

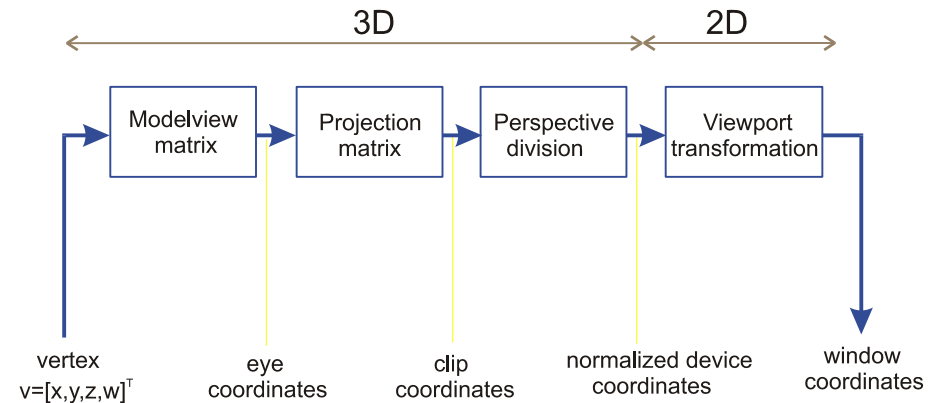


Objects are usually modeled in a basic position
(in the origin)

A way how to get an object in a different position is
to transform it.

Transformations of a vertex are:

- viewing
- modeling
- projection
- clipping
- viewport



Stages of vertex transformations

- Viewing consists of two parts:

positioning and projection

OpenGL supports orthographic and perspective projections

Other projections can be defined “by hand”

- coordinate system is represented as a matrix
- transformation is represented as a matrix multiplication
- matrix is 4x4 type



Matrices

- new (transformed) vertex coordinates v'
are calculated from

- old ones: v

- and matrix: M

$v' = Mv$ (always on the right side)

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$



We can

- a) transform objects
 - b) transform camera
- (operations are mutual)

The way it works:

- ~ set transforms
 - ~ send data (the data will be transformed)
- so FIRST we must define the transformation
THEN we use it by sending the data



- viewing and modeling transformations are combined into single modelview matrix
- but it is easier to think about it in this way
- we can position the camera in two ways
 - move objects (modeling transformation)
 - move camera (viewing transformation)



```
void glMatrixMode(GLenum mode)
```

where

mode is GL_MODELVIEW
 or GL_PROJECTION
 or GL_TEXTURE

subsequent command affects the specified matrix
(remember it is state machine!)



```
void glLoadIdentity(void)
```

- sets the currently modifiable matrix to 4x4 identity matrix

```
void glTranslate{fd}(TYPE x,TYPE y,TYPE z)
```

- translates by (x,y,z)

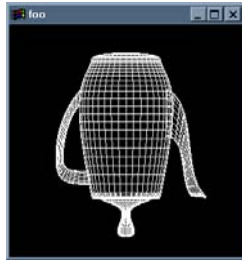
```
void glScale{fd}(TYPE x,TYPE y,TYPE z)
```

- scales by (x,y,z)



example:

```
glLoadIdentity();
DrawObject();
glScalef(0.8, -2.0, 1.3);
DrawObject();
```



(C) Bedrich Benes



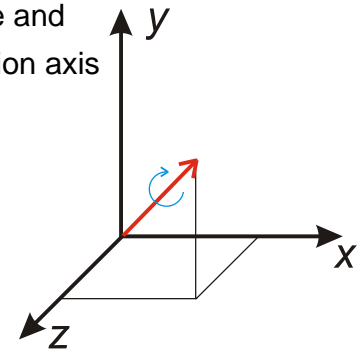
```
void glRotate{fd}(TYPE angle,
                  TYPE x,TYPE y,TYPE z)
```

where

angle specifies the rotation angle and
x, y, and z, specifies the rotation axis

note:

angle is <0,360>

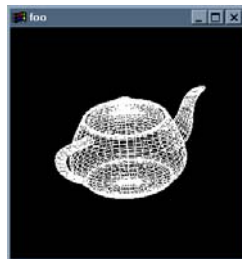


(C) Bedrich Benes



example:

```
glLoadIdentity();
DrawObject();
glRotatef(45,-1,-1.5,0.5);
DrawObject();
```



(C) Bedrich Benes



```
void glLoadMatrix{fd}(const TYPE *m)
```

sets the current matrix values to those specified in m

note:

```
glMatrixMode(GL_PROJECTION);
```

```
glLoadMatrix(myMatrix);
```

is the way, how to define your own projection
(e.g.) two point perspective
(it is not quite common)

(C) Bedrich Benes



```
void glmMultMatrix{fd}(const TYPE *m)
```

multiplies the current matrix values by the sixteen values specified in `m`, and stores the result as the current matrix (it is not quite common as well)

- remember, it is multiplied from the right side

C - current matrix, **M** - new matrix

```
C ← C M
```



in C we declare matrix as `GLfloat m[4][4];`

`m[i][j]` specifies *i*-th column and *j*-th row

exactly *the reverse order* of OpenGL matrices

better declare matrices as `GLfloat m[16];`

```
GLfloat m[16] = {m0,m1,m2,m3,m4,m5,m6,m7,m8,
                 m9,m10,m11,m12,m13,m14,m15};
```

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$



```
void glGetFloatv(GLenum what, GLfloat *m)
```

where

what is `GL_MODELVIEW_MATRIX`

or `GL_PROJECTION_MATRIX`

or `GL_TEXTURE_MATRIX`

`m` is pointer to matrix 4x4

current matrix is returned in `m`

example (read the current projection matrix):

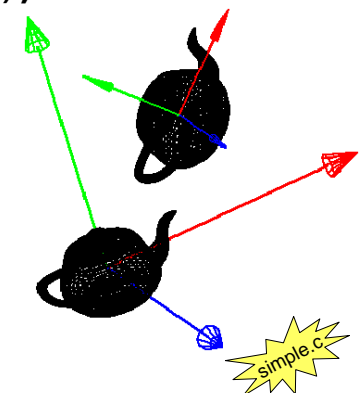
```
GLfloat m[16]; // here the data is stored
```

```
GLfloat m[16]; // here the data is stored
glGetFloatv(GL_MODELVIEW_MATRIX, m);
```



the order of transformation is critical!

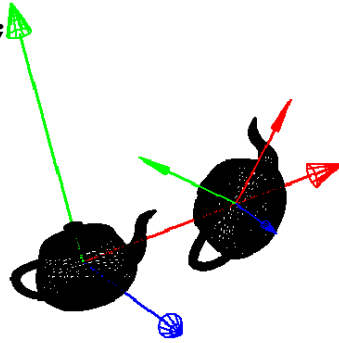
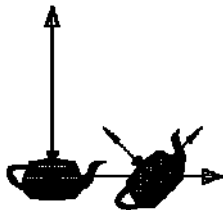
```
glMatrixMode(GL_MODELVIEW);
glColor3ub(0,0,0);
glutWireTeapot(0.3);
glRotatef(45,0,0,1);
glTranslatef(1,0,0);
glutWireTeapot(0.3);
```





the order of transformation is critical!

```
glMatrixMode(GL_MODELVIEW);
glColor3ub(0,0,0);
glutWireTeapot(0.3);
glTranslatef(1,0,0);
glRotatef(45,0,0,1);
glutWireTeapot(0.3);
```



the order of transformation is critical
because matrix multiplication is *not* commutative

$$C' = RT$$

$$C'' = TR$$

$$C' \neq C''$$



The first approach: **the grand fixed coordinate system**

we must read the transform in the inverse order

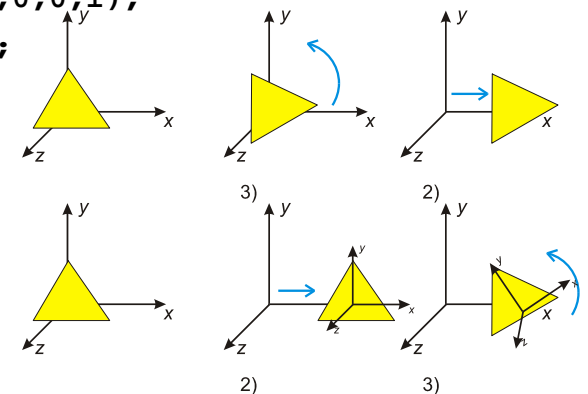
The second approach: **local coordinate system tied to object**

operations are applied to the local coordinate system
operations occur in natural order



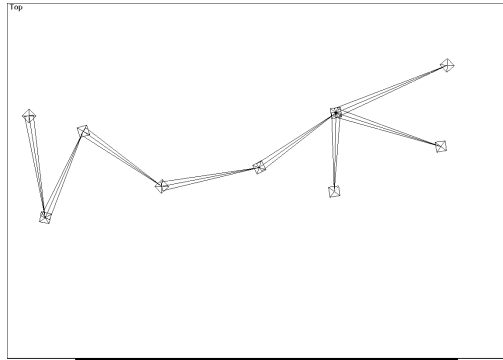
Example

```
1: glLoadIdentity();
2: glTranslatef(1,0,0);
3: glRotatef(45,0,0,1);
4: DrawObject();
```





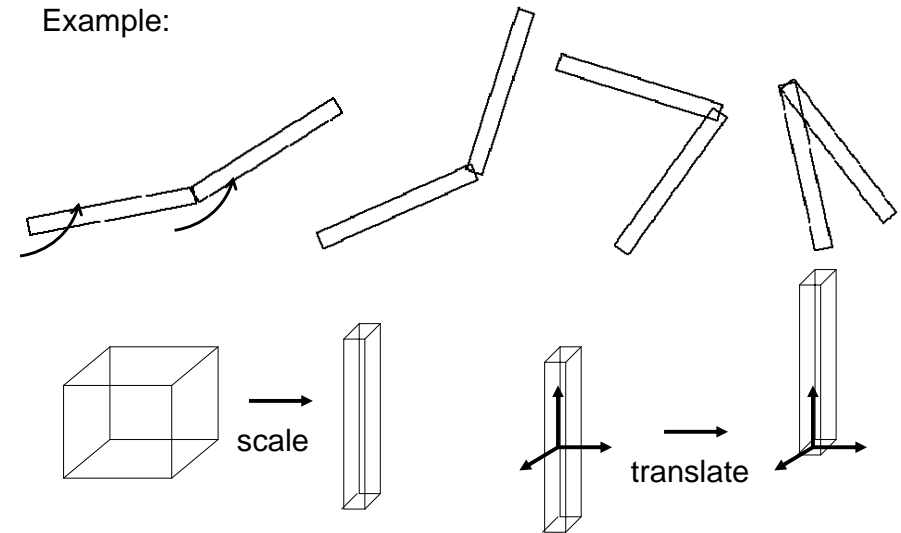
- the second approach is more useful for modeling a sets of joints (robot arms)



- the first approach is problematic in cases of scaling (non-uniform ~ axes may be non perpendicular)



Example:



Example:

```
glLoadIdentity();           //reset
glRotatef(r1,0,0,1);        //the main segment
glTranslatef(0.5,0,0);      //to the origin
glScalef(1,0.1,0.1);       //scale it
glutWireCube(1.f);         //make the cube
glScalef(1,10,10);         //scale it back
glTranslatef(0.5,0,0);      //to the end
glRotatef(r2,0,0,1);        //the second segment
glTranslatef(0.5,0,0);      //to the origin
glScalef(1,0.1,0.1);       //scale
glutWireCube(1.f);         //and display
r1+=0.1;
r2+=0.2;
```



viewing transformation

are application of rotation, translation, and scaling to camera

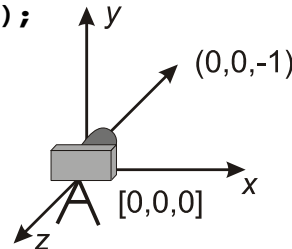
- viewing transformation should be issued before any modeling transformation in a program
⇒ viewing transformation is issued first

- modeling transformation is **discussed** first
- viewing transformation are **issued** first

i.e., first compose a scene then place a camera



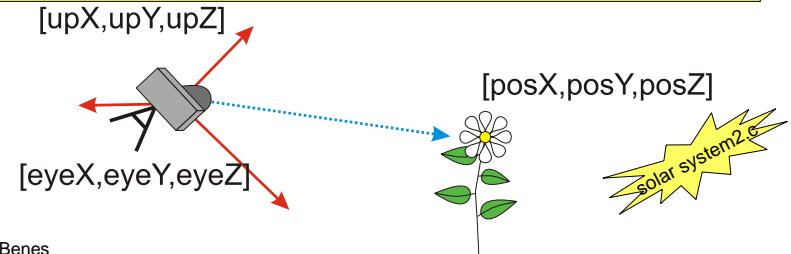
- move camera = move all objects in the opposite way
- rotate camera CW = rotate all objects CCW
- viewing transformation commands should be called before any modeling ones
- initial position and orientation of camera i.e., after
`glMatrixMode(GL_PROJECTION);`
`glLoadIdentity();`
- position [0,0,0]
- looking at (0,0,-1)



Viewing transformations (contd.)

- positioning camera in 3D space is too complex
- it can be simplified using GLU library function

```
gluLookAt(  
GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,  
GLdouble posX, GLdouble posY, GLdouble posZ,  
GLdouble upX, GLdouble upY, GLdouble upZ)
```



The matrices we see are the topmost matrices of stacks
 All the operations are applied to the top

```
glLoadMatrix(), glMultMatrix(), glLoadIdentity()
```

We can push the matrix down and pop it up

When pushed, the matrix is *copied* onto the top

```
void glPopMatrix()  
void glPushMatrix()
```

- the current stack is determined by `glMatrixMode()`
- the stack depth can be found with
`glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH)`
`glGetIntegerv(GL_MAX_PROJECTION_STACK_DEPTH)`



Why this?

- making an inverse transformation is expensive
- saving the matrix is cheap

- any operation can be inverted by

```
glPushMatrix();
```

```
    Perform_any_Transformations_Here();
```

```
glPopMatrix();
```



Example (slide 23)

```
glPushMatrix();           //remember the state
glRotatef(r1,0,0,1);      //the main segment
glTranslatef(0.5,0,0);    //to the origin
glPushMatrix();
    glScalef(1,0.1,0.1);  //scale it
    glutWireCube(1.f);    //make the cube
glPopMatrix()
glTranslatef(0.5,0,0);    //to the end
glRotatef(r2,0,0,1);      //the second segment
glTranslatef(0.5,0,0);    //to the origin
glScalef(1,0.1,0.1);      //scale
glutWireCube(1.f);        //and display
r1+=0.1;r2+=0.2;
glPopMatrix();           //go back
```



The purpose is to define a viewing volume
Objects inside this volume are rendered
others are clipped

projection transformation can be:

- user defined
- orthographic projection (parallel)
- perspective projection

Projection defines so called clipping planes



User defined projection

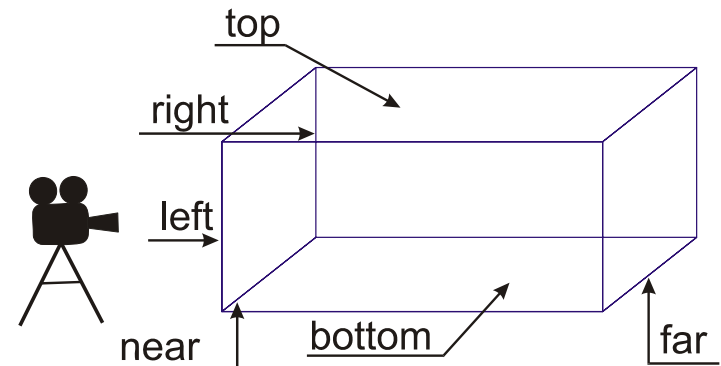
```
glMatrixMode(GL_PROJECTION);
glLoadMatrix(m);
```

where *m* defines the projection matrix
e.g., two point perspective, axonometry etc.



Orthographic projection

```
void glOrtho(GLdouble left, GLdouble right,
             GLdouble bottom, GLdouble top,
             GLdouble near, GLdouble far)
```





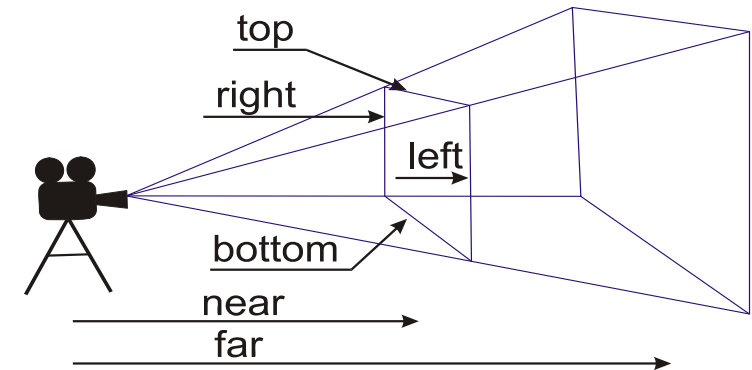
Orthographic projection comments

- camera can be inside the volume
`glOrtho(-1,1,-1,1,-1,1)`
then also the object *behind* the camera are displayed
- camera can be totally shifted
`glOrtho(-12,-10,-1,1,-1,1)`
it's like using a periscope



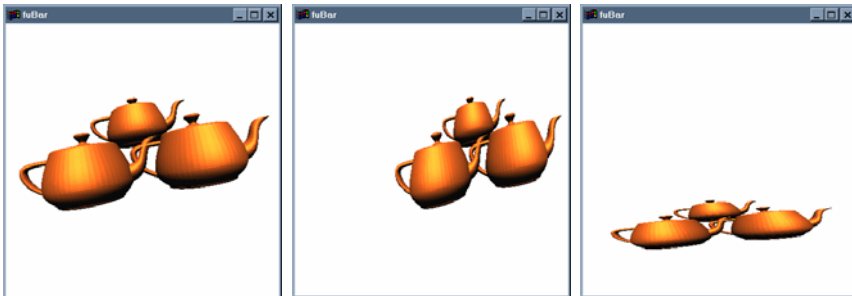
Perspective projection

```
void glFrustum(GLdouble left, GLdouble right,
               GLdouble bottom, GLdouble top,
               GLdouble near, GLdouble far)
```



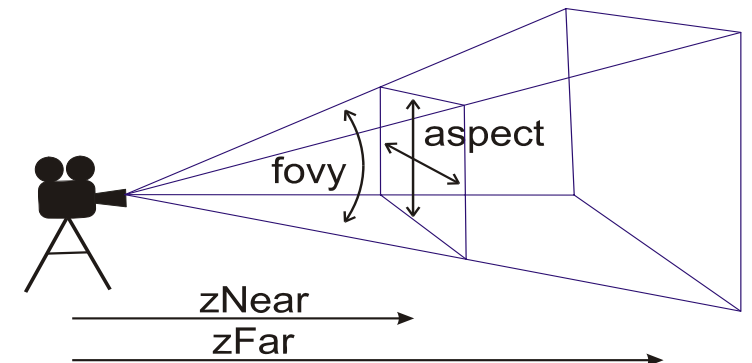
glFrustum

- is not intuitive
- can be asymmetric
(sometimes useful, sometimes confusing)



Perspective projection

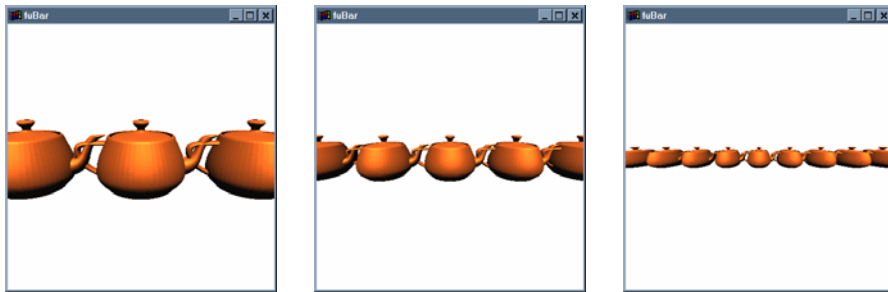
```
void gluPerspective(GLdouble fovy,
                    GLdouble aspect,
                    GLdouble zNear, GLdouble zFar)
```





gluPerspective

- **fovy** is field of view in the y axis
- **aspect** is aspect ratio of the actual viewport
- it cannot be asymmetric (sometimes desired)



fovy=60

90

120

(C) Bedrich Benes

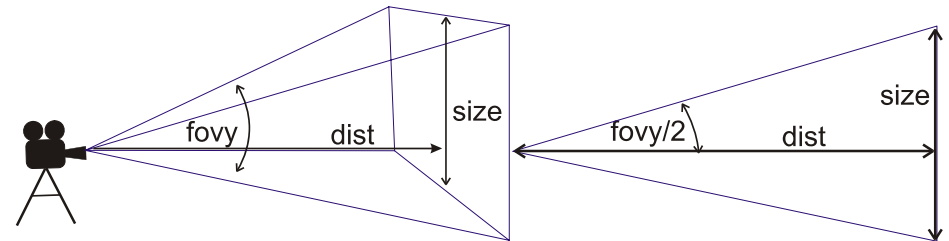
37



gluPerspective

Could we set fovy from distance and size of an object?

```
GLdouble FOVY(GLdouble size, GLdouble dist){
    GLdouble theta;
    theta = 2.0*atan2(size/2.0, dist);
    return(180.0*theta/π);
}
```



(C) Bedrich Benes

38



How to get image in double the maximal resolution?

- 1) Draw it four times setting the projections as depicted
- 2) Save the images
- 3) Glue them in Photoshop



x=[-1, 0] y=[0, 1]	x=[0, 1] y=[0, 1]
x=[-1, 0] y=[-1, 0]	x=[0, 1] y=[-1, 0]

(C) Bedrich Benes

39



A very big image:

code example:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1,0,0,1, 1,10); //upper left
Render();
glLoadIdentity();
glFrustum(0,1,0,1, 1,10); //upper right
Render();
glLoadIdentity();
glFrustum(-1,0,-1,0, 1,10); //lower left
Render();
glLoadIdentity();
glFrustum(0,1,-1,0, 1,10); //lower right
Render();
```

(C) Bedrich Benes

40



Walkthrough

Callback of the special keyboard keys

```
switch(a){
    case GLUT_KEY_LEFT : alpha+=3;break; //rotate
    case GLUT_KEY_RIGHT : alpha-=3;break;
    case GLUT_KEY_DOWN : { //go back
cam[0] -= (GLfloat)sin(alpha*3.14/180)*K;
cam[2] -= (GLfloat)cos(alpha*3.14/180)*K;
break;}
    case GLUT_KEY_UP :{ //go ahead
cam[0] += (GLfloat)sin(alpha*3.14/180)*K;
cam[2] += (GLfloat)cos(alpha*3.14/180)*K;
break;}
}
glutPostRedisplay();
```



Walkthrough

Display callback

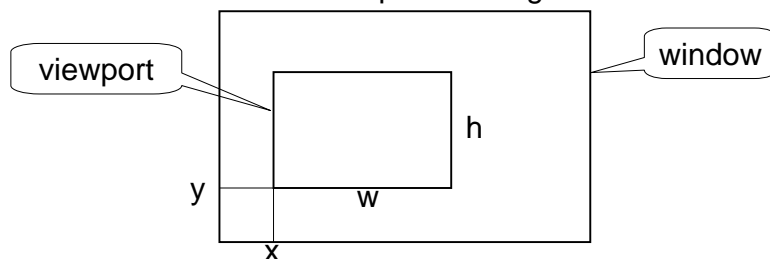
```
void Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();//reset modelview
    glRotatef(360.0f-alpha,0,1,0);//rotate everything
    glTranslatef(cam[0],cam[1],cam[2]);//translate all
    RenderObjects();//render it
    glutSwapBuffers();
}
```



Viewport is the part of the window that is used for rendering

```
void glViewport(GLint x,GLint y,
               GLsizei w, GLsizei h)
```

- x and y specifies the lower left corner
- w and h are size of the viewport rectangle



Viewport is usually set in the reshape callback

```
void Resize(int w, int h){
    glViewport(0,0,w,h);
    ...
}

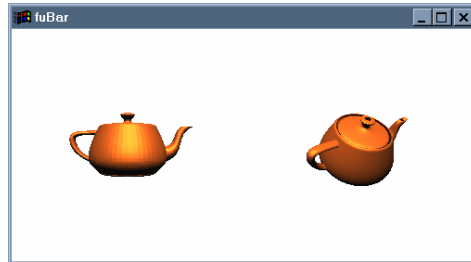
int main(int argc, char **argv){
    glutInit(&argc, argv);
    ...

    glutDisplayFunc(Display);
    glutReshapeFunc(Resize);
    ...
}
```



Example:
rendering into two viewports

```
int x = 600, y = 300;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, x/(2.0*y), 1, 10.0);//!!!
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0,0,x/2,y);
glutSolidTeapot(0.8);
glViewport(x/2,0,x/2,y);
glRotatef(60, 1, 1, 0);
glutSolidTeapot(0.8);
```



User can define own clipping planes

```
void glClipPlane(GLenum plane,
                 const GLdouble *eqn)
```

plane is GL_CLIP_PLANE0,..., GL_CLIP_PLANE5

*eqn is pointer to array of four numbers A,B,C,D

and $Ax+By+Cz+D=0$ specifies the plane

points (x y z w) satisfying

$(A \ B \ C \ D)M^{-1}(x \ y \ z \ w)^T \geq 0$

are not rendered (M is the current modelview matrix)

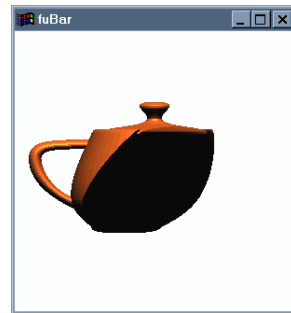
plane must be enabled e.g., glEnable(GL_CLIP_PLANE1)



Example:

```
GLdouble eqn[]={-1.0,1.0,-1.0,0.0};
```

```
glLoadIdentity();
glTranslatef(0,0,-5);
glClipPlane(GL_CLIP_PLANE0,eqn);
glEnable(GL_CLIP_PLANE0);
glutSolidTeapot(1.5);
```



GLUT provides tools for stereo viewing,

sometimes we need better control over this

- there is no interface in OpenGL for stereo glasses
- OpenGL supports several rendering buffers
- in order to render frame in stereo:
 - the display must support this
 - left/right eye must be rendered in left/right back buffer
 - back buffers must be displayed properly



OpenGL supports so called rendering buffers

switching the rendering buffers

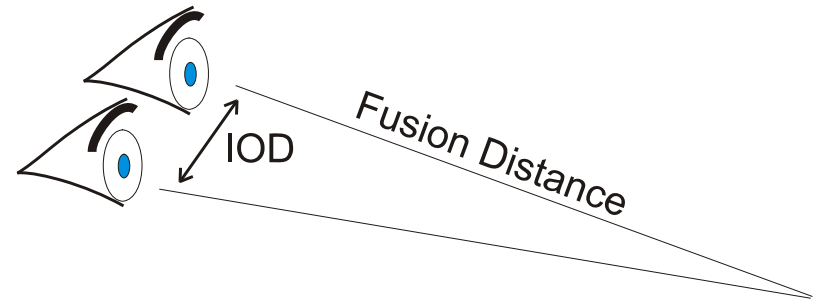
```
void glDrawBuffer(GLenum mode)
void glReadBuffer(GLenum mode)
```

mode is GL_NONE, GL_FRONT_LEFT,
GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT,
GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT,
GL_FRONT_AND_BACK, GL_AUXi



Two values must be specified

- *Interocular distance* (IOD)
- *Fusion distance*



program pseudocode:

```
//left back buffer
glPushMatrix();glDrawBuffer(GL_BACK_LEFT);
gluLookAt(  -IOD/2.0, 0.0, EYE_BACK, //position
           0.0, 0.0, 0.0, //direction
           0.0, 1.0, 0.0); //up vector
RenderScene();

//right back buffer
glDrawBuffer(GL_BACK_RIGHT);
gluLookAt(  IOD/2.0, 0.0, EYE_BACK, //position
           0.0, 0.0, 0.0, //direction
           0.0, 1.0, 0.0); //up vector
RenderScene();
glutSwapBuffers(); //swap the buffers
```



- Coordinate transforms
 - Modeling transforms
 - Viewing transforms
 - Projections
 - Special tricks
- (big images, stereo viewing, clipping planes)



- Jackie Neider, Tom Davis, Mason Woo
OpenGL Programming Guide,
 Addison-Wesley Publication Company
 ON LINE at <http://www.opengl.org.ru/docs/>
- www.opengl.org/developers/code/tutorials.html
- SIGGRAPH 2001
An Interactive Introduction To OpenGL Programming
www.opengl.org/developers/code/s2001/index.html
- SIGGRAPH '99
Lighting and Shading Techniques for Interactive Applications
www.opengl.org/developers/code/sig99/index.html