Geometric Transformations

- Geometric transformations are used to convert from one coordinate system to another.
- From model coordinates to window coordinates
- Transformation are represented as matrices and can be combined by matrix multiplication.

Homogeneous Coordinates

- For a point in three-dimensions, we represent it as a vector:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- For now $w = 1$ for any point.
- We can represent vectors (directions) with $w = 0$.
- Homogeneous coordinates allow us to perform all transformations by matrix multiplication

Coordinate Systems and Frames

- A coordinate system is represented by three independent (usually orthogonal) vectors $\vec{v}_1$, $\vec{v}_2$, $\vec{v}_3$. For example, $(1, 0, 0, 0)$, $(0, 1, 0, 0)$ and $(0, 0, 1, 0)$.
- A coordinate frame is a coordinate system together with a point P that is the origin of the system
- A point in a coordinate frame can be represented as a coordinate $(\alpha_1, \alpha_2, \alpha_3, 1)$, which has "world coordinate" $(\alpha_1, \alpha_2, \alpha_3, 1) \cdot (\vec{v}_1, \vec{v}_2, \vec{v}_3, P)$

Change of Coordinate Frames

- Suppose Frame 1 is represented by $(\vec{v}_1, \vec{v}_2, \vec{v}_3, P)$ and Frame 2 is represented by $(\vec{w}_1, \vec{w}_2, \vec{w}_3, Q)$.
- It is common to need to convert a point $(\alpha_1, \alpha_2, \alpha_3, 1)$ in Frame 1 to a coordinate $(\beta_1, \beta_2, \beta_3, 1)$ in Frame 2.
- First, represent each of the basis vectors in Frame 2 as a combination of the basis vectors in Frame 1:

$$\vec{w}_i = \gamma_{i1}\vec{v}_1 + \gamma_{i2}\vec{v}_2 + \gamma_{i3}\vec{v}_3$$

- Represent Q in Frame 2 as a coordinate in Frame 1:

$$Q = \gamma_{41}\vec{v}_1 + \gamma_{42}\vec{v}_2 + \gamma_{43}\vec{v}_3 + 1 \cdot P$$

- Let M be the matrix of γ_{ij}. Then

$$\begin{bmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \vec{w}_3 \\ Q \end{bmatrix} = M \begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vec{v}_3 \\ P \end{bmatrix}$$

Change of Coordinate Frames

- So if we have a coordinate/direction in Frame 2, we have

$$(\beta_1, \beta_2, \beta_3, \beta_4) \cdot (\vec{w}_1, \vec{w}_2, \vec{w}_3, Q)$$

and hence

$$(\beta_1, \beta_2, \beta_3, \beta_4) \cdot M \begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vec{v}_3 \\ P \end{bmatrix}$$

- Therefore, the coordinates in Frame 1 are

$$(\alpha_1, \alpha_2, \alpha_3, \alpha_4) = (\beta_1, \beta_2, \beta_3, \beta_4) \cdot M$$

- The book gives the same formula but in transposed form.
- Use M^{-1} to convert from Frame 1 to Frame 2.

Model-View-Projection

- Objects are first defined in model coordinates
- Then placed in object/world coordinates with model transformation
- Then placed in eye/camera coordinates with view transformation
- Then placed in clip coordinates with projection transformation
- Perspective division is done to get normalized device coordinates
- Finally window coordinates are computed
- Programmers work with the model frame, object frame, and eye frame

Affine Transformation

- An affine transformation preserves lines (i.e. lines remain lines after transformations)
- It is represented by a matrix

$$M = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Column j of M is where the j-th standard basis gets transformed to. i.e. $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$, $(0, 0, 0, 1)$

Translation

- Translate a point by the vector $(\alpha_x, \alpha_y, \alpha_z, 0)$.
- Translation matrix is

$$T(\alpha_x, \alpha_y, \alpha_z) = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Note: $T^{-1}(\alpha_x, \alpha_y, \alpha_z) = T(-\alpha_x, -\alpha_y, -\alpha_z)$.

Scaling

- Scale a point by factors of β_x, β_y and β_z in the x, y, and z directions (i.e. $(1, 0, 0, 0)$ becomes $(\beta_x, 0, 0, 0)$)
- Assume β_x, β_y, $\beta_z \neq 0$
- Scaling matrix is

$$S(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Note: $S^{-1}(\beta_x, \beta_y, \beta_z) = S(1/\beta_x, 1/\beta_y, 1/\beta_z)$.

Concatenation of Transformations

- You can combine transformations with matrix multiplication.
- e.g. To scale an object and then translate it, you can use

$$T(\alpha_x, \alpha_y, \alpha_z) \cdot S(\beta_x, \beta_y, \beta_z)$$

- Important: matrix multiplication is not commutative. Order of operands matters!
- If we have a sequence of transformation matrices A, B, C, do we compute $(ABC) \cdot p$ or $A \cdot (B \cdot (C \cdot p))$?
- The latter is more efficient for a single point...
- But if we compute ABC first (more work) and then multiply each point in parallel, it is more efficient for many points. The matrix ABC can be passed to the shader.

Rotation

- To define rotation, we need:
  - An axis of rotation: an entire line that is unchanged by the rotation (commonly the x, y, or z-axis)
  - An angle of rotation: typically counterclockwise when looking from the positive axis toward the origin
- Can be generalized to any axis of rotation

Rotation

For rotation about the z-axis, the matrix is

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

Note that $R_z^{-1}(\theta) = R_z(-\theta)$.

You can define similarly $R_x(\theta)$ and $R_y(\theta)$ (see textbook).

Rotation About a Fixed Point

If one wants to perform rotation about a fixed point P other than origin (let's say rotation about z-axis):

1. translate P to origin
2. rotate
3. translate origin back to P

The transformation matrix is $T(P) \cdot R_z(\theta) \cdot T(-P)$.

General Rotation about Origin

- Arbitrary rotations can be specified by three successive rotations about the axes in some (non-unique) order.
- We can rotate first about z-axis, then y, then x.
- $R = R_x(\theta_x)R_y(\theta_y)R_z(\theta_z)$.
- It may be difficult to find these angles.

Rotation about Arbitrary Axis

- Specified by counterclockwise rotation along an axis defined by the vector from P_1 to P_2. Assume center of rotation is origin (translate otherwise).
- Normalize vector $\vec{u} = P_2 - P_1$ to get $\vec{v}$.
- Rotate $\vec{v}$ to positive z-axis through rotations about x and y axes, perform rotation, and then rotate back:

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

Rotation about Arbitrary Axis

- If $\vec{v} = (\alpha_x, \alpha_y, \alpha_z)$, then

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z/d & -\alpha_y/d & 0 \\ 0 & \alpha_y/d & \alpha_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$R_y(\theta_y) = \begin{bmatrix} d & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $d = \sqrt{\alpha_y^2 + \alpha_z^2}$.

Instance Transformation

- It may be useful to define a model of an object type (e.g. box) only once even if there are many boxes of different sizes, orientations, and positions.
- Different instance transformations can be used on the model for different instances.
- Typically scaling is done first to resize object
- Rotation is done next for correct orientation
- Translation is done last to position it
- $M = TRS$ is the transformation from model to object coordinates

Current Transformation Matrix

- Often we like to maintain a current transformation matrix C that is applied to all vertices.
- We can set C to be a particular matrix.
- Most often we modify the CTM by multiplying another matrix on the right.
- e.g.

$$
\begin{aligned}
C &\leftarrow T \\
C &\leftarrow CR \\
C &\leftarrow CS
\end{aligned}
$$

- Note the order of multiplication is reverse of the order of operations on the vertices.

Pipeline Programming

When do we perform transformation:

- perform transformation in application, send transformed vertices through vertex buffer: does not take advantage of modern hardware
- Compute transformation matrix in application and pass to shaders to transform vertices: GPU performs transformation, matrix computed once in application
- Compute transformation matrix and apply to vertices in shader: GPU performs all computations