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

Three Dimensional Viewing

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

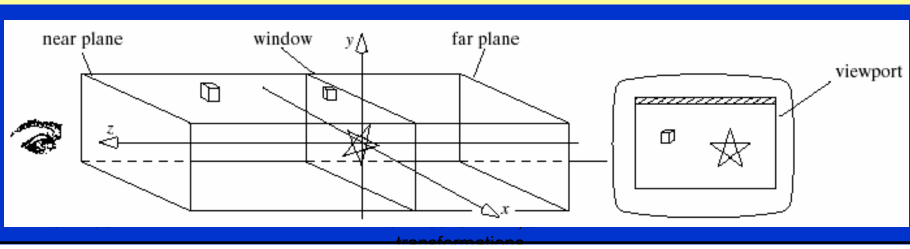
Contents

- **Orthogonal projection**
- **3D perspective Drawing**
  - **camera position and orientation**
  - **perspective projection**
  - **clipping**

Orthogonal projection

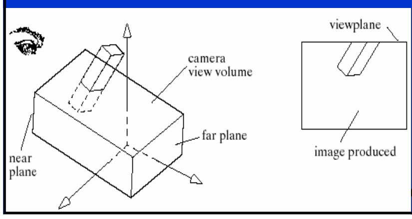
Eye and View Volume in orthogonal projection

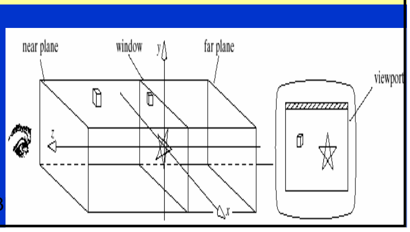
- Eye *by default* looks along –z axis at the *world window*
  - a rectangle in the *xy*-plane.
- view volume of eye is a rectangular parallelepiped
- side walls fixed by window edges & by near/far planes
- everything outside view volume is clipped
- Everything inside is projected to window plane *how?*
- z → 0 (orthogonal projection)



Repositioning the eye and the View Matrix V

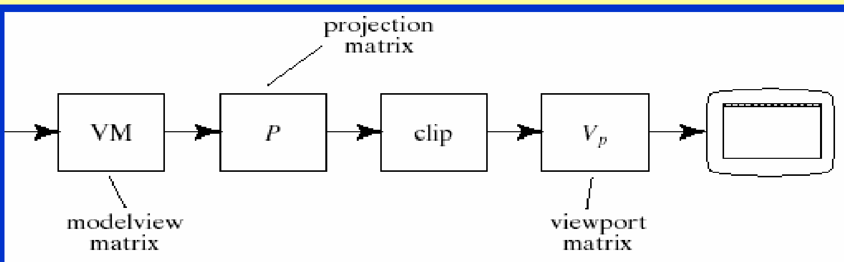
- **eye (and its view volume) can be positioned somewhre in the World. (OpenGL: use *glLookat()*)**
- ***What happens to the View Volume***
- **View Volume (parallelpiped) is accordingly positioned**
- **all objects and eye are then transformed by V**
  - **So that eye repositioned at the standard location**
- **The ModelView matrix multiplied by V: M → VM**





Graphics Pipeline Revisited (1)

- **VM includes all objects, eye, view volume transformations**
- **P is projection onto the World Window**
  - **Here we assume orthogonal transformation z→ 0**
- ***What does the viewport transformation Vp***
- **World Window → viewport to be be drawn on screen**

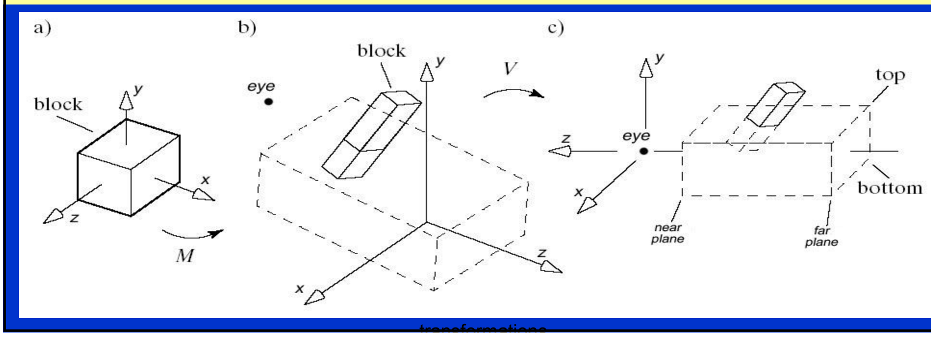


Graphics Pipeline Revisited (2)

- Input vertices in *World Coordinates*
  - using glVertex3d(x, y, z)
- Each vertex is multiplied by the various matrices, clipped if necessary, and if it survives, it is mapped onto the viewport.
- Each vertex encounters three matrices:
  - The *modelview* matrix
  - The *projection* matrix (orthogonal or perspective)
  - The *viewport* matrix

Operation of the *modelview* matrix VM

- *M* scales, rotates, & translates the cube into some block
- eye with its view volume is positioned somewhere
- *V* rotates and translates the block into a new position
  - so that Eye and view volume in the standard position
- All objects' coordinates are now called *eye coordinates*
- *Why do we put view volume along axes?*



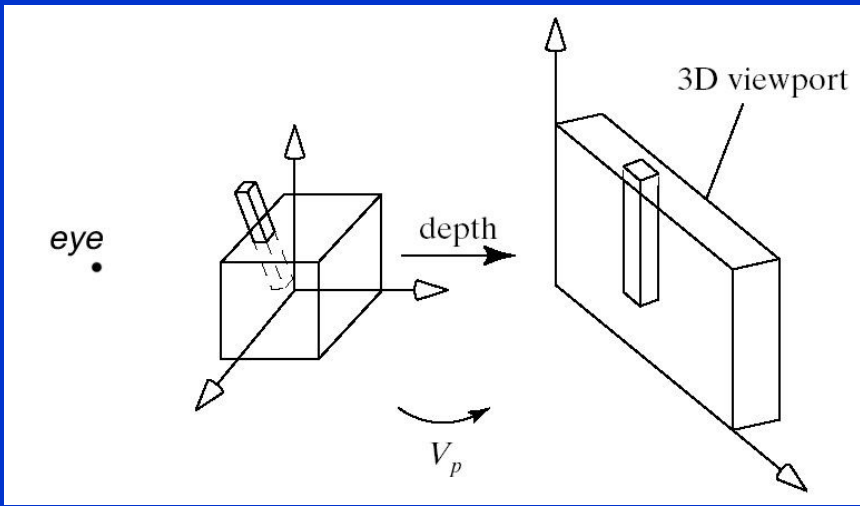
Operation of the *Projection* Matrix, P

- P translates & scales all vertices & view volume
  - the new view volume is the ***Canonical View Volume***
  - extends from -1 to 1 in each dimension
- clipping is done now (relatively easy. *Why?*)
- Coordinates now called *Normalized Device Coords*
- P also reverses the sense of the z coordinates
  - increasing values of *z* now represent relatively farther away points
- *Why do we need these Z values?*

The Viewport Matrix, Vp (1)

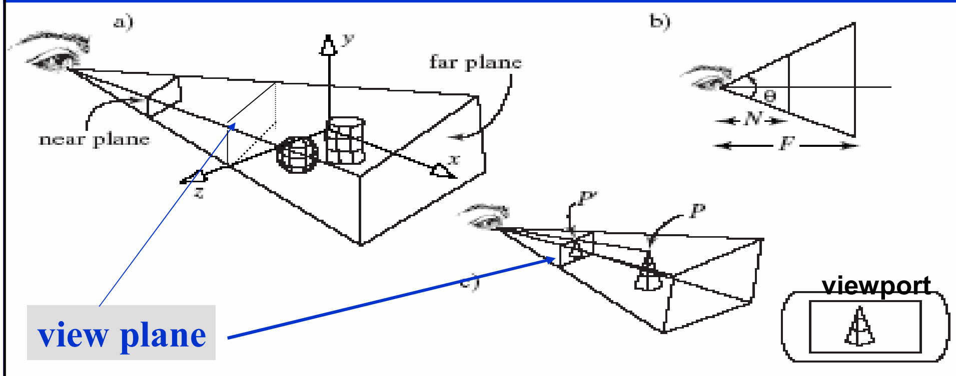
- Vp Transforms all (now clipped) objects once more!
- CVV → the ***3D viewport***:
  - *x*, *y* coordinate values extend across the viewport
  - rectangular area that we will be drawn on the screen
  - Coordinates are now called *screen coordinates*
  - *z*-component extends from 0 to 1
- a measure of the relative depth of each point
- Helps easy identification of hidden surfaces and lines

The viewport Matrix, Vp (2)



3D Perspective Drawing: Camera position and orientation

Camera & its Viewing Volume: Concepts



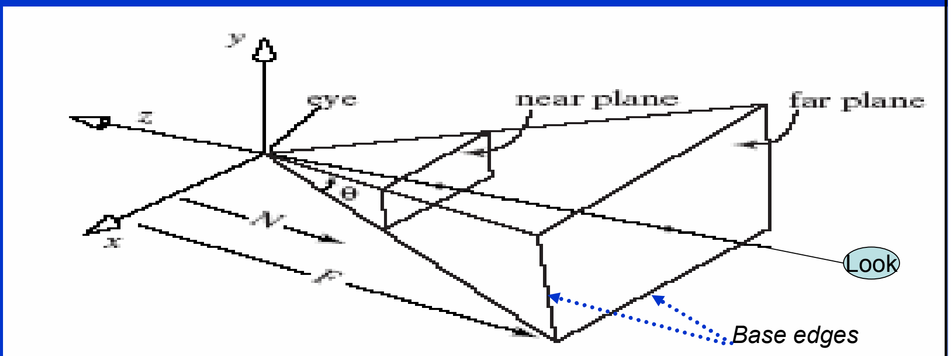
camera/eye/ "view reference point *(VRP)*"

view volume ***(what shape?)*; Near/Far/view/side planes,**

Projectors of points through *view plane* (PP') *how?*

view plane normal VPN (*direction?*), viewing angle θ

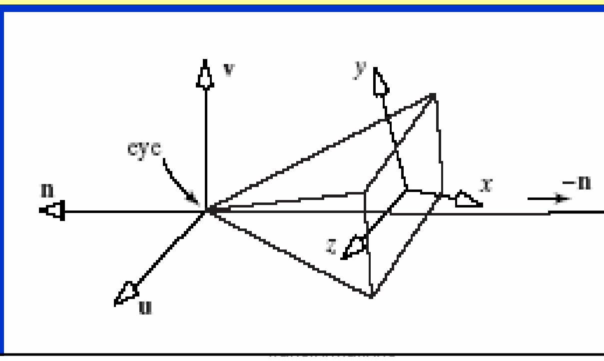
Default Position and orientation of Camera



- **VRP at O**
- **VPN in the direction of -z**
  - **Near, View, Far Planes orthogonal to z**
  - **Base edges along x, y**
- ***What info needed to define arbitrary position/orientation?***

Camera: Arbitrary position and Orientation

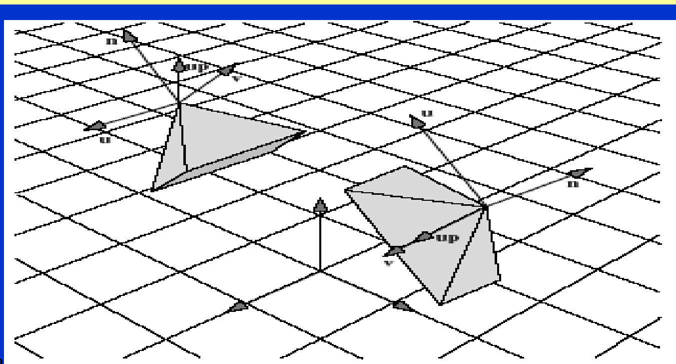
- Points **eye**, **look** **n** = **eye** – **look** view direction –**n**
- Near, View, Far planes orthogonal to **n**
- **u**, **v**, directions of pyramid base
  - **n**, **u**, **v** orthogonal
- *how many parameters we need?*



Look

OpenGL Construction: **eye**, **look**, **up** → **n**, **u**, **v**

- **n** = **eye - look**
- **up** is some "up" vector (typically along y)
  - Not necessarily orthogonal to **n**
- **u** = **up** x **n** *how do we find v*
- **v** = **n** x **u** ; **v** and **up** are not necessarily parallel



The View Transformation, V

- **transform the scene s.t. camera goes to default**
- **Multiplies the** *modelview* matrix, after modeling

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

d_x = **-(<u>eye</u>·<u>u</u>)**

d_y = **-(<u>eye</u>·<u>v</u>)**

d_z = **-(<u>eye</u>·<u>n</u>)**

<u>eye</u> = **eye - O**

prove that V transforms camera to default. How?

Proof: V transforms (**eye**, **<u>u</u>**, **<u>v</u>**, **<u>n</u>**) → (**O**, **<u>i</u>**, **<u>j</u>**, **<u>k</u>**)

- V**eye** = V(eye_x, eye_y, eye_z, 1)^T = (0, 0,0,1)^T = **O**
- V**<u>u</u>** = V(u_x, u_y, u_z, 0)^T = (1, 0, 0, 0)^T = **<u>i</u>**
- V**<u>v</u>** = V(v_x, v_y, v_z, 0)^T = (0, 1, 0, 0)^T = **<u>j</u>**
- V**<u>n</u>** = V(n_x, n_y, n_z, 0)^T = (0, 0, 1, 0)^T = **<u>k</u>**
- After applying the viewing transformation camera is in the default position and orientation
  - Origin, looking at the *negative z direction*
- All objects in the model have new coordinates
  - The *<u>"eye coordinates"</u>*

Applying View Transformation in OpenGL

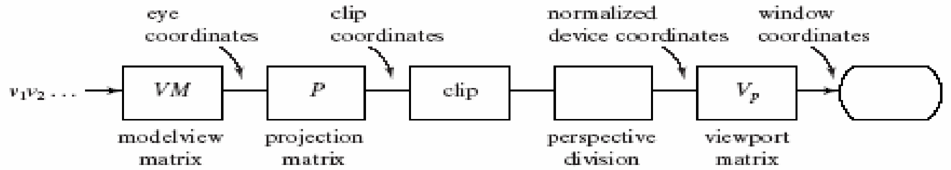
glMatrixMode(GL_MODELVIEW);

glLoadIdentity(); // start with a unit matrix

gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y, up.z)

Graphics pipeline (again) & Coordinates

- Input Vertices coordinates are ***"world coordinates"***
- after VM, coordinates are ***"eye coordinate"***
- After Projection, P, coordinates are ***"clip coordinates"***
- after perspective division, coordinates are ***"Normalized Device Coordinates"***
- After Viewport Transformation, Vp, coordinates are **"screen coordinates"** (of which x, y are pixel values)

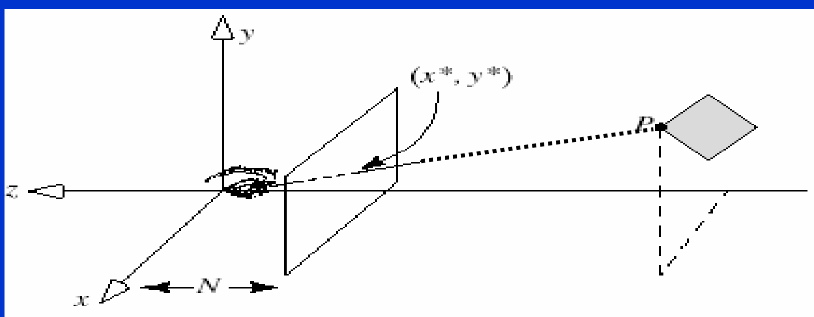


Projection: What do we have to calculate?

Perspective Projection

Perspective Projections of 3-D Point

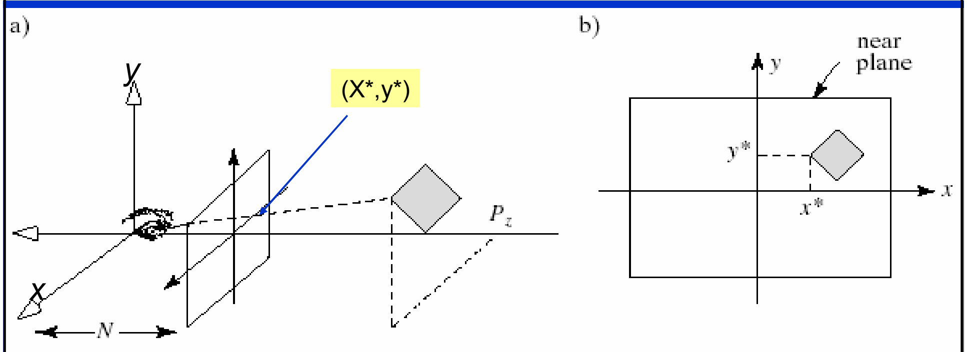
- A vertex located at *P* in eye coordinates is projected to a certain point (x^*, y^*, z*) on the View plane



This is a non-affine transformation!

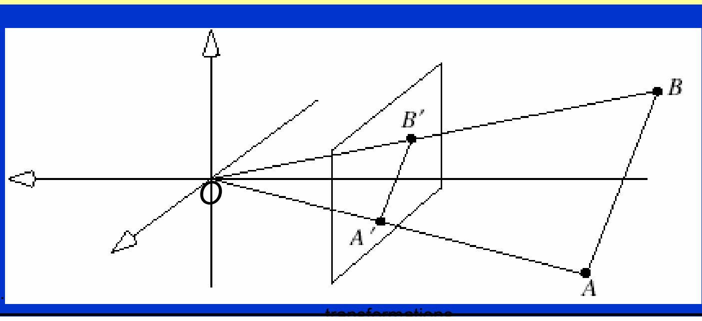
Perspective Projections of Point

- **(P_x, P_y, P_z) projects to P* ≡ (x*, y*, z*)**
- **$x^*/P_x = N/(-P_z)$, $y^*/P_y = N/(-P_z)$ (similar triangles)**
- **$P^* = (x^*, y^*) = N/(-P_z)\ (P_x, P_y)$ (ignore z* now)**
- *What happens to far away points?*



Perspective Projection of lines

- Straight lines project to straight lines:
- Plane through **A**, **B**, **O** intersect the View Plane in a straight line via **A'B'**
- Straight line segment **AB** projected into straight line segment **A'B'**
- *What is the projection of line* **BO**?



Perspective Projection of lines (2)

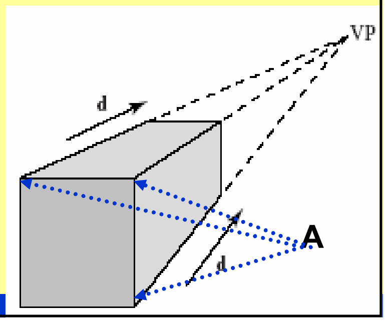
- Line points: $\mathbf{P} = (P_x(t), P_y(t), P_z(t)) =$
- $= \mathbf{A} + \underline{\mathbf{c}}t = (A_x + c_x t, A_y + c_y t, A_z + c_z t)$
  - $\underline{\mathbf{c}}$ determines direction of the line
  - $\mathbf{A}$ determines its location: $\mathbf{A} = \mathbf{P}(0)$
- N is the distance from the eye to the Projection Plane
- $\mathbf{P}^*(t)$ projection of $\mathbf{P}(t)$
  - $\mathbf{P}^*(t) = -(N/[A_z + c_z t])(A_x + c_x t, A_y + c_y t)$
  - *Is this a straight line? Prove it.*
- *How parallel lines are projected?*

Projection of Lines orthogonal to $\underline{\mathbf{n}}$

- $\mathbf{A} \equiv \mathbf{P(0)} \rightarrow \mathbf{P}^*(0) = -(N/A_z)(A_x, A_y)$
- 1) If the line orthogonal to $\underline{\mathbf{n}}$ (parallel to the View Plane)
  - $c_z = 0$, $\mathbf{P}_z = A_z$
  - $\mathbf{P}^*(t) = -N/A_z\,(A_x + c_x t, A_y + c_y t)$.
  - This is a line in View Plane, with slope c_y/c_x
  - lines with same $\underline{\mathbf{c}}$ projected to a line with same slope
- *if two lines are parallel to each other and orthogonal to $\underline{\mathbf{n}}$ (they are parallel to the View Plane), they project to two parallel lines*

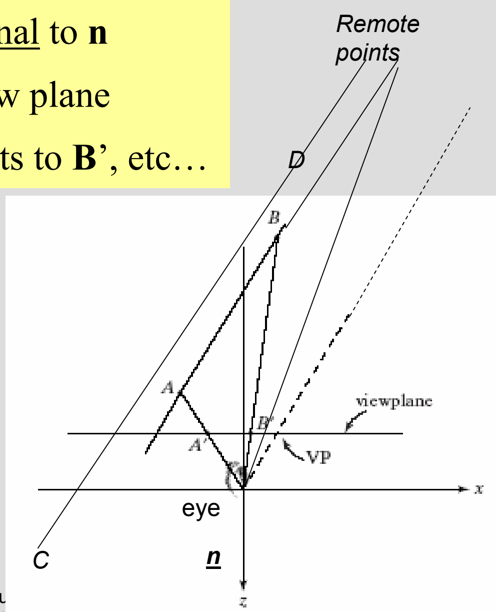
Projection of Lines not parallel to the View Plane

- 2) If the line is not orthogonal to **n** (not parallel to View Plane) then $c_z \neq 0$
- take limit as $t \rightarrow \infty$
  - $\mathbf{P^*}(t) \rightarrow \mathbf{P^*}(\infty) = -N/c_z\ (c_x, c_y)$ Point independent of **A!**
- lines w/ dir **c** are projected to lines
  - projected lines meet at $P^*(\infty)$
  - The "vanishing point", *VP*
  - depends on direction **c**
- projected lines *are not parallel.*



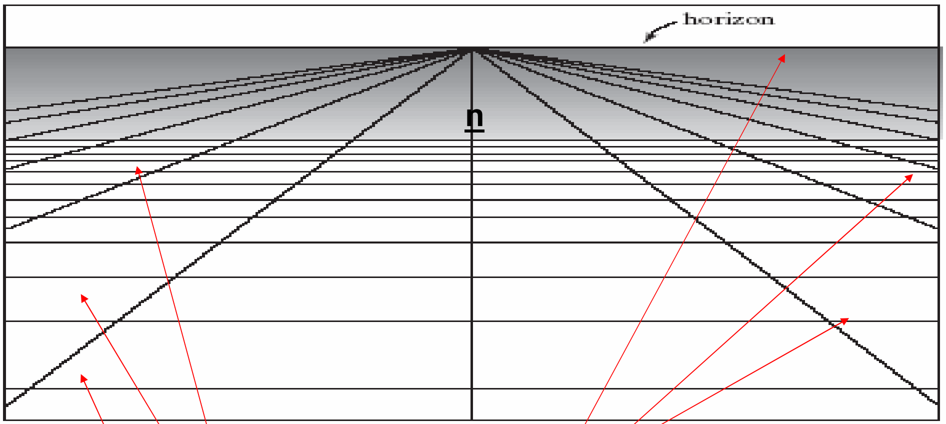
What is the vanishing point (VP)

- Consider line not orthogonal to **n**
  - Line not parallel to view plane
- **A** projects to **A'**, **B** projects to **B'**, etc…
- Very remote points on the line project to VP
- Line from eye to VP parallel to line AB.
- The VP is the image of the point $t = \infty$ of all lines parallel to AB (e.g. CD)



Example: horizontal grid in perspective

lines orthogonal to **n** → parallel lines
Lines parallel to **n** → lines meet at vanishing point

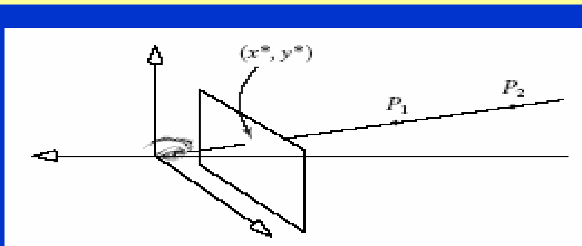


Images of lines orthogonal to **n**

Images lines parallel to **n**

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 $w \neq 0$
- All $w \neq 0$ 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.
- Affine transformation on a point **P** transform it to the same point **P'** irrespective of value of w

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)

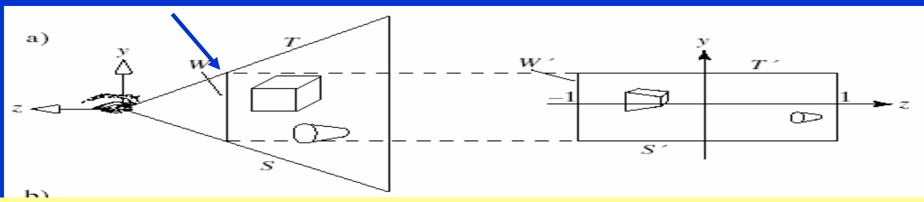
Perspective projection in OpenGL

- perspective projection done in separate steps
- perspective transformation separated from the orthogonal projection (throwing third component)
  - clipping, perspective division, and one additional scaling are inserted between them

Properties of Perspective Transformation

- It preserves straightness and flatness, so lines transform into lines, planes into planes, and polygonal faces into other polygonal faces.
- It preserves in-between-ness, if point a is inside an object, the transformed point will also be inside the transformed object.
- The choice of the pseudo-depth function was guided by the need to preserve these properties.

The transformed view volume (1)



- line through the eye map to <u>a single point</u> on the view plane
  - Its points map to different depth value z*
  - lines through eye transformed to lines parallel to Z axis
- 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

The transformed View Volume (2)

- After the perspective transformation the view volume is bounded by the 6 planes
  - $y = bott$, $y = top$, $x = left$, $x = right$, $z = -1$, $z = +1$
- $top = N*\tan((\pi/180)*viewAngle)$ $\quad bott = -top$
- $right = top*aspect$ $\quad left = -right$
- Clipping is done relative to this view volume
- *Can we simplify clipping by transforming the view volume?*

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

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

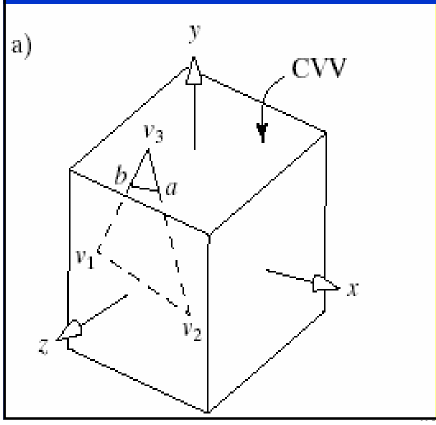
Applying Projection Matrix in OpenGL

- *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

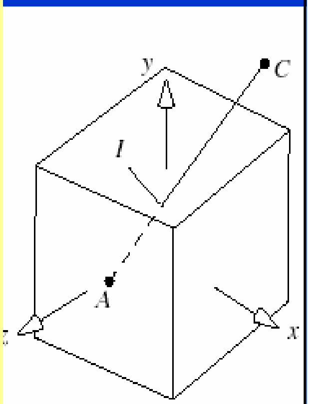
- **A triangle has vertices v_1, v_2,v_3.**
- **v_3 is outside *CVV***



- 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

- 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 <u>same side of a wall</u>



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 does P lies to the right of X= -1 plane?*
- if $x/w > -1 \rightarrow w + x > 0$.
- *When does P lies to the left of plane X = 1?*
- if $x/w < 1 \rightarrow w - x > 0$.
- The 6 quantities $w \pm x$, $w \pm y$, $w \pm z$ 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 $\mathbf{P}(t) = \mathbf{A} + (\mathbf{C}-\mathbf{A})t \qquad 0 <= t <= 1$
  - 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: $BC_1(\mathbf{A}) = a_w - a_x \quad BC_1(\mathbf{C}) = c_w - c_x$ (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

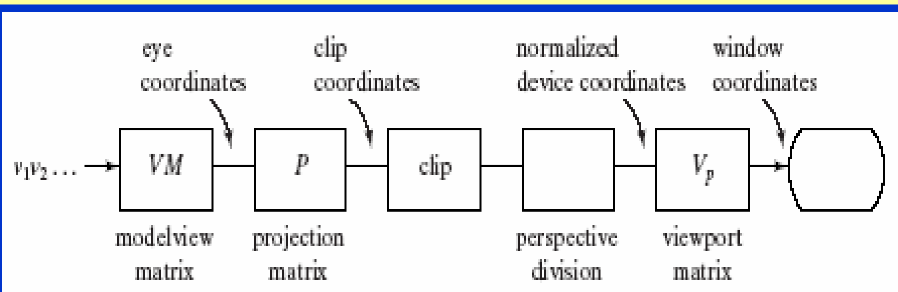
- $CI = [t_{in} = 0.0; t_{out} = 1.0]$
- We test the line segment against each wall i in turn.
- If BCi(A), BCi(C) have opposite signs, find t_{hit}
  - If segment is entering update $t_{in} = max(old\ t_{in}, t_{hit})$
  - if segment is exiting, update $t_{out} = min(old\ t_{out}, t_{hit})$
- If, at 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.

Why Use the CVV?

- It is parameter-free: the algorithm needs no extra information to describe the clipping volume.
- It uses only the values -1 and 1. So the code itself can be highly tuned for maximum efficiency.
- Its planes are aligned with the coordinate axes (after the perspective transformation is performed).
- we can determine which side of a plane a point lies on using a single coordinate, as in $a_x > -1$.
- If the planes were not aligned, an expensive dot product would be needed.

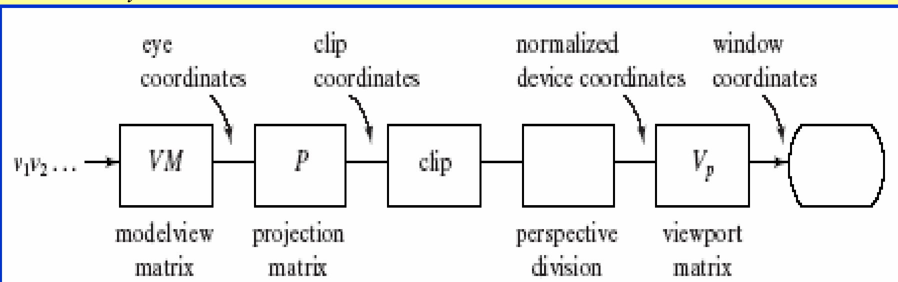
Steps in the path of a vertex P through the Pipeline

- ***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
- The edge having this point as an endpoint is clipped.



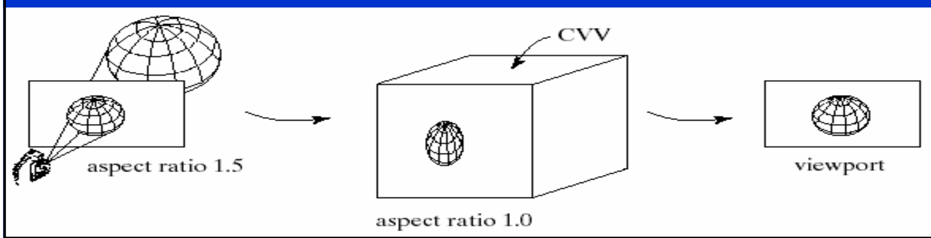
Steps in the path of a vertex P through the Pipeline

- 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)



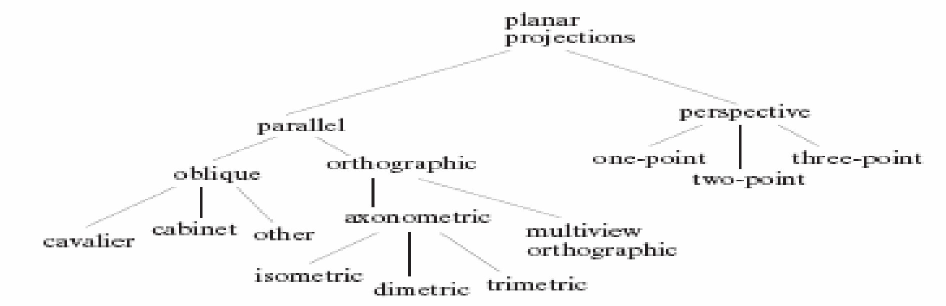
Distortion and Its Removal in OpenGL

- *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.

Appendix

Classification of Perspective Projections

Classifying Planar Projections



•<u>Perspective projections</u>: points projected to view plane via projectors converging to point **eye**.

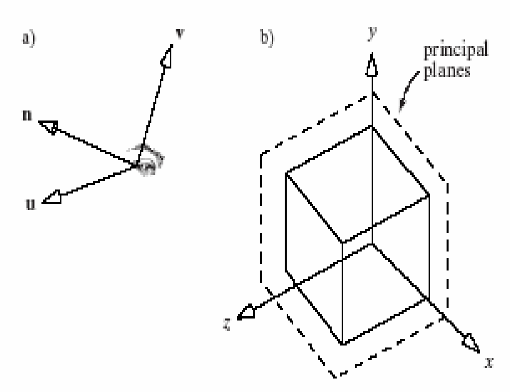
- •Directions of projectors vary

•<u>Parallel projections</u>: Points projected to view plane via projectors that All have the same direction <u>**d**</u>

Types of Perspective Projection

- Types differ by the orientation of the camera relative to the World Coordinate System
- Orientation of the camera is defined by the **n**, **u**, **v** vectors

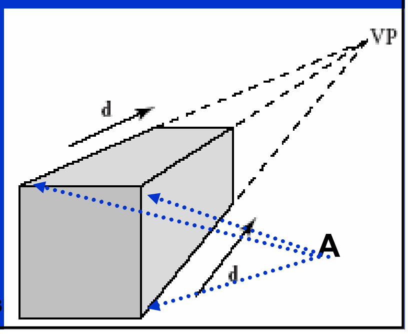
- Camera looks along **n**
- **u**, **v** parallel to view plane
- View volume base edges are along **u**, **v**.



Principal axes and their vanishing points

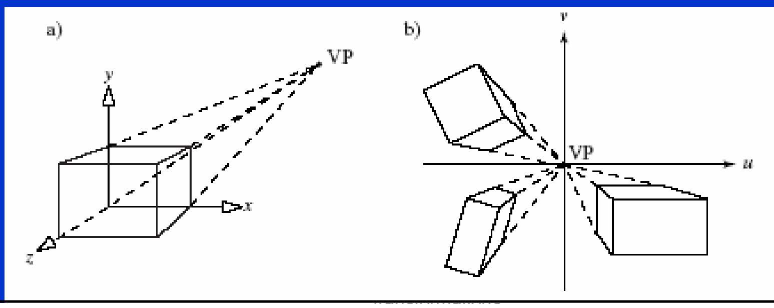
- Principal axes: x, y, z ; Principal planes: (x,y), (y,z), (z,x)
- Reminder: If a principal axis is orthogonal to **n**, (parallel to view plane) it does not have a vanishing point
- lines parallel to such axis will be projected to parallel lines

- Otherwise, if a principal axis is not orthogonal to n, it has a vanishing point
- All lines parallel to such axis will be projected as lines in the view plane, which meet at the vanishing point



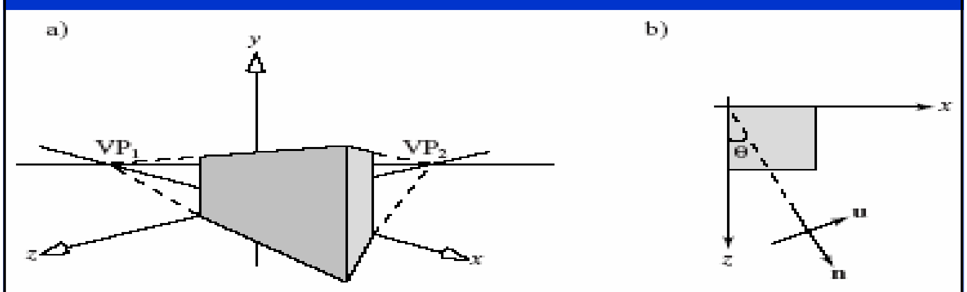
One point Perspective Projection

- exactly one axis has a vanishing point VP, e.g. z axis
  - lines parallel to the axis projected to lines that meet at VP
- The two other axes (e.g. x, y) must be orthogonal to **n**
  - **n** is perpendicular to a principal plane (e.g. (x, y)) plane
  - **n** has two zero coordinates ($n_x = n_y = 0$)



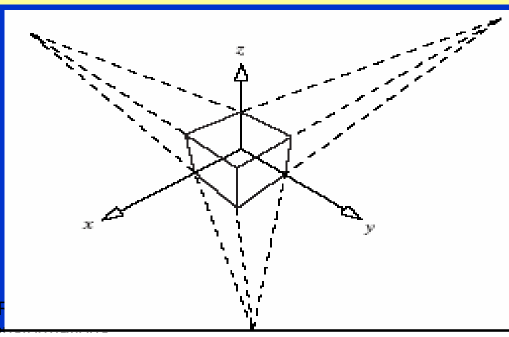
Two points perspective projection

- exactly 2 axes have vanishing points VPs, e.g. x, z axes
- lines parallel to axes projected to lines which meet in VPs
  - The third axis (y) must be orthogonal to **n**
  - **n** has one zero coordinate e.g. $\mathbf{n}_y = 0$
- used for drawings buildings. Camera looks at an edge



Three-point perspective projection

- all 3 axes have vanishing points.
  - Lines parallel to any axis projected to lines meeting at its VP
- No axis is orthogonal to **n**
  - no component of **n** is 0.
- Example: looking up or down at the corner of an object.



Example three-point perspective projection