COMP3421

Intro to 3D, Depth, Perspective

Robert Clifton-Everest

Email: robertce@cse.unsw.edu.au

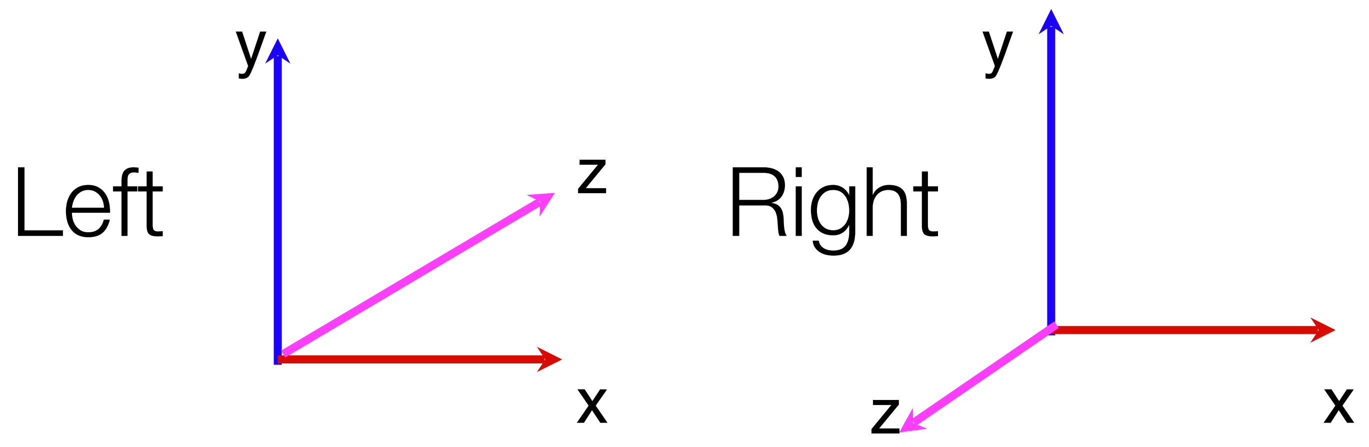
3D coordinates

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

$$P = \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \qquad \mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

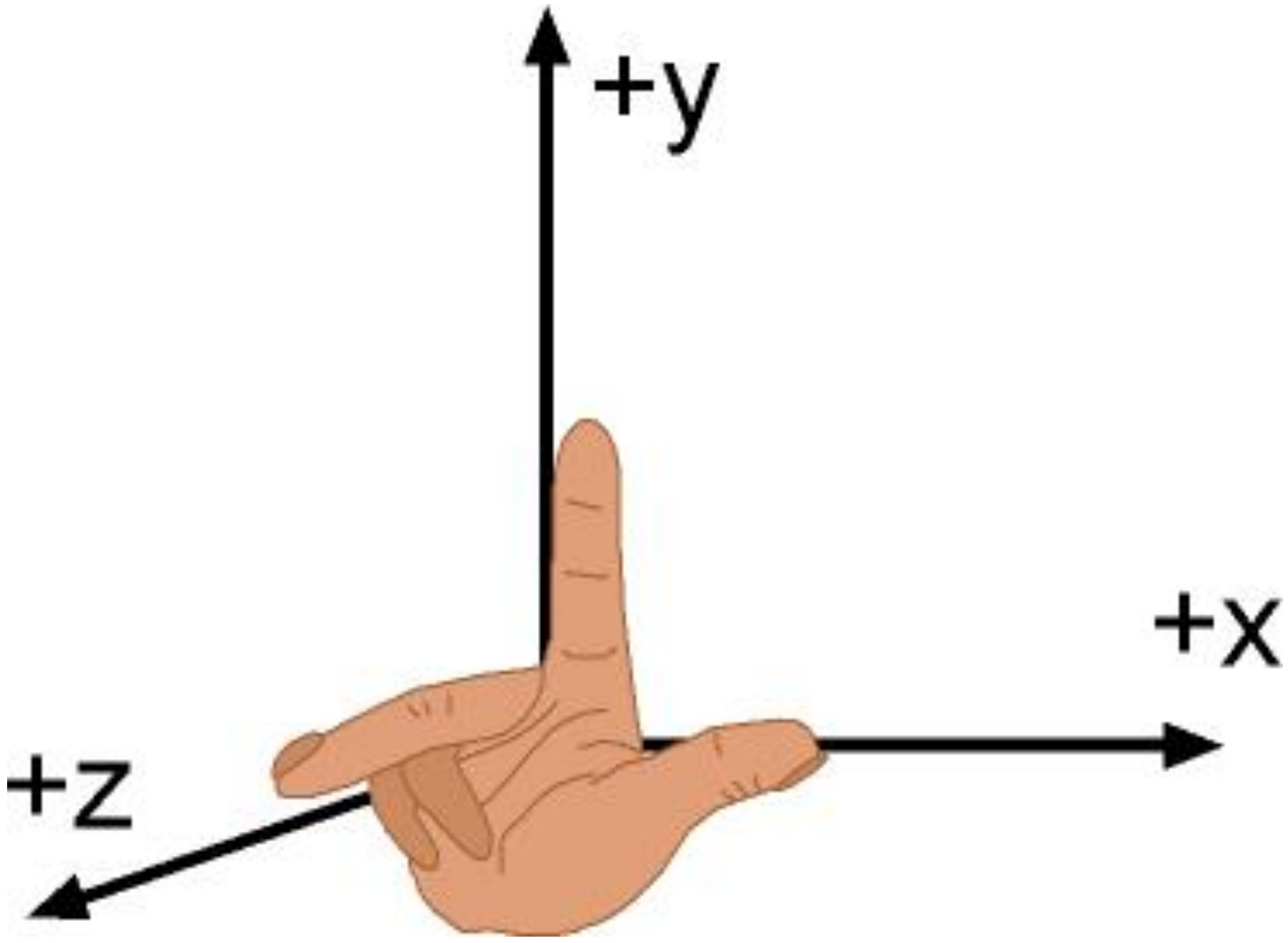
3D coordinates

- 3D coordinate systems can be left or right handed.



- 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



3D scenes

- We ultimately still want to produce a 2D image from our scenes. That introduces 2 new problems:
  - Depth
  - Projection
- This is our focus for today

Depth

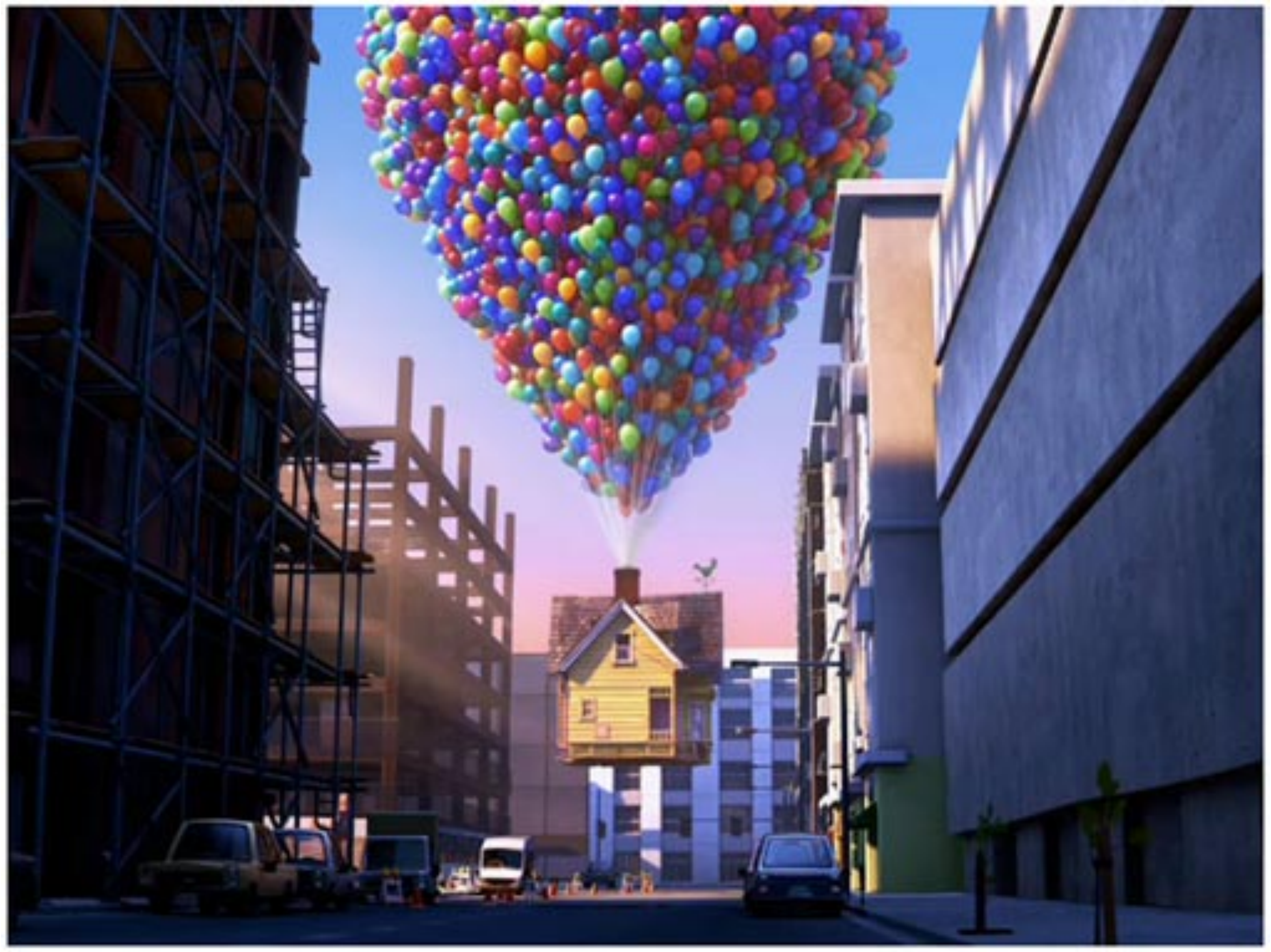
- By default OpenGL draws objects in the order in which they are generated in the code
- Doing that for a 3D scene is hard. How do you decide what order to draw things in?
- UNSWgraph now supports drawing with depth.
- See HelloTriangle3D.java

Projection

- How do we map geometry in a 3D space to a 2D image that “looks like” 3D to our human eyes and brains?

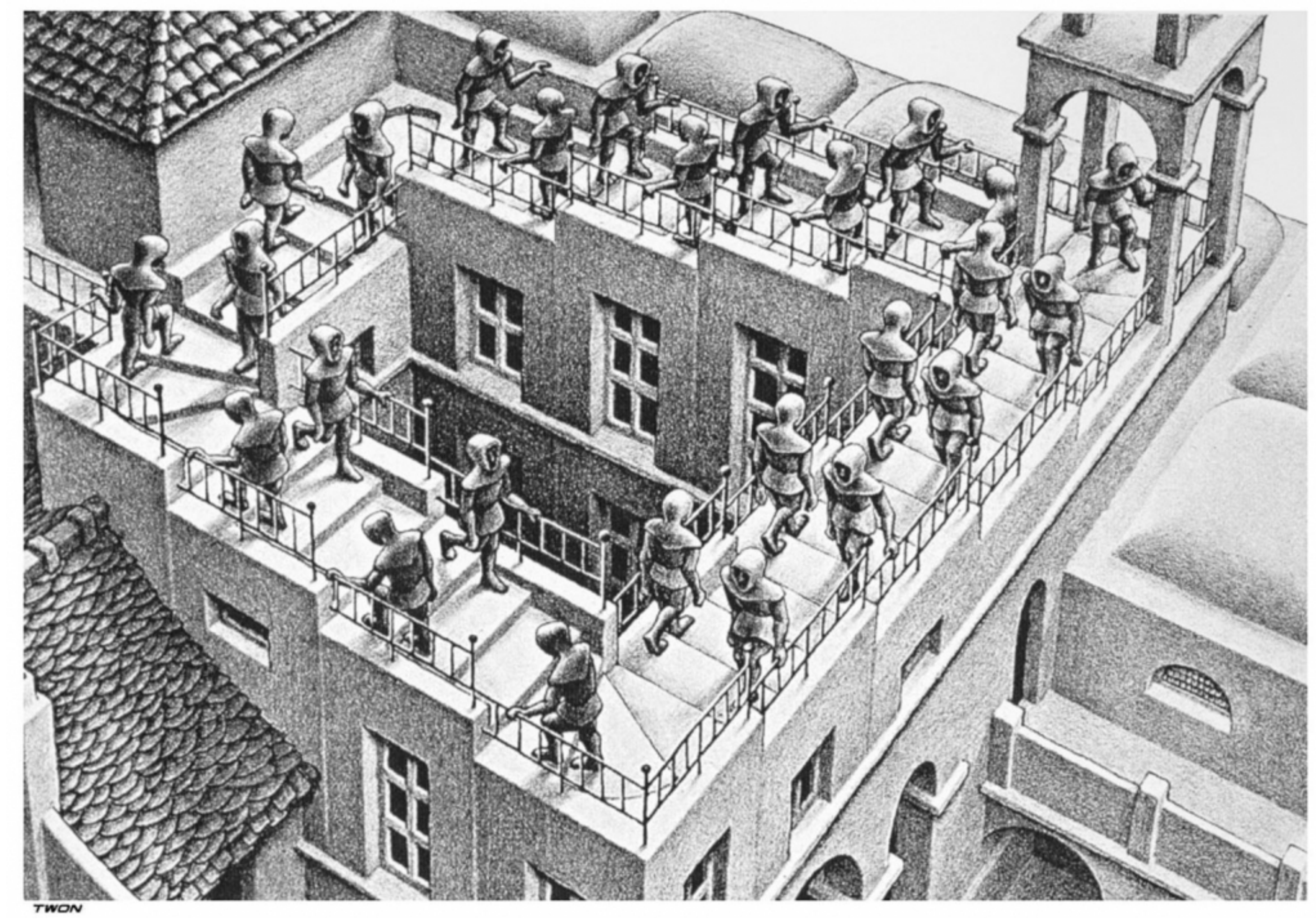
Projection

- We see this image as 3D even though it's really 2D.



Projection

- Our brain can be tricked



3D Transformations in UNSWgraph

```
public class CoordFrame3D {
    public CoordFrame3D translate(float x, float
y, float z) {…}
    public CoordFrame3D rotateX(float degrees)
{…}
    public CoordFrame3D rotateY(float degrees)
{…}
    public CoordFrame3D rotateZ(float degrees)
{…}
    public CoordFrame3D scale(float x, float y,
float z) {…}
}
```

3D transformations

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

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

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

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:

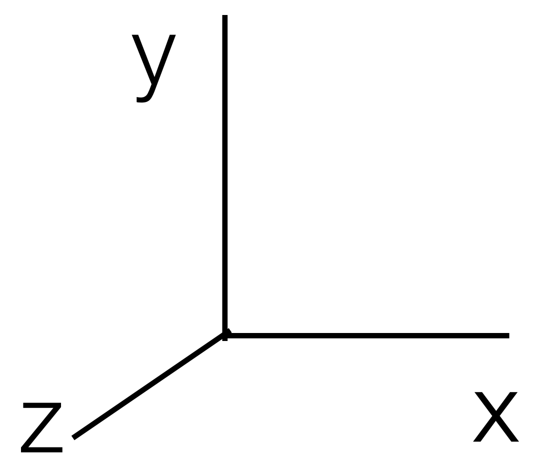
$$\mathbf{M_H} = \begin{pmatrix} 1 & h & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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.

3D Rotation

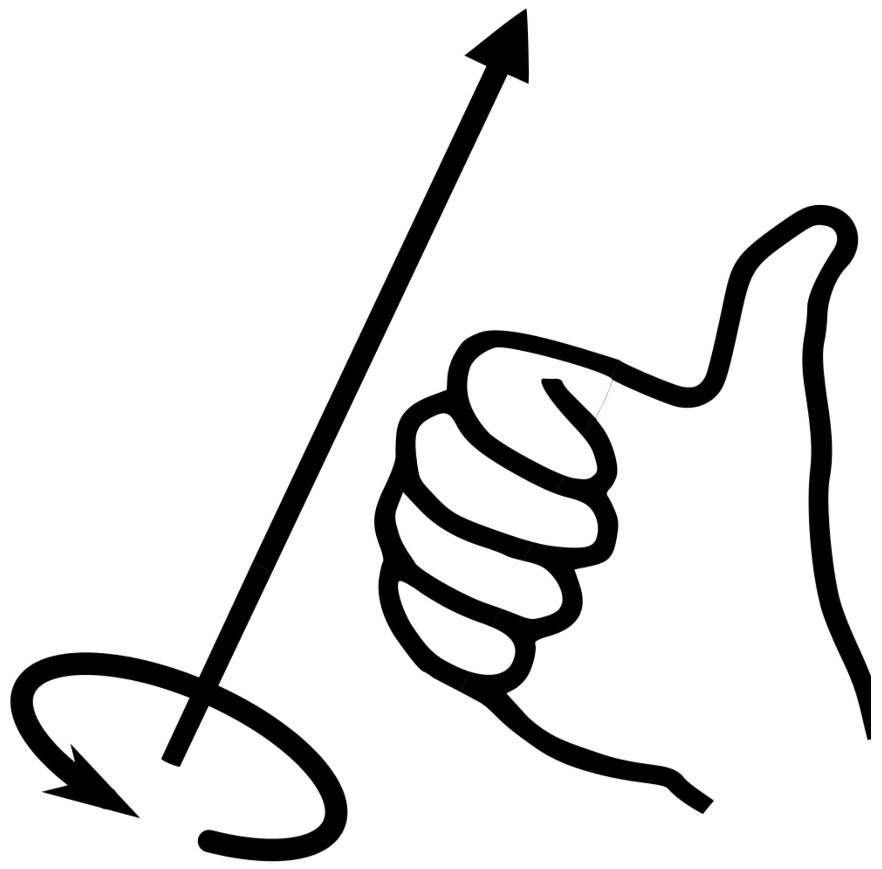
- In each case, positive rotation is CCW from the next axis towards the previous axis.
  - Mx rotates y towards z
  - My rotates z towards x
  - Mz rotates x towards y



- 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

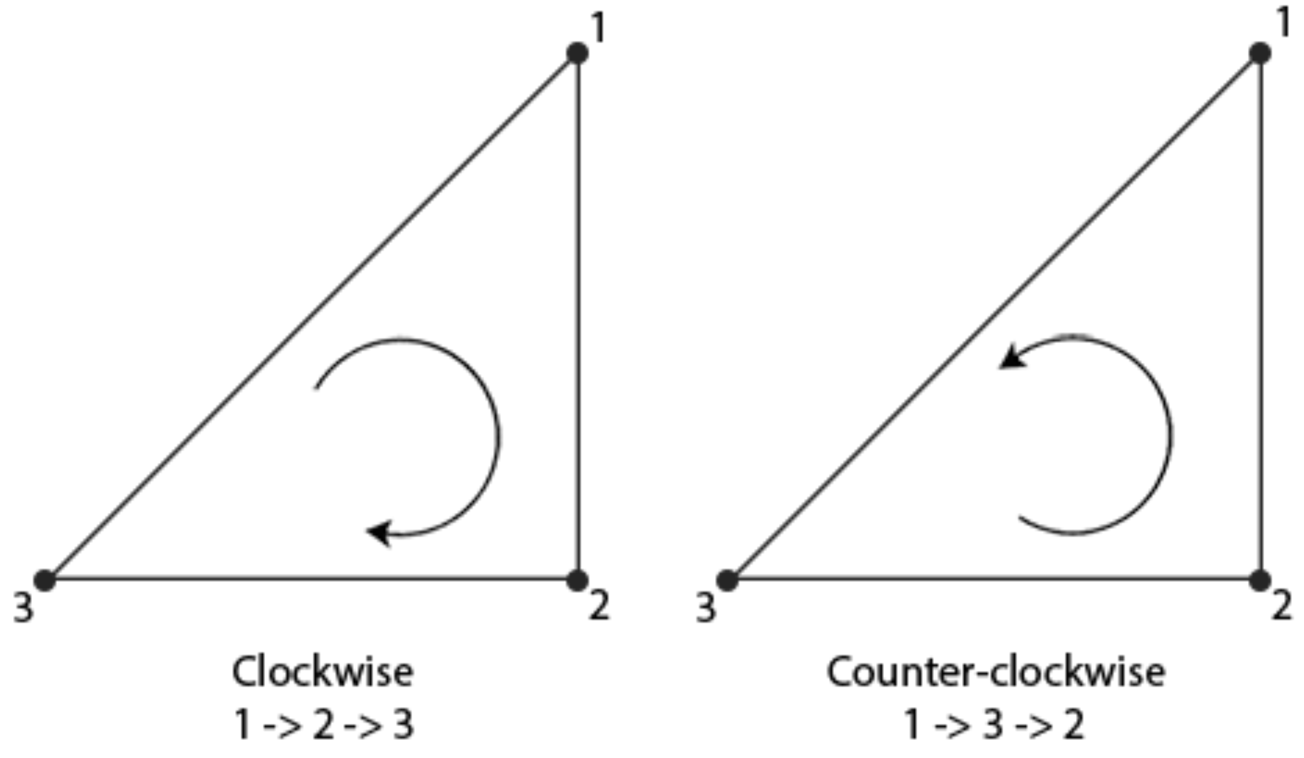
$$\mathbf{M_z} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

My first 3D object: A Cube

- We can create a cube by creating a face and drawing it 6 times in different rotated coordinate frames.
- We use a triangle fan for the face.
- See Cube.java

Winding Order

- By default, triangles are defined with counter-clockwise vertices are processed as front-facing triangles. Clockwise are processed as back-facing triangles.



Back face culling

- An optimisation called face culling allows non-visible triangles of closed surfaces (back faces) to be culled before rasterisation.
- This avoids unnecessary fragments from being created.
- This is based on the winding order of the triangle – front faces are CCW by default.

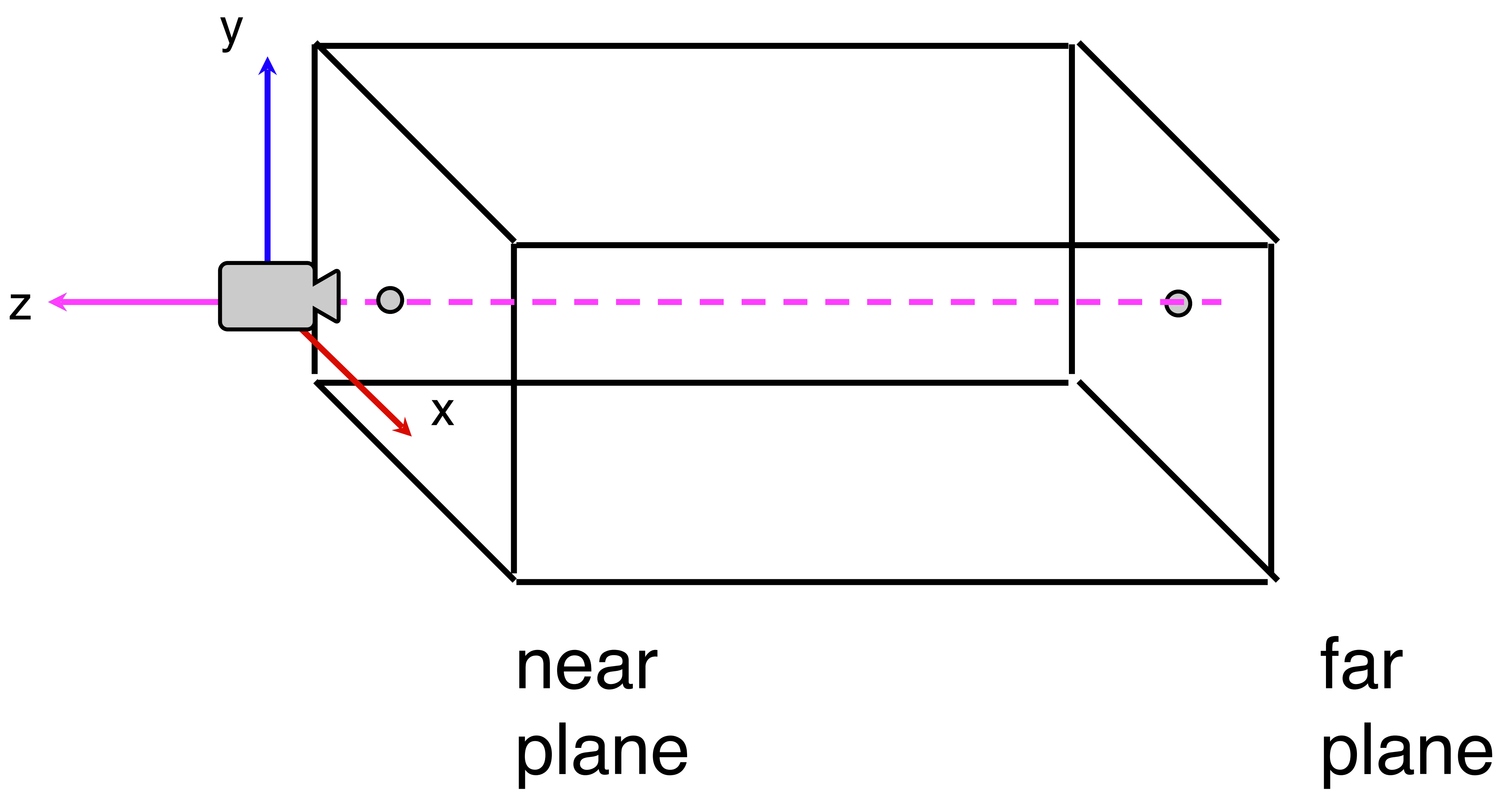
Back face culling

```
// Disabled by default

// Turned on in Application3D

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

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



Orthographic projection in UNSWgraph

- In Matrix4:

```
public static Matrix4 orthographic(float left, float right,
    float bottom, float top, float near, float far)
```

- In Shader:

```
public static void setProjMatrix(GL3 gl, Matrix4 mat)
```

- Example (from Application3D):

```
Shader.setProjMatrix(gl,
    Matrix4.orthographic(-1, 1, -1, 1, 1, 10));
```

Orthographic Projection

- The camera is located at the origin in camera co-ordinates and 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

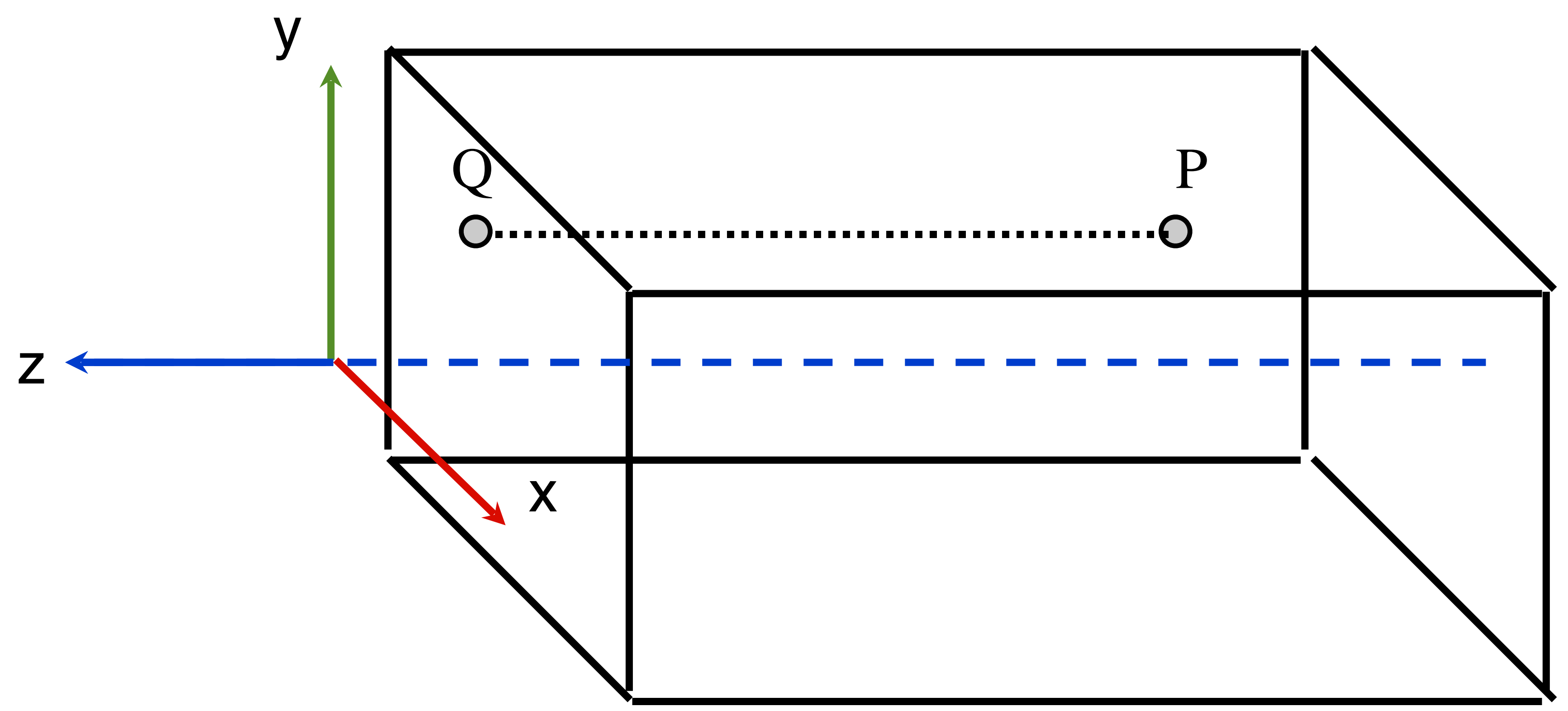
Projection

- We want to project a point from our 3D view volume onto the near plane, which will then be mapped to the viewport.
- Projection happens after the model and view transformations have been applied, so all points are in camera coordinates.
- Points with negative z values in camera coordinates are in front of the camera.

Orthographic projection

- The orthographic projection is simple:

$$q_x = p_x$$
$$q_y = p_y$$

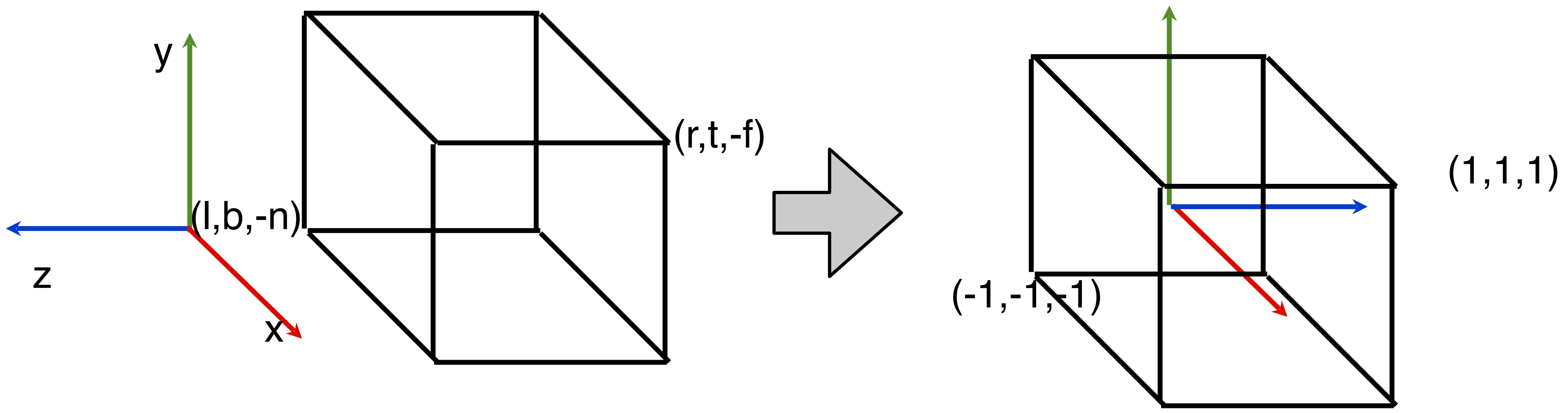


What about depth?

- 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.

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 clipping coordinates

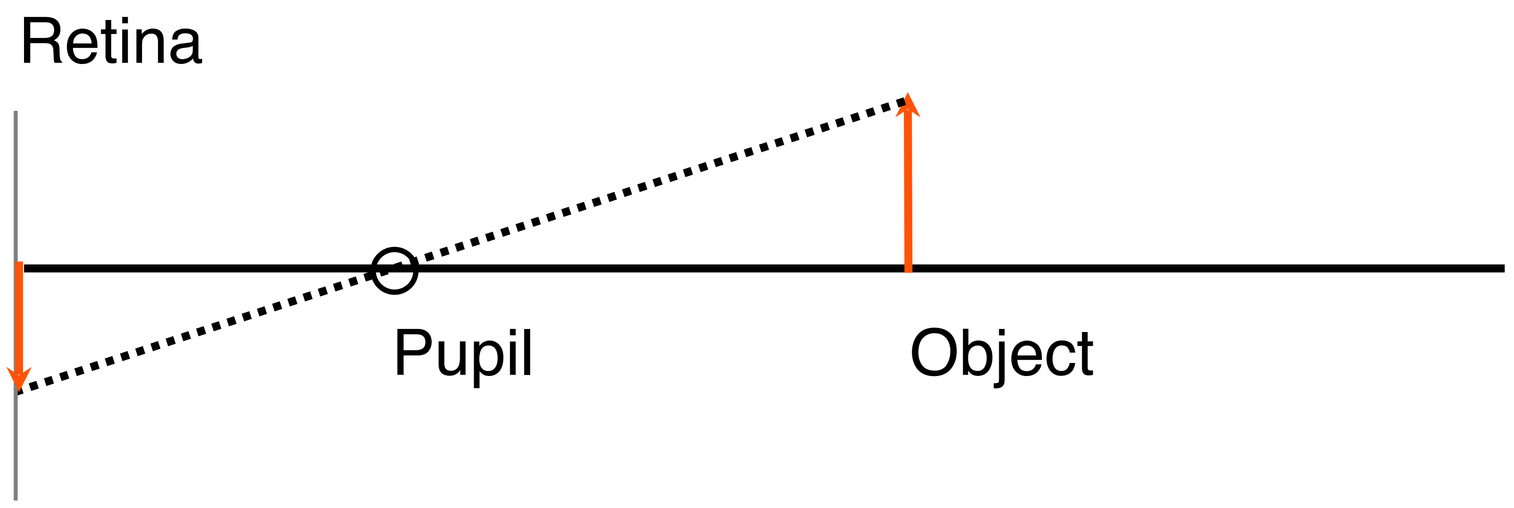
$$\mathbf{M_O} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

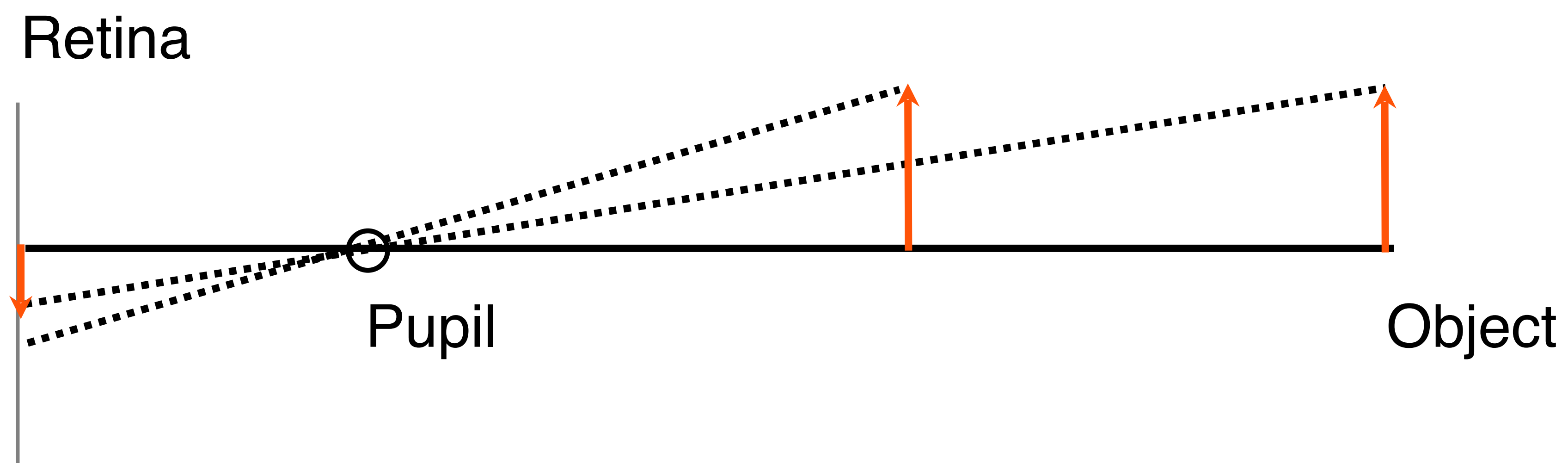
Foreshortening

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



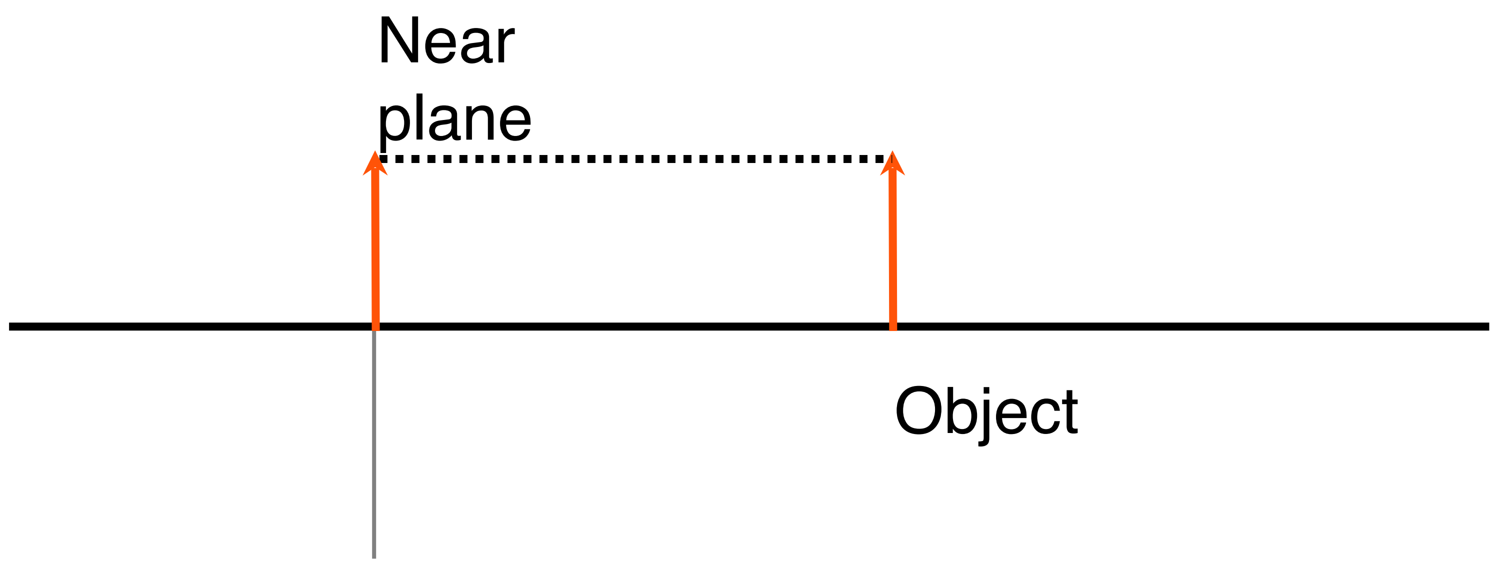
Foreshortening

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



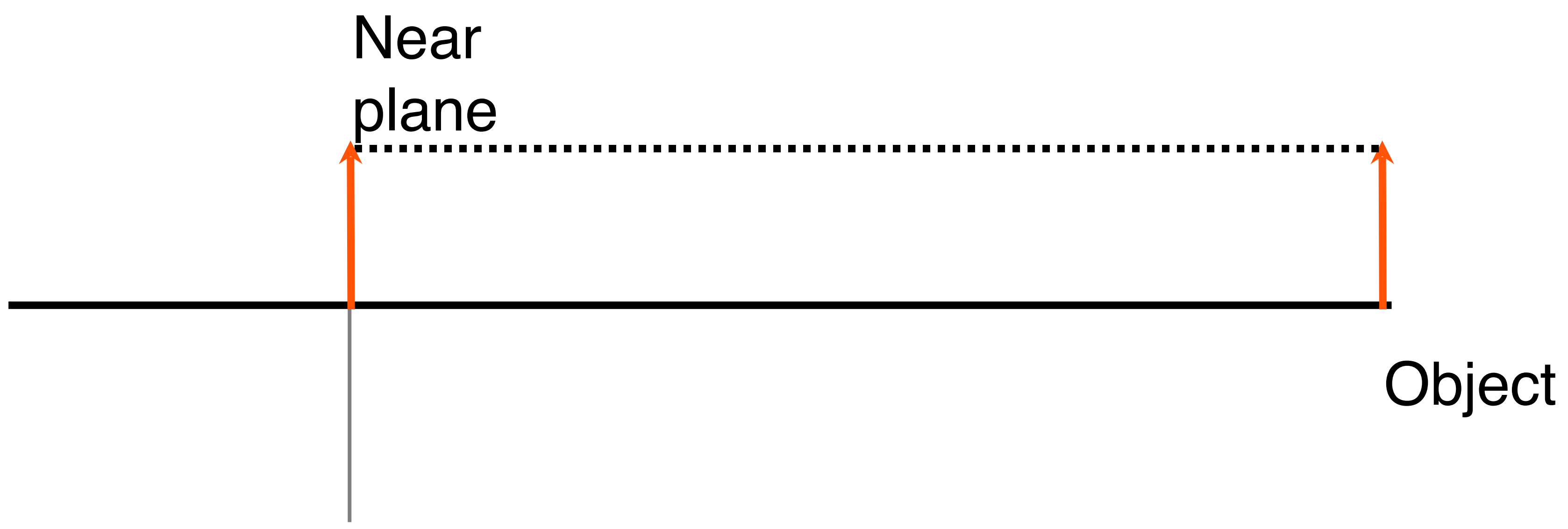
Orthographic camera

- The orthographic camera does not perform foreshortening.
- Objects size is independent of distance from the camera.

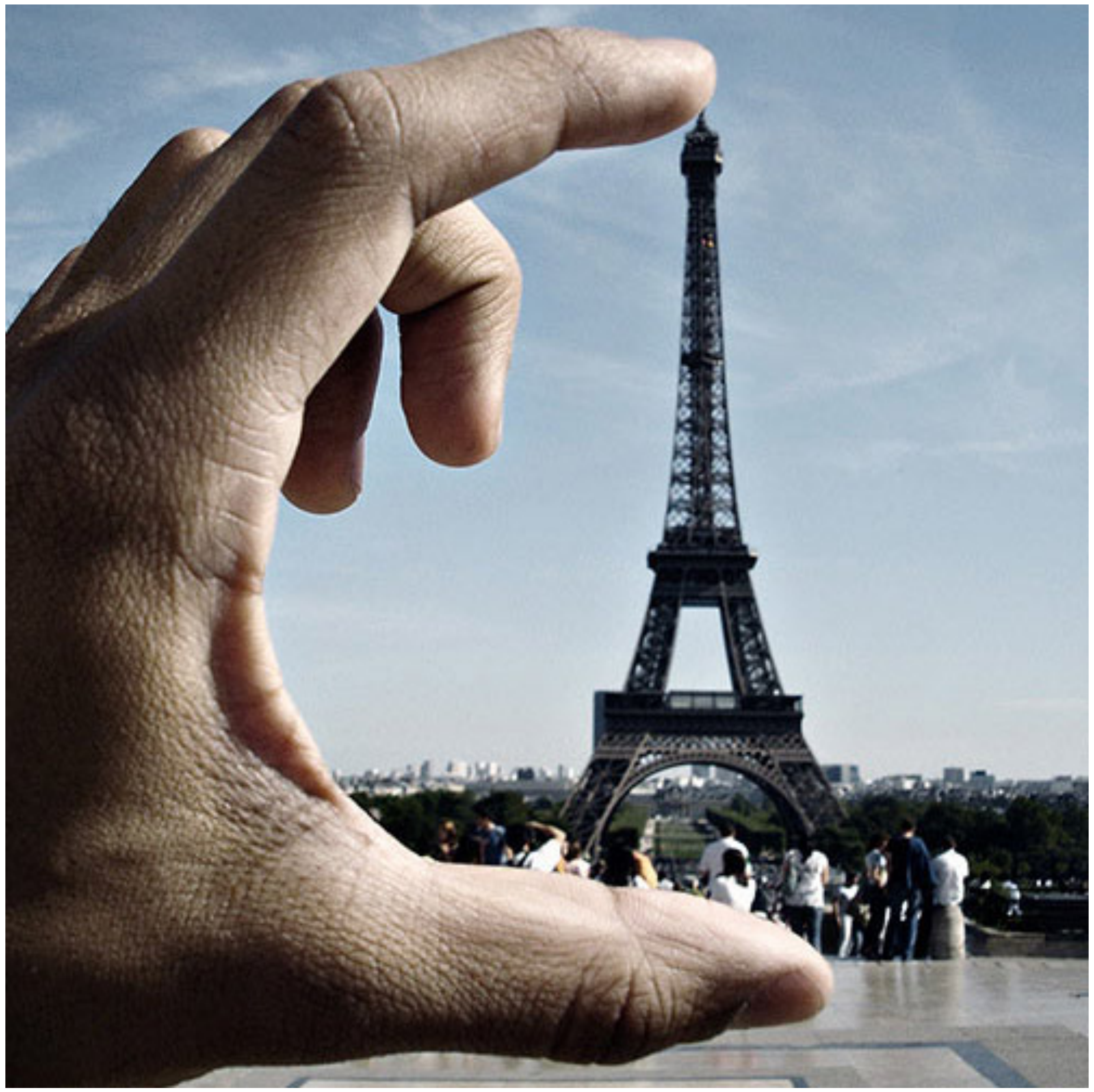


Orthographic camera

- The orthographic camera does not perform foreshortening.
- Objects size is independent of distance from the camera.

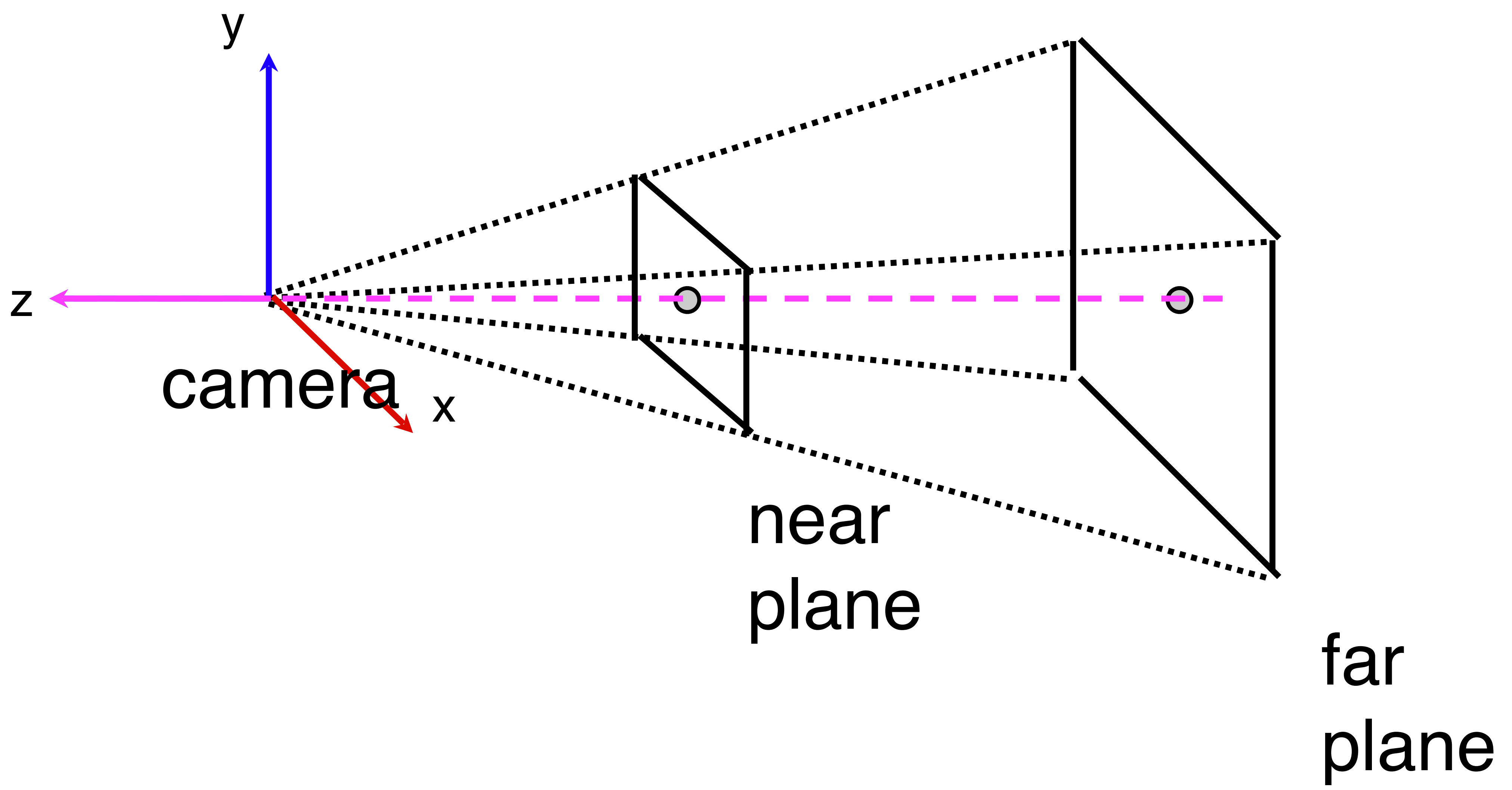


Perspective

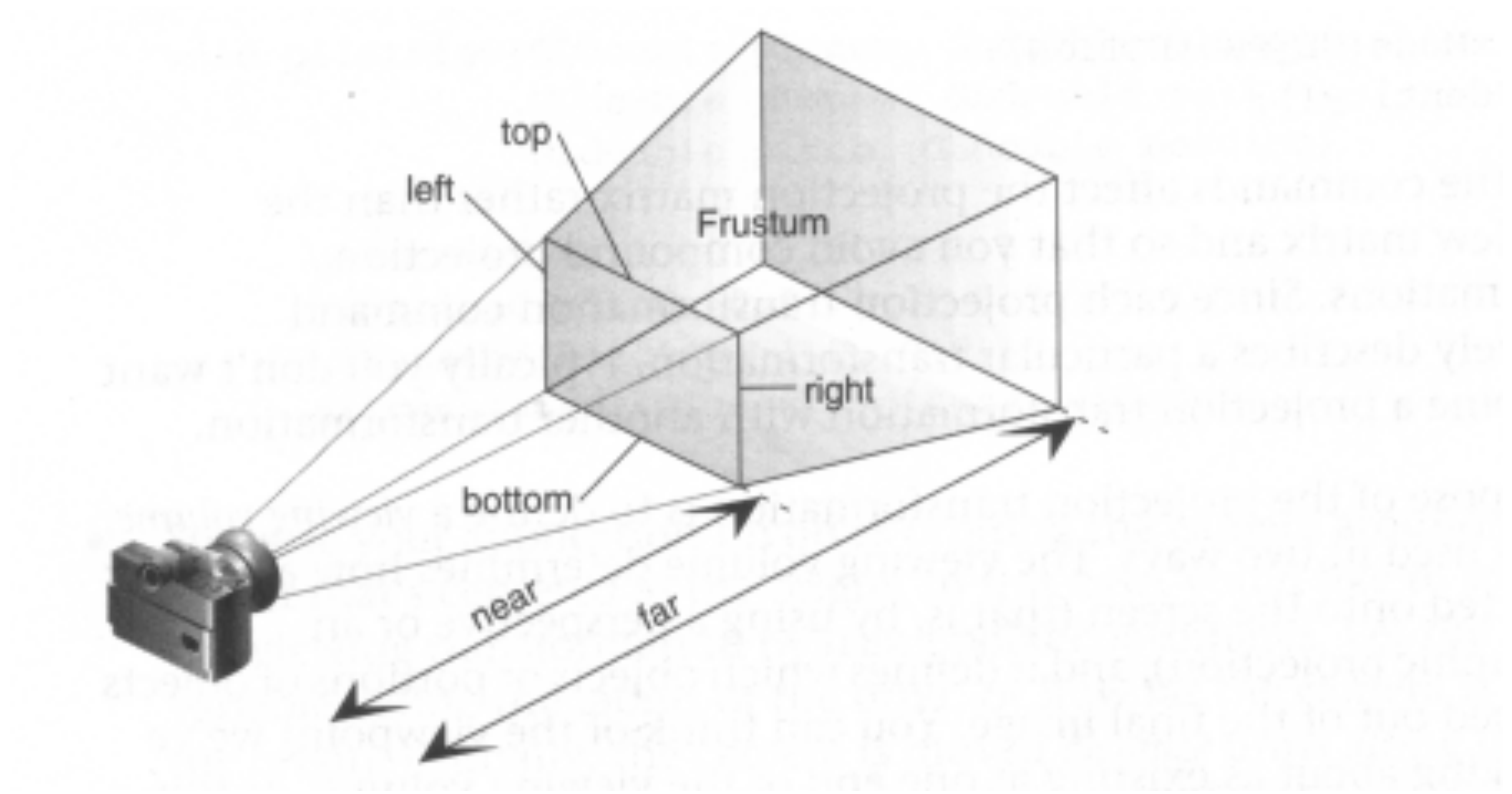


Perspective camera

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



Frustum



Frustums in UNSWgraph

- In Matrix4:

```
public static Matrix4 frustum(float left, float right,
    float bottom, float top, float near, float far) {
```

- An example from Cube.java:

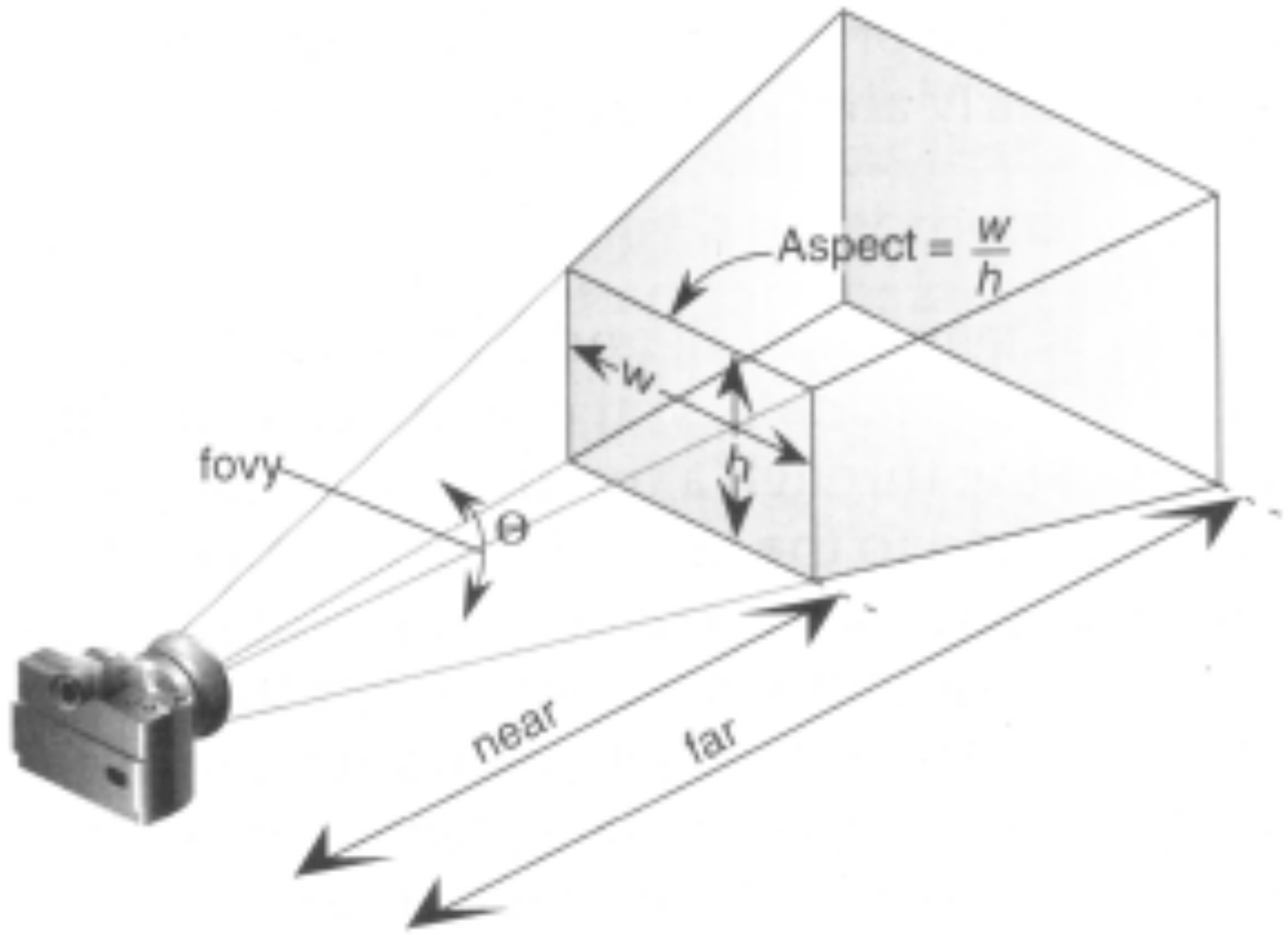
```
Shader.setProjMatrix(gl, Matrix4.frustum(-1, 1, -1, 1, 1,
100));
```

Perspective

- Defining the frustum you want is tricky.
  - What should the boundaries be?
- If we define a frustum projection with this method instead, it’s easier.

```
public static Matrix4 perspective(float fovy,
    float aspectRatio, float near, float far)
```

gluPerspective

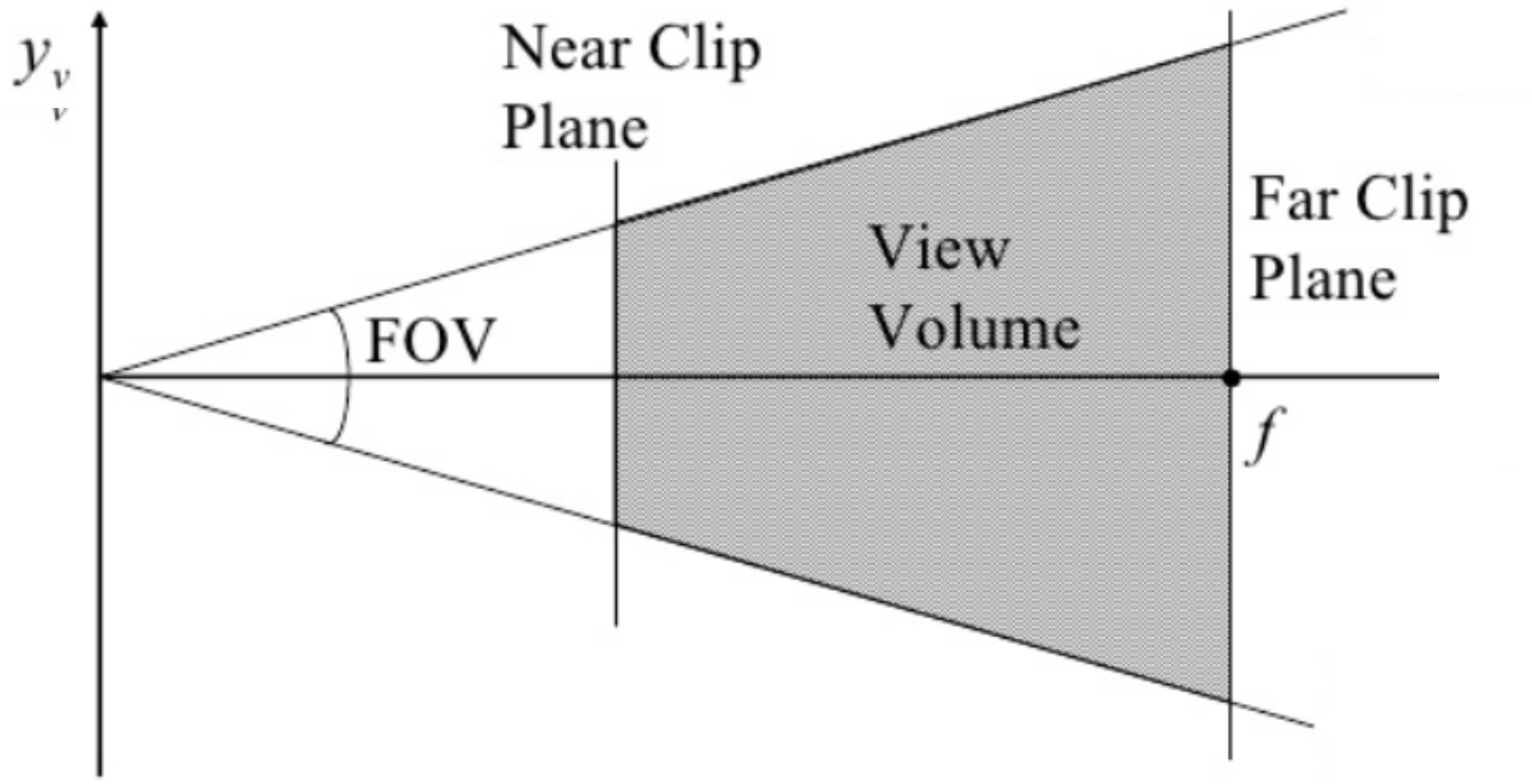


Side on View

Assuming a symmetric view volume

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

width = height * aspectRatio

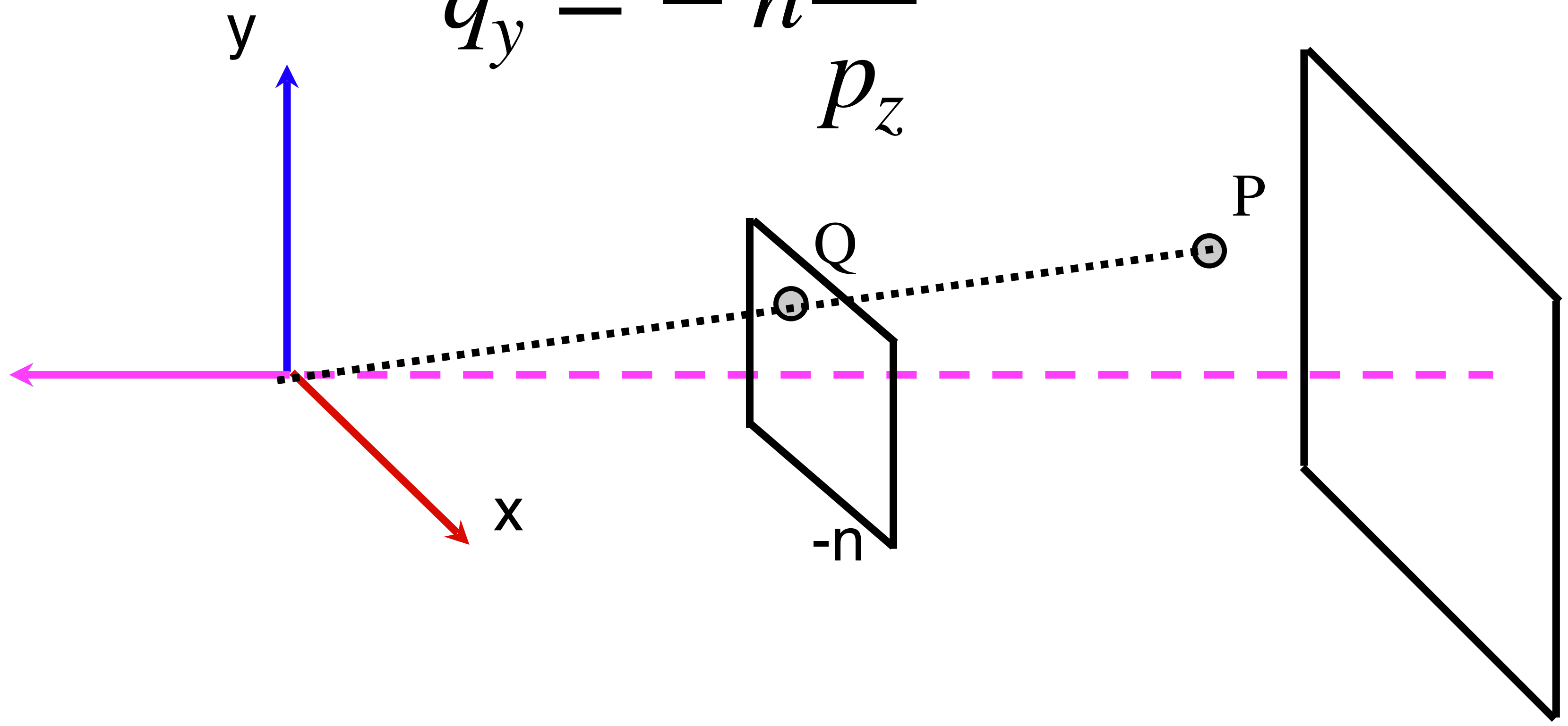


Perspective projection

- The perspective projection:

$$q_x = -n\frac{p_x}{p_z}$$

$$q_y = -n\frac{p_y}{p_z}$$



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.

Pseudodepth

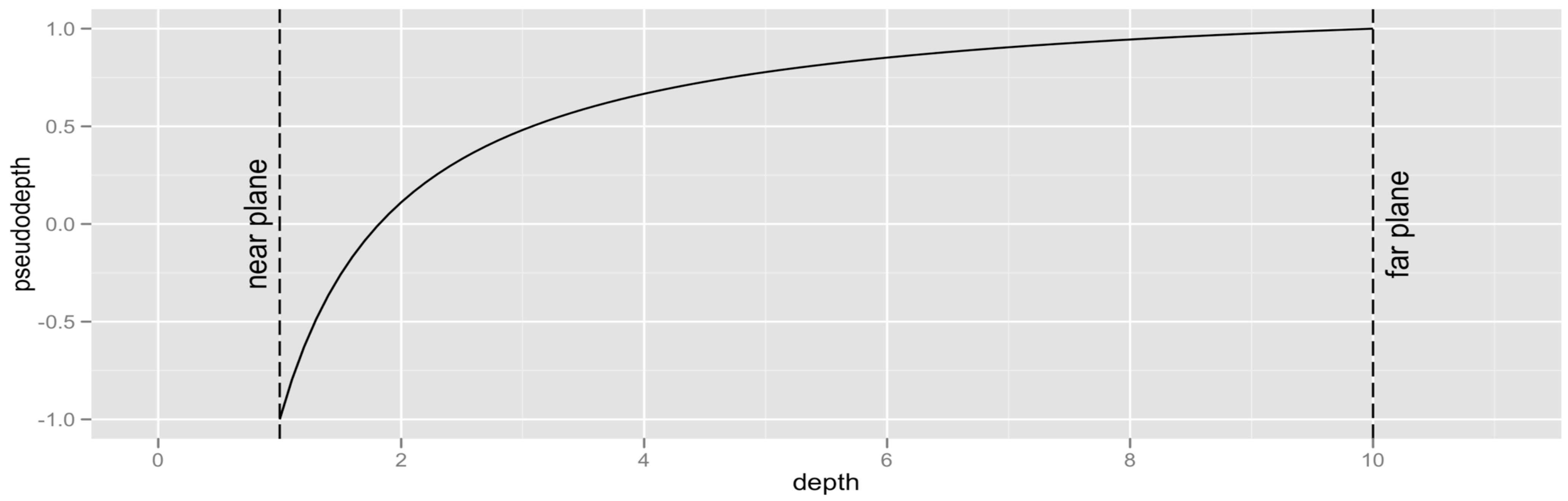
- These constraints yield an equation for pseudodepth:

$$q_z = \frac{ap_z + b}{-p_z}$$

$$a = -\frac{f+n}{f-n}$$

$$b = \frac{-2fn}{f-n}$$

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:

$$P = \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \equiv \begin{pmatrix} wp_x \\ wp_y \\ wp_z \\ w \end{pmatrix} \text{ for any } w \neq 0$$

- 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 we can divide a point by, say, it's z component.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 12 \\ 6 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 12 \\ 6 \\ 3 \\ 3 \end{pmatrix} \equiv \begin{pmatrix} 4 \\ 2 \\ 1 \\ 1 \end{pmatrix}$$

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.

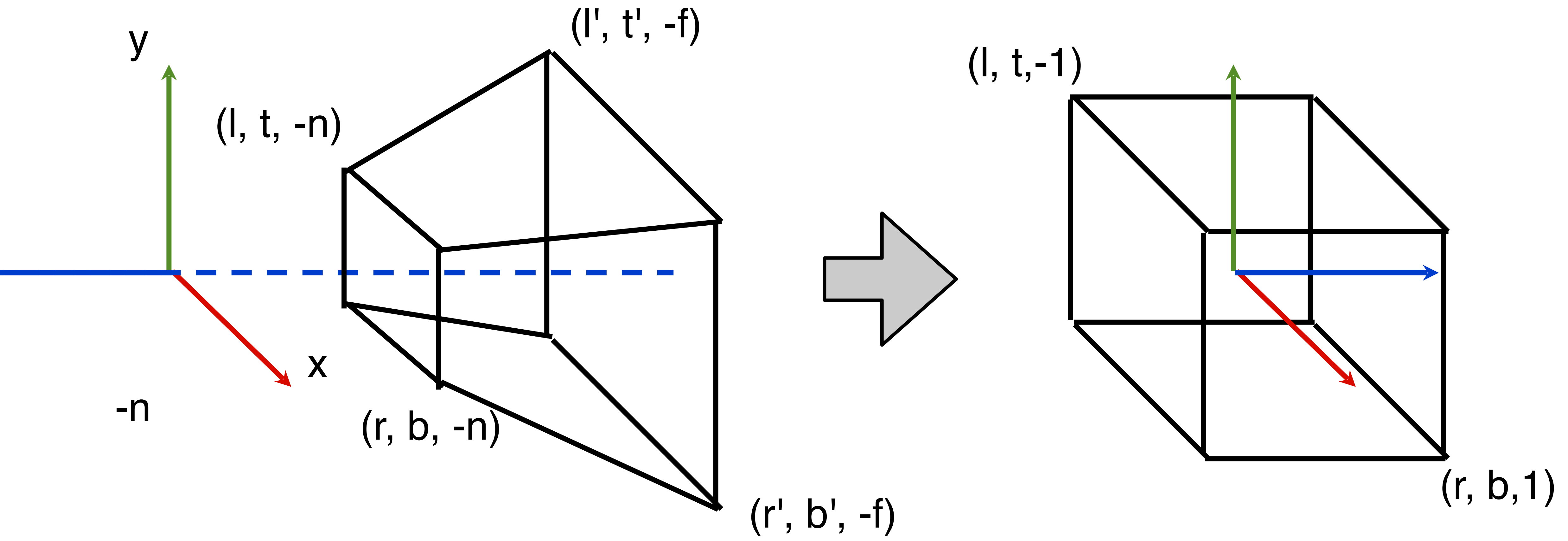
Perspective transform

- To transform a point:

$$
\begin{aligned}
q &= M_{perspective} P \\
&= \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} np_x \\ np_y \\ ap_z + b \\ -p_z \end{pmatrix} \equiv \begin{pmatrix} -np_x/p_z \\ -np_y/p_z \\ -(ap_z + b)/p_z \\ 1 \end{pmatrix}
\end{aligned}
$$

Perspective transform

- This matrix maps the perspective view volume to an axis aligned cube
- Note the z-axis has been flipped.



Perspective projection matrix

- We combine perspective transformation and scaling into a single matrix to map it into the canonical view volume for clipping:

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

Perspective division

- After clipping all points are converted to the form with the fourth component equal to one. These are called normalized device co-ordinates.

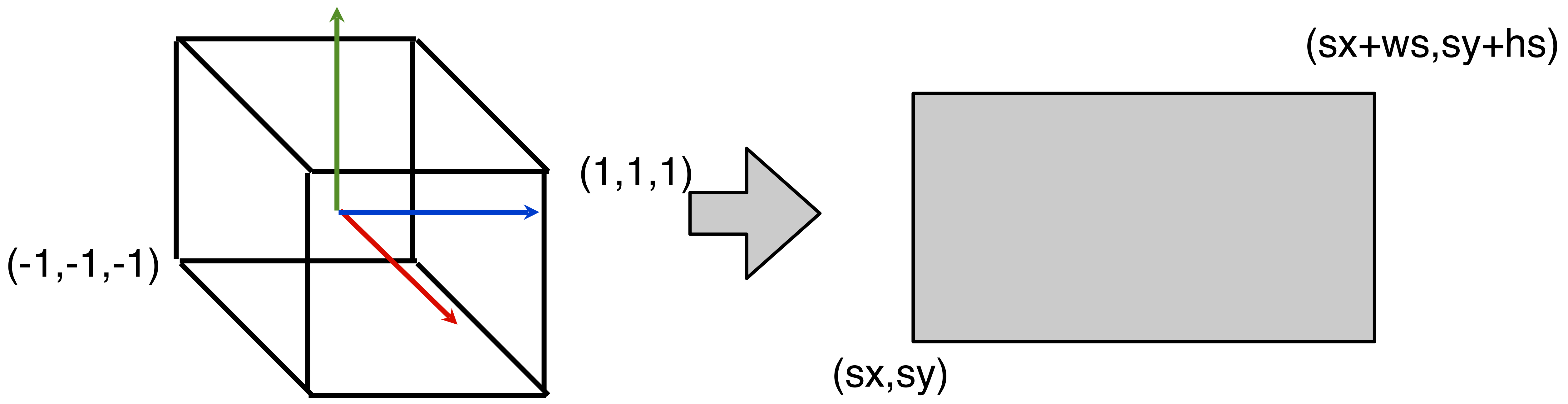
- This is called the perspective division step.

$$Q = \frac{1}{w}\begin{pmatrix} p_x \\ p_y \\ p_z \\ w \end{pmatrix}$$

Viewport transformation

- Finally points are scaled into window coordinates corresponding to pixels on the screen.

- It also maps pseudodepth from -1..1 to 0..1



Viewport transformation

- Again, this can be done with a matrix:

$$M_{\text{viewport}} = \begin{pmatrix} \frac{w_s}{2} & 0 & 0 & s_x + \frac{w_s}{2} \\ 0 & \frac{h_2}{2} & 0 & s_y + \frac{h_s}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The transformation pipeline

- To transform a point:

$$P = (p_x, p_y, p_z)^\top$$

- Extend to homogeneous coordinates:

$$P = (p_x, p_y, p_z, 1)^\top$$

- 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 transformation pipeline

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

$$P_{cvv} = \mathbf{M_P} P_c$$

- Clip to remove points outside CVV.

- Perspective division to eliminate fourth component.

$$P_n = \frac{1}{p_w} P_{cvv}$$

- Viewport transformation to window coordinates.

$$P_v = \mathbf{M_{viewport}} P_n$$

The transformation pipeline

- In vertex shaders, gl_Position is the point in CVV coordinates.

- The subsequent stages of transformation are (by default) performed internally by the OpenGL implementation

Positioning a 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.
- Unlike before, the camera does not have an aspect ratio. That is handled by the projection transform.
- The camera points backwards down its local z-axis.

Fragments

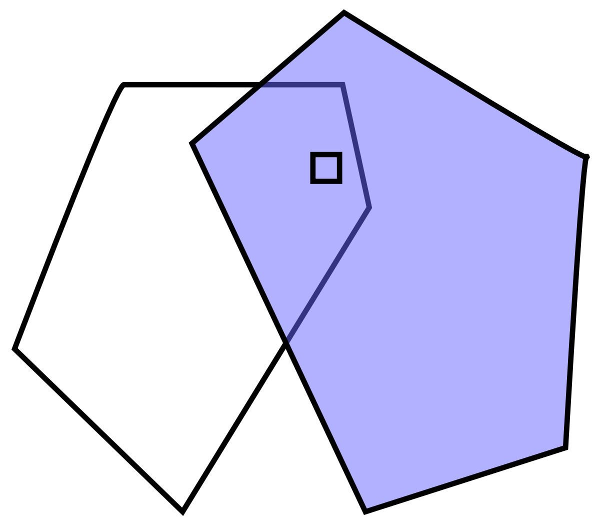
- Rasterisation converts triangles into collections of fragments.
- Any triangles that are culled (eg back face culling) are discarded and not converted into fragments.
- A fragment corresponds to a single image pixel.
- A fragment may not make it to the final image if it is discarded by depth testing

Hidden surface removal

We have a coordinates of fragments expressed in screen coordinates and pseudodepth.

For any particular pixel on the screen, we need to know what we can see at that pixel.

Some fragments may be behind other fragments and should not be seen.



Hidden Surface Removal

We will look at 2 approaches to this problem:

1. Make sure all triangles (and therefore fragments) are drawn in the correct order in terms of depth. This is done at the model level not at the fragment level. This is not built into OpenGL.

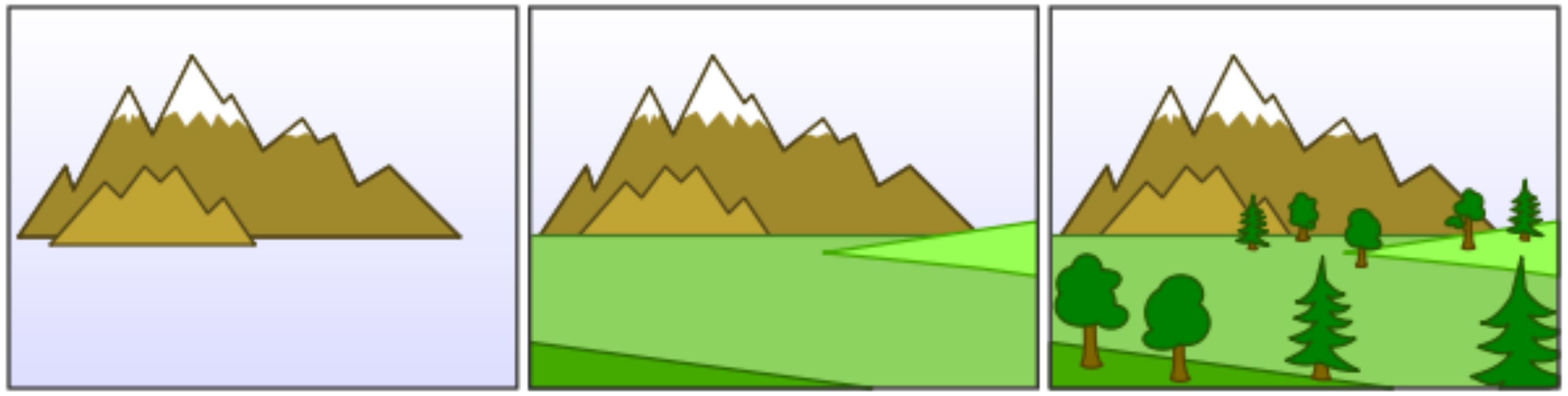
2. Use the depth buffer. This is done at the fragment level and is built into OpenGL

Painter's algorithm

The naive approach to hidden surface removal:

- Sort geometric primitives by depth
- Draw in order from back to front

This is known as the Painter's algorithm



Painter's algorithm

What if each triangle does not have a single z value?

Which one do you sort on?

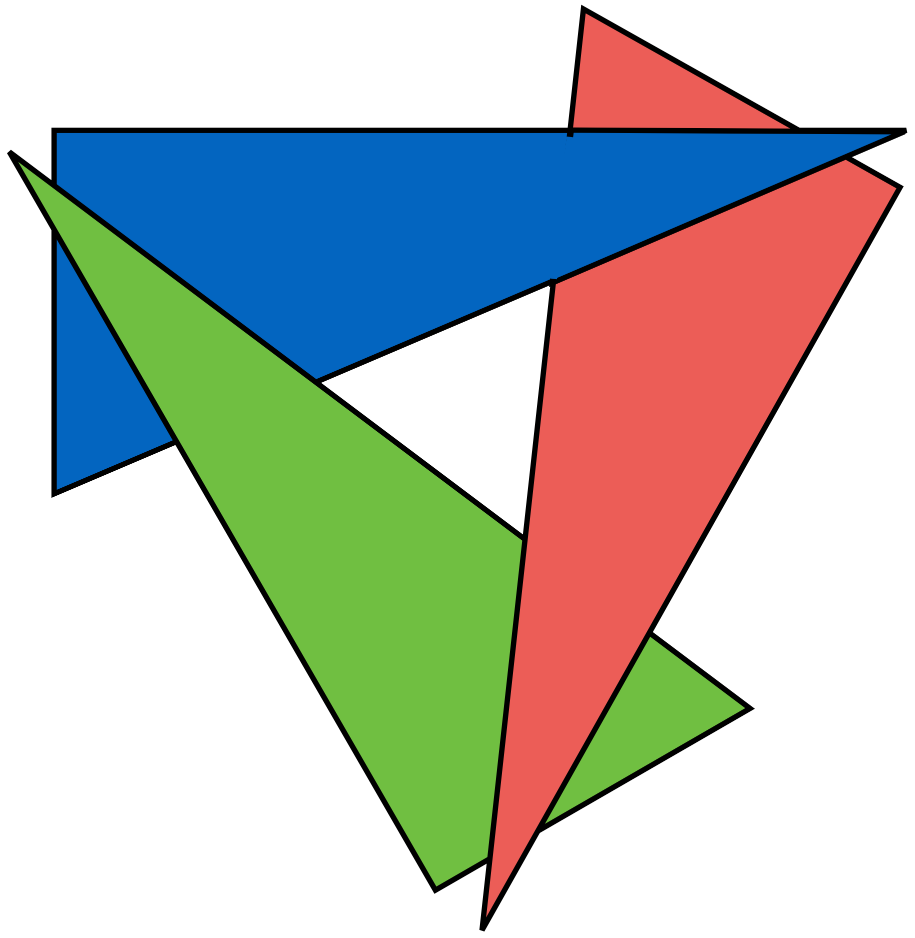
- The center of the polygon?
- The nearest vertex?
- The furthest?

Problem

What about triangles that intersect?

Which one to paint first?

We need to split them into pieces.



BSP Trees

One possible solution is to use Binary Space Partitioning trees (BSP trees)

These are **NOT** implemented in OpenGL

They recursively divide the world into polygons that are behind or in front of other polygons and split polygons when necessary.

Then it is easy to traverse and draw polygons in a front to back order

BSP Trees

This invention made first person shooters possible (i.e. the original Doom).

Building the tree is slow and it needs to be rebuilt every time the geometry changes.

Best for rendering static geometry where tree can just be loaded in.

More on BSP Trees

Depth buffer

Another approach to hidden surface removal is to keep per-pixel depth information.

This is what OpenGL uses.

This is stored in a block of memory called the depth buffer (or z-buffer).

d[x][y] = pseudo-depth of pixel (x,y)

OpenGL

```
// in init()
gl.glEnable(GL2.GL_DEPTH_TEST);

// in display()
gl.glClear(
     GL.GL_COLOR_BUFFER_BIT |
     GL.GL_DEPTH_BUFFER_BIT);
```

Depth buffer

Initially the depth buffer is initialised to 1 (maximum depth in window coords).

Each polygon is drawn fragment by fragment.

Each fragment has its pseudodepth compared to the value in the depth buffer.

If it is closer, we update the pixel in the colour buffer and update the buffer value to the new pseudodepth. If not we discard it.

Pseudocode

```
Initialise db[x][y] = 1 for all x,y

For each triangle:
    For each fragment (px,py):
        d = pseudodepth of (px,py)

        if (d < db[px][py]):
            draw fragment
            db[x][y] = d
```

Example

Triangles

.1									
.1	.1								
.2	.2	.2							
.2	.2	.2	.2						
.3	.3	.3	.3	.3					
.3	.3	.3	.3	.3	.3				
.4	.4	.4	.4	.4	.4	.4			
.4	.4	.4	.4	.4	.4	.4	.4		
.5	.5	.5	.5	.5	.5	.5	.5	.5	
.5	.5	.5	.5	.5	.5	.5	.5	.5	.5

Depth Buffer

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Example

Triangles

.1									
.1	.1								
.2	.2	.2							
.2	.2	.2	.2						
.3	.3	.3	.3	.3					
.3	.3	.3	.3	.3	.3				
.4	.4	.4	.4	.4	.4	.4			
.4	.4	.4	.4	.4	.4	.4	.4		
.5	.5	.5	.5	.5	.5	.5	.5	.5	
.5	.5	.5	.5	.5	.5	.5	.5	.5	.5

Depth Buffer

.1	1	1	1	1	1	1	1	1	1
.1	.1	1	1	1	1	1	1	1	1
.1	.2	.2	1	1	1	1	1	1	1
.2	.2	.2	.2	1	1	1	1	1	1
.3	.3	.3	.3	.3	1	1	1	1	1
.3	.3	.3	.3	.3	.3	1	1	1	1
.4	.4	.4	.4	.4	.4	.4	1	1	1
.4	.4	.4	.4	.4	.4	.4	.4	1	1
.5	.5	.5	.5	.5	.5	.5	.5	.5	1
.5	.5	.5	.5	.5	.5	.5	.5	.5	.5

Example

Triangles

				.7	.7	.7	.7	.7	.7
				.6	.6	.6	.6	.6	.6
				.5	.5	.5	.5	.5	.5
				.4	.4	.4	.4	.4	.4
				.3	.3	.3	.3	.3	.3
				.2	.2	.2	.2	.2	.2

Depth Buffer

.1	1	1	1	1	1	1	1	1	1
.1	.1	1	1	1	1	1	1	1	1
.2	.2	.2	1	1	1	1	1	1	1
.2	.2	.2	.2	1	1	1	1	1	1
.3	.3	.3	.3	.3	1	1	1	1	1
.3	.3	.3	.3	.3	.3	1	1	1	1
.4	.4	.4	.4	.4	.4	.4	1	1	1
.4	.4	.4	.4	.4	.4	.4	.4	1	1
.5	.5	.5	.5	.5	.5	.5	.5	.5	1
.5	.5	.5	.5	.5	.5	.5	.5	.5	.5

Example

Triangles

				.7	.7	.7	.7	.7	.7
				.6	.6	.6	.6	.6	.6
				.5	.5	.5	.5	.5	.5
				.4	.4	.4	.4	.4	.4
				.3	.3	.3	.3	.3	.3
				.2	.2	.2	.2	.2	.2

Buffer

.1	1	1	1	1	1	1	1	1	1
.1	.1	1	1	1	1	1	1	1	1
.2	.2	.2	1	1	1	1	1	1	1
.2	.2	.2	.2	1	1	1	1	1	1
.3	.3	.3	.3	.3	.7	.7	.7	.7	.7
.3	.3	.3	.3	.3	.3	.6	.6	.6	.6
.4	.4	.4	.4	.4	.4	.4	.5	.5	.5
.4	.4	.4	.4	.4	.4	.4	.4	.4	.4
.5	.5	.5	.5	.3	.3	.3	.3	.3	.3
.5	.5	.5	.5	.2	.2	.2	.2	.2	.2