COMP3421

Introduction to 3D Graphics

3D coordinates

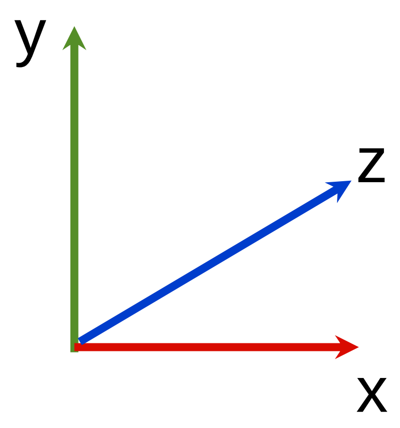
Moving to 3D is simply a matter of adding an extra dimension to our points and vectors:

$$P = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} \quad \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix}$$

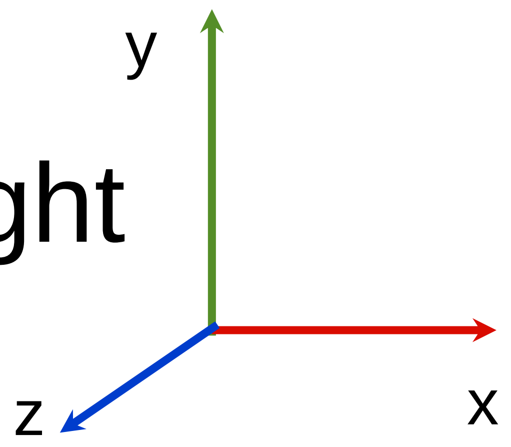
3D coordinates

3D coordinate systems can be left or right handed.

Left

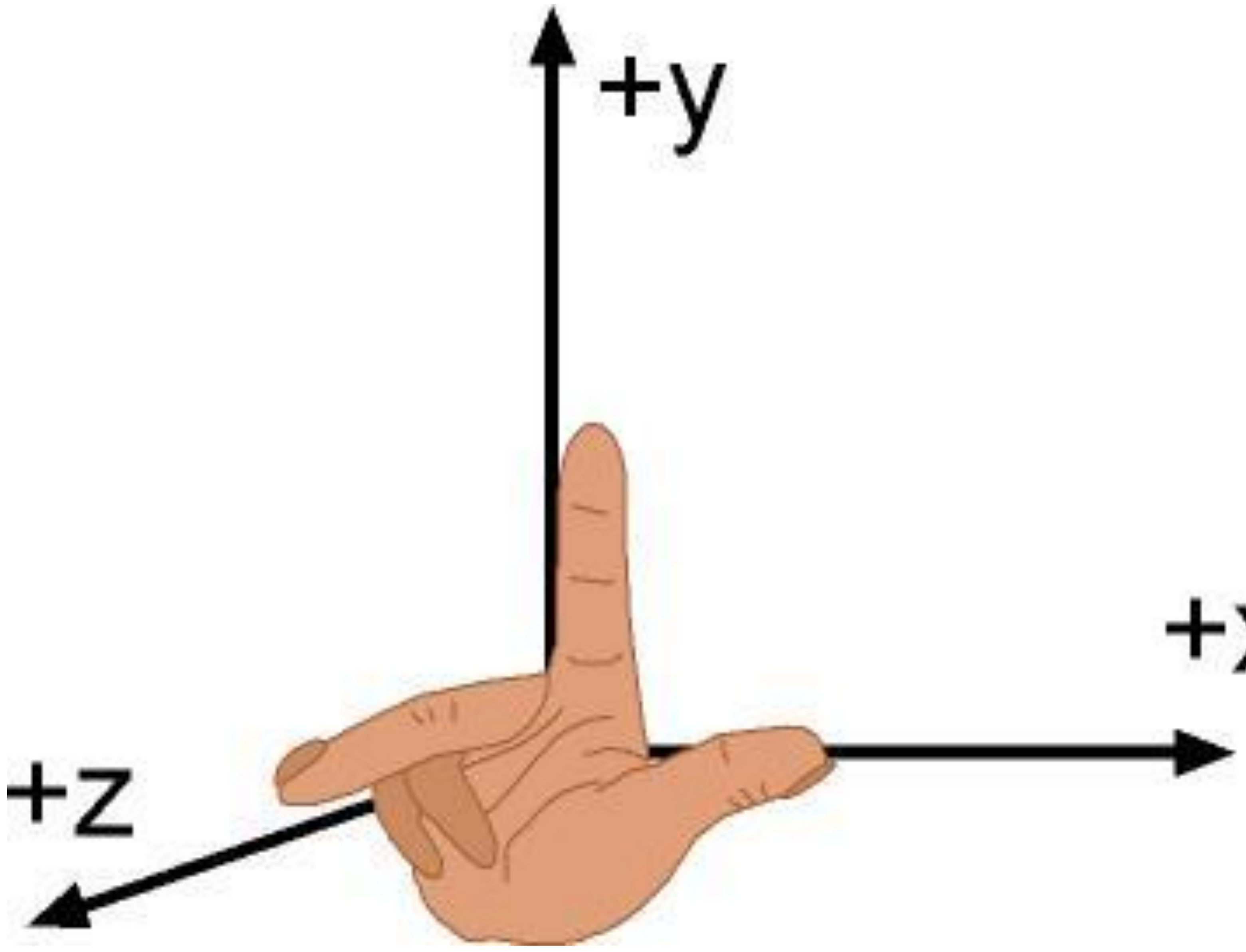


Right



We typically use right-handed systems, but left handed ones can arise (eg, if one of the axes has negative scale).

Right Handed Coordinate System



Depth

Let’s try First3DExample.java

By default OpenGL draws objects in the order in which they are generated in the code

To make sure closer objects are drawn in front of objects behind them

```
    gl.glEnable(GL2.GL_DEPTH_TEST);

    gl.glClear(GL2.GL_DEPTH_BUFFER_BIT);
```

(We will talk in more detail about depth soon)

3D objects

We represent 3D objects as polygonal meshes.

A mesh is a collection of polygons in 3D space that form the skin of an object.

Let’s try the default glut teapot in Second3DExample.java

Lighting

Without lighting, our 3D objects look flat.

```
gl.glEnable(GL2.GL_LIGHTING);
```

Once **lighting** is enabled, by default,

gl.glColor does not work. You need to specify material properties

Enabling lighting does not actually turn any lights on. You also need something like

```
gl.glEnable(GL2.GL_LIGHT0);
```

3D transformations

3D affine transformations have the same structure as 2D but have an extra axis:

$$\mathbf{M} = \begin{array}{c} \\ \\ \end{array} \begin{array}{cccc} \mathbf{i} & \mathbf{j} & \mathbf{k} & \phi \\ \end{array}$$

$$\mathbf{M} = \begin{pmatrix} i_1 & j_1 & k_1 & \phi_1 \\ i_2 & j_2 & k_2 & \phi_2 \\ i_3 & j_3 & k_3 & \phi_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Transformations

Translation:

$$\mathbf{M_T} = \begin{pmatrix} 1 & 0 & 0 & \phi_1 \\ 0 & 1 & 0 & \phi_2 \\ 0 & 0 & 1 & \phi_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scale:

$$\mathbf{M_S} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Shear:

$$\mathbf{M_H} = \begin{pmatrix} 1 & h & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Rotation

The rotation matrix depends on the axis of rotation.

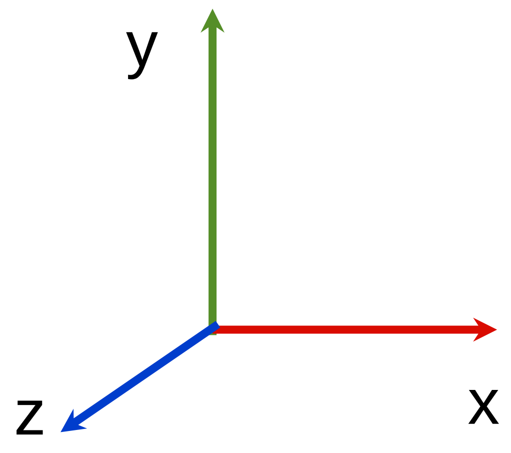
We can decompose any rotation into a sequence of rotations about the x, y and z axes.

Conversely, any sequence of rotations can be expresses as a single rotation about an axis.

3D Rotation

In each case, positive rotation is CCW from the next axis towards the previous axis.

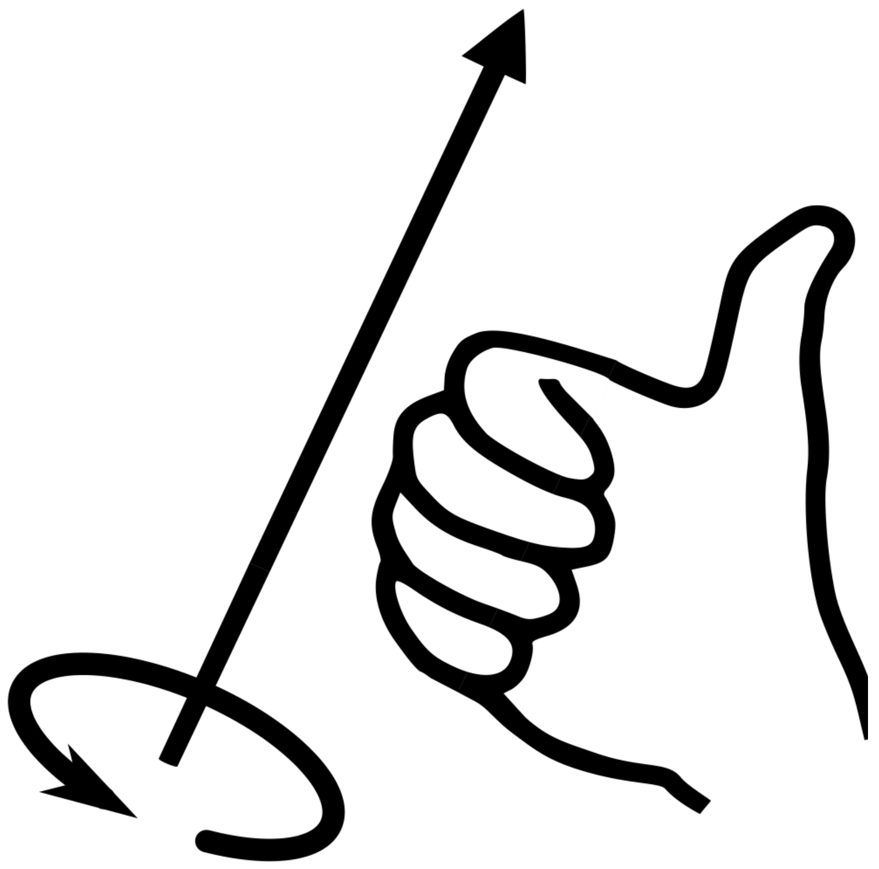
- **Mx** rotates y towards z
- **My** rotates z towards x
- **Mz** rotates x towards y



This works no matter whether the frame is left or right handed.

Right Hand Rule

For any axis, if the right thumb points in the positive direction of the axis the right fingers curl in the direction of rotation



3D Rotation

$$\mathbf{M_x} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{M_y} = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Rotation

$$\mathbf{M_z} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In OpenGL

```
gl.glTranslated(dx, dy, dz);

gl.glScaled(sx, sy, sz);

gl.glRotated(angle, vx, vy, vz);
```

angle of rotation

axis of rotation

Our Own 3D Mesh: A Cube

```
gl.glBegin(GL2.GL_QUADS);
   // front
   gl.glVertex3d(0, 0, 0);
   gl.glVertex3d(1, 0, 0);
   gl.glVertex3d(1, 1, 0);
   gl.glVertex3d(0, 1, 0);

   // back
   gl.glVertex3d(0, 0, -1);
   gl.glVertex3d(0, 1, -1);
   gl.glVertex3d(1, 1, -1);
   gl.glVertex3d(1, 0, -1);
   //etc
```

A Cube

Notice that each face is drawn so the polygon is facing outwards.

In a right-handed frame, this means the points are in anticlockwise order.

Note: If you use your right hand, your curved fingers represent the winding order and your thumb the outwards direction.

Exercise

```
//top face

//bottom face

//left face

//right face

//see code (rotateCube.java)
//for solns
```

Lighting and Normals

Once lighting is on, it is no longer enough to model the coordinates of your vertices, you need to provide normals as well. Eg

```
gl.glNormal3d(0,0,1);
```

These are used during the lighting calculations. Otherwise lighting does not work properly.

Note: The glut teapot already has normals defined – but we will need to add these ourselves for our own meshes.

OpenGL

```
gl.glBegin(GL2.GL_POLYGON);
{
    // set normal before vertex
    gl.glNormal3d(nx, ny, nz);
    gl.glVertex3d(px, py, pz);
    // etc...
}
gl.glEnd();
```

A Cube With Normals

```
gl.glBegin(GL2.GL_POLYGON); // front
   gl.glNormal3f(0,0,1);
   gl.glVertex3d(0, 0, 0);
   gl.glVertex3d(1, 0, 0);
   gl.glVertex3d(1, 1, 0);
   gl.glVertex3d(0, 1, 0);
gl.glEnd();

gl.glBegin(GL2.GL_POLYGON); // back
    gl.glNormal3f(0,0,-1);
    gl.glVertex3d(0, 0, -1);
    gl.glVertex3d(0, 1, -1);
    gl.glVertex3d(1, 1, -1);
    gl.glVertex3d(1, 0, -1);
gl.glEnd();
```

Lighting and Normal Normals

For the lighting calculations to work as expected normals passed to it must be unit length.

OpenGL transforms normals using a version of the modelview matrix called the *inverse transpose modelview* matrix. This means scaling *also* changes the length of normals.

To avoid this problem use

gl.glEnable(GL.GL_NORMALIZE);

Mesh Representation

It is common to represent a mesh in terms of three lists:

- vertex list: all the vertices used in the mesh
- normal list: all the normals used in the object
- face list: each face's vertices and normals as indices into the above lists.

Cube

vertex	x	y	z
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Cube

normal	x	y	z
0	-1	0	0
1	1	0	0
2	0	-1	0
3	0	1	0
4	0	0	-1
5	0	0	1

Cube

face	vertices	normals
0	0,1,3,2	0,0,0,0
1	4,6,7,5	1,1,1,1
2	0,4,5,1	2,2,2,2
3	2,3,7,6	3,3,3,3
4	0,2,6,4	4,4,4,4
5	7,3,1,5	5,5,5,5

Modeling Normals

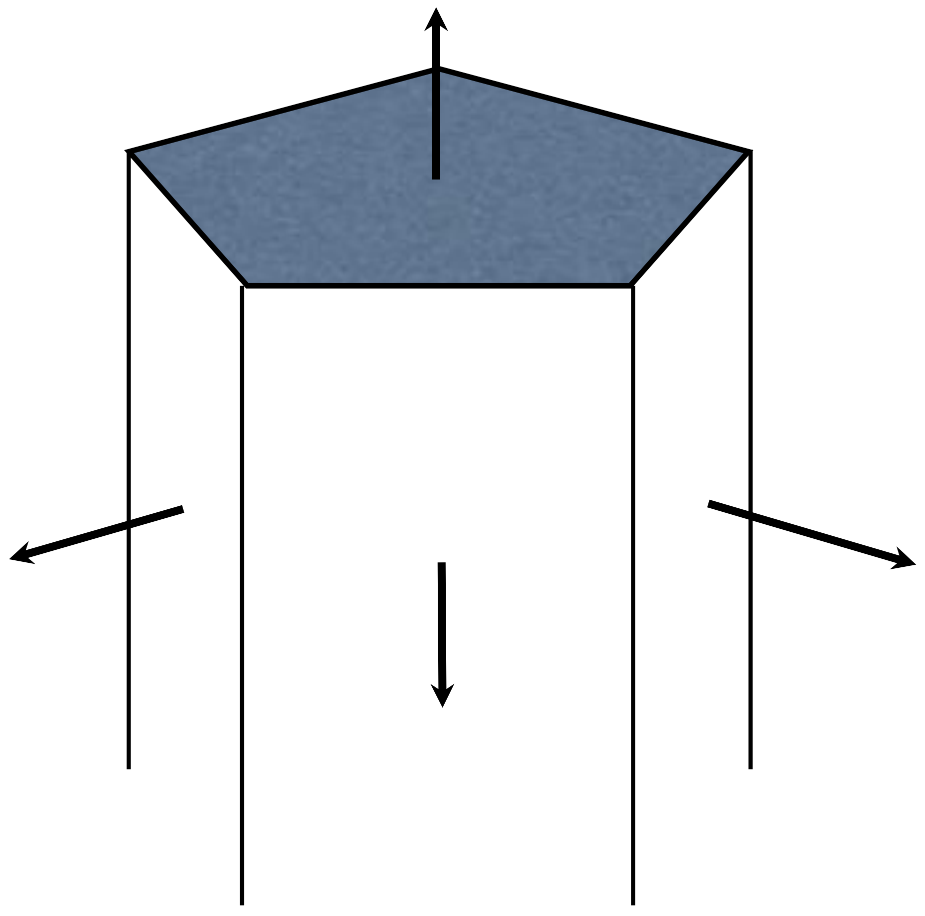
Every vertex has an associated normal

Default normal is (0,0,1)

On flat surfaces, we want to use face normals set the normals perpendicular to the face (this is what we did with our cube).

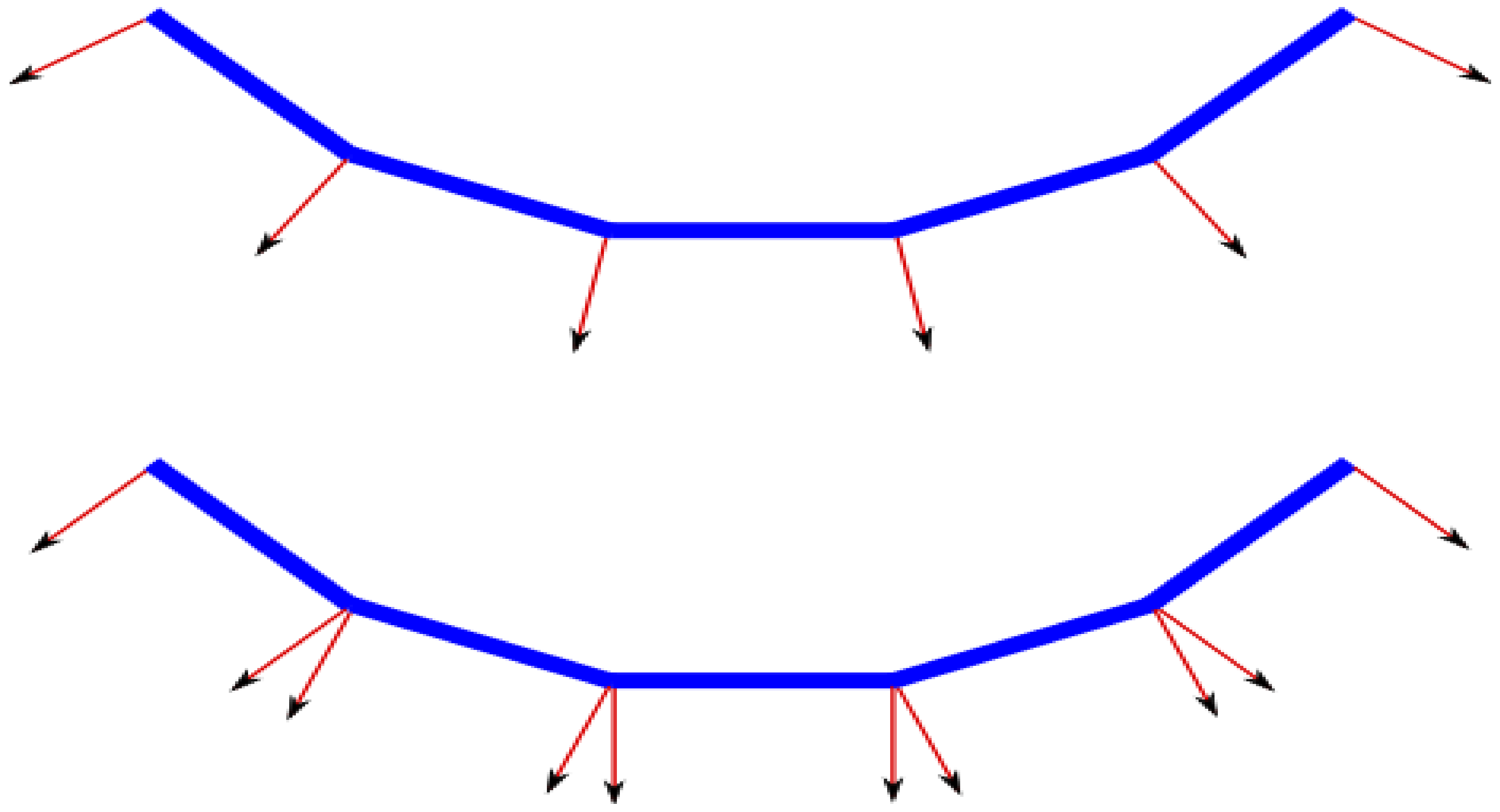
On curved surfaces, we may want to specify a different value for the normal, so the normals change more gradually over the curvature.

Face Normals



Smooth vs Flat Normals

Imagine this is a top down view of a prism



Calculation of Face Normals

Every vertex for a given face will be given the same normal.

This normal can be calculated by

- Finding cross product of 2 sides if the face is planar (triangles are always planar)
- Using Newell's method for arbitrary polygons which may not be planar

Newell's Method

A robust approach to computing **face normal** for arbitrary polygons:

$$
\begin{aligned}
n_x &= \sum_{i=0}^{N-1} (y_i - y_{i+1})(z_i + z_{i+1}) \\
n_y &= \sum_{i=0}^{N-1} (z_i - z_{i+1})(x_i + x_{i+1}) \\
n_z &= \sum_{i=0}^{N-1} (x_i - x_{i+1})(y_i + y_{i+1})
\end{aligned}
$$

$$\text{where } (x_N, y_N, z_N) = (x_0, y_0, z_0)$$

Vertex Normals

For smooth surfaces we can calculate each normal based on

- maths if it is a surface with a mathematical formula
- **averaging the face normals of adjacent vertices** (if this is done without normalising the face normals you get a weighted average). This is the basic way and can be fined tuned to exclude averaging normals that meet at a sharp edge etc.

Cylinder Example

For a cylinder we want smooth vertex normals for the curved surface – as we do not want to see edges there.

But face normals for the top and bottom where there should be a distinct edge

3D Camera

A 3D camera can be at any 3D point and orientation.

As before, the view transform is the world-to-camera transform, which is the inverse of the usual local-to-global transformation for objects.

The camera points backwards down its local z-axis.

The view volume

A 2D camera shows a 2D world window.

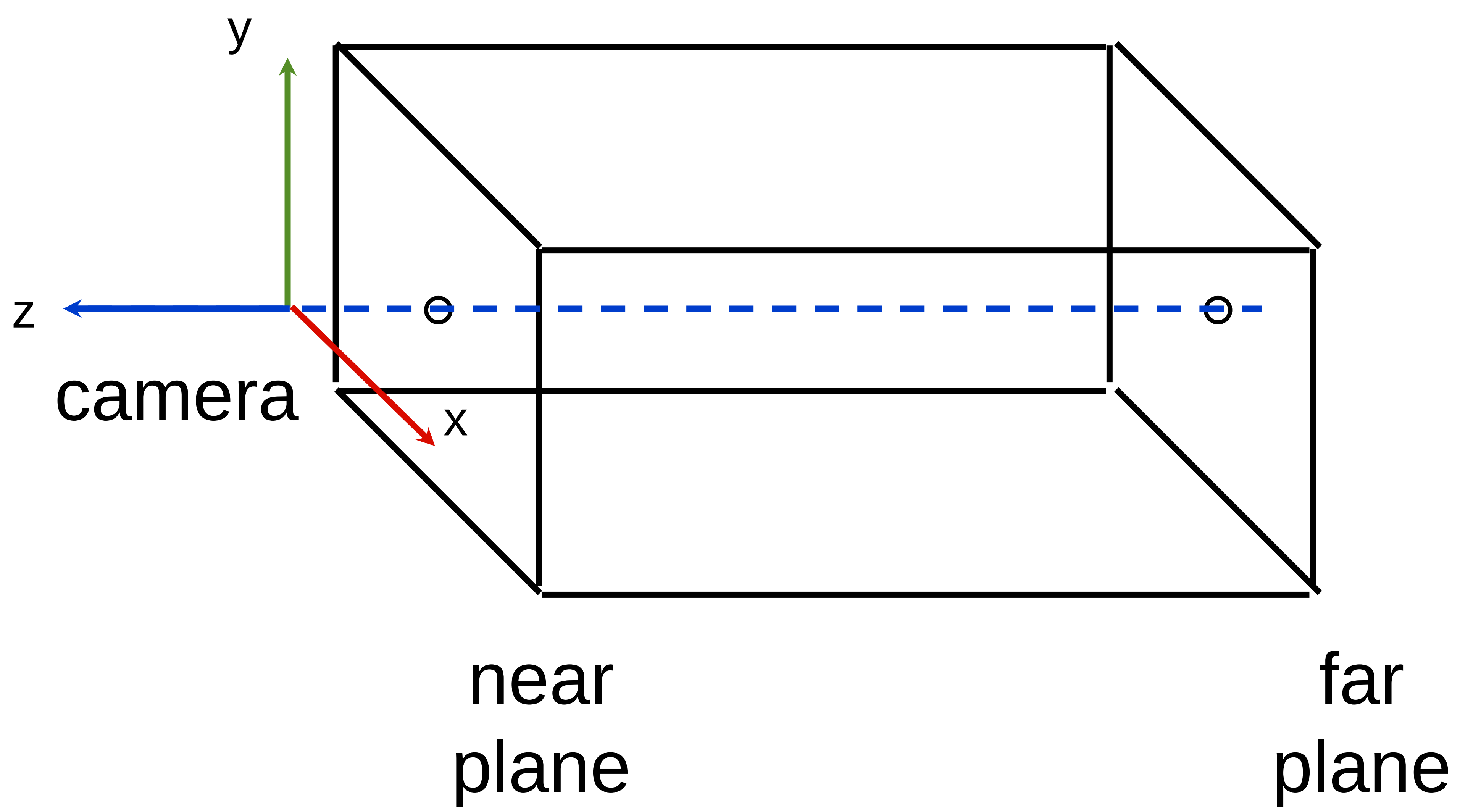
A 3D camera shows a 3D view volume.

This is the area of space that will be displayed in the viewport.

Objects outside the view volume are clipped.

The view volume is in camera coordinates.

Orthographic view volume



Projection

We want to project a point from our 3D view volume onto the near plane, which will then be mapped to the viewport.

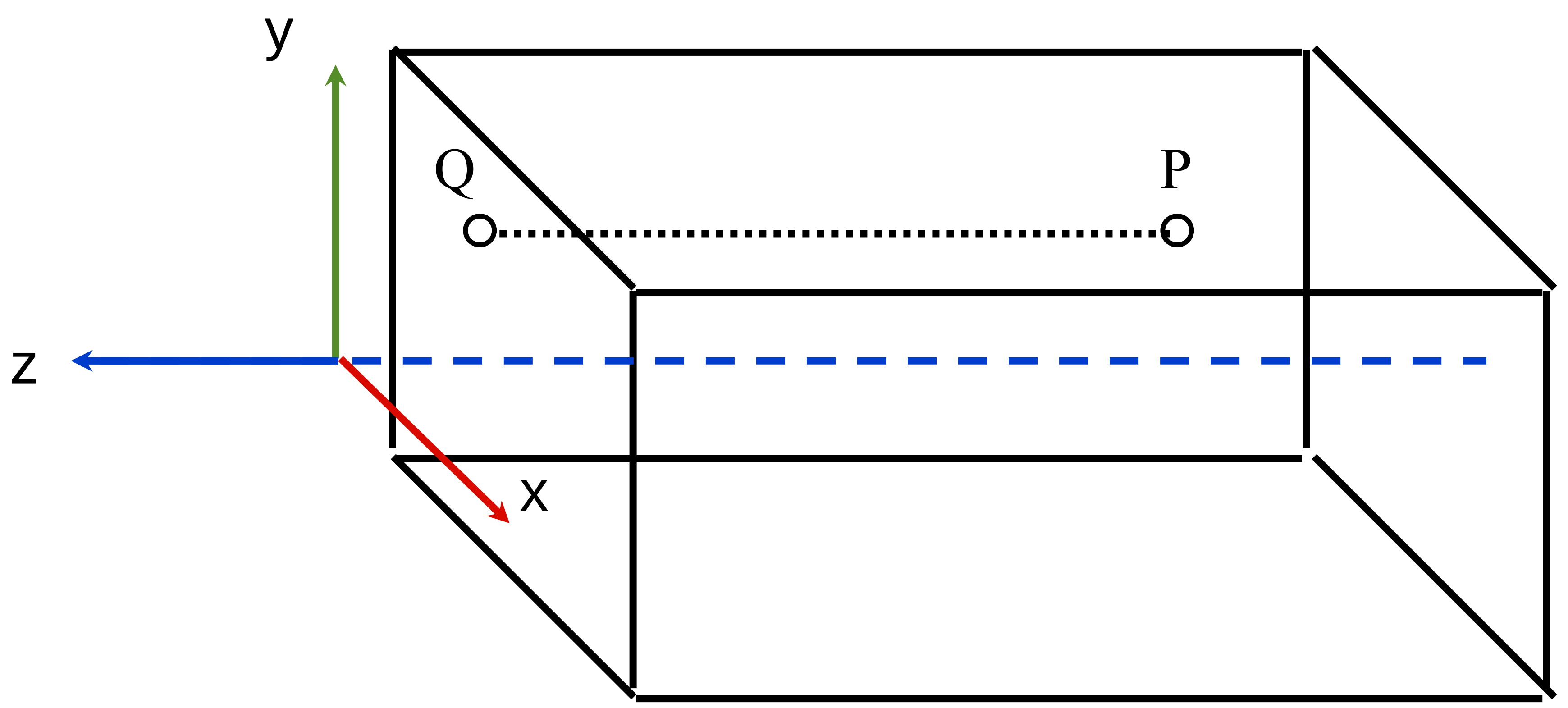
Project happens after the model-view transformation has been applied, so all points are in camera coordinates.

Points with negative z values are in front of the camera.

Orthographic projection

The orthographic projection is simple:

$$\begin{aligned} q_1 &= p_1 \\ q_2 &= p_2 \end{aligned}$$



glOrtho()

```
// create a 3D orthographic
// projection

gl.glMatrixMode(GL2.GL_PROJECTION);
gl.glLoadIdentity();

gl.glOrtho(left, right,
           bottom, top,
           near, far);
```

glOrtho

The default camera is located at the origin oriented down the negative z-axis.

Using a value of 2 for near means to place the near plane at z = -2

Similary far =8 would place it at z=-8

Orthographic Projections

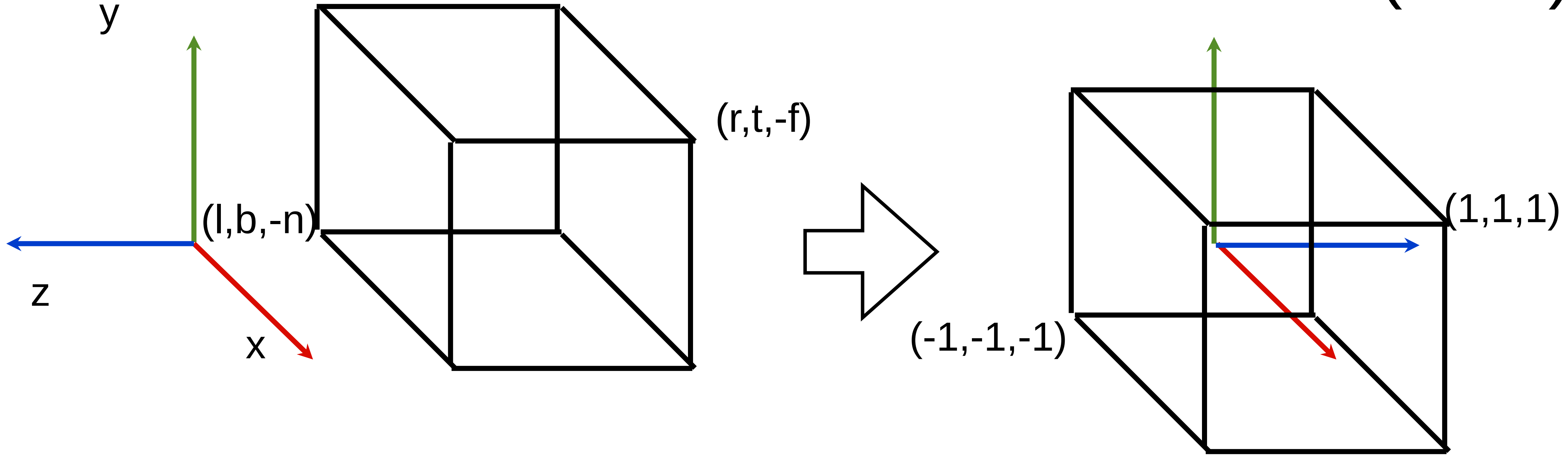
Orthographic projections are commonly used in:

- design – the projection maintains parallel lines and describes shapes of objects completely and exactly
- They are also used in some computer games

Canonical View Volume

It is convenient for clipping if we scale all coordinates so that visible points lie within the range (-1,1). Note the z axis signs are flipped. It is now a left handed system.

This is called the canonical view volume (CVV).



Orthographic transformation matrix

This maps the points in camera/eye co-ordinates into the canonical view volume.

$$\mathbf{M_O} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

What about depth?

We still need depth information attached to each point so that later we can work out which points are in front.

The projection matrix maps z values of visible points between to between -1 for near and 1 for far.

So we are still working in 4D (x,y,z,w) homogenous co-ordinates.

Setting the Camera view

```
// the camera is at 2,3,3
// rotated 45 deg about y
// then -10 deg about x

gl.glMatrixMode(GL2.GL_MODELVIEW);
gl.glLoadIdentity();

gl.glRotated(10, 1, 0, 0);
gl.glRotated(-45, 0, 1, 0);
gl.glTranslated(-2, -3, -3);
```

gluLookAt

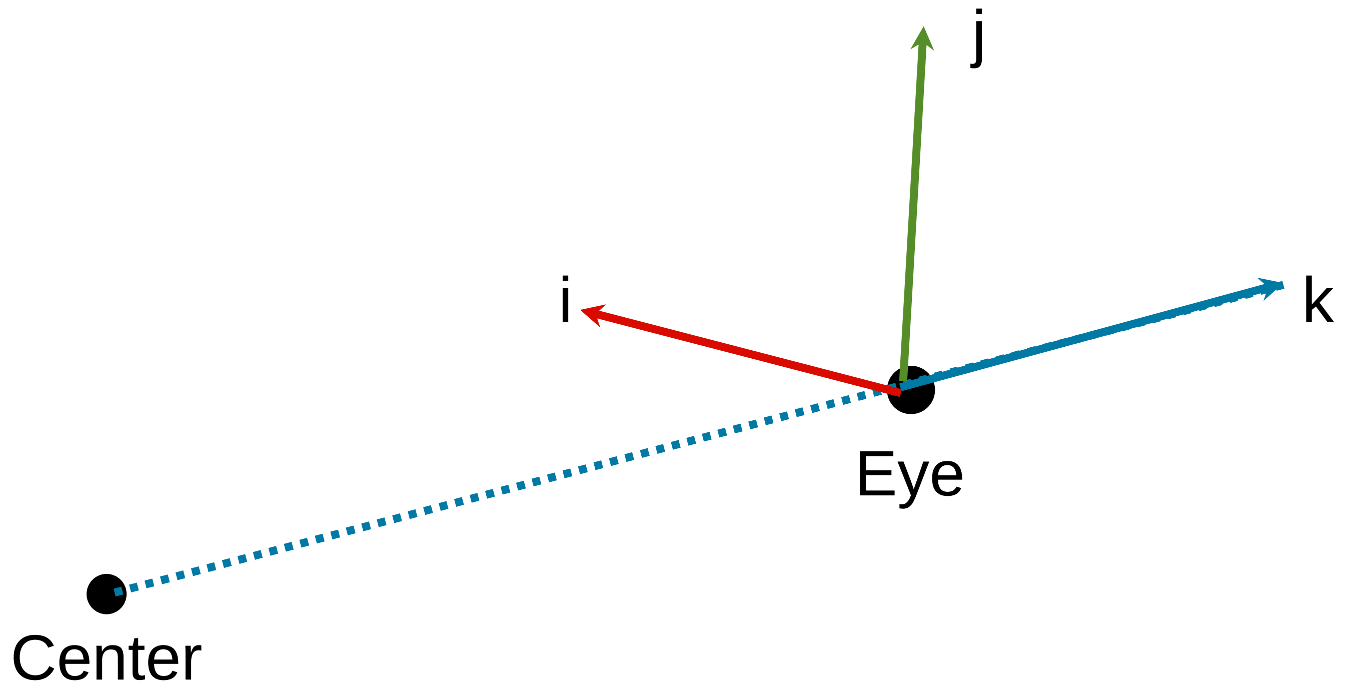
A convenient shortcut for setting the camera view transform.

```
gluLookAt(eyeX, eyeY, eyeZ,
          centerX, centerY, centerZ,
          upX, upY, upZ)
```

This places a camera an (eyeX, eyeY, eyeZ) looking towards (centreX, centreY, centreZ)

gluLookAt

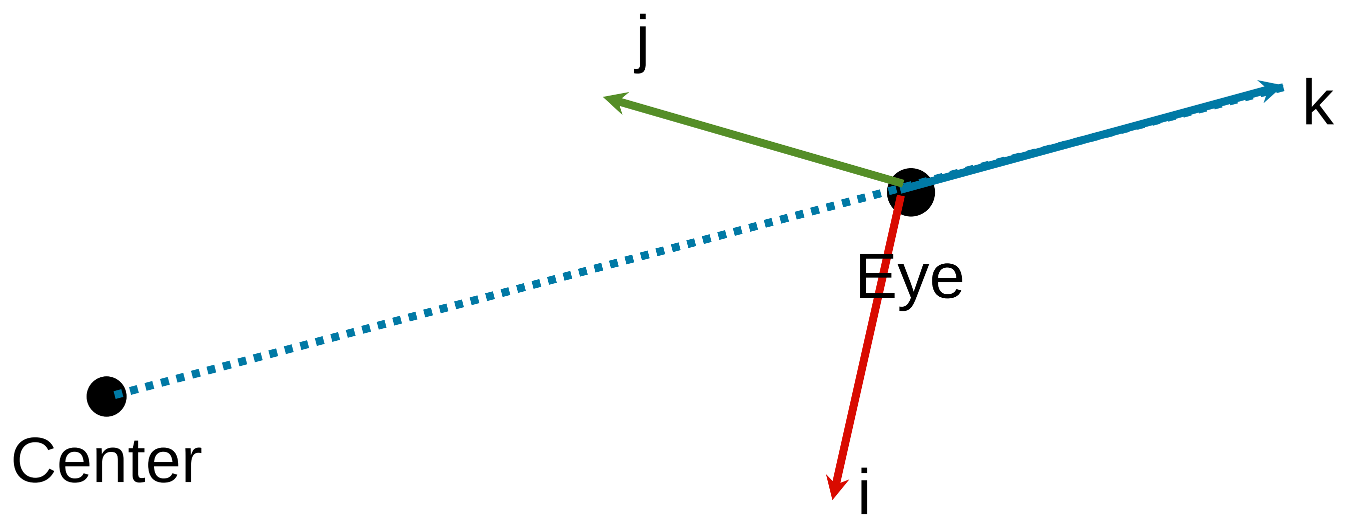
A position and a target alone do not provide a complete 3D coordinate frame.



k is fixed but i and j can rotate.

gluLookAt

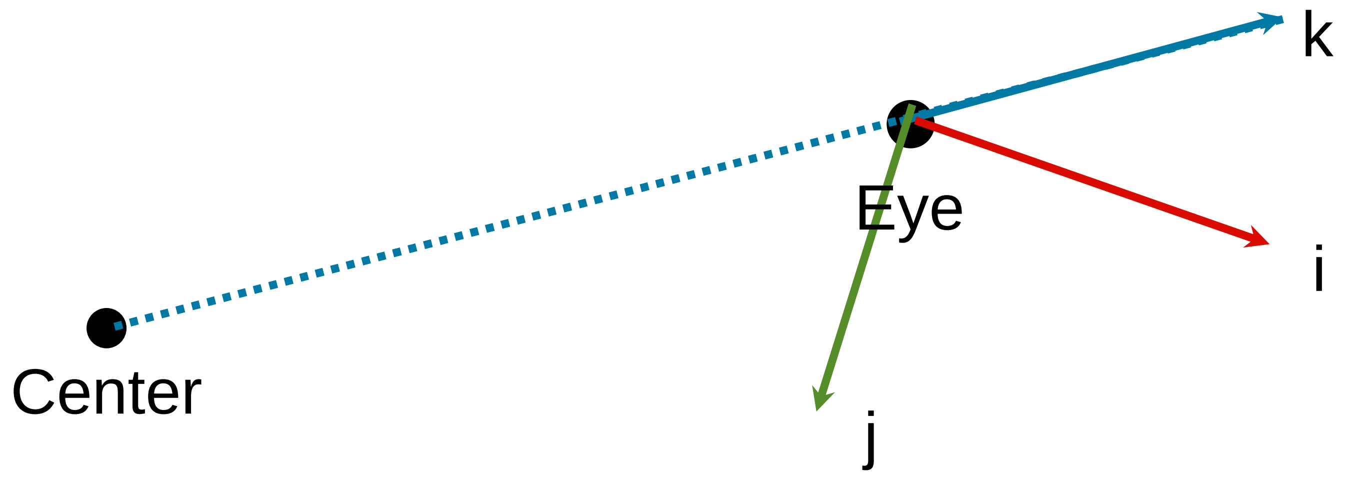
A position and a target alone do not provide a complete 3D coordinate frame.



k is fixed but i and j can rotate.

gluLookAt

A position and a target alone do not provide a complete 3D coordinate frame.



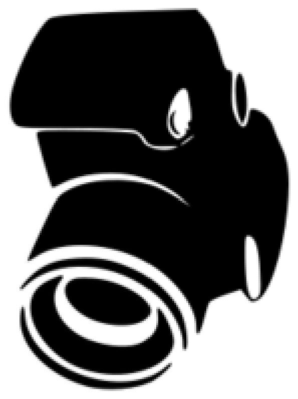
k is fixed but i and j can rotate.

gluLookAt

Generally we want the camera angled so that the image is vertical.

So we don't (unless we want to) have our camera upside down or on its side like

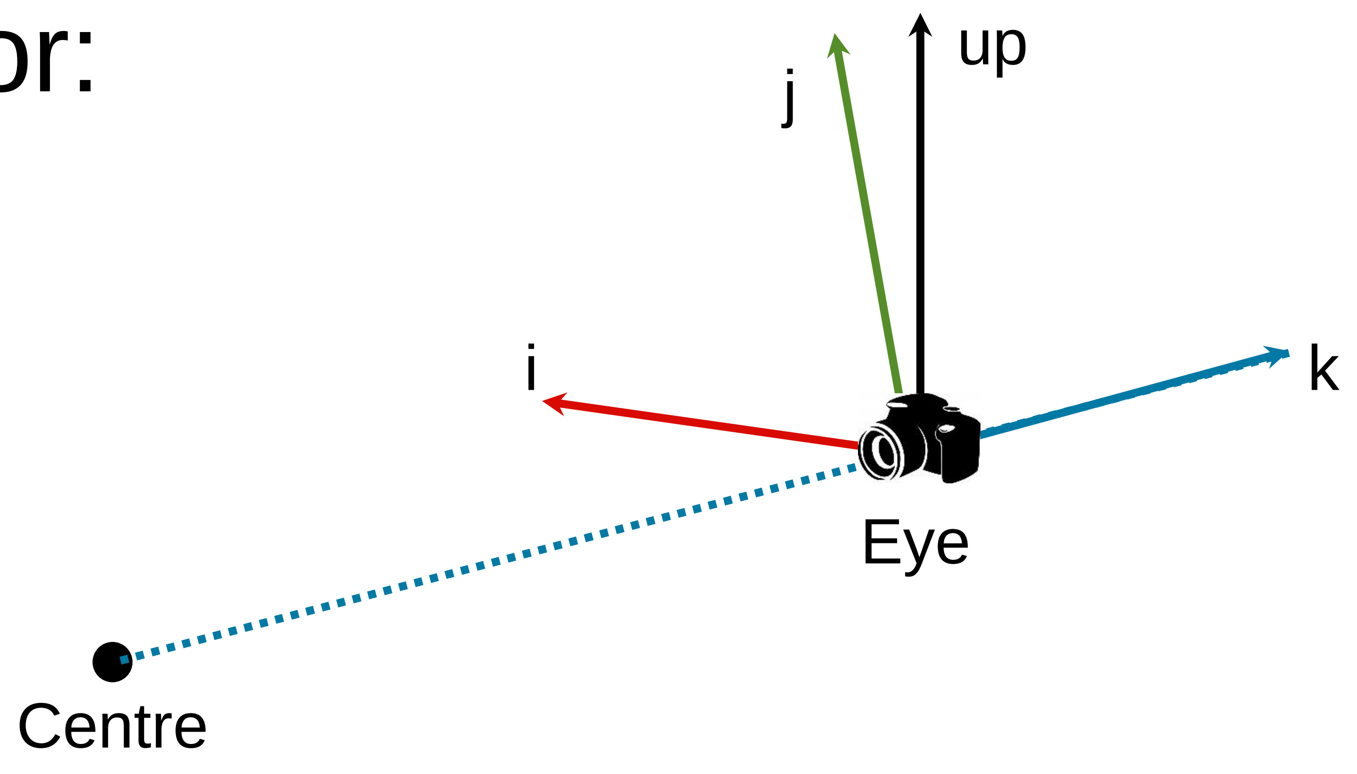
or



We achieve this by specifying an 'up' vector in world coordinates, which points upwards.

gluLookAt

Now we want the camera's i vector to be at right angles to both the k vector and the up vector:



gluLookAt

We can calculate this as:

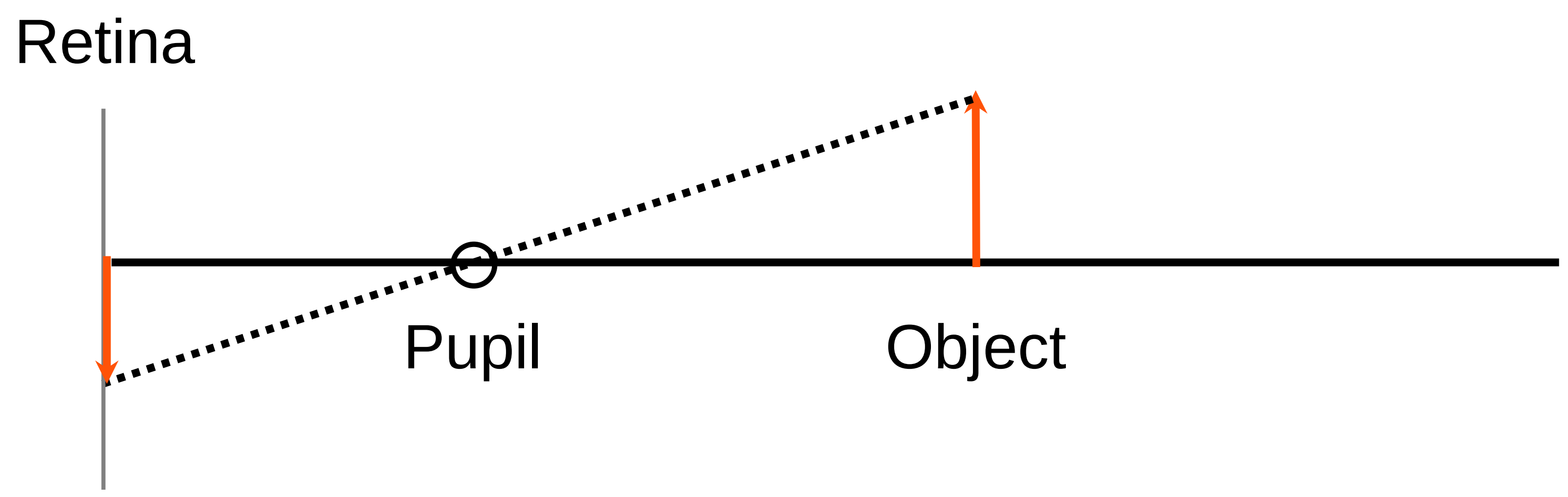
$$\begin{aligned}
\mathbf{k} &= E - C \\
\mathbf{i} &= \mathbf{up} \times \mathbf{k} \\
\mathbf{j} &= \mathbf{k} \times \mathbf{i} \\
\phi &= E
\end{aligned}$$

This is the view transformation computed by gluLookAt.

Foreshortening

Foreshortening is the name for the experience of things appearing smaller as they get further away.

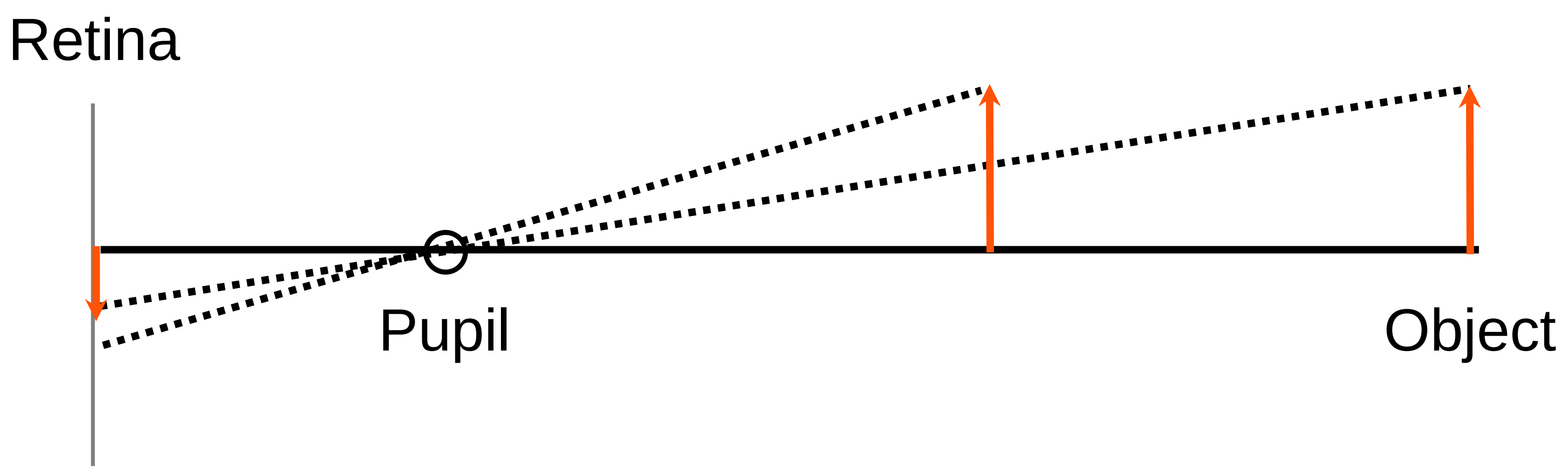
Foreshortening happens because our eye is a point camera.



Foreshortening

Foreshortening is the name for the experience of things appearing smaller as they get further away.

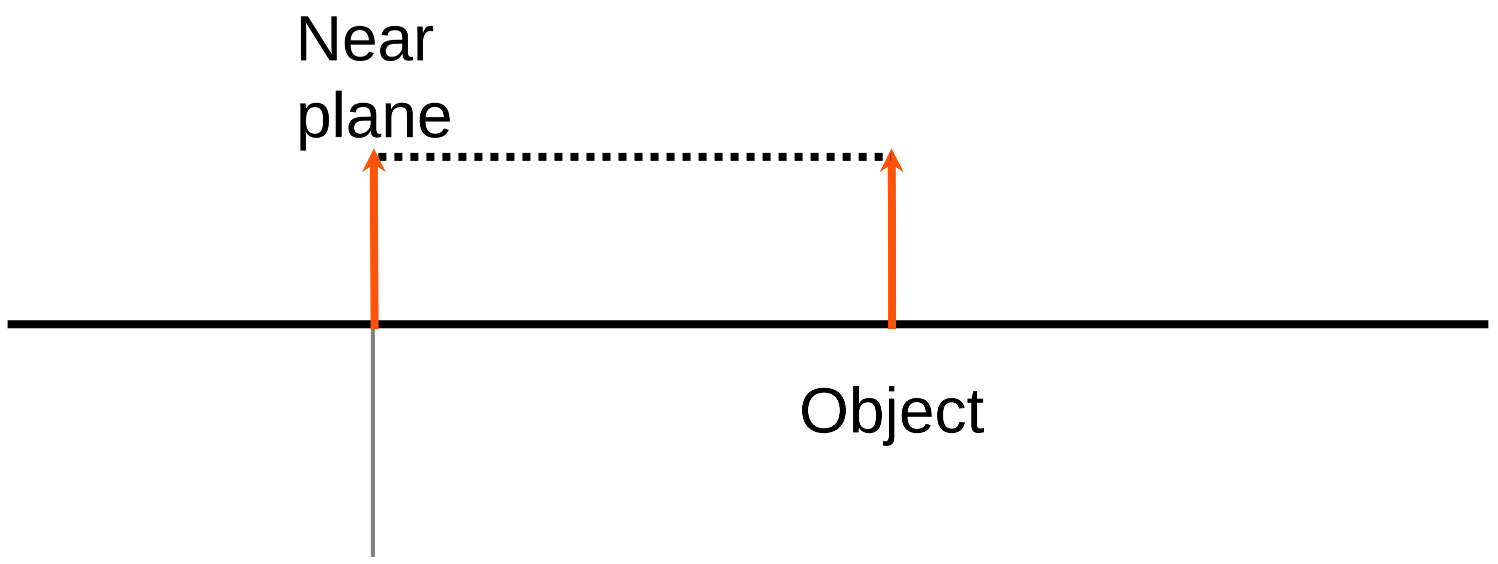
Foreshortening happens because our eye is a point camera.



Orthographic camera

The orthographic camera does not perform foreshortening.

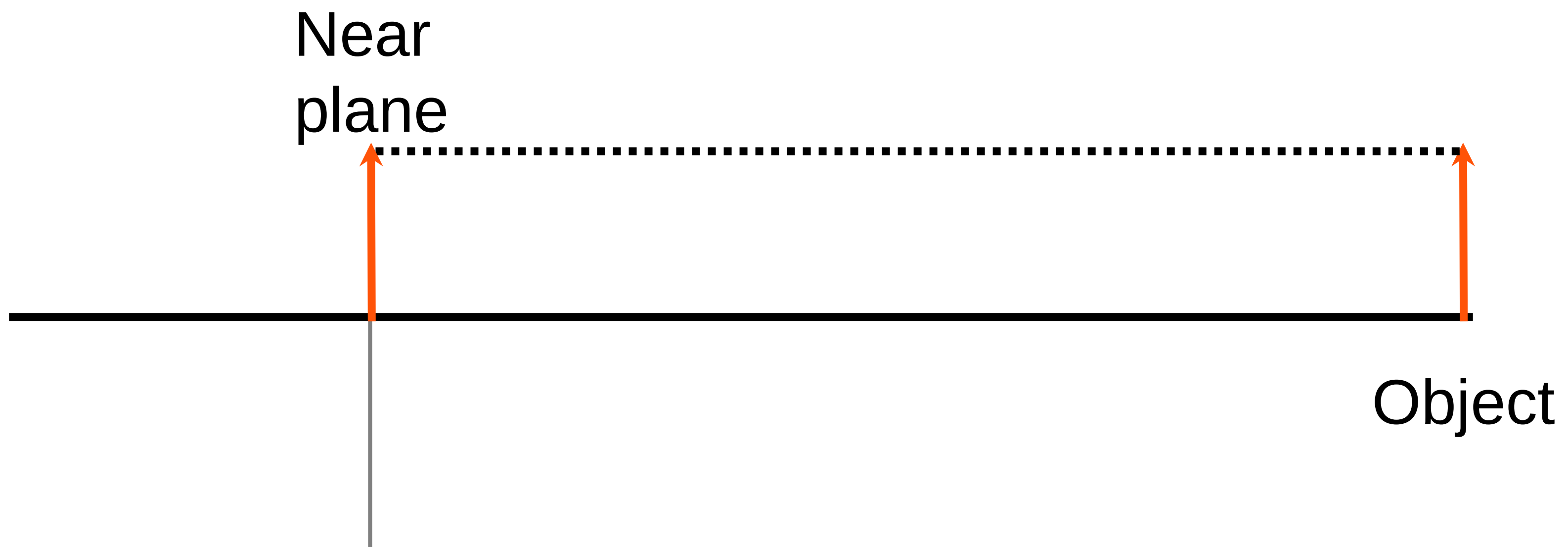
Objects size is independent of distance from the camera.



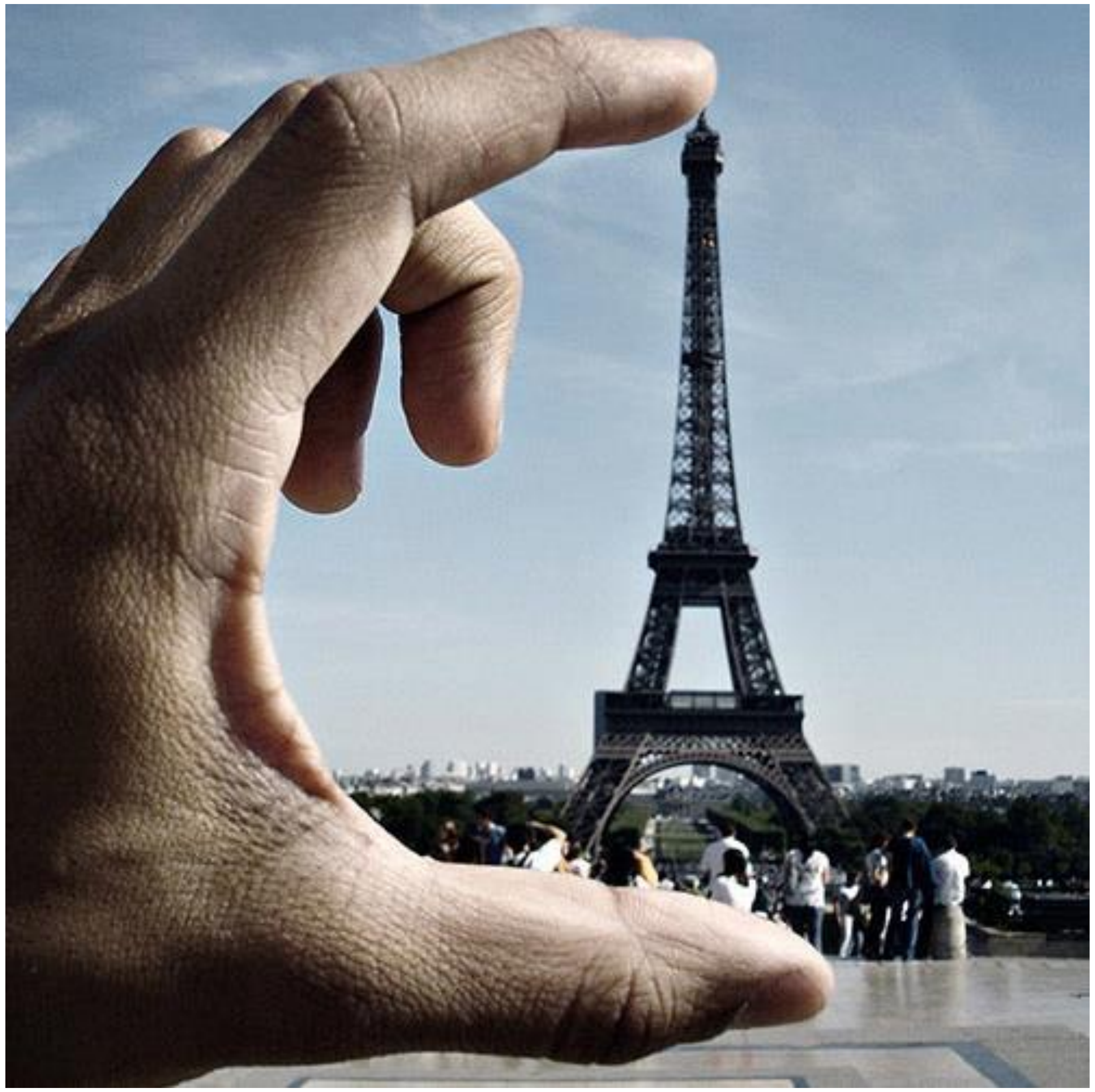
Orthographic camera

The orthographic camera does not perform foreshortening.

Objects size is independent of distance from the camera.

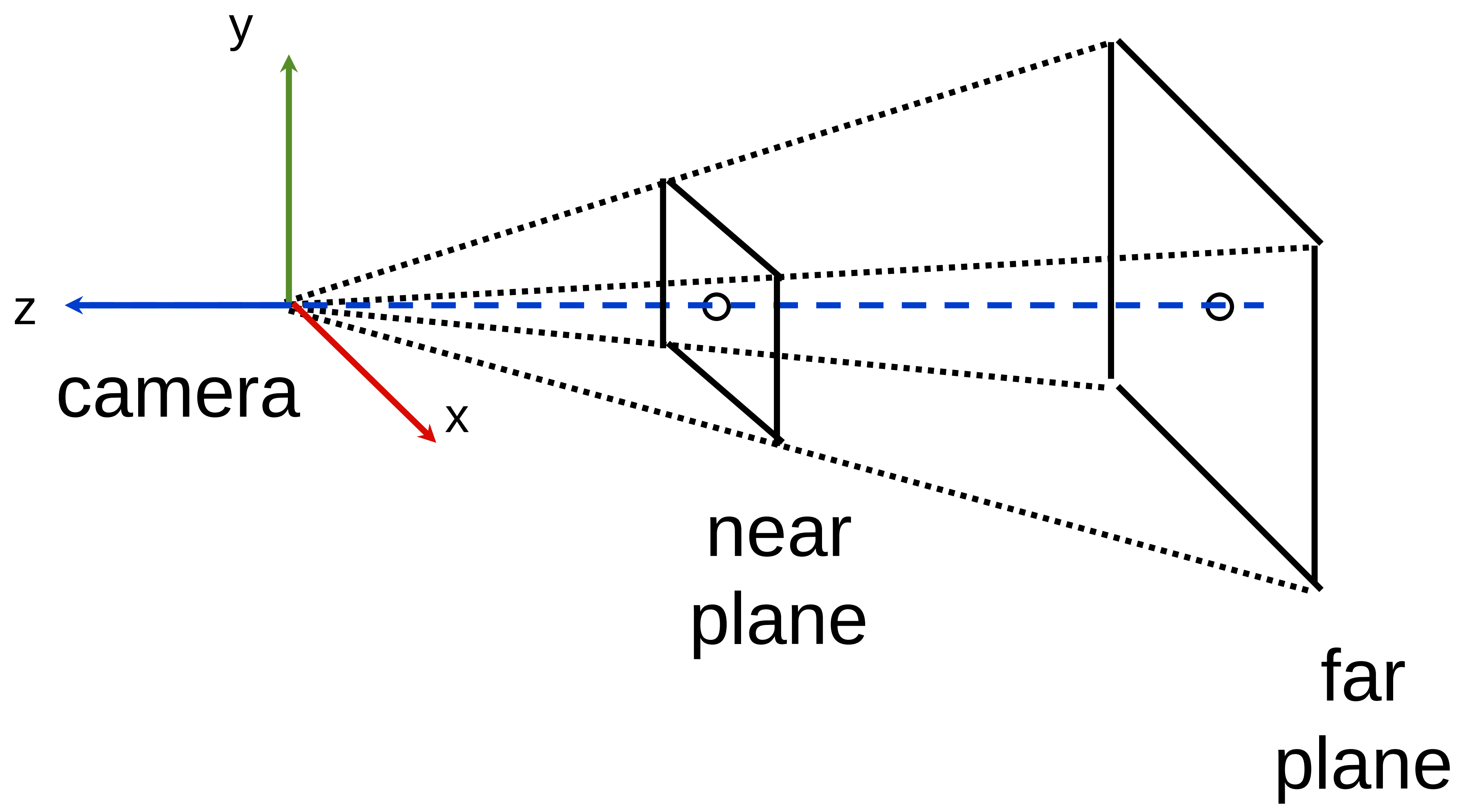


Perspective



Perspective camera

We can define a different kind of projection that implements a perspective camera. The view volume is a frustum.



glFrustum()

```
// create a perspective projection

gl.glMatrixMode(GL2.GL_PROJECTION);
gl.glLoadIdentity();
gl.glFrustum(left, right,
             bottom, top,
             near, far);

// left, right, bottom, top are the
// sides of the *near* clip plane

// near and far are *positive*
```

glFrustum

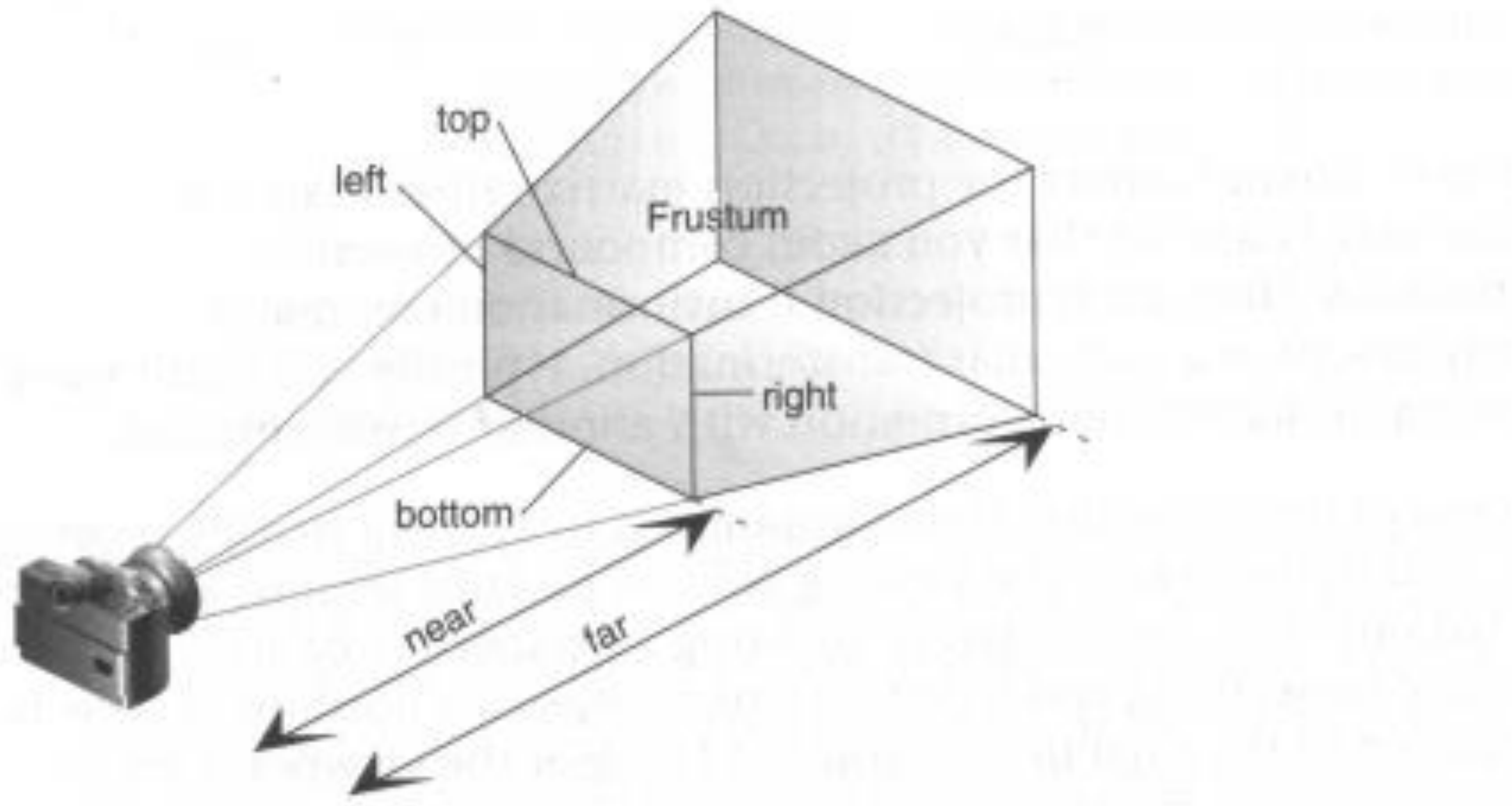


Figure 3-13 Perspective Viewing Volume Specified by glFrustum()

gluPerspective()

```
//a more convenient form instead
//of glFrustum

gl.glMatrixMode(GL2.GL_PROJECTION);
gl.glLoadIdentity();

GLU glu = new GLU();

//fieldOfView is in the y direction
glu.gluPerspective(fieldOfView,
           aspectRatio, near, far);
```

gluPerspective

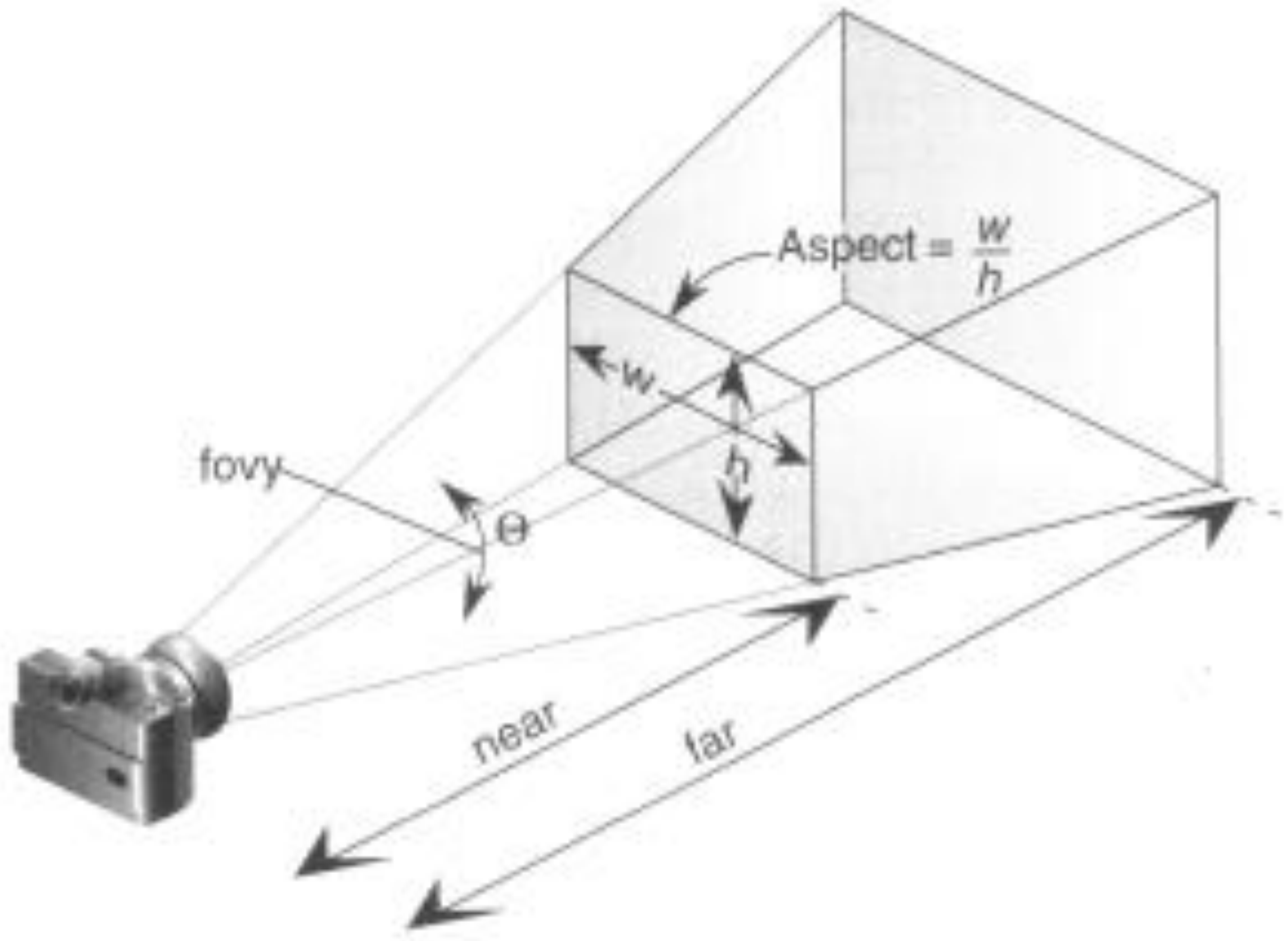


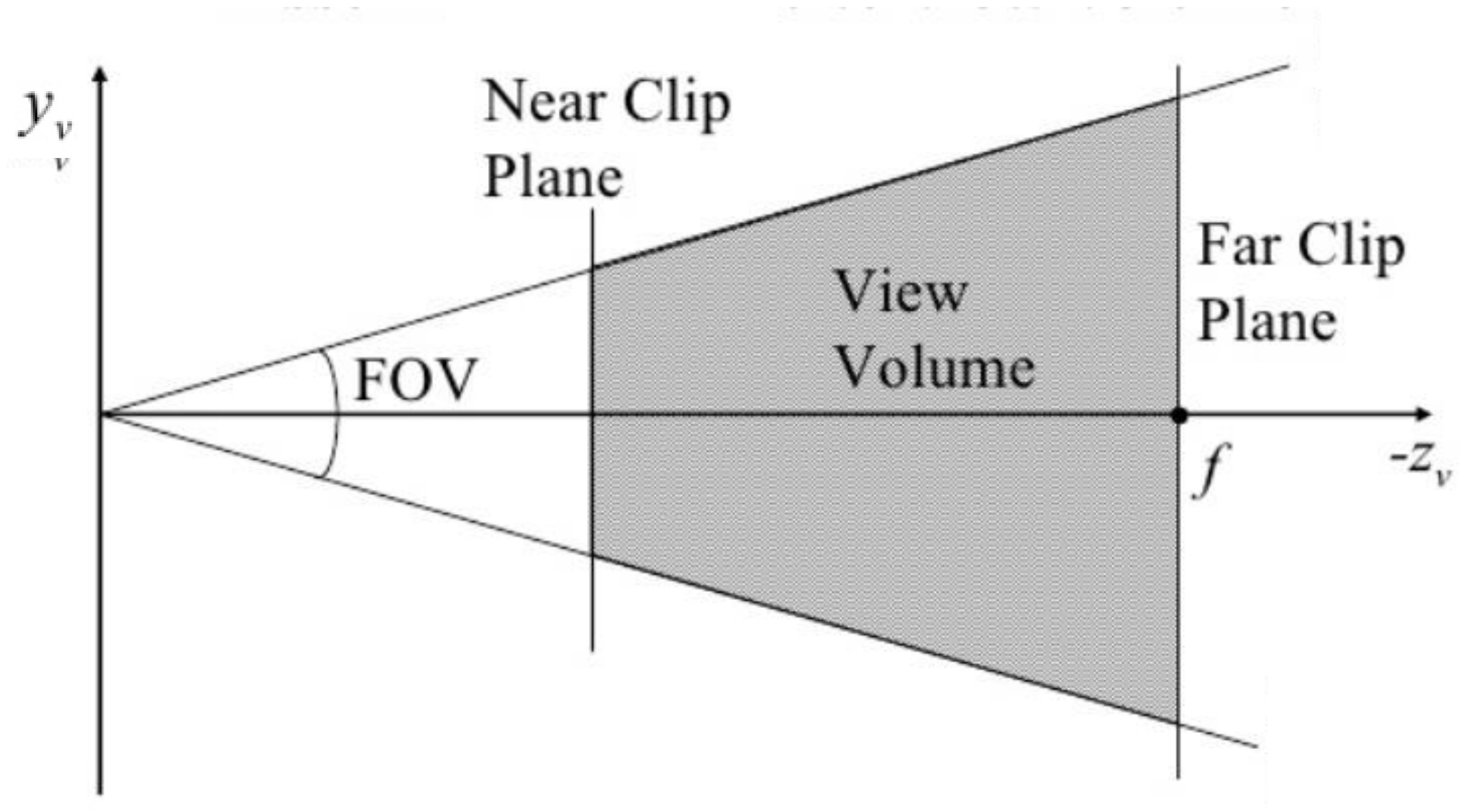
Figure 3-14 Perspective Viewing Volume Specified by gluPerspective()

Side on View

Assuming a symmetric view volume

tan(FOV/2) = height/(2 * near)

width = height * aspectRatio



‘Zoom’

‘Zoom’: Increasing/decreasing the FOV (changes the perspective)

Dolly: Translating the camera along the z-axis

Dolly Zoom (Hitchcock effect): Zooming In/Out and Dollying backwards/forwards at the same time

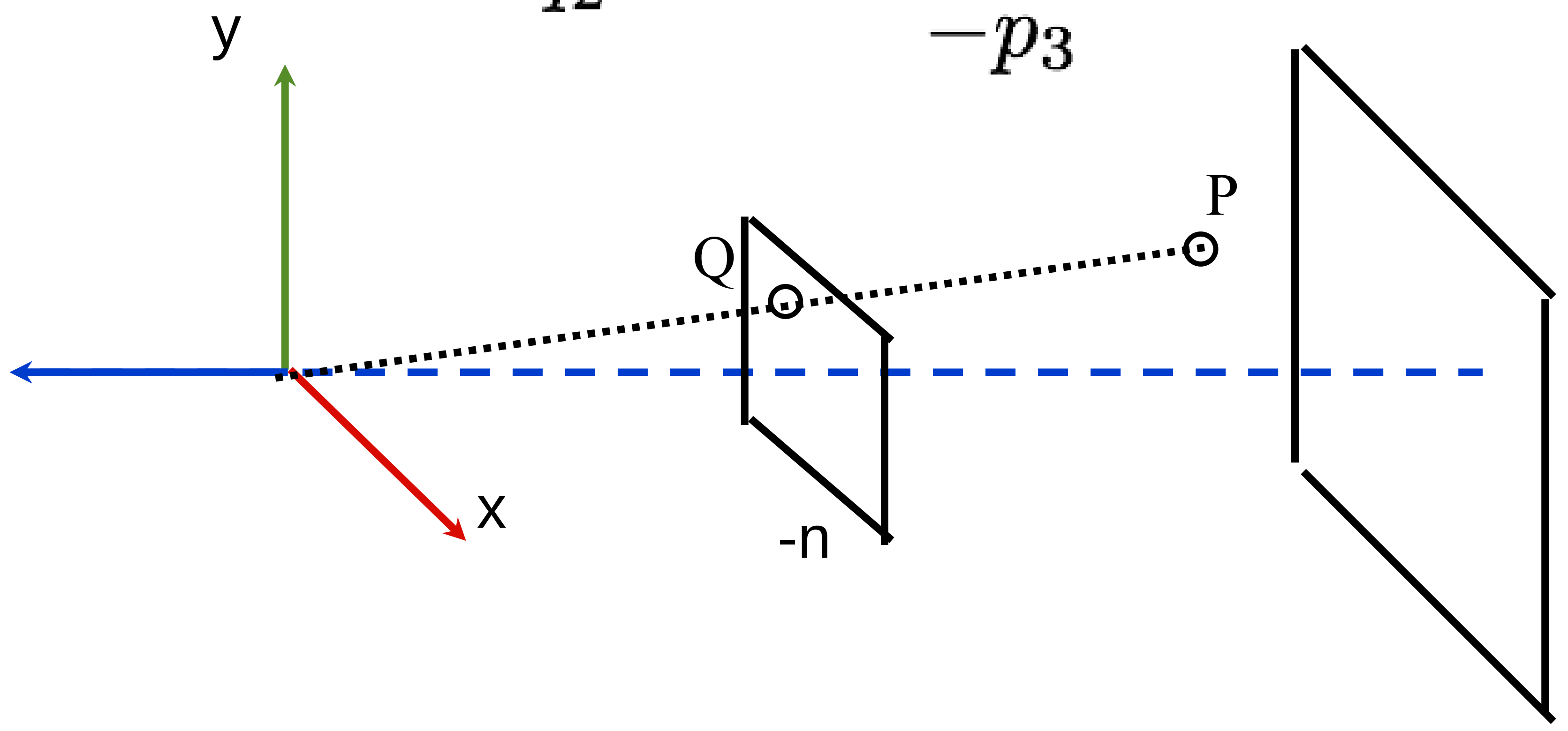
https://docs.unity3d.com/Manual/DollyZoom.html

https://www.youtube.com/watch?v=9wLTeugnl8o

Perspective projection

The perspective projection:

$$\begin{aligned} q_1 &= n\frac{p_1}{-p_3} \\ q_2 &= n\frac{p_2}{-p_3} \end{aligned}$$



Pseudodepth

We still need depth information attached to each point so that later we can work out which points are in front. And we want this information to lie between -1 and 1

We need to give each point a pseudodepth value that preserves front-to-back ordering.

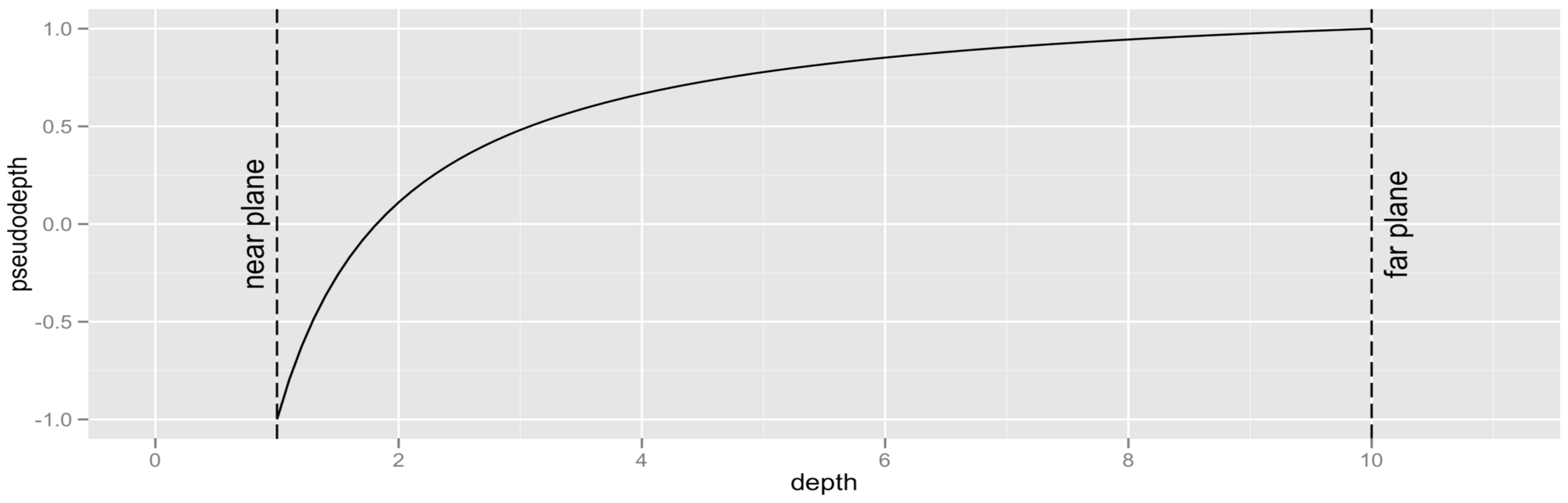
We want the equation for q3 to have the same denominator (-p3) as q1 and q2

Pseudodepth

These constraints yield an equation for pseudodepth:

$$\begin{aligned} q_3 &= \frac{ap_3 + b}{-p_3} \\ a &= -\frac{f+n}{f-n} \\ b &= \frac{-2fn}{f-n} \end{aligned}$$

Pseudodepth



Not linear. More precision for objects closer to the near plane. Rounding errors worse towards far plane.

Tip: Avoid setting near and far needlessy small/big for better use of precision

Homogeneous coordinates

We extend our representation for homogeneous coordinates to allow values with a fourth component other than zero or one.

We define an equivalence:

$$P = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} \equiv \begin{pmatrix} wp_1 \\ wp_2 \\ wp_3 \\ w \end{pmatrix} \text{ for any } w \neq 0$$

These two values represent the same point.

Example

(1,3,-2,1) is equivalent to

(2,6,-4,2) in homogeneous co-ordinates

This also means I can divide by say 3 using matrix multiplication. So if multiplying by M set my w value to 3 left other values unchanged such as

M (12,6,-3,1) = (12,6,-3,3)

Which would be equivalent to (4,2,-1,1)

Perspective transform

We can now express the perspective equations as a single matrix:

$$M_{perspective} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Note that this matrix is not affine.

Perspective transform

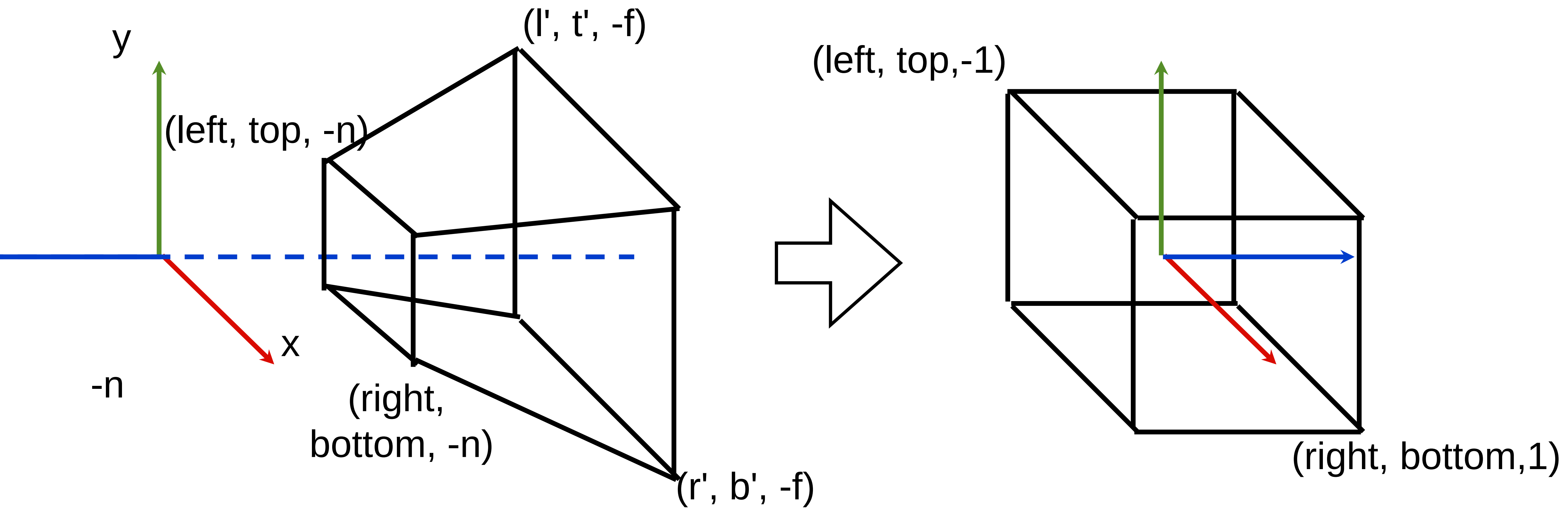
To transform a point:

$$\begin{aligned}
Q &= M_{perspective} P \\
&= \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} np_1 \\ np_2 \\ ap_3 + b \\ -p_3 \end{pmatrix} \equiv \begin{pmatrix} np_1/-p_3 \\ np_2/-p_3 \\ (ap_3 + b)/-p_3 \\ 1 \end{pmatrix}
\end{aligned}$$

Perspective transform

This matrix maps the perspective view volume to an axis aligned cube.

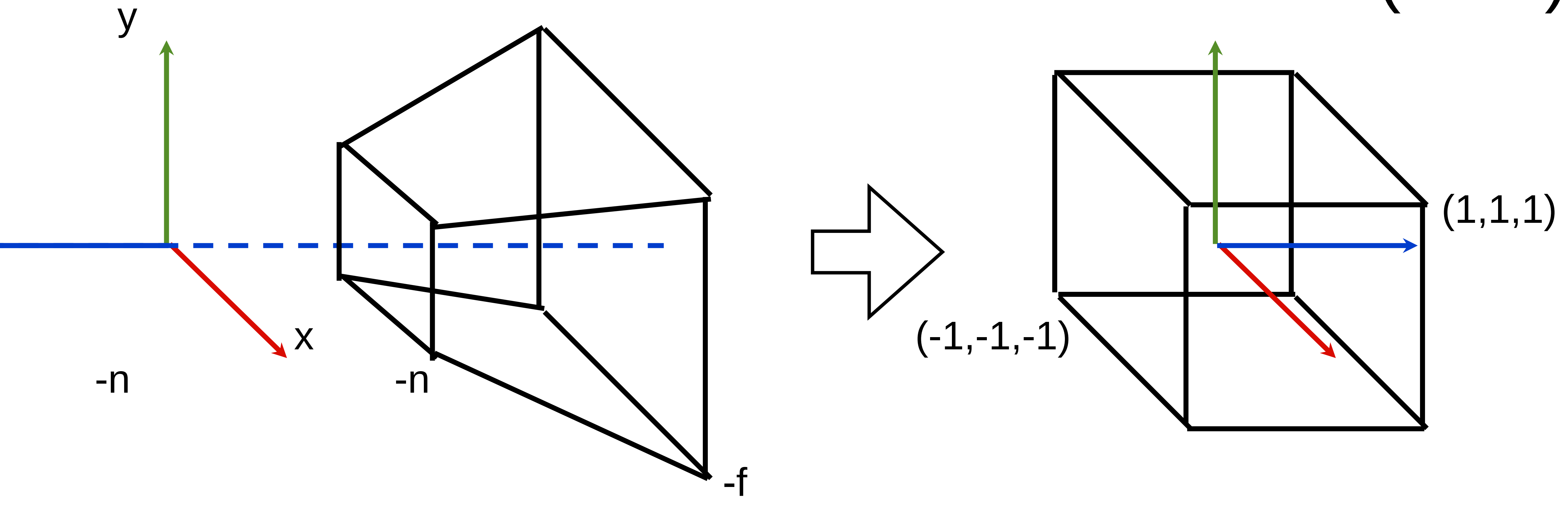
Note the z-axis has been flipped.



Canonical View Volume

It is convenient for clipping if we scale all coordinates so that visible points lie within the range (-1,1). Note the z axis signs are flipped. It is now a left handed system.

This is called the canonical view volume (CVV).



Perspective projection matrix

We can combine perspective transformation and scaling into a single matrix:

$$\mathbf{M_P} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Clipping

We can now clip points against the CVV.

The Liang-Barsky algorithm is a variant of Cyrus-Beck extended to handle 3D and homogeneous coordinates.

Details are in the textbook if you're interested.

Perspective division

After clipping we need to convert all points to the form with the fourth component equal to one.

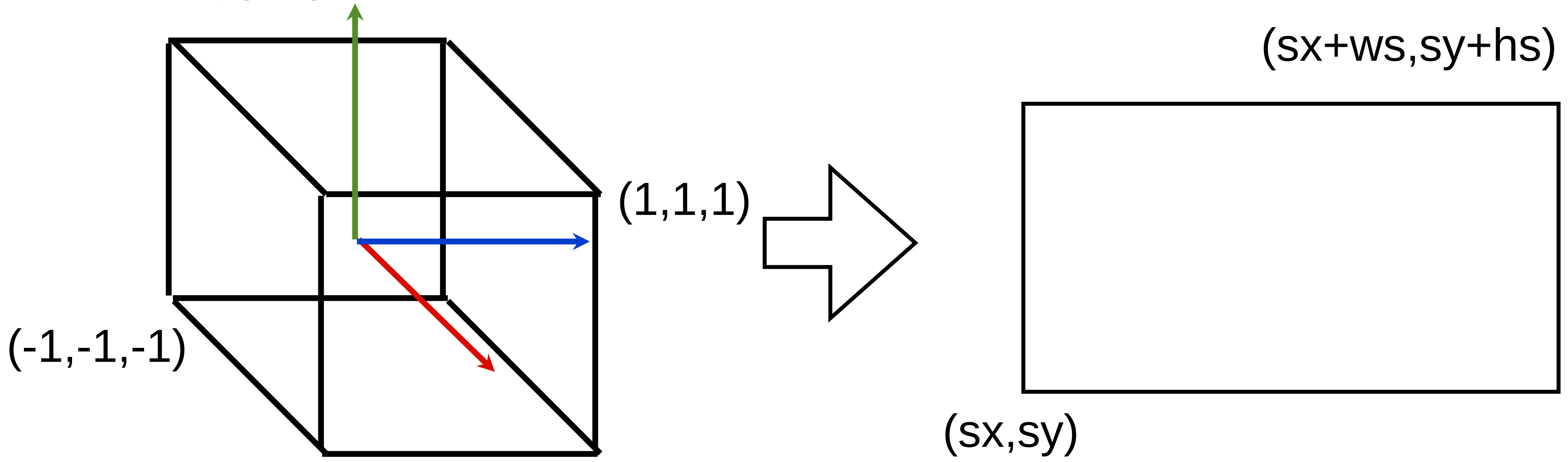
This is called the perspective division step.

$$\mathbf{q} = \frac{1}{p_4}\begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}$$

Viewport transformation

Finally we scale points into window coordinates corresponding to pixels on the screen. It also maps pseudodepth from

-1..1 to 0..1



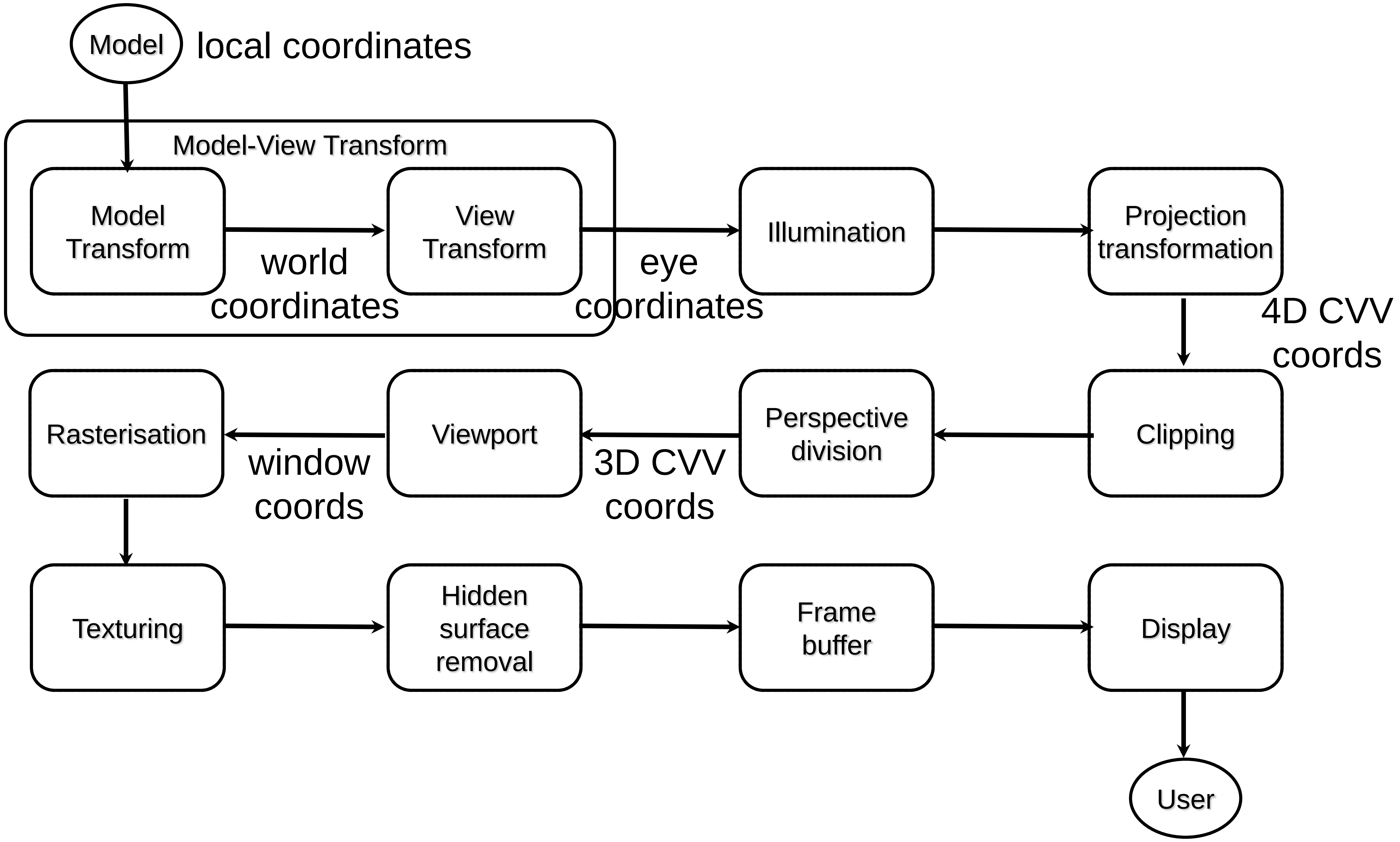
Viewport transformation

Again, we can do this with a matrix:

$$\mathrm{M}_{\mathbf{viewport}} = \begin{bmatrix} \frac{w_s}{2} & 0 & 0 & s_x + \frac{w_s}{2} \\ 0 & \frac{h_s}{2} & 0 & s_y + \frac{h_s}{2} \\ 0 & 0 & \frac{f_s - n_s}{2} & \frac{n_s + f_s}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where ns is 0 and fs is 1

The graphics pipeline



The graphics pipeline

To transform a point: $P = (p_1, p_2, p_3)^\top$

Extend to homogeneous coordinates:

$$P = (p_1, p_2, p_3, 1)^\top$$

Multiply by model matrix to get world coordinates: $P_w = \mathbf{M}_{\mathbf{model}} P$

Multiply by view matrix to get camera (eye) coordinates: $P_c = \mathbf{M}_{\mathbf{view}} P_w$

The graphics pipeline

Multiply by projection matrix to get clip coordinates (with fourth component):

$$P_{ccv} = \mathbf{M}_{\mathbf{perspective}} P_c$$

Clip to remove points outside CVV.

Perspective division to eliminate fourth component. $$P_n = \frac{1}{p_4} P_{ccv}$$

Viewport transformation to window coordinates. $P_v = \mathbf{M}_{\mathbf{viewport}} P_n$

Exercises 1

Write a snippet of jogl code to draw a triangle with vertices:

(2,1,-4)

(0,-1,-3)

(-2,1,-4)

Make sure you specify face normals for the vertices.

Exercises 2

We want to use a perspective camera to view our triangle. Which command/s would work?

```
gl.glOrtho(-3,3,-3,3,0,8);

gl.glFrustum(-3,3,-3,3,0,8);

gl.glFrustum(-3,3,-3,3,-2,8);

glu.gluPerspective(60,1,2,8);

glu.gluPerspective(60,1,0,8);
```

Exercises 3

What would be an equivalent way to specify your perspective camera?

Where would the x and y vertices in our triangle be projected to on the near plane? What would the pseudo-depth of our vertices be in CVV co-ordinates (-1..1)?

Exercises 4

Suppose we wanted to add another triangle with vertices

(-0.5,0,0)

(0.5,0.5,0)

(0.5,-0.5,0)

Would this appear on the screen? How could we fix this?