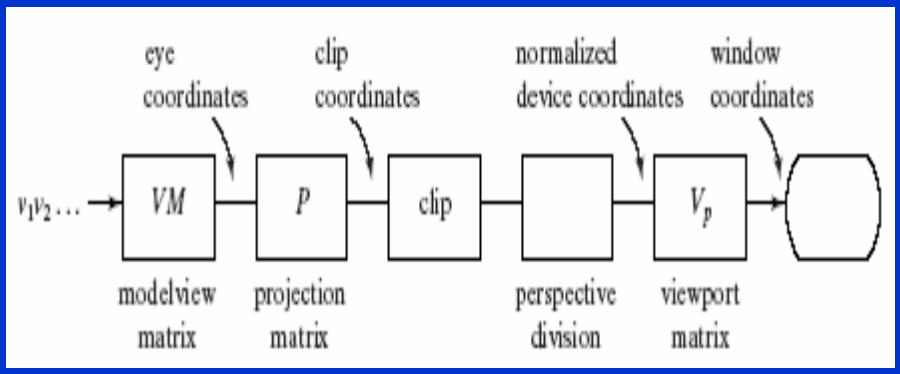
Prof. Reuven Aviv
Department of Computer Science
Tel Hai Academic College
Computer Graphics

Three Dimensional Viewing Part II

Slides adapted from F. Hill, J. Kelley, Computer Graphics

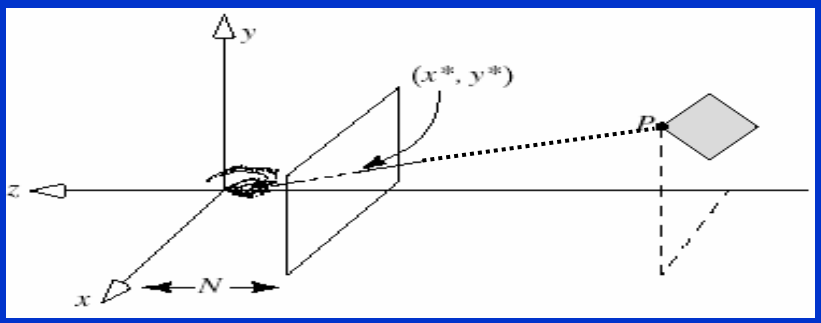
Perspective Projection

Graphic Pipeline



Perspective Projections of 3-D Point

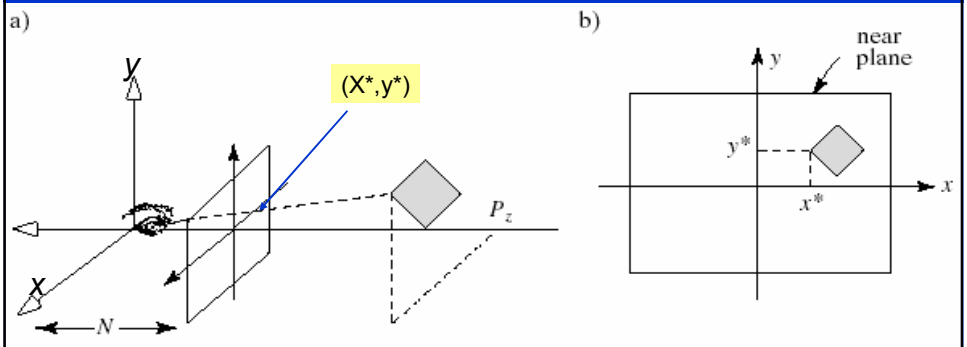
- **A vertex located at *P* in eye coordinates is projected to a certain point (*x**, *y**, z*) on the View plane**



This is a non-affine transformation!

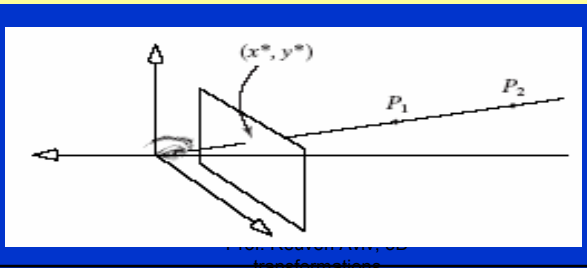
Perspective Projections of Point

- (P_x, P_y, P_z) projects to $P^* \equiv (x^*, y^*, z^*)$
- $x^*/P_x = N/(-P_z)$, $y^*/P_y = N/(-P_z)$ (similar triangles)
- $P^* = (x^*, y^*) = N/(-P_z)\ (P_x, P_y)$
- *What's the value of z*?*



What's the value of z*?

- Lost info: P1 closer than P2 to View Plane. *So what?*
- Define pseudo depth, z*, increasing with -Pz
  - $z^*(Pz) = (aPz + b)/(-Pz)$
  - So that $z^* = -1$ for $P_z = -N$ $\quad z^* = 1$ for $P_z = -F$
- define *the 3-D projection point* of (Px, Py, Pz)
  - $(x^*, y^*, z^*) = [1/(-P_z)][NP_x, NP_y, (a + bP_z)]$
- Now we want to write the transformation via a matrix!



homogeneous coordinates representation (1)

- The homogeneous representation of a Point **P** is given by *a family* of 4 components columns
- with arbitrary w, provided w ≠ 0
- All w ≠ 0 define same point

$$P = \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix}$$

- Before we display a point, we divide all 4 coordinates by w.
- Affine transformations do not change w; *why?*
- last row of an affine matrix is (0, 0, 0, 1).

Homogeneous coordinates representation (2)

- To convert a point from *ordinary coordinates* to *homogeneous coordinates*, append a 1.
- To convert a point from *homogeneous coordinates* to *ordinary coordinates*, divide all components by the last component and discard the fourth component.

perspective transformation and division

- Operate on a point by the non-affine perspective transformation matrix with the last row (0, 0, -1, 0)

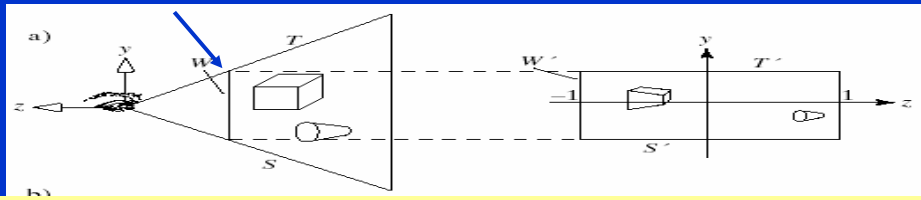
$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wNP_x \\ wNP_y \\ w(aP_z + b) \\ -wP_z \end{pmatrix}$$

- Divide by the fourth coordinate
  - $(P_x, P_y, P_z) \rightarrow -(1/P_z)(NP_x, NP_y, aPz+b) \equiv (x^*, y^*, z^*)$
  - This operation is called the perspective division

Perspective projection

- perspective transformation + perspective division transform 3D point to 3D point
  - $(P_x, P_y, P_z) \rightarrow -(1/P_z)(NP_x, NP_y, aP_z+b)$
  - The third component is used for depth testing
  - first 2 components used for mapping to viewport
- Projection is discarding the third dimension
  - Also called orthographic (or trivial) projection
- (perspective projection) = (perspective transformation) + (perspective division) + (orthographic/trivial projection)

The transformed view volume



- Top, bottom & side walls map to planes parallel to *z*-axis.
- Near, Far planes map to planes at $z = -1, +1$.
- view volume is now a rectangular parallelepiped
- Left as an exercise
- *Can we simplify clipping by transforming the view volume once more?*

Facilitating Clipping: Canonical View volume

- *CVV*, a cube bounded by -1 1 in each dimension
- Translate by –(*right*+*left*)/2 in x, –(*top*+*bott*)/2 in y
- Scale by 2/(*right* – *left*) in x, 2/(*top* – *bott*) in y
- The combined perspective transformation and this scaling is the *Projection Matrix*
- The distortion (due to uneven scaling) will be eliminated in the final viewport transformation

The Projection Matrix

$$R = \begin{pmatrix} \frac{2N}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

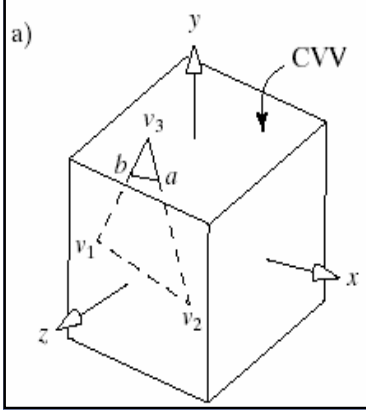
Applying Projection Matrix in OpenGL

- *glMatrixMode*(GL_Projection);
- *glLoadIdentity();* // start with a unit matrix
- *glFrustum*(Left, Right, bott, top, N, F)
- Or *gluPerspective*(viewAngle, aspect, Near, Far)
  - OpenGL calculates from the arguments:
  - *top* = N*tan((π/180)*viewAngle) *bott* = -*top*
  - *right* = *top***aspect* *left* = -*right*

Clipping against the Canonical Volume

Clipper Against the View Volume: Example

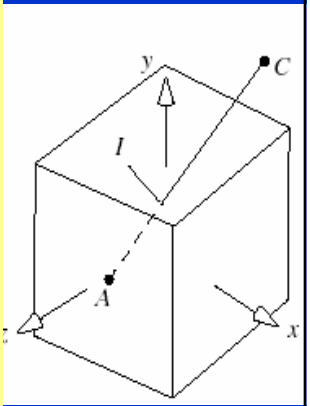
- **A triangle has vertices v_1, v_2,v_3.**
- **v_3 is outside *CVV***



- first clips edge v_1v_2, finding the entire edge is inside *CVV*.
- Then clips edge v_2v_3, records the new vertex **a**
- Finally clips edge v_3v_1 and records the new vertex **b**

Clipping in homogeneous coordinates

- Example: clipping segment **AC**
- points in homogeneous coordinates
- **A** = (a_x, a_y, a_z, a_w), **C** = (c_x, c_y, c_z, c_w)
- If **A**, **C** lie on opposite sides of a wall we need to compute intersection
  - **I** = (I_x, I_y, I_z, I_w).
- *When intersection calculation is not required?*
- If both **A**, **C** on same side of a wall



The Inside /Outside Test of a point

- A point **P** = (x, y, z, w); True coordinates (x/w, y/w, z/w)
- We test whether P is inside the CVV
- *When P lies to the right of X= -1 plane?* (inside)
- if x/w > -1 → w + x > 0.
- *When P lies to the left of plane X = 1?* (inside)
- if x/w < 1 → w - x > 0.
- The 6 quantities w ± x, w ± y, w ± z are the "Boundary Coordinates" of point **P**
  - If all BCi >0, point is inside CVV; else outside

Table: Boundary Coordinates & Clip Planes

boundary coordinate	*homogeneous value*	*clip plane*
BC_0	$w + x$	$x = -1$
BC_1	$w - x$	$x = 1$
BC_2	$w + y$	$y = -1$
BC_3	$w - y$	$y = 1$
BC_4	$w + z$	$z = -1$
BC_5	$w - z$	$z = 1$

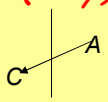
- If all BCi > 0, point is inside CVV
- Else, a BCi < 0, the point is outside CVV

Clip a line segment

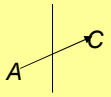
- *What are the condition for trivial decisions?*
- Trivial accept: both endpoints inside the CVV (all BCi >0)
- Trivial reject: both endpoints lie outside same plane of CVV
- Else Algorithm Similar to Cyrus-Beck clipper
  - Line **P**(t) = **A** + (**C**-**A**)t 0 <= t <= 1
  - Jump from wall to wall: intersect line with wall
  - Maintain a Candidate Interval (*CI*) of t within which the segment might still be inside the *CVV*.

Enter/Exit test and Intersection

- segment **AC** (from **A** to **C**) ;
- $\boldsymbol{P}(t) = (a_x+(c_x-a_x)t,\ a_y+(c_y-a_y)t,\ a_z+(c_z-a_z)t,\ a_w+(c_w-a_w)t)$
- Intersection relation to wall 1, X = 1
  - Denote: $BC_1(\mathbf{A}) = a_w - a_x$ $BC_1(\mathbf{C}) = c_w - c_x$ (see table)
  - If $BC_1(\mathbf{A}) < 0$ then **A** is outside wall 1, line enters *(why)*
  - If $BC_1(\mathbf{C}) < 0$ then **C** is outside wall 1, line exits



- Intersection is calculated in both these cases (only)
  - $[a_x + (c_x - a_x)t]/[a_w + (c_w - a_w)t] = 1$
  - $t_{hit} = [a_w - a_x]/[(a_w - a_x) - (c_w - c_x)] =$
  - $t_{hit} = BC_1(A)/[BC_1(A) - BC_1(C)]$

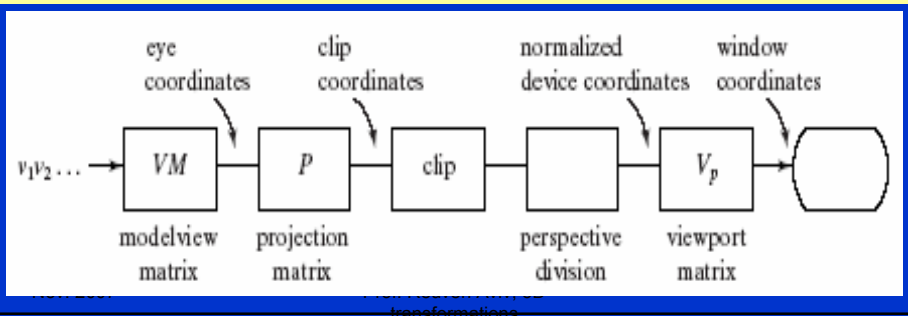


Clip against CVV: Liang Barski Algorithm

- $CI = [t_{in} = 0.0;\ t_{out} = 1.0]$
- We test the line segment against each wall i in turn.
- If BCi(A), BCi(C) have opposite signs, find t_{hit}
  - If segment is entering update $t_{in} = max(old\ t_{in},\ t_{hit})$
  - if segment is exiting, update $t_{out} = min(old\ t_{out},\ t_{hit})$
- If, at any time the *CI* is reduced to the empty interval ($t_{out} > t_{in}$), the entire segment is clipped off
- an "early out", of the algorithm.

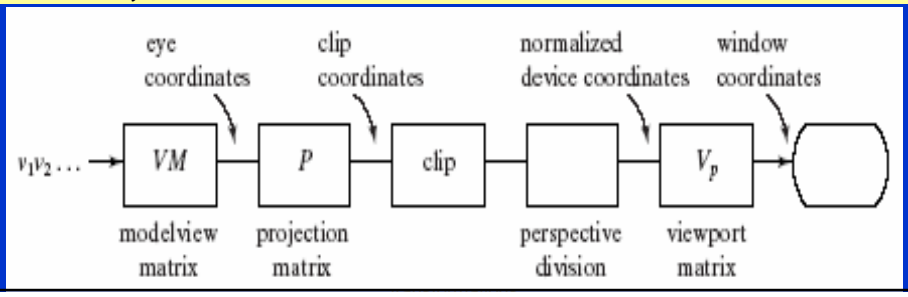
Steps in the path of a vertex P through the Pipeline

- ***P*** extended to a homogeneous 4-tuple point by appending 1
- **P** multiplied by the *modelview matrix*, producing its eye coordinates, then
- Multiplied by the *projection matrix*, producing its clip coordinates, then
- Clipping against CVV



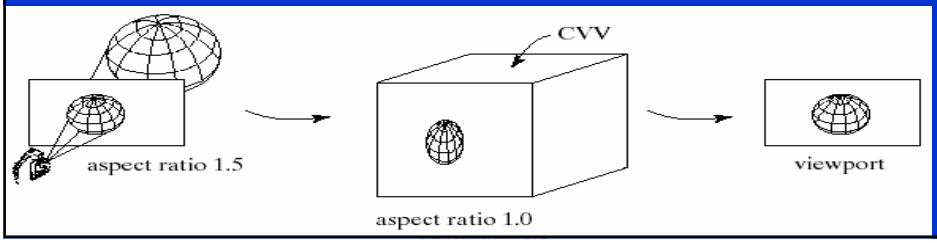
Steps in the path of a vertex P through the Pipeline

- Perspective division is performed, returning a 3-tuple point in Normalized Device Coordinates, then
- Point multiplied by viewport matrix; result (s_x, s_y, d_z)
- result used for depth calculations and drawing:
  - d_z is a measure relative of the depth of the original point
  - (s_x, s_y) is the point in screen coordinates to be displayed



Distortion and its removal

- *Projection matrix* transformed view volume to CVV
- If aspect ratio of the original view volume is not 1, say 1.5, there is distortion
- viewport transformation can undo this *how?*
- Map window in view plane to viewport with original aspect ratio (e.g. 1.5)



Distortion and Its Removal in OpenGL

- *glViewport(x, y, wid, ht)* specifies viewport
  - lower left corner (x,y) in screen coordinates
  - wid pixels wide and ht pixels high.
  - aspect ratio wid/ht
- User usually specifies these so that original aspect ratio (specified in *gluPerspective*(viewAngle, aspect, Near, Far) is kept
- Note: *glViewport()* also maps pseudodepth from the range -1 to 1 into the range 0 to 1.