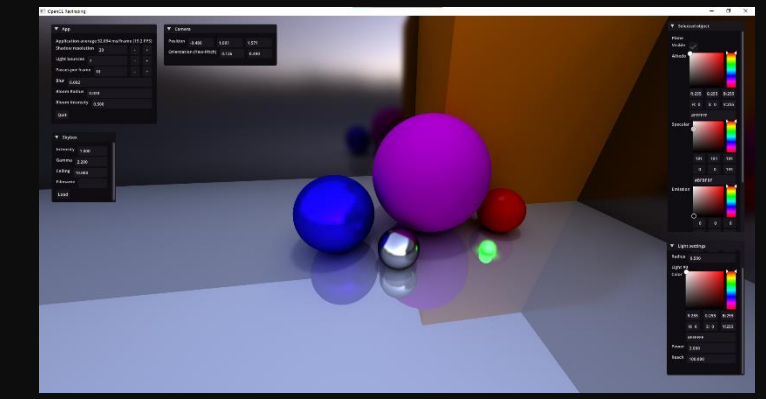
Learning OpenGL



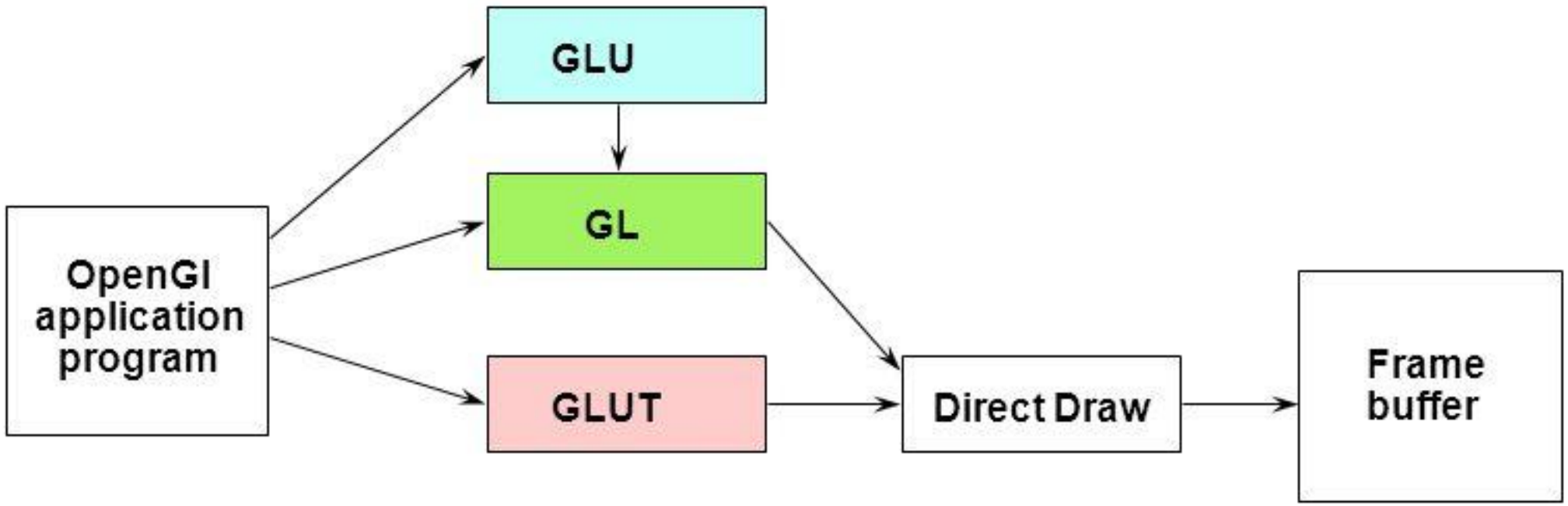
OpenGL

Release	Year
OpenGL 1.0	1992
OpenGL 2.0	2004
OpenGL 3.1	2009
OpenGL ES 2.0	2010
WebGL 1.0	2011
OpenGL 4.5	2014
Vulcan 1.0	2016
WebGL 2.0	2016

OpenGL Libraries

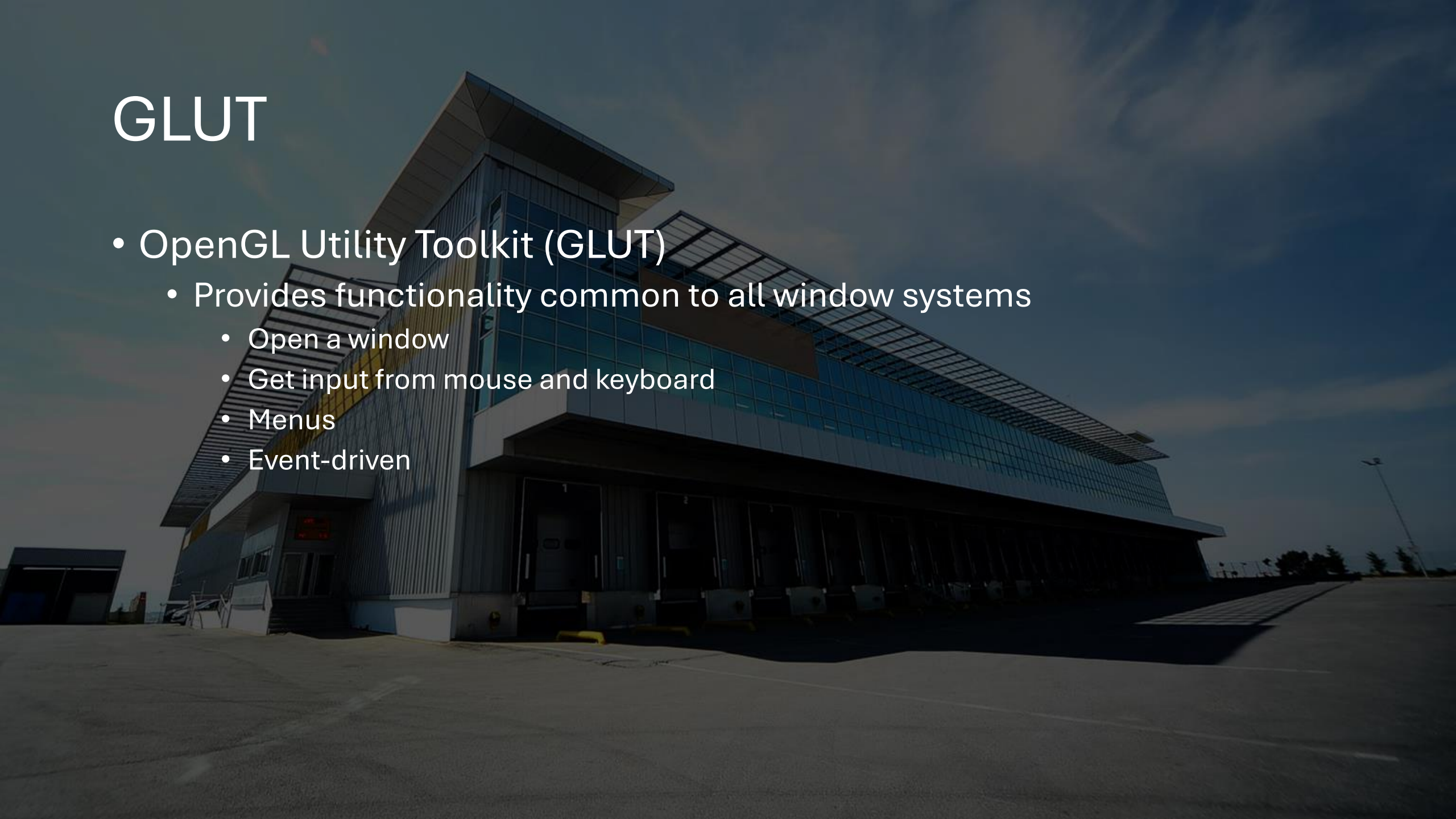
OpenGL core library	• OpenGL32 on Windows • GL on most unix/linux systems (libGL.a)
OpenGL Utility Library (GLU)	• Provides functionality in OpenGL core but avoids having to rewrite code
Links with window system	• GLX for X window systems • WGL for Windows • AGL for Macintosh

GL Library Organization Under Microsoft Windows



GLUT

- OpenGL Utility Toolkit (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven



In the Beginning …

OpenGL 1.0 was released between 1992-1993

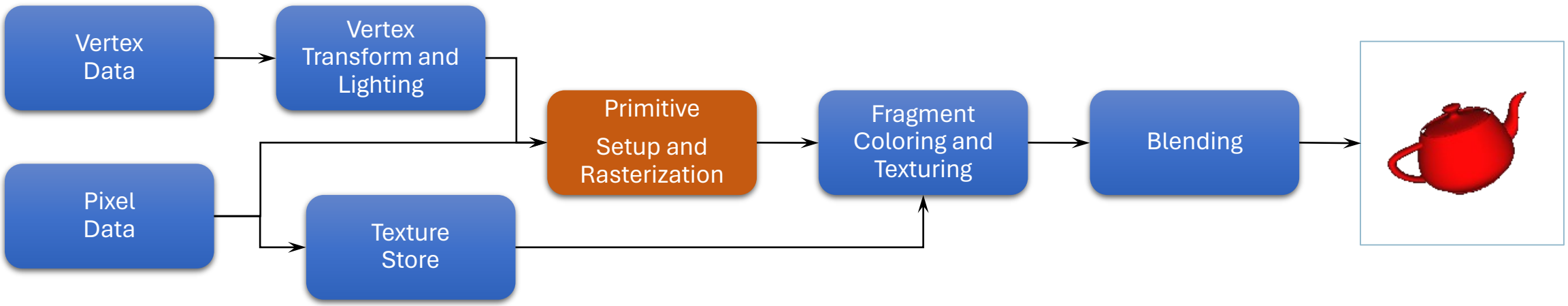
Its pipeline was entirely fixed-function

- the only operations available were fixed by the implementation

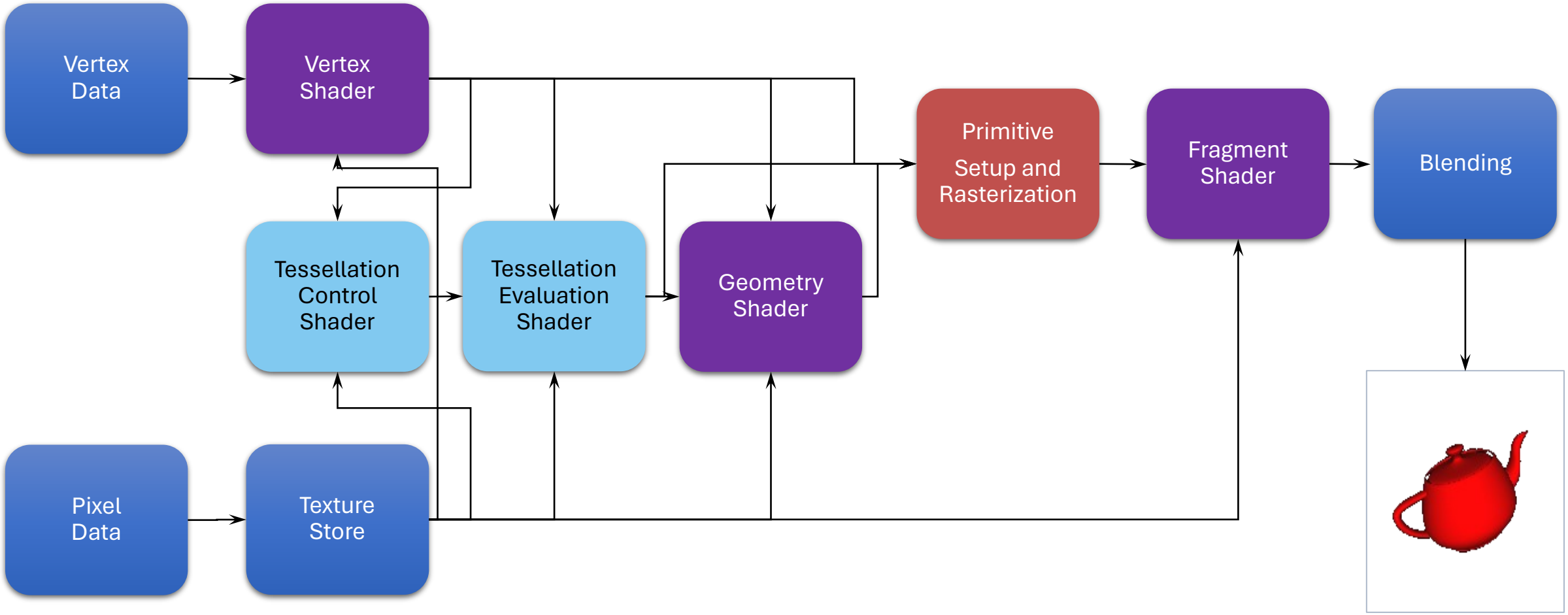
The pipeline evolved

- but remained based on fixed-function operation through OpenGL versions 1.1 through 2.0 (Sept. 2004)

OpenGL 1.0


Vertex Data
Vertex Transform and Lighting
Pixel Data
Texture Store
Primitive Setup and Rasterization
Fragment Coloring and Texturing
Blending

OpenGL 4.1


Vertex Data
Vertex Shader
Tessellation Control Shader
Tessellation Evaluation Shader
Geometry Shader
Primitive Setup and Rasterization
Fragment Shader
Blending
Pixel Data
Texture Store

OpenGL Functions

- Primitives
  - Points
  - Line Segments
  - Polygons
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (GLUT)
- Input (GLUT)
- Query

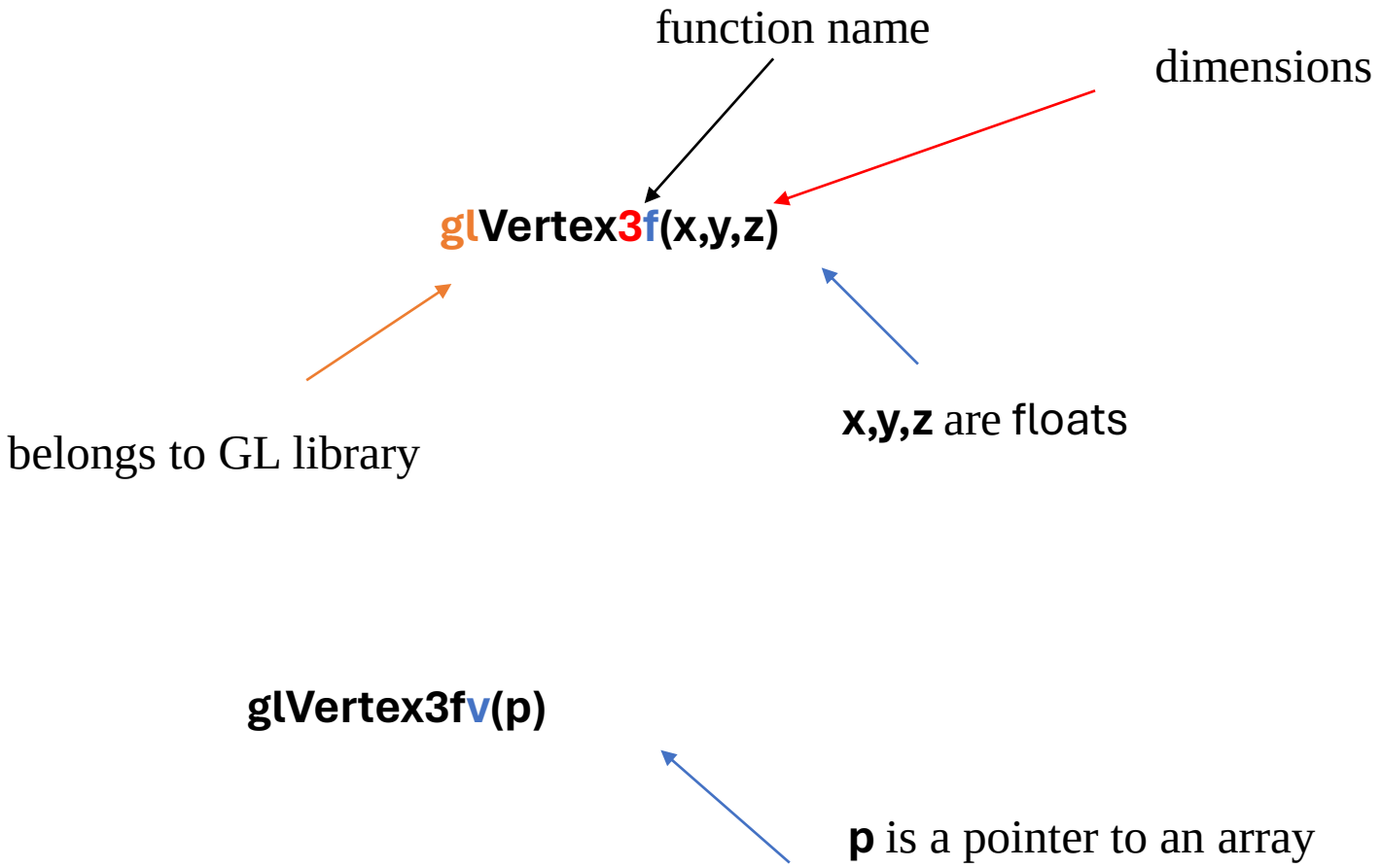
OpenGL State

OpenGL is a state machine

OpenGL functions are of two types

- Primitive generating
  - Can cause output if primitive is visible
  - How vertices are processed, and appearance of primitive are controlled by the state
- State changing
  - Transformation functions
  - Attribute functions

OpenGL function format



OpenGL #defines

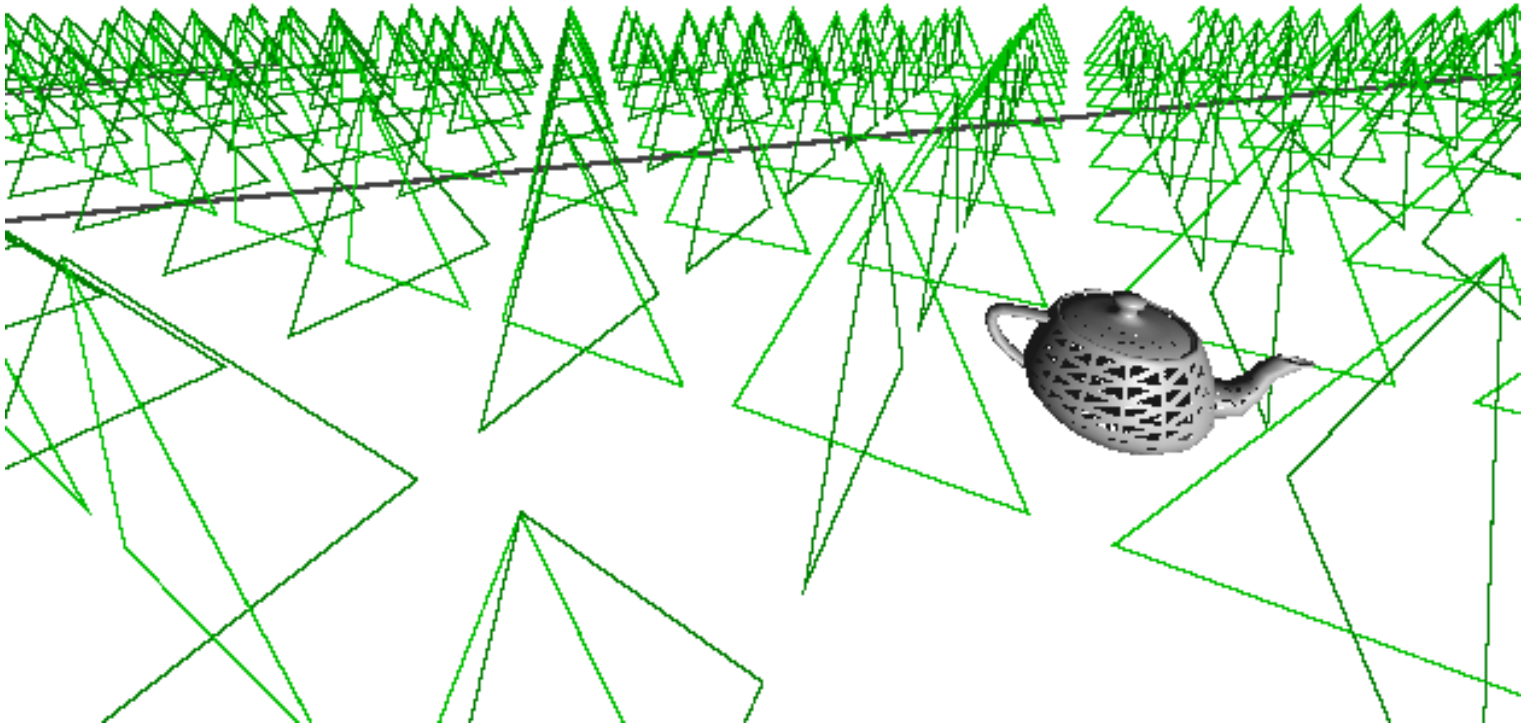
Most constants are defined in the include files **gl.h**, **glu.h** and **glut.h**

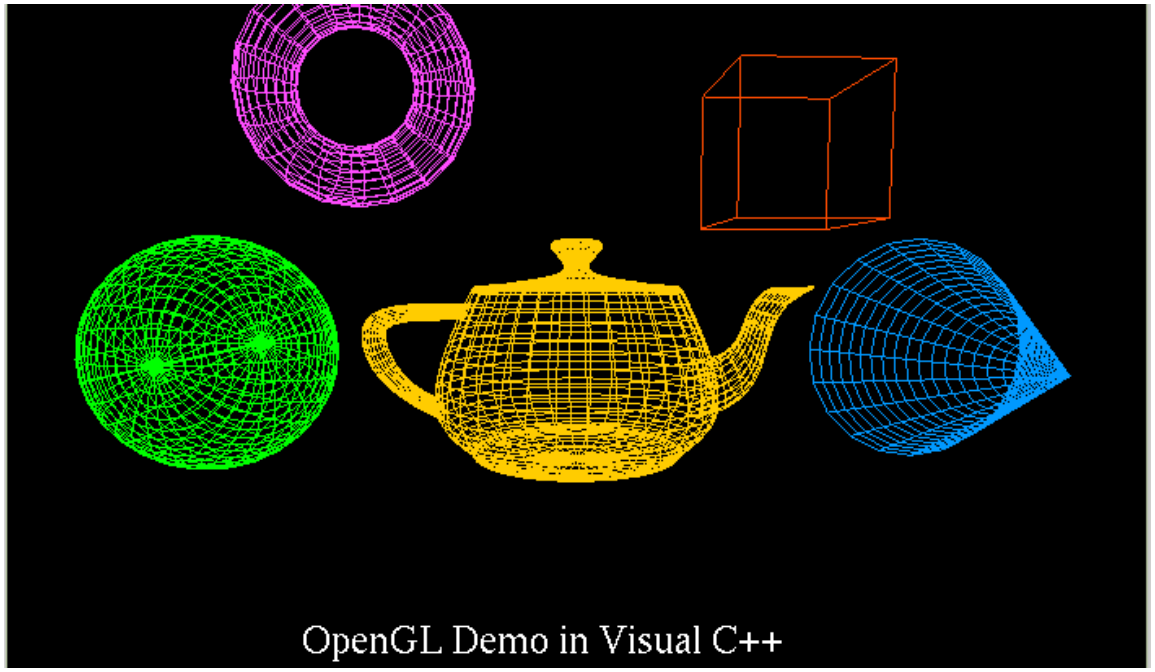
- Note **#include <GL/glut.h>** should automatically include the others
- Examples
- **glBegin(GL_POLYGON)**
- **glClear(GL_COLOR_BUFFER_BIT)**

include files also define OpenGL data types: **GLfloat**, **GLdouble**,....

glVertex

https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glVertex.xml



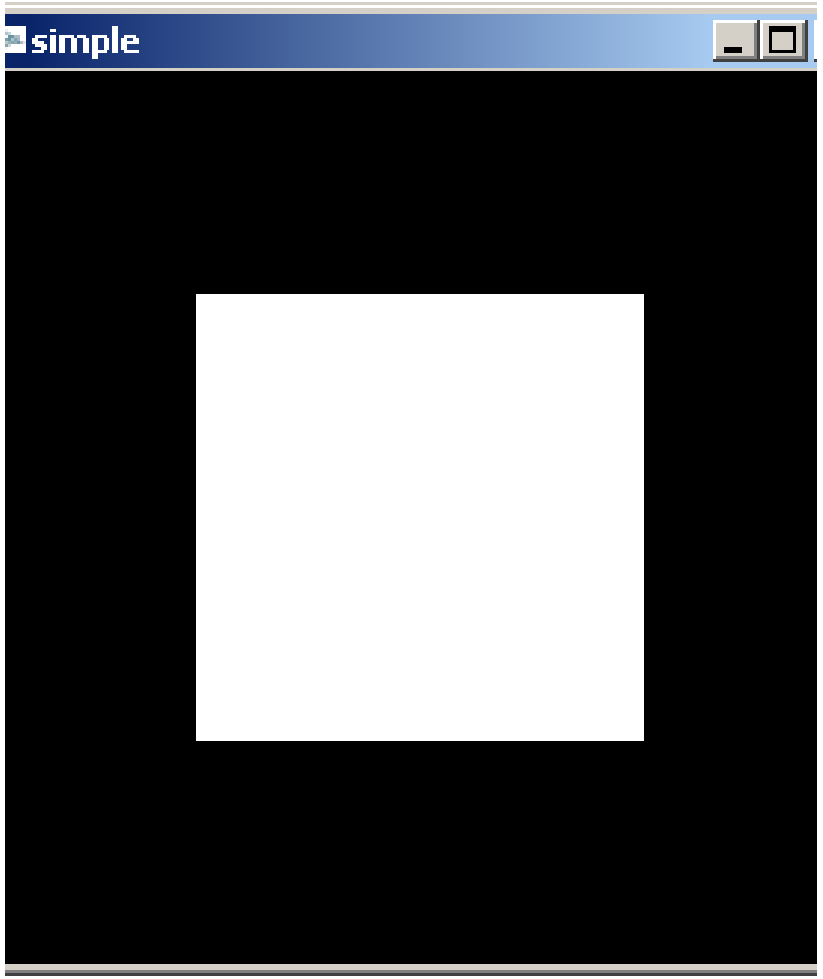


OpenGL portion of white-square code

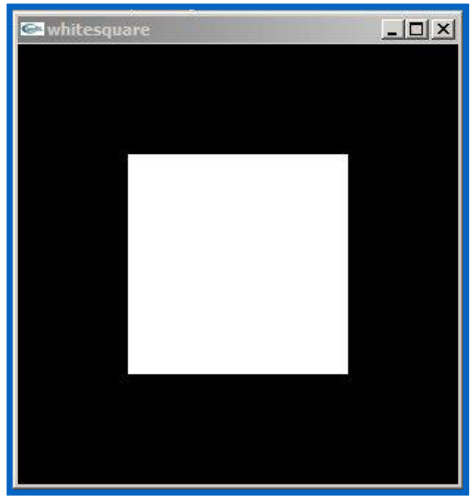
```
glClear(GL_COLOR_BUFFER_BIT); // black background
glLoadIdentity();
glOrtho(0, 4, 0, 4, -1, 1);

glBegin(GL_POLYGON);
// draw white square
  glVertex2i(1, 1);
  glVertex2i(3, 1);
  glVertex2i(3, 3);
  glVertex2i(1, 3);
glEnd();
glFlush();
// force completion
```

A Simple Program

Generate a square on a solid background



White-square code



OpenGL

GLUT

```
// Draw a white square against a black background

#include <windows.h>
#include <stdio.h>
#include <GL/glut.h>

void draw() {
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glOrtho(0, 4, 0, 4, -1, 1);
    glBegin(GL_POLYGON);
    glVertex2i(1, 1);
    glVertex2i(3, 1);
    glVertex2i(3, 3);
    glVertex2i(1, 3);
    glEnd();
    glFlush();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutCreateWindow("whitesquare");
    glutDisplayFunc(draw);
    glutMainLoop();
}
```

simple.c

```
#include <GL/glut.h>

void mydisplay(){

glClear(GL_COLOR_BUFFER_BIT);

  • glBegin(GL_POLYGON);
    • glVertex2f(-0.5, -0.5);
    • glVertex2f(-0.5, 0.5);
    • glVertex2f(0.5, 0.5);
    • glVertex2f(0.5, -0.5);
  • glEnd();
  • glFlush();

}

int main(int argc, char** argv){

  • glutCreateWindow("simple");
  • glutDisplayFunc(mydisplay);
  • glutMainLoop();

}
```

- Note that the program defines a *display callback* function named **`mydisplay`**
  - Every glut program must have a display callback
  - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened
  - The **`main`** function ends with the program entering an event loop

Event Loop

Compilation on Windows

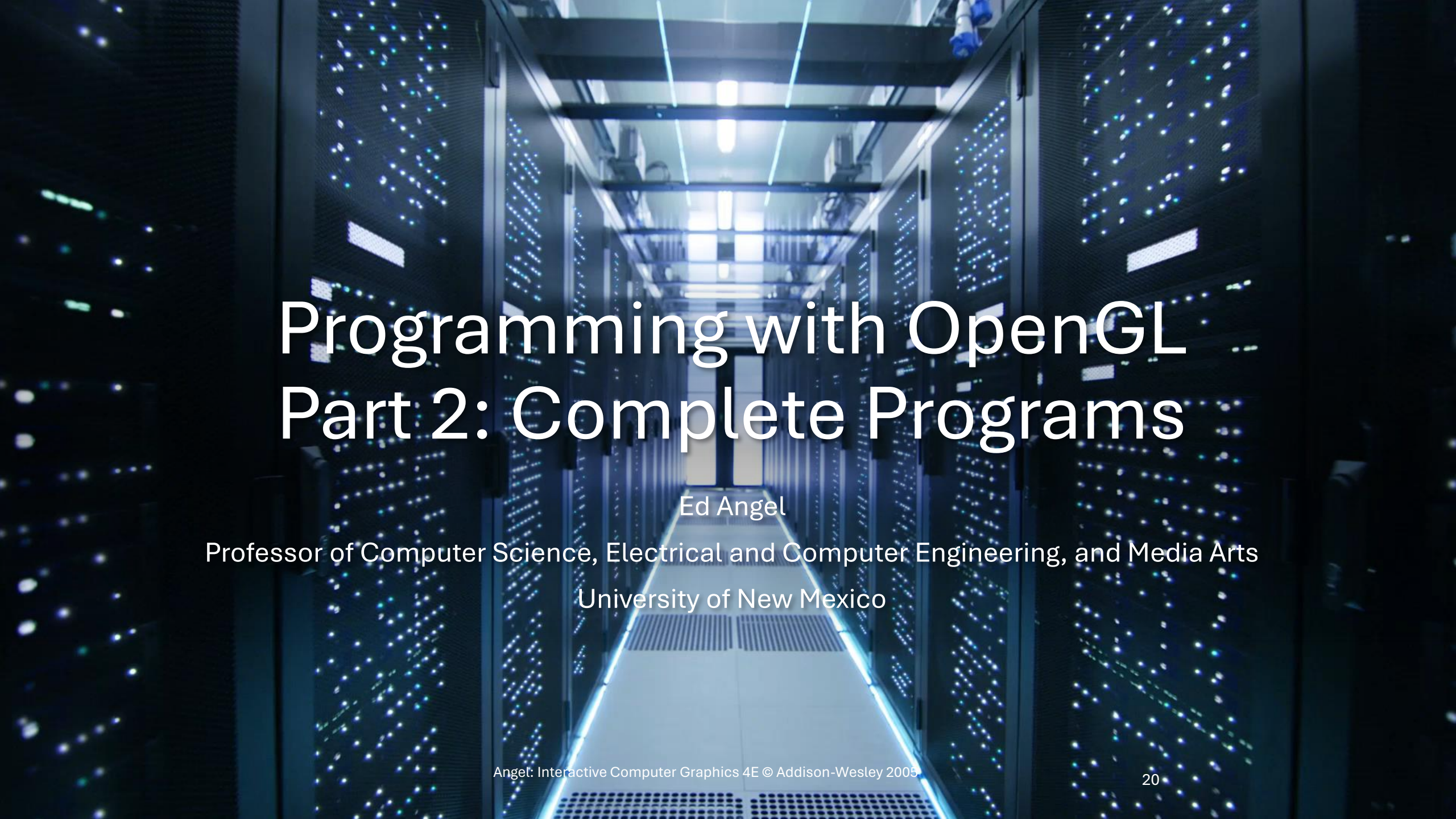
Visual C++

- Get glut.h, glut32.lib and glut32.dll from web
- Create a console application
- Add opengl32.lib, glut32.lib, glut32.lib to project settings (under link tab)

Borland C similar

Cygwin (linux under Windows)

- Can use gcc and similar makefile to linux
- Use –lopengl32 –lglu32 –lglut32 flags



Programming with OpenGL Part 2: Complete Programs

Ed Angel

Professor of Computer Science, Electrical and Computer Engineering, and Media Arts

University of New Mexico

Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - **`main()`**:
    - defines the callback functions
    - opens one or more windows with the required properties
    - enters event loop (last executable statement)
  - **`init()`**: sets the state variables
    - Viewing
    - Attributes
  - callbacks
    - Display function
    - Input and window functions

simple.c revisited

- In this version, we shall see the same output, but we have defined all the relevant state values through function calls using the default values
- We set
  - Colors
  - Viewing conditions
  - Window properties

main.c

```
#include <GL/glut.h>

int main(int argc, char** argv)
{
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
  glutInitWindowSize(500,500);
  glutInitWindowPosition(0,0);
  glutCreateWindow("simple");
  glutDisplayFunc(mydisplay);

  init();

  glutMainLoop();
}
```

includes **gl.h**

define window properties

display callback

set OpenGL state

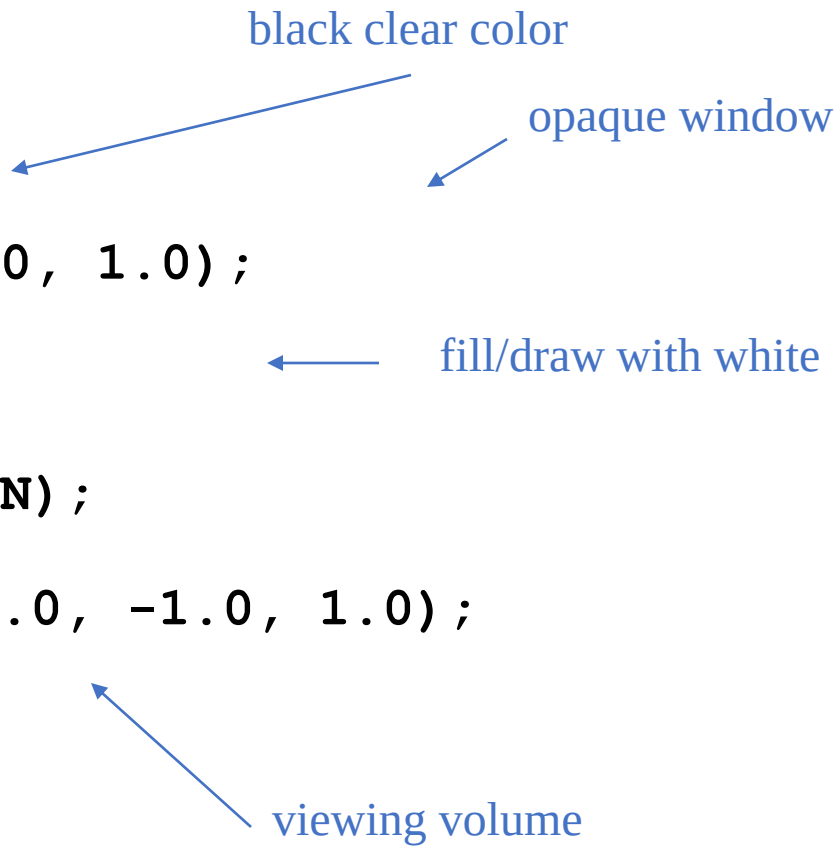
enter event loop

GLUT functions

- **`glutInit`** allows application to get command line arguments and initializes system
- **`gluInitDisplayMode`** requests properties for the window (the *rendering context*)
  - RGB color
  - Single buffering
  - Properties logically ORed together
- **`glutWindowSize`** in pixels
- **`glutWindowPosition`** from top-left corner of display
- **`glutCreateWindow`** create window with title “simple”
- **`glutDisplayFunc`** display callback
- **`glutMainLoop`** enter infinite event loop

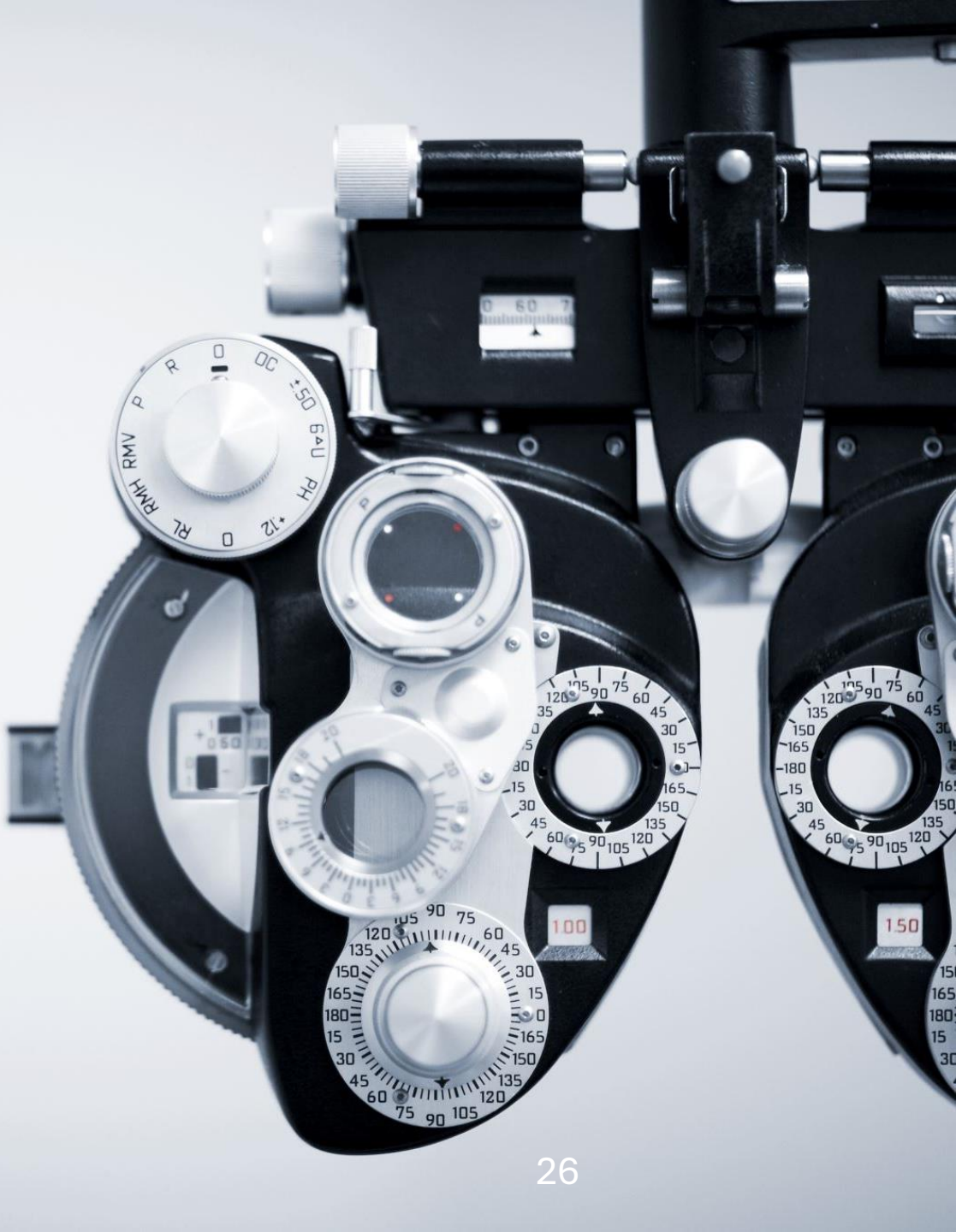
init.c

```
void init()
{
  glClearColor (0.0, 0.0, 0.0, 1.0);

  glColor3f(1.0, 1.0, 1.0);

  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```



Coordinate Systems

- The units in **`glVertex`** are determined by the application and are called *object* or *problem coordinates*
- The viewing specifications are also in object coordinates and it is the size of the viewing volume that determines what will appear in the image
- Internally, OpenGL will convert to *camera (eye) coordinates* and later to *screen coordinates*
- OpenGL also uses some internal representations that usually are not visible to the application



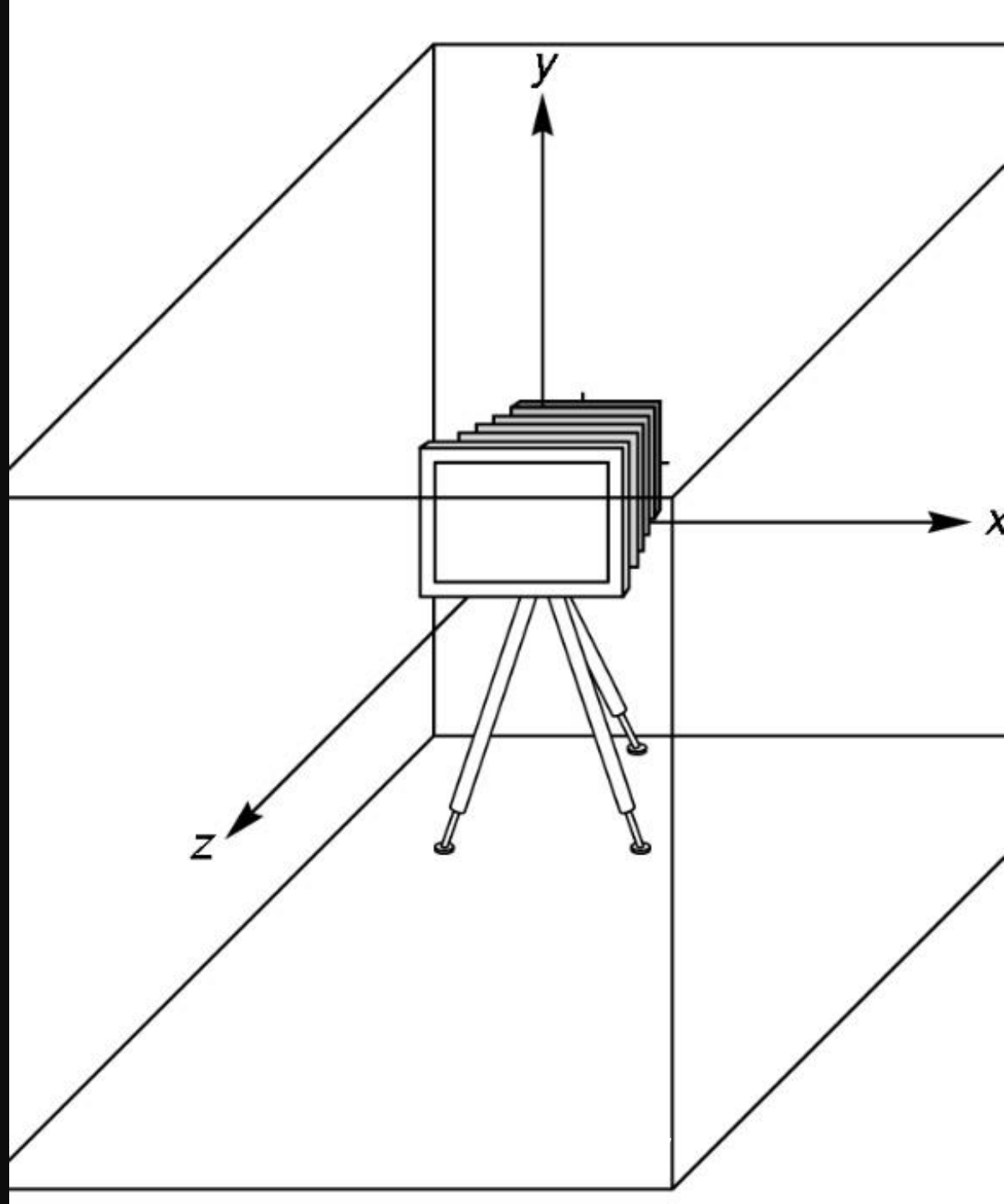
OpenGL Camera

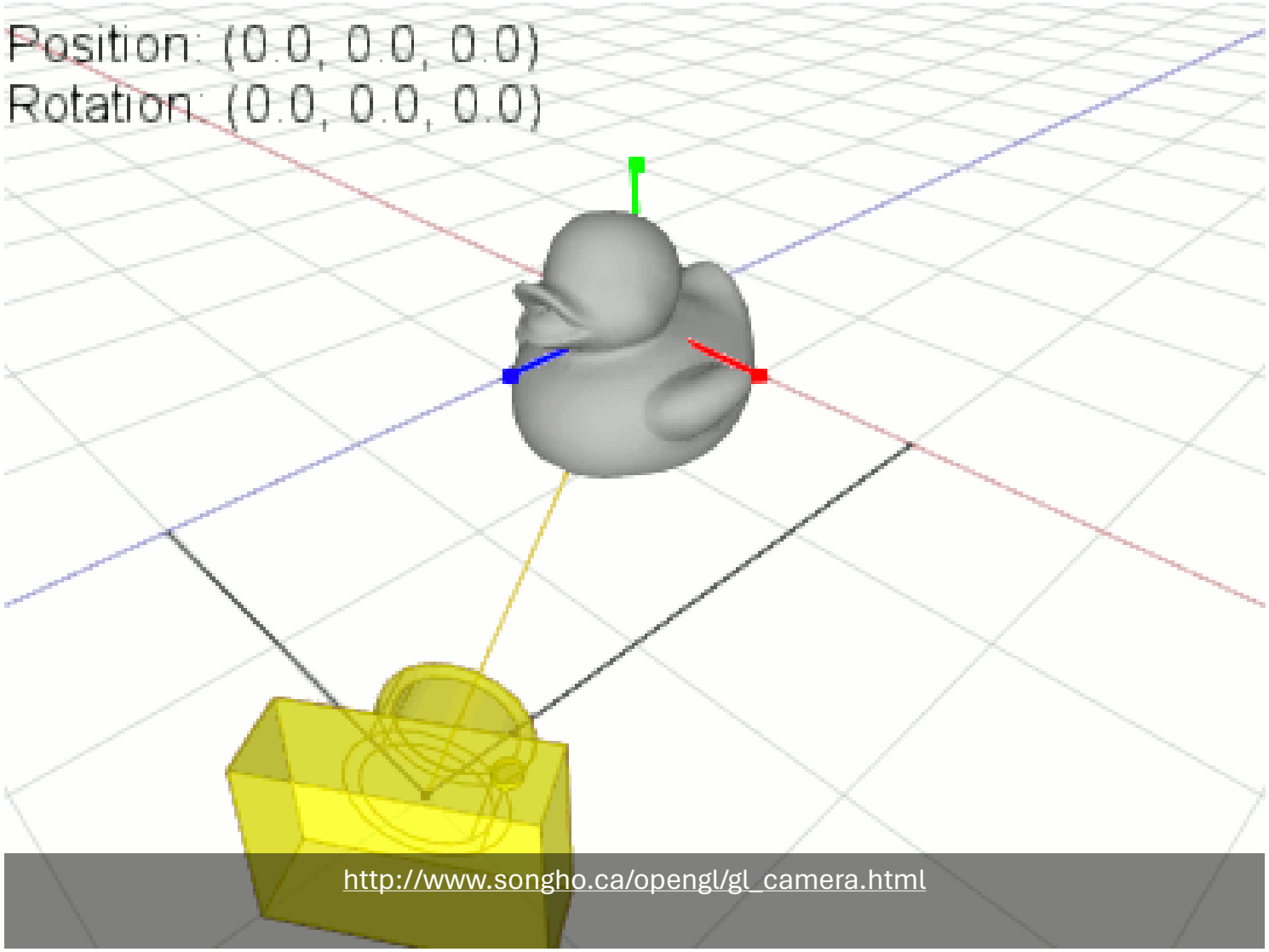
- OpenGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume

is a box centered at the

origin with a side of

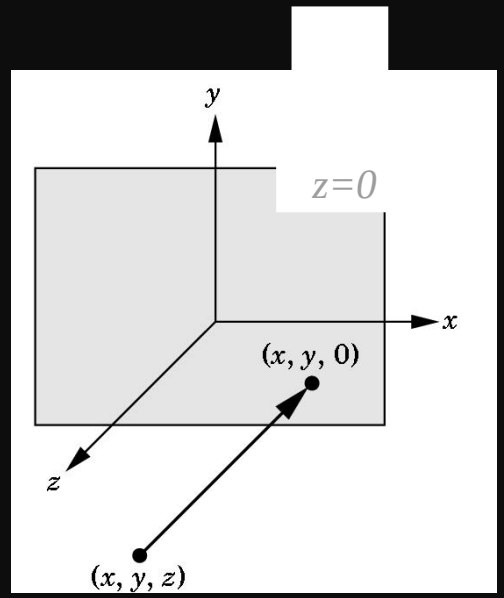
length 2

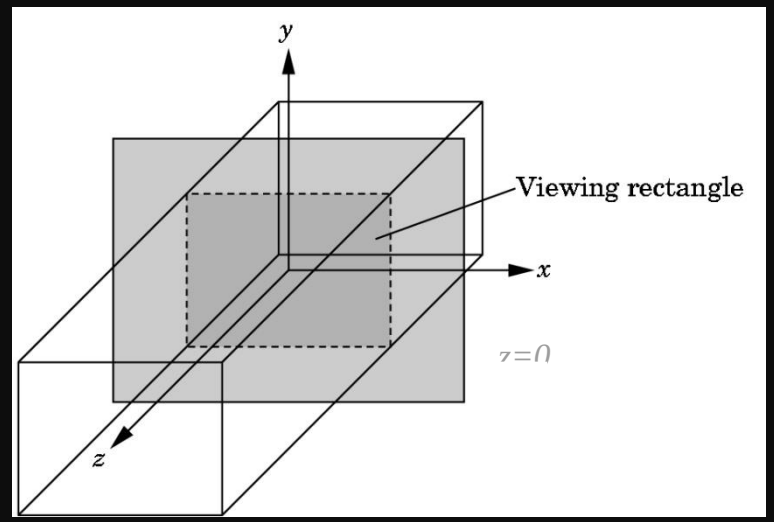



Position: (0.0, 0.0, 0.0)
Rotation: (0.0, 0.0, 0.0)
http://www.songho.ca/opengl/gl_camera.html

Orthographic Viewing

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





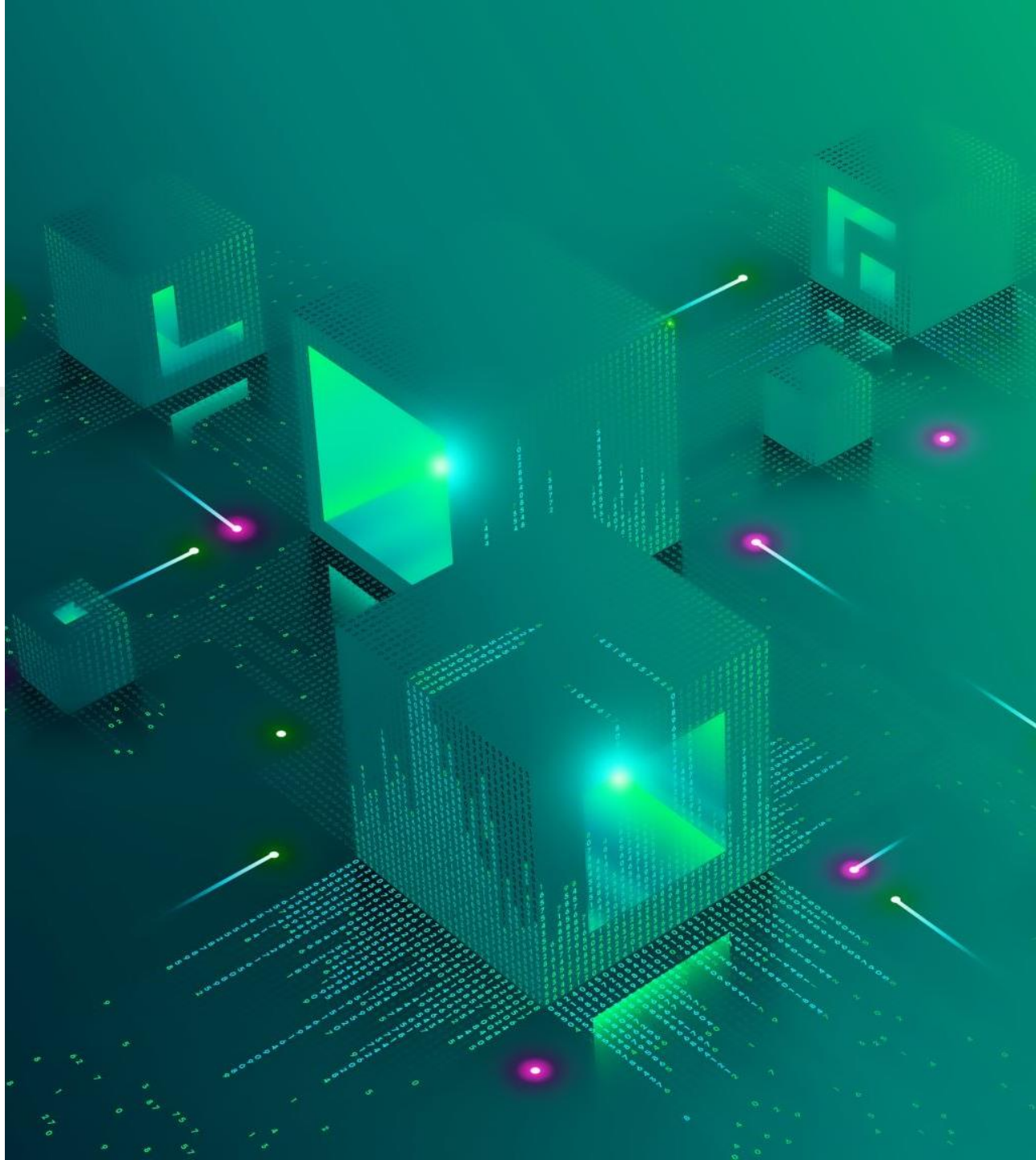
Transformations and Viewing

- In OpenGL, projection is carried out by a projection matrix (transformation)
- There is only one set of transformation functions so we must set the matrix mode first

```
glMatrixMode (GL_PROJECTION)
```

- Transformation functions are incremental, so we start with an identity matrix and alter it with a projection matrix that gives the view volume

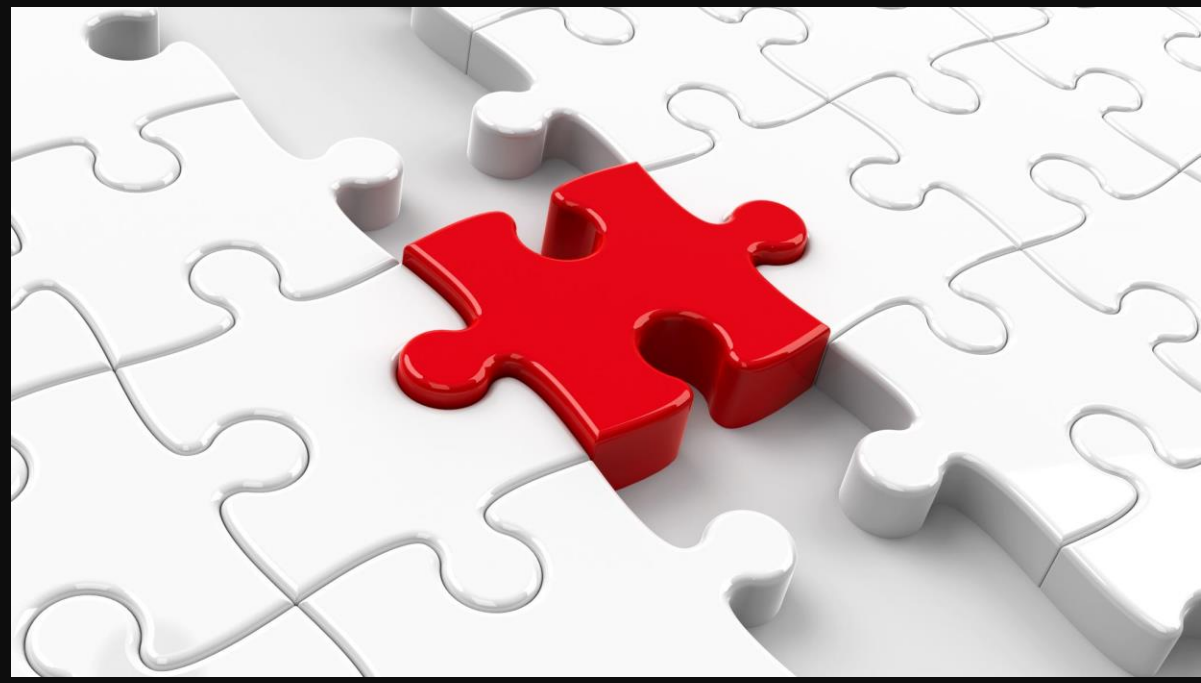
```
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -
1.0, 1.0);
```



How transformation matrix work?

- Suppose you have a 3D object with vertices defined in its local coordinate system. Let's say the object has a **vertex at position (1, 2, 3) in its local coordinate system, and you want to translate it by (2, 3, 4)** units in the world coordinate system.

- To calculate the transformation matrix for this translation, you can use the following steps:



How transformation matrix work?

1. Start with the identity matrix, which represents no transformation:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

2. Set the translation values in the last column of the matrix:

$$\begin{matrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{matrix}$$

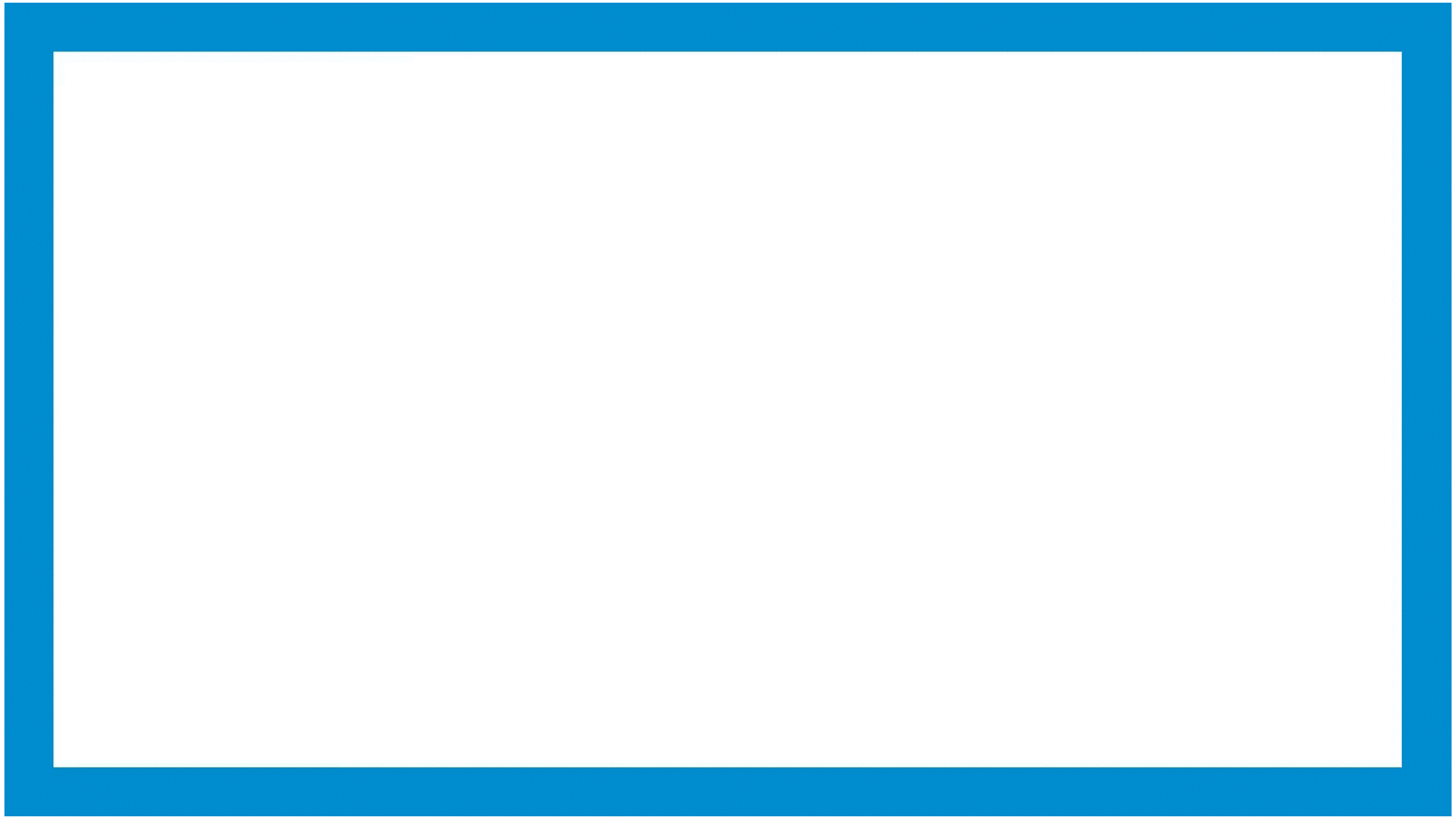
How transformation matrix work?

Note that the first three columns of the matrix represent the orientation and scaling of the object, which remain unchanged by the translation.

3.Multiply the transformation matrix by the object's local vertex coordinates as a column vector to obtain the transformed vertex coordinates:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 7 \\ 1 \end{bmatrix}$$

Further Learning



Two- and three-dimensional viewing

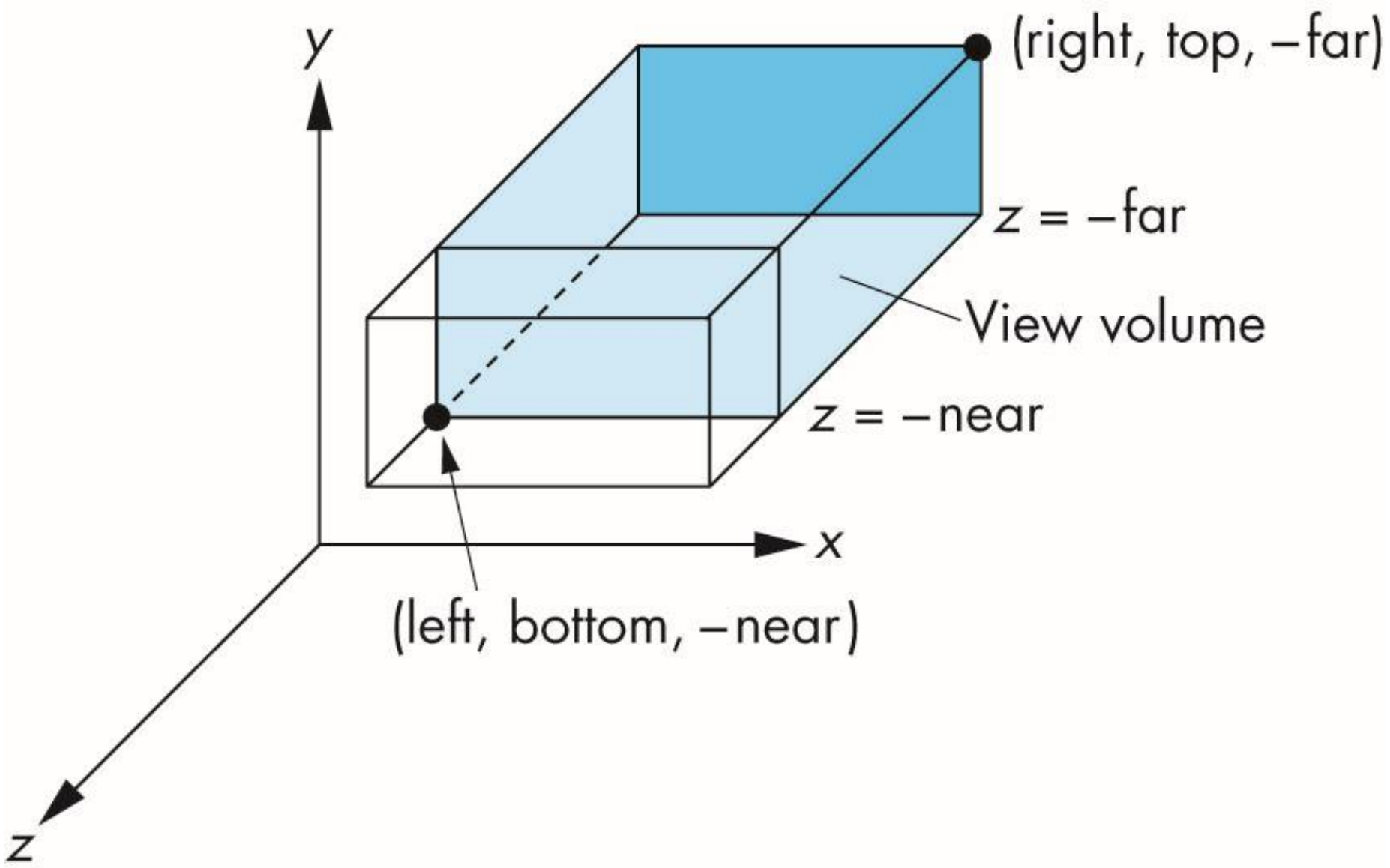
In **glOrtho(left, right, bottom, top, near, far)** the near and far distances are measured from the camera

Two-dimensional vertex commands place all vertices in the plane z=0

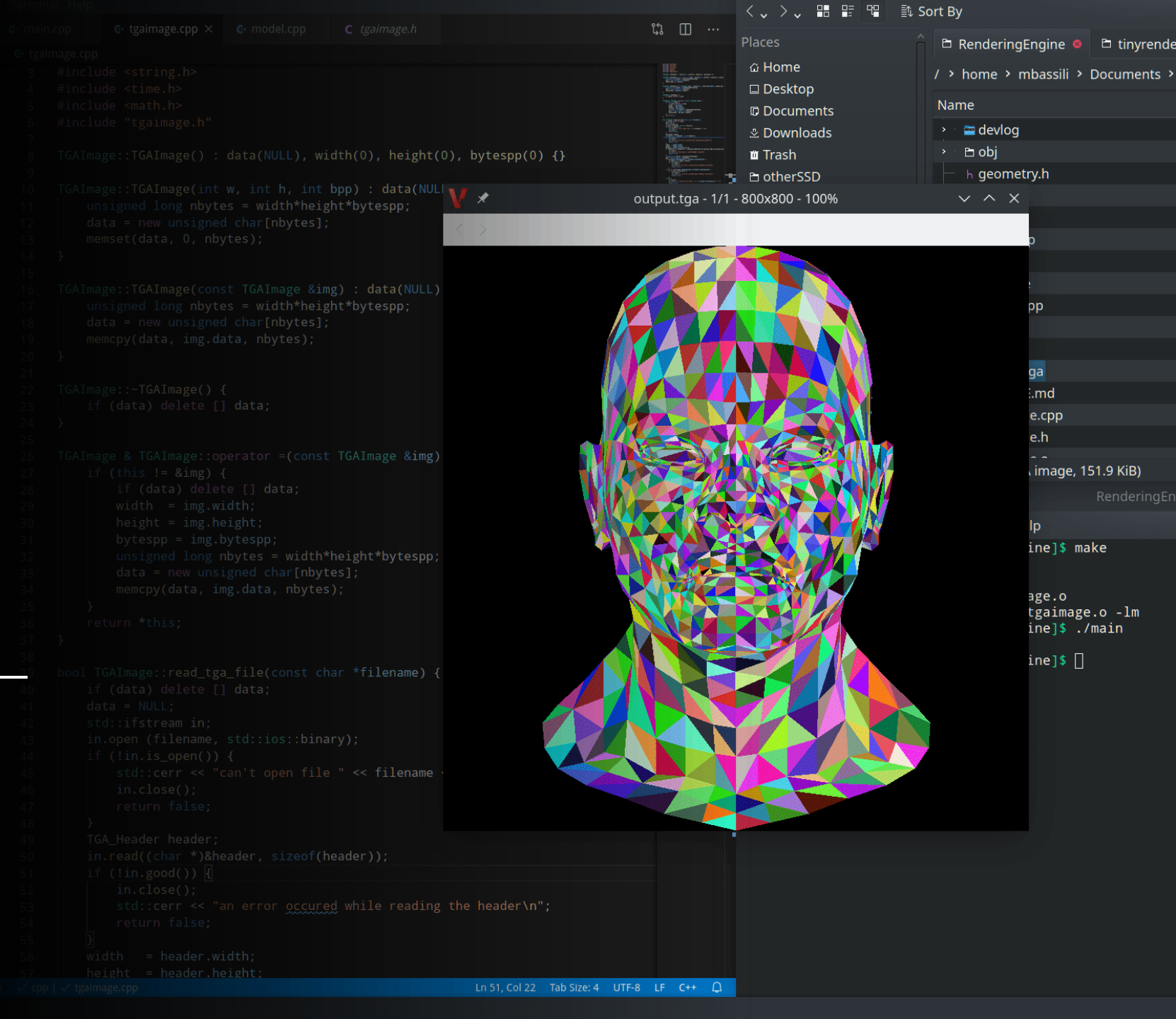
If the application is in two dimensions, we can use the function

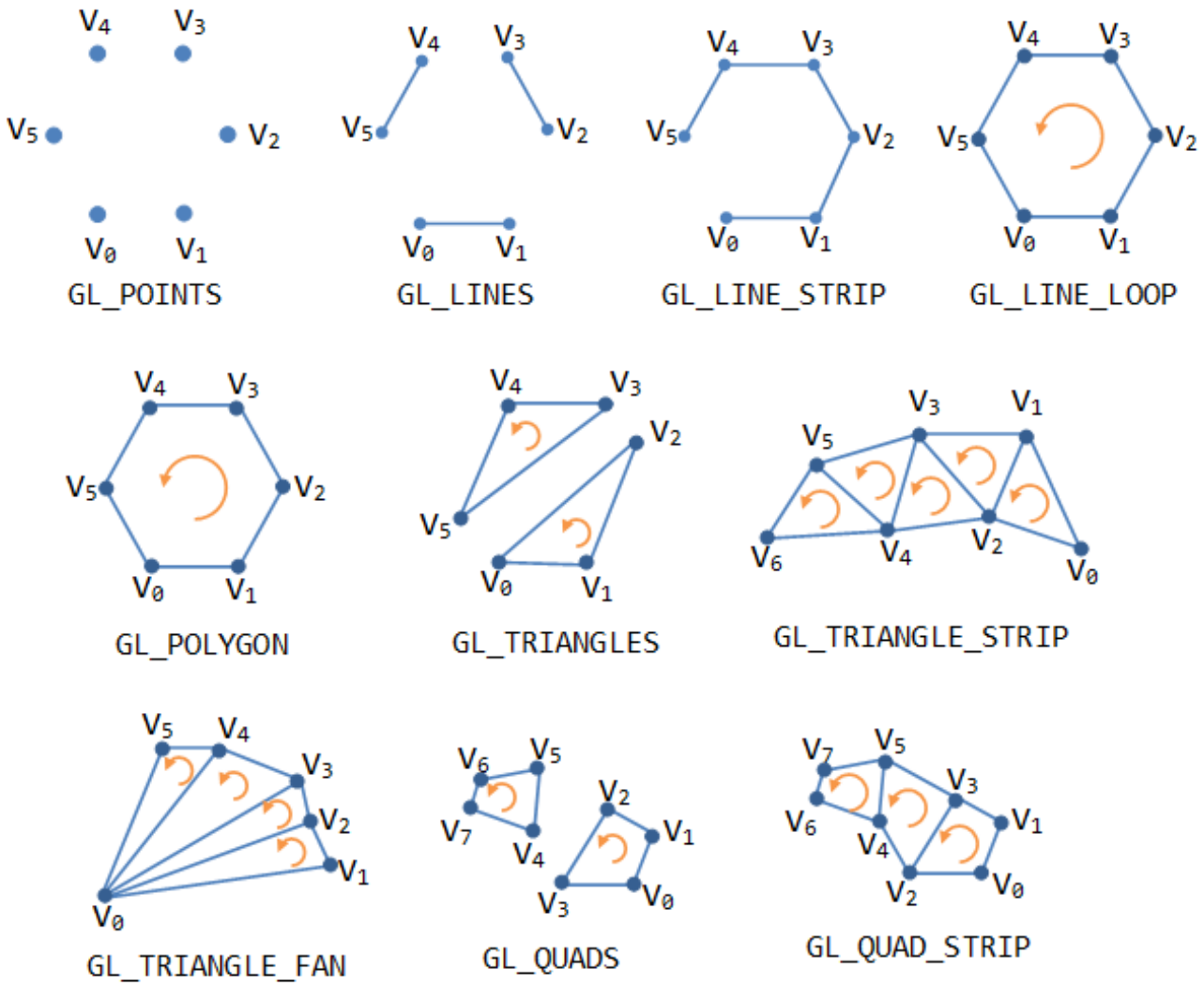
gluOrtho2D(left, right,bottom,top)

In two dimensions, the view or clipping volume becomes a *clipping window*


y
(right, top, –far)
z = –far
View volume
z = –near
x
(left, bottom, –near)
z

OpenGL Primitives

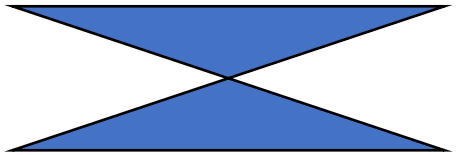




OpenGL Primitives

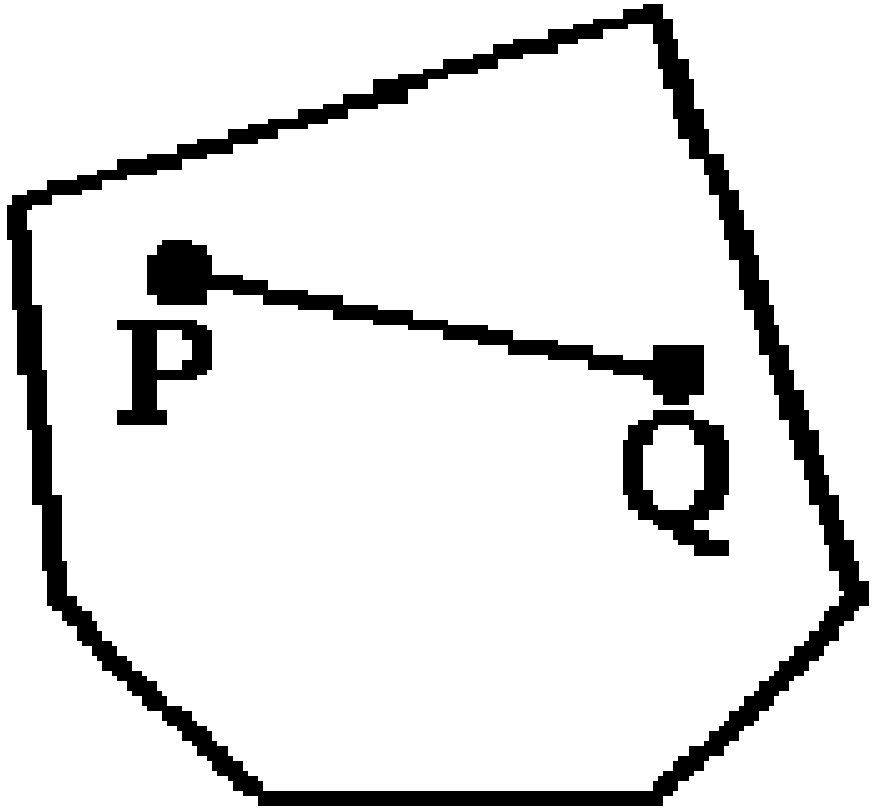
Polygon Issues

- OpenGL will only display polygons correctly that are
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- User program can check if above true
  - OpenGL will produce output if these conditions are violated but it **MAY** not be what is desired
- Triangles satisfy all conditions

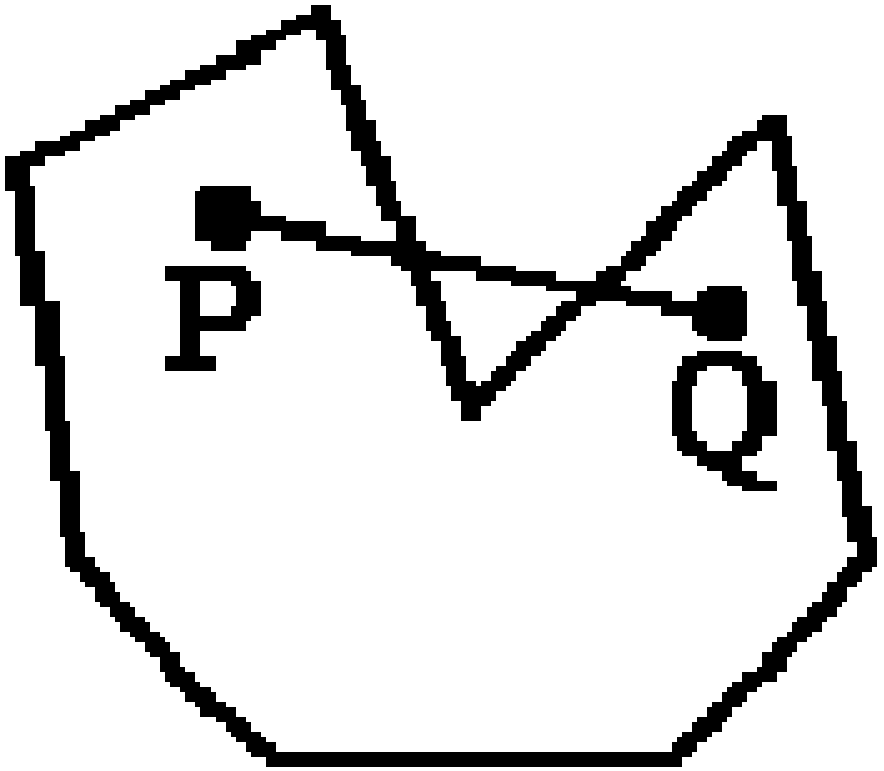


nonsimple polygon

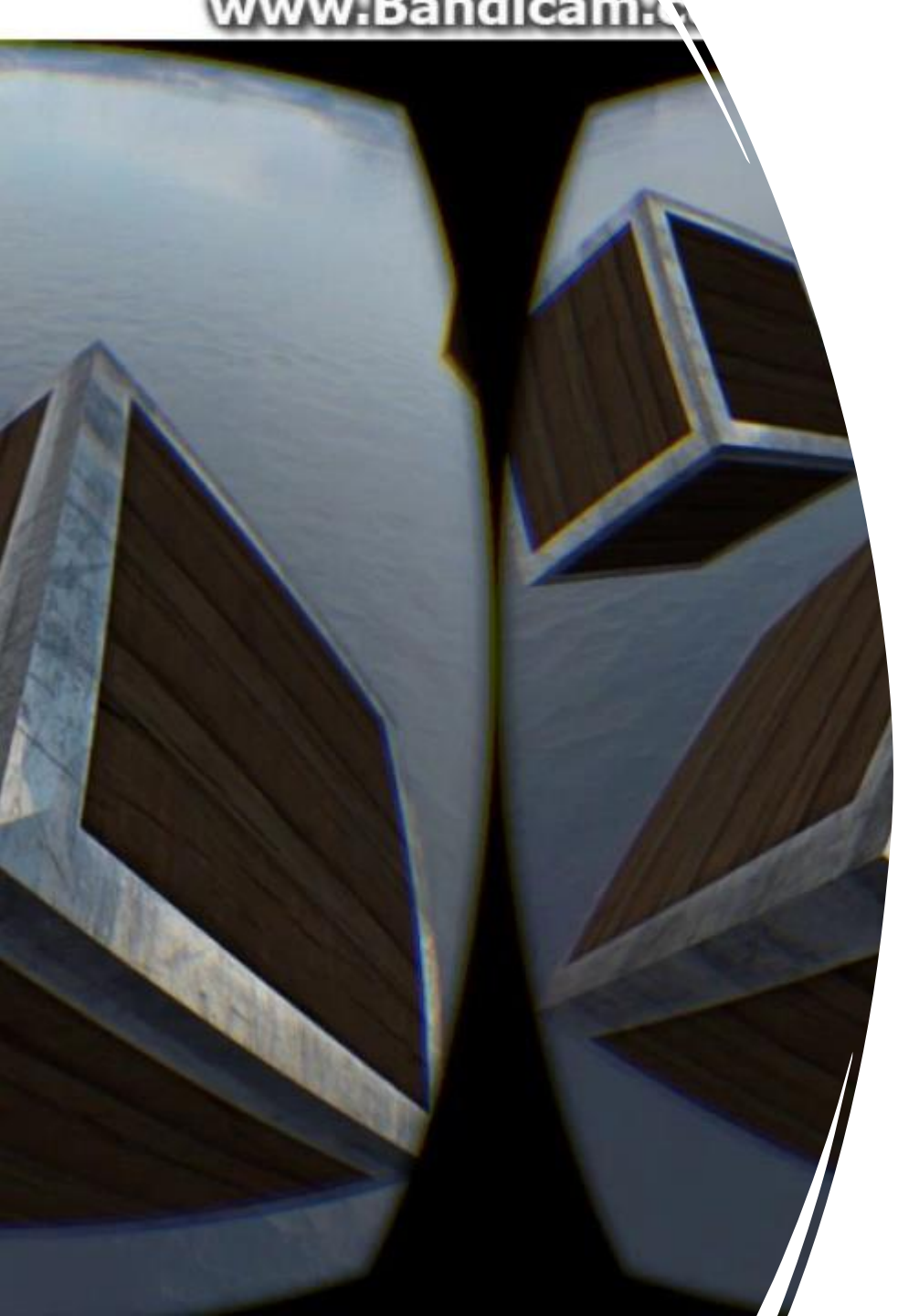
nonconvex polygon



Convex



NonConvex

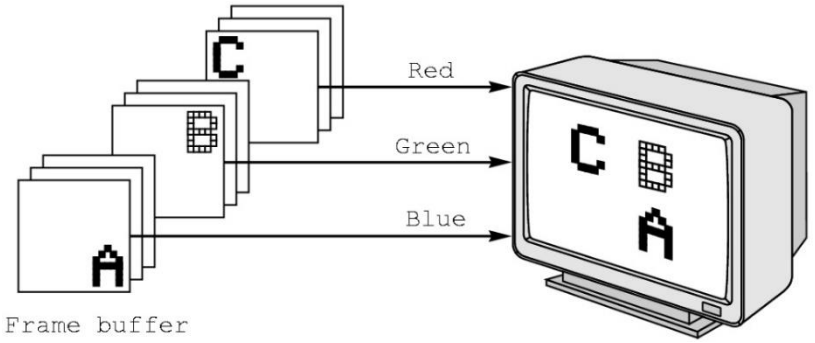


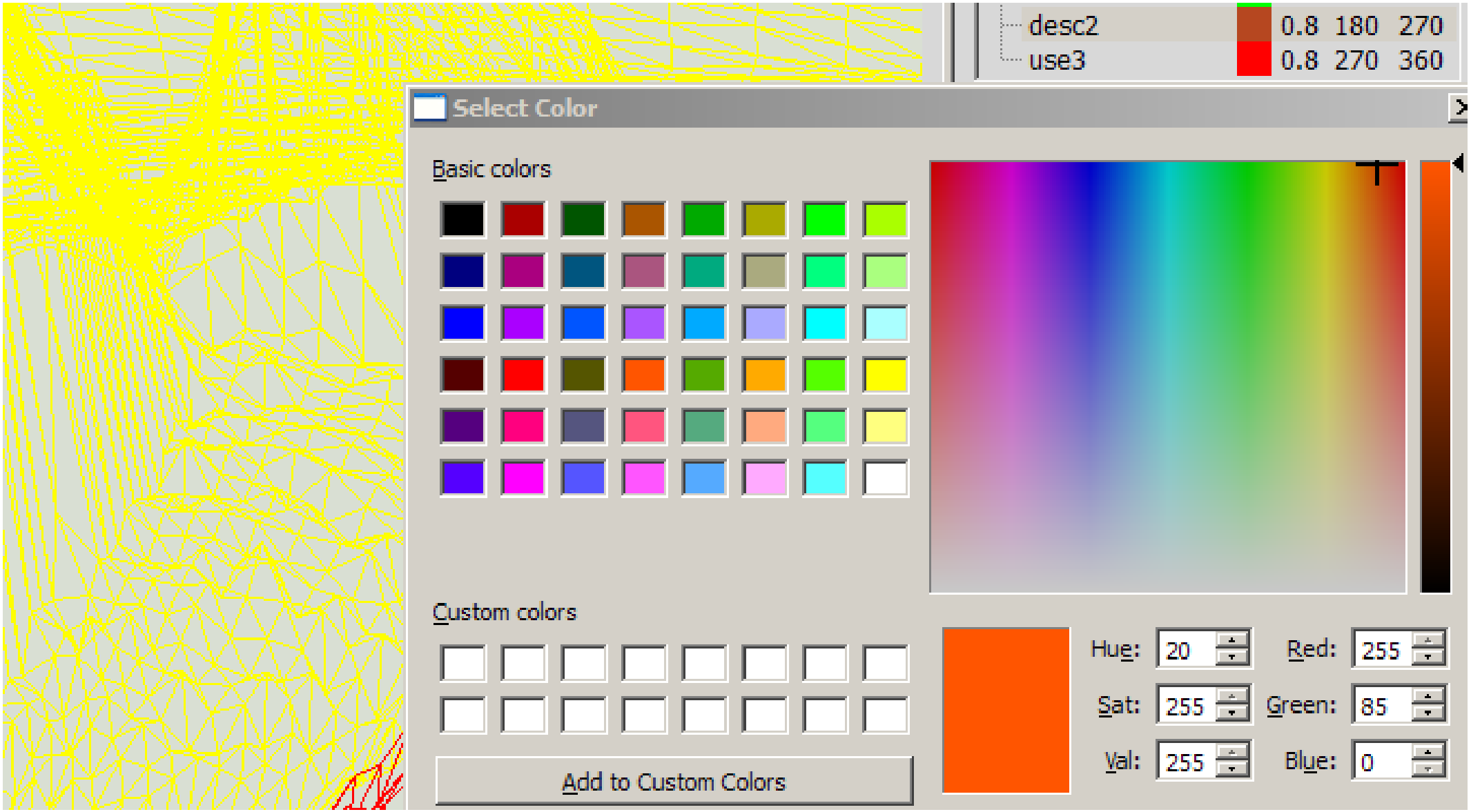
Attributes

- Attributes are part of the OpenGL state and determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges
    - Display vertices

RGB color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Note in **`glColor3f`** the color values range from 0.0 (none) to 1.0 (all), whereas in **`glColor3ub`** the values range from 0 to 255



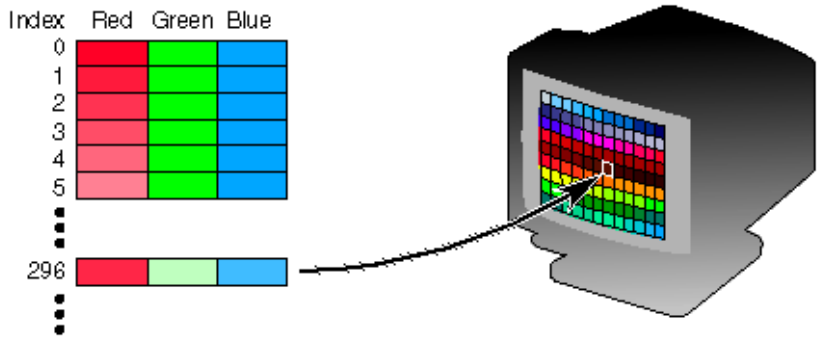

desc2
0.8 180 270
use3
0.8 270 360
Select Color
Basic colors
Custom colors
Add to Custom Colors
Hue: 20
Sat: 255
Val: 255
Red: 255
Green: 85
Blue: 0

Setting colors of Objects

```
glColor3f(Red, Green, Blue)
```

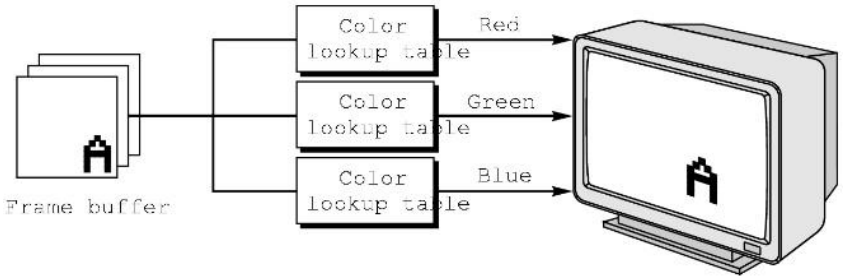
`glColor3f(0.0, 0.0, 0.0);`	black
`glColor3f(1.0, 0.0, 0.0);`	red
`glColor3f(0.0, 1.0, 0.0);`	green
`glColor3f(1.0, 1.0, 0.0);`	yellow
`glColor3f(0.0, 0.0, 1.0);`	blue
`glColor3f(1.0, 0.0, 1.0);`	magenta
`glColor3f(0.0, 1.0, 1.0);`	cyan
`glColor3f(1.0, 1.0, 1.0);`	white

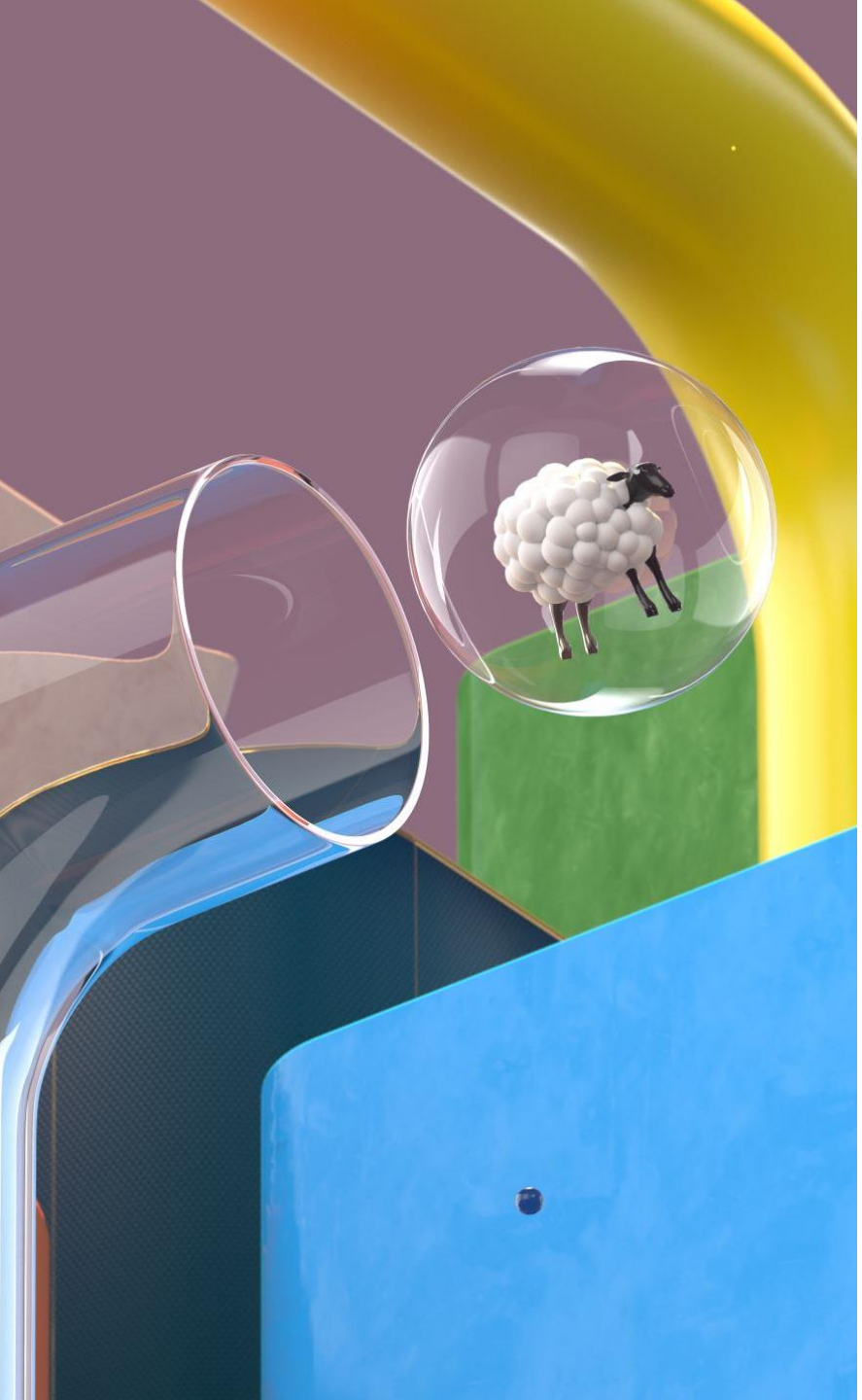
`glColor3f(0.5, 0.0, 0.0);`	dark red
`glColor3f(0.0, 0.5, 0.0);`	dark green
`glColor3f(0.0, 0.0, 0.5);`	dark blue



Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive
    - Need more colors for shading





Color and State

- The color as set by **`glColor`** becomes part of the state and will be used until changed
  - Colors and other attributes are not part of the object but are assigned when the object is rendered
- We can create conceptual *vertex colors* by code such as

```
glColor
  glVertex
  glColor
  glVertex
```

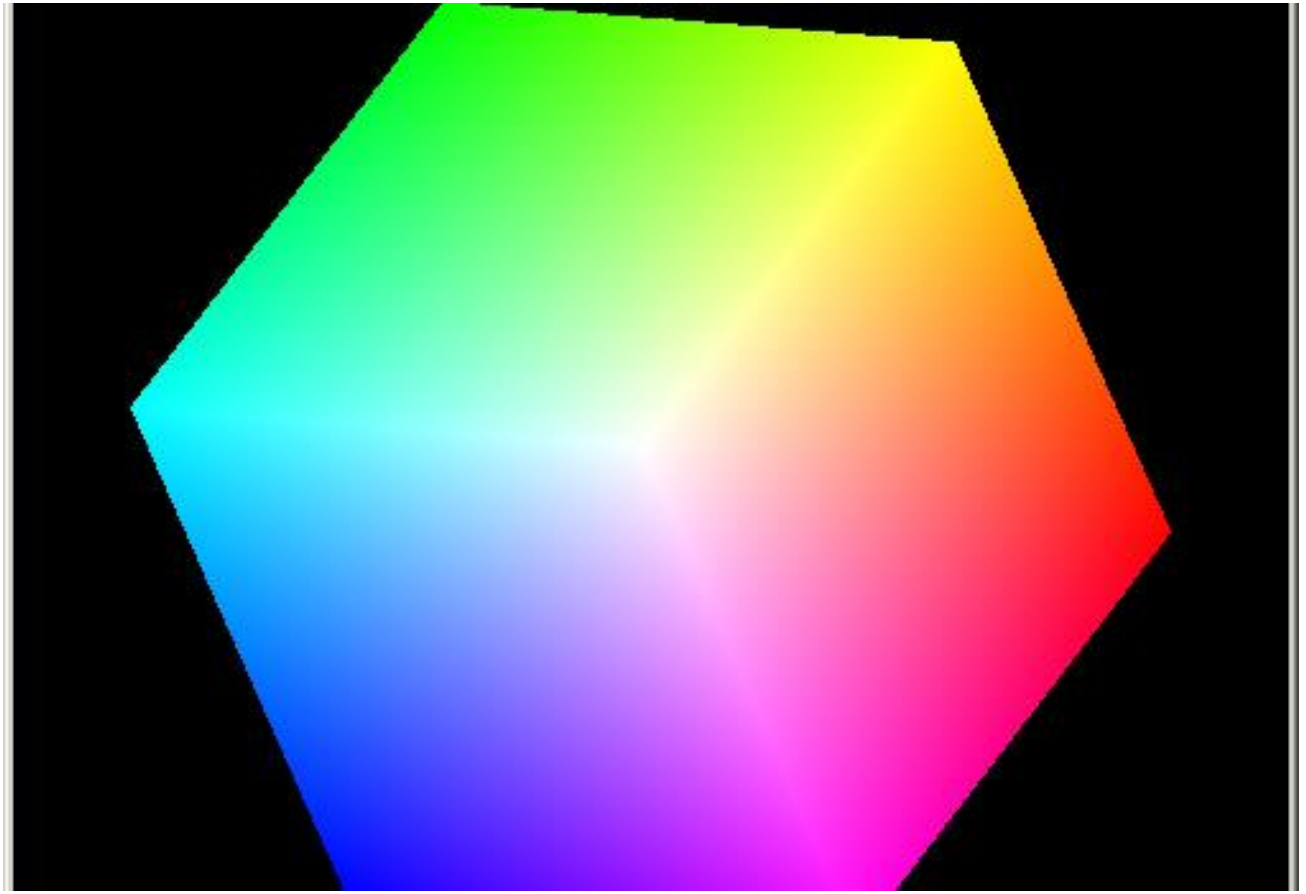
Smooth Color

- Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is *flat shading*
  - Color of first vertex

  determines fill color
- **`glShadeModel`**

  `(GL_SMOOTH)`

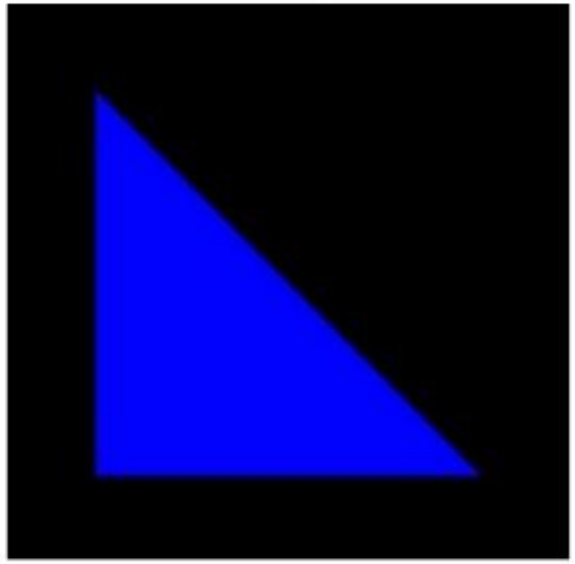
  or **`GL_FLAT`**



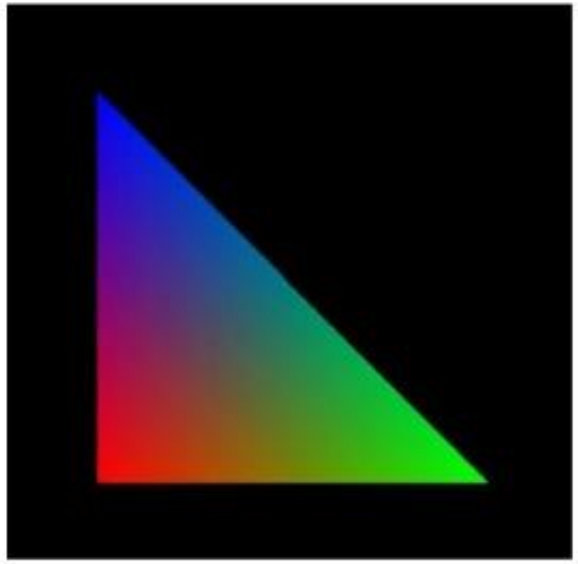
The Image

- Color of last vertex with flat shading

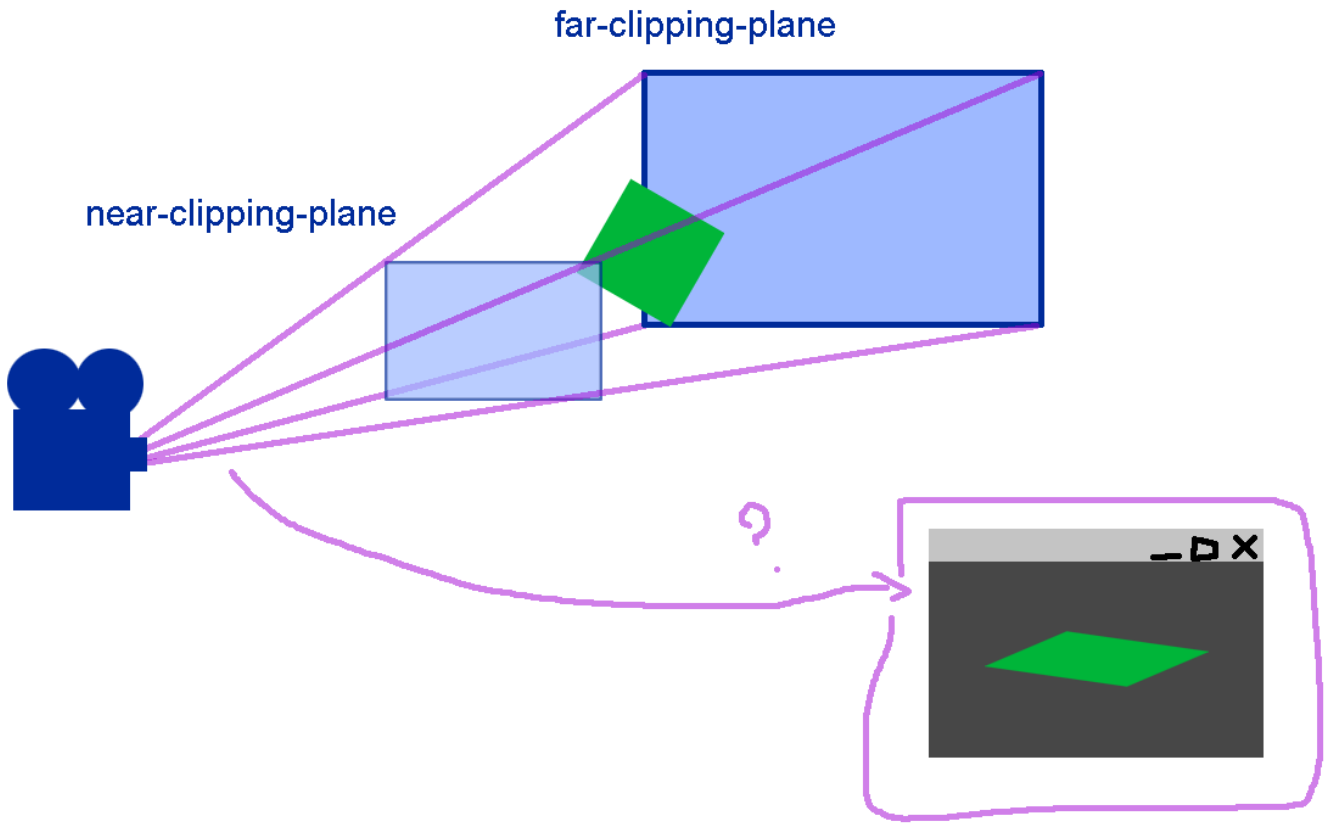
glShadeModel(GL_FLAT)



glShadeModel(GL_SMOOTH)

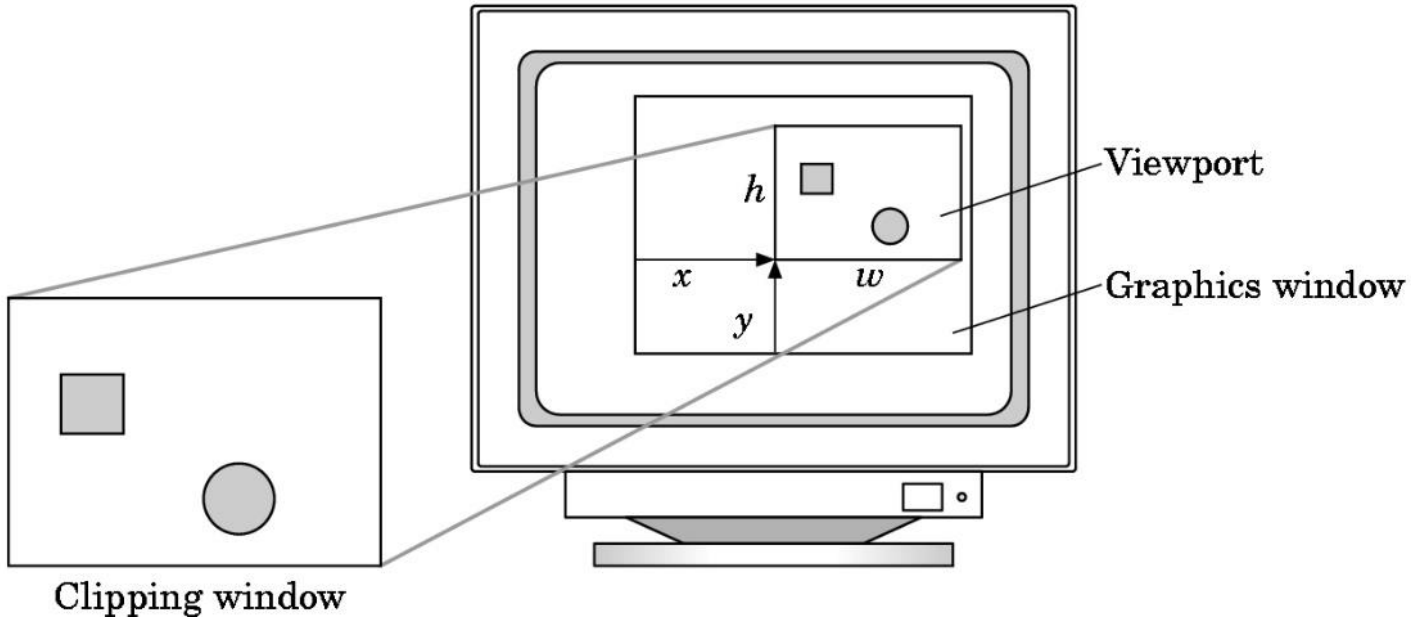


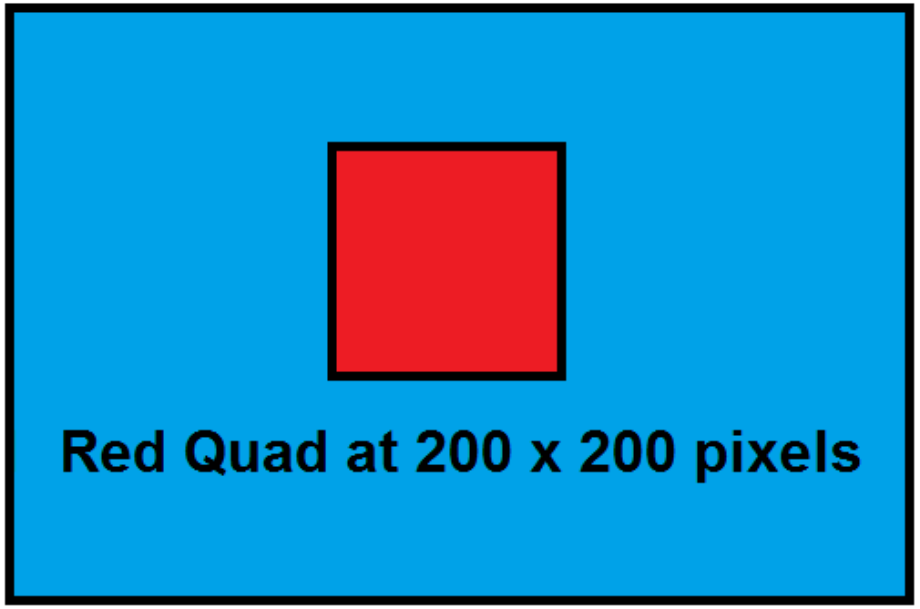
OpenGL Viewports



Viewports

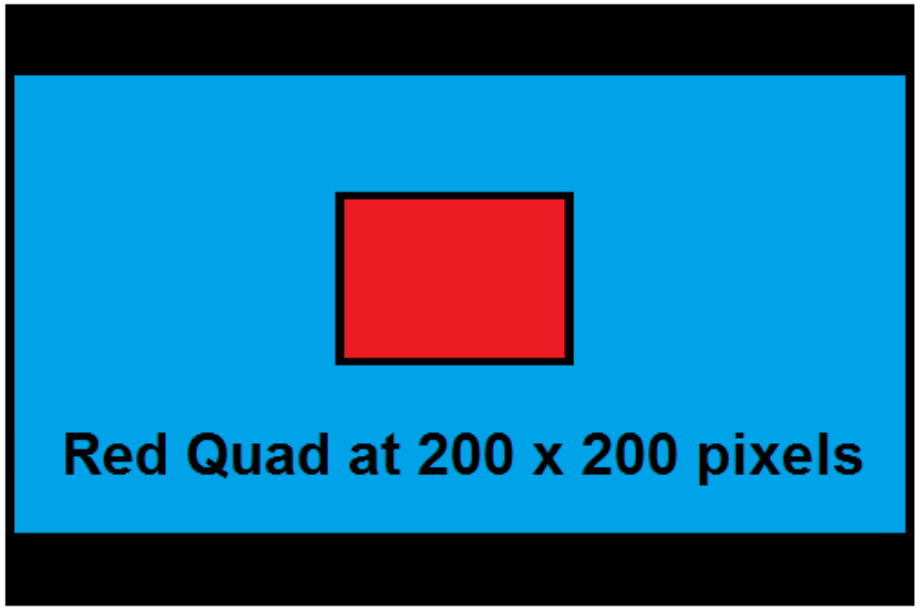
- Do not have use the entire window for the image: **`glViewport(x,y,w,h)`**
- Values in pixels (screen coordinates)





Device Screen

GL Viewport takes up whole screen



Device Screen

GL Viewport scaled vertically

The End