Chap. 5 3D Viewing and Projections

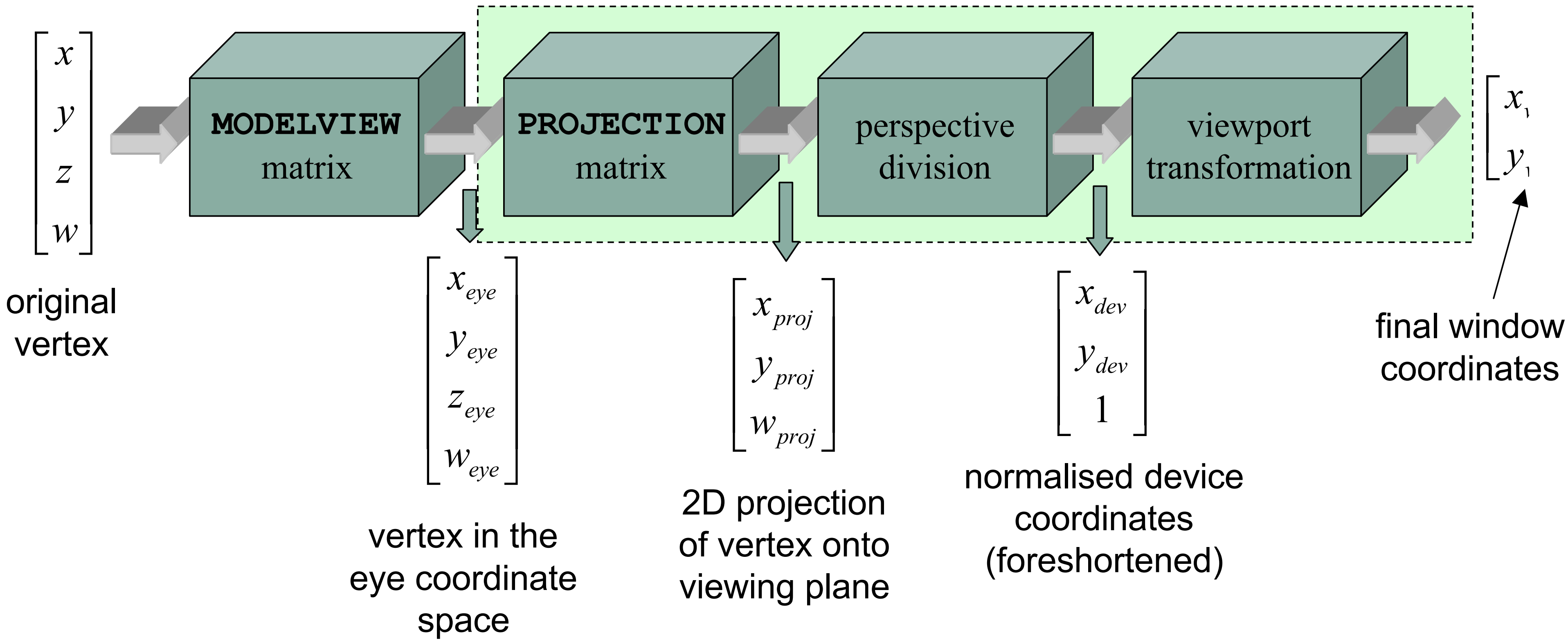
4BA6 - Topic 4

Dr. Steven Collins

References

- "*Computer graphics: principles & practice*", Foley, vanDam, Feiner, Hughes, S-LEN 500.1644 M23*1;1-6 (has a good appendix on linear algebra)
- *"Advanced Animation and Rendering Techniques"*, Watt and Watt, S-LEN 500.18 N26;2-5
- *"The OpenGL Programming Guide"*, Woo, Neider & Davis, S-LEN 500.18 N72;0-2
- *"Interactive Computer Graphics",* Edward Angel

OpenGL® Geometry Pipeline



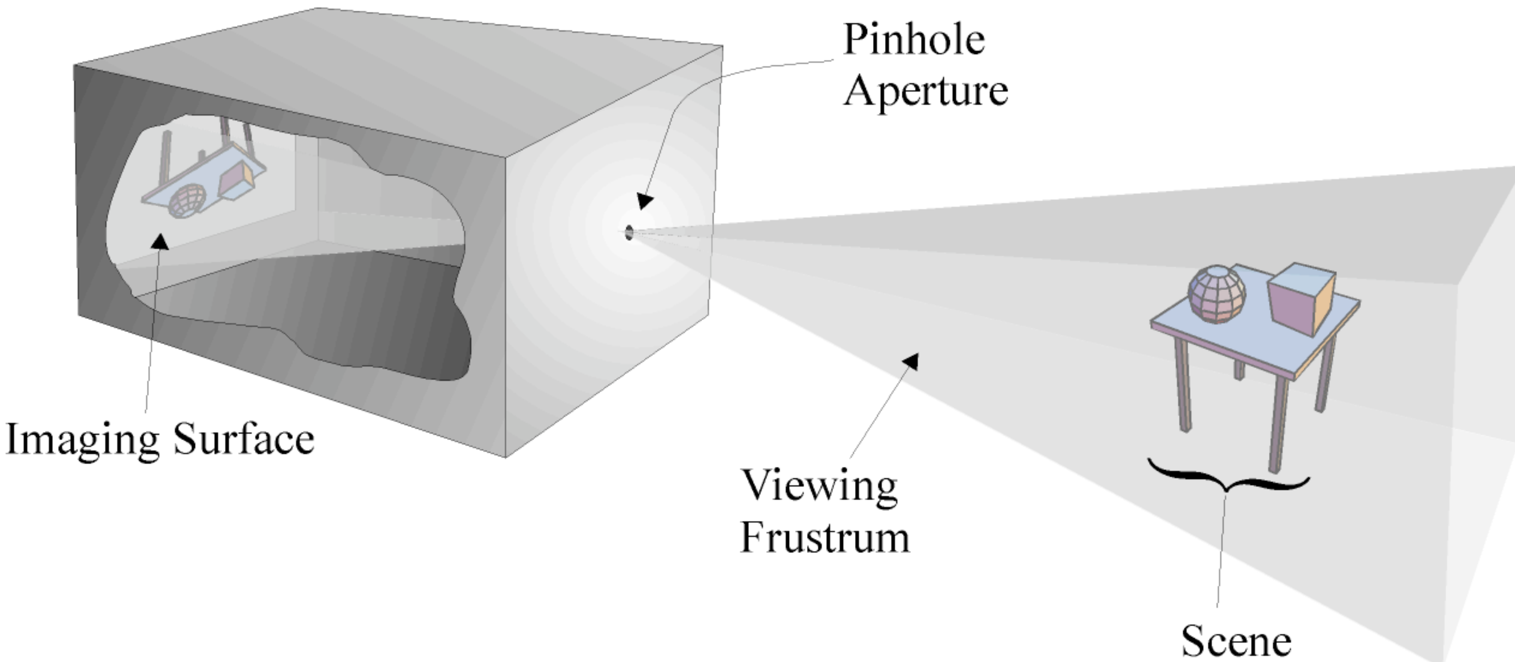
The Camera System

- To create a view of a scene we need:
  - **a description of the scene geometry**
  - **a camera or view definition**
- Default OpenGL camera is located at the origin looking down the ***-z*** axis.
- The camera definition allows *projection* of the 3D scene geometry onto a 2D surface for display.
- This projection can take a number of forms:
  - ***orthographic*** **(parallel lines preserved)**
  - ***perspective*** **(foreshortening):** ***1-point*****,** ***2-point*** **or** ***3-point***
  - ***skewed orthographic***

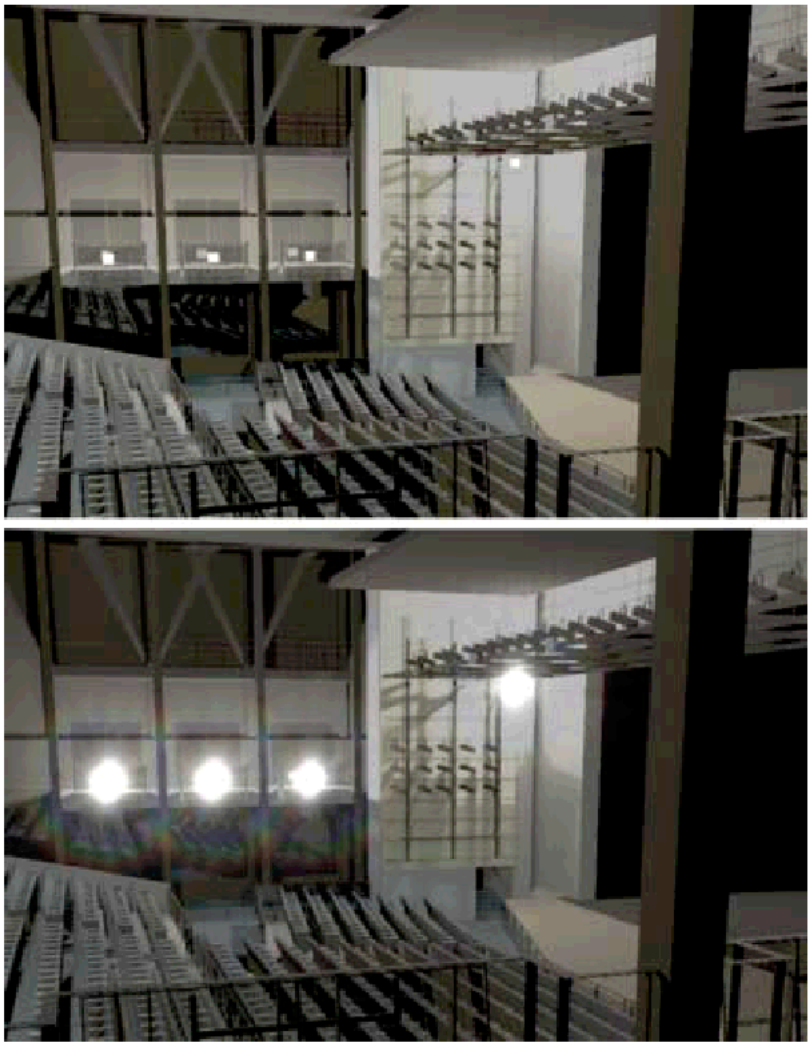
Camera Types

- Before generating an image we must choose our viewer:
- The *pinhole camera model* is most widely used:
  - infinite *depth of field* (everything is in focus)
- Advanced rendering systems model the camera
  - *double gauss lens* as used in many professional cameras
  - model depth of field and non-linear optics (including *lens flare*)
- *Photorealistic rendering systems* often employ a physical model of the eye for rendering images
  - model the eyes response to varying *brightness* and *colour* levels
  - model the internal optics of the eye itself (*diffraction* by lens fibres etc.)

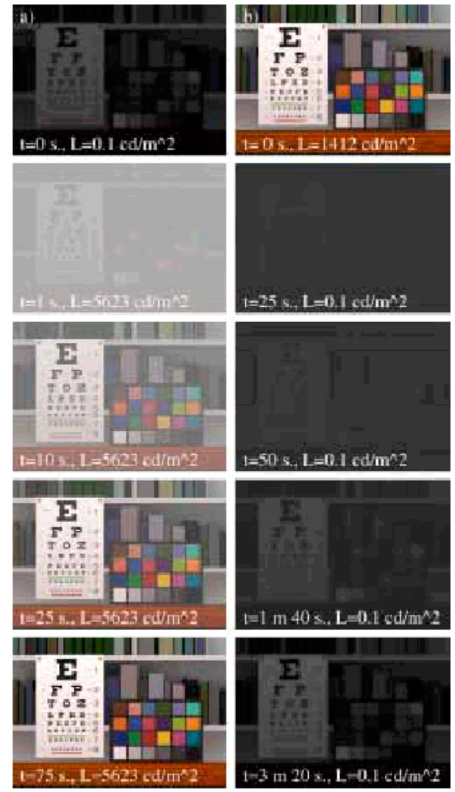
Pinhole Camera Model



Modeling the Eye's Response

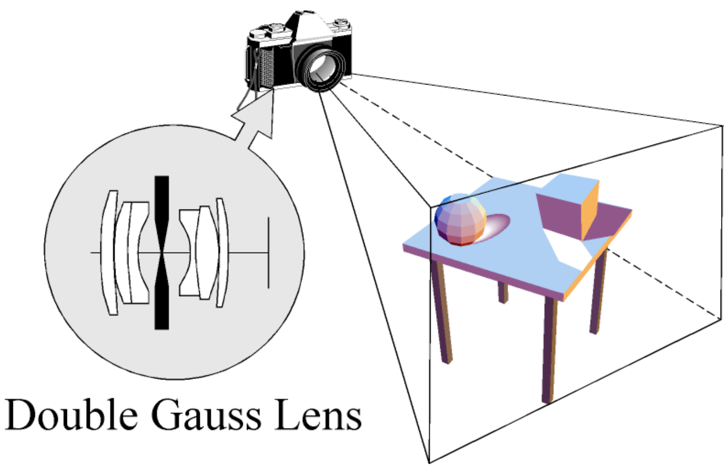


Glare & Diffraction



Adaptation

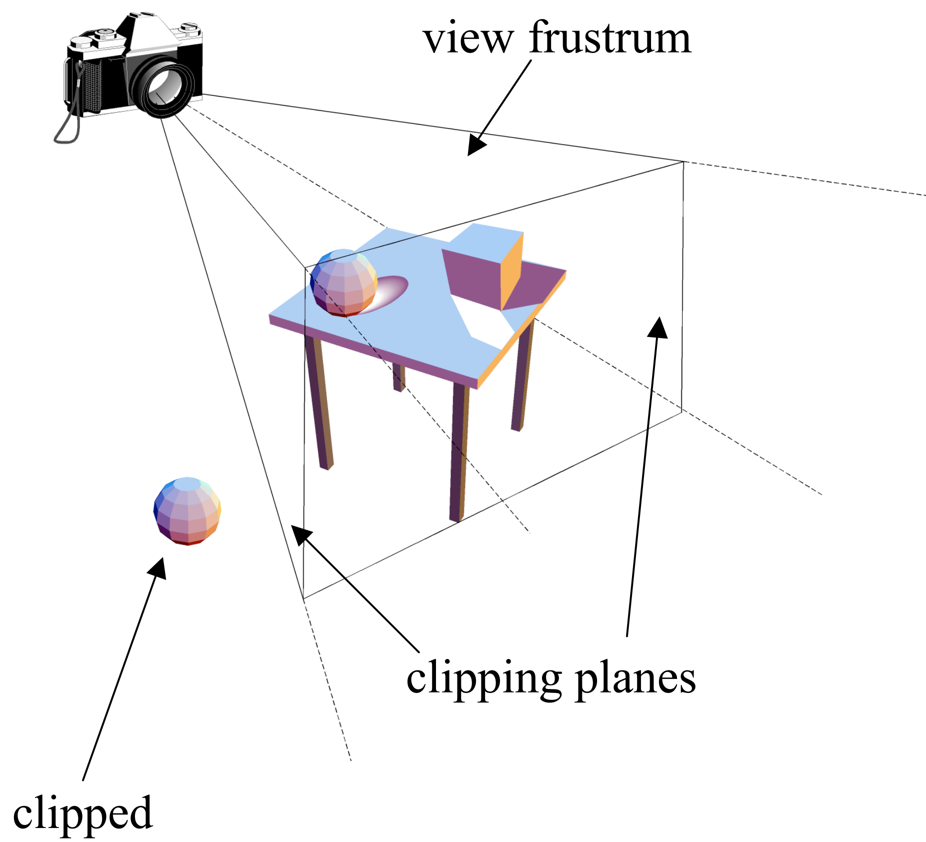
Camera Systems



A camera model implemented in Princeton University (1995)

Viewing System

- We are only concerned with the *geometry* of viewing at this stage.
- The camera's position and orientation define a *view-volume* or *view-frustrum*.
  - **objects completely or partially within this volume are potentially visible on the viewport.**
  - **objects fully outside this volume cannot be seen ⇒ *clipped***



CTM= Current Transformation Matrix

Camera Models

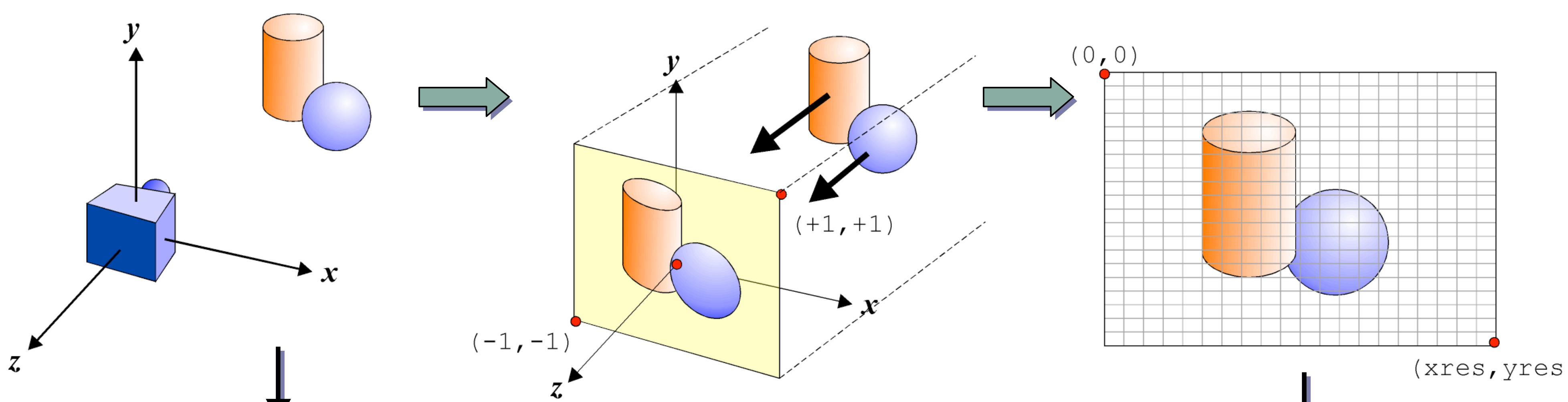
- Each vertex in our model must be projected onto the 2D *camera viewport* plane in order to be display on the screen.
- The *CTM* is employed to determine the location of each vertex in the camera coordinate system:

$$\vec{x}' = \mathbf{M}_{CTM}\vec{x}$$

- We then employ a projection matrix defined by `GL_PROJECTION` to map this to a 2D viewport coordinate.
- Finally, this 2D coordinate is mapped to device coordinates using the viewport definition (given by `glViewport()`).

Camera Modeling in OpenGL ®

camera coordinate system → viewport coordinate system → device/screen coordinate system



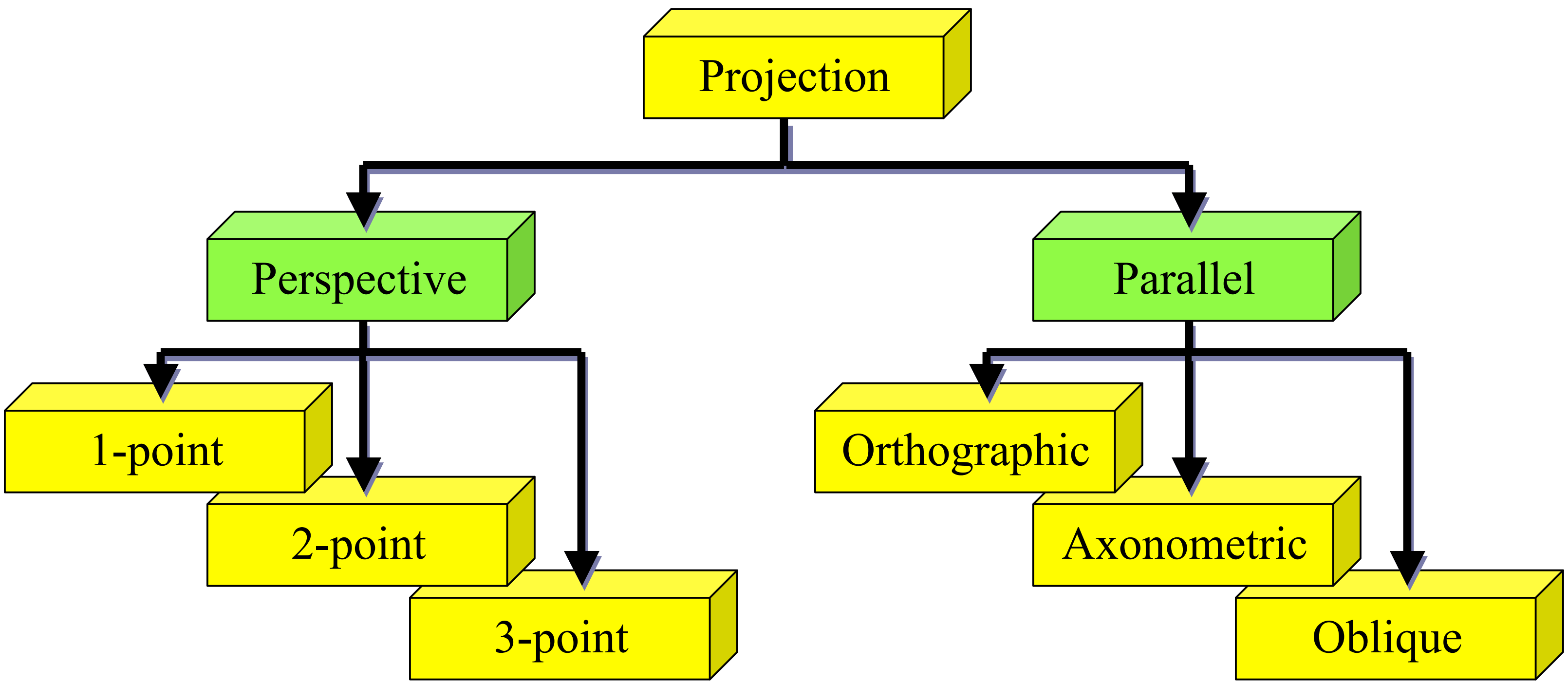

```
glMatrixMode(GL_MODELVIEW)
...
```

```
glMatrixMode(GL_PROJECTION)
...
```

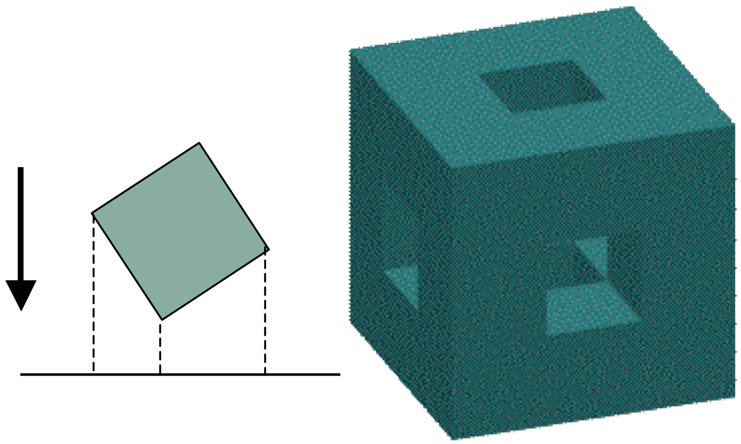
```
glViewport(0,0,xres,yres)
```

3D → 2D Projection

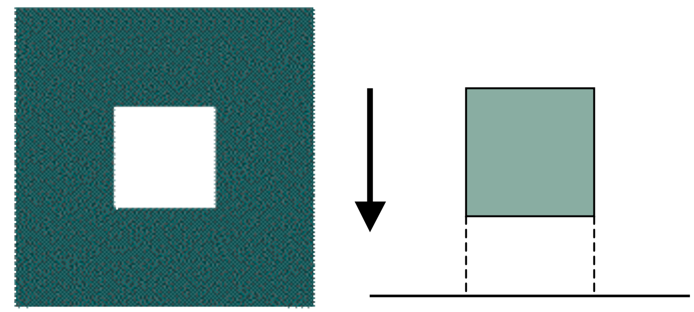
- Type of projection depends on a number of factors:
  - *location* and *orientation* of the viewing plane (*viewport*)
  - direction of projection (described by a vector)
  - projection type:



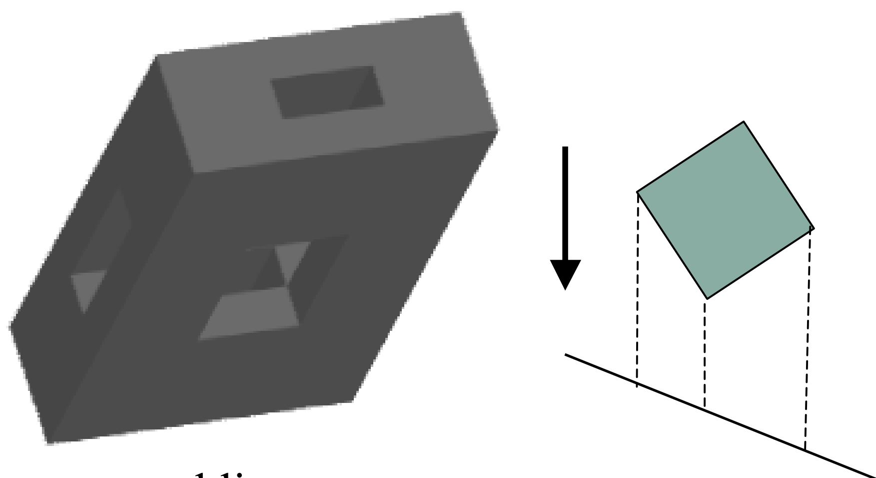
Parallel Projections



axonometric



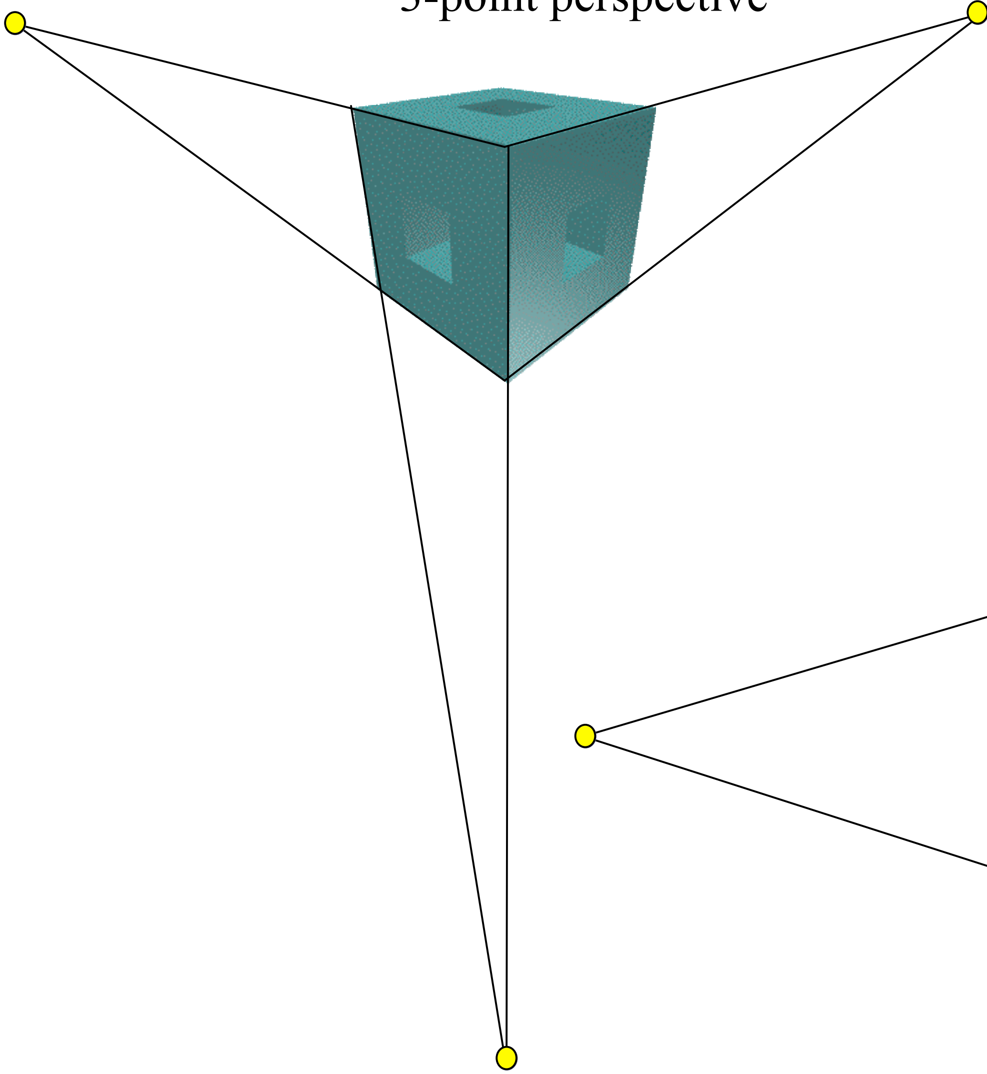
orthographic

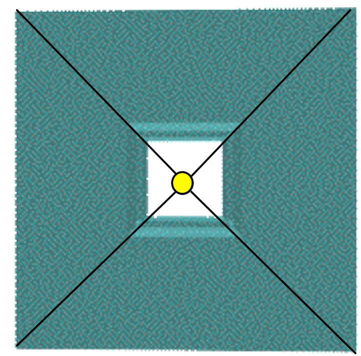


oblique

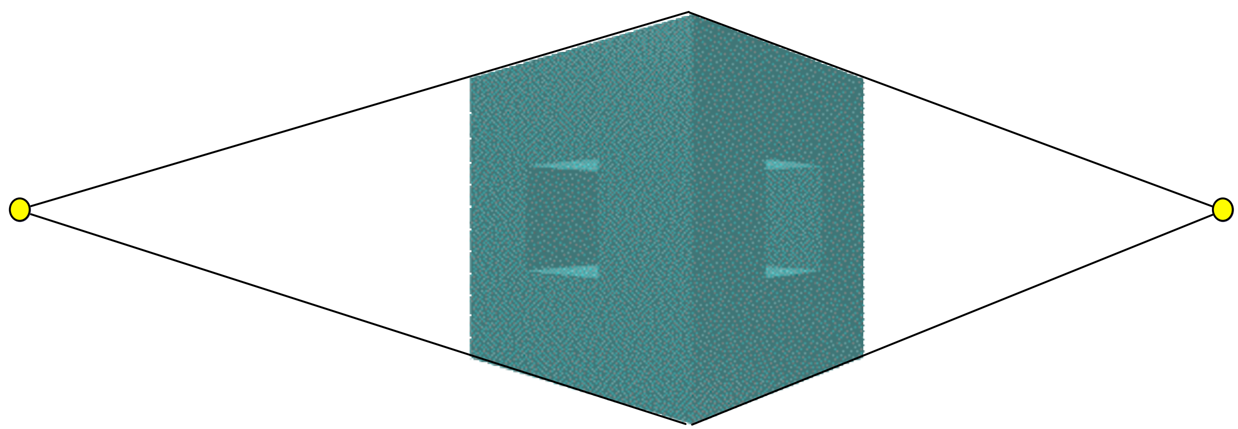
Perspective Projections

3-point perspective





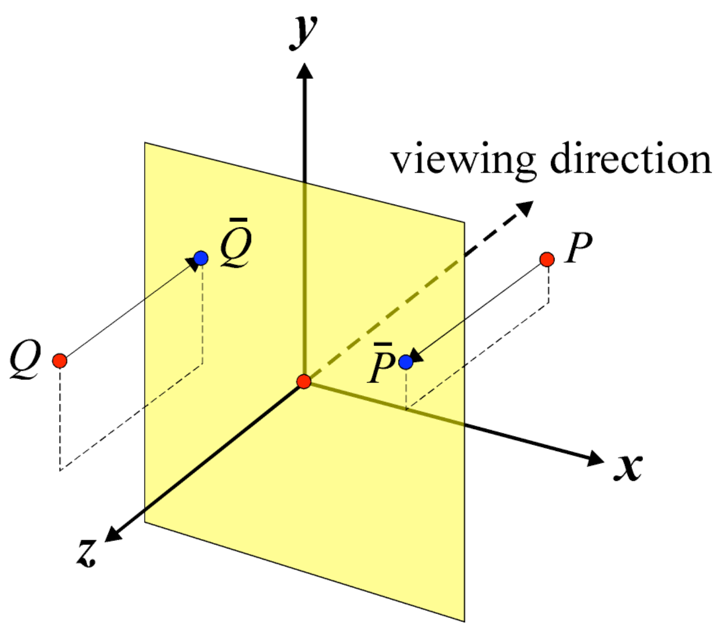
1-point perspective



2-point perspective

Orthogonal Projections

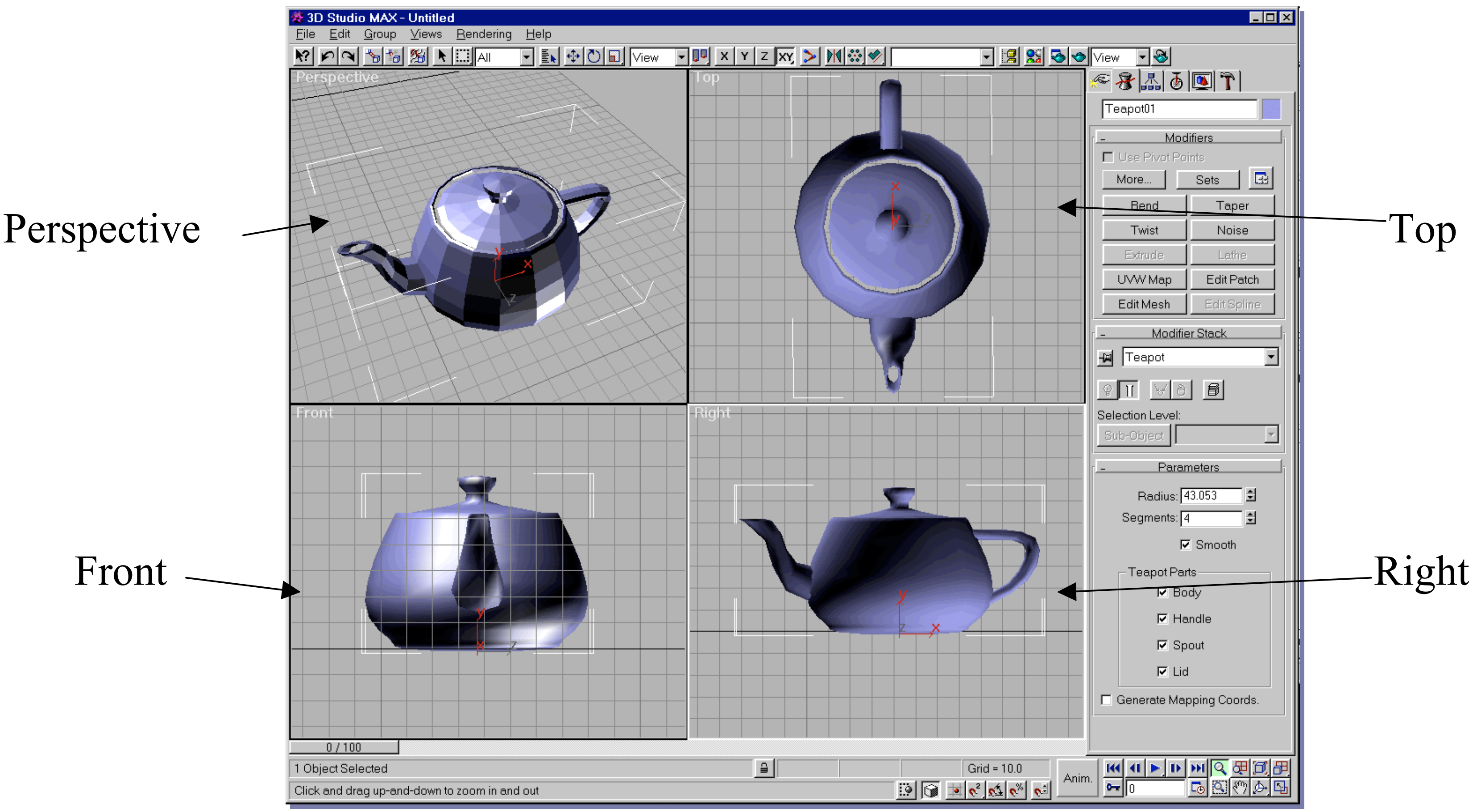
- The simplest of all projections, *parallel project* onto view-plane.
- Usually view-plane is *axis aligned* (often at *z=0*)



$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} \Rightarrow \overline{P} = \mathbf{M}P \quad \text{where} \quad \mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

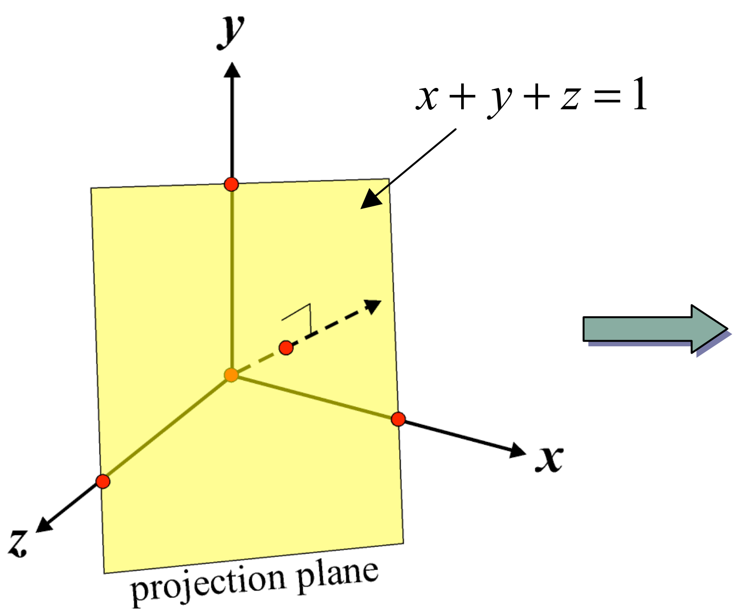
Multiple Projections

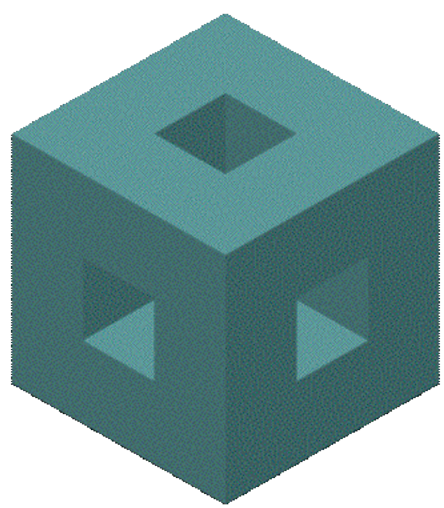
- It is often useful to have *multiple projections* available at any given time
  - usually: plan (top) view, front & left or right elevation (side) view



Orthogonal Projections

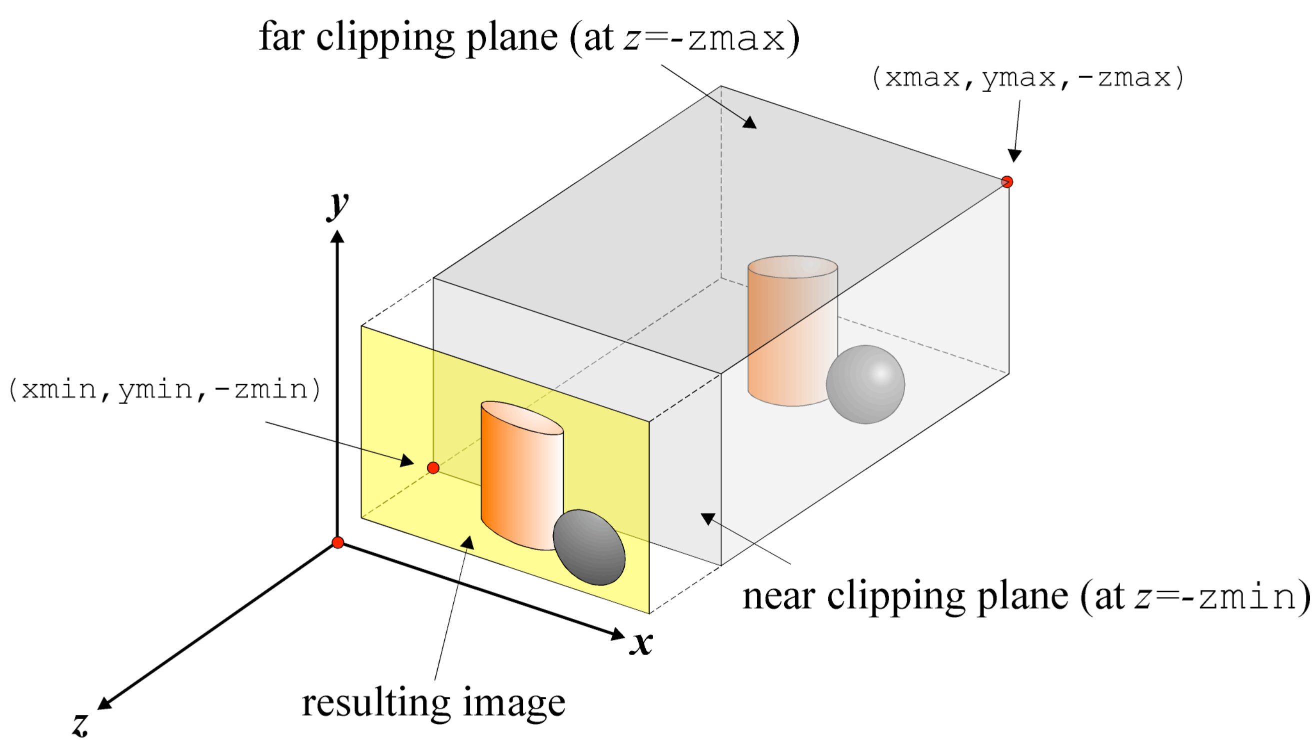
- The result is an *orthographic* projection if the object is axis aligned, otherwise it is an *axonometric* projection.
- If the projection plane intersects the principle axes at the same distance from the origin the projection is *isometric*.





Parallel Projections in OpenGL®

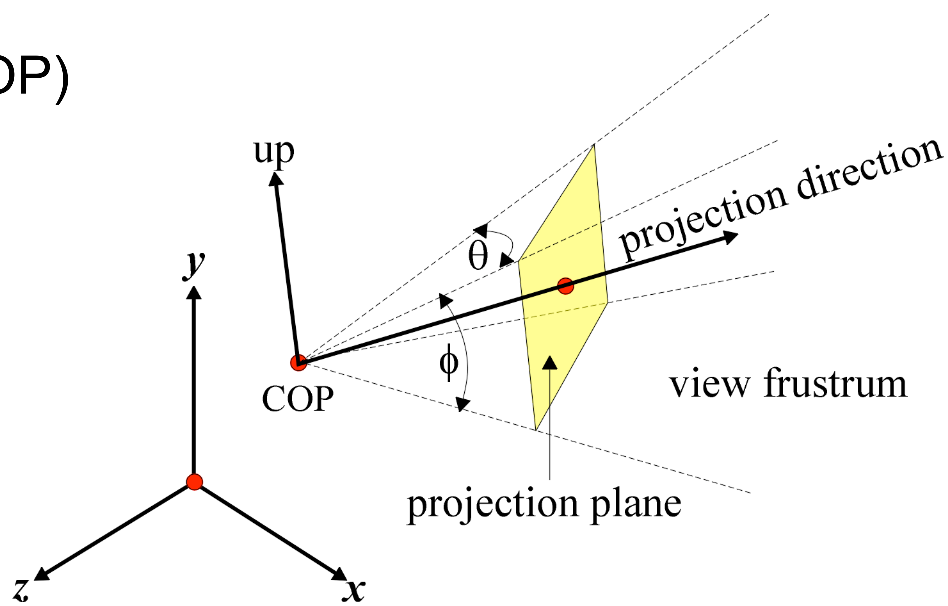
```
glOrtho(xmin, xmax, ymin, ymax, zmin, zmax);
```



Note: we always view in -z direction need to transform world in order to view in other arbitrary directions.

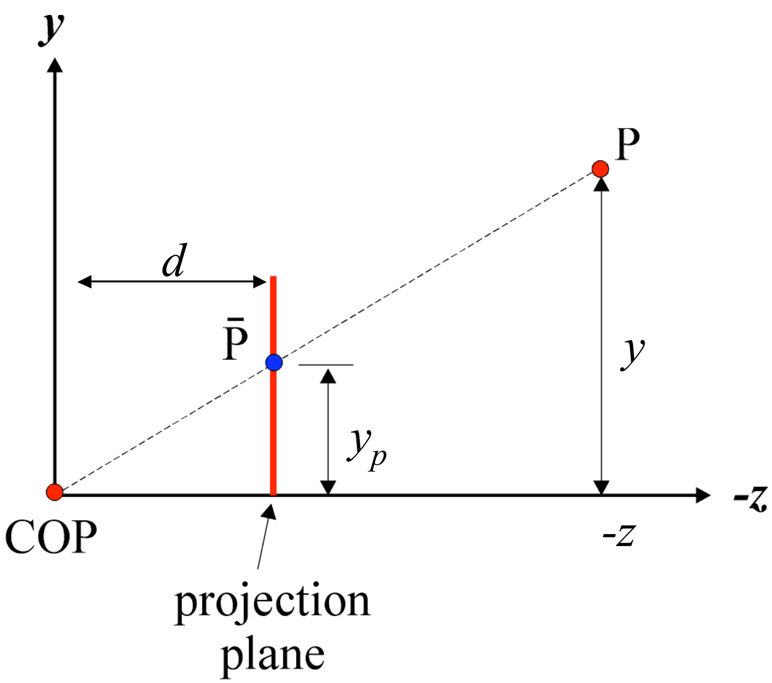
Perspective Projections

- Perspective projections are more complex and exhibit *fore-shortening* (parallel appear to converge at points).
- Parameters:
  - centre of projection (COP)
  - field of view (θ,ϕ)
  - projection direction
  - up direction



Perspective Projections

Consider a perspective projection with the viewpoint at the origin and a viewing direction oriented along the positive -***z*** axis and the view-plane located at *z* = -*d*



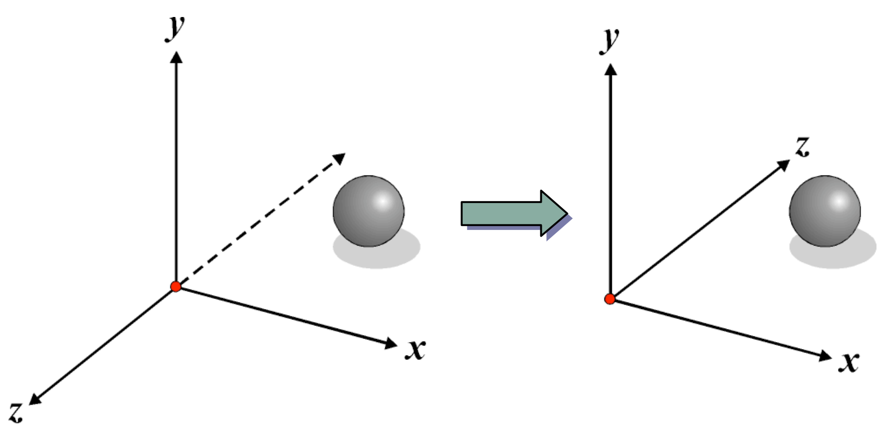
$$\frac{y}{z} = \frac{y_P}{d} \Rightarrow y_P = \frac{y}{z/d}$$

a similar construction for $x_p \Rightarrow$

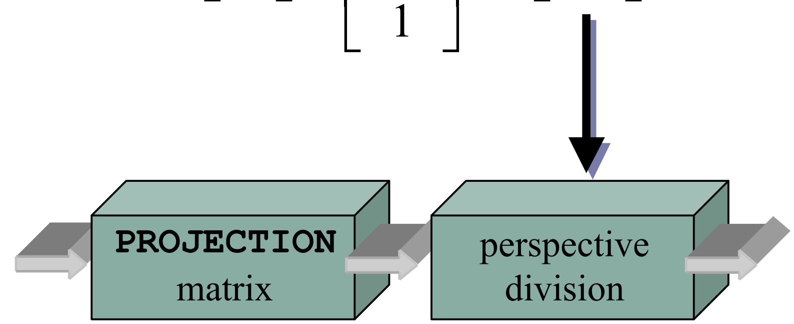
$$\begin{bmatrix} x_P \\ y_P \\ z_P \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ -d \\ 1 \end{bmatrix} \leftrightarrow \begin{bmatrix} x \\ y \\ -z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

divide by homogenous ordinate to map back to 3D space

Perspective Projections Details



$$\begin{bmatrix} x_P \\ y_P \\ z_P \\ 1 \end{bmatrix} = \begin{bmatrix} \dfrac{x}{z/d} \\ \dfrac{y}{z/d} \\ -d \\ 1 \end{bmatrix} \leftrightarrow \begin{bmatrix} x \\ y \\ -z \\ z/d \end{bmatrix}$$



$$\begin{bmatrix} x \\ y \\ -z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

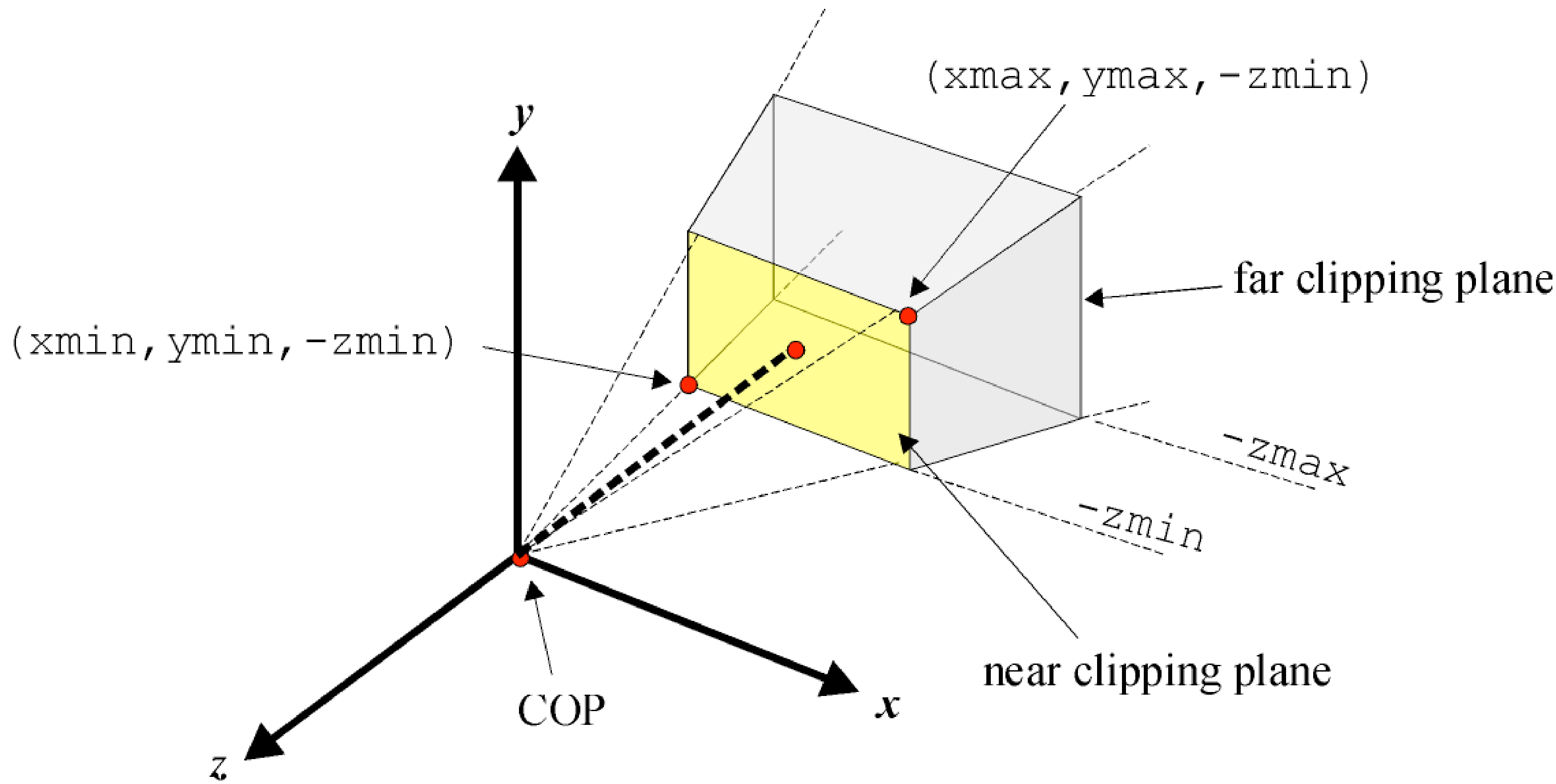
Flip ***z*** to transform to a left handed co-ordinate system ⇒ increasing *z* values mean increasing distance from the viewer.

Perspective Projection

- Depending on the application we can use different mechanisms to specify a perspective view.
- <u>Example</u>: the *field of view* angles may be derived if the distance to the viewing plane is known.
- <u>Example</u>: the viewing direction may be obtained if a point in the scene is identified that we wish to look at.
- OpenGL supports this by providing different methods of specifying the perspective view:
  - **`gluLookAt`**, **`glFrustrum`** and **`gluPerspective`**

Perspective Projections

```
glFrustrum(xmin, xmax, ymin, ymax, zmin, zmax);
```



glFrustrum

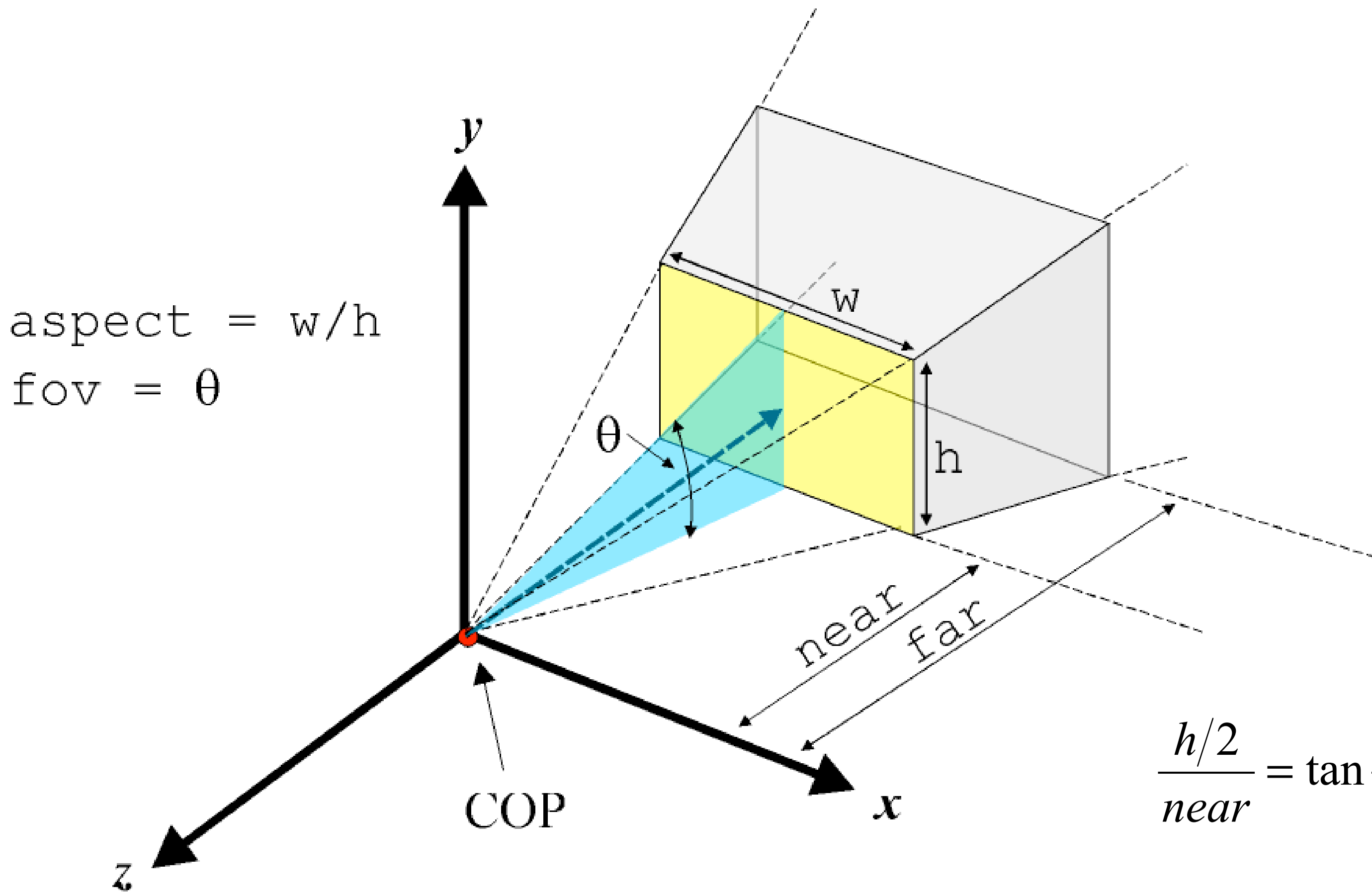
- Note that all points on the line defined by (`xmin`,`ymin`,-`zmin`) and `COP` are mapped to the *lower left* point on the viewport.
- Also all points on the line defined by (`xmax`,`ymax`,-`zmin`) and `COP` are mapped to the upper right corner of the viewport.
- The viewing direction is always parallel to ***-z***
- It is not necessary to have a *symmetric frustrum* like:

```
glFrustrum(-1.0, 1.0, -1.0, 1.0, 5.0, 50.0);
```

- Non symmetric frustrums introduce *obliqueness* into the projection.
- `zmin` and `zmax` are specified as <u>positive</u> distances along ***-z***

Perspective Projections

```
gluPerspective(fov, aspect, near, far);
```

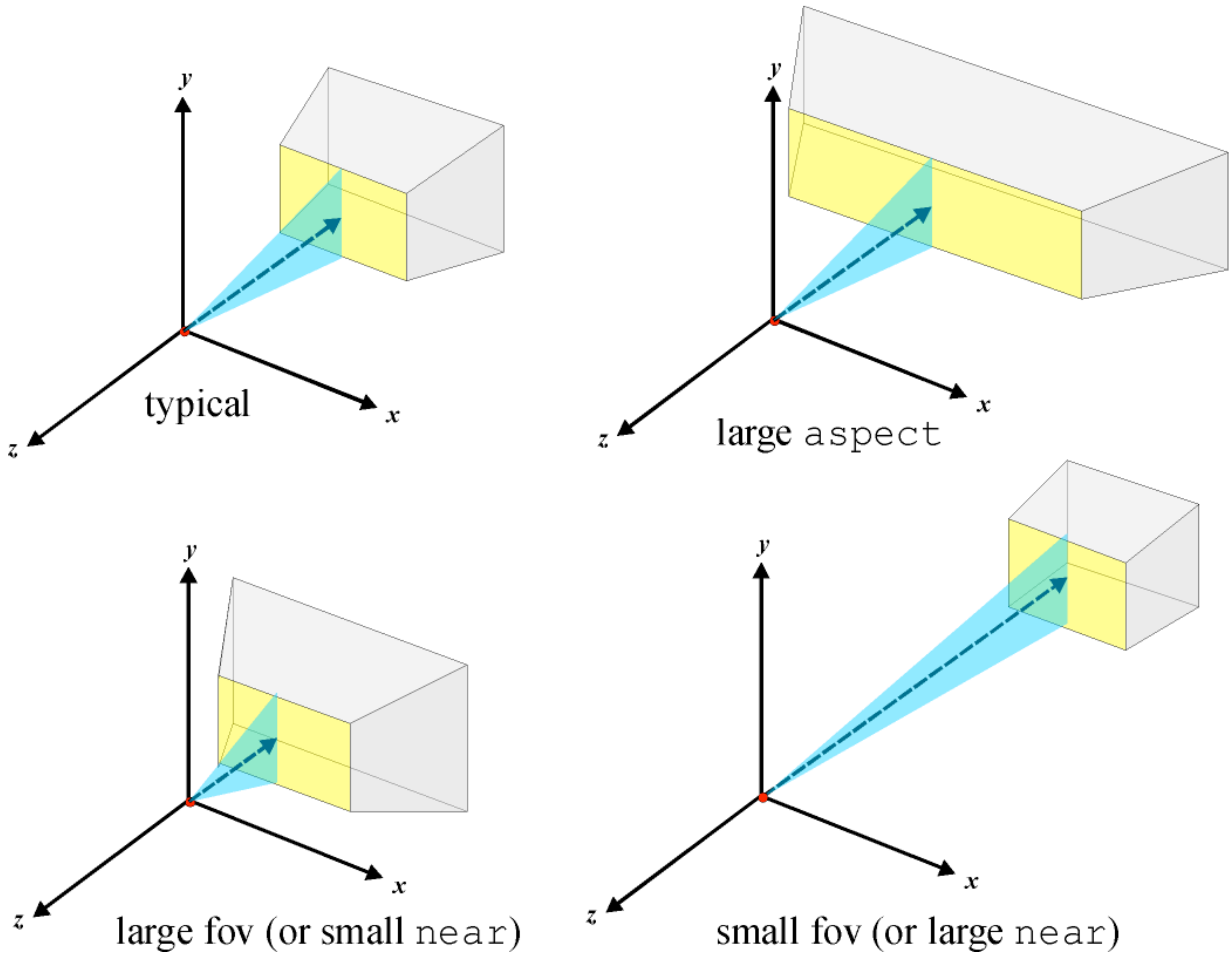


$$\frac{h/2}{near} = \tan\frac{\theta}{2} \Rightarrow h = 2\,near\tan\frac{\theta}{2}$$

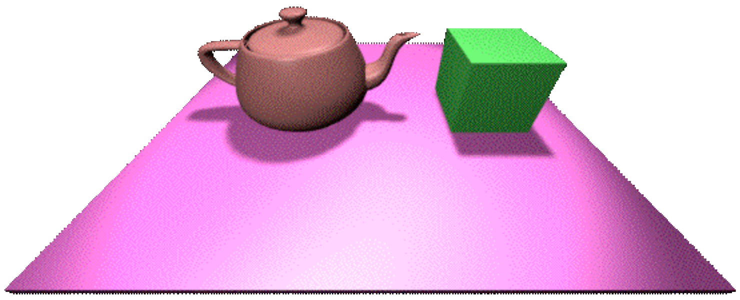
`gluPerspective`

- A utility function to simplify the specification of perspective views.
- Only allows creation of *symmetric frustrums*.
- Viewpoint is at the origin and the viewing direction is the ***-z*** axis.
- The *field of view* angle, `fov`, must be in the range [0..180]
- `aspect` allows the creation of a view frustrum that matches the *aspect ratio* of the viewport to eliminate distortion.

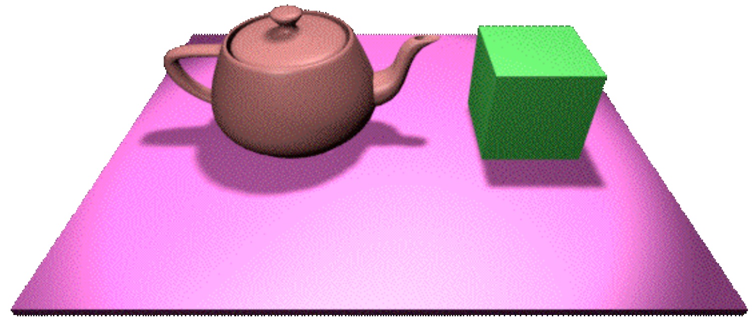
Perspective Projections



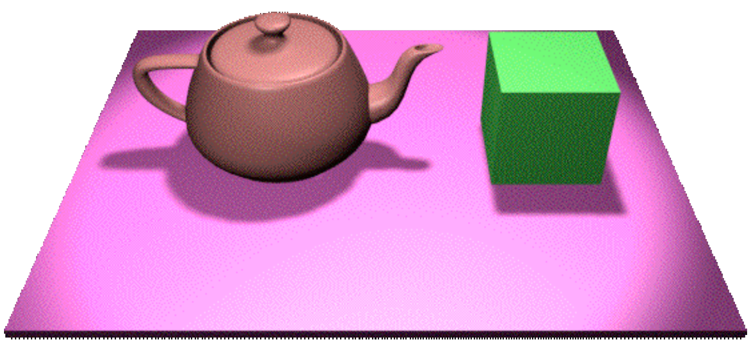
Lens Configurations



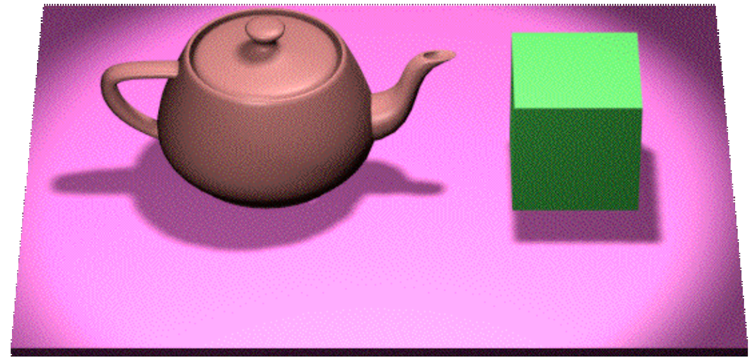
10mm Lens (fov = 122°)



20mm Lens (fov = 84°)



35mm Lens (fov = 54°)



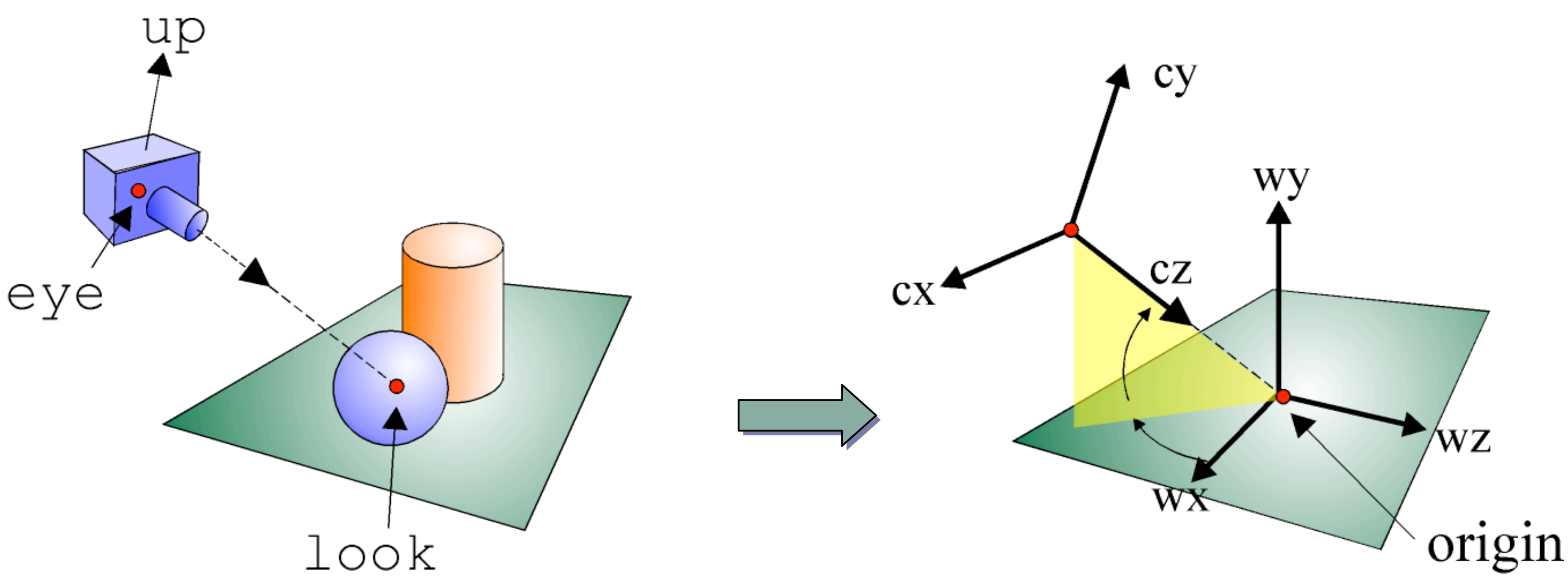
200mm Lens (fov = 10°)

Positioning the Camera

- The previous projections had limitations:
  - usually fixed origin and fixed projection direction
- To obtain arbitrary camera orientations and positions we manipulate the `MODELVIEW` matrix prior to creation of the models. This positions the camera w.r.t. the model.
- We wish to position the camera at (10, 2, 10) w.r.t. the world
- Two possibilities:
  - transform the world prior to creation of objects using `translatef` and `rotatef`: **`glTranslatef(-10, -2, -10);`**
  - use `gluLookAt` to position the camera with respect to the world co-ordinate system: **`gluLookAt(10, 2, 10, … );`**
- Both are *equivalent*.

Positioning the Camera

```
gluLookAt(eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz);
```



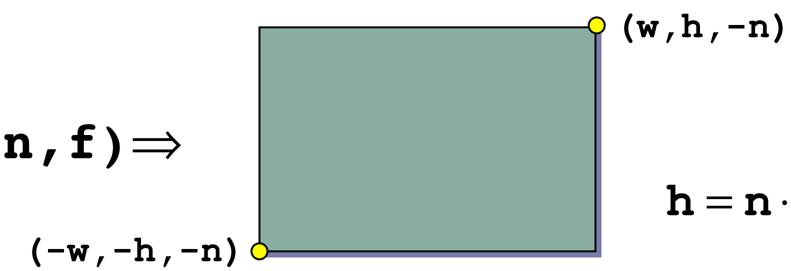
equivalent to:

```
glTranslatef(-eyex, -eyey, -eyez);
glRotatef(theta, 1.0, 0.0, 0.0);
glRotatef(phi, 0.0, 1.0, 0.0);
```

Projection window

- The projection matrix defines the mapping from a 3D world co-ordinate to a 2D viewport co-ordinate.
- The window extents are defined as a parameter of the projection:
  - **glFrustrum(l,r,b,t,n,f)** ⇒

  - **gluPerspective(f,a,n,f)** ⇒

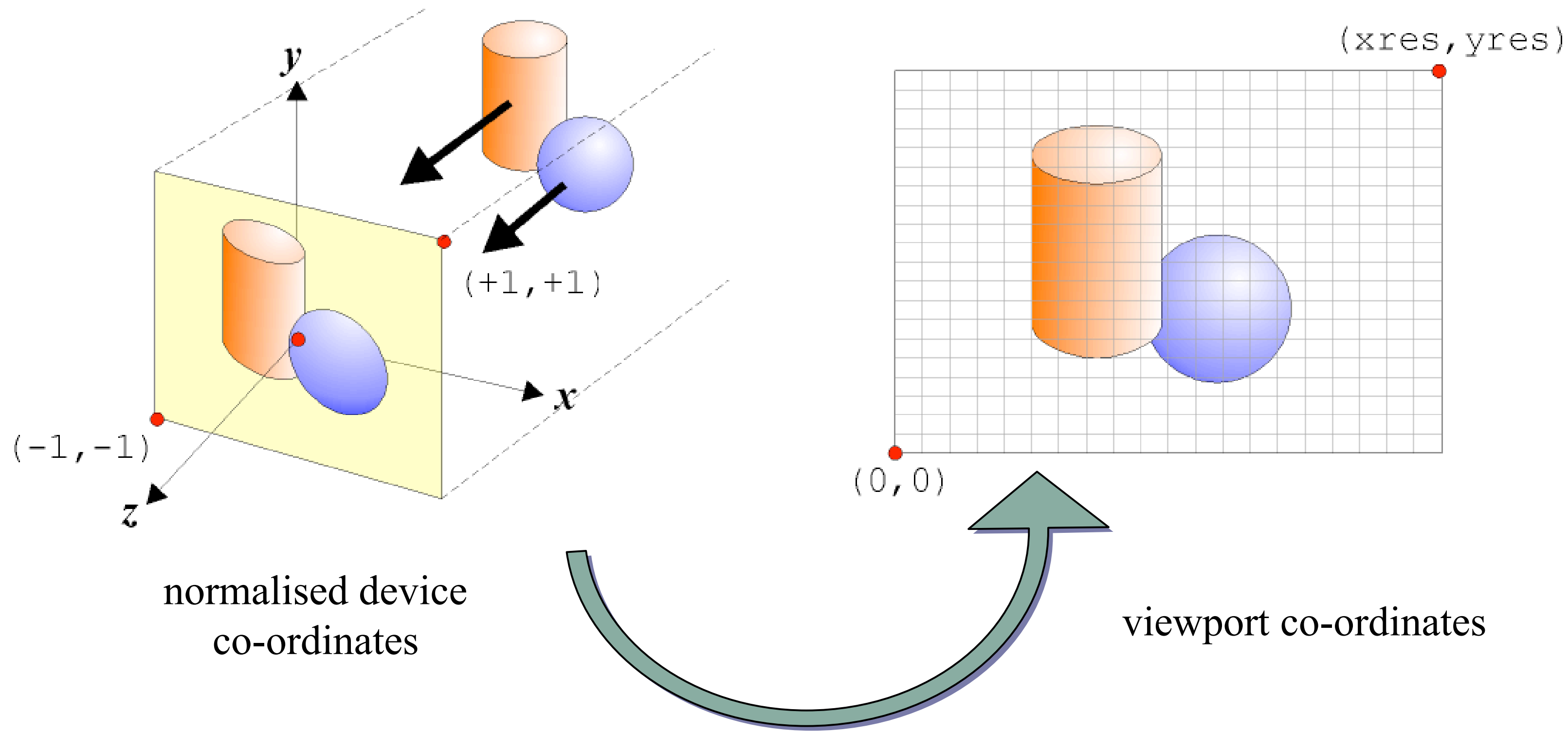


$$\mathbf{h} = \mathbf{n} \cdot \tan\frac{\mathbf{f}}{2}$$

$$\mathbf{w} = \mathbf{h} \cdot \mathbf{a}$$

Projection window

- We need to associate the 2D *window co-ordinate system* with the *viewport co-ordinate system* in order to determine the correct pixel associated with each vertex.



Window to Viewport Transformation: review

- An *affine* planar transformation is used.
- After projection to the window, all points are transformed to normalised device co-ordinates: `[-1,1]x[1,1]`

$$x_n = 2\left(\frac{x_p - x_{\min}}{x_{\max} - x_{\min}}\right) - 1$$

$$y_n = 2\left(\frac{y_p - y_{\min}}{y_{\max} - y_{\min}}\right) - 1$$

- `glViewport` used to relate the co-ordinate systems:

```
glViewport(int x, int y, int width, int height);
```

Window to Viewport Transformation: review

- (x,y) = location of bottom left of viewport within the window
- width,height = dimension in pixels of the viewport ⇒

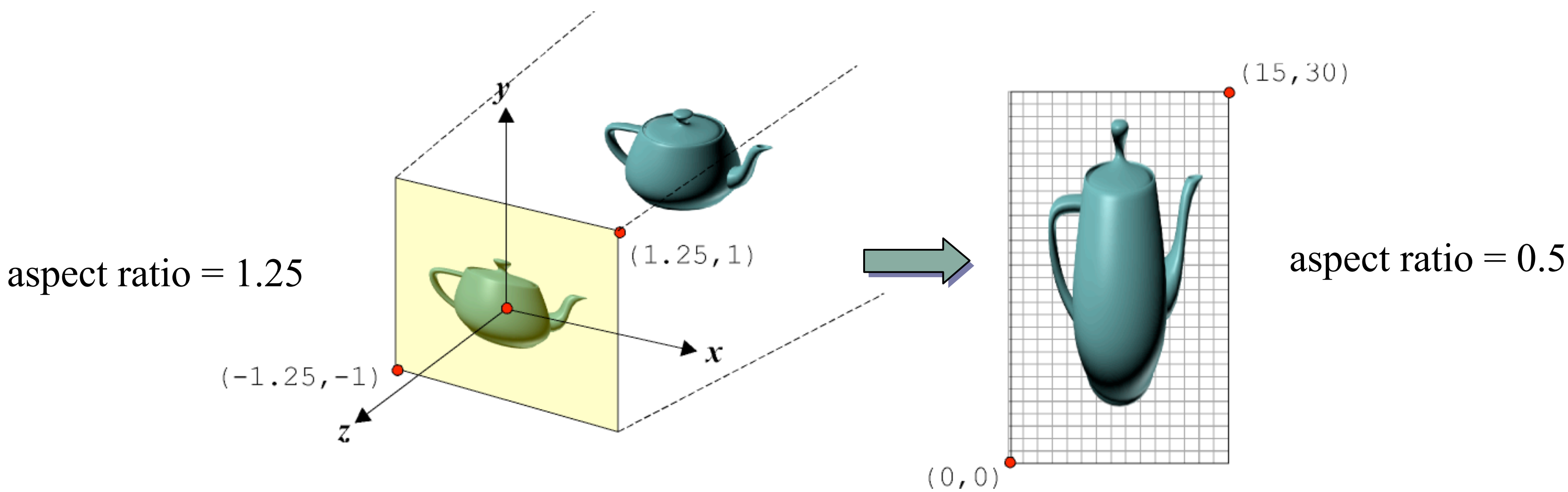
$$x_w = (x_n + 1)\left(\frac{\mathbf{width}}{2}\right) + \mathbf{x} \quad y_w = (y_n + 1)\left(\frac{\mathbf{height}}{2}\right) + \mathbf{y}$$

- normally we re-create the window after a window resize event to ensure a correct mapping between window and viewport dimensions:

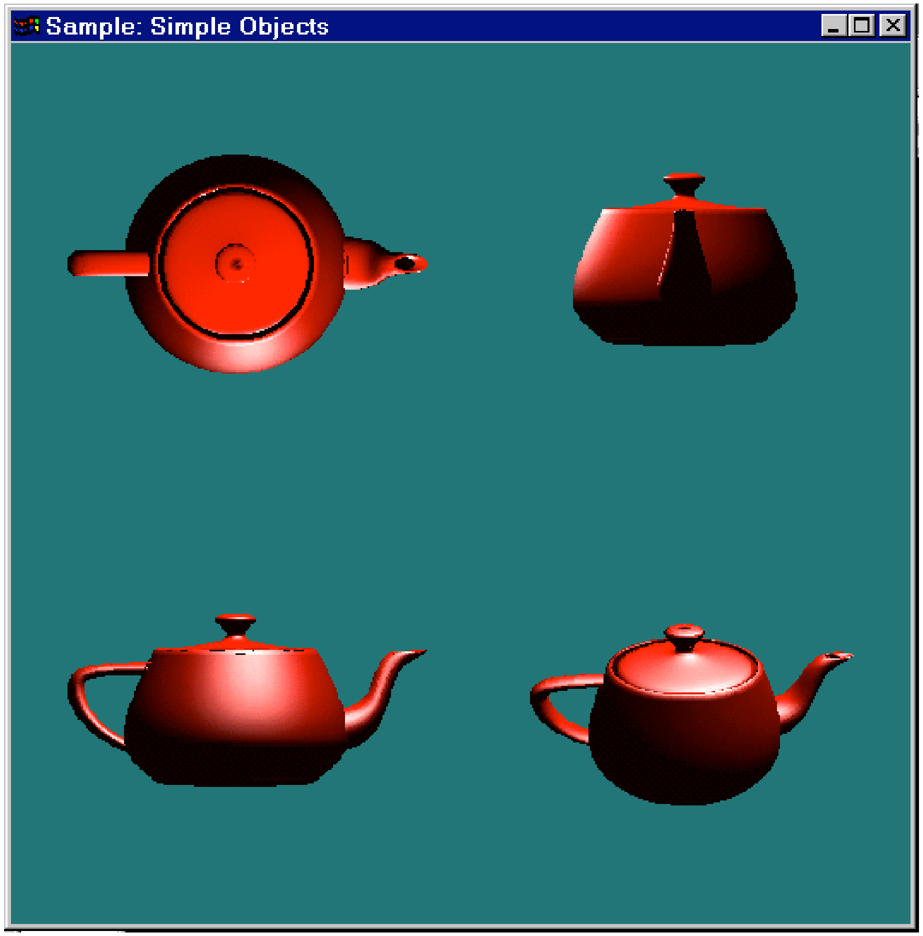
```
static void reshape(int width, int height)
{
        glViewport(0, 0, width, height);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(85.0, 1.0, 5, 50);
}
```

Aspect Ratio

- The *aspect ratio* defines the relationship between the width and height of an image.
- Using `gluPerspective` an viewport aspect ratio may be explicitly provided, otherwise the aspect ratio is a function of the supplied viewport width and height.
- The aspect ratio of the window (defined by the user) must match the viewport aspect ratio to prevent unwanted *affine* distortion:



Sample Viewport Application




```
// top left: top view
glViewport(0, win_height/2, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-3.0, 3.0, -3.0, 3.0, 1.0, 50.0);
gluLookAt(0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glCallList(object);

// top right: right view
glViewport(win_width/2, win_height/2, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-3.0, 3.0, -3.0, 3.0, 1.0, 50.0);
gluLookAt(5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glCallList(object);

// bottom left: front view
glViewport(0, 0, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-3.0, 3.0, -3.0, 3.0, 1.0, 50.0);
gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glCallList(object);

// bottom right: rotating perspective view
glViewport(win_width/2, 0, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(70.0, 1.0, 1, 50);
gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(30.0, 1.0, 0.0, 0.0);
glRotatef(Angle, 0.0, 1.0, 0.0);
glCallList(object);
```