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) --> P' (x',y',z')

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

Homogeneous Translation Matrix

- Given three translation components tx, ty, tz

  P' = T * P
- T is the following 4 X 4 translation matrix:

$$T = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling with respect to origin

- Given three scaling factors sx, sy, sz

  P' = S * P

- S is the following 4 X 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

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

X-Axis Rotation matrix

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

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 T1, T2, ..., Tn
- Equivalent to a single transformation
  - Represented by composite transformation matrix T
  - T is given by the matrix product:
    T = Tn*...*T2*T1
  - First one on the left, last one on the right
- Just like in 2-D, except matrices are 4 X 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 setrotatex3d(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 X4 homogeneous transformation matrix
- vi and vo are 3-D point structures; inpoly and outpoly are polygons

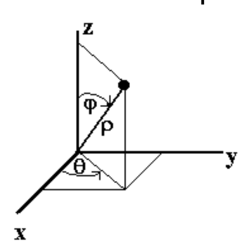
Rotation about an Arbitrary Axis

- Rotate point P by angle α 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:

ρ = sqrt((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2)

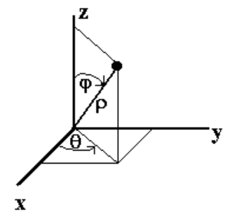
ϕ = arccos((z2-z1)/rho)

θ = arctan((y2-y1)/(x2-x1))



– 3. Rotate about z-axis by -θ so line lies in x-z plane: T2 = Rz(-θ)

– 4. Rotate about y-axis by (90-ϕ) to make line coincide with x-axis: T3 = Ry(90-ϕ)



– 5. Rotate about x-axis by given angle α: T4 = Rx(α)

– 6. Rotate back to undo step 4: T5 = Ry(ϕ-90)

– 7. Rotate back to undo step 3: T6 = Rz(θ)

– 8. Translate back to undo step 1: T7 = T(x1,y1,z1)

- Composite transformation then will be:

  T = T7*T6*T5*T4*T3*T2*T1

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:

  Tcoord(x,y,z) = Tgeom(-x,-y,-z)
- For each principal-axis, rotation relationship is:

  Rcoord(θ) = Rgeom(-θ)
- Useful in deriving 3-D viewing transformation

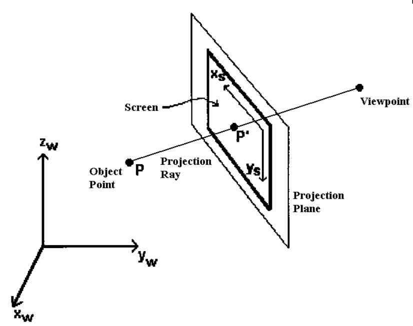
3D Viewing and Projection

- 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



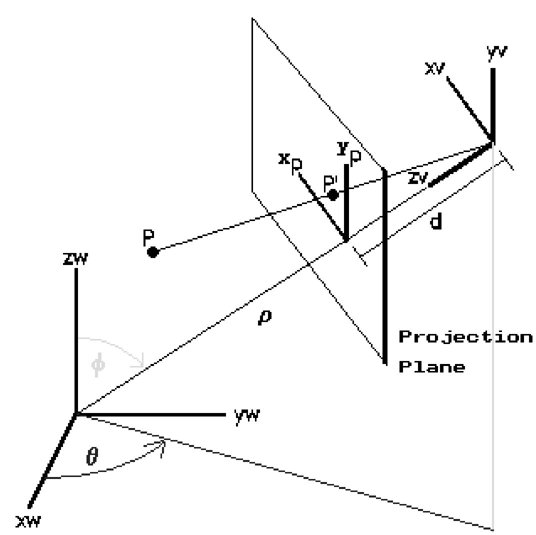
3-D Viewing Transformation

- Converts world coordinates (xw,yw,zy) of a point to viewing coordinates (xv,yv,zv) of the point
  - As seen by a "camera" that is going to "photograph" the scene

(xw,yw,zw) ------------------------> (xv,yv,zv)

Viewing transformation

3-D Viewing Transformation



Projection Transformation

- Converts viewing coordinates (xv,yv,zv) of a point to 2-D coordinates (xp,yp) of that point's projection onto a projection plane
- Think of projection plane as containing screen upon which the image is to be displayed

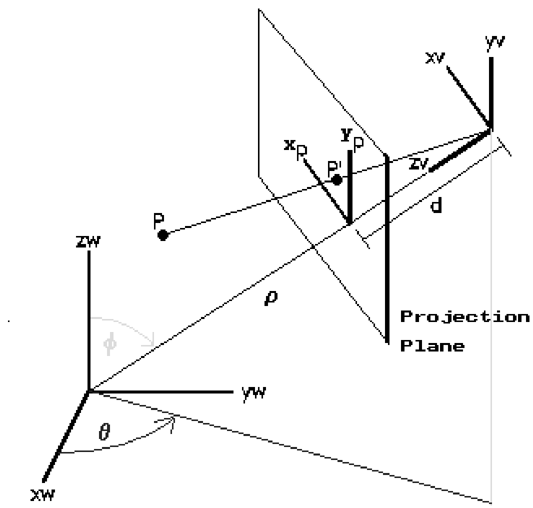
(xv,yv,zv) --------------------------> (xp,yp)

Projection transformation

Viewing Setups

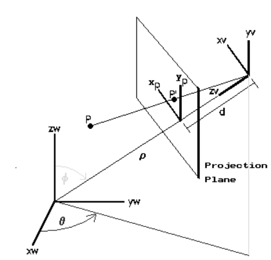
- Specify position/orientation of coordinate systems & projection plane
- Many possible viewing setups
- We'll use a simple, 4-parameter viewing setup
  - Camera located at origin of viewing coordinate system
  - Somewhat restricted
  - But adequate for most common situations

4-Parameter Viewing Setup



Parameters

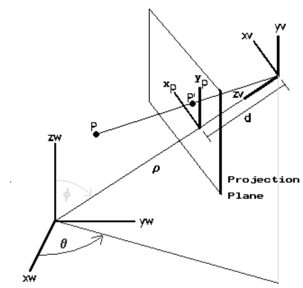
- Position of viewpoint (camera location)
  - Position of origin of Viewing Coordinate System (VCS)
  - Specify in spherical coordinates
    - distance ρ from world coordinate system (WCS) origin
    - azimuthal angle θ
    - polar angle ϕ
- Distance d of Projection Plane from viewpoint



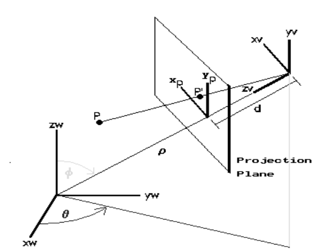
Viewing Setup Properties

- 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 ρ must be greater than d
- Center of projection coincides with VCS origin

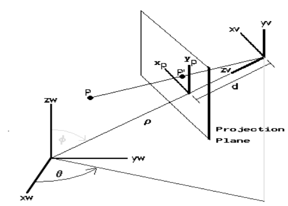


- VCS's yv-axis is parallel to projection of WCS's zw-axis
  - So WCS zw-axis defines "screen up" direction
- VCS's xv-axis is chosen so that xv-yv-zv axes form a <u>left-handed</u> coordinate system
  - objects far from the VCS's origin have large zv
- 2-D Projection Plane coordinate system's origin is at intersection of ρ 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

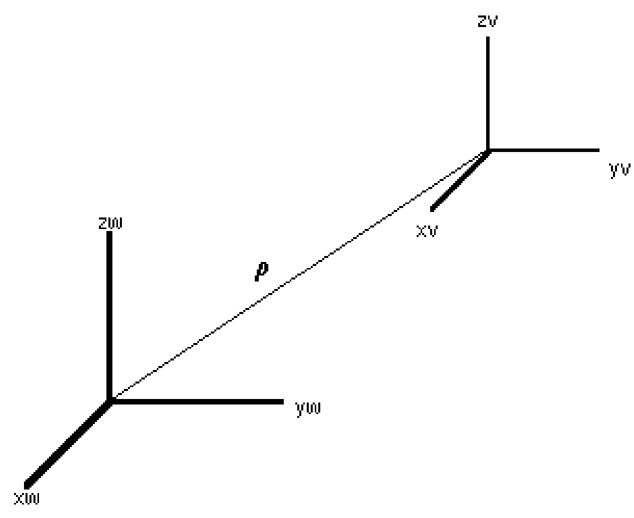


3-D Viewing Transformation

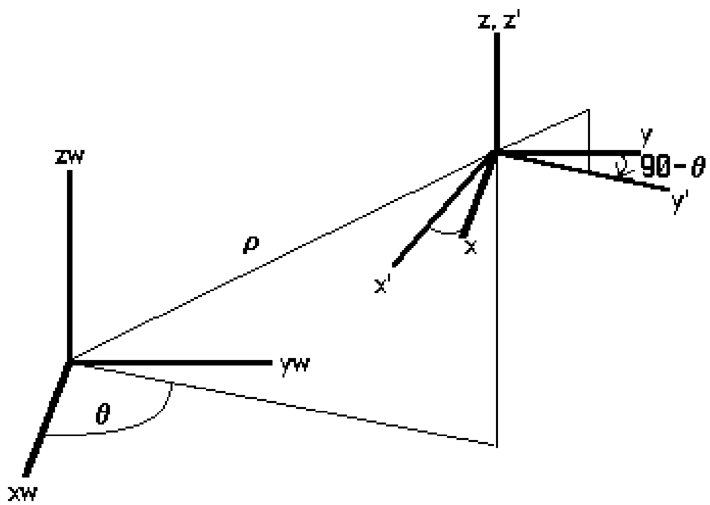
- Must convert xw-yw-zw to xv-yv-zv system
- A coordinate system transformation
- Perform the following steps:
  1. Translate origin by distance ρ in direction (θ, ϕ)
  2. Rotate by -(90-θ) degrees about z-axis to bring new y-axis into plane of zw and ρ
  3. Rotate by (180-ϕ) about x-axis to point transformed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



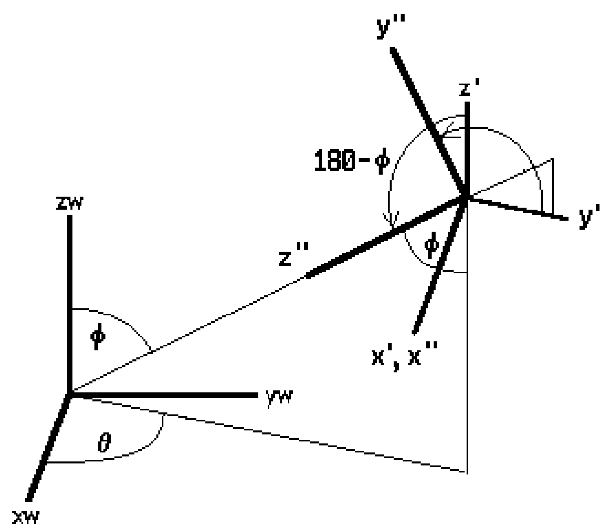
Viewing Xform: 1. Translate by ρ



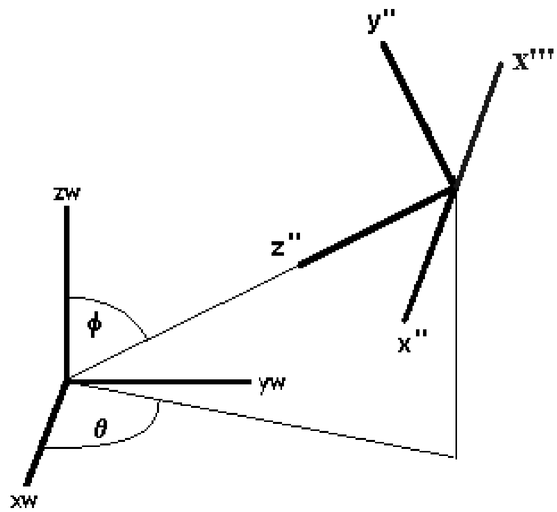
2. Rotate by -(90-θ) about z



3. Rotate by (180-φ) about x



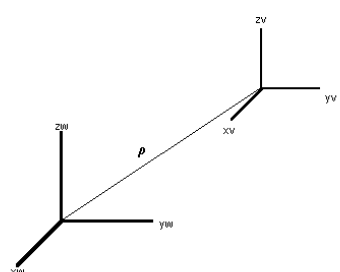
4. Invert x-axis



1. Translate by ρ

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

$$T_{geom} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- Use relationship between coordinate system transformations & geometric transformations:

  Tcoord(x,y,z) = Tgeom(-x,-y,-z)

- So first transformation matrix, T1:

$$T1 = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Express x, y, z in terms of ρ, θ, φ (spherical coordinates)

  x = ρ*sin(φ)*cos(θ)

  y = ρ*sin(φ)*sin(θ)

  z = ρ*cos(φ)

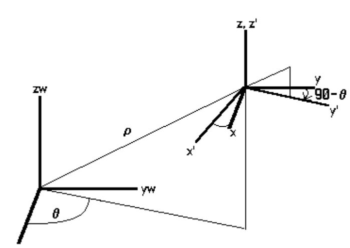
2. Rotate by -(90-θ) about z

- Use relationship between coordinate system rotations & geometric rotations:

  Tcoord(alpha) = Tgeom(-alpha)

- So transformation is T2 = Rz(90-θ):

$$T2 = \begin{bmatrix} \cos(90-\theta) & -\sin(90-\theta) & 0 & 0 \\ \sin(90-\theta) & \cos(90-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

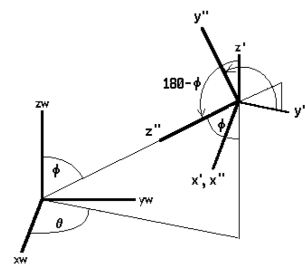


3. Rotate by (180-ϕ) about x

- Again use relationship between geometric & coordinate system rotations:

So $T3 = Rx(\phi - 180)$:

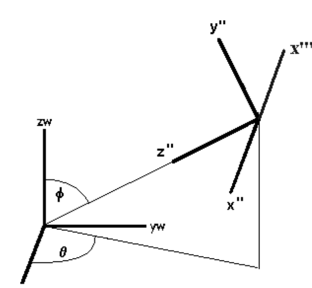
$$T3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi-180) & -\sin(\phi-180) & 0 \\ 0 & \sin(\phi-180) & \cos(\phi-180) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



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

$$T4 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Composite Viewing Transformation Matrix

- Tv = T4*T3*T2*T1
- Important Result (after simplification):

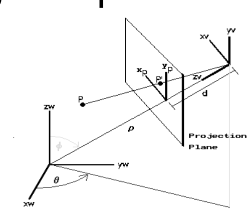
$$
Tv = \begin{bmatrix} -\sin(\theta) & \cos(\theta) & 0 & 0 \\ -\cos(\phi)*\cos(\theta) & -\cos(\phi)*\sin(\theta) & \sin(\phi) & 0 \\ -\sin(\phi)*\cos(\theta) & -\sin(\phi)*\sin(\theta) & -\cos(\phi) & \rho \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Projection Transformation

- Look down xv axis at viewing setup:

Triangles OAP' & OBP are similar

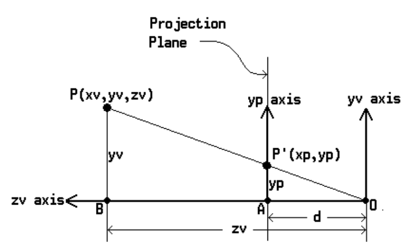
So set up proportion:



$$\frac{yp}{yv} = \frac{d}{zv}$$

Solve for yp:

$yp = (yv*d)/zv$



Look down yv axis for xp:

Result: xp = (xv*d)/zv

Plotting Points on Screen

- Get screen coordinates (xs,ys) from Projection Plane coordinates (xp,yp)
- Final Transformation:

  2D Window-to Viewport Transformation

  (xs,ys) <--- (xp,yp)

  See earlier notes
  - Replace xv,yv with xs,ys
  - Replace xw,yw with xp,yp

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:
float v11,v12,v21,v22,v23,v31,v32,v33,v34; // view xform matrix elements
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, projection , and
                                   // window-to-viewport transformations
void  build_pyramid (void);   // sets up pyramid points and polygons
                              // arrays (see last set of notes)
void  draw_polygon (int poly);   // draws polygon  poly
```

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
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
  // elements: v11,v12,v21,v22,v23,v31,v32,v33,v43 }

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  poly)
{
     // Code to draw polygon poly by:
     // obtaining its vertex index values 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
}
```