CS 4204 Computer Graphics

3D Viewing and Projection

Yong Cao

Virginia Tech

Objective

- ***We will develop methods to camera through scenes.***
- ***We will develop mathematical tools to handle perspective projection.***

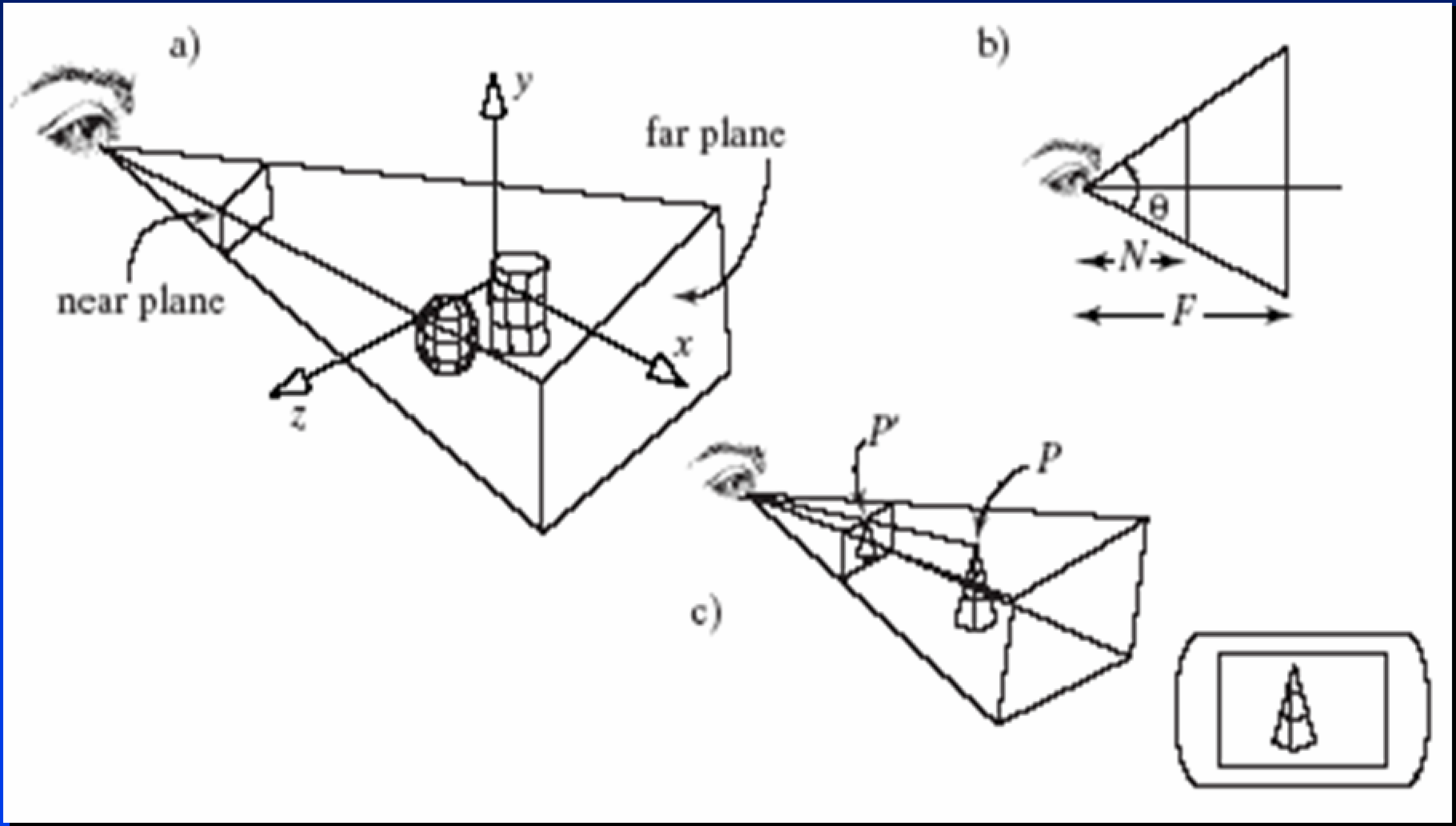
The Camera and Perspective Projection

- ***The camera has an*** *eye* ***(****or view reference point* ***VRP) at some point in space.***
- ***Its*** *view volume* ***is a portion of a pyramid, whose apex is at the eye. The straight line from a point*** **P** ***to the eye is called the*** *projector* ***of*** **P*****. (All projectors of a point meet at the eye.)***
- ***The axis of the view volume is called the*** view plane normal, ***or*** VPN.
- ***The opening of the pyramid is set by the*** *viewangle* ***θ (see part b of the figure).***

The Camera and Perspective Projection (Refer to the picture on next slide)

- ***Three planes are defined perpendicular to the VPN: the** near plane, **the** view plane, **and the** far plane.*
- ***Where the planes intersect the VPN they form rectangular windows. The windows have an** aspect ratio **which can be set in a program.***
- ***OpenGL clips points of the scene lying outside the view volume. Points P inside the view volume are projected onto the** view plane **to a corresponding point P' (part c).***
- ***Finally, the image formed on the view plane is mapped into the viewport (part c), and becomes visible on the display device.***

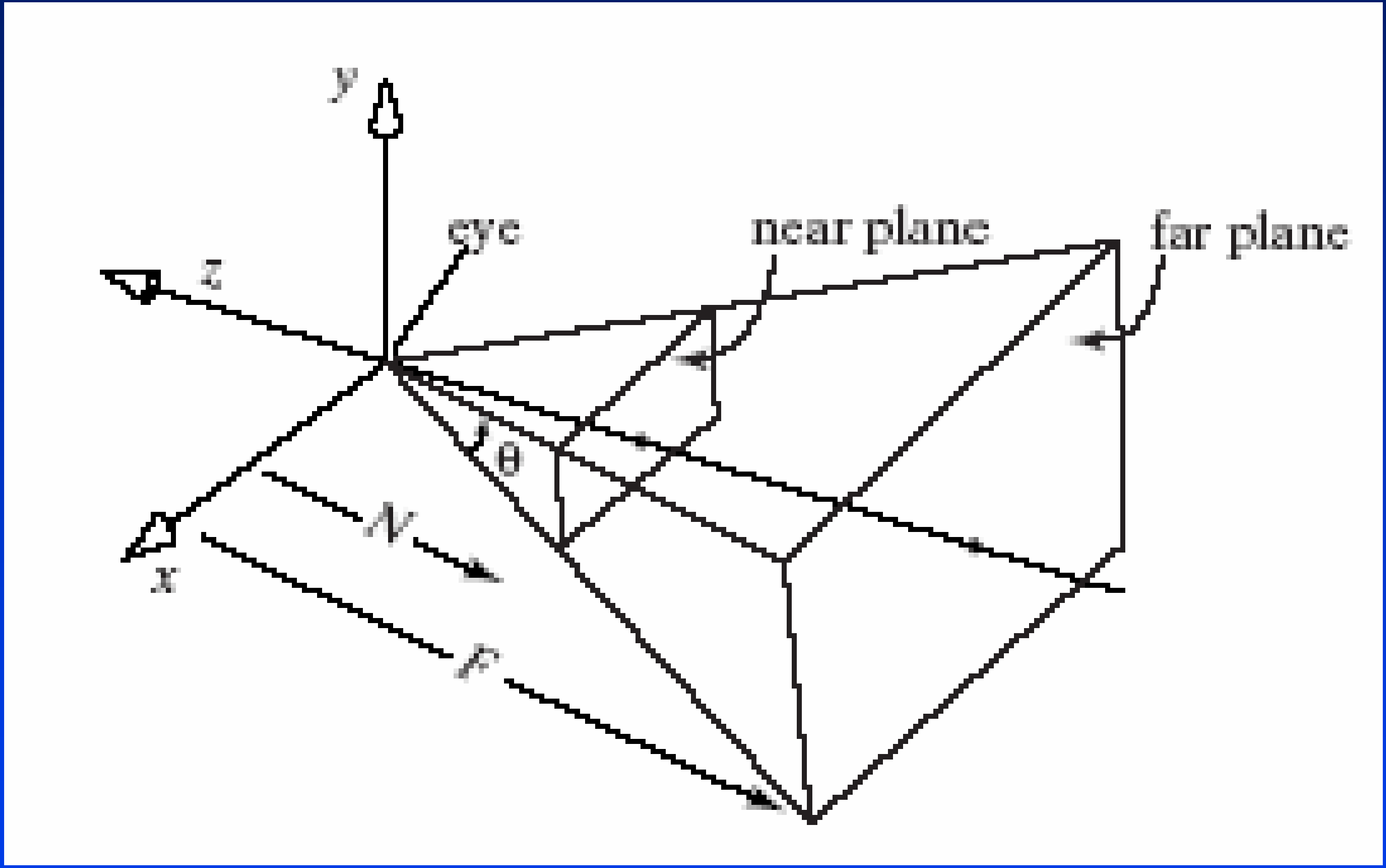
The Camera and Perspective Projection



Setting the View Volume

- ***The default camera position has the eye at the origin and the*** *VPN* ***aligned with the z-axis.***
- ***The programmer defines a*** **look** ***point as a point of particular interest in the scene, and together the two points eye and look define the VPN as*** **eye – look**.
  - This is later normalized to become the vector **n**, which is central in specifying the camera properly. (VPN points from *look* to *eye*.)

Setting the View Volume (2)



Setting the View Volume (3)

- *To view a scene, we move the camera and aim it in a particular direction.*
- *To do this, perform a rotation and a translation, which become part of the* modelview matrix.
- *Set up the camera's position and orientation in exactly the same way we did for the parallel-projection camera.*

```
glMatrixMode(GL_MODELVIEW);
  // make the modelview matrix current
        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);
```

Setting the View Volume (4)

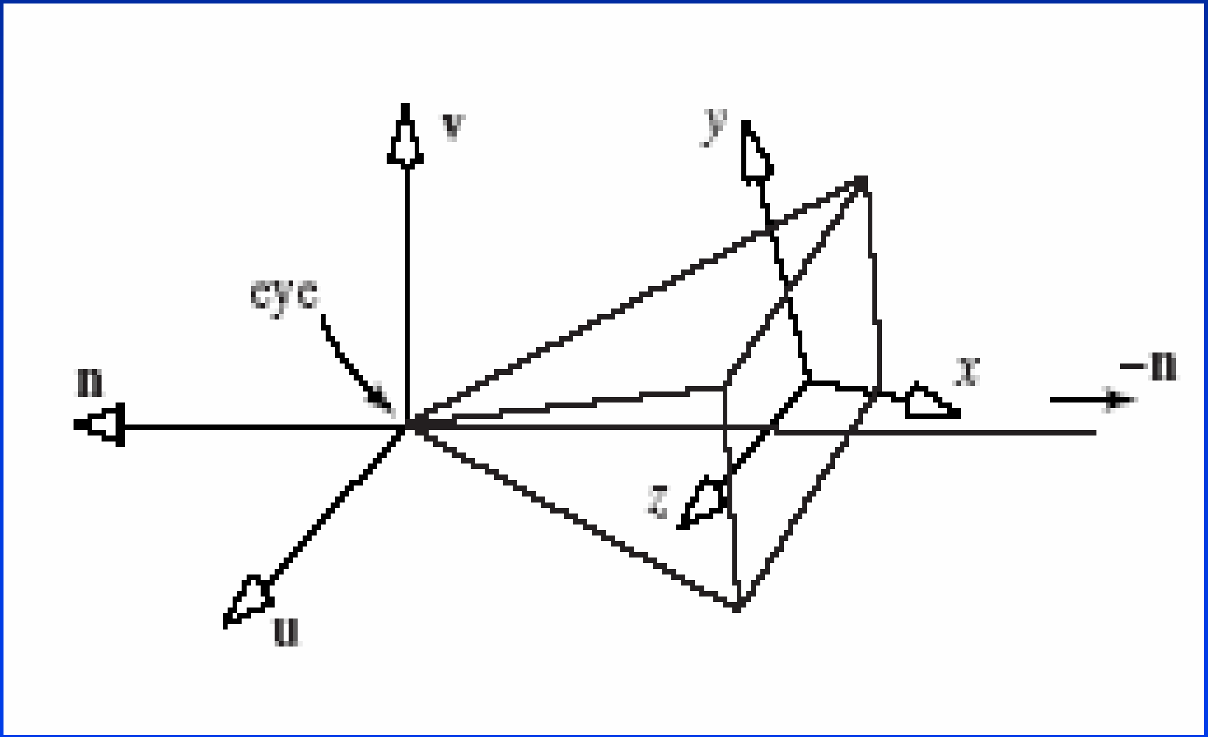
- *As before, this moves the camera so its eye resides at point* eye, *and it "looks" towards the point of interest,* look.
- *The "upward" direction is generally suggested by the vector* up, *which is most often set simply to (0, 1, 0).*

Camera with Arbitrary Orientation and Position

- *A camera can have any position and orientation in the scene.*
- *Imagine a transformation that picks up the camera and moves it somewhere in space, then rotates it around to aim it as desired.*
- *To do this we need a coordinate system attached to the camera: u, v, and n.*

Camera with Arbitrary Orientation and Position (2)

***v* points vertically upward, *n* away from the view volume, and *u* at right angles to both *n* and *v*. The camera looks toward -*n*. All are normalized.**



gluLookAt and the Camera Coordinate System

***gluLookAt* takes the points eye and look, and the vector** *up*

n **must be parallel to eye - look, so it sets** *n* **= eye - look**

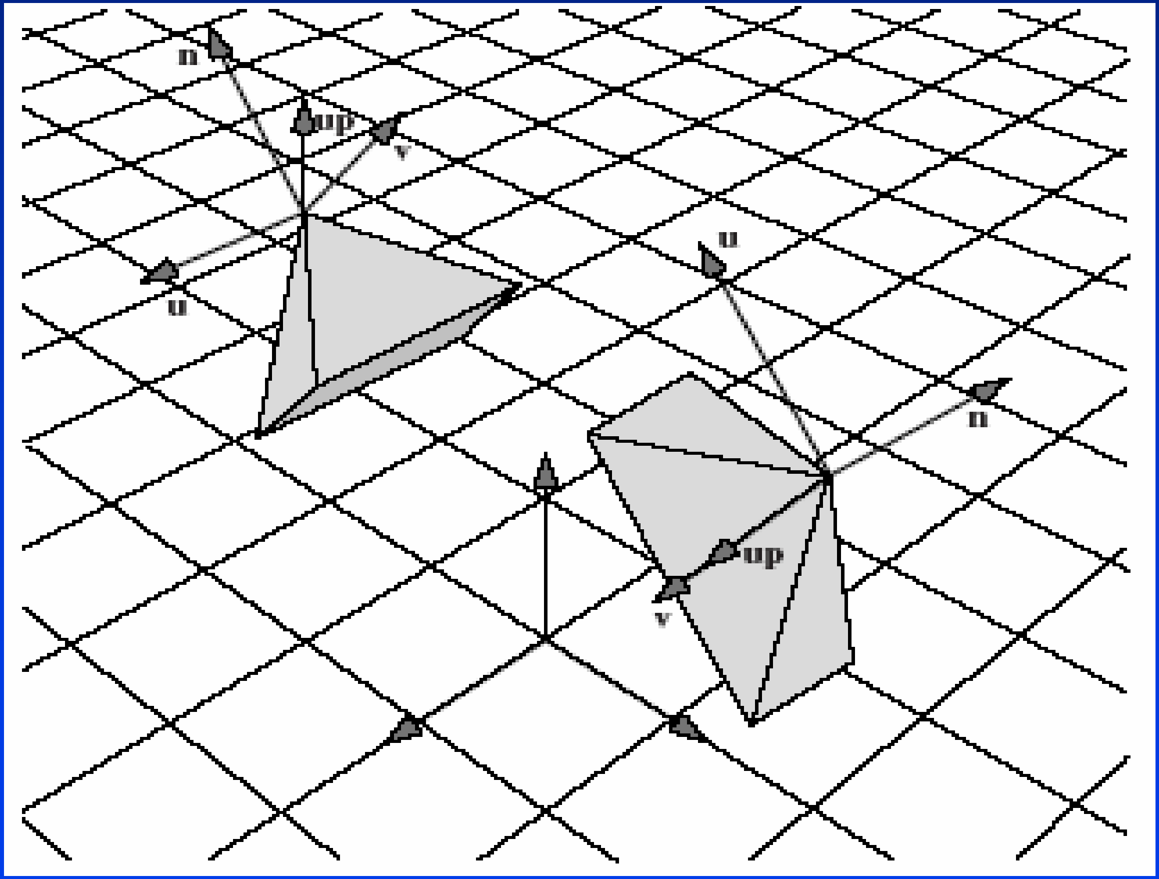
u **points "off to the side", so it makes** *u* **perpendicular to both** *n* **and** *up*: *u* **=** *up* **x** *n*

v **must be perpendicular to** *n* **and** *u*, **so it lets** *v* **=** *n* **x** *u*

Note that *v* **and** *up* **are not necessarily in the same direction, since** *v* **must be perpendicular to** *n*, **and** *up* **need not be.**

gluLookAt and the Camera Coordinate System (2)

Effect of gluLookAt (Demo)



gluLookAt and the Camera Coordinate System (3)

The view matrix V created by gluLookAt is

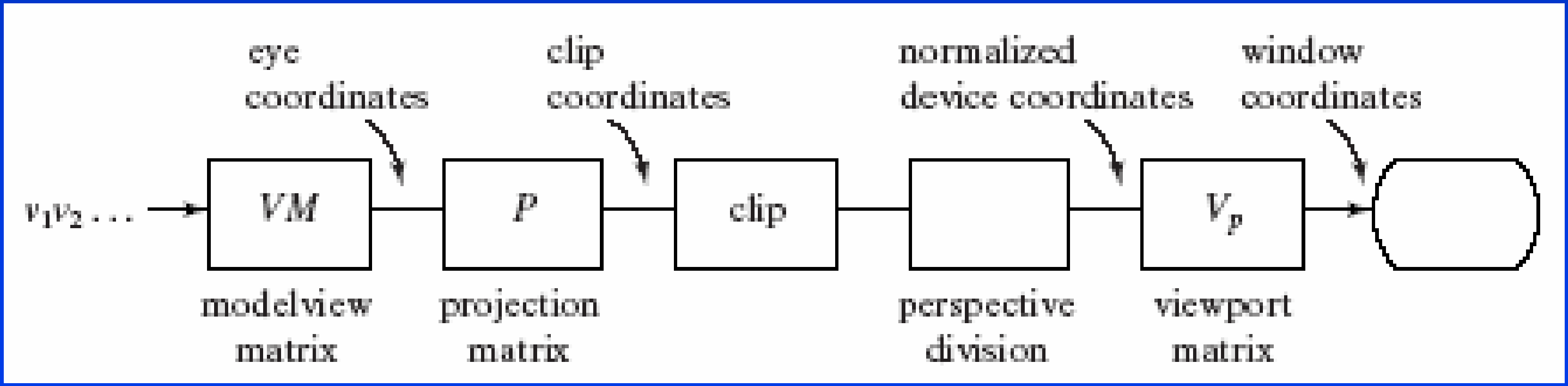
$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

where d_x = -eye·u, d_y= -eye·v, d_z= -eye·n

V is postmultiplied by M to form the modelview matrix VM.

Perspective Projections of 3-D Objects

The graphics pipeline: vertices start in world coordinates; after MV, in eye coordinates, after P, in clip coordinates; after perspective division, in normalized device coordinates; after V, in screen coordinates.

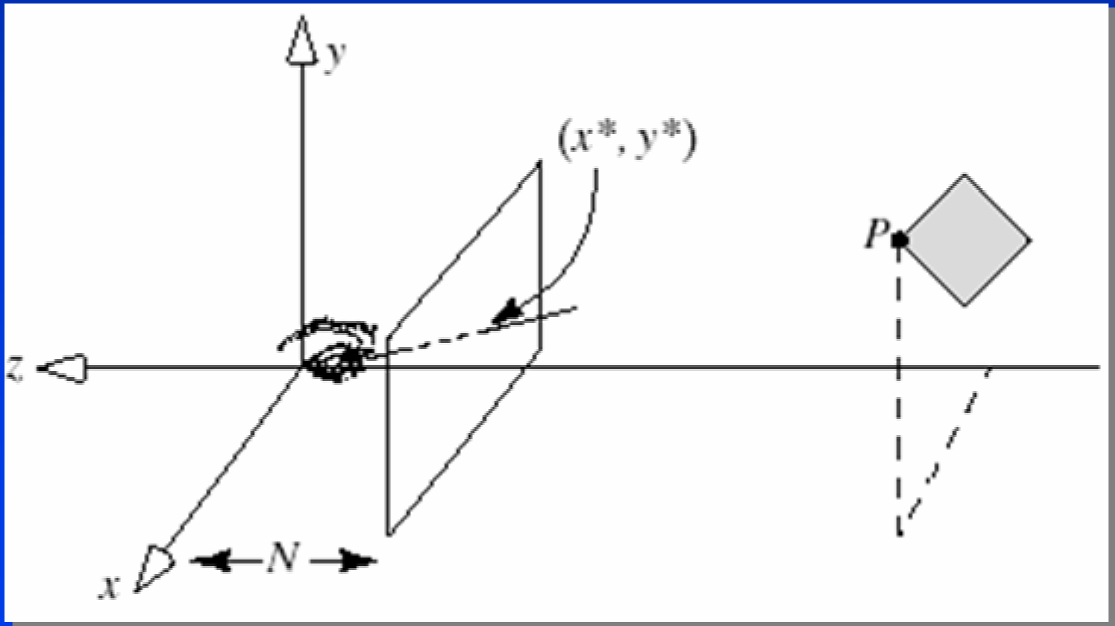


Perspective Projections of 3-D Objects (2)

- *Each vertex v is multiplied by the modelview matrix (VM), containing all of the modeling transformations for the object; the viewing part (V) accounts for the transformation set by the camera's position and orientation. When a vertex emerges from this matrix it is in* eye coordinates, *that is, in the coordinate system of the eye.*
- *The figure shows this system: the eye is at the origin, and the near plane is perpendicular to the z-axis, located at z = -N.*

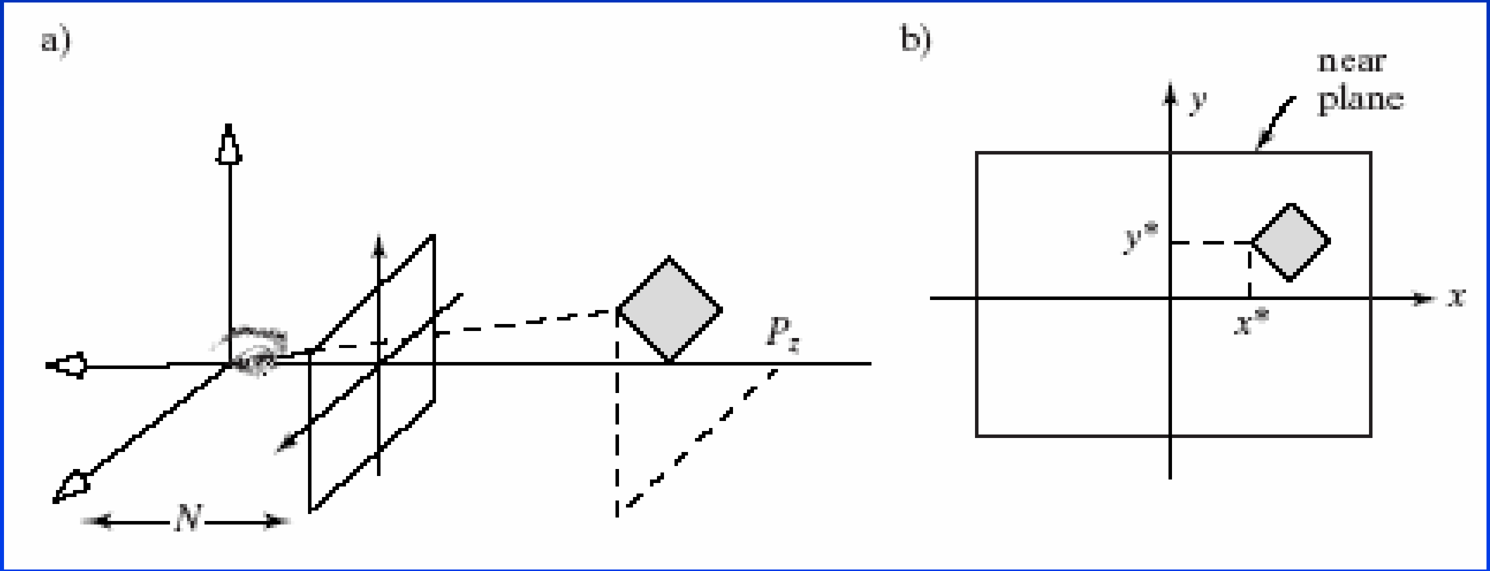
Perspective Projections of 3-D Objects (3)

A vertex located at* P *in eye coordinates is passed through the next stages of the pipeline where it is projected to a certain point (x*, y*) on the near plane, clipping is carried out, and finally the surviving vertices are mapped to the viewport on the display.



Perspective Projections of 3-D Objects (4)

We erect a local coordinate system on the near plane, with its origin on the camera's z-axis. Then it makes sense to talk about the point x* units right of the origin, and y* units above the origin.

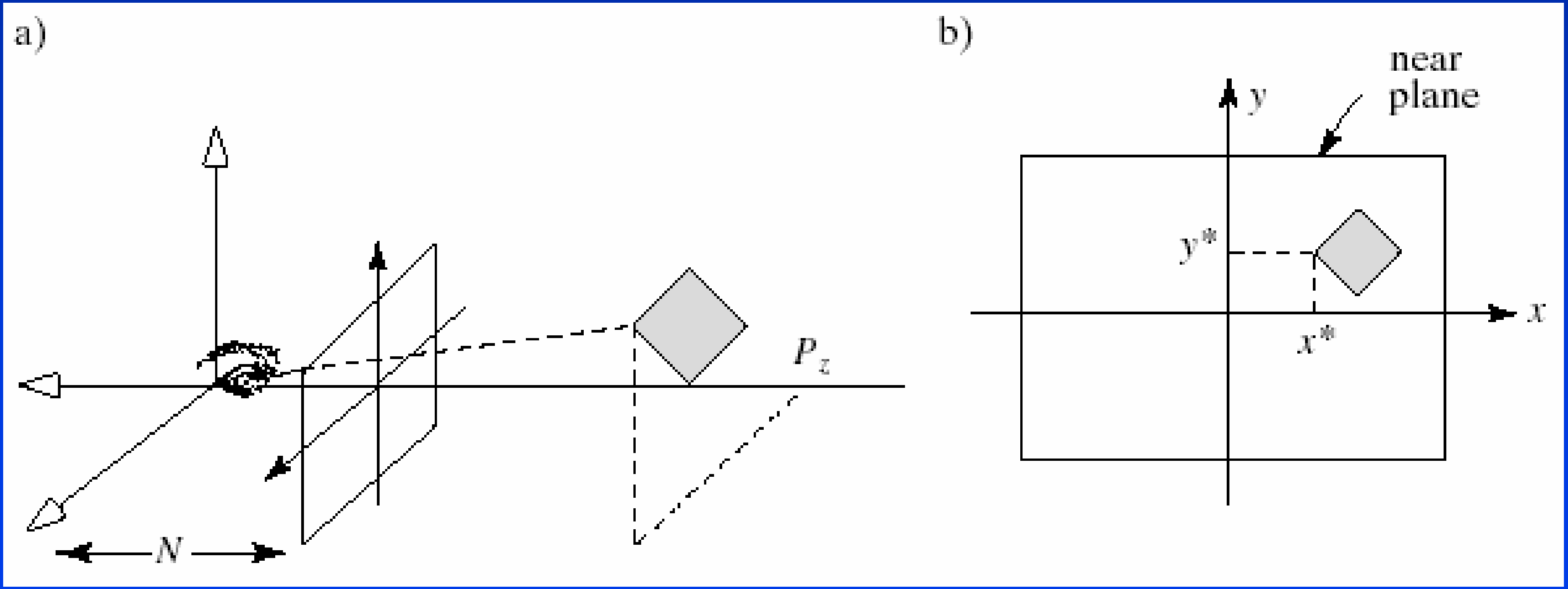


Perspective Projections of 3-D Objects (5)

(P_x, P_y, P_z) projects to (x^*, y^*).

$x^*/P_x = N/(-P_z)$ and $y^*/P_y = N/(-P_z)$ by similar triangles.

Thus $P^* = (x^*, y^*) = (N P_x/(-P_z), N P_y/(-P_z))$.

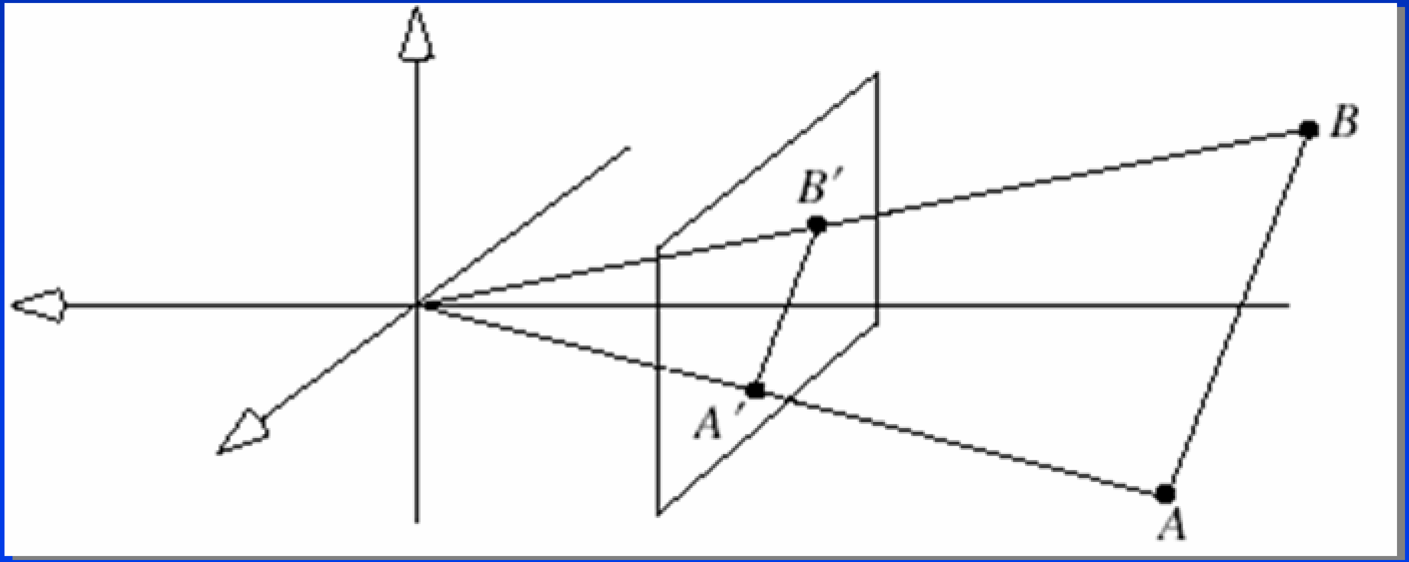


Perspective Projection Properties

- *$|P_z|$ is larger for points further away from the eye, and, because we divide by it, causes objects further away to appear smaller (perspective foreshortening).*
- *We do not want $P_z \geq 0$; generally these points (at or behind eye) are clipped.*
- *Projection to a plane other than N simply scales P*; since the viewport matrix will scale anyway, we might as well project to N.*

Perspective Projection Properties (2)

- *Straight lines project to straight lines. Consider the line between* A *and* B. A *projects to* A' *and* B *projects to* B'.
- *In between: consider the plane formed by* A, B, *and the origin. Since any two planes intersect in a straight line, this plane intersects the near plane in a straight line. Thus line segment* AB *projects to* line segment A'B'.

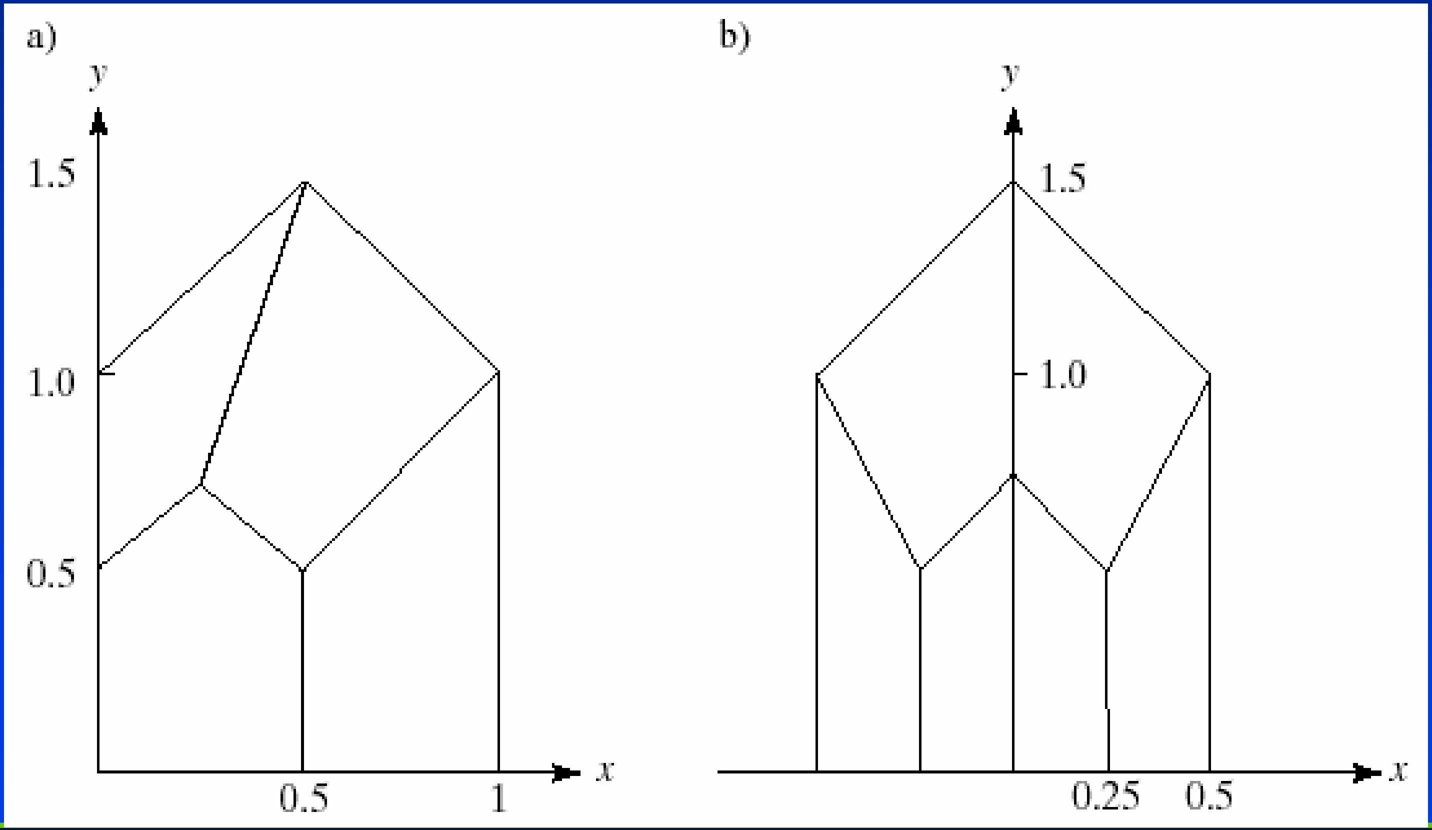


Example Projections of the Barn

- *View #1: The near plane coincides with the front of the barn.*
- *In camera coordinates all points on the front wall of the barn have $P_z = -1$ and those on the back wall have $P_z = -2$. So any point (P_x, P_y, P_z) on the front wall projects to $P' = (P_x, P_y)$ and any point on the back wall projects to $P' = (P_x/2, P_y/2)$.*
- *The foreshortening factor is two for points on the back wall. Note that edges on the rear wall project at half their true length. Also note that edges of the barn that are actually parallel in 3D need not project as parallel.*

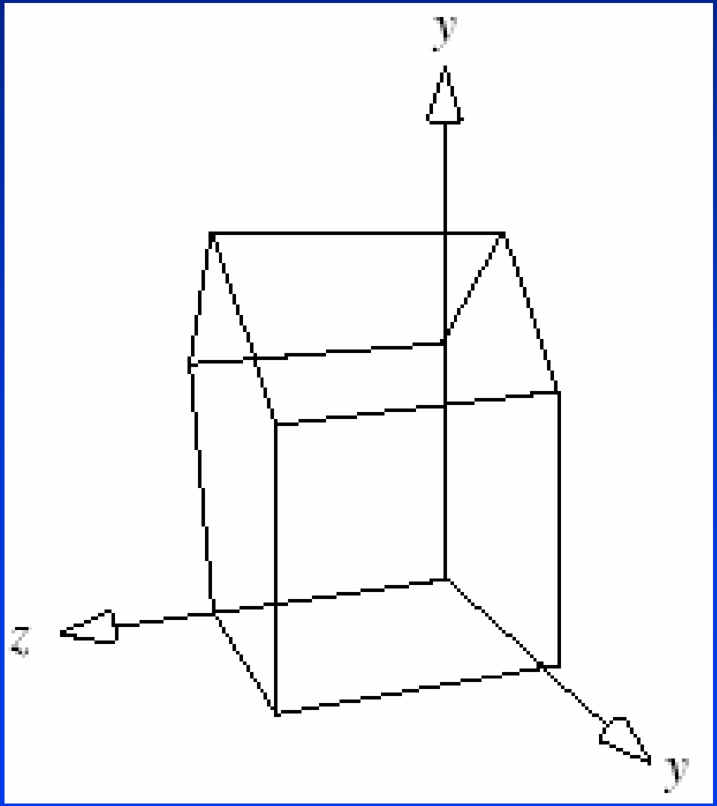
Example (2)

In part b, the camera has been moved right, but everything else is the same.



Example (3)

In part c, we look down from above and right on the barn.



Perspective Projection of Lines

- *Straight lines are transformed to straight lines.*
- *Lines that are parallel in 3D project to lines, but not necessarily parallel lines. If not parallel, they meet at some vanishing point.*
- *If $P_z \geq 0$, lines that pass through the camera undergo a catastrophic "passage through infinity"; such lines must be clipped.*
- *Perspective projections usually produce geometrically realistic pictures. But realism is strained for very long lines parallel to the viewplane.*

Projection of Straight Lines (2)

- *Effect of projection → on parallel lines: $P = A + ct \rightarrow p(t) = -N([A_x + c_x t]/[A_z + c_z t], [A_y + c_y t]/[A_z + c_z t]) = - N/[A_z + c_z t] (A_x + c_x t, A_y + c_y t)$.*
  - N is the distance from the eye to the near plane.
- *Point $A \rightarrow p(0) = - N/A_z\ (A_x, A_y)$.*
- *If the line is parallel to plane N, $c_z = 0$, and $p(t) = - N/A_z (A_x + c_x t, A_y + c_y t)$.*
- *This is a line with slope c_y/c_x and all lines with direction $c \rightarrow$ a line with this slope.*
- *Thus if two lines in 3D are parallel to each other and to the viewplane, they project to two parallel lines.*

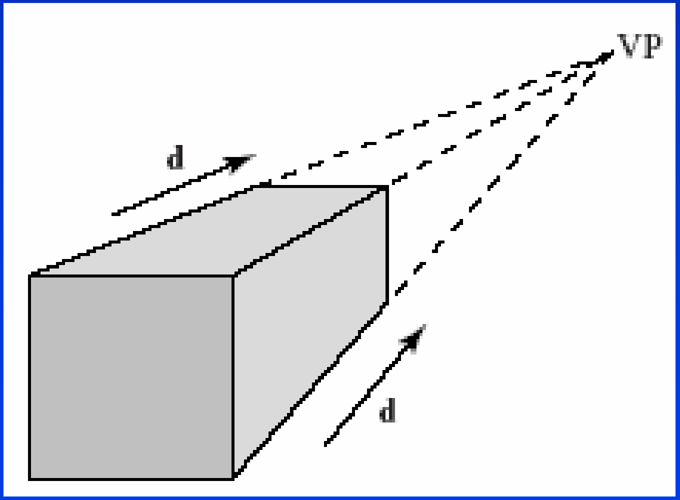
Projection of Straight Lines (3)

If the line is not parallel to plane N (near plane), look at limit as t becomes ∞ for p(t), which is -N/c_z (c_x, c_y), a constant.

- All lines with direction **c** reach this point as t becomes ∞; it is called the vanishing point.

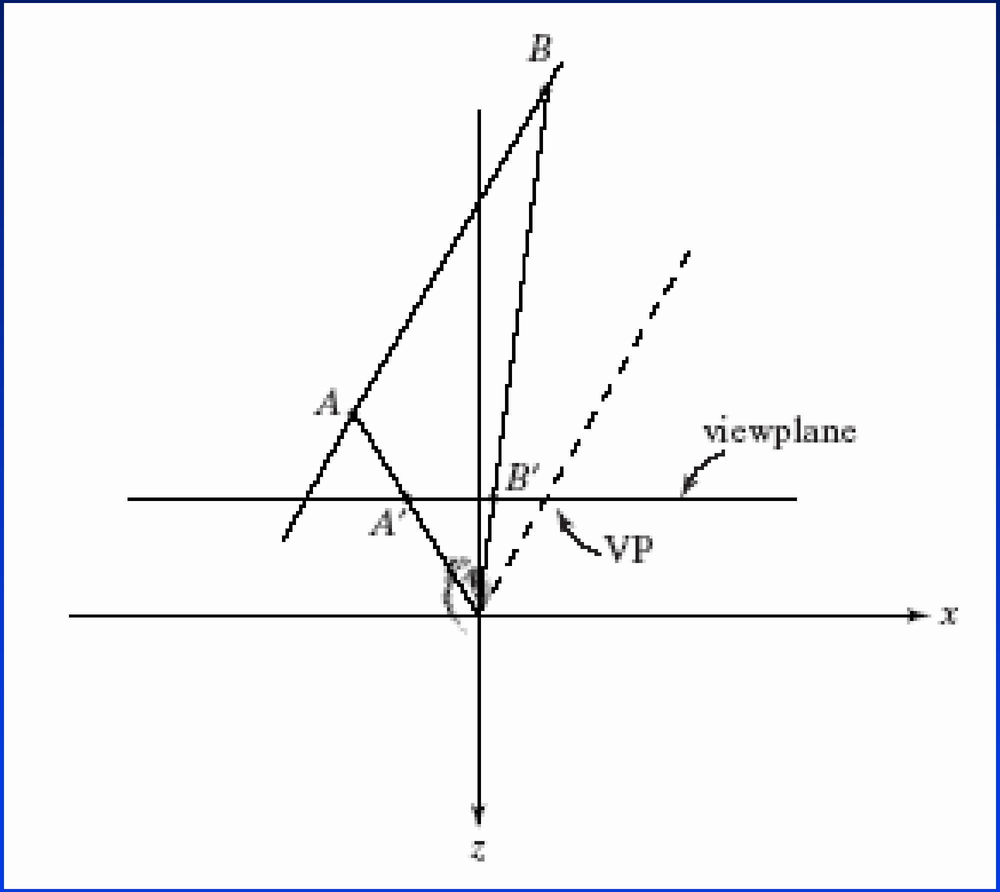
Thus all parallel lines share the same vanishing point.

In particular, these lines project to lines that are not parallel.

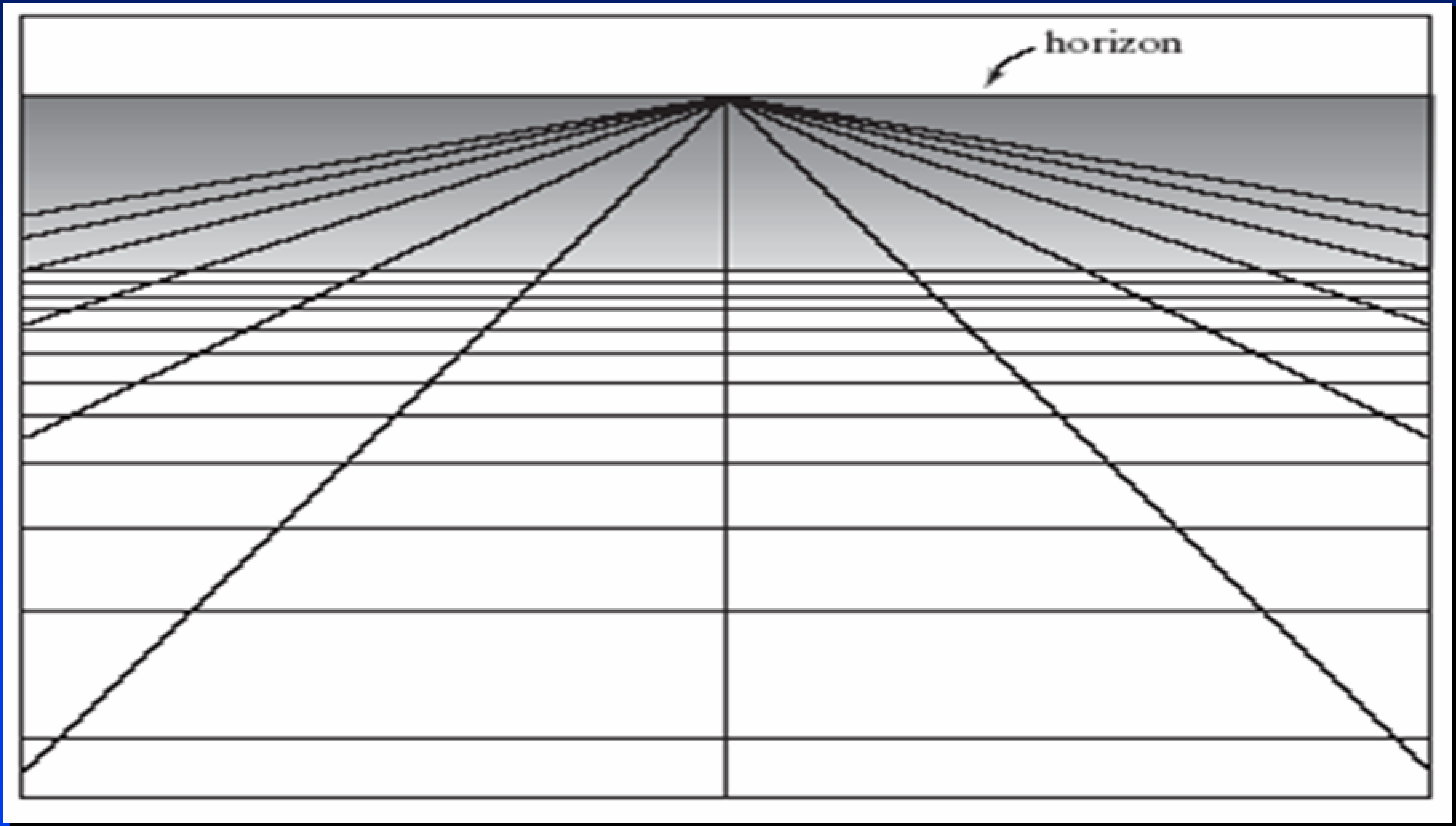


Projection of Straight Lines (≤)

- *Geometry of vanishing point: A projects to A', B projects to B', etc. Very remote points on the line project to VP as shown.*
- *Line from eye to VP becomes parallel to line AB.*



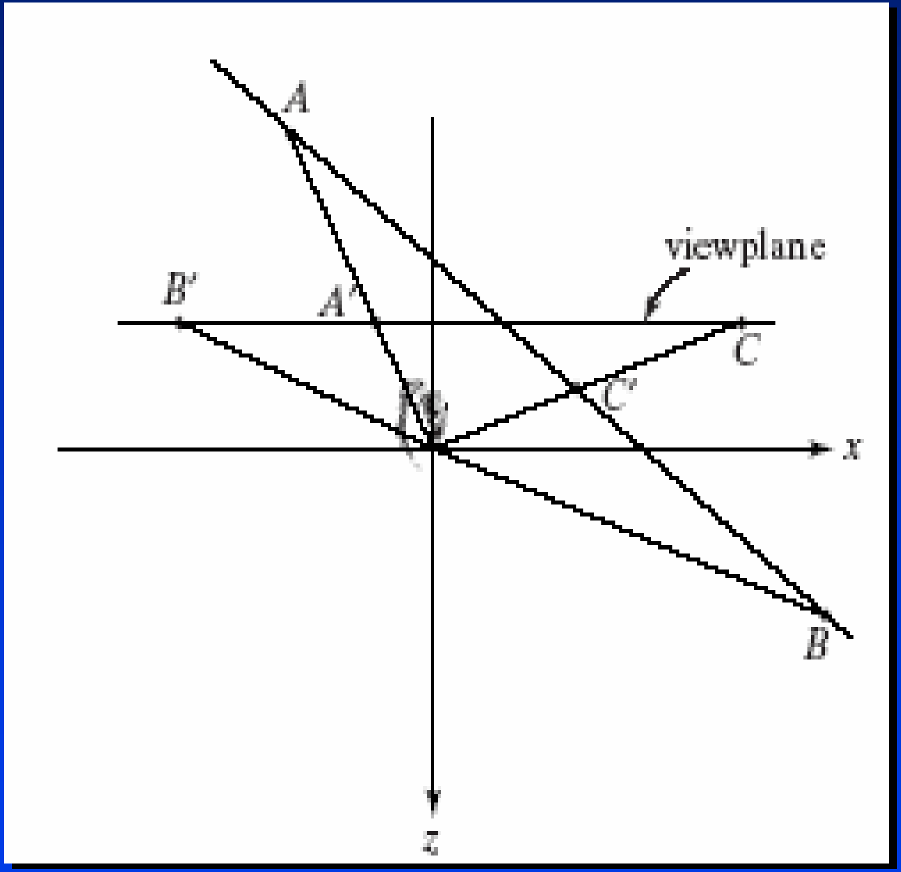
Example: horizontal grid in perspective



Projection of Straight Lines (5)

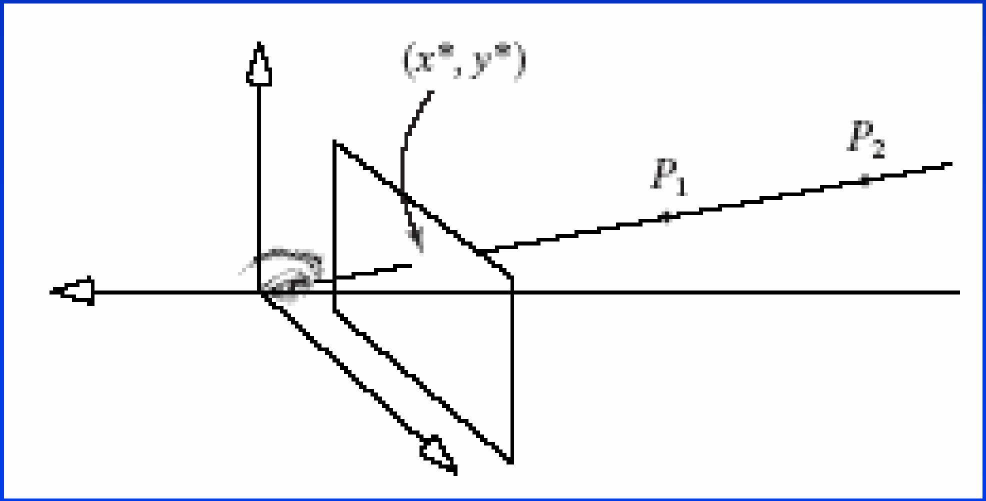
Lines that pass behind the eye have a different geometry for the vanishing point; as C approaches the eye plane, its projection moves infinitely far to the right.

When it reaches the eye plane, it jumps infinitely far to the left and starts moving right.



Incorporating Perspective in the Graphics Pipeline

- ***We need to add depth information (destroyed by projection).***
- ***Depth information tells which surfaces are in front of other surfaces, for hidden surface removal.***



Incorporating Perspective in the Graphics Pipeline (2)

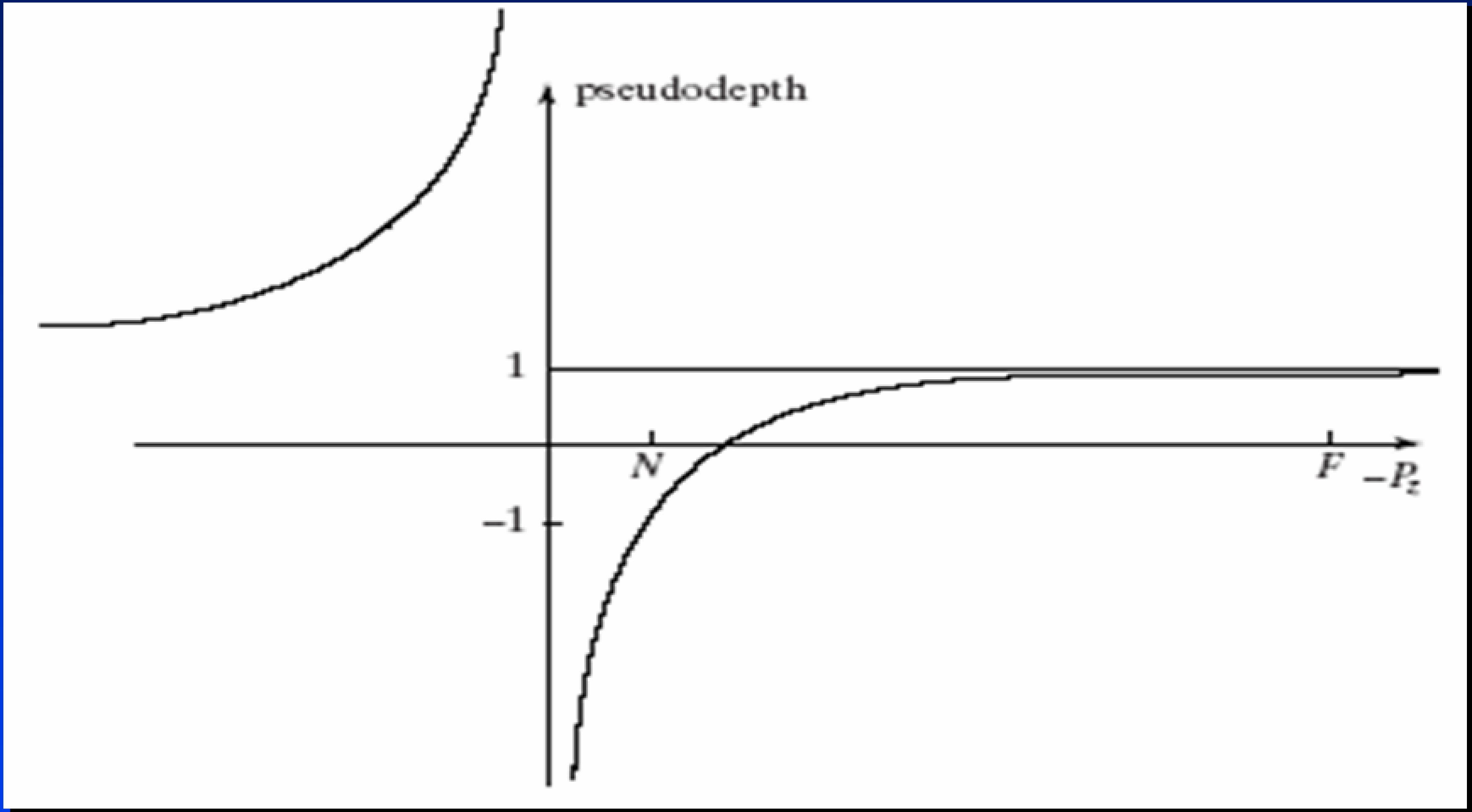
Instead of Euclidean distance, we use a pseudodepth, $-1 \leq P_z' \leq 1$ for $-N > z > -F$. This quantity is faster to compute than the Euclidean distance.

We use a projection point $(x^*, y^*, z^*) = [N/(-P_z)][NP_x, NP_y, N(a + bP_z)]$, and choose a and b so that $P_z' = -1$ when $P_z = -N$ and 1 when $P_z = -F$.

Result: $a = -(F + N)/(F - N)$, $b = -2FN/(F - N)$.

P_z' increases (becomes more positive) as P_z decreases (becomes more negative, moves further away).

Illustration of Pseudo-depth Values



Incorporating Perspective in the Graphics Pipeline (3)

Pseudodepth values bunch together as $-P_z$ gets closer to F, causing difficulties for hidden surface removal.

When N is much smaller than F, as it normally will be, pseudodepth can be approximated by

$$pseudodepth \approx 1 + \frac{2N}{P_z}$$

Perspective Transformation and Homogeneous Coordinates

We have used homogeneous coordinates for points and vectors. Now we want to extend them to work with perspective transformations.

Point P for any w ≠ 0 is given by

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

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

Perspective Transformation and Homogeneous Coordinates (2)

- *For example, the point (1, 2, 3) has the representations (1, 2, 3, 1), (2, 4, 6, 2), (0.003, 0.006, 0.009, 0.001), (-1, -2, -3, -1), etc.*
- *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 Homogeneous Coordinates (3)

Suppose we multiply a point in this new form by a matrix with the last row (0, 0, -1, 0). (The perspective projection matrix will have this form.)

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

Perspective Transformation and Homogeneous Coordinates (4)

The resulting point corresponds (after dividing through by the 4th component) to (x*, y*, z*) =

$$(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z+b}{-P_z})$$

Using homogeneous coordinates allows us to capture perspective using a matrix multiplication.

To make it work, we must always divide through by the fourth component, a step which is called *perspective division.*

Perspective Transformation and Homogeneous Coordinates (5)

- ***A matrix that has values other than (0,0,0,1) for its fourth row does not perform an affine transformation. It performs a more general class of transformation called a*** *perspective transformation**.***
- ***It is a transformation, not a projection. A projection reduces the dimensionality of a point, to a 3-tuple or a 2-tuple, whereas a perspective transformation takes a 4-tuple and produces a 4-tuple.***

Perspective Transformation and Homogeneous Coordinates (6)

Where does the projection part come into play?

- The first two components of this point are used for drawing: to locate in screen coordinates the position of the point to be drawn.
- The third component is used for depth testing.

As far as locating the point on the screen is concerned, ignoring the third component is equivalent to replacing it by 0; this is the projection part (as in orthographic projection, Ch. 5).

Perspective Transformation and Homogeneous Coordinates (7)

(perspective projection) = (perspective transformation) + (orthographic projection).

OpenGL does the transformation step separately from the projection step.

It inserts clipping, perspective division, and one additional mapping between them.

Perspective Transformation and Homogeneous Coordinates (7)

- ***When we wish to display a mesh model we must send thousands or even millions of vertices down the graphics pipeline.***
- ***Clearly it will be much faster if we can subject each vertex to a single matrix multiplication rather than to a sequence of matrix multiplications.***
- ***This is what OpenGL does: it multiplies all of the required matrices into a single matrix once and then multiplies each vertex by this combined matrix.***

Geometry of Perspective Transformation

•The perspective transformation alters 3D point P into another 3D point, to prepare it for projection. It is useful to think of it as causing a warping of 3D space and to see how it warps one shape into another.

•Very importantly, it preserves straightness and flatness, so lines transform into lines, planes into planes, and polygonal faces into other polygonal faces.

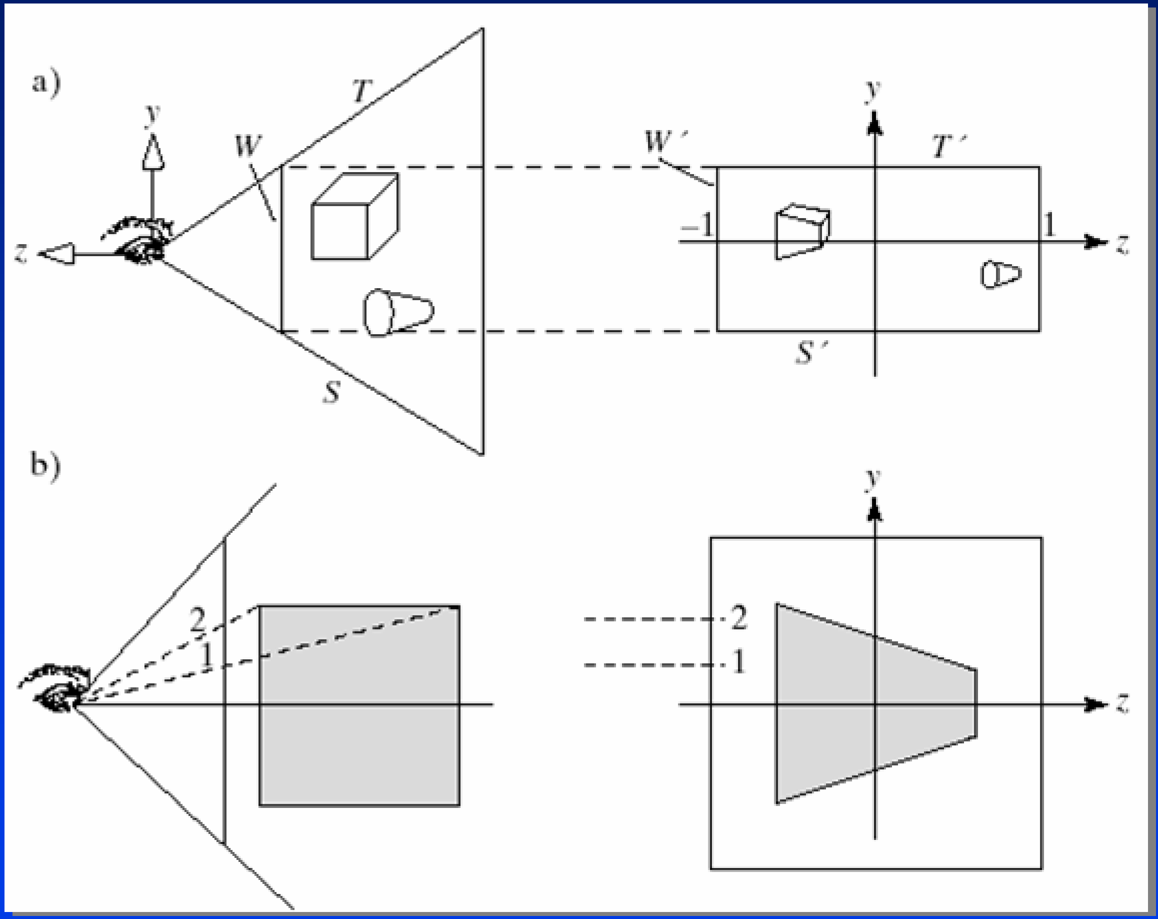
•It also preserves in-between-ness, so if point a is inside an object, the transformed point will also be inside the transformed object.

- Our choice of a suitable pseudodepth function was guided by the need to preserve these properties.

Geometry of Perspective Transformation (2)

How does it transform the camera view volume?

We must clip to this volume.



Geometry of Perspective Transformation (3)

- *The near plane W at z = -N maps into the plane W' at z = -1, and the far plane maps to the plane at z = +1.*
- *The top wall T is tilted into the horizontal plane T' so that it is parallel to the z-axis.*
- *The bottom wall S becomes the horizontal S', and the two side walls become parallel to the z-axis.*
- *The camera's view volume is transformed into a parallelepiped.*

Geometry of Perspective Transformation (4)

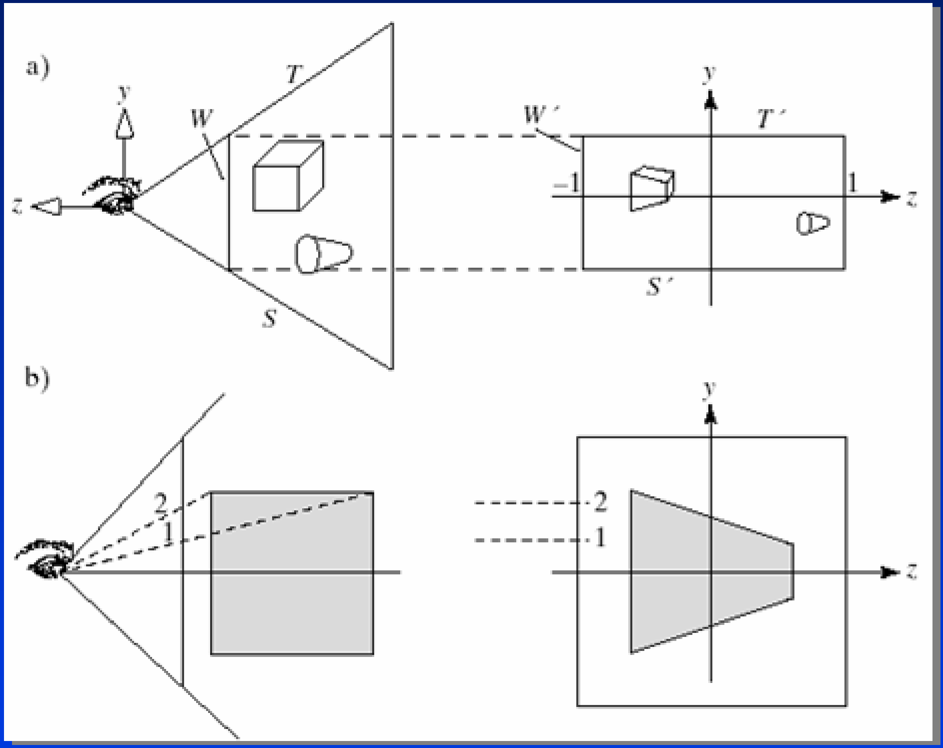
Lines through the eye map into lines parallel to the z-axis.

- **Proof**: All points of such a line project to a single point, say (x^*, y^*), on the viewplane. So all of the points along the line transform to all of the points (x, y, z) with $x = x^*$, $y = y^*$, and z taking on all pseudodepth values between -1 and 1.

Geometry of Perspective Transformation (5)

Lines perpendicular to the z-axis map to lines perpendicular to the z-axis.

- **Proof:** All points along such a line have the same z-coordinate, so they all map to points with the same pseudodepth value.



Geometry of Perspective Transformation (6)

The transformation also warps objects into new shapes.

The perspective transformation warps objects so that, when viewed with an orthographic projection, they appear the same as the original objects do when viewed with a perspective projection.

Geometry of Perspective Transformation (7)

- *We want to put some numbers on the dimensions of the view volume before and after it is warped.*
- *Consider the top plane, and suppose it passes through the point* (left, top, -N) *at* z = -N*.*
- *It is composed of lines that pass through the eye and through points in the near plane all of which have a y-coordinate of* top, so *it must transform to the plane* y = top*. Similarly,the bottom plane transforms to the* y = bott *plane; the left plane transforms to the* x = left *plane; and the right plane transforms to the* x = right *plane.*

Geometry of Perspective Transformation (8)

We now know the transformed view volume precisely: a parallelepiped with dimensions that are related to the camera's properties in a very simple way.

This is a splendid shape to clip against as we shall see, because its walls are parallel to the coordinate planes, but it would be even better for clipping if its dimensions didn't depend on the particular camera being used.

OpenGL composes the perspective transformation with another mapping that scales and translates this parallelepiped into the canonical view volume, a cube that extends from -1 to 1 in each dimension.

Because this scales things differently in the x- and y-dimensions, it introduces some distortion, but the distortion will be eliminated in the final viewport transformation.

Perspective Projection Matrix used by OpenGL

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

Perspective Projection Matrix (2)

Recall that gluPerspective(viewAngle, aspect, N, F) is usually used instead, as its parameters are more intuitive.

gluPerspective() sets up the same matrix, after computing values for **top, bott,** ***etc. using***

$$top = N\tan\left(\frac{\pi}{180} viewAngle / 2\right)$$

bott = -top, right = top * aspect, ***and*** **left = -right.**

The Graphics Pipeline in OpenGL

model in world coordinates → modelview matrix (eye coordinates) → projection matrix (canonical view volume coordinates) → clipper → perspective division → viewport matrix →display (screen coordinates)

The Graphics Pipeline in OpenGL (2)

•*Following clipping* perspective division *is finally done and the 3-tuple (x, y, z) is passed through the viewport transformation.*

•*The perspective transformation squashes the scene into the canonical cube. If the aspect ratio of the camera's view volume (that is, the aspect ratio of the window on the near plane) is 1.5, there is obvious distortion introduced.*

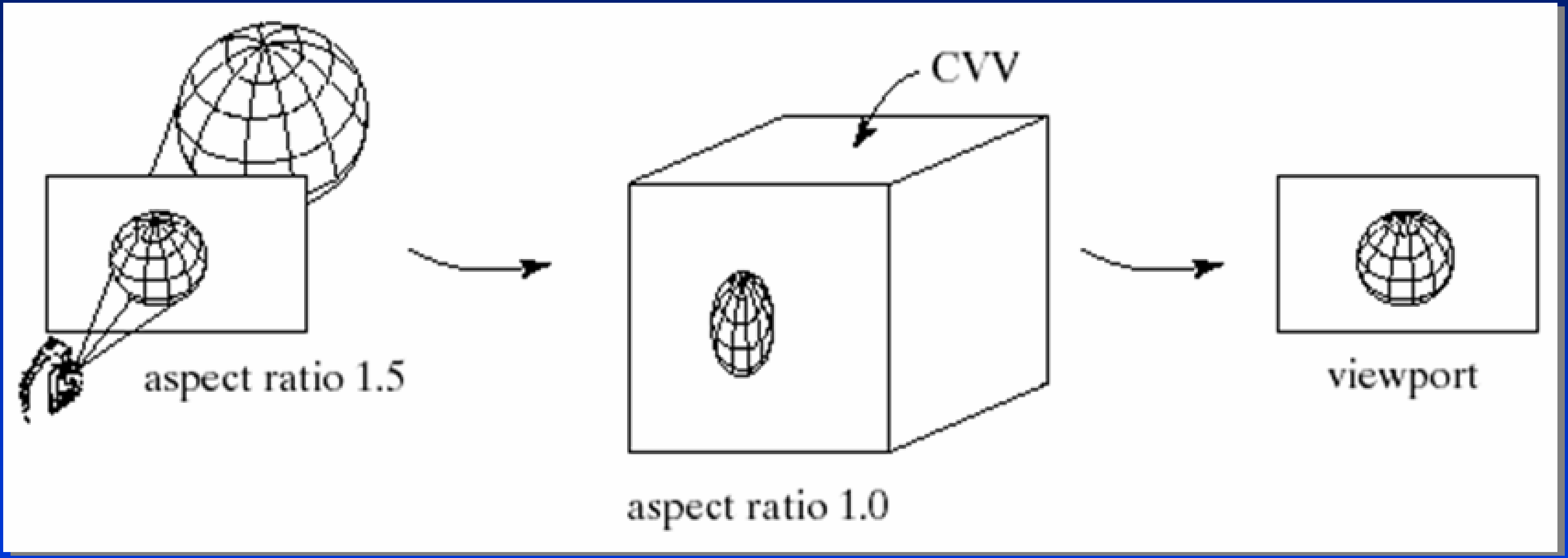
•*But the viewport transformation can undo this distortion by mapping a square into a viewport of aspect ratio 1.5. We normally set the aspect ratio of the viewport to be the same as that of the view volume.*

Distortion and Its Removal

glViewport(x, y, wid, ht) specifies that the viewport will have lower left corner (x,y) in screen coordinates and will be wid pixels wide and ht pixels high. It thus specifies a viewport with aspect ratio wid/ht.

The viewport transformation also maps pseudodepth from the range -1 to 1 into the range 0 to 1.

Distortion and Its Removal (2)



Steps in the Pipeline: Each Vertex P Undergoes Operations Below

- ***P is** extended* ***to a homogeneous 4-tuple by appending a 1, and this 4-tuple is multiplied by the** modelview matrix**, producing a 4-tuple giving the position in eye coordinates.***
- ***The point is then multiplied by the** projection matrix**, producing a 4-tuple in clip coordinates.***
- ***The edge having this point as an endpoint is** clipped**.***
- *Perspective division* ***is performed, returning a 3-tuple.***
- ***The** viewport transformation **multiplies the 3-tuple by a matrix; the result (s_x, s_y, d_z) is used for drawing and depth calculations. (s_x, s_y) is the point in screen coordinates to be displayed; d_z is a measure of the depth of the original point from the eye of the camera.***