Transformations

- Vectors, bases, and matrices
- Translation, rotation, scaling
- Homogeneous coordinates
- 3D transformations
- 3D rotations
- Transforming normals

Uses of Transformations

- **Modeling transformations**
  - build complex models by positioning simple components
  - transform from object coordinates to world coordinates
- **Viewing transformations**
  - placing the virtual camera in the world
  - i.e. specifying transformation from world coordinates to camera coordinates
- **Animation**
  - vary transformations over time to create motion
Rigid Body Transformations

Rotation angle and line about which to rotate

Non-rigid Body Transformations
General Transformations

\[ Q = T(P) \] for points
\[ V = R(u) \] for vectors

Background Math: Linear Combinations of Vectors

• Given two vectors, A and B, walk any distance you like in the A direction, then walk any distance you like in the B direction
• The set of all the places (vectors) you can get to this way is the set of linear combinations of A and B.
• A set of vectors is said to be linearly independent if none of them is a linear combination of the others.

\[ V = v_1A + v_2B, \ (v_1, v_2) \in \mathbb{R} \]
Bases

• A basis is a linearly independent set of vectors whose combinations will get you anywhere within a space, i.e. span the space

• $n$ vectors are required to span an $n$-dimensional space

• If the basis vectors are normalized and mutually orthogonal the basis is orthonormal

• There are lots of possible bases for a given vector space; there’s nothing special about a particular basis—but our favorite is probably one of these.

Vectors Represented in a Basis

• Every vector has a unique representation in a given basis
  – the multiples of the basis vectors are the vector’s components or coordinates
  – changing the basis changes the components, but not the vector
  – $V = v_1E_1 + v_2E_2 + \ldots + v_nE_n$

The vectors $\{E_1, E_2, \ldots, E_n\}$ are the basis
The scalars $(v_1, v_2, \ldots, v_n)$ are the components of $V$ with respect to that basis
Rotation and Translation of a Basis

A function (or map, or transformation) $F$ is **linear** if

\[ F(A+B) = F(A) + F(B) \]

\[ F(kA) = kF(A) \]

for all vectors $A$ and $B$, and all scalars $k$.

Any linear map is **completely specified** by its effect on a set of basis vectors:

\[
V = v_1E_1 + v_2E_2 + v_3E_3 \\
F(V) = F(v_1E_1 + v_2E_2 + v_3E_3) \\
= F(v_1E_1) + F(v_2E_2) + F(v_3E_3) \\
= v_1F(E_1) + v_2F(E_2) + v_3F(E_3)
\]

A function $F$ is **affine** if it is linear plus a translation

- Thus the 1-D transformation $y = mx + b$ is not linear, but affine
- Similarly for a translation and rotation of a coordinate system
- Affine transformations preserve lines
Transforming a Vector

- The coordinates of the transformed basis vector (in terms of the original basis vectors):

\[ F(E_1) = f_{11}E_1 + f_{21}E_2 + f_{31}E_3 \]
\[ F(E_2) = f_{12}E_1 + f_{22}E_2 + f_{32}E_3 \]
\[ F(E_3) = f_{13}E_1 + f_{23}E_2 + f_{33}E_3 \]

- The transformed general vector \( V \) becomes:

\[ F(V) = \sum v_i F(E_i) = (f_{11}v_1 + f_{12}v_2 + f_{13}v_3)E_1 + (f_{21}v_1 + f_{22}v_2 + f_{23}v_3)E_2 + (f_{31}v_1 + f_{32}v_2 + f_{33}v_3)E_3 \]

and its coordinates (still w.r.t. \( E \)) are

\[ v_1 = (f_{11}v_1 + f_{12}v_2 + f_{13}v_3) \]
\[ v_2 = (f_{21}v_1 + f_{22}v_2 + f_{23}v_3) \]
\[ v_3 = (f_{31}v_1 + f_{32}v_2 + f_{33}v_3) \]

or just \( v = \sum f_{ij}v_j \) The matrix multiplication formula!

Matrices to the Rescue

- An \( n \times n \) matrix \( F \) represents a linear function in \( n \) dimensions
  - \( i \)-th column shows what the function does to the corresponding basis vector
- Transformation = linear combination of columns of \( F \)
  - first component of the input vector scales first column of the matrix
  - accumulate into output vector
  - repeat for each column and component
- Usually compute it a different way:
  - dot row \( i \) with input vector to get component \( i \) of output vector

\[ \begin{align*}
\{ v_1 \} &= \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \{ v_1 \} \\
\{ v_2 \} &= \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \{ v_2 \} \\
\{ v_3 \} &= \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \{ v_3 \}
\end{align*} \]

\[ v_i = \sum f_{ij}v_j \]
### Basic 2D Transformations

**Translate**

\[ x' = x + t_x \]
\[ y' = y + t_y \]

**Scale**

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

**Rotate**

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

Parameters \( t, s, \) and \( \theta \) are the “control knobs”

### Compound Transformations

- **Build compound transformations** by stringing basic ones together, e.g.
  - “translate \( p \) to the origin, rotate, then translate back”
  
  can also be described as a rotation about \( p \)

- Any sequence of linear transformations can be collapsed into a single matrix formed by multiplying the individual matrices together

\[
\begin{align*}
  v_i &= \sum_j f_{ij} \left( \sum_k g_{jk} v_k \right) \\
  m_{ij} &= \sum_k f_{ij} g_{jk}
\end{align*}
\]

- This is good: can apply a whole sequence of transformation at once

---

**Translate to the origin, rotate, then translate back.**
Homogeneous Coordinates

• Translation is not linear—how to represent as a matrix?
• Trick: add extra coordinate to each vector

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & t_x \\
  0 & 1 & t_y \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix}
\]

• This extra coordinate is the *homogeneous* coordinate, or \( w \)
• When extra coordinate is used, vector is said to be represented in *homogeneous coordinates*
• Drop extra coordinate after transformation (project to \( w=1 \))
• We call these matrices *Homogeneous Transformations*

W!? Where did that come from?

• Practical answer:
  – W is a clever algebraic trick.
  – Don’t worry about it too much. The \( w \) value will be 1.0 for the time being.
  – If \( w \) is not 1.0, divide all coordinates by \( w \) to make it so.

• Clever Academic Answer:
  – \((x,y,w)\) coordinates form a 3D *projective space*.
  – All nonzero scalar multiples of \((x,y,1)\) form an equivalence class of points that project to the same 2D Cartesian point \((x,y)\).
  – For 3-D graphics, the 4D projective space point \((x,y,z,w)\) maps to the 3D point \((x,y,z)\) in the same way.
Homogeneous 2D Transformations

The basic 2D transformations become

\[
\begin{align*}
\text{Translate:} & & \text{Scale:} & & \text{Rotate:} \\
\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} & & \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} & & \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{align*}
\]

Any affine transformation can be expressed as a combination of these.
We can combine homogeneous transforms by multiplication.
Now any sequence of translate/scale/rotate operations can be collapsed into a single homogeneous matrix!

3D Transformations

- 3-D transformations are very similar to the 2-D case
- Homogeneous coordinate transforms require 4x4 matrices
- Scaling and translation matrices are simply:
  \[
  S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
  \quad T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}
  \]
- Rotation is a bit more complicated in 3-D
  - left- or right-handedness of coordinate system affects direction of rotation
  - different rotation axes
3-D Coordinate Systems

- Right-handed vs. left-handed

- Z-axis determined from X and Y by cross product: \( Z = X \times Y \)

\[
Z = X \times Y = \begin{bmatrix}
X_2Y_3 - X_3Y_2 \\
X_3Y_1 - X_1Y_3 \\
X_1Y_2 - X_2Y_1
\end{bmatrix}
\]

- Cross product follows right-hand rule in a right-handed coordinate system, and left-hand rule in left-handed system.

Sequences of Transformations

- Often the same transformations are applied to many points

- Calculation time for the matrices and combination is negligible compared to that of transforming the points

- Reduce the sequence to a single matrix, then transform
Collapsing a Chain of Matrices.

- Consider the composite function ABCD, i.e. \( p' = ABCDp \)
- Matrix multiplication isn’t commutative - the order is important
- But matrix multiplication is associative, so can calculate from right to left or left to right: \( ABCD = (((AB) C) D) = (A (B (CD))). \)
- Iteratively replace either the leading or the trailing pair by its product

<table>
<thead>
<tr>
<th>Premultiply</th>
<th>Postmultiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>M ← D</td>
<td>M ← A</td>
</tr>
<tr>
<td>M ← CM</td>
<td>M ← MB</td>
</tr>
<tr>
<td>M ← BM</td>
<td>M ← MC</td>
</tr>
<tr>
<td>M ← AM</td>
<td>M ← MD</td>
</tr>
</tbody>
</table>

both give the same result.

- **Postmultiply:** left-to-right (reverse of function application.)
- **Premultiply:** right-to-left (same as function application.)

Implementing Transformation Sequences

- Calculate the matrices and cumulatively multiply them into a global Current Transformation Matrix
- Postmultiplication is more convenient in hierarchies -- multiplication is computed in the opposite order of function application
- The calculation of the transformation matrix, M,
  - initialize M to the identity
  - in reverse order compute a basic transformation matrix, T
  - post-multiply T into the global matrix M, \( M ← MT \)
- Example - to rotate by \( \theta \) around \([x, y]\):

```c
glMatrixMode(GL_MODELVIEW) /* transform objects in scene */
gLoadIdentity() /* initialize M to identity mat */
glTranslatef(x, y, 0) /* LAST: undo translation */
glRotatef(theta, 0, 0, 1) /* rotate about z axis */
glTranslatef(-x, -y, 0) /* FIRST: move [x,y] to origin */
```

- Remember the last T calculated is the first applied to the points
  - calculate the matrices in reverse order
**Column Vector Convention**

- The convention in the previous slides
  - transformation is by matrix times vector, \( Mv \)
  - textbook uses this convention, 90% of the world too

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} =
\begin{bmatrix}
  m_{11} & m_{12} & m_{13} \\
  m_{21} & m_{22} & m_{23} \\
  m_{31} & m_{32} & m_{33}
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix}
\]

- The composite function \( A(B(C(D(x)))) \) is the matrix-vector product \( ABCDx \)

---

**Beware: Row Vector Convention**

- The transpose is also possible

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} =
\begin{bmatrix}
  m_{11} & m_{12} & m_{13} \\
  m_{21} & m_{22} & m_{23} \\
  m_{31} & m_{32} & m_{33}
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix}
\]

- How does this change things?
  - all transformation matrices must be transposed
  - \( ABCDx \) transposed is \( x'^T C^T B^T A^T \)
  - pre- and post-multiply are reversed
- OpenGL uses transposed matrices!
  - You only notice this if you pass matrices as arguments to OpenGL subroutines, e.g. \( \text{glLoadMatrix} \).
  - Most routines take only scalars or vectors as arguments.
What is a Normal?
Indication of outward facing direction for lighting and shading

Order of definition of vertices in OpenGL

Right hand rule

Note: GL conventions...
glFrontFace(GL_CCW)
glFrontFace(GL_CW)

Transforming Normals

- It’s tempting to think of normal vectors as being like porcupine quills, so they would transform like points
- Alas, it’s not so, consider the 2D affine transformation below.
- We need a different rule to transform normals.
Normals Do Not Transform Like Points

- If M is a 4x4 transformation matrix, then
  - To transform points, use $p' = Mp$, where $p = [x \ y \ z \ 1]^T$
  - So to transform normals, $n' = Mn$, where $n = [a \ b \ c \ 1]^T$
  - Right? Wrong! This formula doesn’t work for general M.

Normals Transform Like Planes

A plane $ax + by + cz + d = 0$ can be written

$$n \cdot p = n^T p = 0, \quad \text{where} \quad n = [a \ b \ c \ d]^T, \quad p = [x \ y \ z \ 1]^T$$

$(a, b, c)$ is the plane normal, $d$ is the offset.

If $p$ is transformed, how should $n$ transform?

To find the answer, do some magic:

$$0 = n^T Ip \quad \text{equation for point on plane in original space}$$

$$= n^T (M^{-1}M)p$$

$$= (n^T M^{-1})(Mp)$$

$$= n'^T p' \quad \text{equation for point on plane in transformed space}$$

$p' = Mp$ to transform point

$n' = (n'M^{-1})' = M^{-T}n$ to transform plane
Transforming Normals - Cases

- For general transformations $M$ that include perspective, use full formula ($M$ inverse transpose), use the right $d$
  $-d$ matters, because parallel planes do not transform to parallel planes in this case
- For affine transformations, $d$ is irrelevant, can use $d=0$.
- For rotations only, $M$ inverse transpose $= M$, can transform normals and points with same formula.

Euler Angles for 3-D Rotations

- Euler angles - 3 rotations about each coordinate axis, however
  - rotations are order-dependent, and there are no conventions about the order to use
  - angle interpolation for animation generates bizarre motions
- Widely used anyway, because they're “simple”
- Coordinate axis rotations (right-handed coordinates):
  
  $$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$  
  $$R_y = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$  
  $$R_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
**Euler Angles for 3-D Rotations**

The matrix $R$ rotates by $\alpha$ about axis (unit) $v$:

$$R = vv^T + \cos \alpha (I - vv^T) + \sin \alpha v^*$$

- $vv^T$ Project onto $v$
- $I - vv^T$ Project onto $v$’s normal plane
- $v^*$ Dual matrix. Project onto normal plane, flip by $90^\circ$
- $\cos \alpha, \sin \alpha$ Rotate by $\alpha$ in normal plane
  (assumes $v$ is unit.)
The Dual Matrix

• If \( \mathbf{v} = [x, y, z] \) is a vector, the skew-symmetric matrix

\[
\mathbf{v}^* = \begin{bmatrix}
0 & -z & y \\
z & 0 & -x \\
-y & x & 0
\end{bmatrix}
\]

is the dual matrix of \( \mathbf{v} \)

• Cross-product as a matrix multiply: \( \mathbf{v}^* \mathbf{a} = \mathbf{v} \times \mathbf{a} \)
  • helps define rotation about an arbitrary axis
  • angular velocity and rotation matrix time derivatives

• Geometric interpretation of \( \mathbf{v}^* \mathbf{a} \)
  • project \( \mathbf{a} \) onto the plane normal to \( \mathbf{v} \)
  • rotate \( \mathbf{a} \) by 90° about \( \mathbf{v} \)
  • resulting vector is perpendicular to \( \mathbf{v} \) and \( \mathbf{a} \)

Quaternions

• Complex numbers can represent 2-D rotations
• Quaternions, a generalization of complex numbers, can represent 3-D rotations
• Quaternions represent 3-D rotations with 4 numbers:
  – 3 give the rotation axis - magnitude is \( \sin \alpha/2 \)
  – 1 gives \( \cos \alpha/2 \)
  – unit magnitude - points on a 4-D unit sphere

• Advantages:
  – no trigonometry required
  – multiplying quaternions gives another rotation (quaternion)
  – rotation matrices can be calculated from them
  – direct rotation (with no matrix)
  – no favored direction or axis
Spatial Deformations

• Linear transformations
  – take any point \((x,y,z)\) to a new point \((x',y',z')\)
  – Non-rigid transformations such as shear are “deformations”

• Linear transformations aren’t the only types!
  • A transformation is any rule for computing \((x',y',z')\) as a function of \((x,y,z)\).

• Nonlinear transformations would enrich our modeling capabilities.
  • Start with a simple object and deform it into a more complex one.

Bendy Twisties

• Method:
  – define a few simple shapes
  – define a few simple non-linear transformations (deformations e.g. bend/twist, taper)
  – make complex objects by applying a sequence of deformations to the basic objects

• Problem:
  – a sequence of non-linear transformations can not be collapsed to a single function
  – every point must be transformed by every transformation
Example: Z-Taper

- **Method:**
  - Align the simple object with the z-axis
  - Apply the non-linear taper (scaling) function to alter its size as some function of the z-position

- **Example:**
  - Applying a linear taper to a cylinder generates a cone

  **“Linear” taper:**
  \[
  x' = (k_1 z + k_2) x \\
  y' = (k_1 z + k_2) y \\
  z' = z
  \]

  **General taper (f is any function you want):**
  \[
  x' = f(z) x \\
  y' = f(z) y \\
  z' = z
  \]
Example: Z-twist

- Method:
  - align simple object with the z-axis
  - rotate the object about the z-axis as a function of z
- Define angle, $\theta$, to be an arbitrary function $f(z)$
- Rotate the points at $z$ by $\theta = f(z)$

“Linear” version: $f(z) = kz$

$$
\theta = f(z), \\
x' = x\cos(\theta) - y\sin(\theta), \\
y' = x\sin(\theta) + y\cos(\theta), \\
z' = z
$$

Extensions

- Incorporating deformations into a modeling system
  - How to handle UI issues?
- “Free-form deformations” for arbitrary warping of space
  - Use a 3-D lattice of control points to define Bezier cubics:
    - $(x',y',z')$ are piecewise cubic functions of $(x,y,z)$
    - Widely used in commercial animation systems
- Physically based deformations
  - Based on material properties
  - Reminiscent of finite element analysis