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

3-D Viewing Transformation

Converts world coordinates (xw, yw, zw) 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

3-D Viewing Transformation

3-D Viewing Transformation

3-D 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 point's projection onto a projection plane.
- Think of projection plane as containing screen upon which image is to be displayed.

\[
(x_v, y_v, z_v) \rightarrow (x_p, y_p)
\]

Viewing Setups

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

4-Parameter Viewing Setup

- Position of viewpoint (camera location).
  - Position of origin of Viewing Coordinate System (VCS).
  - Specify in spherical coordinates:
    - distance \(d\) from world coordinate system (WCS) origin.
    - azimuthal angle \(\theta\).
    - polar angle \(\phi\).
- Distance \(d\) of projection plane (PP) from viewpoint.

Parameters

- VCS \(z_v\)-axis points toward WCS origin.
  - So objects we want to be visible must be placed close to WCS origin.
- PP is perpendicular to the \(z_v\)-axis at a distance \(d\) from VCS origin.
  - So \(d\) must be greater than \(d\).
- Center of projection coincides with VCS origin.

Viewing Setup Properties

- VCS \(z_v\)-axis is parallel to projection of WCS\(z_w\)-axis.
  - So WCS \(z_w\)-axis defines "screen up" direction.
- VCS\(x_v\)-axis is chosen so that \(x_v-y_v-z_v\) axes form a left-handed coordinate system.
  - Objects far from the VCS's origin have large \(z_v\).
- 2-D Projection Plane coordinate system's origin is at intersection of \(z_v\) and PP.
  - Its \(x_p-y_p\)-axes are projections of \(x_v-y_v\) axes onto PP.
    - i.e., \(x_v-y_v\) translated a distance \(d\) along \(z_v\) axis.
### 3-D Viewing Transformation

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

- 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:
  $T_{coord}(x, y, z) = T_{geom}(-x, -y, -z)$

### Rotate by $-90-\theta$ about $z$

### 3. Rotate by $180-\phi$ about $x$

### 4. Invert $x$-axis
1. So first transformation matrix, T1:
\[
T_1 = \begin{bmatrix}
1 & 0 & 0 & -x \\
0 & 1 & 0 & -y \\
0 & 0 & 1 & -z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Express x, y, z in terms of ?, ?, ?
\[
x = ? \sin(?) \cos(?)
\]
\[
y = ? \sin(?) \sin(?)
\]
\[
z = ? \cos(?)
\]

2. Rotate by -(90-?) about z
- Use relationship between coordinate system rotations & geometric rotations:
  \[T_{\text{coord}}(\alpha) = T_{\text{geom}}(-\alpha)\]
- So transformation is \(T_2 = R_z(90-)\):
\[
T_2 = \begin{bmatrix}
\cos(90-) & -\sin(90-) & 0 & 0 \\
\sin(90-) & \cos(90-) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

3. Rotate by (180-?) about x
- Again use relationship between geometric & coordinate system rotations:
  So \(T_3 = R_x(-180)\):
\[
T_3 = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(-180) & -\sin(-180) & 0 \\
0 & \sin(-180) & \cos(-180) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

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
\[
T_4 = \begin{bmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Composite Viewing Transformation Matrix
\[
T_v = T_4 \cdot T_3 \cdot T_2 \cdot T_1
\]
Result (after simplification):
\[
T_v = \begin{bmatrix}
-\sin(?) & \cos(?) & 0 & 0 \\
-\cos(?) \cos(?) - \sin(?) \sin(?) & \cos(?) & \sin(?) & 0 \\
-\sin(?) \cos(?) - \sin(?) \sin(?) & -\cos(?) & \cos(?) & ? \\
1 & 0 & 0 & 1
\end{bmatrix}
\]

Projection Transformation
- Look from origin down xv axis at viewing setup:
  Triangles OAP' & OBP are similar
  - Set up proportion:
\[
\frac{y_p}{y_v} = \frac{d}{z_v}
\]
  - Solve for \(y_p\):
\[
y_p = (y_v \cdot d)/z_v
\]
  - Look down yv axis for \(x_p\):
Result: \(x_p = (xv \cdot d)/zv\)
Plotting Points on Screen
- Get screen coordinates \((xs,ys)\) from Projection Plane coordinates \((xp,yp)\)
- Final Transformation:
  2D Window-to-Viewport Transformation
  \((xs,ys) \leftarrow (xp,yp)\)
  See earlier notes
  - Replace \(xv, yv\) with \(xs,ys\)
  - Replace \(xw, yw\) with \(xp,yp\)

Simple Hidden Surface Removal (Back-Face Culling)
- For complex objects there are lots of polygons
- Many polygons not visible
  - Because they face away from observer
- More realistic, less complex image is produced if only visible polygons are drawn
  - So draw only those facing toward observer
- Technique of back-face culling determines if polygon is visible or not

Back-Face Culling
- Define one side of each polygon to be the visible side
  - That side is the outward-facing side
- Defining each polygon in the polygons array:
  - Systematically number vertices in counter-clockwise fashion as seen from outside of the object

First: Review of Vector Products
- Dot (Scalar) Product
  \[ s = A \cdot B = |A| \cdot |B| \cdot \cos(\theta) \]
  \(\theta\) is the angle between vectors \(A\) and \(B\)
  In terms of components (RH coord system):
  \[ s = A_xB_x + A_yB_y + A_zB_z \]

In the following diagram:
- \(V = A \times B, \) a vector
- Magnitude: \(|V| = |A| \cdot |B| \cdot \sin(\theta)\)
  \(\theta\) is angle between vectors \(A\) and \(B\)
- Direction: Given by right-hand rule
  - 1. Align fingers of right hand with first vector
  - 2. Rotate toward second
  - 3. Thumb points in direction of \(V\)

Cross (Vector) Product
- \(V = A \times B, \) a vector
- Magnitude: \(|V| = |A| \cdot |B| \cdot \sin(\theta)\)
  \(\theta\) is angle between vectors \(A\) and \(B\)
- Direction: Given by right-hand rule
  - 1. Align fingers of right hand with first vector
  - 2. Rotate toward second
  - 3. Thumb points in direction of \(V\)

In the following diagram:
- \(V = A \times B, \) would point out of the screen toward the observer

In terms of components (RH coordinate system):
\[ \begin{vmatrix} i & j & k \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix} \]
\(i, j, k\) are unit vectors along \(x,y,z\) axes
**Triple Product**

\[ A \cdot (B \times C) = \begin{vmatrix} \ Ax & \ Ay & \ Az \\ \ Bx & \ By & \ Bz \\ \ Cx & \ Cy & \ Cz \end{vmatrix} \]

(Components in terms of RH coord system)

---

**Back-Face Culling**

Consider triangle with vertices 0, 1, 2
- Visible side of the triangle: 0, 1, 2
  - Vertices numbered in counter-clockwise order
  - Invisible side is: 0, 2, 1
    - (clockwise vertex ordering)

[Diagram of a triangle with vertices 0, 1, 2]

---

**Define vector N**
- Outward normal to triangle

**Define Vector V0**
- Vector from observer to vertex 0

**Some Cases:**
- N and V0 nearly parallel (V0 \cdot N = 1)
- Visible side of triangle 0, 1, 2 invisible to viewer

Rotate triangle about side 01 by 90 degrees
- Now N and V0 are perpendicular (V0 \cdot N = 0)
- Triangle is about to become visible
- At all other points between these two orientations:
  - V0 \cdot N is positive
  - Triangle is invisible to viewer

[Diagram showing viewpoint, V0, N, and rotation]

---

**Continue rotation about side 01**
- Triangle becomes visible to the viewer
- 90 degrees more, N and V0 are antiparallel
  - V0 \cdot N = -1
  - Triangle facing toward viewer and is visible
    - At all intermediate orientations:
      - Triangle is visible
      - And V0 \cdot N is negative

[Diagram showing viewpoint, V0, N, and rotation]

---

**Criterion for Invisibility**
- If V0 \cdot N > 0, triangle 012 is invisible
- Now place triangle 012 in an arbitrary position relative to viewer V
Outward normal $N$ is vector (cross) product of $V01$ and $V02$
$V01$ is vector from vertex 0 to vertex 1
$V02$ is vector from vertex 0 to vertex 2

So: $N = V01 \times V02$

Criterion for invisibility:
$V0 \cdot (V01 \times V02) > 0$

But:
$V01 = V1 - V0$
$V02 = V2 - V0$

Substituting we get:
$V0 \cdot [(V1 - V0) \times (V2 - V0)] > 0$, invisibility

Expanding:
$V0 \cdot (V1 \times V2) - V0 \cdot (V1 \times V0) - V0 \cdot (V0 \times V2) + V0 \cdot (V0 \times V0) > 0$

Last Term = 0
(Cross product of any vector with itself = 0)

Middle two terms:
Quantity inside ( ) is a vector perpendicular to $V0$
So dot product of either vector with $V0$ is 0

Final Criterion for Invisibility
$|X0 \ Y0 \ Z0|$
$|X1 \ Y1 \ Z1| < 0$
$|X2 \ Y2 \ Z2|$

Result can be applied to any planar polygon
Use viewing coordinates of three consecutive polygon vertices
Could implement as a "visibility" function
- Computes and returns value of determinant
  Positive means visible, negative invisible

Skeleton Pyramid Program:
Data Structures
// Build/display polygon mesh model of 4-sided pyramid:
struct point3d {float x; float y; float z;}; // a 3d point
struct polygon {int n; int *inds;};
struct point3d w_pts[5]; // 5 world coord. vertices
struct point3d v_pts[5]; // 5 viewing coord. vertices
POINT s_pts[5]; // 5 screen coord. vertices
struct polygonpolys[5]; // 5 polygons define pyramid

// global variables:
float v11,v12,v21,v22,v23,v31,v32,v33,v34; // view xform matrix els
int screen_dist; float rh, 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 xform matrix elements
void convert (float x, float y, float z,
float *xv, float *yv, float *zv,
int *xs, int *ys); // converts a 3D world coordinate pt to
// 3D viewing & 2D screen coords
void build_pyramid (void); // sets up pyr. points and polygons arrays
void draw_polygon (int p); // draws polygon p if polygon is visible
float visible(int p); // returns a negative # if polygon p is visible
// or back-face culling is turned off
Skeleton Pyramid Program: Function Skeletons

// Main Function--Called when pyramid is to be displayed
// hide_flag determines if backface culling is to be done
void main_ftn(int hide_flag)
{
    // Set values of rho, theta, phi, and screen_dist here
    build_pyramid();  // 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();
for (int f=0; f<num_polygons; f++)
    // Loop to draw each polygon if hide_flag is off or if polygon
    // is visible; must call visible() and draw_polygon();
}

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,
    // perspective, and window-to-viewport transformations described
    // in class }

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

float visible(int p)
{  // Code to compute and return value of visibility determinant for
    // polygon p. Negative means invisible, positive visible. }

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