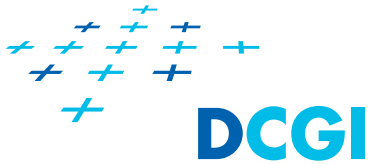


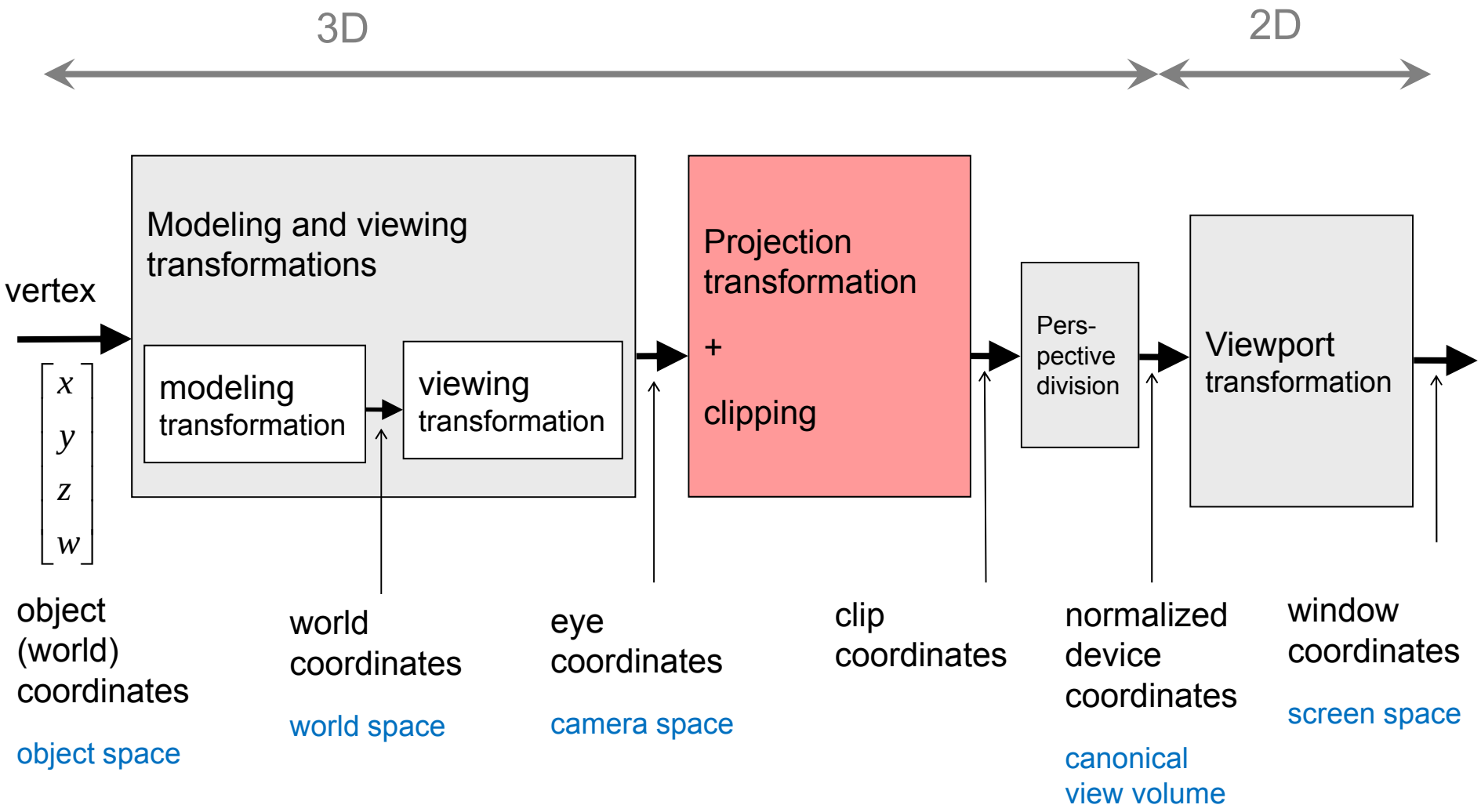
Transformace (2)
příloha

Jaroslav Sloup
Katedra počítačové grafiky a interakce, ČVUT FEL
místnost KN:E-413 (Karlovo náměstí, budova E)
E-mail: felkel@fel.cvut.cz

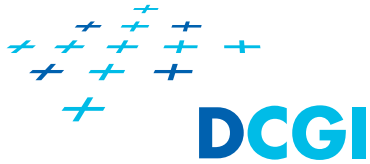
S použitím materiálů Bohuslava Hudce, Jaroslava Sloupa a Vlastimila Havrana

Poslední změna: 13.3.2013

Projection transformations



Projection transformations (contd.)

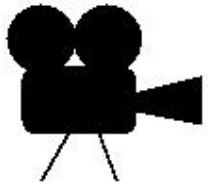


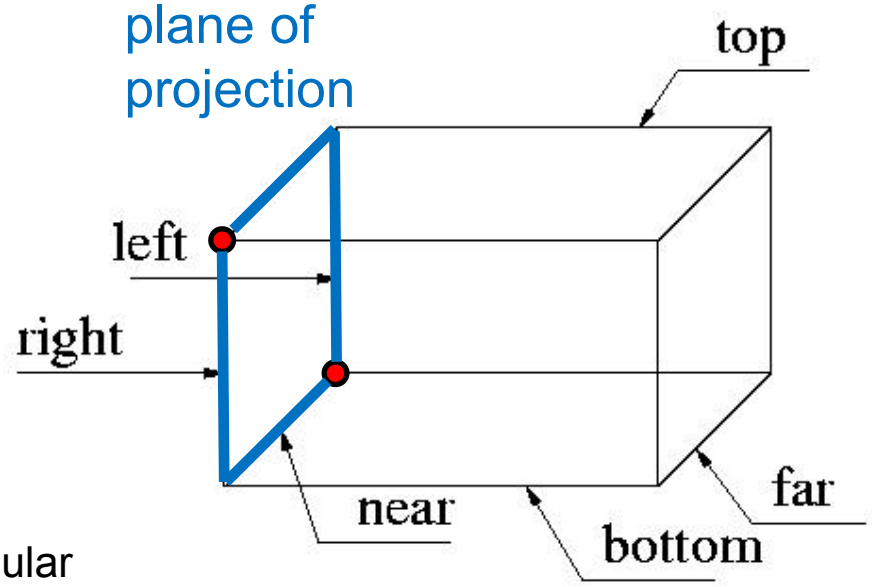
- the purpose of the projection transformation is to define a **viewing volume**, which is used in two ways
  - the viewing volume determines how an object is projected onto the screen (*that is, by using a perspective or an orthographic projection*)
  - it defines which objects or portions of objects are clipped out of the final image
- there are three types of projections supported by OpenGL
  - user defined (manually defined transformation matrix)
  - orthographic projection (parallel)
  - perspective projection
- note that projection transformation defines also so called clipping planes (6 planes – left, right, top, bottom, near, and far)

Orthographic projection

```
#include <glm/gtc/matrix_transform.hpp>
glm::mat4 glm::ortho(
        float left, float right,        /* range on x-axis  */
        float bottom, float top,        /* range on y-axis  */
        float near, float far );        /* range on z-axis  */
```

- creates a matrix for an orthographic parallel viewing volume
- [left, bottom, *] and [right, top, *] are points on the near/far clipping plane that are mapped to the lower-left and upper-right corners of the viewport window

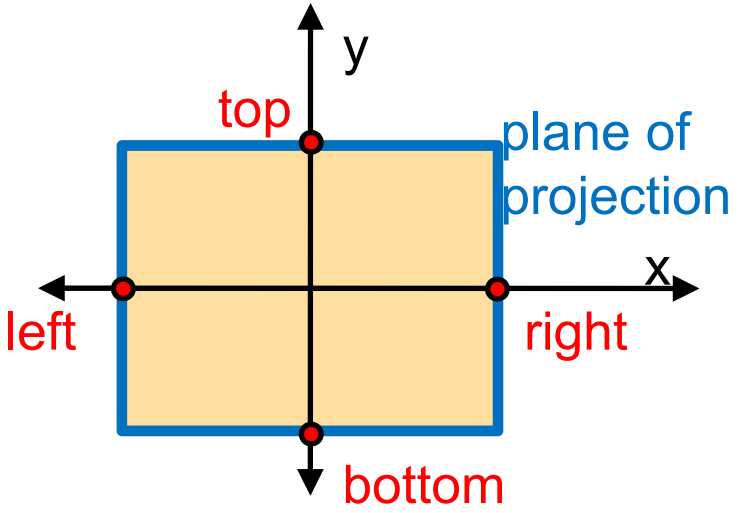




viewing volume = rectangular parallelepiped (box)

Orthographic projection (contd.)

- plane of projection size (*the xy-plane cutout*)



- transformation matrix of orthographic projection

$$M_{parallel} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- camera may be inside the viewing volume (**e.g. glm::ortho(-1,1,-1,1,-1,1)**)

  $\Rightarrow$ the objects *behind the camera are also displayed*

- camera can be totally shifted (i.e. outside the viewing volume, *it's like using a periscope*) e.g. **glm::ortho(-20,-10,-1,1,-1,1)**

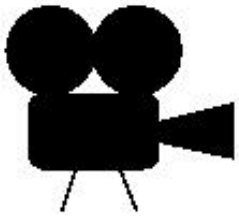
Perspective projection

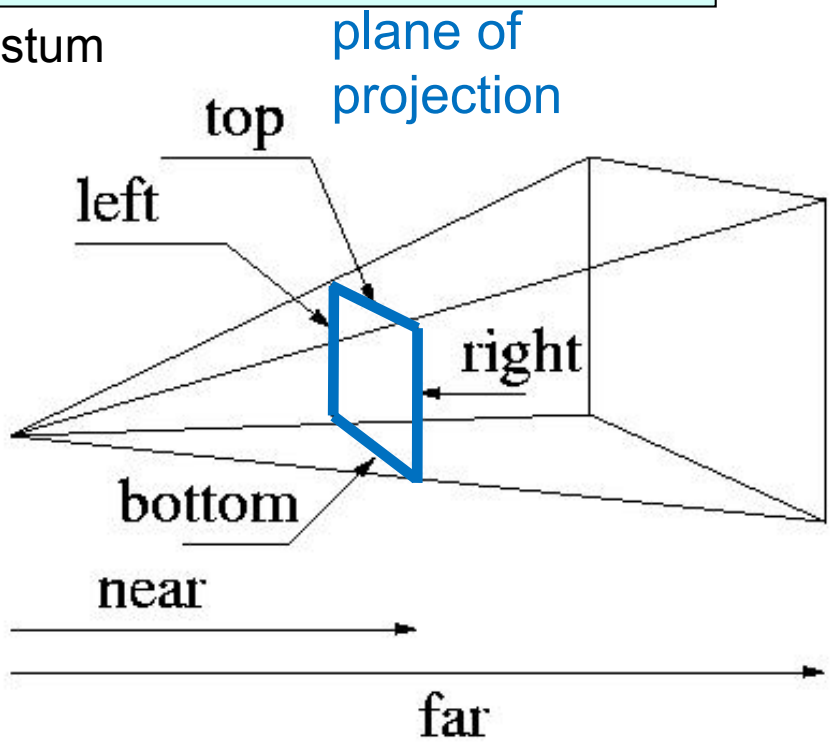
```
#include <glm/gtc/matrix_transform.hpp>
glm::mat4 glm::frustum(
        float left, float right,
        float bottom, float top,
        float near, float far );
```

- creates a matrix for a perspective-view frustum

- the viewing volume is a truncated pyramid whose top has been cut off by a plane parallel to its base

- objects that are closer to the viewpoint appear larger because they occupy a proportionally larger amount of the viewing volume





Perspective projection (contd.)

- transformation matrix of perspective projection

$$M_{frustum} = \begin{bmatrix} \frac{2.near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2.near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{(far+near)}{far-near} & -\frac{2.far.near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- viewing volume can be asymmetric

glm::frustum(-1,1,-1,1,1,3.5)

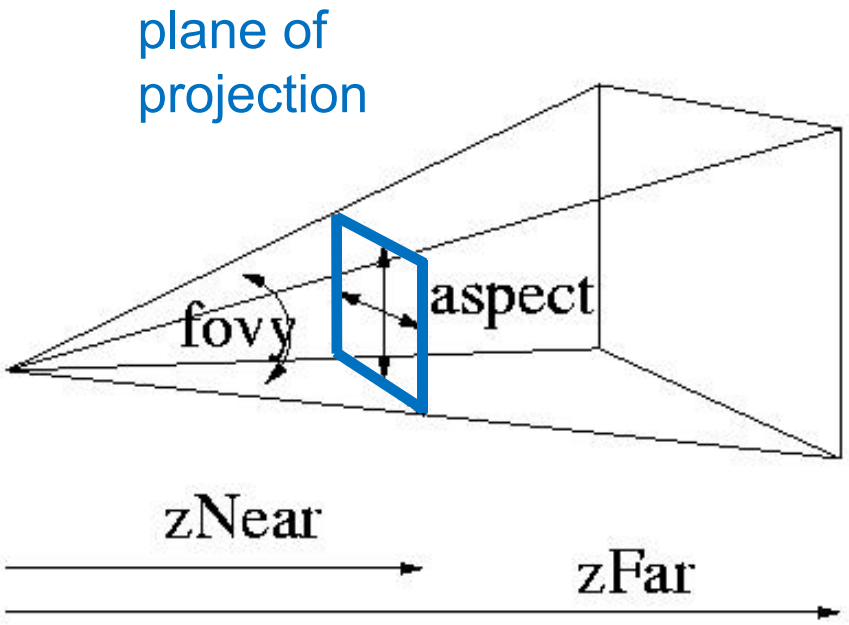
glm::frustum(-1,1,-1,1,1.6,3.5)

glm::frustum(-0.1,1,0.1,1,1,3.5)

Perspective projection (contd.)

```
#include <glm/gtc/matrix_transform.hpp>
glm::mat4 glm::perspective(
            float fovy, float aspect,
            float near, float far );
```

- creates a matrix for a symmetric perspective-view frustum
- **fovy** is the angle of the field of view in the x-z plane, it must be in the range [0.0,180.0]
- **aspect** is the aspect ratio of the frustum, its width divided by its height
- near and far values should always be positive (near > 0)

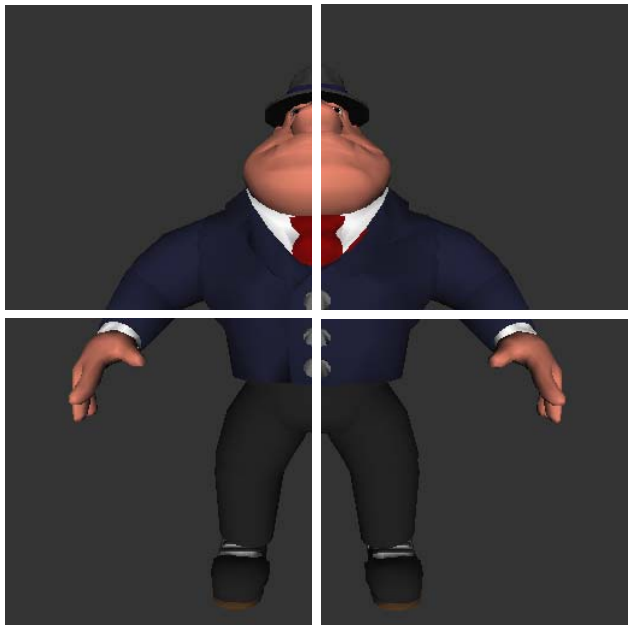


Perspective projection (contd.)

How to get image in double the maximal resolution?

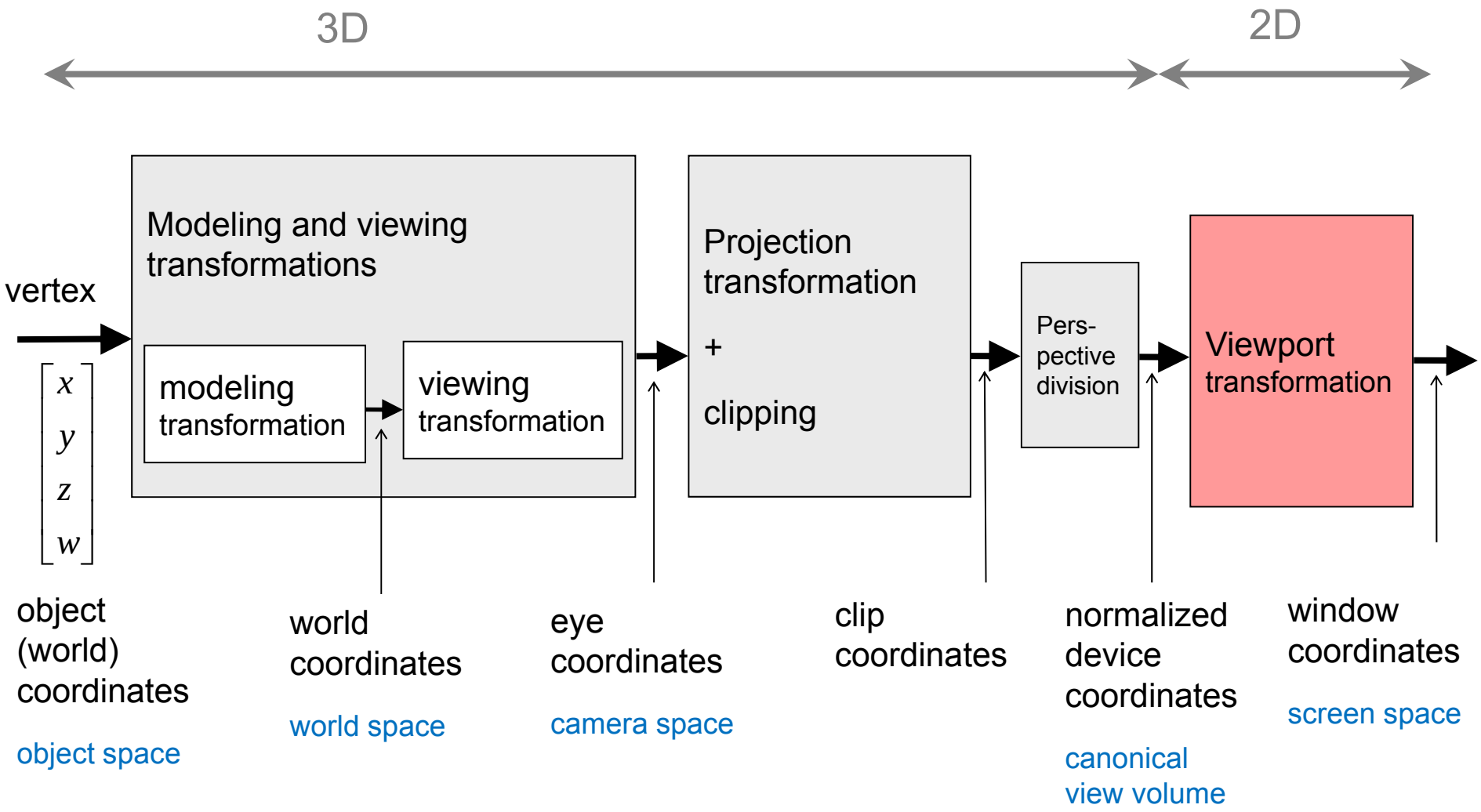
- draw the scene four times setting the projections as shown in the table
- save the rendered images
- glue these images together (*e.g. in Photoshop or Gimp*)

x=[left, right] y=[bottom, top]	
x=[-1, 0] y=[0, 1]	x=[0, 1] y=[0, 1]
x=[-1, 0] y=[-1, 0]	x=[0, 1] y=[-1, 0]



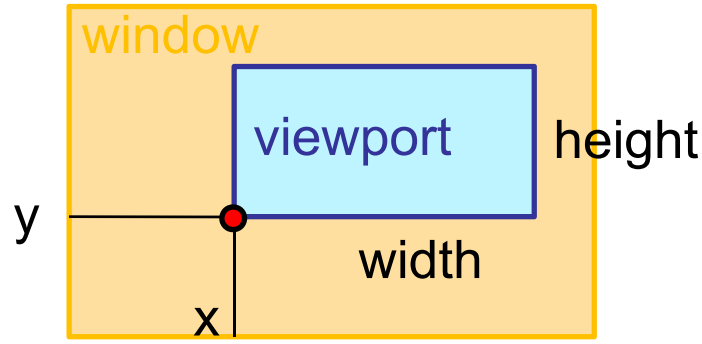
```
glm::mat4 matrix;
/* upper left part */
matrix = glm::frustum(-1,0,0,1, 1.6,3.5);
RenderModel();
/* upper right part */
matrix = glm::frustum(0,1,0,1, 1.6,3.5);
RenderModel();
/* lower left part */
matrix = glm::frustum(-1,0,-1,0, 1.6,3.5);
RenderModel();
/* lower right part */
matrix = glm::frustum(0,1,-1,0, 1.6,3.5);
RenderModel();
```

Viewport transformation



Viewport transformation (contd.)

- the viewport is the rectangular region of the window where the image is drawn

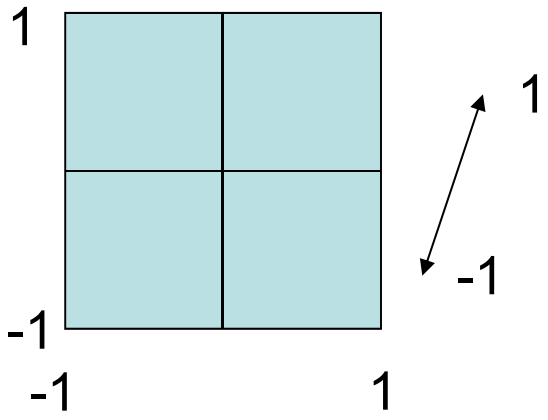



```
void glViewport(
        GLint x, GLint y, GLsizei width, GLsizei height);
```

- defines a pixel rectangle in the window into which the final image is mapped
- viewport is usually set in the reshape callback
- the aspect ratio of a viewport should generally equal the aspect ratio of the viewing volume, if these two ratios are different, the projected image will be distorted when mapped into the viewport
- your application should detect window resize events and modify the viewport appropriately

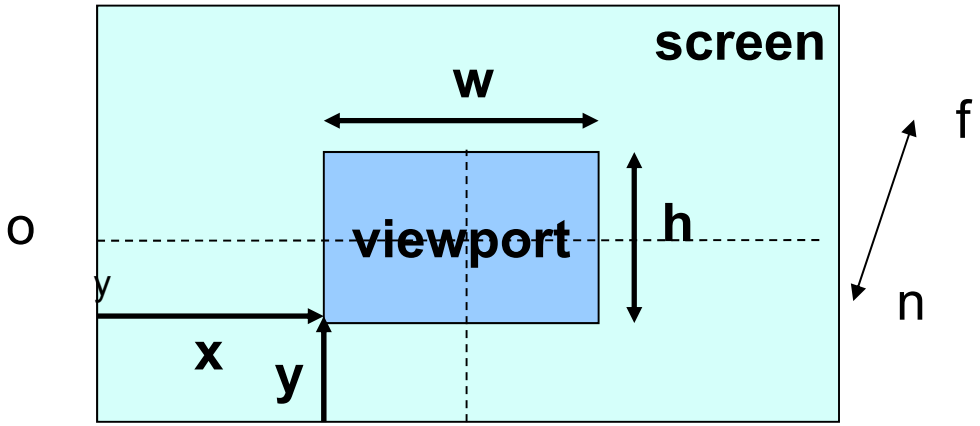
Viewport transformation (contd.)

normalized device coordinates $[x_d, y_d, z_d]$



window coordinates

$[x_w, y_w, z_w]$



$x_w = (w / 2)\, x_d + o_x$

$y_w = (h / 2)\, y_d + o_y$

$z_w = [(f-n) / 2]\, z_d + (n+f) / 2$

z_w visibility testing (Z-buffer)

$o_x = x + w/2$

$o_y = y + h/2$

glDepthRange(n, f)

Set depth range → clamp(n,f)

near near clipping plane 0.0

far far clipping plane 1.0

Viewport transformation (contd.)

Example: Rendering into two different viewports.

```
glm::mat4 matrix = glm::mat4(1.0);
glm::mat4 projectionMatrix = glm::perspective(
            60, winW/(2.0*winH), 0.1, 10 );
glViewport( 0, 0, winW/2, winH );   /* left viewport */
passMatrixToVertexShader( projectionMatrix );
passMatrixToVertexShader( matrix );
drawModel();

/* right viewport */
matrix  = glm::rotate(
  matrix, 45, glm::vec3(1.0f, 1.0f, 0.0f));
glViewport( winW/2, 0, winW/2, winH );
passMatrixToVertexShader( matrix );
drawModel();
```

PGR

winW = 600… window width
winH = 300 … window height