Computação Gráfica

Computer Graphics

Engenharia Informática (11569) – 3º ano, 2º semestre

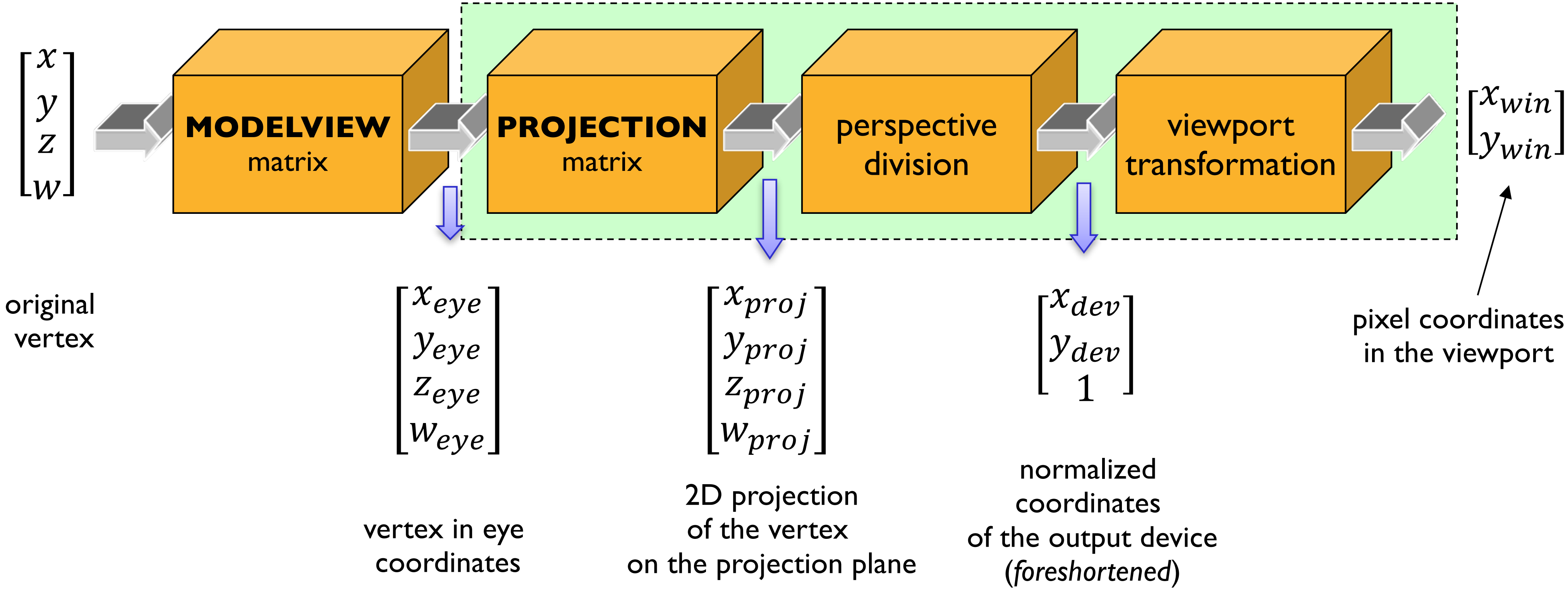
Chap. 5 – 3D Projections and Scenes

Outline

…:

- OpenGL rendering pipeline.
- Camera+plane+scene model.
- Camera types: classical camera, double-lens camera of Gauss, photorealsitic rendering camera.
- Rendering 3D scenes in OpenGL.
- Projection types: parallel projection and perspective projection.
- Projections in OpenGL.
- Moving camera.
- Projection window. Window-viewport transformation: revisited. Aspect ratio revisited.
- OpenGL examples.

OpenGL® graphics pipeline



$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$ → **MODELVIEW** matrix → **PROJECTION** matrix → perspective division → viewport transformation → $\begin{bmatrix} x_{win} \\ y_{win} \end{bmatrix}$

original vertex

$\begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix}$

vertex in eye coordinates

$\begin{bmatrix} x_{proj} \\ y_{proj} \\ z_{proj} \\ w_{proj} \end{bmatrix}$

2D projection of the vertex on the projection plane

$\begin{bmatrix} x_{dev} \\ y_{dev} \\ 1 \end{bmatrix}$

normalized coordinates of the output device (*foreshortened*)

pixel coordinates in the viewport

How to render 3D scenes through graphics pipeline?

Gnerating a view of a given scene requires:

- A scene (i.e., geometric description of a scene)
- A camera or viewer (i.e., observer)
- A projection plane

Location/direction of the default OpenGL camera:

- It is at the origin and looking in the direction of the negative z-axis
- The camera allows us to project the 3D scene (geometry) onto a plane, as needed for graphics output.

Such projection can be accomplished as follows:

- orthogonal projection (parallelism of lines is preserved)
- perspective projection : 1-point, 2-points ou 3-points
- oblique orthogonal projection

Camera types

Before generating an image, we must choose the kind of camera (or viewer).

Classical camera (or *pinhole camera*)

- The most used camera model, also in OpenGL
- Infinite depth of field: everything is focused

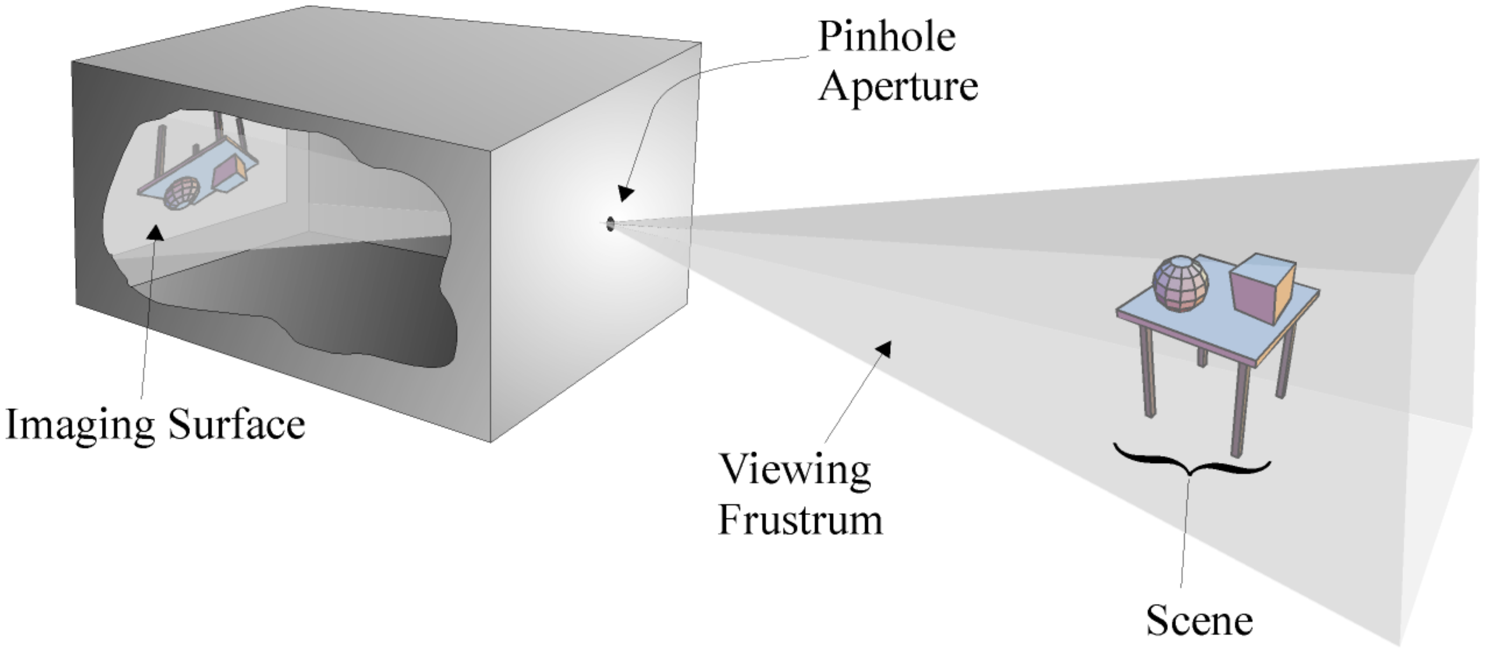
Double Gauss lens

- This camera model was implemented in Princeton University (1995)
- It is used in many professional cameras
- It models the depth of field and non-linear optics (including lens flare)

Photo-realistic rendering camera

- It often employs the physical model of the human eye to render images
- It models the eye response to brightness and color levels
- It models the internal optics of the human eye (difraction by lens fibers, etc.)

Classical camera



Double Gauss lens camera

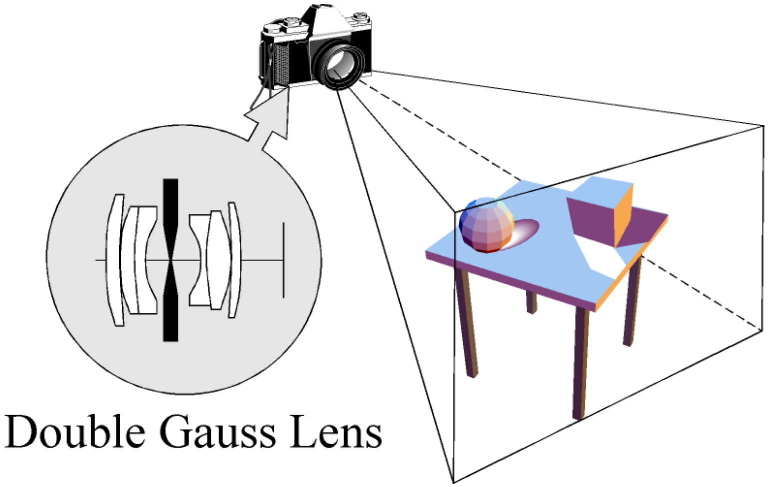
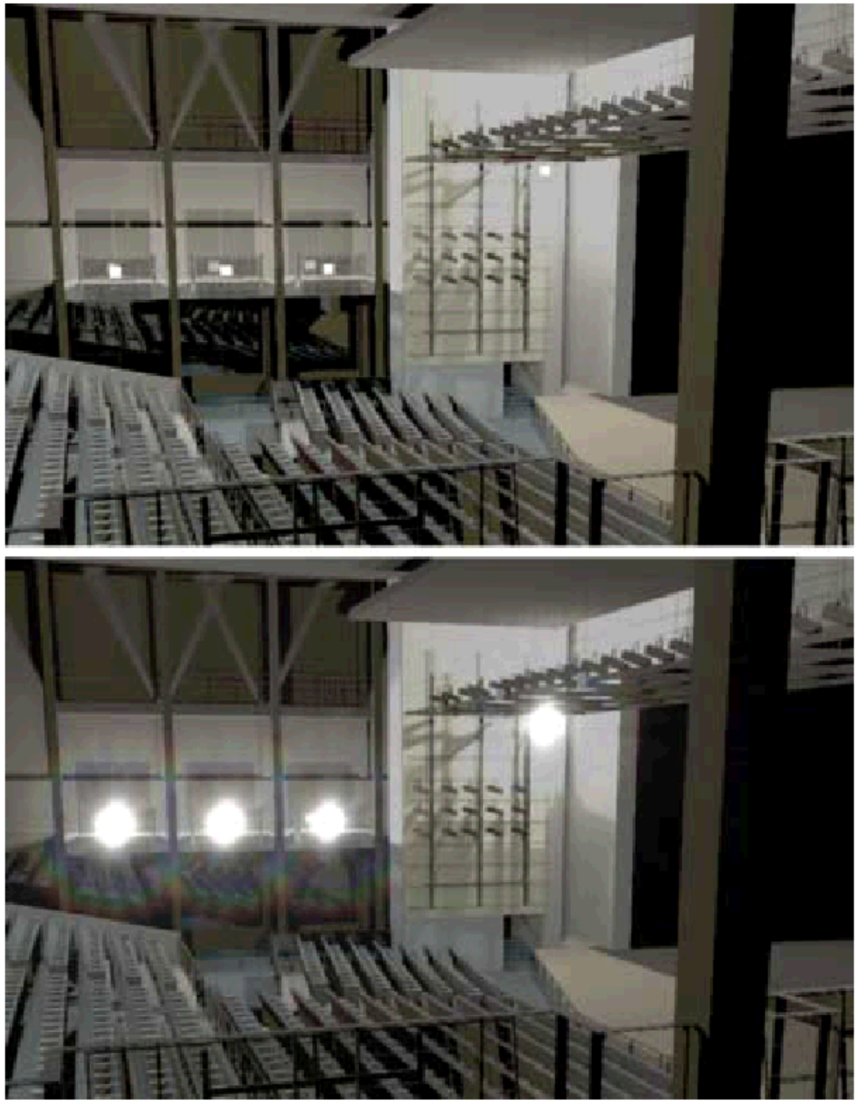
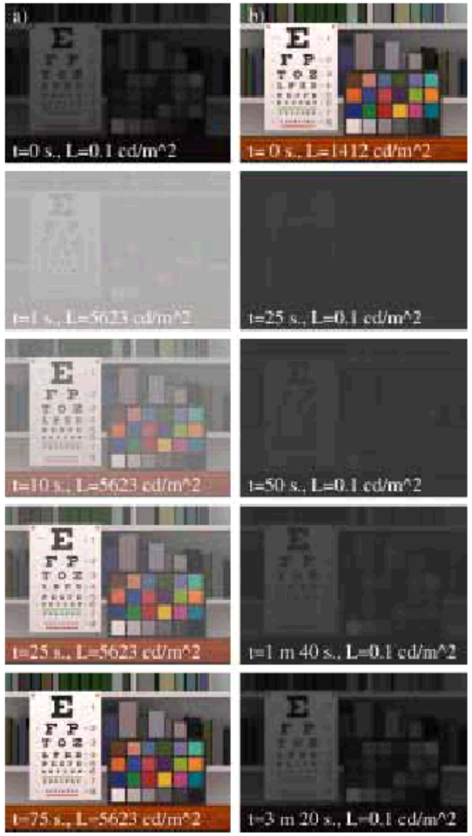


Photo-realistic camera



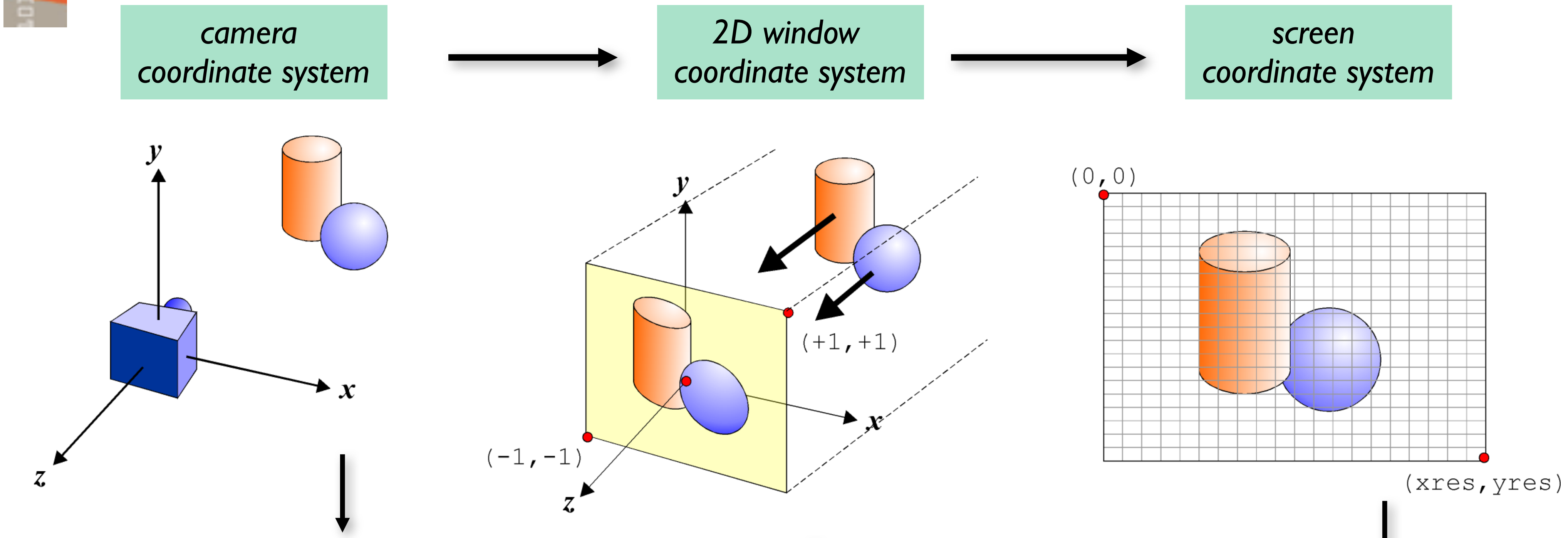
Glare & Difraction



Adaptation

Rendering 3D scenes in OpenGL®

Have a look at the graphics pipeline on page 3 for comparison sake



camera coordinate system → *2D window coordinate system* → *screen coordinate system*

glViewport(0,0,xres,yres)

MODELVIEW matrix = View * Model

```
// Model matrix: an identity matrix
// (model at origin)
glm::mat4 Model = glm::mat4(1.0f);

// Camera matrix for perspective projection;
glm::mat4 View = glm::lookAt(...)

// Camera at the infinite when we have an
// orthognal projection; so there is no need
// to define the view marix
```

PROJECTION matrix

```
// Projection matrix: 45° Field of View, 4:3 ratio,
// display range : 0.1 unit <-> 100 units
glm::mat4 Projection =
glm::perspective(glm::radians(45.0f),4.0f/3.0f,0.1f,100.0f);
```

Rendering 3D scenes in OpenGL®: from geometry to image

http://www.songho.ca/opengl/gl_transform.html

MODELVIEW matrix

- It is the product of the modelling matrix (scene coordinate system) and view matrix (eye or camera coordinate system).
- It serves to change from the scene coordinate system to the camera coordinate system.

PROJECTION matrix

- Then, we apply the projection matrix to map camera coordinates to projection plane (window) coordinates.

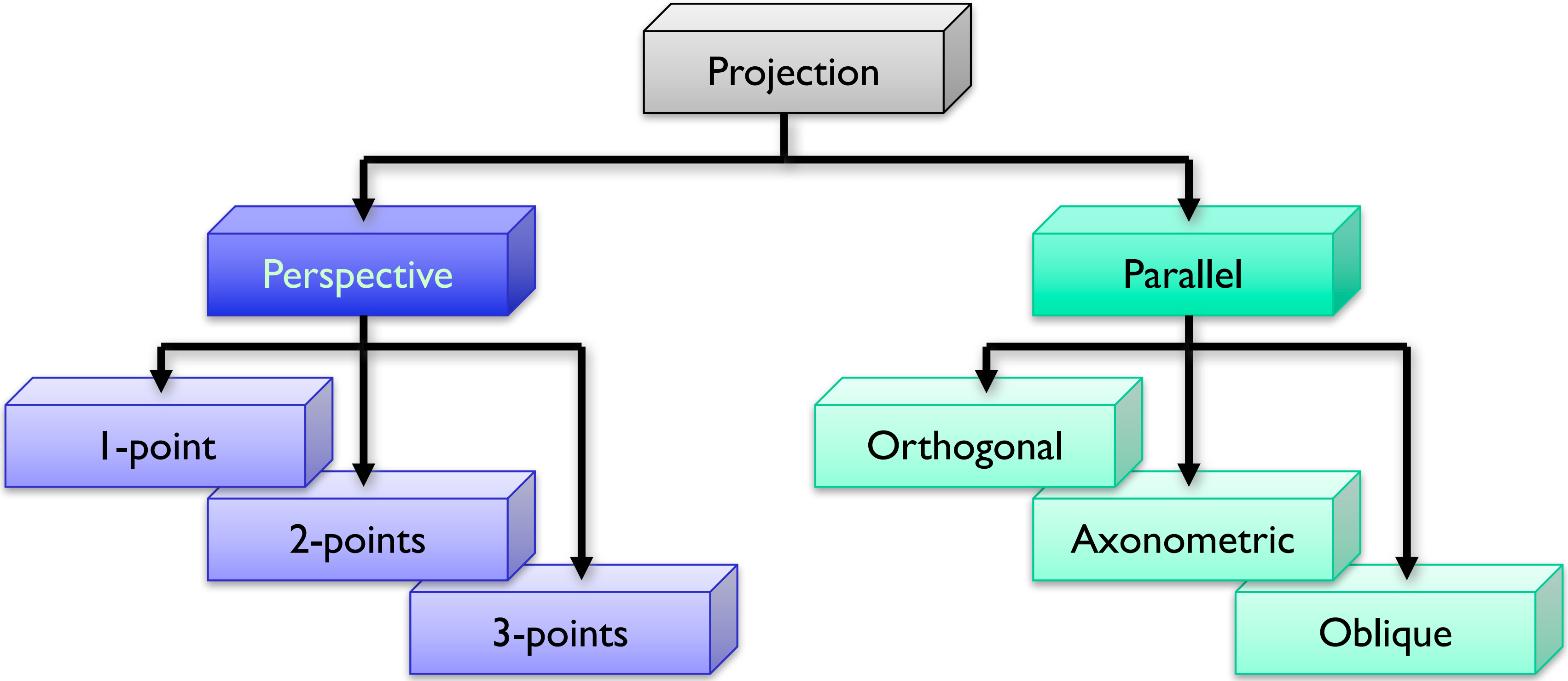
glViewport

- Finally, window coordinates are mapped to screen coordinates of the viewport, what is done in na automated manner through the window-viewport transformation.

3D→2D projection types

A kind of projection depends on 2 factors:

- Viewer's location (which determines the direction of projection or visual)
- Location and orientation of the projection plane (where the viewing window lies in)



Projection

Perspective

Parallel

1-point

2-points

3-points

Orthogonal

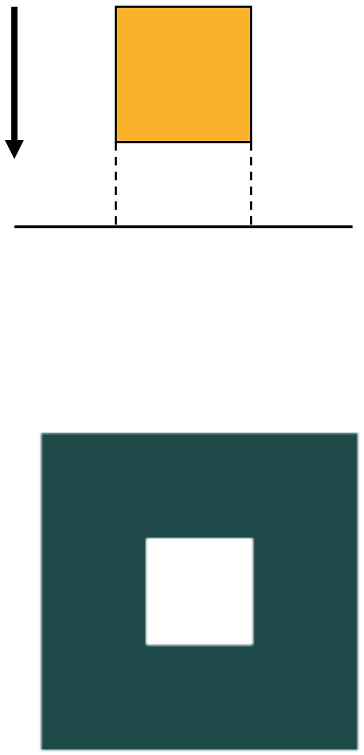
Axonometric

Oblique

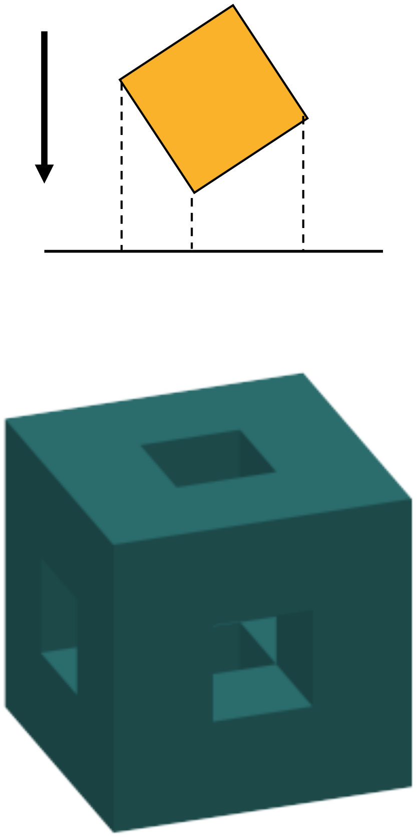
Parallel projections

- *The viewer is at the infinite.*
- *Projection or visual rays are parallel.*

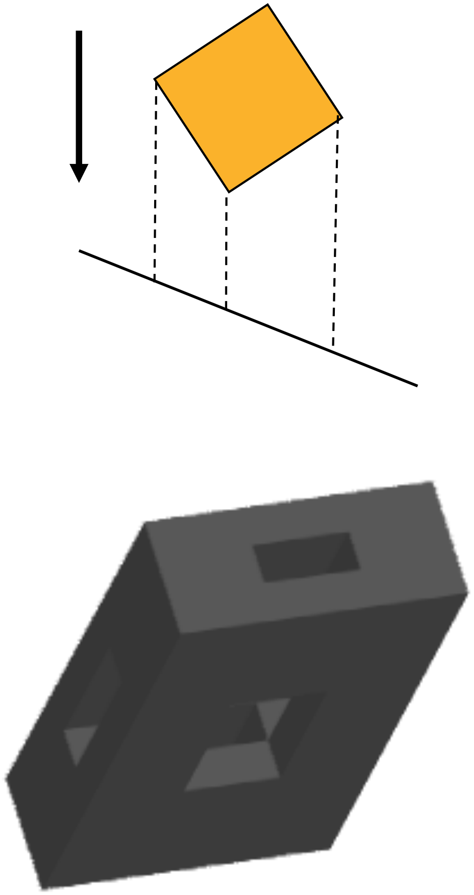
orthogonal



axonometric

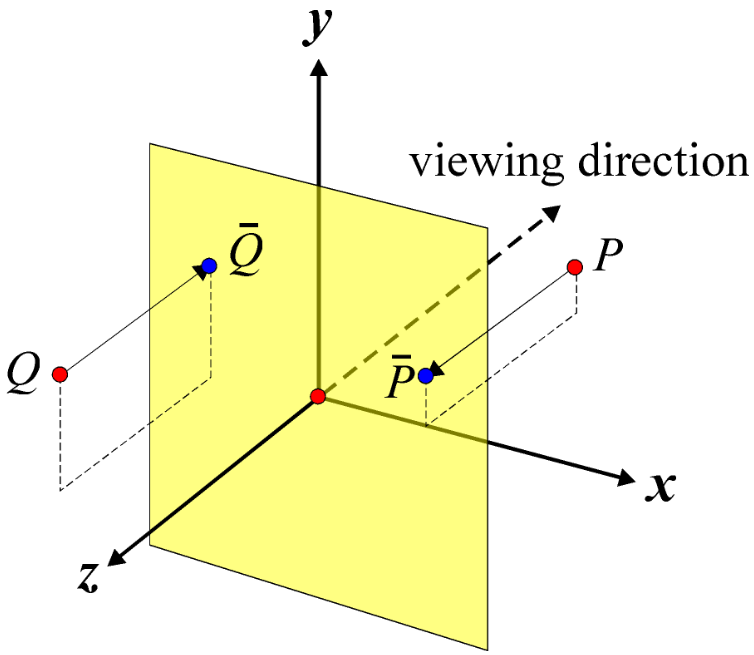


oblique



Orthogonal parallel projection matrix

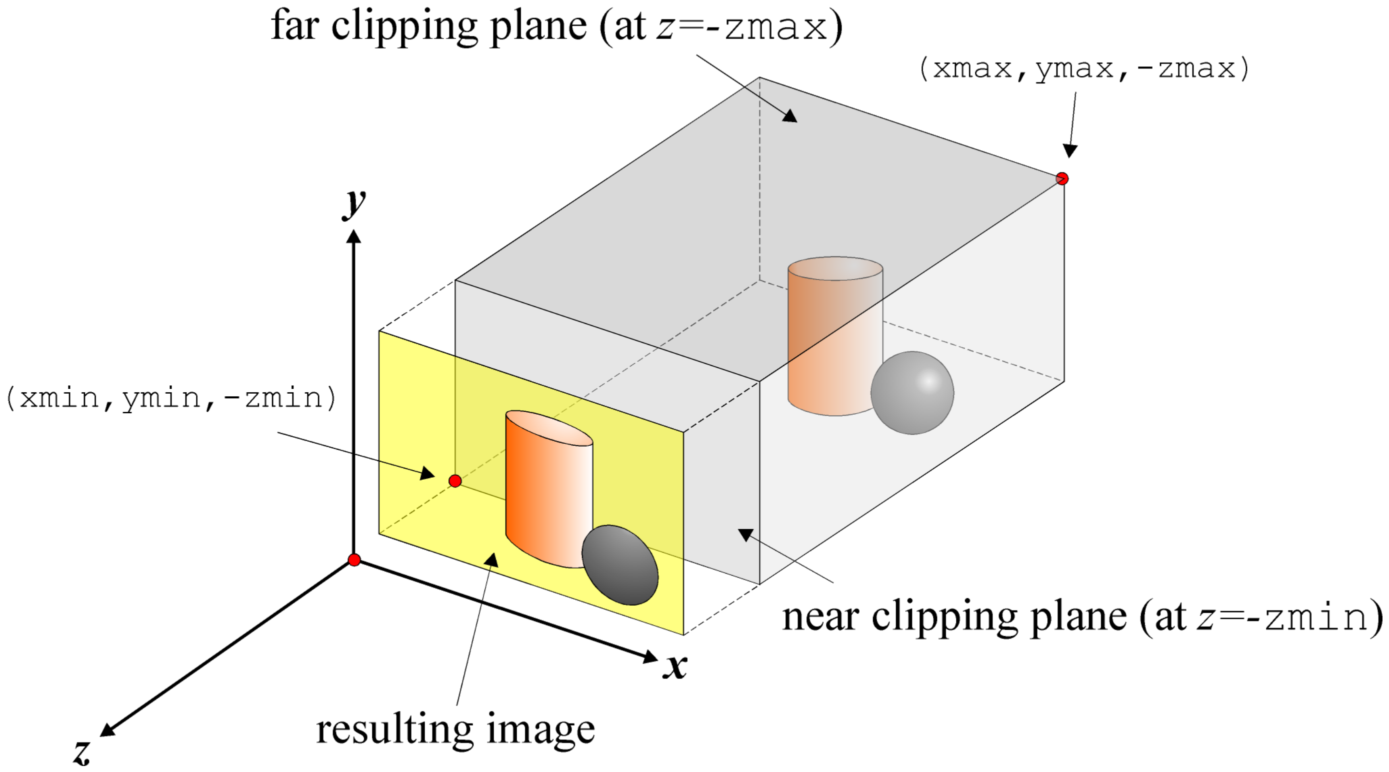
- *It is the simpler projection: the visual rays are perpendicular to the projection plane.*
- *Usually, the projection plane is aligned with coordinate axes (z=0).*
- *Orthogonal parallel projections are also known as views (in technical drawing or drafting).*



$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} \Longrightarrow \bar{P} = \mathbf{M}P, \text{where } \mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

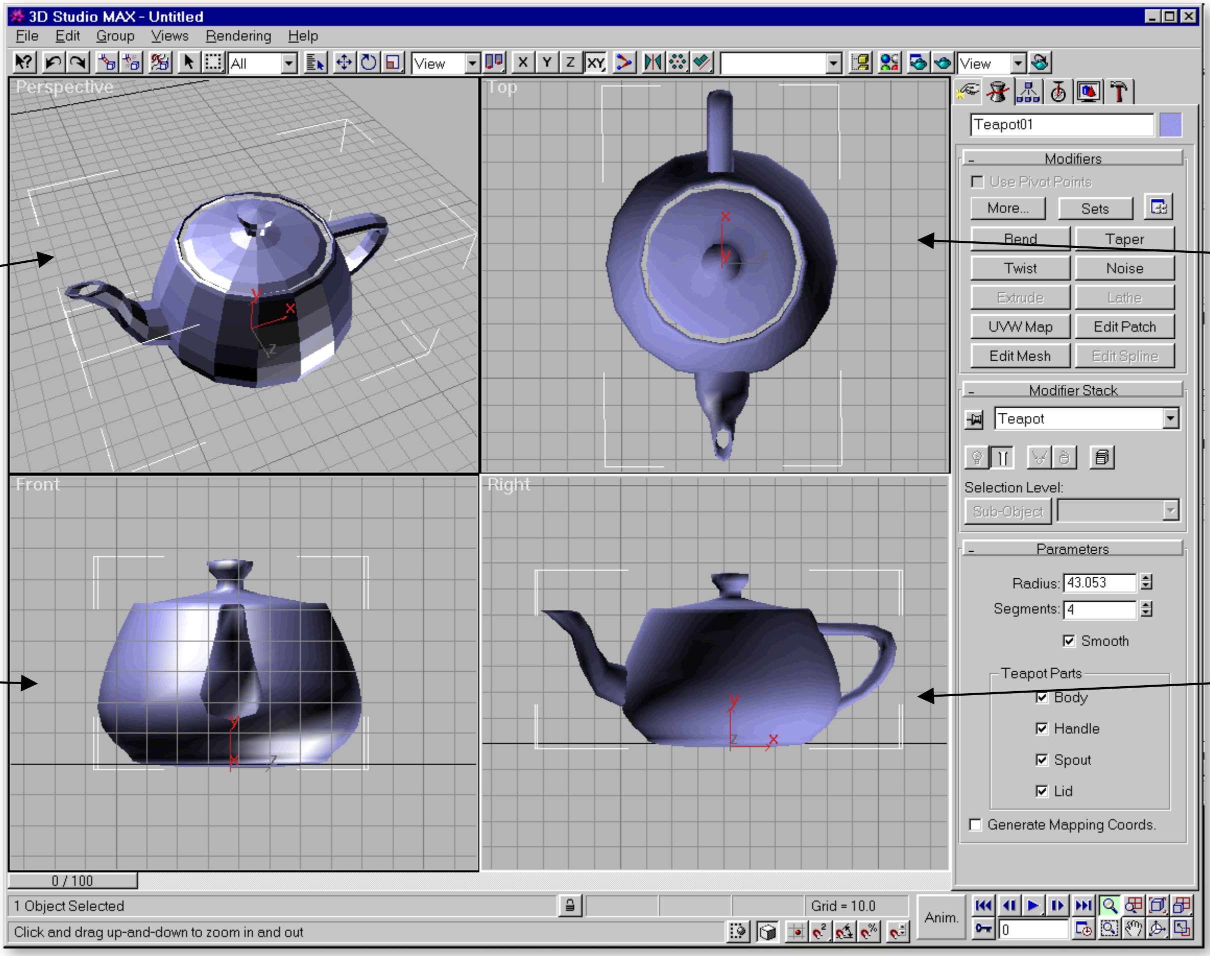
Orthogonal parallel projections in OpenGL®

glm::ortho(xmin, xmax, ymin, ymax, zmin, zmax**);**



Multi–projections in distinct viewports: *example*

- *This is performed through the re-positioning of the camera.*
- *Alternately, we can get the same result through the re-positioning of the object/scene.*



Teapot01

☐ Use Pivot Points

Teapot

Radius: 43.053

Segments: 4

☑ Smooth

Teapot Parts

☑ Body

☑ Handle

☑ Spout

☑ Lid

☐ Generate Mapping Coords.

0 / 100

1 Object Selected

Grid = 10.0

Click and drag up-and-down to zoom in and out

Perspective

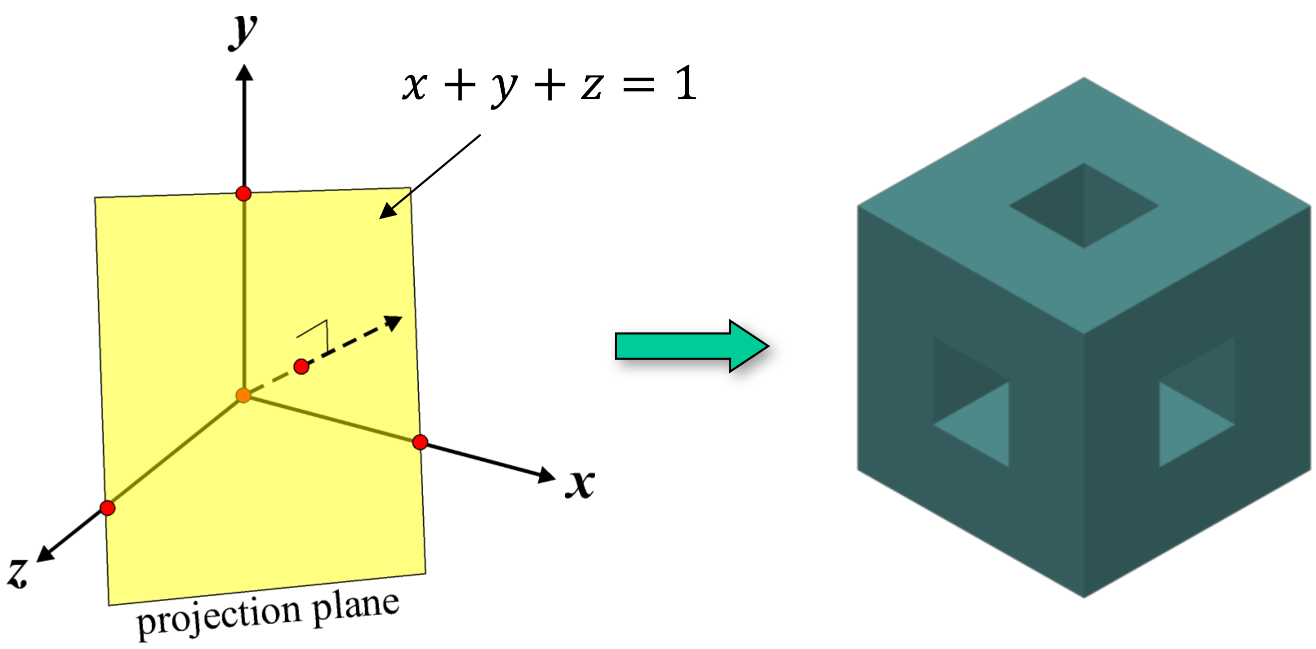
Top view

Left side view

Front view

Axonometric parallel projections: isometric, dimetric, and trimetric

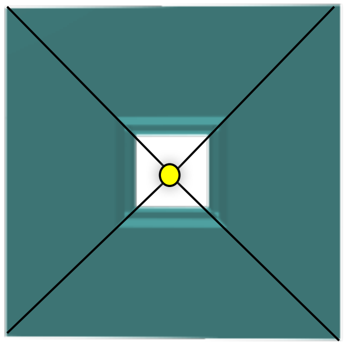
– *If the object is aligned with the axes, we obtain an* ***orthogonal*** *projection;*
– *Otherwise, we have na* ***axonometric*** *projection.*
– *If the projection plane intersects the axes XYZ to the same distance relative to the origin, we obtain na* ***isometric*** *projection.*



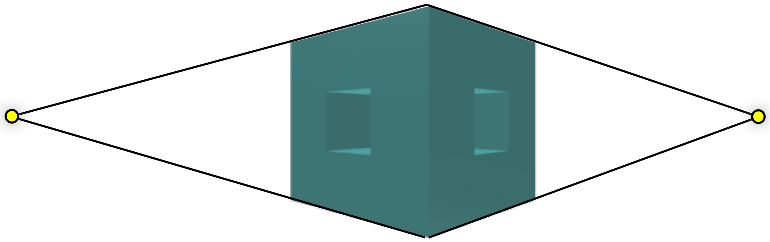
Perspective projections

- *The viewer is located at a finite distance from the object/scene.*
- *The visual rays are not parallel and converge to one or more points (viewers).*

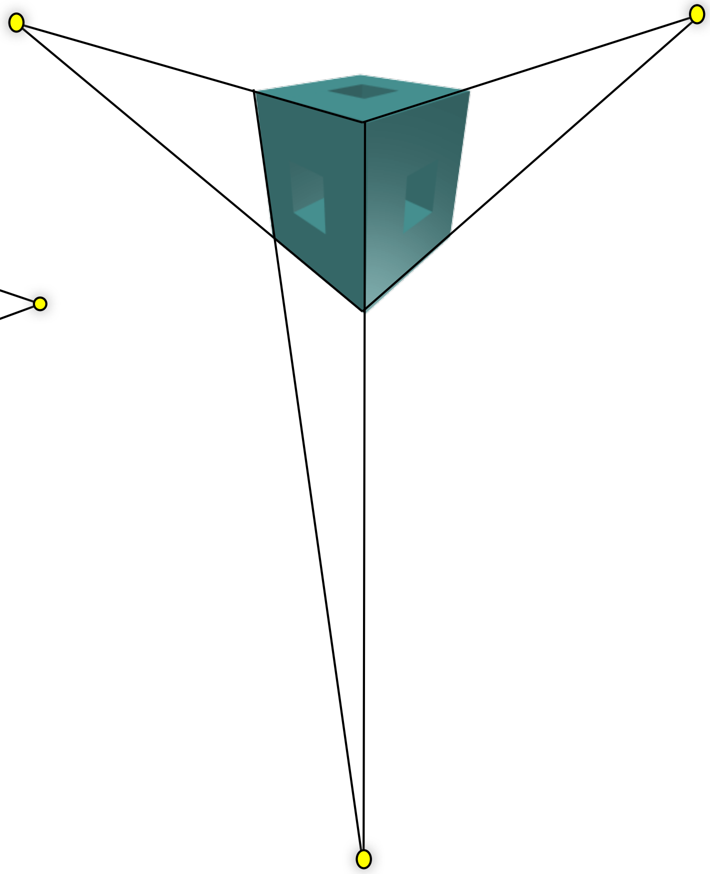
perspective with 1 point



perspective with 2 points



perspective with 3 points

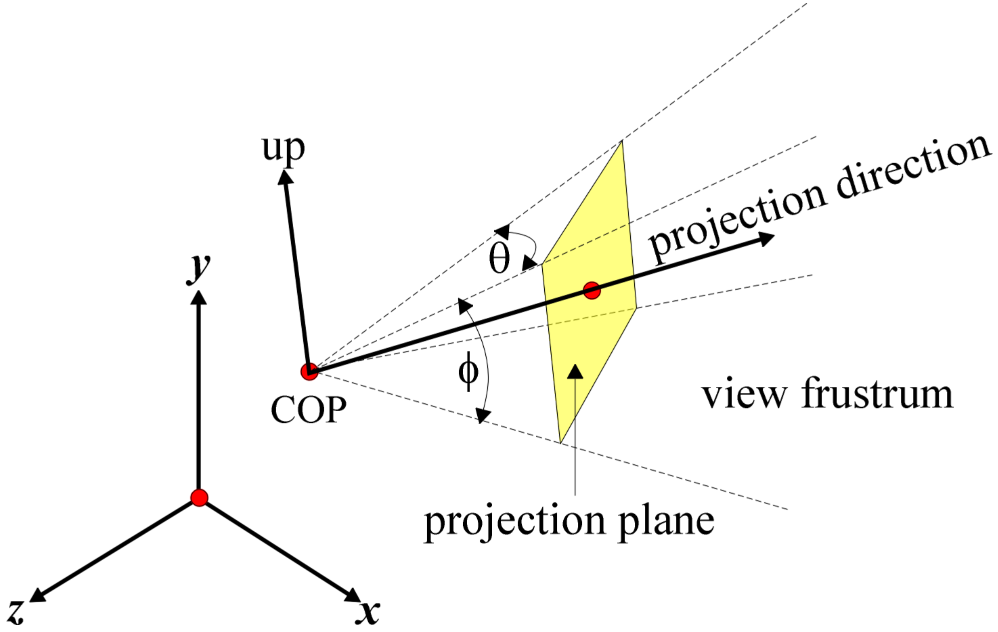


Projeções em perspectiva com um observador

– *The viewer is located at a* *<u>finite distance</u>* *from the object/scene.*
– *The visual rays converge to one point (viewer), known as COP (center of projection).*
– *In OpenGL, we use a single viewer.*

Projection parameters:

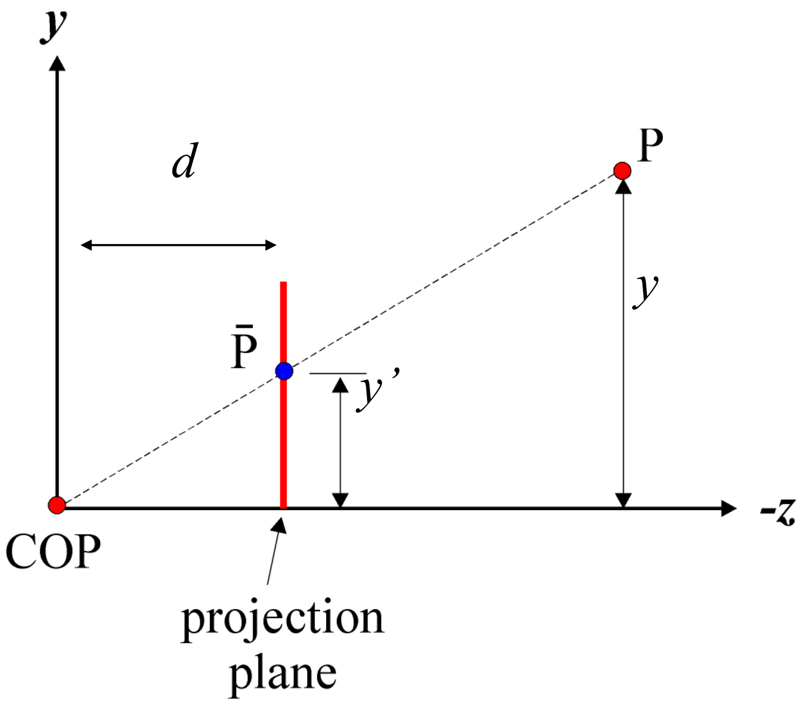
- center of projection (COP)
- view frustum (θ,ϕ), or field of view (FoV)
- projection direction
- up direction of the camera (or viewer) axis



Perspective projection matrix with a single viewer

- *Consider a perspective projection with:*

(a) the camera at the origin;
(b) view direction given by the negative z-axis;
(c) Projection plane at z = -d.

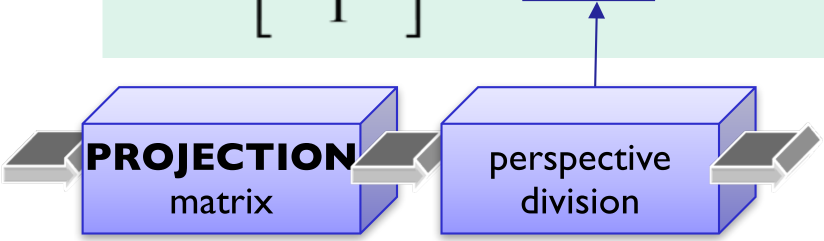


$$\frac{x}{-z} = \frac{x'}{d} \Rightarrow x' = \frac{x}{-z/d}$$

$$\frac{y}{-z} = \frac{y'}{d} \Rightarrow y' = \frac{y}{-z/d}$$

$$\begin{cases} x' = \dfrac{-xd}{z} = \dfrac{x}{-z/d} \\ y' = \dfrac{y}{-z/d} \\ z' = -d \end{cases}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \dfrac{x}{-z/d} \\ \dfrac{y}{-z/d} \\ -d \\ 1 \end{bmatrix} \Leftrightarrow \begin{bmatrix} x \\ y \\ z \\ \boxed{-z/d} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

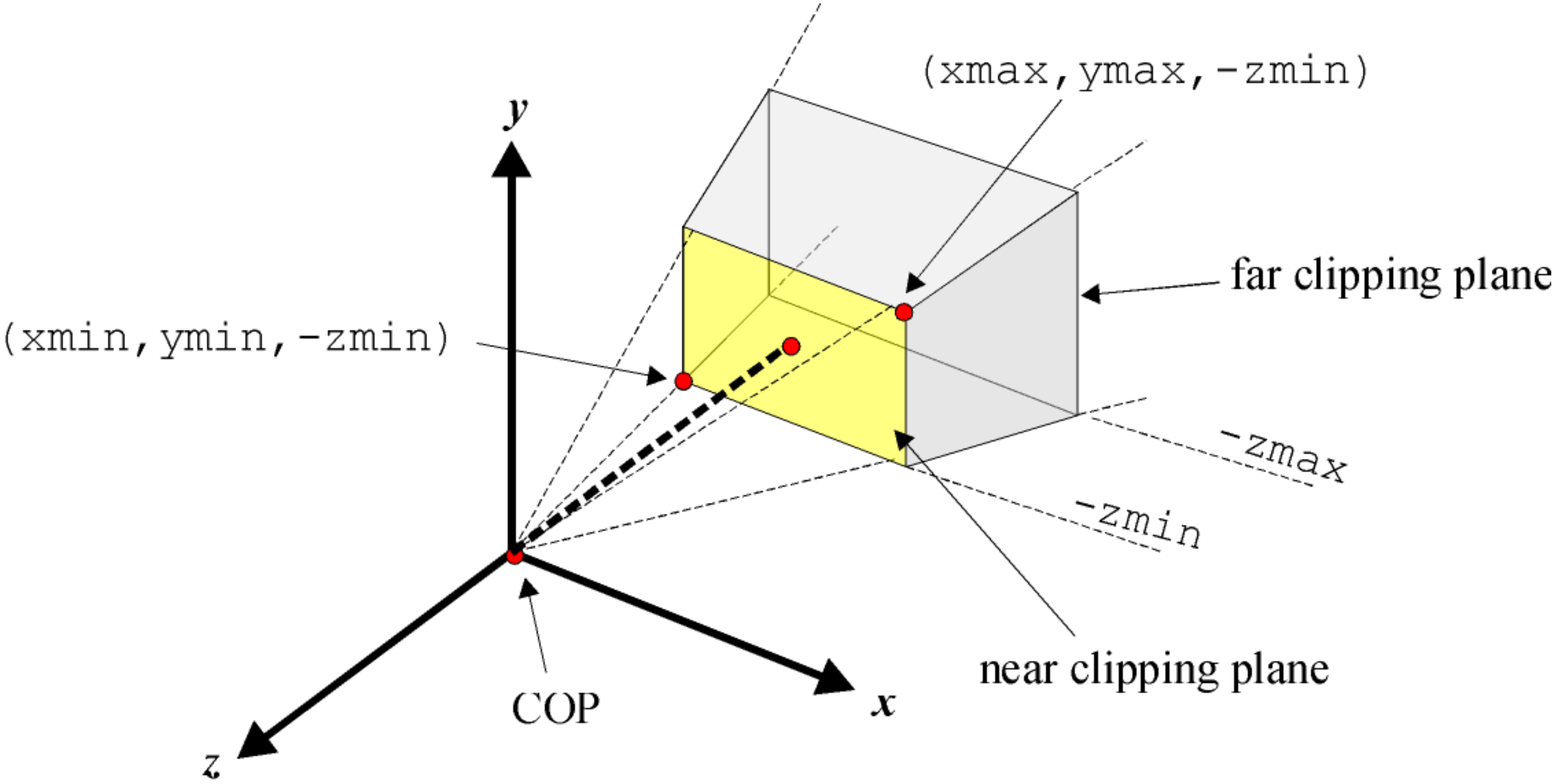


Perspective projection matrix with a single viewer in OpenGL®

frustum = truncated pyramid of the FoV

– *Using **glm::frustrum***

glm::frustum(xmin, xmax, ymin, ymax, zmin, zmax**);**



Perspective projection matrix with a single viewer in OpenGL® (cont.)

– *Using* ***glm::frustrum****:*

Specifying a *glm::frustrum*

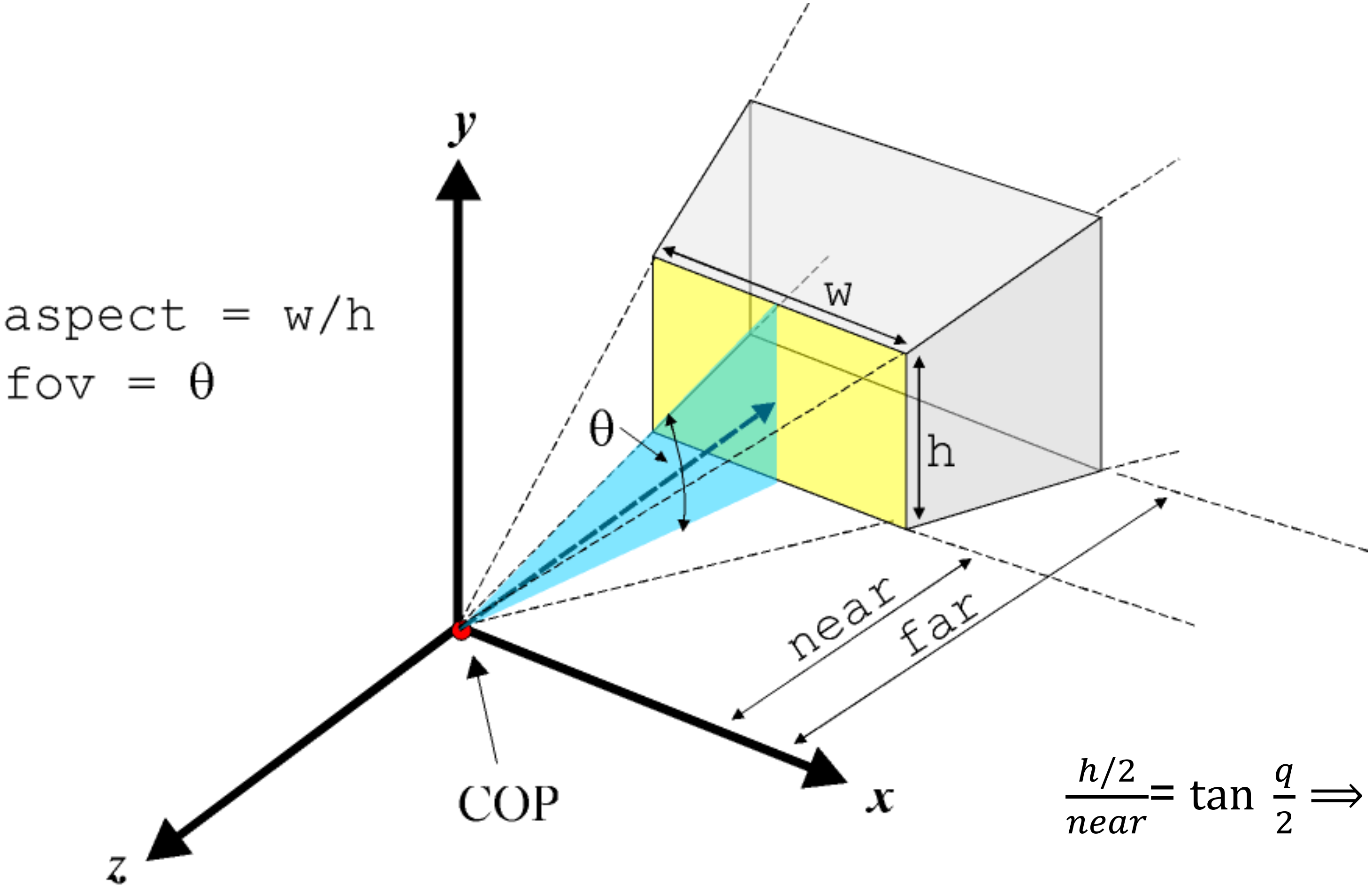
- All points belonging to the line defined by the COP and (xmin,ymin,-zmin) are mapped to the bottom-leftmost corner of the window.
- All points belonging to the line defined by the COP and (xmax,ymax,-zmin) are mapped to the top-righmost corner of the window.
- zmin e zmax are positive distances along -z
- The view direction is always parallel to –z
- It is not mandatory to have a symmetric frustrum, but a non-symmetric frustrum introduces *obliquity* in the projection.
  - For example, the following specification defines a non-symmetric frustum in OpenGL:

```
glm::frustrum(-1.0, 1.0, -1.0, 2.0, 5.0, 50.0);
```

Perspective projection matrix with a single viewer in OpenGL® (cont.)

– Using ***glm::perspective***

glm::perspective(fov, aspect, near, far**);**



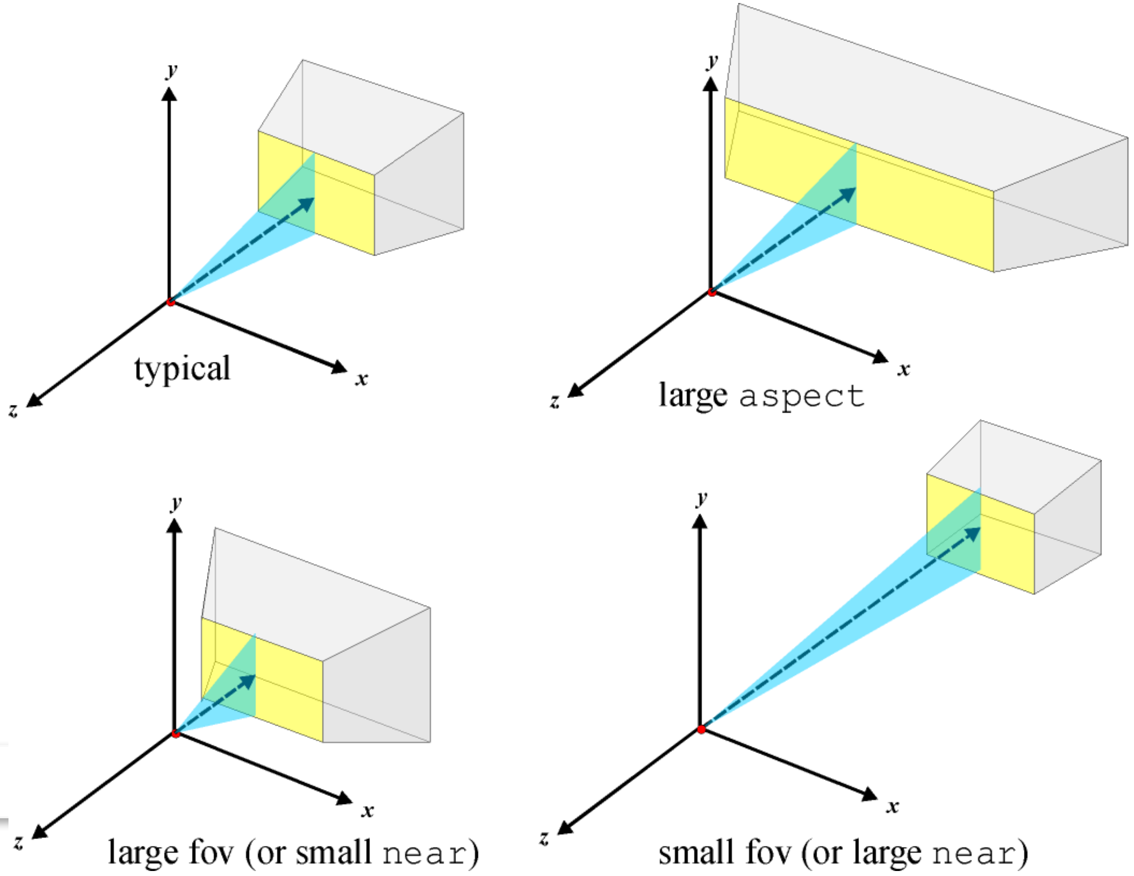
$$\frac{h/2}{near} = \tan\frac{q}{2} \Longrightarrow h = 2.near\,.\tan\frac{q}{2}$$

Perspective projection matrix with a single viewer in OpenGL® (cont.)

– *Using **glm::perspective***

Specifying a *glm::perspective*

- It only allows for symmetric frusta.
- COP at the origin, view direction along –z.
- FoV angle is in [0,180].
- *aspect* allows for a frustum with the same aspect ratio as the viewport as a way to avoid image distortion.



Exemplo:

glm::perspective(60, 1.0, 1.0, 50.0**);**

Moving camera in 3D

Limitations of *glm::frustum* and *glm::perspective*:

- fixed COP and fixed projection direction (or viewing direction)

Arbitrary positioning and orientation of the camera:

- For this purpose, we need to manipulate the MODELVIEW matrix before the generation of the scene objects. This way, we position the camera relative to scene objects.

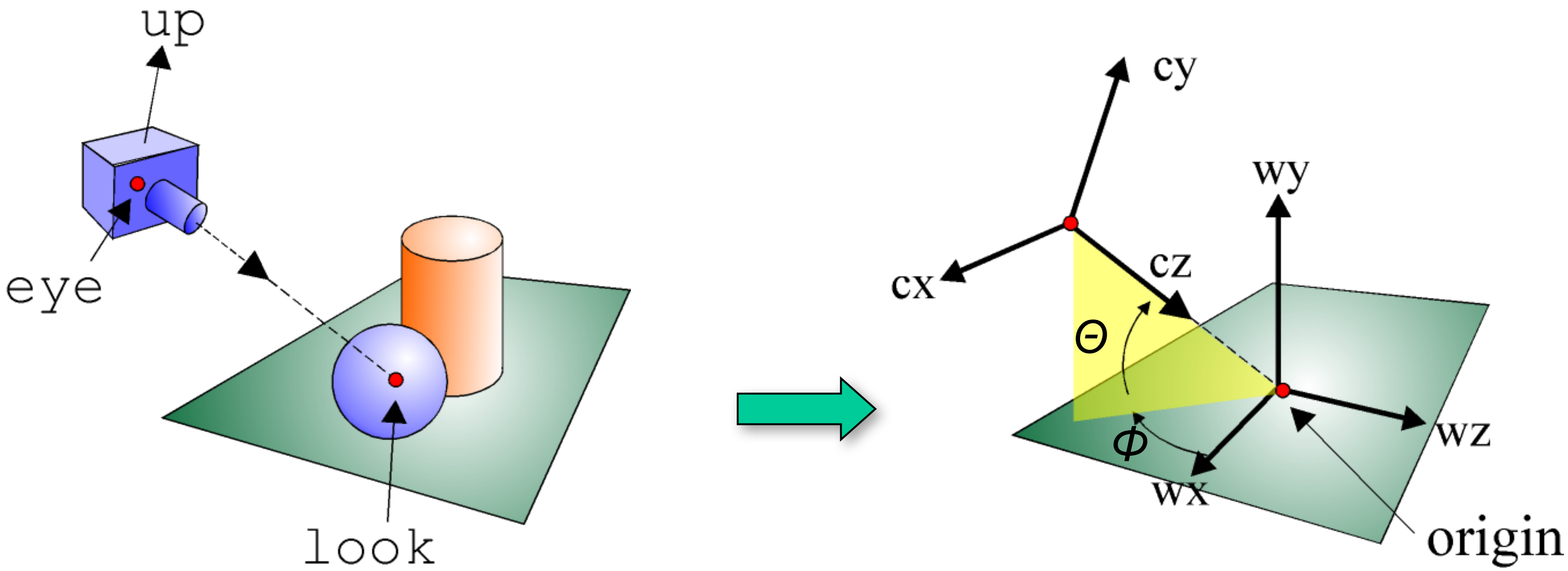
Example:

- There are 2 options to position the camera at (10.0, 2.0, 10.0) relative to the scene domain coordinate system:
  - To change the coordinate system of the scene domain before creating the scene objects, what is done using glm::**translate**(-10.0,-2.0,-10.0) and glm::rotate;
  - To use lookAt to position the camera relative to coordiante system of the scene domain: glm::**lookAt**(10, 2, 10, …);
- These 2 options are equivalent.

Moving camera in 3D using OpenGL®

– *Using **glm::lookAt**:*

glm::lookAt(eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz**);**



The same as:

```
glm::translate(-eyex, -eyey, -eyez);
glm::rotate(theta, 1.0, 0.0, 0.0);
glm::rotate(phi, 0.0, 1.0, 0.0);
```

Projection window

–After projecting a 3D scene onto a window of the projection plane, renderization takes place as in 2D.

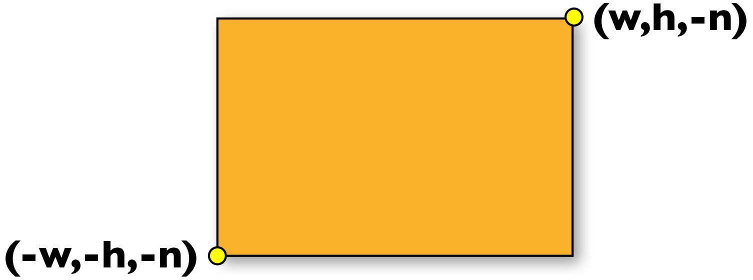
Definition:

- The projection matrix defines a transformation from the 3D scene domain coordinate system to a 2D window coordinate system belonging to the projection plane.
- The size of the projection window is defined as projection parameters:
  - **glm::frustrum(l,r,b,t,n,f)**

  - **glm::perspective(f,a,n,f)**

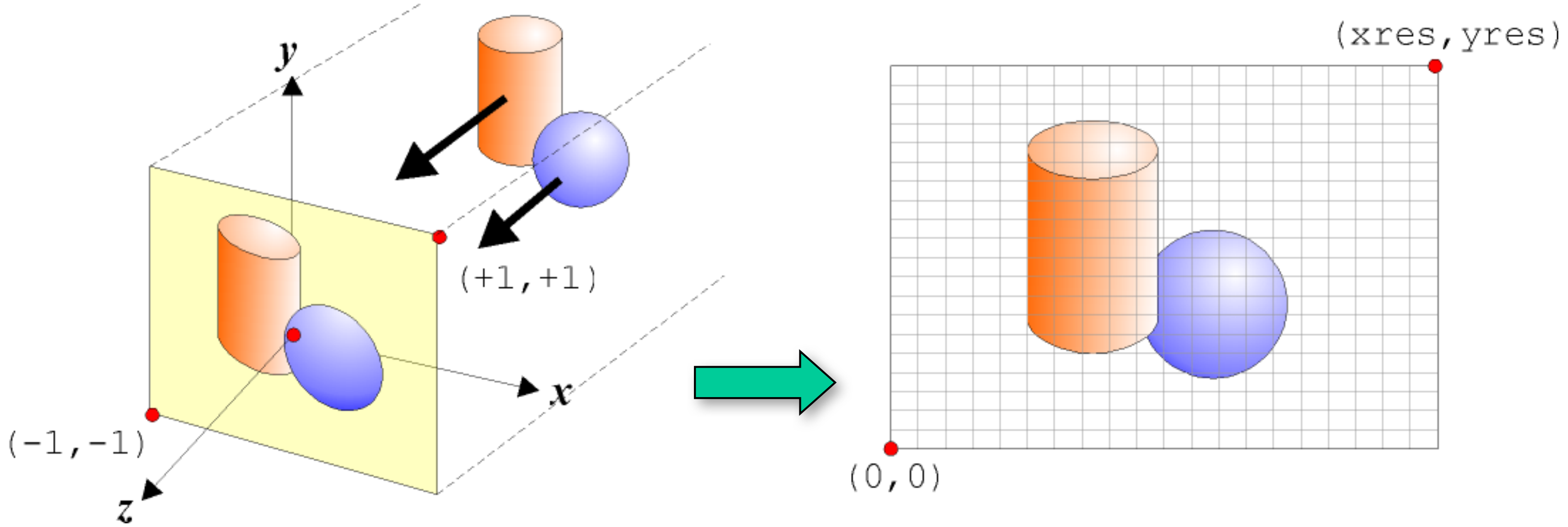
$$h = n.\tan\left(\frac{f}{2}\right)$$

$$w = h.a$$



Window-viewport transformation: *revisited*

–*After projecting a 3D scene onto a window of the projection plane, renderization takes place as in 2D.*
–*Indeed, it is necessary to map window points to viewport pixels todetermine the pixel associated to each vertex of the scene objects.*



coordinates of the
normalized output device

coordinates of the viewport

Window-viewport transformation: *revisited (cont.)*

Normalized coordinates:

- After the projection onto the plane, every point (x_p, y_p) of the projection window are transformed into (x_n, y_n) of the *normalized output device*: [-1,-1]×[+1,+1].

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

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

Viewport coordinates:

- Then, the graphics pipeline maps 2D normalized coordinates to one or more viewports

$$x_v = (x_n + 1)\left(\frac{width}{2}\right) + left$$

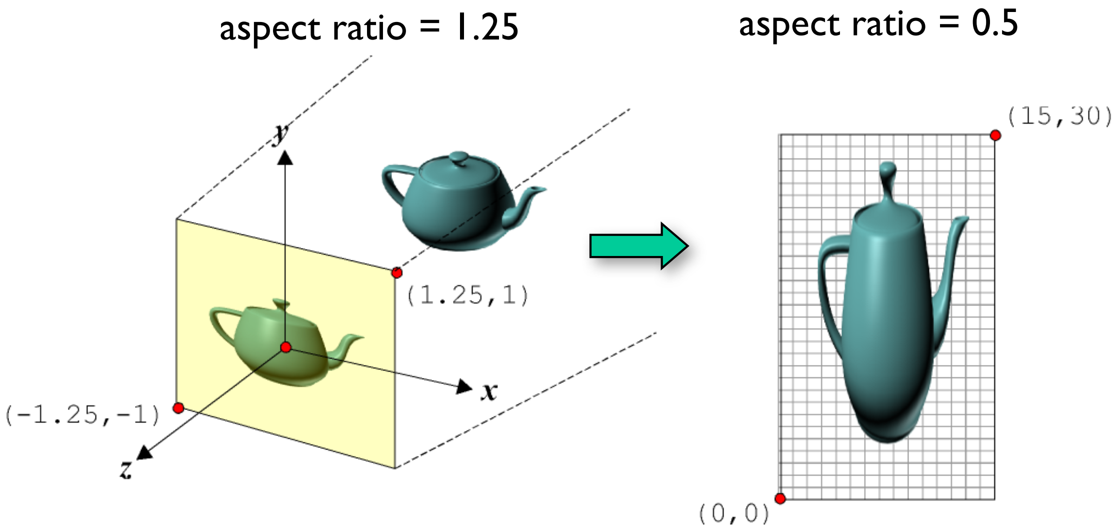
$$y_v = (y_n + 1)\left(\frac{height}{2}\right) + bottom$$

Event *resize*:

- Usually, we need to redefine the window after the resize event taking place to ensure the correct window-viewport transformation

```
static void reshape(int width, int height)
{
  glViewport(0, 0, width, height);
  glm::mat4 P = glm::perspective(85.0, 1.0, 5, 50);
}
```

Aspect ratio: *revisited*



Definition:

- The *aspect ratio* defines the ratio of width to height of a window or viewport.

In OpenGL:

- Explicitly given by a parameter or argument of glm::perspective.

How to avoid distortion?

- Both aspect ratios of window and viewport must be the same.

Examples in OpenGL

Example I: cube in a single view

– *Download **cube.zip** from course's web page for the full code of this graphics application.*

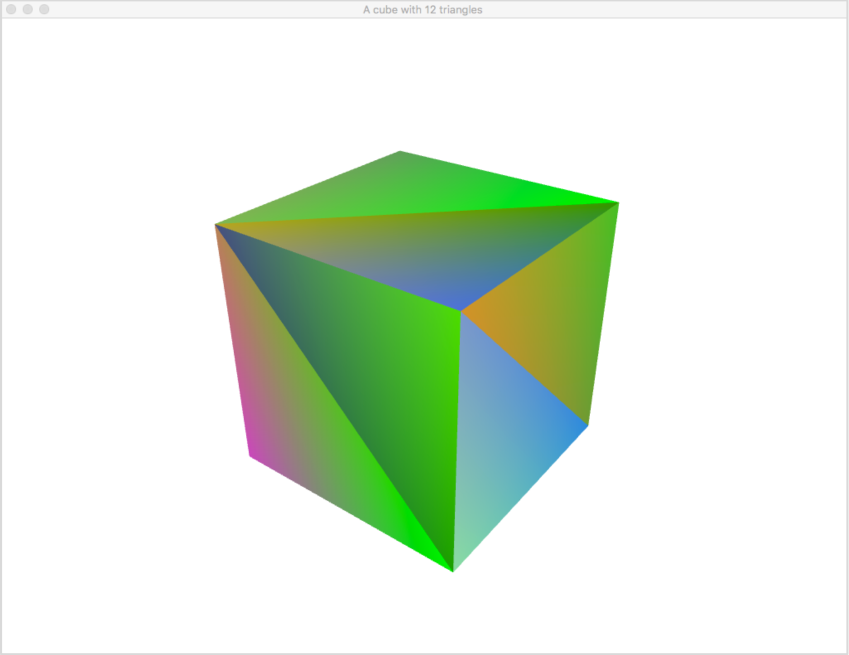
```
void setMVP(void)
{
   // Get a handle for our "MVP" uniform
   MatrixID = glGetUniformLocation(programID, "MVP");

   // Projection matrix :
   // 45° Field of View, 4:3 ratio,
   // display range : 0.1 unit <-> 100 units
    glm::mat4 Projection = glm::perspective(
                  glm::radians(45.0f),
                  4.0f / 3.0f,
                  0.1f,
                  100.0f);

    // Camera matrix
    glm::mat4 View = glm::lookAt(
      glm::vec3(4,3,-3),// Camera at (4,3,-3) in world space
      glm::vec3(0,0,0), // and looks at the origin
      glm::vec3(0,1,0)  // Head is up
     );

    // Model matrix: an identity matrix (model at origin)
    glm::mat4 Model = glm::mat4(1.0f);

    // Our MVP: multiplication of our 3 matrices
    MVP = Projection * View * Model;
}
```



Example 2: teapot in four views

```
void setMVP(void)
{

  MatrixID = glGetUniformLocation(programID, "MVP");

  // top left: top view
  glViewport(0, Height/2, Width/2, Height/2);
  glm::mat4 P = glm::ortho(-3.0, 3.0, -3.0, 3.0, 1.0, 50.0);
  glm::mat4 V = glm::lookAt(
                    0.0, 5.0, 0.0,
                    0.0, 0.0, 0.0,
                    0.0, 0.0, -1.0);
  glm::mat4 M = glm::mat4(1.0f);
  MVP = P * V * M;
  teapot();
. . .
  // bottom right: rotating perspective view
  glViewport(Width/2, 0, Width/2,  Height/2);
  glm::mat4 P = glm::perspective(70.0, 1.0, 1, 50);
  glm::mat4 V = glm::lookAt(
                    0.0, 0.0, 5.0,
                    0.0, 0.0, 0.0,
                    0.0, 1.0, 0.0);
  glm::mat4 M = glm::mat4(1.0f);
  glm::mat4 R = glm::rotate(45.0, 1.0, 0.0, 0.0);
  MVP = P * V * M * R;
  teapot();

  glutSwapBuffers();
}
```

Summary:

…:

- OpenGL rendering pipeline.
- Camera+plane+scene model.
- Camera types: classical camera, double-lens camera of Gauss, photorealsitic rendering camera.
- Rendering 3D scenes in OpenGL.
- Projection types: parallel projection and perspective projection.
- Projections in OpenGL.
- Moving camera.
- Projection window. Window-viewport transformation: revisited. Aspect ratio revisited.
- OpenGL examples.