3-D Geometric Transformations
3-D Viewing Transformation
Projection Transformation

3-D Geometric Transformations

- Move objects in a 3-D scene
- Extension of 2-D Affine Transformations
- Three important ones:
  - Translation
  - Scaling
  - Rotations
**Representing 3-D Points**

- Homogeneous coordinates
- $P (x,y,z) \rightarrow P' (x',y',z')$

\[
\begin{bmatrix}
  x & x' \\
  y & y' \\
  z & z' \\
  1 & 1 \\
\end{bmatrix}
\]

**Homogeneous Translation Matrix**

- Given three translation components $tx$, $ty$, $tz$
- $P' = T \times P$

$T$ is the following 4 X 4 scaling matrix:

\[
\begin{bmatrix}
  1 & 0 & 0 & tx \\
  0 & 1 & 0 & ty \\
  0 & 0 & 1 & tz \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Scaling with respect to origin

- Given three scaling factors \( sx, sy, sz \)
  \[ P' = S \cdot P \]
- \( S \) is the following \( 4 \times 4 \) scaling matrix:
  \[
  S = \begin{bmatrix}
    sx & 0 & 0 & 0 \\
    0 & sy & 0 & 0 \\
    0 & 0 & sz & 0 \\
    0 & 0 & 0 & 1 \\
  \end{bmatrix}
  \]

Rotations

- Need to specify angle of rotation
- And axis about which the rotation is to be performed
- Infinite number of possible rotation axes
  - Rotation about any axis: linear combinations of rotations about x-axis, y-axis, z-axis
Z-Axis Rotation Matrix

\[
Rz = \begin{array}{cccc}
\cos(\theta) & -\sin(\theta) & 0 & 0 \\
\sin(\theta) & \cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

X-Axis Rotation matrix

\[
Rx = \begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & \cos(\theta) & -\sin(\theta) & 0 \\
0 & \sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 0 & 1 \\
\end{array}
\]
Y-Axis Rotation Matrix

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

Rotation Sense

- Positive sense
  - Defined as counter clockwise as we look down the rotation axis toward the origin
Composite 3-D Geometric Transformations

- Series of consecutive transformations
  - Represented by homogeneous transformation matrices $T_1, T_2, \ldots, T_n$
- Equivalent to a single transformation
  - Represented by composite transformation matrix $T$
  - $T$ is given by the matrix product:
    $$T = T_n \cdots T_2 T_1$$
  - First one on the left, last one on the right
- Just like in 2-D, except matrices are $4 \times 4$

Library of 3-D Transformation Functions

- 3-D Transformation Package
- Straightforward Extension of 2-D
- Enables setting up and transforming points & polygons
- 4 X 4 Matrices have 12 non-trivial matrix elements
- Package Might contain the following functions:
3-D Transformation Functions

void settranslate3d(a[12], tx, ty, tz);
void setscale3d(a[12], sx, sy, sz);
void setrotate3d(a[12], theta);
void setrotatey3d(a[12], theta);
void setrotatez3d(a[12], theta);
void combine3d(c[12], a[12], b[12]);  // C = A * B
void xformcoord3d(c[12], vi, *vo);    // vo = C * vi
void xformpoly3d(inpoly[], outpoly[], float c[12]);

- a, b, and c are arrays
  - Contain 12 non-trivial matrix elements of a 4 X 4 homogeneous transformation matrix
- vi and vo are 3-D point structures; inpoly and outpoly are polygons

Rotation about an Arbitrary Axis

- Rotate point P by angle $\alpha$ about a line
- Given: endpoints $P1=(x1,y1,z1)$ & $P2=(x2,y2,z2)$
- Convert problem into rotation about x-axis
  1. Translate so that P1 is at origin: $T1 = T(-x1,-y1,-z1)$
  2. Compute spherical coordinates of the other endpoint:
     $\rho = \sqrt{(x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2}$
     $\phi = \arccos((z2-z1)/\rho)$
     $\theta = \arctan((y2-y1)/(x2-x1))$
3-D Coordinate System Transformations

- There’s a symmetrical relationship between 3-D geometric transformations
  - (moving the object)
  - and 3-D coordinate system transformations
  - (moving the coordinate system)

- For translations, relationship is:
  \[ T_{\text{coord}}(x,y,z) = T_{\text{geom}}(-x,-y,-z) \]

- For each principal-axis, rotation relationship is:
  \[ R_{\text{coord}}(\theta) = R_{\text{geom}}(-\theta) \]

- Useful in deriving 3-D viewing transformation
3D Viewing and Projection

- See CS-460/560 notes on 3-D Viewing and Projection Transformations
  http://www.cs.binghamton.edu/~reckert/460/3dview.htm

3D Viewing/Projection Transformations
- 3-D points in model must be transformed to viewing coordinate system
  – the Viewing Transformation
- Then projected onto a projection plane
  – Projection Transformation
3-D Viewing Transformation

- Converts world coordinates \((x_w, y_w, z_w)\) of a point to viewing coordinates \((x_v, y_v, z_v)\) of the point
  - As seen by a "camera" that is going to "photograph" the scene

\[(x_w, y_w, z_w) \rightarrow (x_v, y_v, z_v)\]

Viewing transformation
Projection Transformation

- Converts viewing coordinates \((x_v,y_v,z_v)\) of a point to 2-D coordinates \((x_p,y_p)\) of that point’s projection onto a projection plane
- Think of projection plane as containing screen upon which the image is to be displayed

\[
\begin{align*}
(x_v,y_v,z_v) & \quad \longrightarrow \quad (x_p,y_p) \\
\text{Projection transformation}
\end{align*}
\]

Viewing Setups

- Specify position/orientation of coordinate systems & projection plane
- Many possible viewing setups
- We’ll use a simple, 4-parameter viewing setup
  - Camera located at origin of viewing coordinate system
  - Somewhat restricted
  - But adequate for most common situations
4-Parameter Viewing Setup

Parameters

- Position of viewpoint (camera location)
  - Position of origin of Viewing Coordinate System (VCS)
  - Specify in spherical coordinates
    - distance $\rho$ from world coordinate system (WCS) origin
    - azimuthal angle $\theta$
    - polar angle $\phi$

- Distance $d$ of Projection Plane from viewpoint
**Viewing Setup Properties**

- VCS $zv$-axis points toward WCS origin
  - So objects we want to be visible must be placed close to WCS origin
- Proj. Plane is perpendicular to $zv$-axis at a distance $d$ from VCS origin
  - So $\rho$ must be greater than $d$
- Center of projection coincides with VCS origin

- VCS’s $yv$-axis is parallel to projection of WCS’s $zw$-axis
  - So WCS $zw$-axis defines “screen up” direction
- VCS’s $xv$-axis is chosen so that $xv$-$yv$-$zv$ axes form a left-handed coordinate system
  - objects far from the VCS’s origin have large $zv$
- 2-D Projection Plane coordinate system’s origin is at intersection of $\rho$ and Projection Plane
  - Its $xp$-$yp$-axes are projections of $xv$-$yv$ axes onto Proj. Plane
    - i.e., $xv$-$yv$ translated a distance $d$ along $zv$ axis
3-D Viewing Transformation

- Must convert xw-yw-zw to xv-yv-zv system
- A coordinate system transformation
- Perform the following steps:
  1. Translate origin by distance $\rho$ in direction ($\theta$, $\phi$)
  2. Rotate by $-(90-\theta)$ degrees about z-axis to bring new y-axis into plane of zw and $\rho$
  3. Rotate by $(180-\phi)$ about x-axis to point transformed z-axis toward origin of world coordinate system
  4. Invert x-axis

Viewing Xform: 1. Translate by $\rho$
2. Rotate by \(-(90-\theta)\) about \(z\)

3. Rotate by \((180-\phi)\) about \(x\)
4. Invert x-axis

1. Translate by \( \rho \)

- Homogeneous transformation matrix for translation by \((x,y,z)\):

\[
T_{\text{geom}} = \begin{bmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- Use relationship between coordinate system transformations & geometric transformations:

\[
T_{\text{coord}}(x,y,z) = T_{\text{geom}}(-x,-y,-z)
\]
So first transformation matrix, $T_1$:

\[
T_1 = \begin{bmatrix}
  1 & 0 & 0 & -x \\
  0 & 1 & 0 & -y \\
  0 & 0 & 1 & -z \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

- Express $x, y, z$ in terms of $\rho, \theta, \phi$ (spherical coordinates):
  - $x = \rho \sin(\phi) \cos(\theta)$
  - $y = \rho \sin(\phi) \sin(\theta)$
  - $z = \rho \cos(\phi)$

2. Rotate by $-(90-\theta)$ about $z$

- Use relationship between coordinate system rotations & geometric rotations:
  
  $T_{\text{coord}}(\alpha) = T_{\text{geom}}(-\alpha)$

- So transformation is $T_2 = R_z(90-\theta)$:

\[
T_2 = \begin{bmatrix}
  \cos(90-\theta) & -\sin(90-\theta) & 0 & 0 \\
  \sin(90-\theta) & \cos(90-\theta) & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]
3. Rotate by \((180-\phi)\) about \(x\)

- Again use relationship between geometric & coordinate system rotations:
  
  So \( T3 = R_x(\phi - 180) \):

  \[
  T3 = \begin{vmatrix}
  1 & 0 & 0 & 0 \\
  0 & \cos(\phi - 180) & -\sin(\phi - 180) & 0 \\
  0 & \sin(\phi - 180) & \cos(\phi - 180) & 0 \\
  0 & 0 & 0 & 1 \\
  \end{vmatrix}
  \]

4. Invert \(x\)-axis

- Result of step 3: \(x\)-axis points opposite from direction it should
  
  - Because WCS is right-handed, while VCS is left-handed

- So need to reflect across \(y''-z''\) plane
  
  - Will convert \(x\) to \(-x\)

  \[
  T4 = \begin{vmatrix}
  -1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1 \\
  \end{vmatrix}
  \]
Composite Viewing Transformation Matrix

- $Tv = T_4 * T_3 * T_2 * T_1$
- Result (after simplification):

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

Projection Transformation

- Look down $xv$ axis at viewing setup:

  Triangles OAP' & OBP are similar
  So set up proportion:
  \[
  \frac{yp}{yv} = \frac{d}{zv}
  \]
  Solve for $yp$:
  \[
  yp = \frac{yv \cdot d}{zv}
  \]

  Look down $yv$ axis for $xp$:
  Result: $xp = \frac{xv \cdot d}{zv}$
Plotting Points on Screen

- Get screen coordinates \((xs,ys)\) from Projection Plane coordinates \((xp,yp)\)
- Final Transformation:
  \[
  \begin{align*}
  (xs,ys) &<--- (xp,yp) \\
  \text{See earlier notes} \\
  &\bullet \text{ Replace } xv,yv \text{ with } xs,ys \\
  &\bullet \text{ Replace } xw,yw \text{ with } xp,yp
  \end{align*}
  \]

Skeleton Pyramid Program: Data Structures

// Build and display a polygon mesh model of a 4-sided pyramid:
struct point3d {float x; float y; float z;};  // a 3d point
struct polygon {int n; int *inds;};            // a polygon
struct point3d w_pts[5];    // 5 world coordinate vertices
struct point3d v_pts[5];     // 5 viewing coordinate vertices
POINT s_pts[5];               // 5 screen coordinate vertices
struct polygon polys[5];    // 5 polygons define the pyramid

// global variables:
int screen_dist; float rho, theta, phi;  // viewing parameters
int xmax,ymax;           // Screen dimensions
int num_vertices=5, num_polygons=5;
Skeleton Pyramid Program:
Function Prototypes

void coeff (float r, float t, float p);  // calculates viewing transformation
   // matrix elements, vii

void convert (float x, float y, float z,
   float *xv, float *yv, float *zv,
   int *xs, int *ys);        // converts a 3D world coordinate point to
   // 3D viewing & 2D screen coordinates
   // i.e., viewing and projection transformations

void build_pyramid (void);   // sets up pyramid points and polygons
   // arrays (see last set of notes)

void draw_polygon (int p);   // draws polygon  p

Skeleton Pyramid Program:
Function Skeletons

// Main Function--Called whenever pyramid is to be displayed
void main_ftn ( )
{
   // Get or set values of rho, theta, phi, and screen_dist here
   build_pyramid (void);   // build polygon model of the pyramid
   coeff (rho,theta,phi);  // compute transformation matrix elements
   for (int i=0; i<num_vertices; i++)
   {
      // Loop to convert polygon vertices from world coordinates
      // to viewing and screen coordinates; must call convert () each time
      for (int f=0; f<num_polygons; f++)
      {
         // Loop to draw each polygon face
         // must call draw_polygon (f) }
   }
}
void coeff (float r, float t, float p)
{ // Code to compute non-trivial viewing transformation matrix

void convert (float x, float y, float z,
       float *xv, float *yv, float *zv, int *xs, int *ys)
{ // Code to compute viewing coordinates and screen coordinates of
  // a point from its 3-D world coordinates. Must implement viewing,
  // projection, and window-to-viewport transformations described
  // in class }

void build_pyramid (void)
{ // Code to define the pyramid by setting up w_pts & polys arrays }

void draw_polygon (int p)
{
  // Code to draw polygon p by:
  // obtaining its vertex numbers from the polys array
  // getting the screen coordinates of each vertex from the s_pts array
  // making appropriate calls to the system  polygon-drawing primitive
}