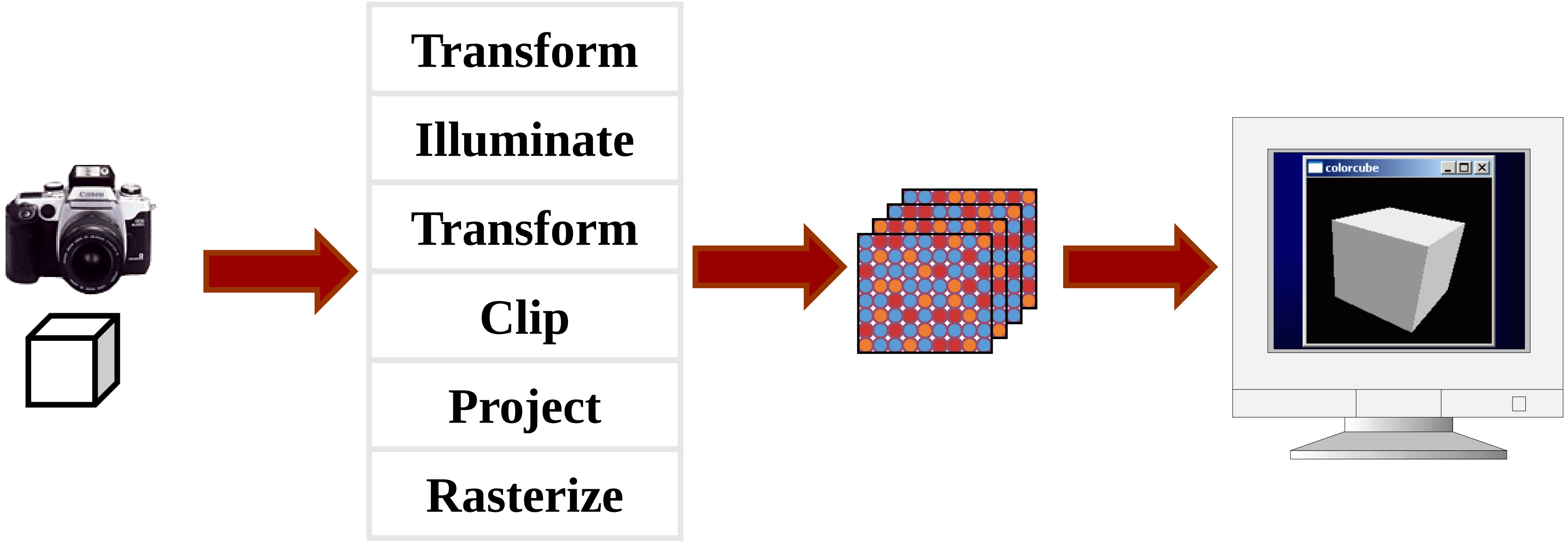
2
3D Viewing pipeline

3D rendering pipeline



Model and Camera Parameters

Viewing Pipeline

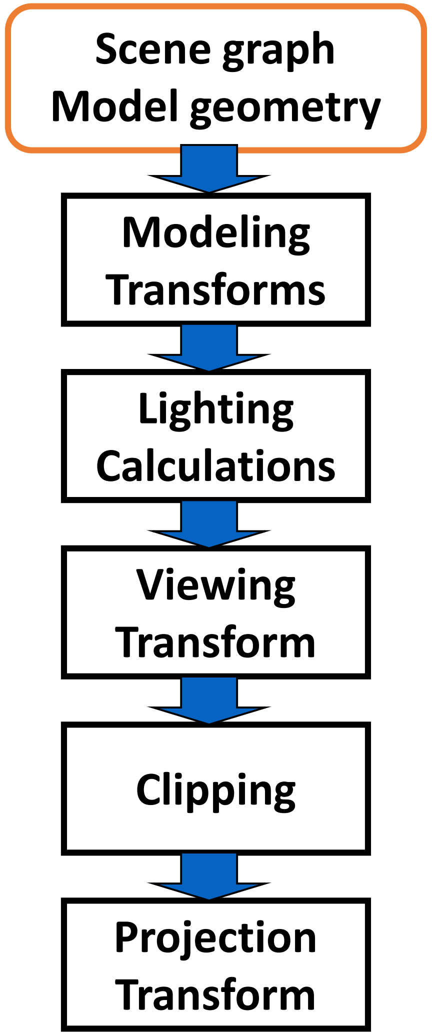
Framebuffer

Display

3D viewing pipeline

- Goal: map 3D locations, represented by (x,y,z) coordinates, to coordinates in the screen, expressed in units of pixel.
- Depends on various parameters: camera position, camera orientation, type of projection, field of view, and resolution of the screen.
- Most graphics systems do this with the following transformations:
  - Camera transformation (or eye transformation): places the camera at the origin
  - Projection transformation: projects points from camera space to normalized device coordinates (i.e. points fall in the range -1 to 1)
  - Viewport transformation (or windowing transformation): maps the image rectangle (in the range -1 to 1) to the desired rectangle in pixel coordinates

3D viewing pipeline



Result:

- All vertices of scene in 3-D "world" coordinate system
- Vertices shaded according to lighting model
- Scene vertices in 3-D "view" or "camera" coordinate system
- Exactly those vertices & portions of polygons in view frustum
- 2-D screen coordinates of clipped vertices

Modeling transformation

- Bring the object from its local coordinate system to the world coordinate system
- Scale, move, rotate objects or their parts w.r.t. each other
- Simple modeling transformation in OpenGL:
  - **glTranslatef(*GLfloat x*, *GLfloat y*, *GLfloat z*);**
    Moves an object by the given *x*, *y* and *z* values.
  - **glRotatef(*GLfloat angle*, *GLfloat x*, *GLfloat y*, *GLfloat z*);**
    Rotates an object counterclockwise about the vector from the origin through the point (*x*, *y, z*). The *angle* is in degrees.
  - **glScalef(*GLfloat x*, *GLfloat y*, *GLfloat z*);**
    Scale each *x*, *y* and *z* coordinate of every point in the object by the corresponding argument *x*, *y* or *z*.

Transformation order

Transformation order matters.

The transformations are right-concatenated. Meaning that the following code:

```
glTranslatef(1.f,0.f,0.f);
glRotatef(45.f,0.f,1.f,0.f);
drawModel();
```

First rotate by 45 degrees about the Y (0,1,0) vector, and then translate by 1 unit along X.

Saving/restoring transformations

Transformations are concatenated to the right (first to be applied is the last called).

Sometimes we may want to save the current transformation state and restore it later.

glPushMatrix(): to save the current transformation state.

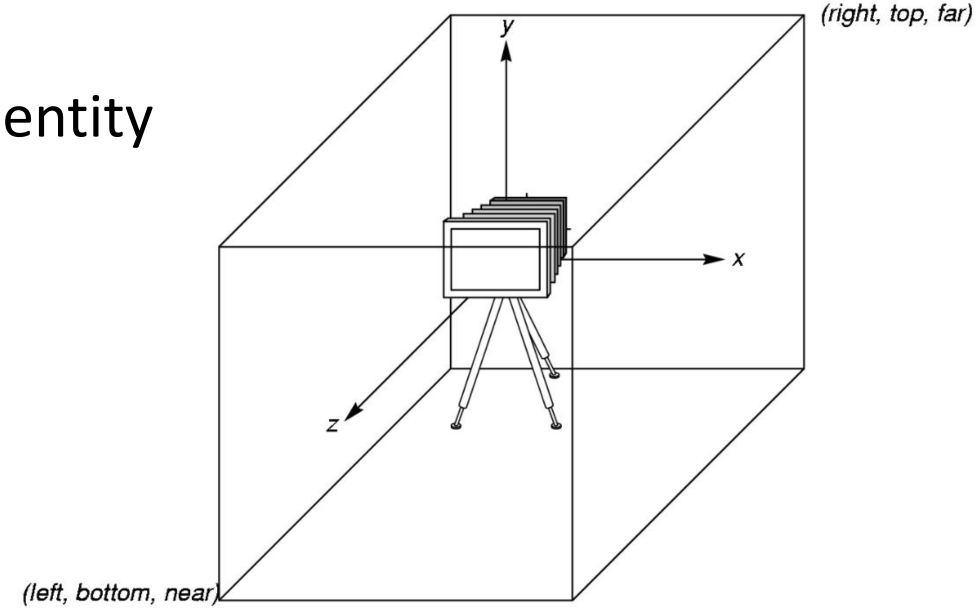
glPopMatrix(): to restore the last save transformation state.

Saving/restoring transformations

```
glPushMatrix(); // save the current state of transformation A
glTranslatef(1.f,0.f,0.f); // apply a couple of transformations
glRotatef(45.f,0.f,1.f,0.f);
glTranslatef(0.f,1.f,0.f);
drawModel();
glPopMatrix(); // restore to the transformation state we were in A
```

OpenGL camera

- In OpenGL, initially the object and camera frames are the same
  - Default model-view transformation is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection transformation is an identity



Moving the camera frame

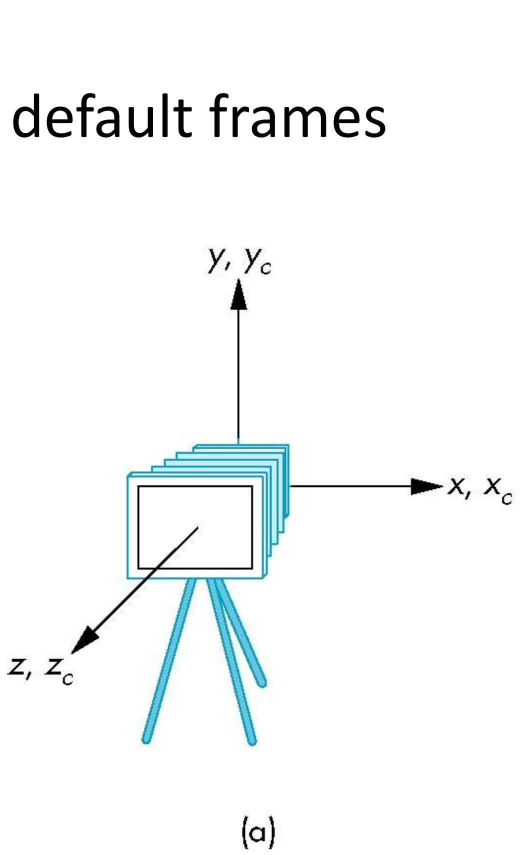
- Once objects are positioned in world space, we need to position the scene w.r.t. the camera
- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined by the model-view transformation
  - Want a translation (gl**`Translatef(0.0f,0.0f,-d);`**)
  - **`d > 0`**

Moving the camera frame from the origin

frames after translation by –d

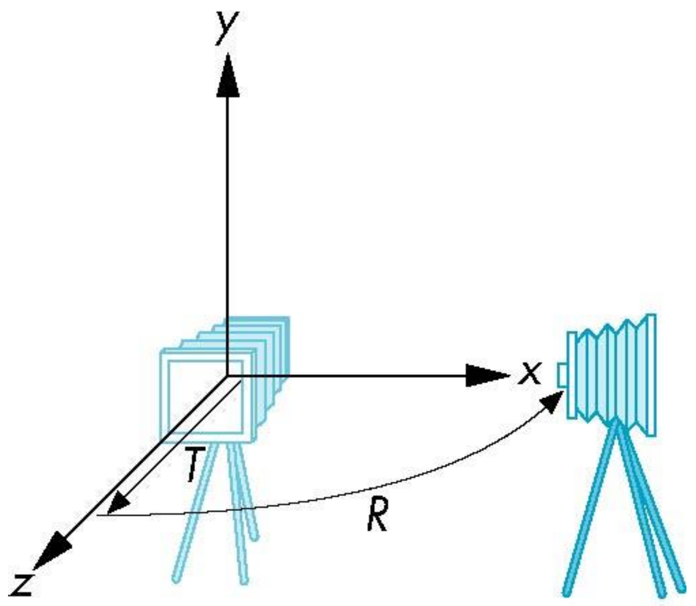
d > 0

default frames



Moving the camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - Move the camera away from origin
  - Then rotate it
  - Or: symmetrically, we can rotate the scene then move it away from the camera



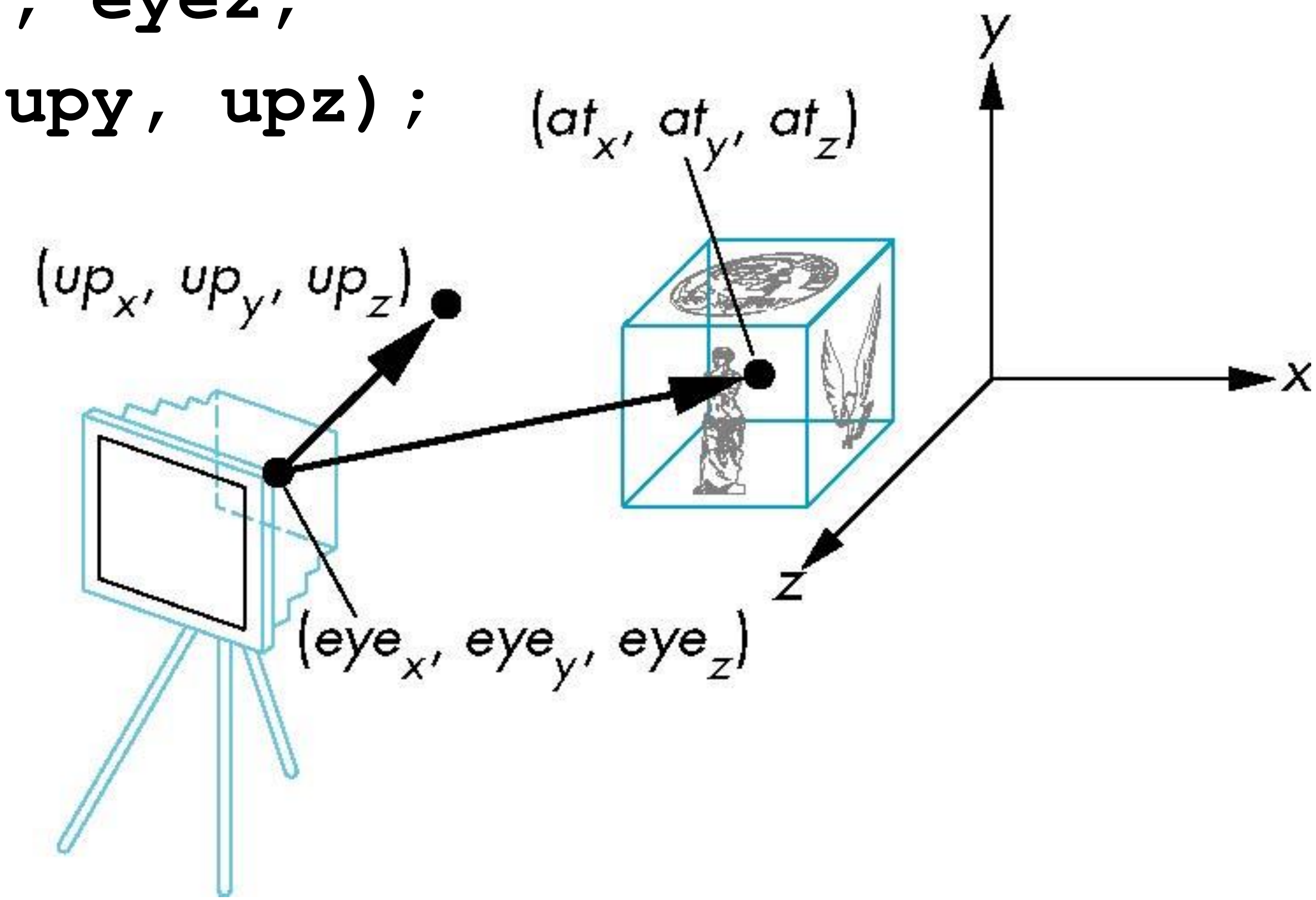
LookAt function

- The GLU library contains the function gluLookAt to form the required model-view transformation through a simple interface
- Note the need for setting an up direction
- Can be obtained with modeling transformations
- Example: isometric view of cube aligned with axes

LookAt

`gluLookAt(eyex, eyey, eyez,`

`atx, aty, atz, upx, upy, upz);`



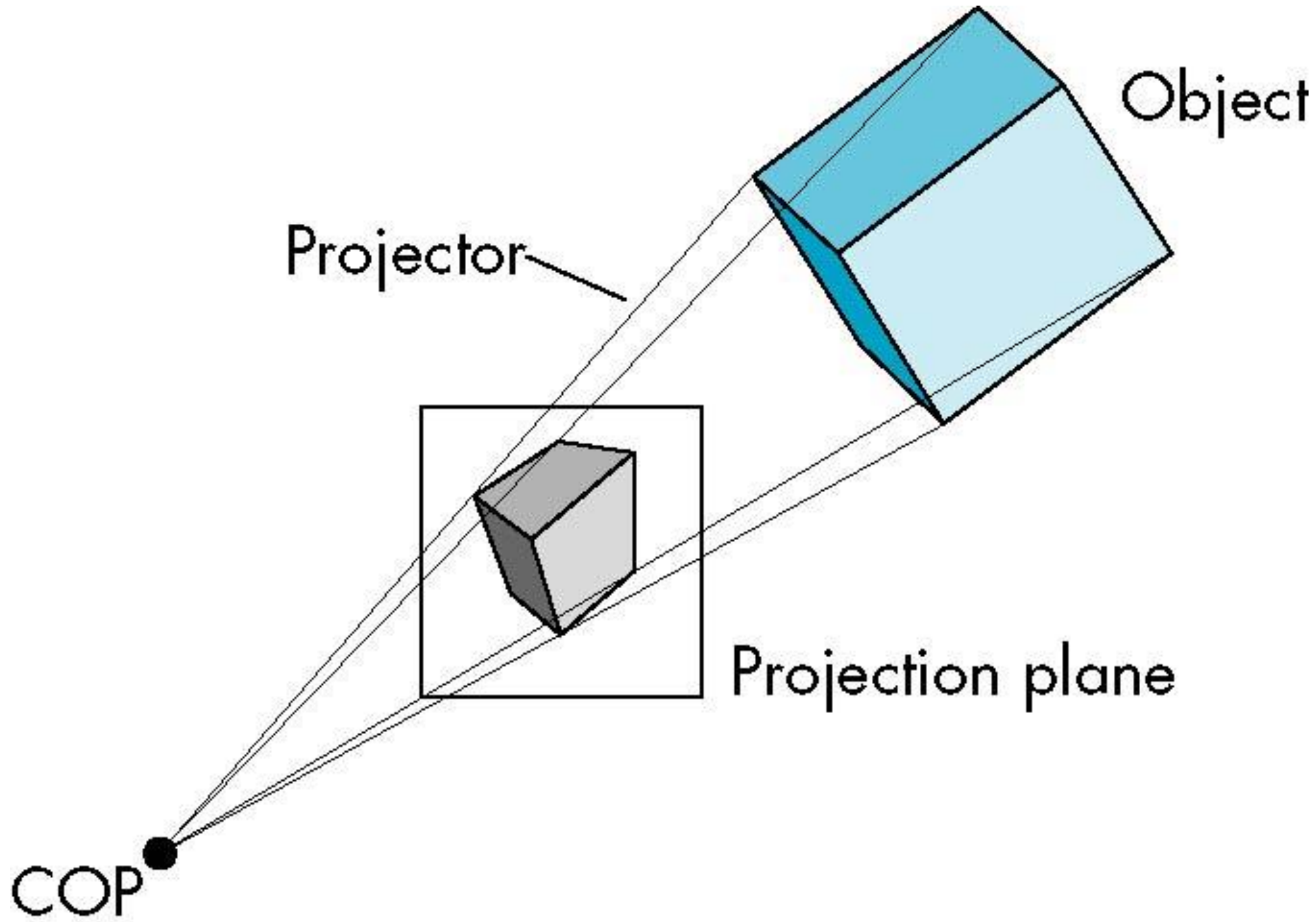
Projections

- Standard projections project onto a plane
- Projectors are lines that either
  - converge at a center of projection
  - are parallel
- Such projections preserve lines
  - but not necessarily angles
- Nonplanar projections are needed for applications such as map construction

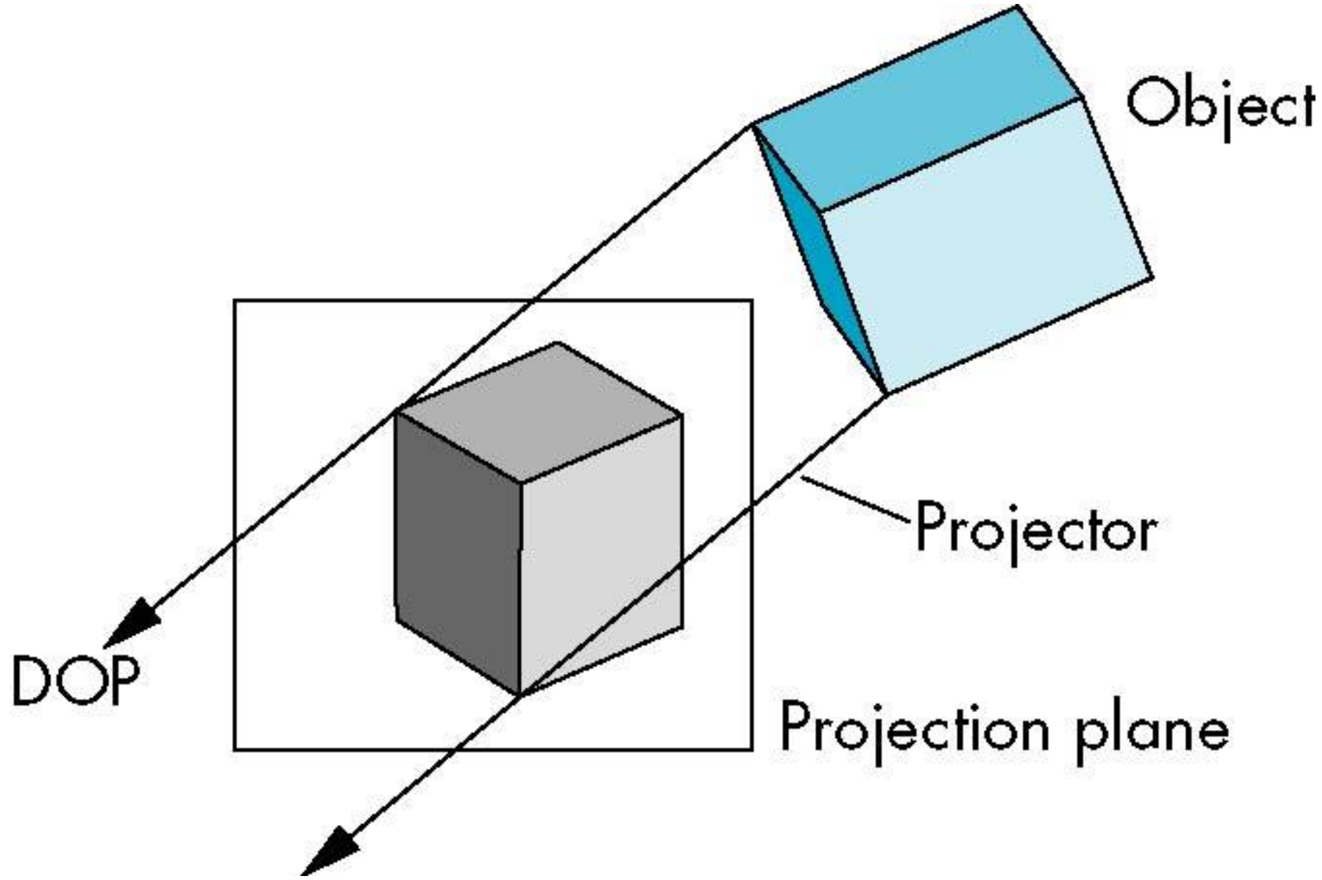
Perspective vs parallel

- Computer graphics treats all projections the same and implements them with a single pipeline
- Classical viewing developed different techniques for drawing each type of projection
- Fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the limit of perspective viewing

Perspective projection

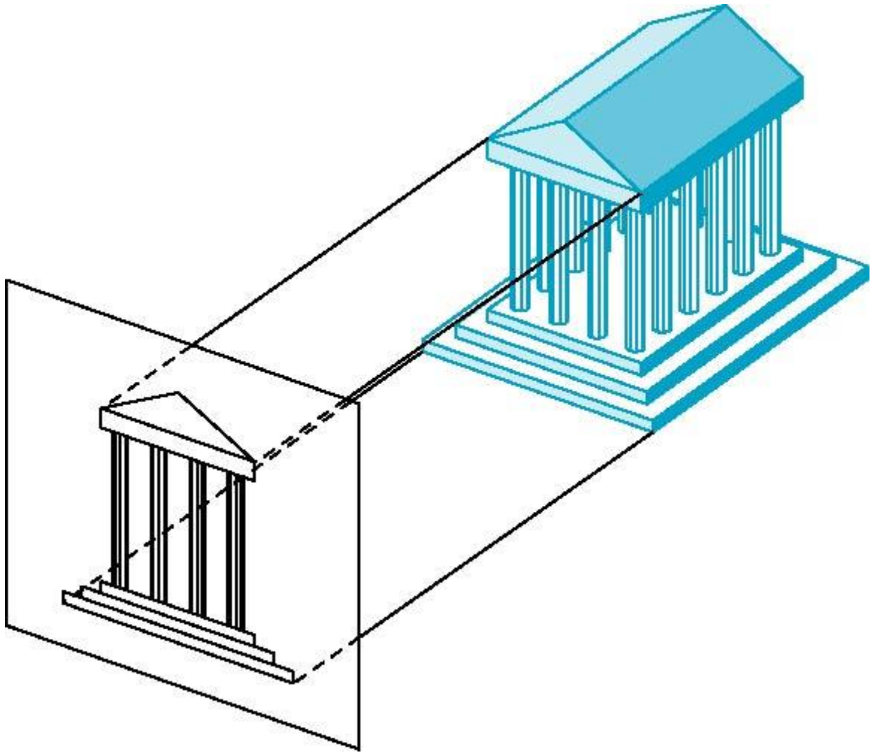


Parallel projection



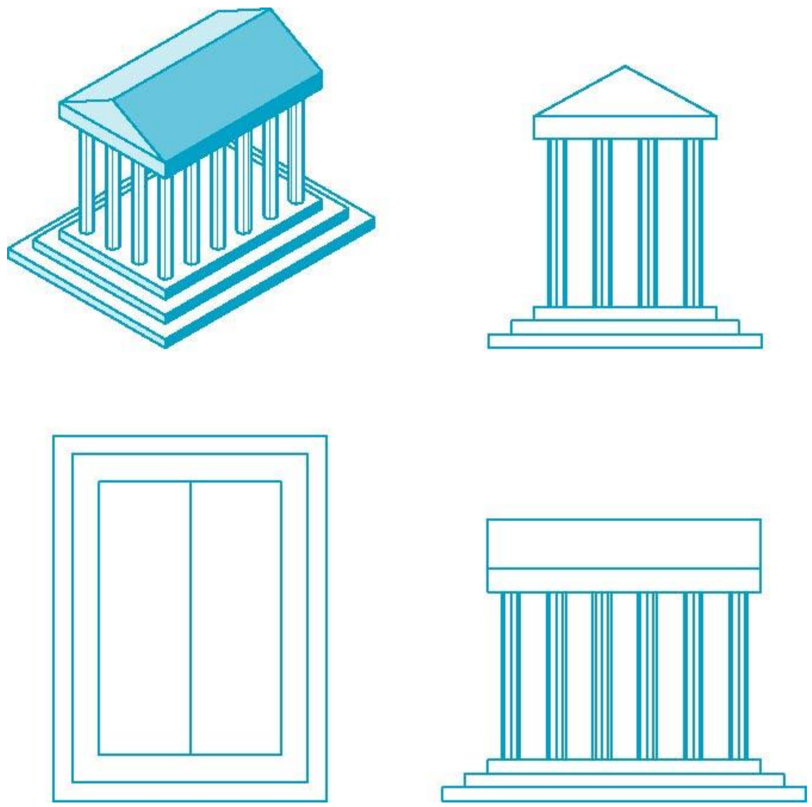
Orthographic projection

- Projection is parallel to one of the principal faces
- Projectors are orthogonal to the projection surface



Multi-view orthographic projection

- Typical of CAD systems
- One oblique (or sometimes perspective) view
- And three orthographic views corresponding to front, side and top

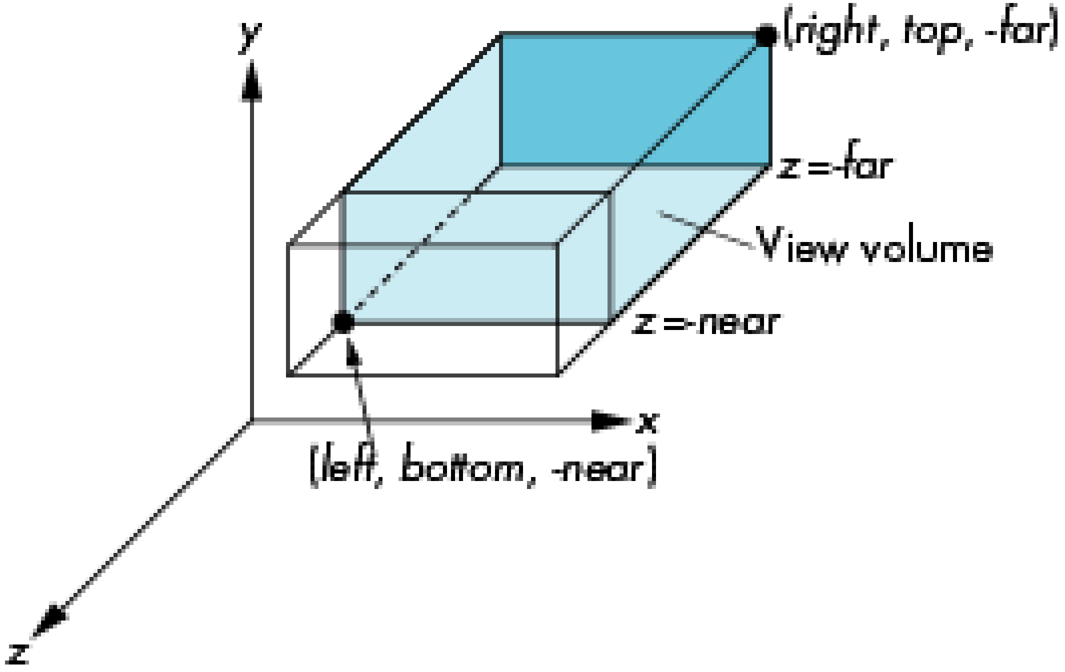


Orthographic projection

- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals
- Cannot see what object really looks like because many surfaces hidden from view
  - Often we add the isometric

OpenGL orthographic projection

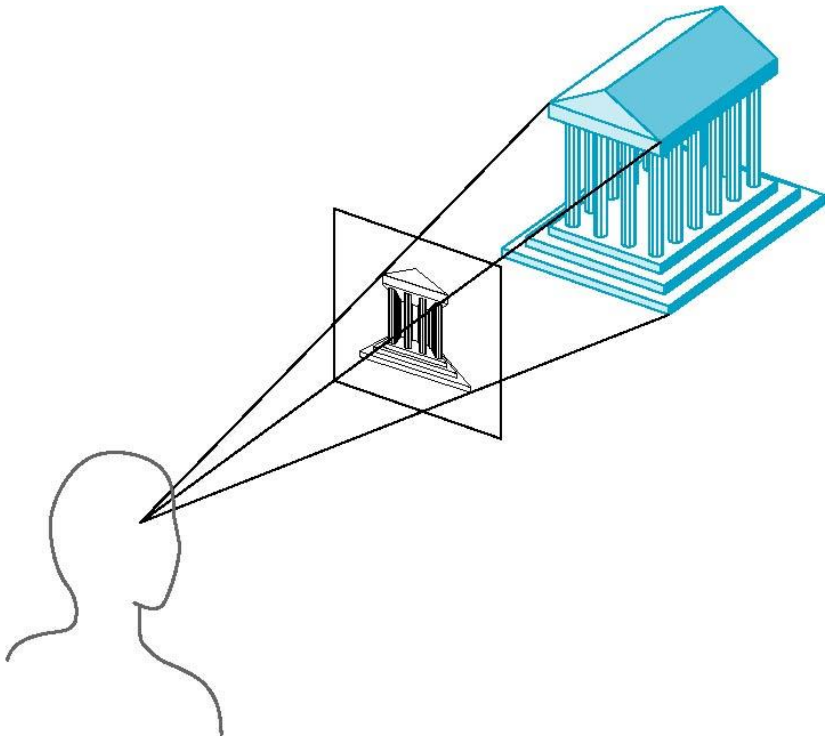
`glOrtho(left,right,bottom,top,near,far)`



`near` and **`far`** measured <u>from</u> camera

Perspective projection

- Projectors converge at center of projection

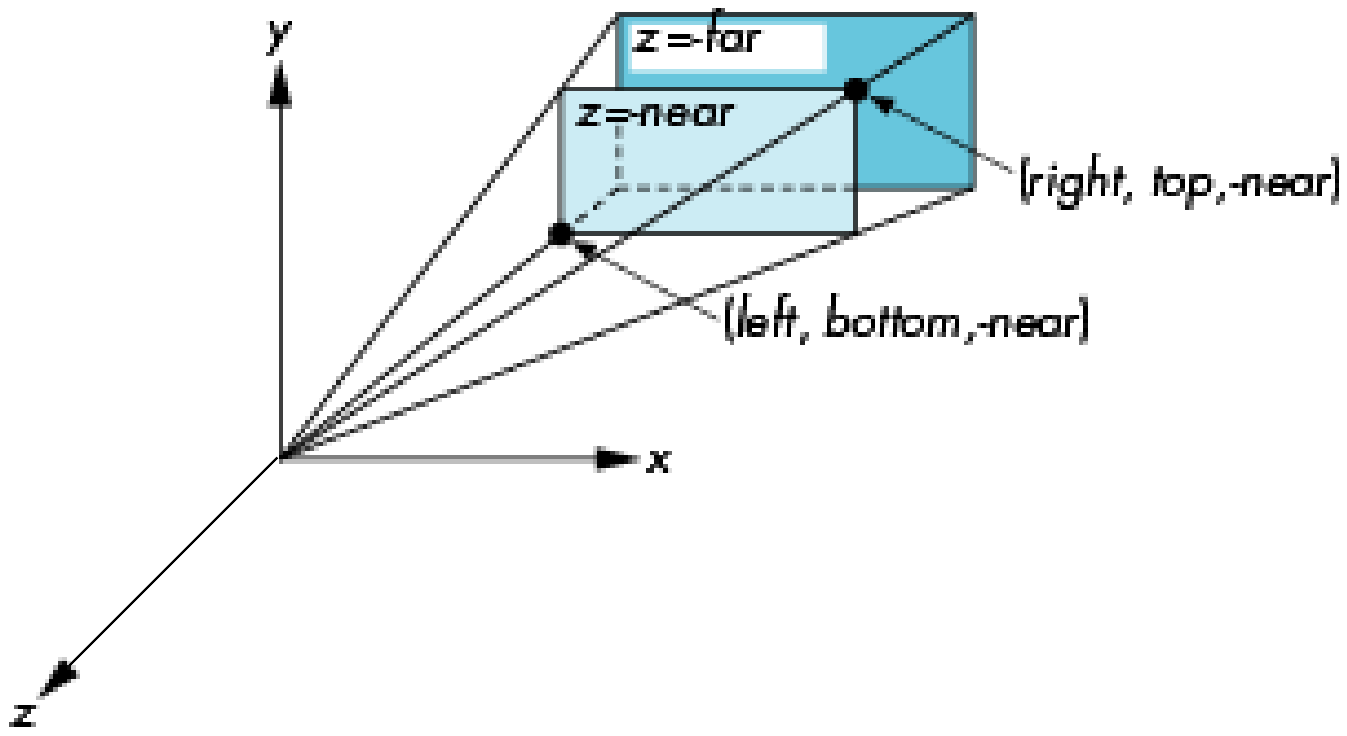


Perspective projection

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*non-uniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

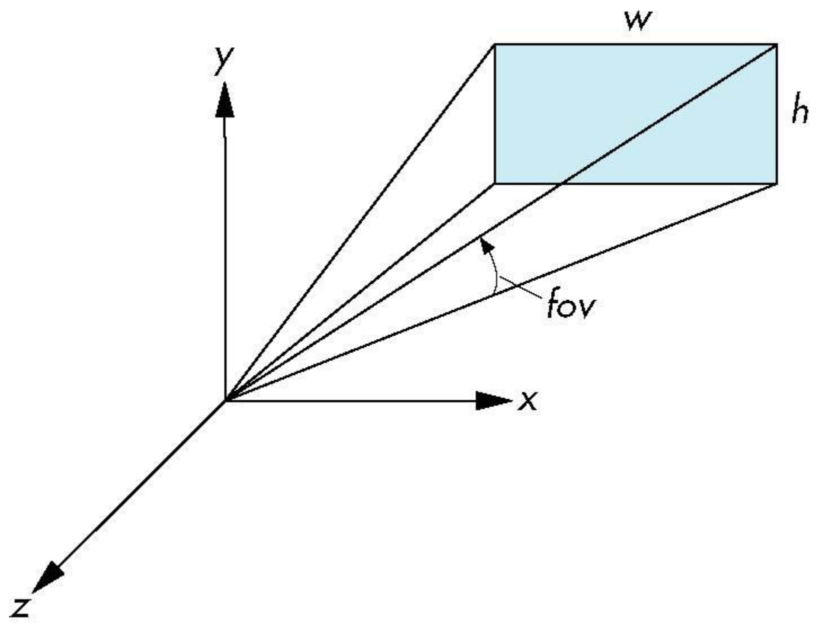
OpenGL perspective projection

glFrustum(left,right,bottom,top,near,far)



Using Field of View

- With `Frustum` it is often difficult to get the desired view
- `gluPerpective(fovy, aspect, near, far)` often provides a better interface



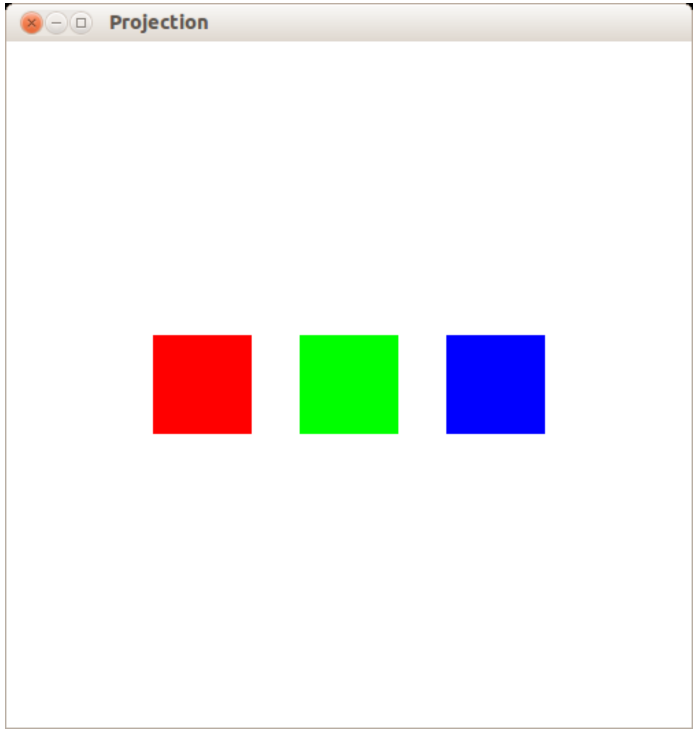
front plane

`aspect = w/h`

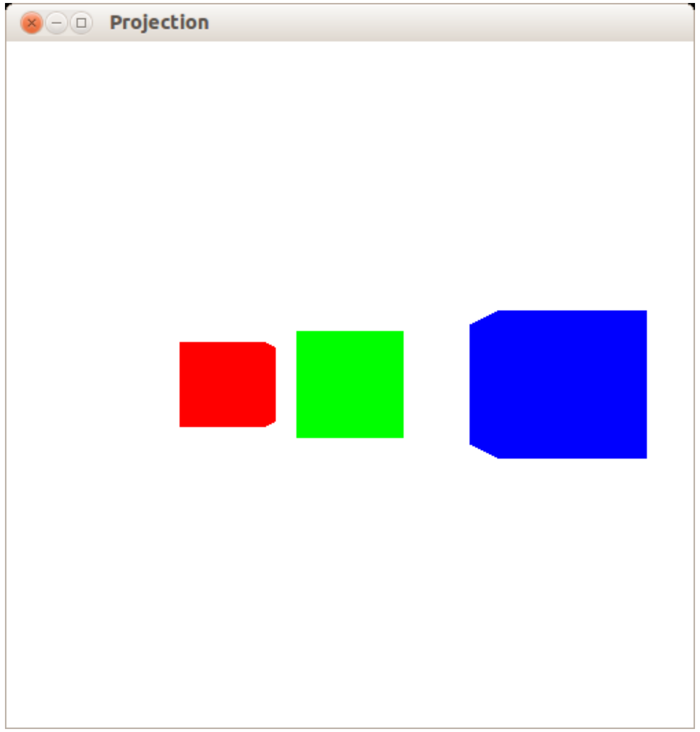
Orthographic projection vs perspective projection

- Consider the results of applying an orthographic projection or a perspective projection to identical cubes but at different distance along the z-axis from the camera

Orthographic projection

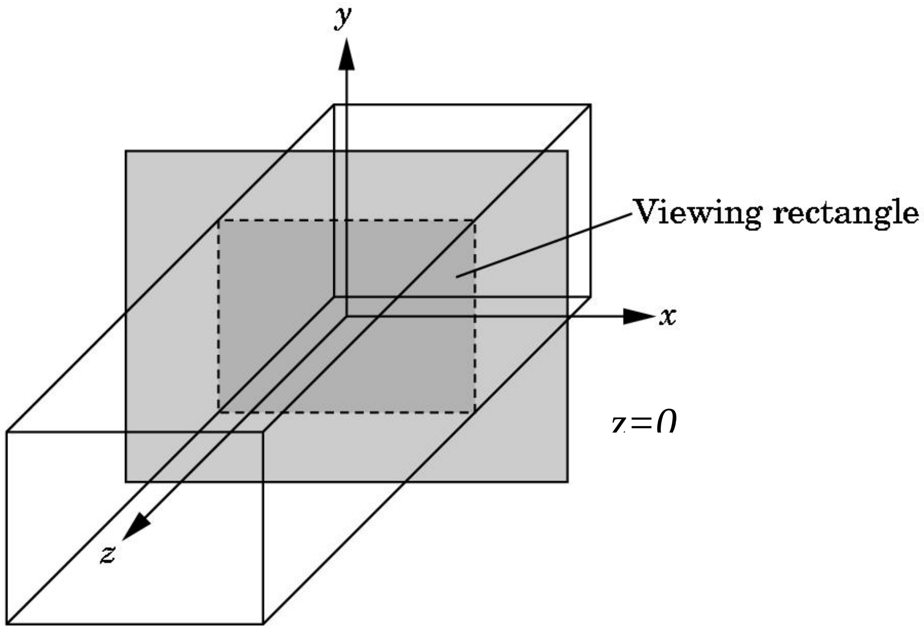


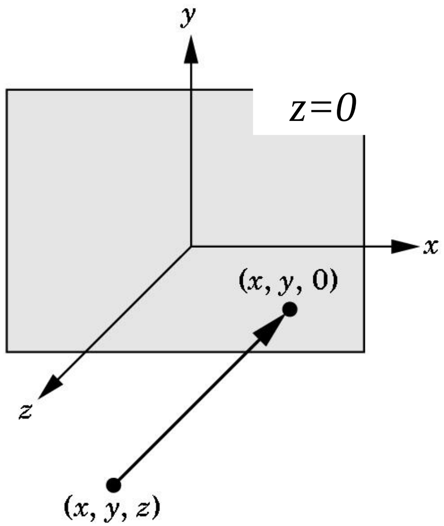
Perspective projection



Default projection

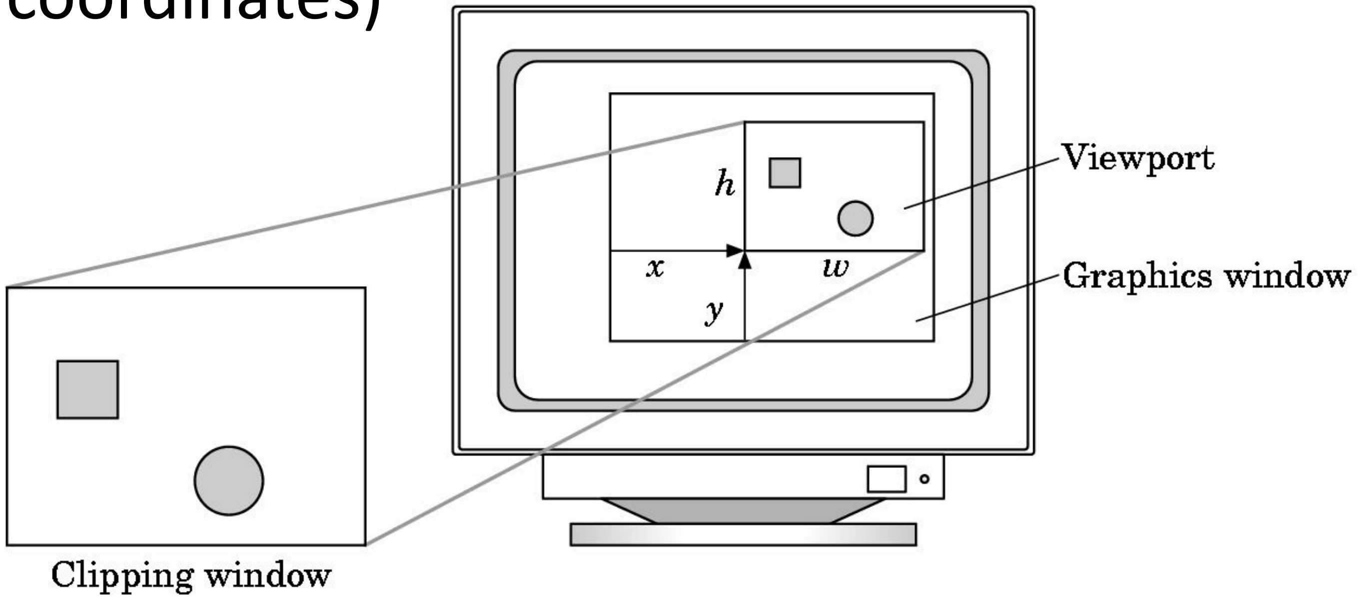
- In the default orthographic view, points are projected forward along the z axis onto the plane $z=0$





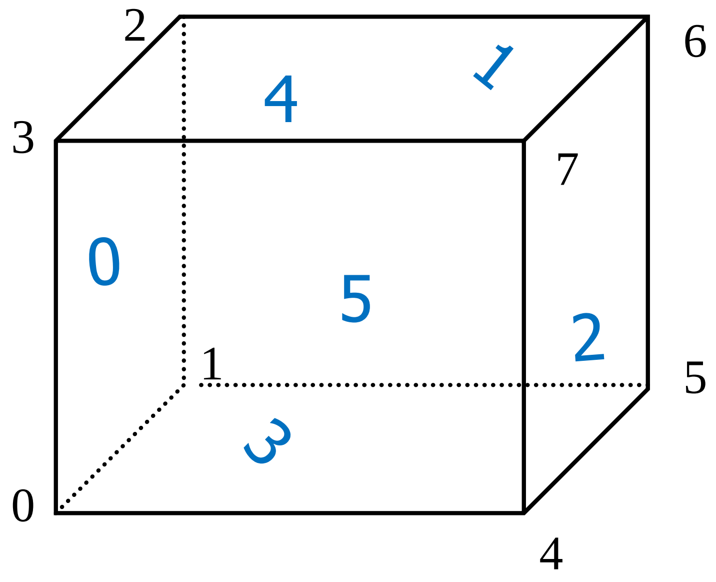
Viewports

- Convert from normalized device coordinates to screen coordinates (in pixel units):
  `glViewport(x,y,w,h)`
- Do not have use the entire window for the image:
- Values in pixels (window coordinates)



Example: 3D cube

```
#include <GL/glut.h>
#include <stdlib.h>
GLint faces[6][4] = {{0, 1, 2, 3}, {3, 2, 6, 7},
  {7, 6, 5, 4}, {4, 5, 1, 0},
  {5, 6, 2, 1}, {7, 4, 0, 3} };
GLfloat v[8][3];
```



Vertex specification

```
void init(void)
{
  glClearColor (0.0, 0.0, 0.0, 0.0);
  v[0][0] = v[1][0] = v[2][0] = v[3][0] = -1;
  v[4][0] = v[5][0] = v[6][0] = v[7][0] =  1;
  v[0][1] = v[1][1] = v[4][1] = v[5][1] = -1;
  v[2][1] = v[3][1] = v[6][1] = v[7][1] =  1;
  v[0][2] = v[3][2] = v[4][2] = v[7][2] = -4;
  v[1][2] = v[2][2] = v[5][2] = v[6][2] = -6;
}

void keyboard(unsigned char key, int x, int y) {
   if(key=='q' || key=='Q') exit(0);
}
```

Cube in wireframe mode

```
void drawBox(void)
{
  int i;
  for (i = 0; i < 6; i++) {
    glBegin(GL_LINE_LOOP);
    glVertex3fv(&v[faces[i][0]][0]);
    glVertex3fv(&v[faces[i][1]][0]);
    glVertex3fv(&v[faces[i][2]][0]);
    glVertex3fv(&v[faces[i][3]][0]);
    glEnd();
  }
}
```

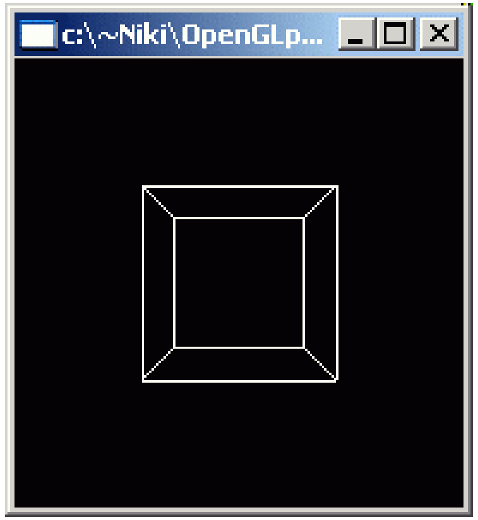
Display / reshape

```
void display(void) {
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   drawBox();
   glFlush();
}

void reshape (int w, int h){
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

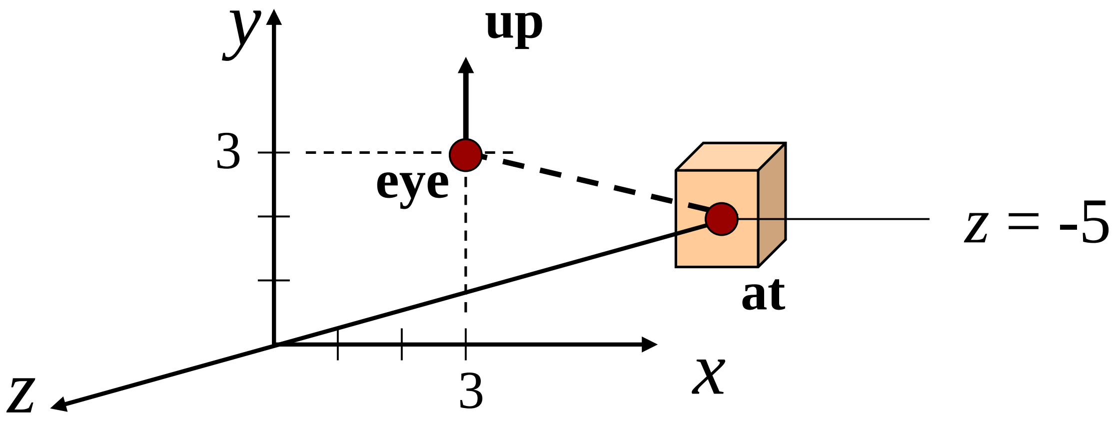
Main

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (200, 200);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```



Using gluLookAt

```
gluLookAt(eyex,eyey,eyez,atx,aty,atz,upx,upy,upz);
```



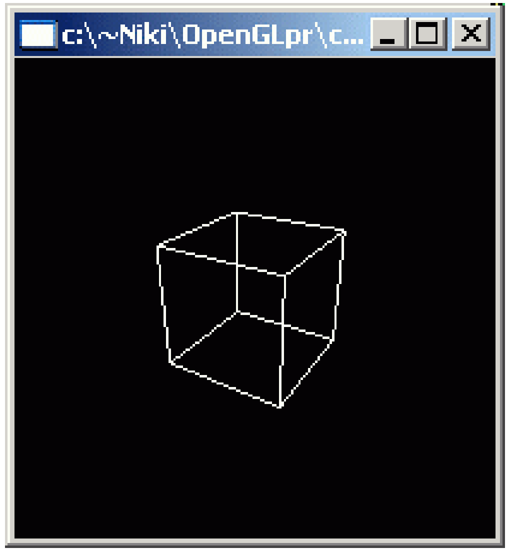

```
gluLookAt(3.,3.,0., 0.,0.,-5., 0.,1.,0.);
```

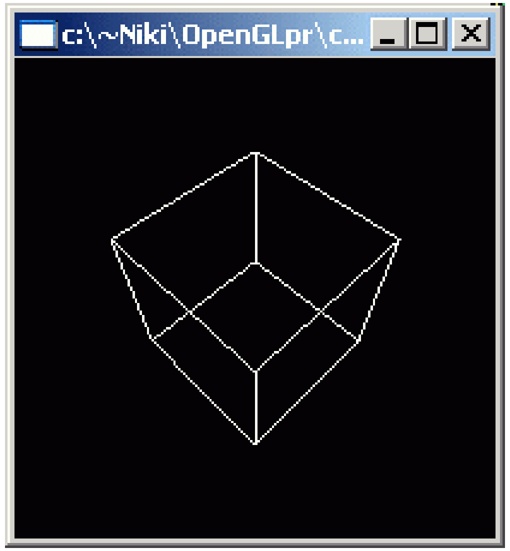
Using gluLookAt

```
void display(void){
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glLoadIdentity();
   gluLookAt(3.,3.,0., 0.,0.,-5., 0.,1.,0.);
   drawBox();
   glFlush ();
}

void reshape (int w, int h){
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,(GLfloat)w/h,1.0, 40.0);
   glMatrixMode (GL_MODELVIEW);
}
```

Results





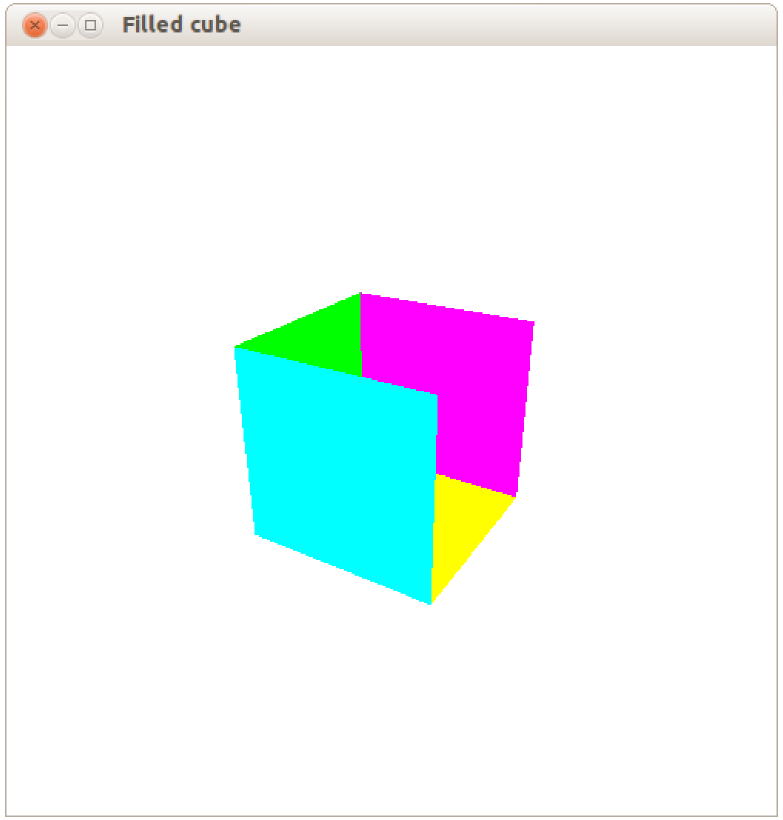

```
gluLookAt(3.0, 3.0, 0.0,
          0.0, 0.0, -5.0,
          0.0, 1.0, 0.0);
```

```
gluLookAt(2.0, 4.0, -3.0,
          0.0, 0.0, -5.0,
          0.0, 1.0, 0.0)
```

Rendering filled cube (hidden surface removal)

- Let us try to fill the cube faces by replacing: `glBegin(GL_LINE_LOOP)` by: `glBegin(GL_POLYGON)`
- and by using different colors for each face.

- The result is not satisfactory

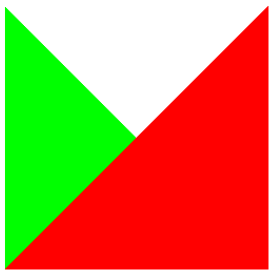


Rendering filled cube

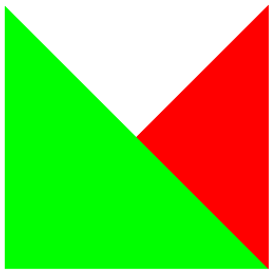
```
  // closest triangle
glColor3f(1.0f, 0.0f, 0.0f);
glBegin(GL_TRIANGLES);
		glVertex3f(0.25f, 0.25f, 0.0f);
		glVertex3f(0.75f, 0.25f, 0.0f);
		glVertex3f(0.75f, 0.75f, 0.0f);
glEnd();


// farthest triangle
glColor3f(0.0f, 1.0f, 0.0f);
glBegin(GL_TRIANGLES);
		glVertex3f(0.25f, 0.25f, -0.5f);
		glVertex3f(0.75f, 0.25f, -0.5f);
		glVertex3f(0.25f, 0.75f, -0.5f);
glEnd();
```

Expected



Obtained



Explanation

Polygons are sent to the pipeline and rendered *in order*.

In the example:

- First defined triangle is the red one
- Then the green one
- Therefore the obtained result (independently of the position of the triangles w.r.t the camera)

Rule: the last triangle to be rendered is the one that will be seen

Hidden surface removal

- This problem is called: *Hidden Surface Elimination*
- We want to see the triangles closest to the camera (and not the last defined triangles)
- Solution: use an additional buffer, called the *depth-buffer* (also sometimes z-buffer) that stores depth value for each pixel and perform tests (*depth-test*) to determine what to draw

Depth-buffer and depth-test

- *Depth-buffer* stores depth values for each pixel.
- The idea is then to stop writing a fragment if it is in the back of the current pixel. This is called the *depth-test* (sometimes also called z-buffer algorithm).

Depth-buffer and depth-test in OpenGL

To activate depth-testing:

`glEnable(GL_DEPTH_TEST)`

The corresponding `glDisable` causes depth-testing to stop.

Before doing depth-tests, we need to initialize the depth-buffer

In order to clear the depth buffer, we use `glClear` (like for the color buffer):

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Steps to use depth-test in OpenGL

1. Create a GL rendering context with depth-buffer enabled:
   `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);`
2. Clear the depth-buffer before usage:
   `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`
3. Enable the depth-test: `glEnable(GL_DEPTH_TEST);`

Example

```
static GLshort g_use_depth_buffer = 0;


void initGL() {
	glClearColor(1.0, 1.0, 1.0, 1.0);

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
```

Example (continued)

```
void handleDisplay() {
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


  if (g_use_depth_buffer) {
    glEnable(GL_DEPTH_TEST);
    printf("Depth-buffer enabled¥n");
  } else {
    printf("Depth-buffer disabled¥n");
    glDisable(GL_DEPTH_TEST);
   }

    // closest triangle
          glColor3f(1.0f, 0.0f, 0.0f);
          glBegin(GL_TRIANGLES);
          glVertex3f(0.25f, 0.25f, 0.0f);
          glVertex3f(0.75f, 0.25f, 0.0f);
          glVertex3f(0.75f, 0.75f, 0.0f);
          glEnd();

    // ...
```

```
void handleDisplay() {
    // ...

    // farthest triangle
    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_TRIANGLES);
        glVertex3f(0.25f, 0.25f, -0.5f);
        glVertex3f(0.75f, 0.25f, -0.5f);
        glVertex3f(0.25f, 0.75f, -0.5f);
    glEnd();

    glFlush();
}
```

Example (continued)

```
void
handleKeyboardEvents(unsigned char key, int x, int y)
{
  if (key == 'q' || key == 'Q') {
    exit(0);
  }

  if (key == 'z') {
    g_use_depth_buffer = 1 - g_use_depth_buffer;
    glutPostRedisplay();
    return;
   }
}
```

```
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Depth-buffer");

    initGL();

    glutDisplayFunc(handleDisplay);
    glutKeyboardFunc(handleKeyboardEvents);

    glutMainLoop();

    return EXIT_SUCCESS;
}
```

Result

- Let us return to the filled cube example, this time using the depth-buffer and a depth-test: