \\
y'
\end{bmatrix} =
  \begin{bmatrix}
1 & 0 & d_x \\
0 & 1 & d_y \\
0 & 0 & 1
\end{bmatrix}
  \begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
  \]

- Scale: 
  \[
  \begin{bmatrix}
x' \\
y'
\end{bmatrix} =
  \begin{bmatrix}
s_x & 0 & 0 \\
0 & s_y & 0 \\
0 & 0 & 1
\end{bmatrix}
  \begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
  \]

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

- Shear: 
  \[
  S_{H_x} = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
  \]

- Reflection: 
  \[
  F = \begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{bmatrix}
  \]
Composition of 2D Transforms

• Rotate about a point $P_1$
  – Translate $P_1$ to origin
  – Rotate
  – Translate back to $P_1$

$$P' = T \cdot P$$

Composition of 2D Transforms

• Scale object around point $P_1$
  – $P_1$ to origin
  – Scale
  – Translate back to $P_1$
  – Compose into $T$

$$T(x_1, y_1) \cdot S(S_x, S_y) \cdot T(-x_1, -y_1)$$

Composition of 2D Transforms

• Scale + rotate object around point $P_1$ and move to $P_2$
  – $P_1$ to origin
  – Scale
  – Rotate
  – Translate to $P_2$

$$P' = T \cdot P$$

Composition of 2D Transforms

• Be sure to multiple transformations in proper order!

$$P' = (T \cdot (R \cdot (S \cdot (T \cdot P))))$$

The Window-to-Viewport Transformation

• Problem: Screen windows cannot display the whole world (window management)
• How to transform and clip: Objects to Windows to Screen

Window-to-Viewport Transformation

• Given a window and a viewport, what is the transformation from WCS to VPCS?

Three steps:
• Translate
• Scale
• Translate
Transforming World Coordinates to Viewports

- 3 steps
  1. Translate
  2. Scale
  3. Translate

Overall Transformation:
\[ M_{WV} = T(t_{x_{\text{min}}, t_{y_{\text{min}}}}) \cdot S\left(\frac{x_{\text{max}} - x_{\text{min}}}{y_{\text{max}} - y_{\text{min}}}, \frac{y_{\text{max}} - y_{\text{min}}}{y_{\text{max}} - y_{\text{min}}}\right) \cdot T(-x_{\text{max}}, y_{\text{min}}) \]
\[ P' = M_{WV}P \]

Representation of 3D Transformations

- Z axis represents depth
- Right Handed System
  - When looking “down” at the origin, positive rotation is CCW
- Left Handed System
  - When looking “down”, positive rotation is in CW
  - More natural interpretation for displays, big z means “far”

3D Homogenous Coordinates

- Homogenous coordinates for 2D space requires 3D vectors & matrices
- Homogenous coordinates for 3D space requires 4D vectors & matrices
- \[ S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
- \[ T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

3D Transformations: Scale

- Scale
  - Parameters for each axis direction
  \[ S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

3D Transformations: Translate

- Translation
  \[ T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
3D Transformations:

Rotation

- One rotation for each world coordinate axis

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

\[ P' = R \cdot P \]

Rotation Around an Arbitrary Axis

- Rotate a point \( P \) around axis \( n(x,y,z) \) by angle \( \theta \)

\[ R = \begin{bmatrix}
x^2 + c & txy - zc & txz + yc & 0 \\
txy + zc & y^2 + c & tyz - xc & 0 \\
txz - yc & tyz + xc & z^2 + c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

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

Also can be expressed as the Rodrigues Formula

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

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

Quaternions are not the only (nor best) way of representing rotations. For one thing, they are real-valued 4-vectors instead of 3D 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 real values with the operations of addition and multiplication defined. Just as complex numbers allow us to multiply and divide two-dimensional vectors, quaternions allow us to multiply and divide four-dimensional vectors.

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

\[ q^2 = k^2 = -1 \quad ij = k, \quad jk = i, \quad ki = j \]

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

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

Quaternions addition is the usual vector addition, the quaternion product is defined as:

\[ q_1 q_2 = (s_1, s_2 - (\mathbf{v}_1 \cdot \mathbf{v}_2), s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2) \]

Quaternion Facts

conjugate: \( q^* = (s, -\mathbf{v}) \)

magnitude: \( |q| = \sqrt{s^2 + \mathbf{v} \cdot \mathbf{v}} \)

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

inverse: \( q^{-1} = \frac{1}{s^2 + \mathbf{v} \cdot \mathbf{v}} q^* \)

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 \( 2p - q \))
Rotation by Quaternion

\[ R_q(p) = qpq^{-1} \quad p = (0, x) \]

\( q = (\cos(\phi/2), \sin(\phi/2)\phi) \), where \( \phi = 1 \)

\[ R_q(p) = (0, (a^2 - x^2)^2 + 2a(\alpha - x) + 2x(\phi - x) ) \]

\[ R_q(p) = (0, (\cos^2(\phi/2) - \sin^2(\phi/2))x + (2\sin(\phi/2))\alpha(x - x) + (2\cos(\phi/2)\sin(\phi/2))(\alpha - x) ) \]

\[ R_q(p) = (0, (\cos^2(\phi/2) - \sin^2(\phi/2))x + (1 - \cos(\phi))\alpha(x - x) + (\sin(\phi))(\alpha - x) ) \]

Recognize this? It is the Rodrigues formula!

Quaternion 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.

\[ q = (\cos(\phi/2), \sin(\phi/2)\phi) \]

\[ R_q = \begin{bmatrix}
1 - 2y^2 - 2z^2 & 2yx - 2z\phi & 2zx + 2yw & 0 \\
2yx + 2yw & 1 - 2x^2 - 2z^2 & 2yz - 2xw & 0 \\
2zx - 2yw & 2yz + 2xw & 1 - 2x^2 - 2y^2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

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

\[ R_q(p)R_q(p) = R_q(q) \]

where \( q^2 = 1 \).

- 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 \( q \).

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 . . .

Quatetion Interpolation

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

A & B are quaternions

\[ C(t) = slerp(A, B, t) = A\sin(\Omega(1-t)) \]

\[ \sin \Omega \]

\[ B\sin(\Omega) \]

\[ A\sin(\Omega) \]

\[ B\sin(\Omega) \]

\[ \text{slerp} – \text{Spherical linear interpolation} \]

Need to take equals steps on the sphere

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

3D Transformations: Reflect

- Reflection:

\[ F_x = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

about x-y plane

\[ F_y = \begin{bmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

about y-z plane

\[ F_z = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]
Reflection

corresponds to negative scale factors

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

3D Transformations: Shear

- Shear: \((\text{function of } y)\)

\[
H_s = \begin{pmatrix}
  1 & s_h & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]

\[ x' = x + s_h \cdot y \]

Example: Composition of 3D Transformations

- Goal: Transform \( P_1, P_2 \) and \( P_1, P_3 \)

Example (Cont.)

- Process
  1. Translate \( P_1 \) to \((0,0,0)\)
  2. Rotate about \( y \)
  3. Rotate about \( z \)
  4. Rotate about \( z \)

Final Result

- What we’ve really done is transform the local coordinate system \( R_x, R_y, R_z \) to align with the origin \( x,y,z \)
Example 2: Composition of 3D Transformations

- Airplane defined in x,y,z
- Problem: want to point it in Dir of Flight (DOF) centered at point P
- Note: DOF is a vector
- Process:
  - Rotate plane
  - Move to P

Example 2 (cont.)

- Zp axis to be DOF
- Xp axis to be a horizontal vector perpendicular to DOF
  \[ y \times DOF \]
- Yp vector perpendicular to both Zp and Xp (i.e. \( Z_p \times X_p \))

Transformations to Change Coordinate Systems

- Issue: the world has many different relative frames of reference
- How do we transform among them?
- Example: CAD Assemblies & Animation Models

Coordinate System Example (1)

- Translate the House to the origin
  \[ M_{1\rightarrow 2} = T(x_1, y_1) \]
  \[ M_{2\rightarrow 1} = (M_{1\rightarrow 2})^{-1} \]
  \[ = T(-x_1, -y_1) \]

The matrix \( M_{ij} \) that maps points from coordinate system j to i is the inverse of the matrix \( M_{ji} \) that maps points from coordinate system j to coordinate system i.
World Coordinates and Local Coordinates

• To move the tricycle, we need to know how all of its parts relate to the WCS
• Example: front wheel rotates on the ground wrt the front wheel’s z axis:
  Coordinates of \( P \) in wheel coordinate system:
  \[
P^{(wh)} = \mathbf{T}(0,0,\alpha) \cdot \mathbf{R}_z(\alpha) \cdot P^{(wh)}
  \]

Properties of Transformation Matrices

• Note that matrix multiplication is not commutative
• i.e. in general \( M_1 M_2 \neq M_2 M_1 \)

  \[
  \mathbf{T} \quad \text{reflection around y axis}
  \]
  \[
  \mathbf{T}' \quad \text{rotation in the plane}
  \]

Programming Assignment 6

• Implement parametric form of superellipsoids
• Iterate through \( u \) and \( v \) angular parameters
• Calculate point and normal for each \((u,v)\) pair
• Only calculate one point at each of the poles
• Top and bottom rows should be a triangle fan with poles at center
• Other rows are quads that are broken into triangles
• Output mesh as Open Inventor
  – If shading, output normals