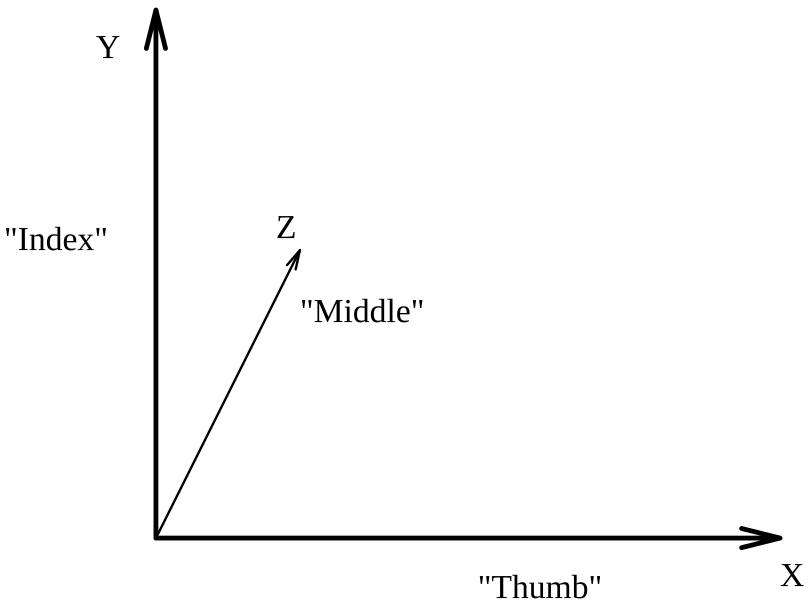
Three Dimensional Rotation and Translation



A “Left Handed” System.

Translation and Scaling

Translation:

$$(x', y', z', 1) = (x, y, z, 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{pmatrix}$$

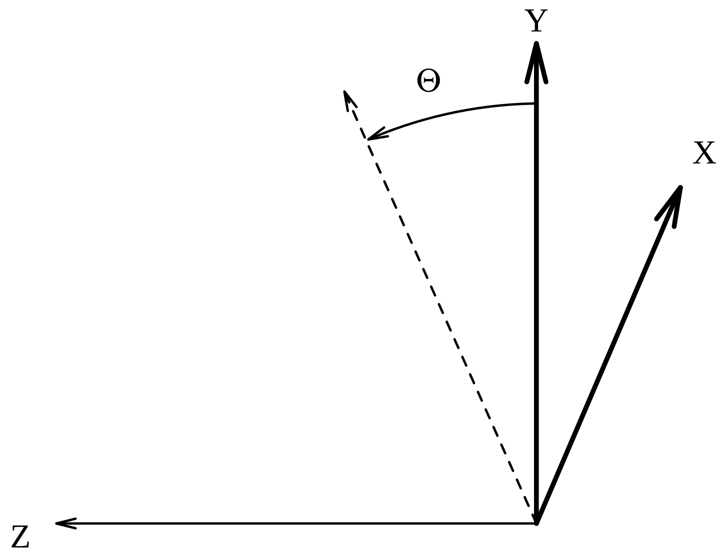
Scaling (from the origin):

$$(x', y', z', 1) = (x, y, z, 1) \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotations

Rotation counter-clockwise around the X axis by angle θ:

$$(x', y', z', 1) = (x, y, z, 1)\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

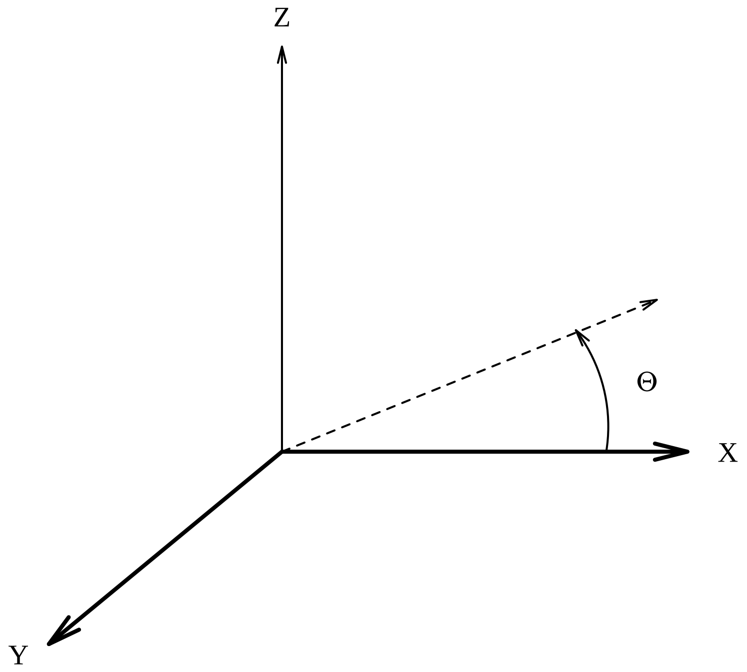


We are looking from $-\infty$ along the X axis.

Rotation Around y Axis

Rotation counter-clockwise around the Y axis by angle θ:

$$(x', y', z', 1) = (x, y, z, 1) \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

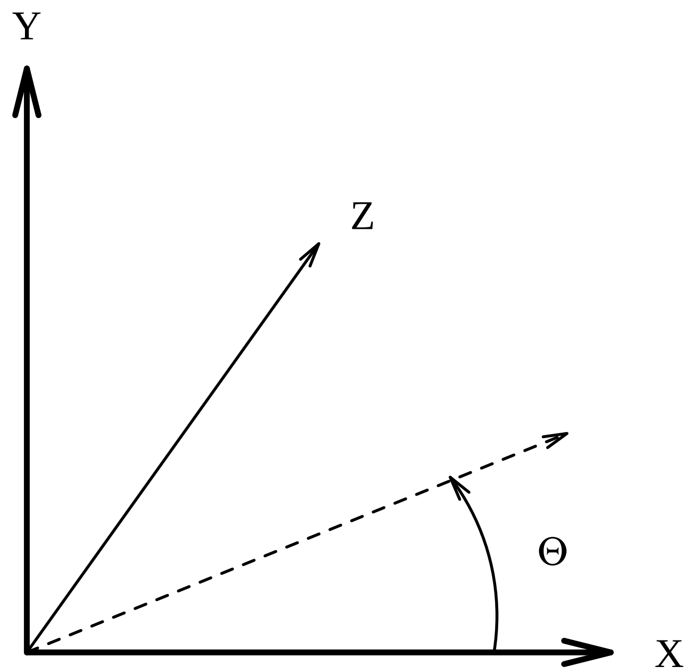


NOTE: We are looking from $+\infty$ along the Y axis.

Rotation Around z Axis

Rotation counter-clockwise around the Z axis by angle θ:

$$(x', y', z', 1) = (x, y, z, 1)\begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

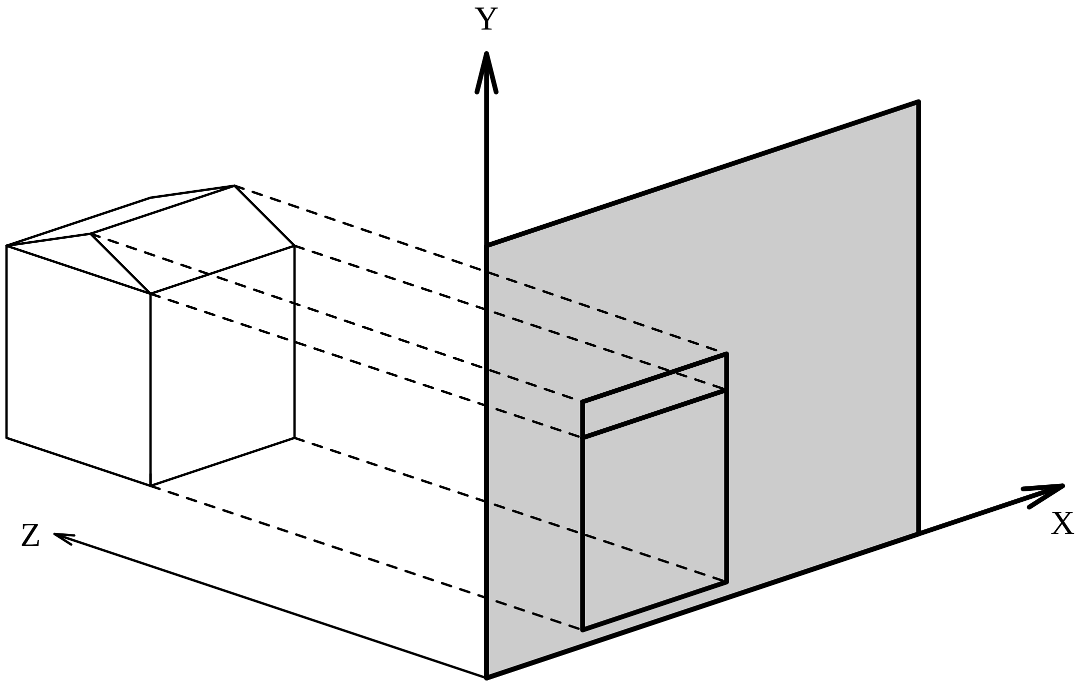


We are looking from $-\infty$ along the Z axis.

Projections

Nearly all output devices are two-dimensional (even 3-D goggles are built from two 2-D screens). To draw a 3-D scene, we must project it onto a two-dimensional viewing plane.

The simplest projection is **Orthographic Projection**



Orthographic Projection

We define a vector (x_p, y_p, z_p) along which we will project the image. Each scene point (x, y, z) is projected in parallel along this vector until it strikes the X-Y viewing plane at point (x_{pl}, y_{pl}).

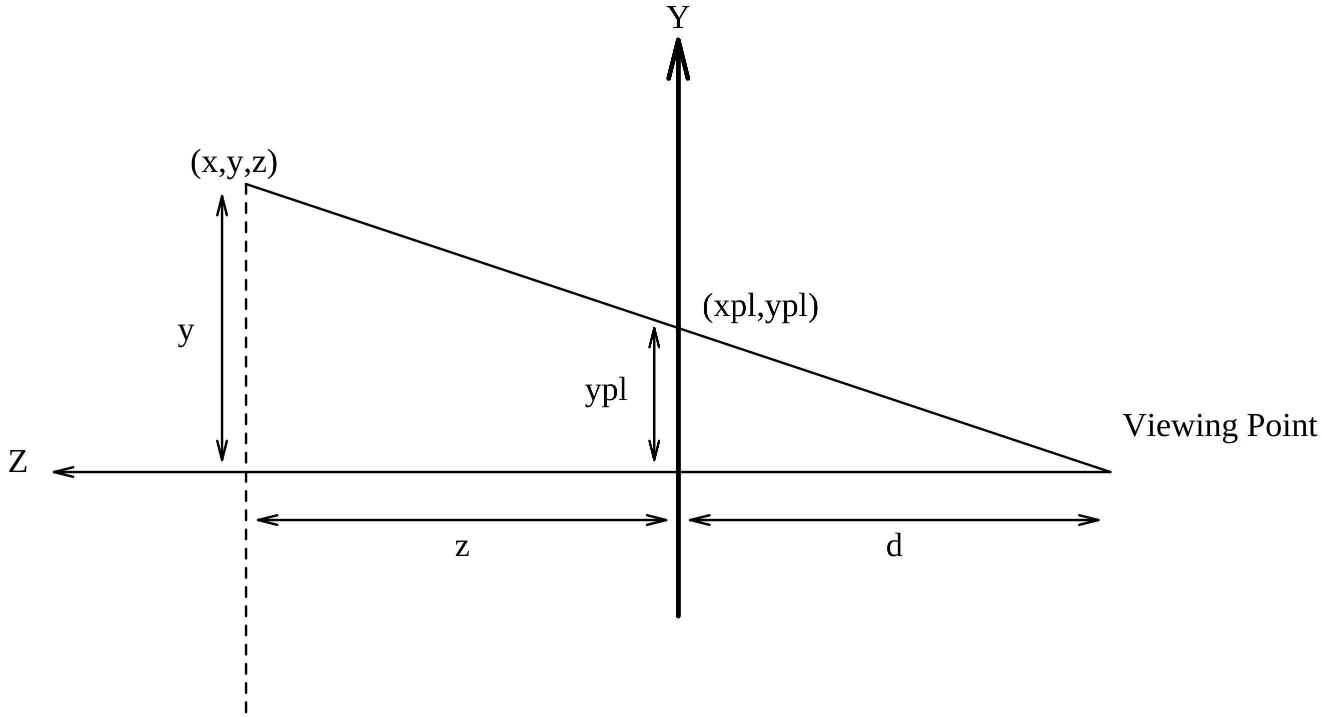
$$\begin{aligned} x_{pl} &= x - \frac{z}{z_p} x_p \\ y_{pl} &= y - \frac{z}{z_p} y_p \\ z_{pl} &= 0 \end{aligned}$$

When the projection vector is $(0, 0, -1)$ (i.e., projection along the $-Z$ axis), this reduces to

$$\begin{aligned} x_{pl} &= x \\ y_{pl} &= y \\ z_{pl} &= 0 \end{aligned}$$

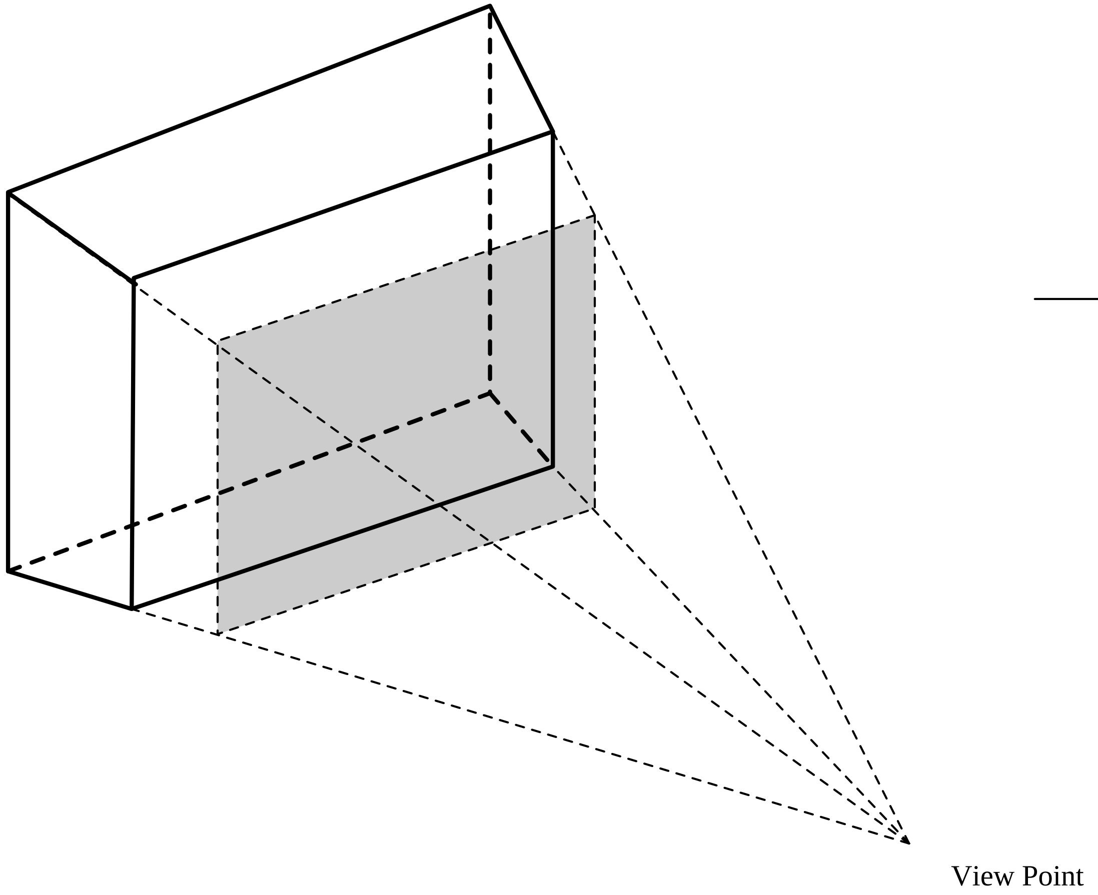
Perspective Projection

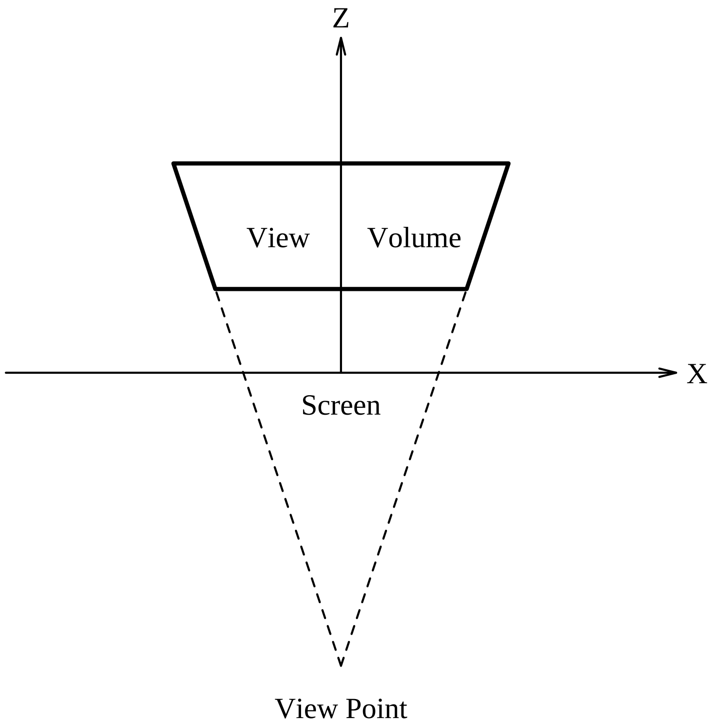
In our visual experience, three-dimensional objects are projected to a point—the center of our eye aperture. This produces the **Perspective Projection**.



$$x_{pl} = \frac{d \cdot x}{d + z} \qquad y_{pl} = \frac{d \cdot y}{d + z} \qquad z_{pl} = 0$$

Three-Dimensional Window = View Volume





We will always place the viewpoint at $(0, 0, -d)$.

Perspective Depth Transformation

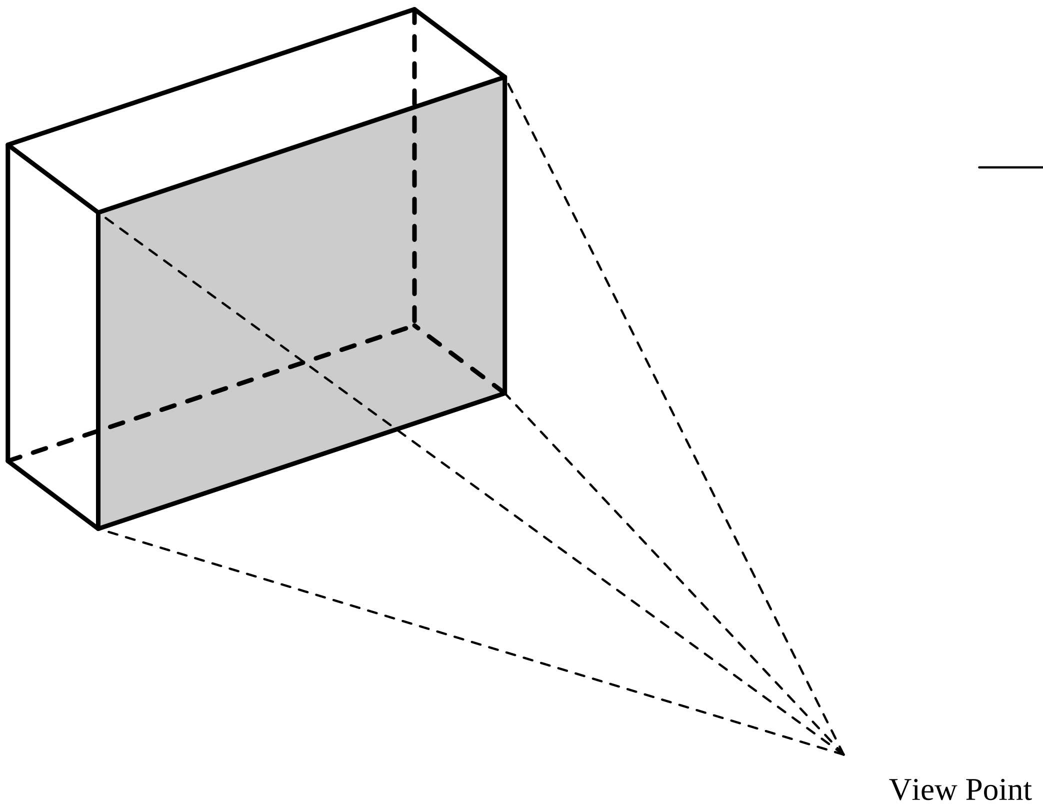
For hidden surface elimination, ray tracing, and many other algorithms, we need to keep **depth information** even after performing a perspective transformation. This is done by using the **Perspective Depth Transformation**:

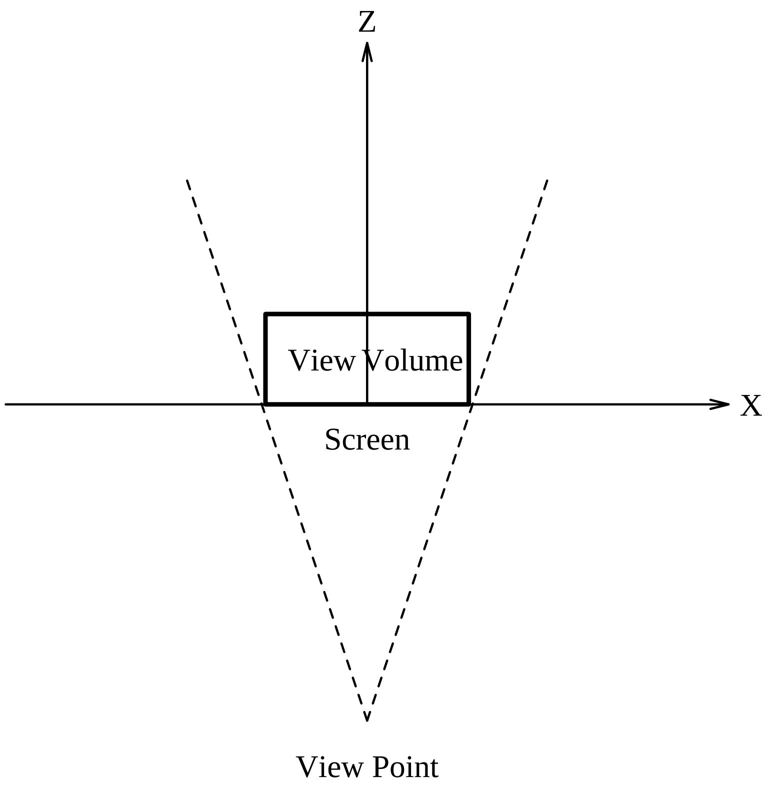
$$x_{pl} = \frac{d \cdot x}{d + z} \qquad y_{pl} = \frac{d \cdot y}{d + z} \qquad z_{pl} = \frac{z}{d + z}$$

- **Maps lines to lines**
- **Maintains depth ordering**
- **Converts the View Pyramid into a View Box**

Perspective Depth Transformation

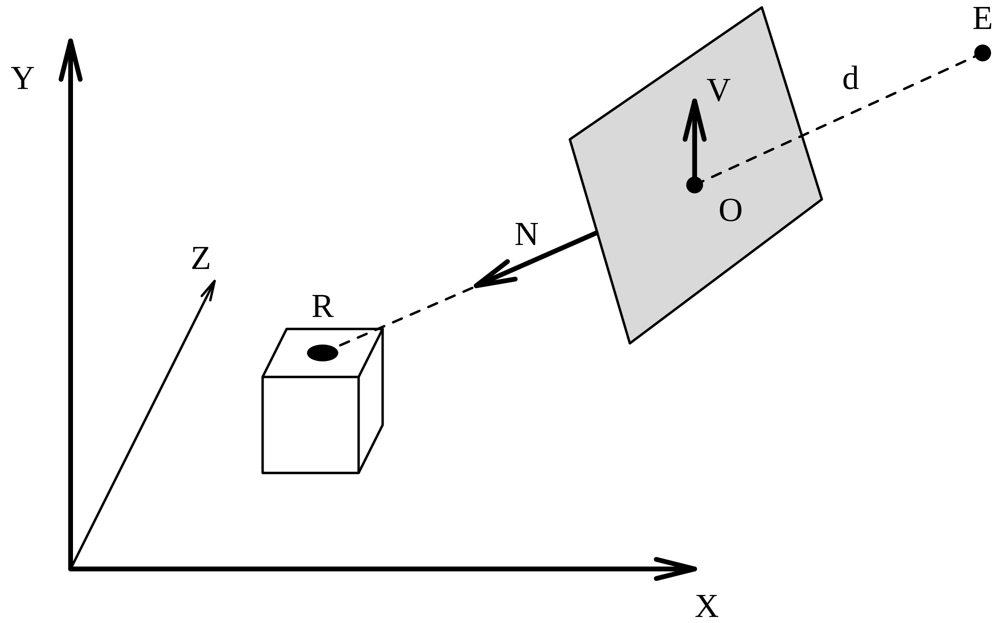
All Z coordinates are scaled to lie between 0 and 1.





Specifying a Direction from which to View the Scene

Given a scene, we must specify how we want to view the scene.



- **Camera Position**. Point E.
- **Object Point at Center of Window**. Point R.
- **Center of View Plane**. Point O.
- **"Up" direction**. Vector V.

Constructing a View Transformation

From this information, we want to construct a transformation that will convert any point (x, y, z) in the initial coordinate system to a new point (x', y', z') in the desired coordinate system. This is called the view transformation matrix, $VMAT$.

$VMAT = T \cdot M$, where T is a translation and M is a rotation and scaling component. T just translates O to the origin of the old system.

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -O_x & -O_y & -O_z & 1 \end{pmatrix}$$

Constructing the View Transformation (2)

Step 1: Compute N and normalize to get Z

$$
\begin{aligned}
N &= R - E \\
Z &= \frac{N}{\|N\|}
\end{aligned}
$$

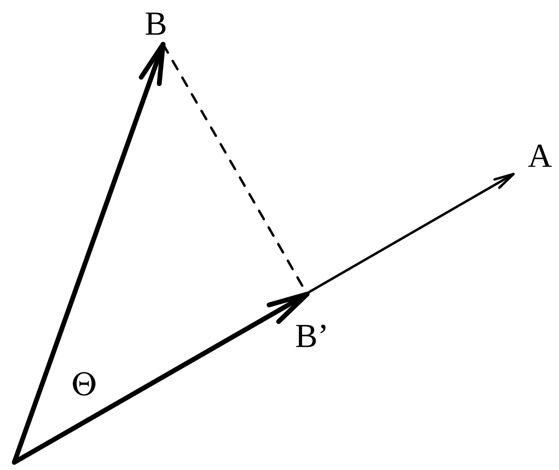
Step 2: Project V onto the view plane and normalize to get Y.

$$
\begin{aligned}
V' &= V - (Z \cdot V)Z \\
Y &= \frac{V'}{\|V'\|}
\end{aligned}
$$

Step 3: Compute X by cross product.

$$
X = Y \times Z
$$

Projection onto a Line



$$\|B'\| = \|B\| \cos\theta$$

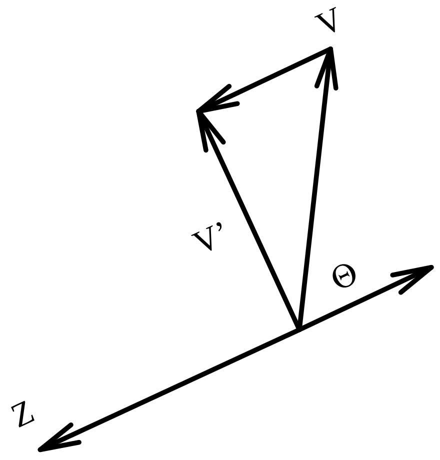
$$A \cdot B = \|A\| \|B\| \cos\theta$$

$$B' = \frac{A \cdot B}{\|A\|} \frac{A}{\|A\|}$$

In our case, A is a unit vector, Z, so this becomes

$$B' = (Z \cdot B)Z$$

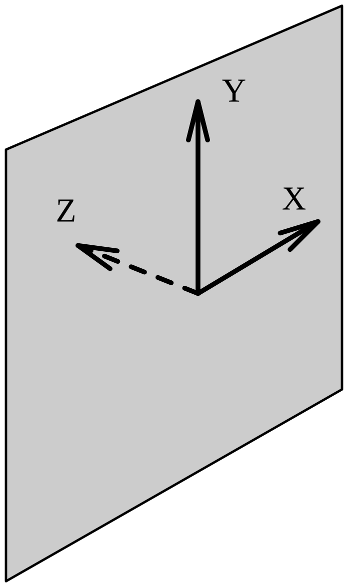
Computing the Y axis



$$V' = V - (Z \cdot V)Z$$

$$Y = \frac{V'}{\|V'\|}$$

Computing the X axis



$$X = Y \times Z$$

Final Rotation Matrix

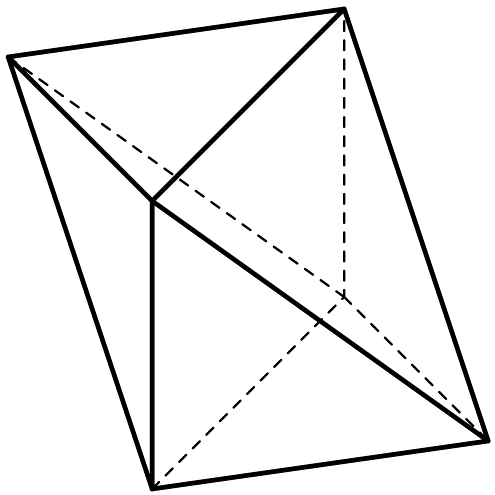
$$M = \begin{pmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$VMAT = T \cdot M$$

$$(x', y', z') = (x, y, z) \cdot VMAT$$

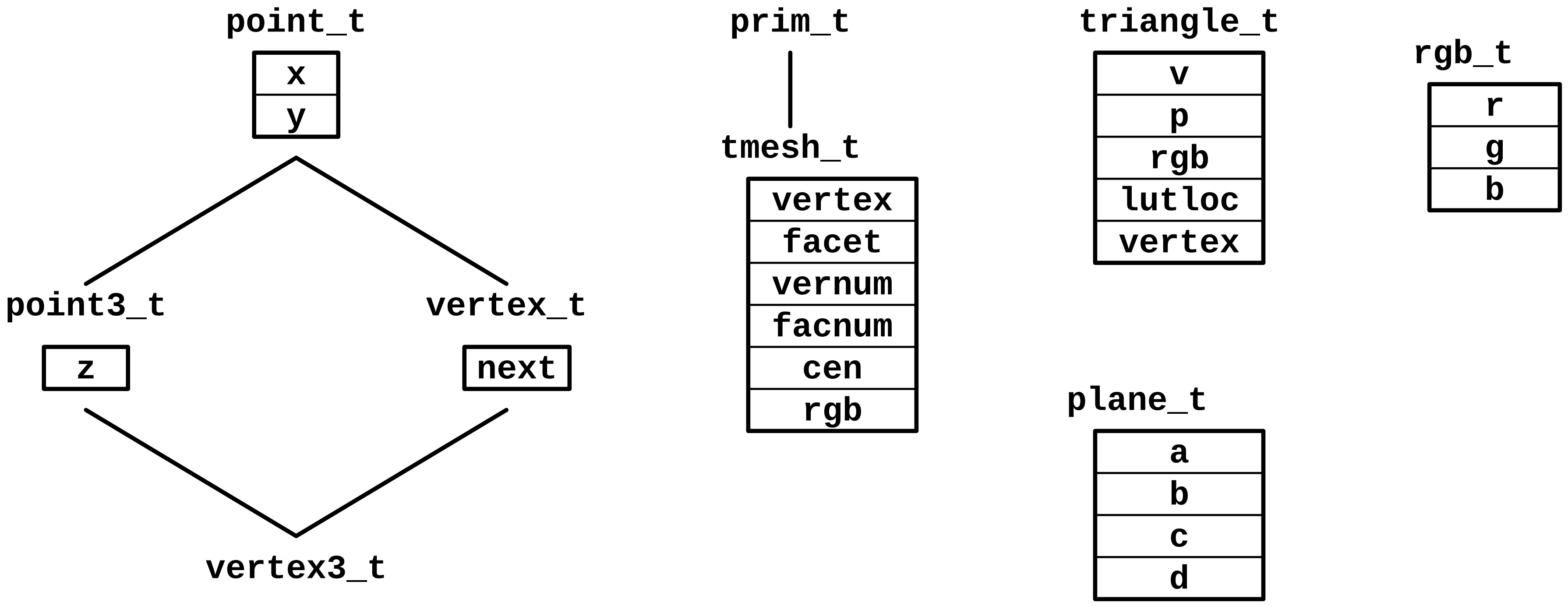
3-D Polyhedra

We represent polyhedra as polygon meshes:

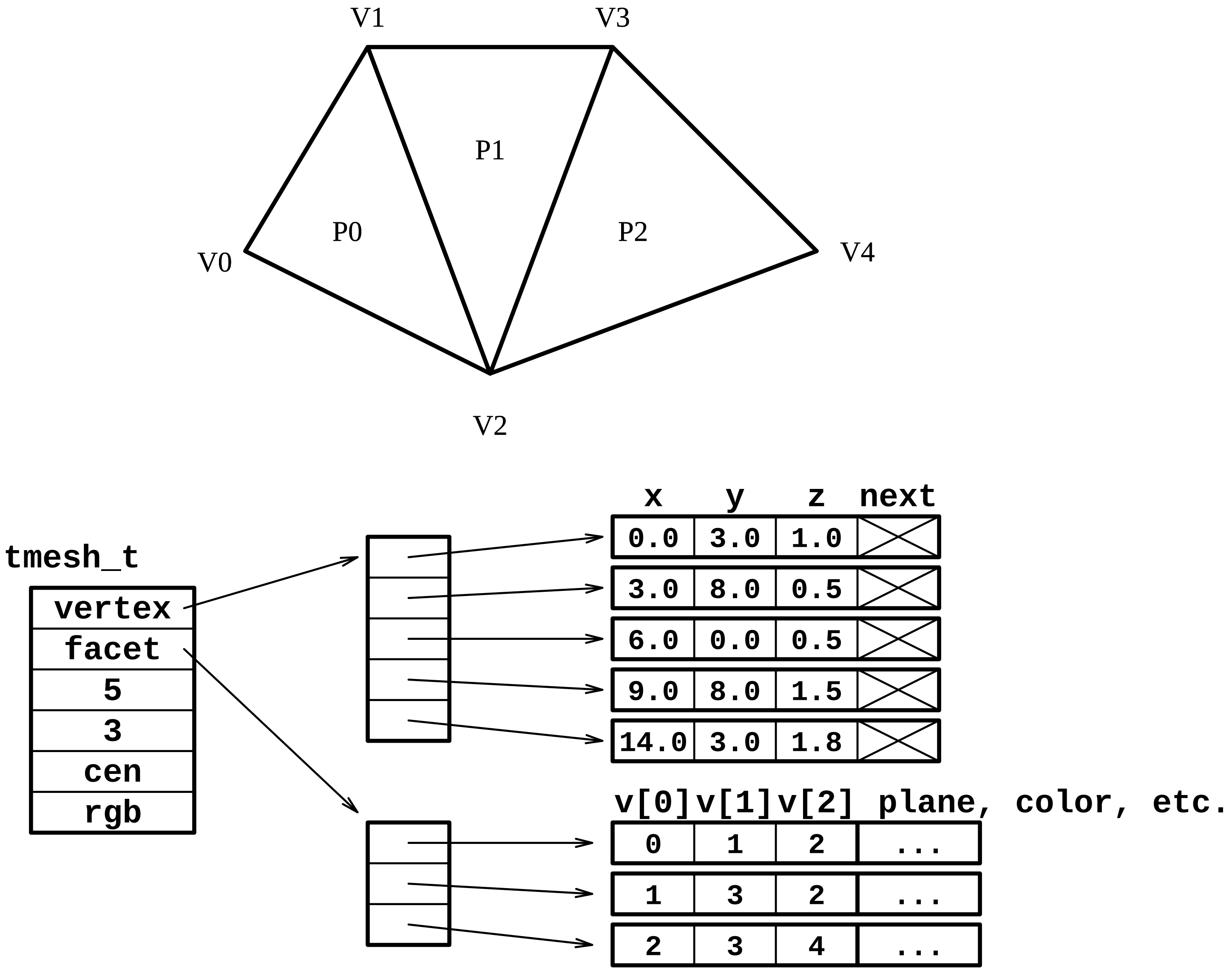


This has 6 vertexes and 8 faces. Each face is a 2D polygon.

Representing Triangular Meshes

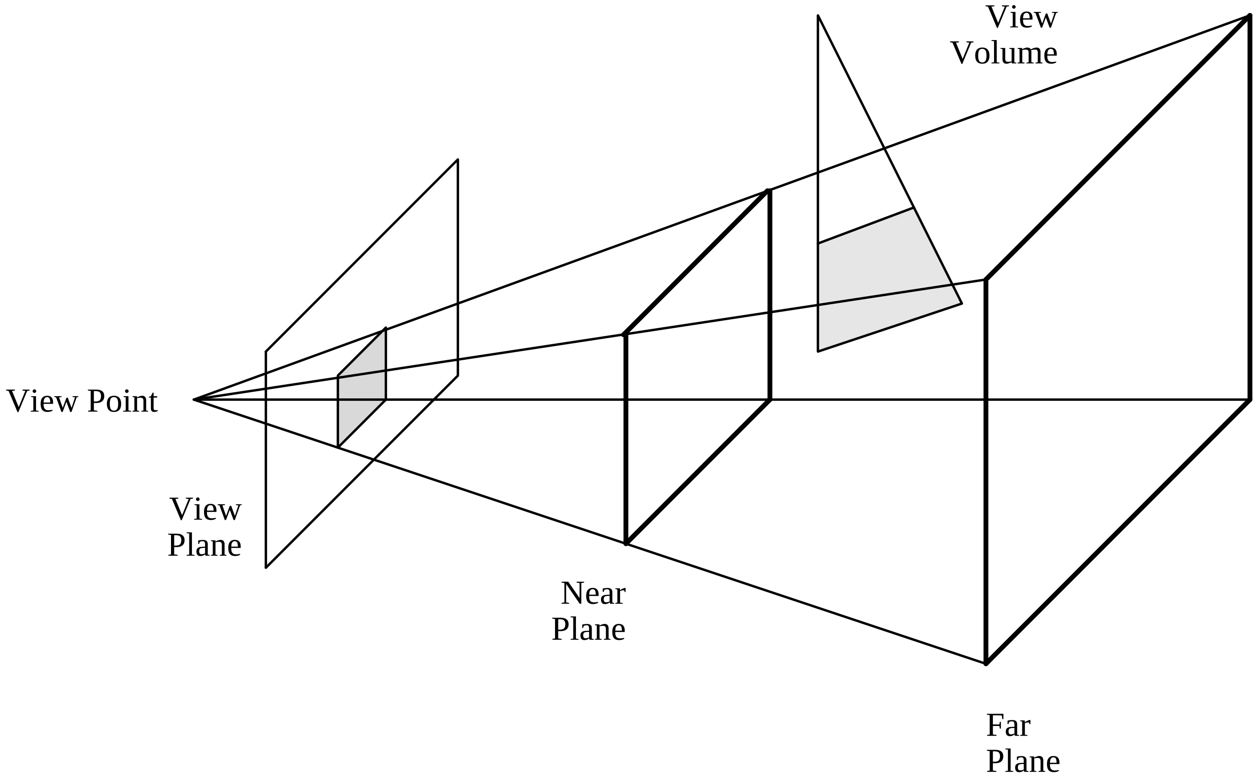


An Example Triangular Mesh (tmesh_t)

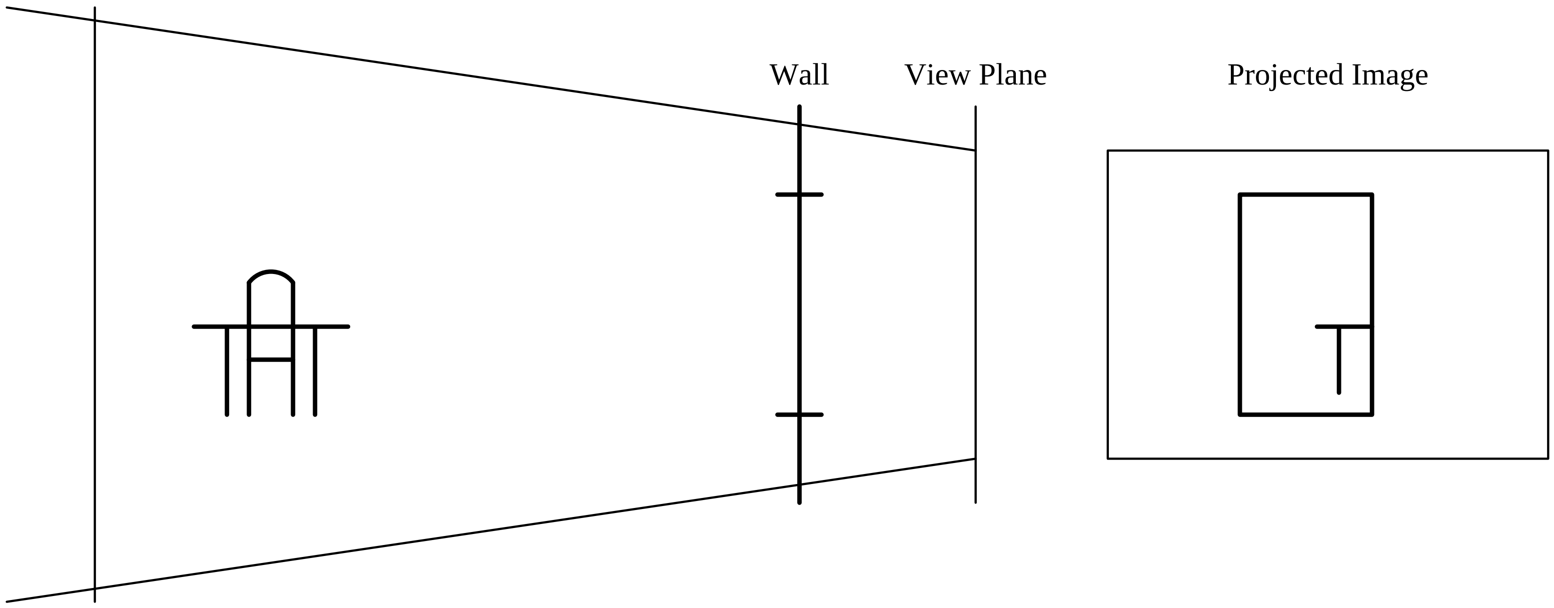


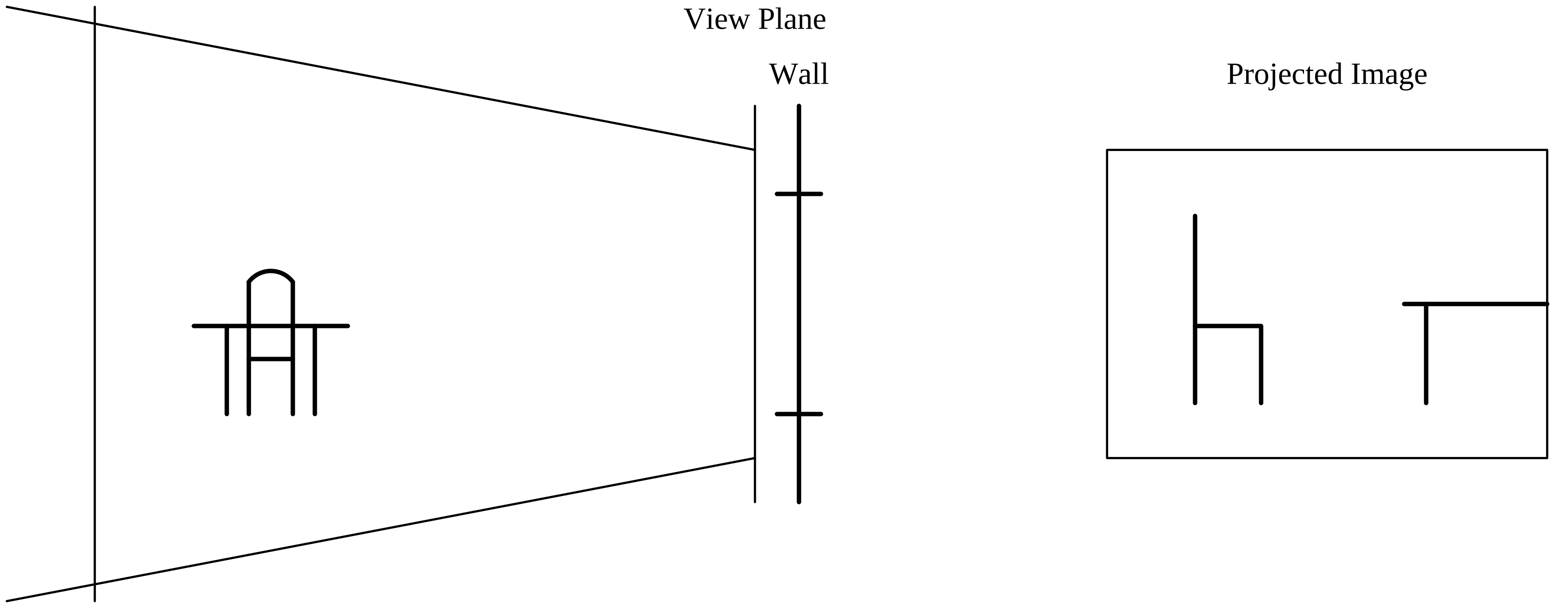
3-D Polygon Clipping

It is sufficient to clip each polygon against the view pyramid.



Why Clip Against Near and Far?

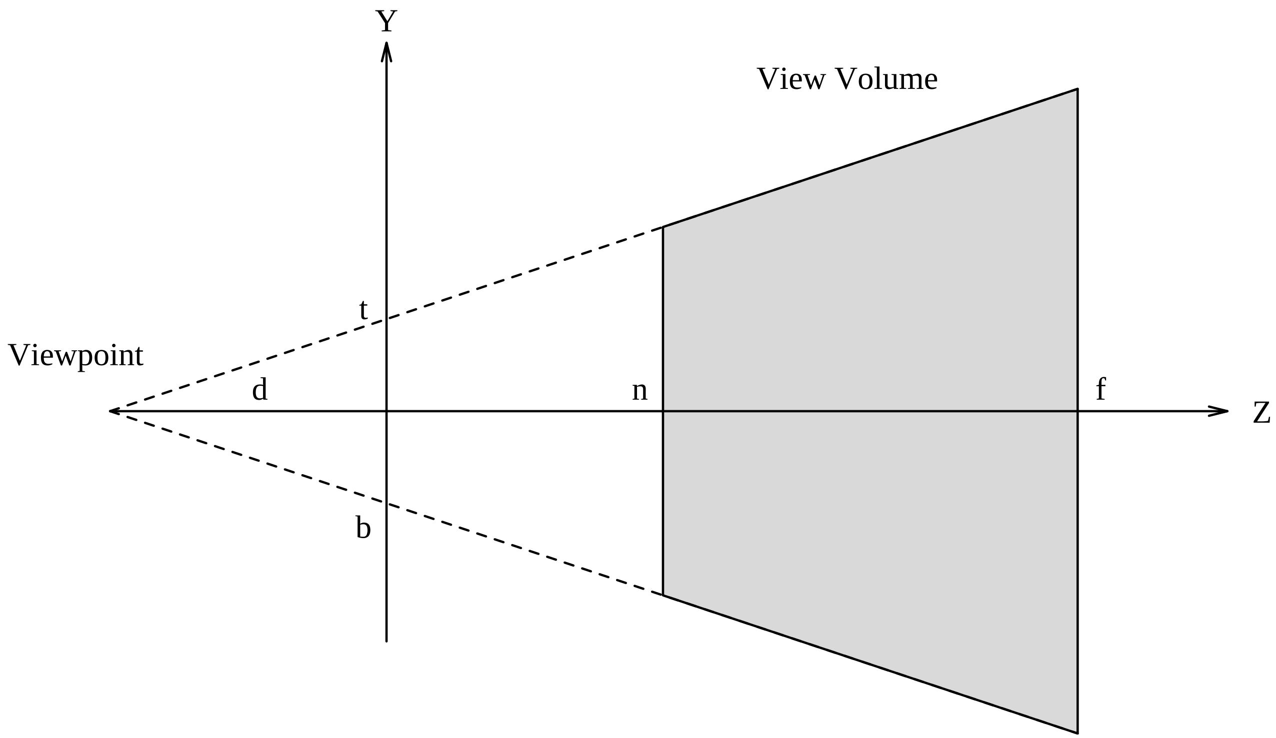




Clipping Against Pyramid Sides

Let l, r, b, t be the points where the sides of the view pyramid intersect the view plane. Let d be the distance from the origin to the view point. Then

slope of the left plane:	$s_L = -1/2(r-l)/d$
slope of the right plane:	$s_R = 1/2(r-l)/d$
slope of the bottom plane:	$s_B = -1/2(t-b)/d$
slope of the top plane:	$s_T = 1/2(t-b)/d$



Equations of the Sides of the View Pyramid

$$\begin{aligned}
L: &\qquad x = l + s_L z \\
R: &\qquad x = r + s_R z \\
B: &\qquad y = b + s_B z \\
T: &\qquad y = t + s_T z \\
N: &\qquad z = n \\
F: &\qquad z = f
\end{aligned}$$

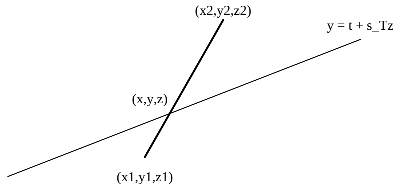
A line from (x_1, y_1, z_1) to (x_2, y_2, z_2) intersects the top plane at u value

$$u_T = \frac{y_1 - t - s_T z_1}{y_1 - y_2 + s_T(z_2 - z_1)}$$

so we compute the (x, y, z) point of intersection as

$$x = x_1 + u_T(x_2 - x_1) \qquad y = y_1 + u_T(y_2 - y_1) \qquad z = z_1 + u_T(z_2 - z_1)$$

Derivation



There are three equations:

$$
\begin{aligned}
y &= t + s_T z && (1)\\
y &= y_1 + u_T(y_2 - y_1) && (2)\\
z &= z_1 + u_T(z_2 - z_1) && (3)
\end{aligned}
$$

Substitute first into second and solve for z:

$$
\begin{aligned}
t + s_T z &= y_1 + u_T(y_2 - y_1)\\
z &= \frac{y_1 - t + u_T(y_2 - y_1)}{s_T}
\end{aligned}
$$

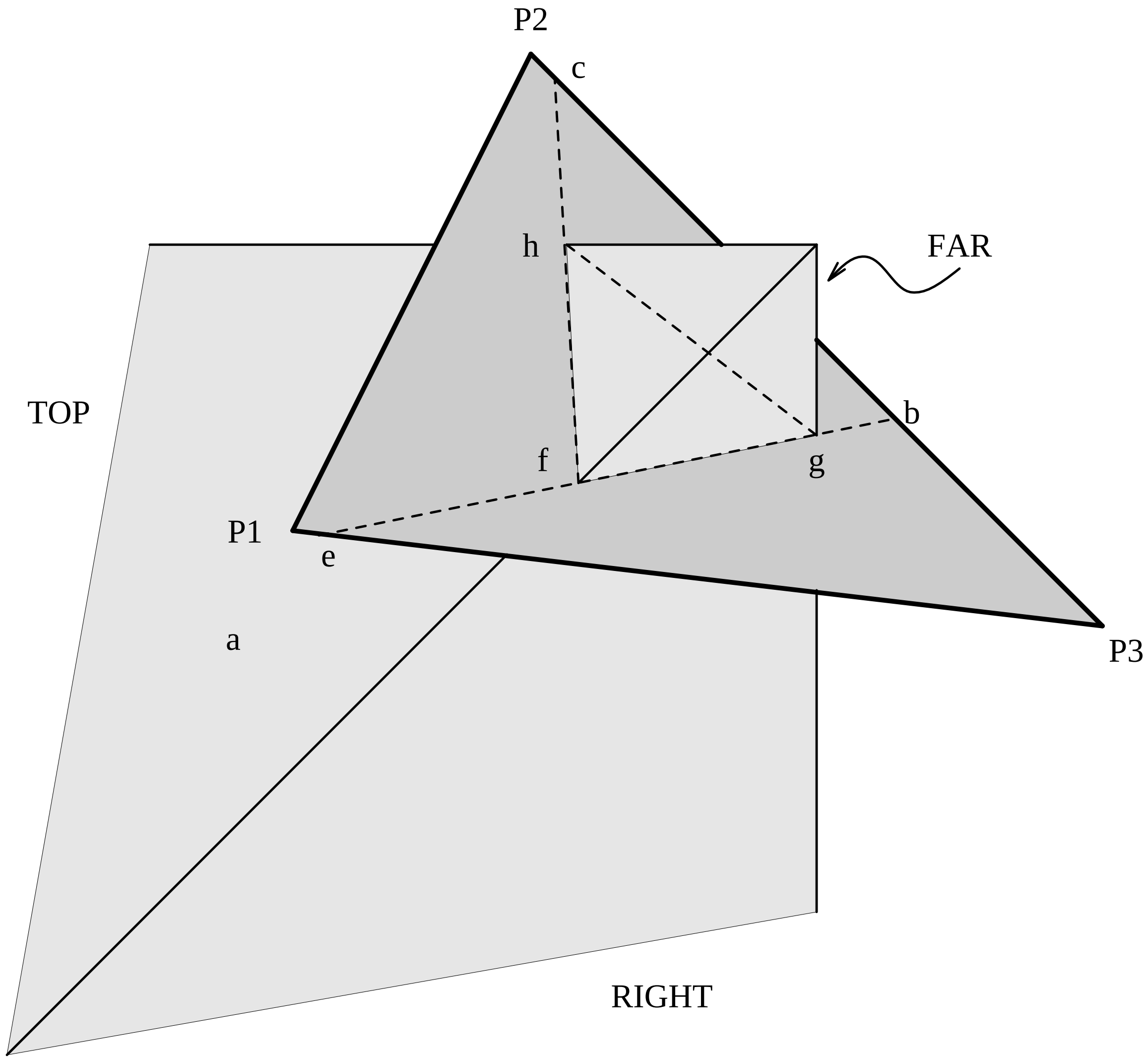
Substitute this into the third equation and collect terms:

$$
\begin{aligned}
\frac{y_1 - t + u_T(y_2 - y_1)}{s_T} &= z_1 + u_T(z_2 - z_1)\\
y_1 - t + u_T(y_2 - y_1) &= s_T z_1 + u_T s_T(z_2 - z_1)\\
u_T(y_2 - y_1) - u_T s_T(z_2 - z_1) &= s_T z_1 - y_1 + t\\
u_T &= \frac{s_T z_1 - y_1 + t}{(y_2 - y_1) - u_T s_T(z_2 - z_1)}
\end{aligned}
$$

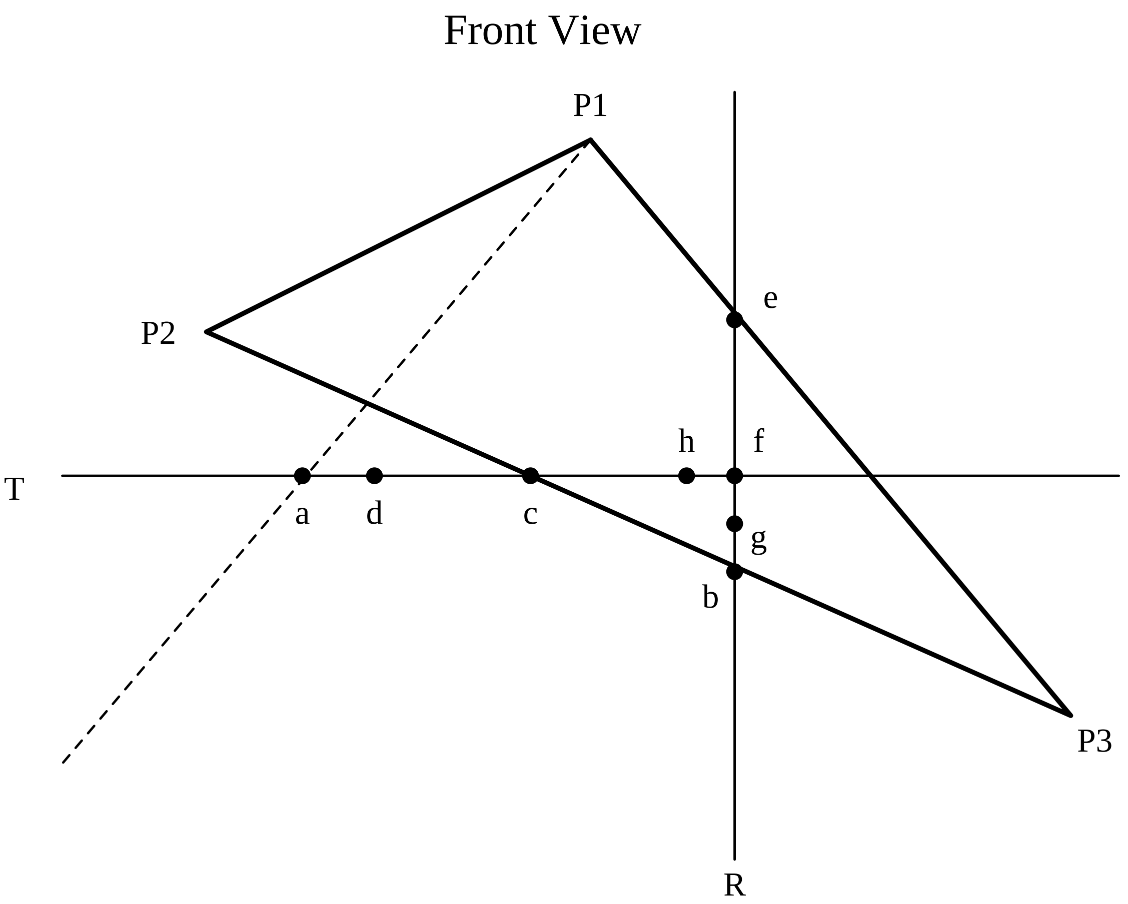
Negate numerator and denominator to get the result in the book.

3D Clipping Pipeline

The code is exactly analogous to the 2D pipeline with two more clippers: N (near) and F (far).

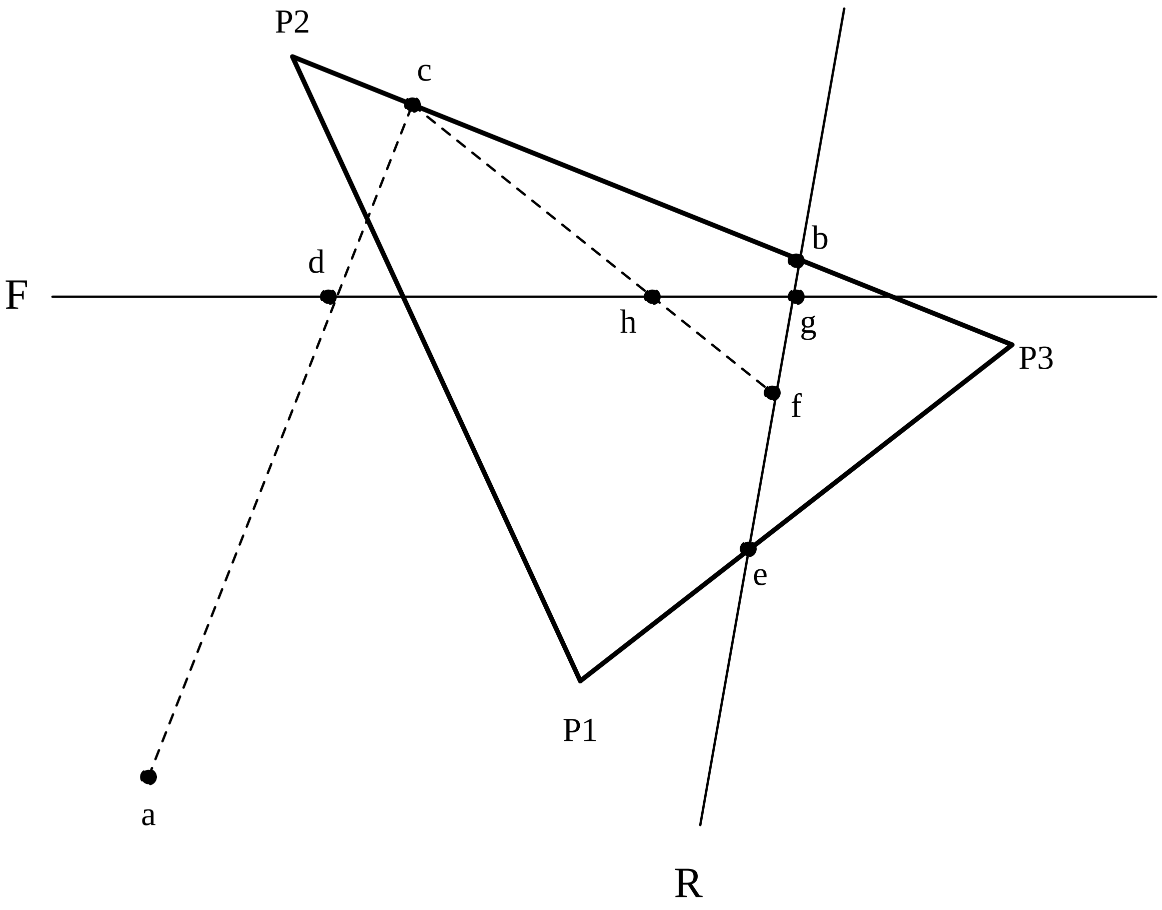


3D Clipping Pipeline (2)



3D Clipping Pipeline (3)

Top View



3D Clipping Pipeline (4)

First iteration:

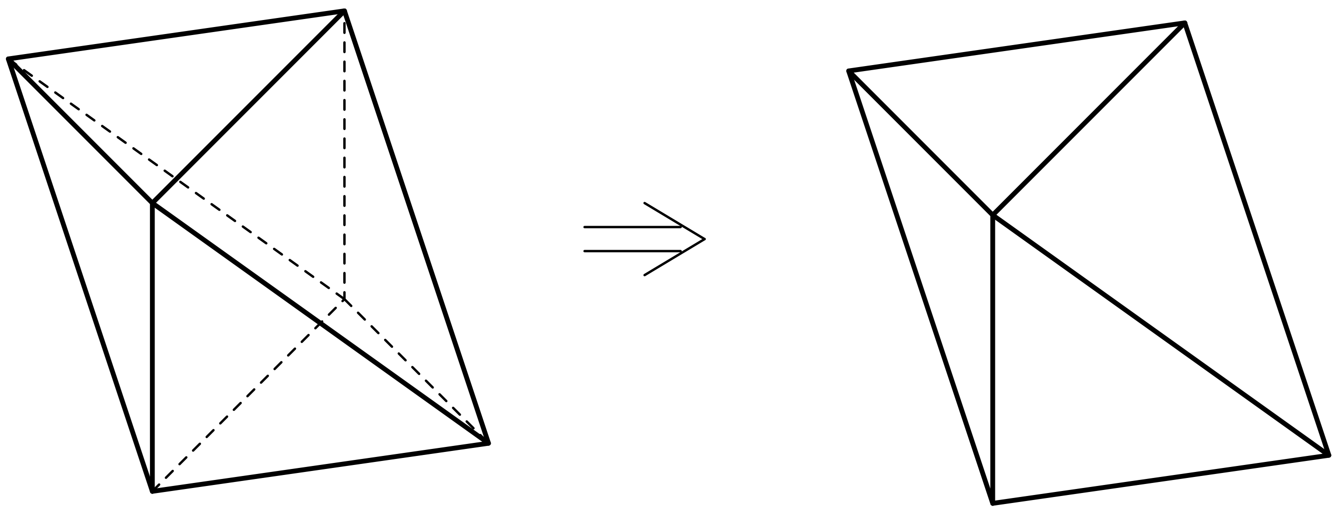
O		O		O		O		O		O		
P_1	L	P_1	R	P_1	B	P_1	T	a	N	a	F	a
P_2	L	P_2	R	P_2	B	P_2	T					
P_3	L	P_3	R	b	B	b	T	c	N	c	F	d
								b	N	b	F	

Second Iteration:

P_3		P_3		b		b		b		b		
P_1	L	P_1	R	e	B	e	T	f	N	f	F	g
												f
				P_1	B	P_1	T					
P_2	L	P_2	R	P_2	B	P_2	T					
P_3	L	P_3	R	b	B	b	T	c	N	c	F	h
								b	N	b	F	

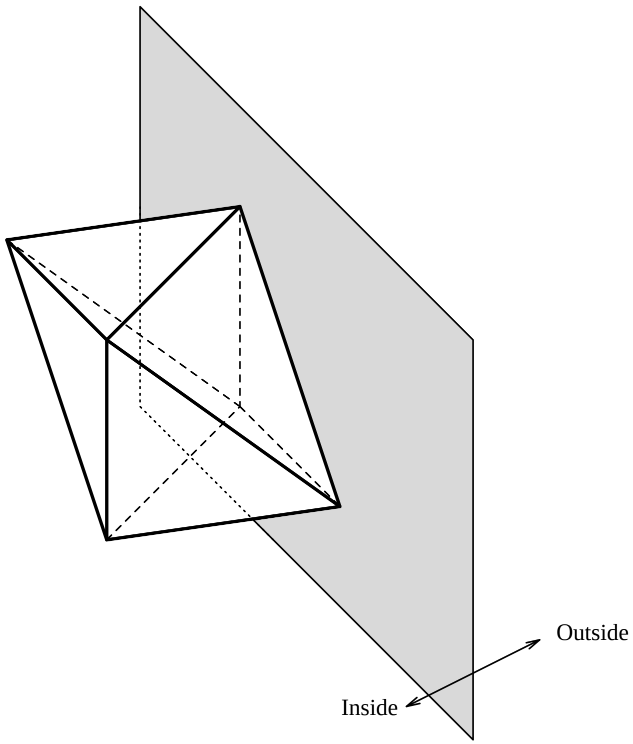
Backface Removal

Our main goal will now be to remove invisible parts of the scene before drawing it. A first step is **backface removal:**



All backfaces are invisible from the viewpoint.
Some frontfaces are also partially or totally invisible if they are blocked by other objects or other parts of a non-convex object.

Backface Removal Strategy



- **Step 1:** For each face, compute an equation for the plane that contains that face. The plane will divide all of space into "inside" (which will contain the object) and "outside".

- **Step 2:** Substitute the viewpoint $(0, 0, -d)$ into this equation. If the viewpoint is "outside", then the plane is visible from the viewpoint, and the face must be kept. Otherwise, the face is removed.

Equation of a Plane

The standard equation for a plane in 3D is

$$Ax + By + Cz - D = 0$$

where $\sqrt{A^2 + B^2 + C^2} = 1$.

The "normal vector" for this plane is (A, B, C), and the plane is distance D away from the origin.

For a given point, $P = (x, y, z)$, if

$Ax + By + Cz - D > 0$	P is on the positive side of the plane.
$Ax + By + Cz - D = 0$	P is in the plane.
$Ax + By + Cz - D < 0$	P is on the negative side of the plane.

If we construct the plane correctly, the backface test is simply

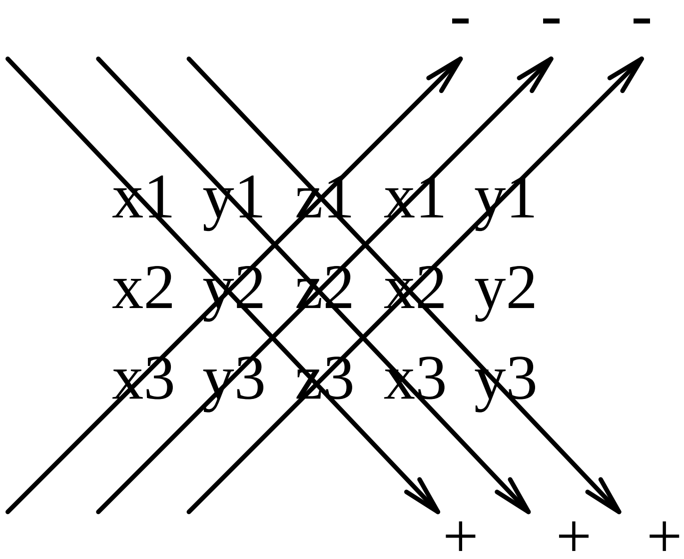
"Delete if $C \cdot (-d) - D < 0$."

Constructing a Plane Equation from each Face

- Position yourself so that you are looking at the face from **the outside.**
- Choose three vertexes of the face in **clockwise order**. Call them P_1, P_2, and P_3.
- Let $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2, z_2)$, and $P_3 = (x_3, y_3, z_3)$. Then the coefficients in the plane equation are computed by the four determinants:

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \quad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix} \quad C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \quad D = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

Mnemonic for Computing Determinants



$$\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} = x_1y_2z_3 + y_1z_2x_3 + z_1x_2y_3 - x_3y_2z_1 - y_3z_2x_1 - z_3x_2y_1$$

C++ Code (from `plane.c`)

```
plane_t::plane_t(const point3_t& a,const point3_t& b,const point3_t& c)
{                       // construct a plane equation from three points
   this->a = b.givy()*c.givz() + c.givy()*a.givz() + a.givy()*b.givz()
           - c.givy()*b.givz() - a.givy()*c.givz() - b.givy()*a.givz();
   this->b = a.givx()*c.givz() + b.givx()*a.givz() + b.givz()*c.givx()
           - a.givz()*c.givx() - b.givx()*c.givz() - b.givz()*a.givx();
   this->c = a.givx()*b.givy() + b.givx()*c.givy() + c.givx()*a.givy()
           - a.givx()*c.givy() - b.givx()*a.givy() - c.givx()*b.givy();
   this->d = a.givx()*b.givy()*c.givz() + b.givx()*c.givy()*a.givz()
           + a.givy()*b.givz()*c.givx() - a.givz()*b.givy()*c.givx()
           - a.givy()*b.givx()*c.givz() - b.givz()*c.givy()*a.givx();
   double l = sqrt(this->a*this->a + this->b*this->b + this->c*this->c);
   this->a /= l;
   this->b /= l;
   this->c /= l;
   this->d /= l;
}
int plane_t::frontface(const point3_t& p) const
{                                // true if plane is frontface for p
   return  p.givx()*a + p.givy()*b + p.givz()*c - d > 0;
}
```

Backface Removal (from `tmesh.c`)

```
void tmesh_t::draw(const scene_t* s) const
{
   for (int i = 0; i < facnum; i++)
     if (facet[i].givp().frontface(                    // frontface test
              point3_t(s→givwv().givcen(),−s→givwv().givd())))
        facet[i].draw(s);
}
```