OpenGL

Unit 10

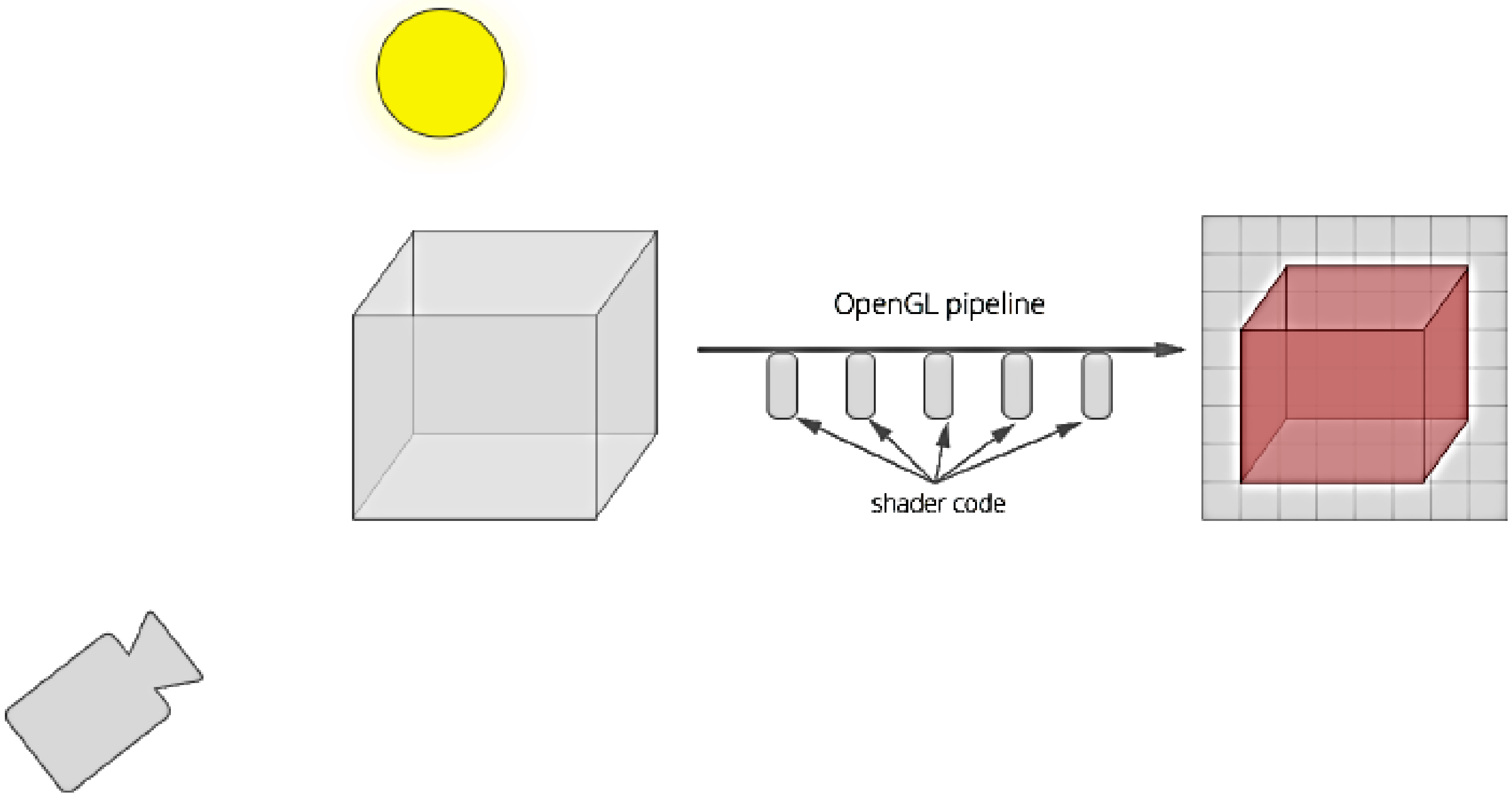
Introduction

- OpenGL (Open Graphics Library) is the computer industry's standard application program interface (API) for defining 2-D and 3-D graphic images.
- Prior to OpenGL, any company developing a graphical application typically had to rewrite the graphics part of it for each operating system platform and had to be cognizant of the graphics hardware as well.
- With OpenGL, an application can create the same effects in any operating system using any OpenGL-adhering graphics adapter.

Introduction…

- OpenGL specifies a set of "commands" or immediately executed functions.
- Each command directs a drawing action or causes special effects. A list of these commands can be created for repetitive effects.
- OpenGL is independent of the windowing characteristics of each operating system, but provides special "glue" routines for each operating system that enable OpenGL to work in that system's windowing environment.
- OpenGL comes with a large number of built-in capabilities request able through the API.
- These include hidden surface removal, alpha blending (transparency), antialiasing , texture mapping, pixel operations, viewing and modeling transformations, and atmospheric effects (fog, smoke, and haze).

Introduction…



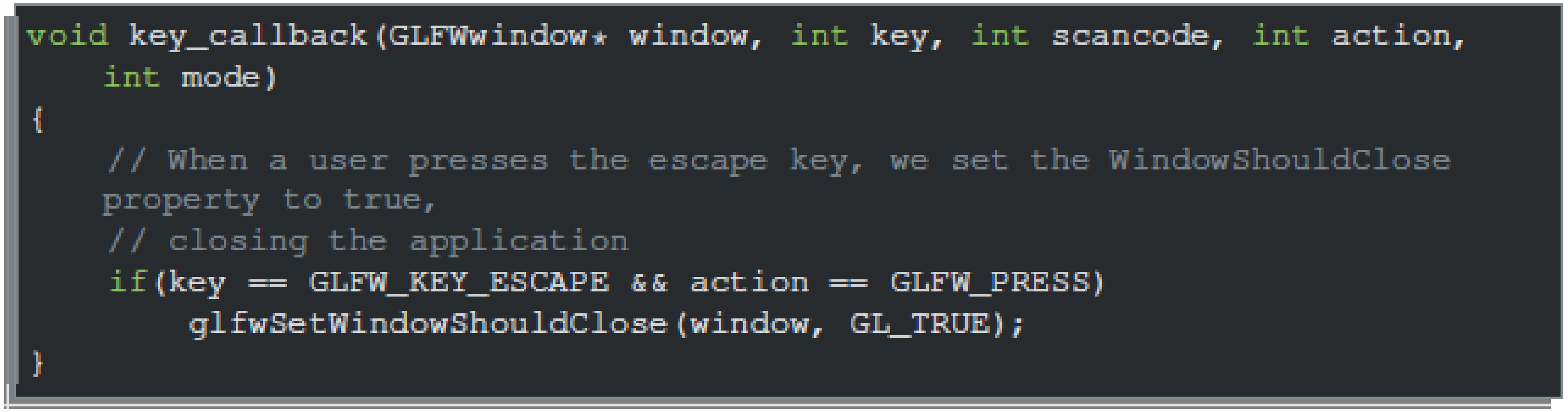
callback functions

- A callback function is basically a function pointer that you can set that GLFW can call at an appropriate time. One of those callback functions that we can set is the KeyCallback function, which should be called whenever the user interacts with the keyboard.
- The prototype of this function is as follows:

```
void key_callback(GLFWwindow* window, int key, int scancode, int action,
    int mode);
```

The key input function takes a GLFWwindow as its first argument, an integer that specifies the key pressed, an action that specifies if the key is pressed or released and an integer representing some bit flags to tell you if shift, control, alt or super keys have been pressed. Whenever a user pressed a key, GLFW calls this function and fills in the proper arguments for you to process.

callback functions



```
void key_callback(GLFWwindow* window, int key, int scancode, int action,
    int mode)
{
    // When a user presses the escape key, we set the WindowShouldClose
    property to true,
    // closing the application
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```

- In our (newly created) key_callback function we check if the key pressed equals the escape key and if it was pressed (not released) we close GLFW by setting its WindowShouldClose property to true using glfwSetwindowShouldClose. The next condition check of the main while loop will then fail and the application closes.

callback function

- A **callback function** is a **function** which the library (GLUT) calls when it needs to know how to proccess something.
- e.g. when glut gets a key down event it uses the glutKeybourdFunc **callback** routine to find out what to do with a key press.

callback function

- GLUT supports a number of callbacks to respond to events. There are three types of callbacks: window, menu, and global. Window callbacks indicate when to redisplay or reshape a window, when the visibility of the window changes, and when input is available for the window. The menu callback is set by the glutCreateMenu call described already. The global callbacks manage the passing of time and menu usage. The calling order of callbacks between different windows is undefined.
- Callbacks for input events should be delivered to the window the event occurs in. Events should not propagate to parent windows.

Color commands

- There are many ways to specify a color in computer graphics, but one of the simplest and most widely used methods of describing a color is the RGB color model. RGB stands for the colors red, green and blue: the additive primary colors. Each of these colors is given a value, in OpenGL usually a value between 0 and 1. 1 means as much of that color as possible, and 0 means none of that color. We can mix these three colors together to give us a complete range of colors, as shown to on the left.

Color commands

- For instance, pure red is represented as (1, 0, 0) and full blue is (0, 0, 1). White is the combination of all three, denoted (1, 1, 1), while black is the absence of all three, (0, 0, 0). Yellow is the combination of red and green, as in (1, 1, 0). Orange is yellow with slightly less green, represented as (1, 0.5, 0).

Using glColor3f

- glColor3f() takes 3 arguments: the red, green and blue components of the color you want. After you use glColor3f, everything you draw will be in that color. For example,

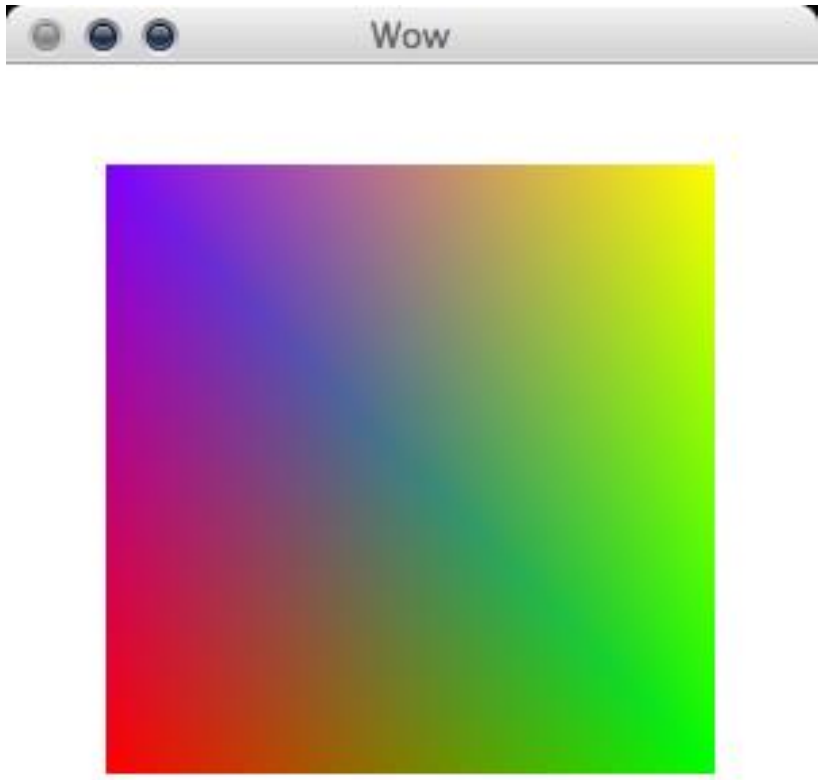
```
void display() {
       glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
       glColor3f(0.5f, 0.0f, 1.0f); // (0.5, 0, 1) is
half red and full blue, giving dark purple.

       glBegin(GL_QUADS);
              glVertex2f(-0.75, 0.75);
              glVertex2f(-0.75, -0.75);
              glVertex2f(0.75, -0.75);
              glVertex2f(0.75, 0.75);
       glEnd();
       glutSwapBuffers();
}
```

Giving Individual Vertices Different Colors

- glColor3f can be called in between glBegin and glEnd. When it is used this way, it can be used to give each vertex its own color. The resulting rectangle is then shaded with an attractive color gradient, as shown on the right.

```
void display() {
glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
        glBegin(GL_QUADS);
        glColor3f(1.0f, 0.0f, 1.0f); // make this vertex purple
        glVertex2f(-0.75, 0.75);
        glColor3f(1.0f, 0.0f, 0.0f); // make this vertex red
        glVertex2f(-0.75, -0.75);
        glColor3f(0.0f, 1.0f, 0.0f); // make this vertex green
        glVertex2f(0.75, -0.75);
        glColor3f(1.0f, 1.0f, 0.0f); // make this vertex yellow
        glVertex2f(0.75, 0.75);
glEnd();
glutSwapBuffers(); }
```

Output



Drawings pixels

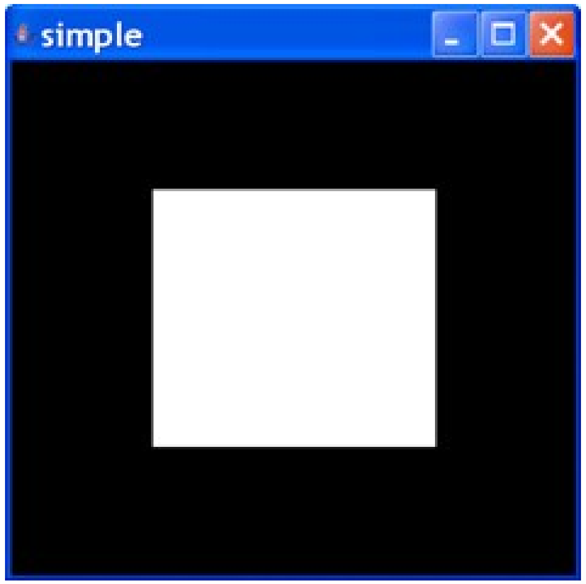
- OpenGL provides only the lowest level of support for drawing strings of characters and manipulating fonts.
- The commands **glRasterPos*()** and **glBitmap()** position and draw a single bitmap on the screen.
- In addition, through the display-list mechanism, you can use a sequence of character codes to index into a corresponding series of bitmaps representing those characters.
- You'll have to write your own routines to provide any other support you need for manipulating bitmaps, fonts, and strings of characters.

Drawing lines

```
glBegin(GL_LINES);
        glVertex2f(.25,0.25);
        glVertex2f(.75,.75);
glEnd();
```

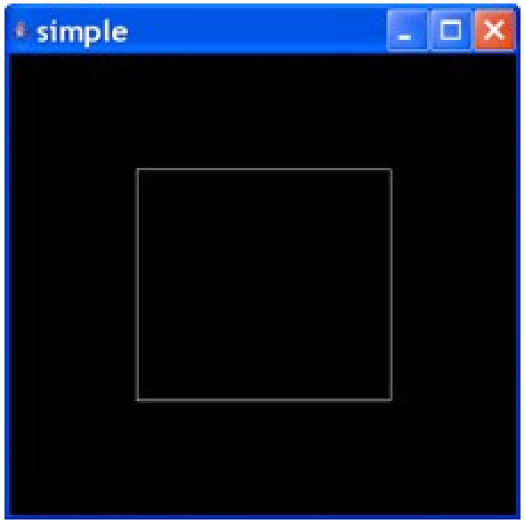
Drawing lines

```
/* Draws two horizontal lines */
glBegin(GL_LINES);
        glVertex2f(0.5f, 0.5f);
        glVertex2f(-0.5f, 0.5f);
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(0.5f, -0.5f);
glEnd();
```



Loop of lines

```
/* Draws a square */
glBegin(GL_LINE_LOOP);
        glVertex2f(0.5f, 0.5f);
        glVertex2f(-0.5f, 0.5f);
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(0.5f, -0.5f);
glEnd();
```

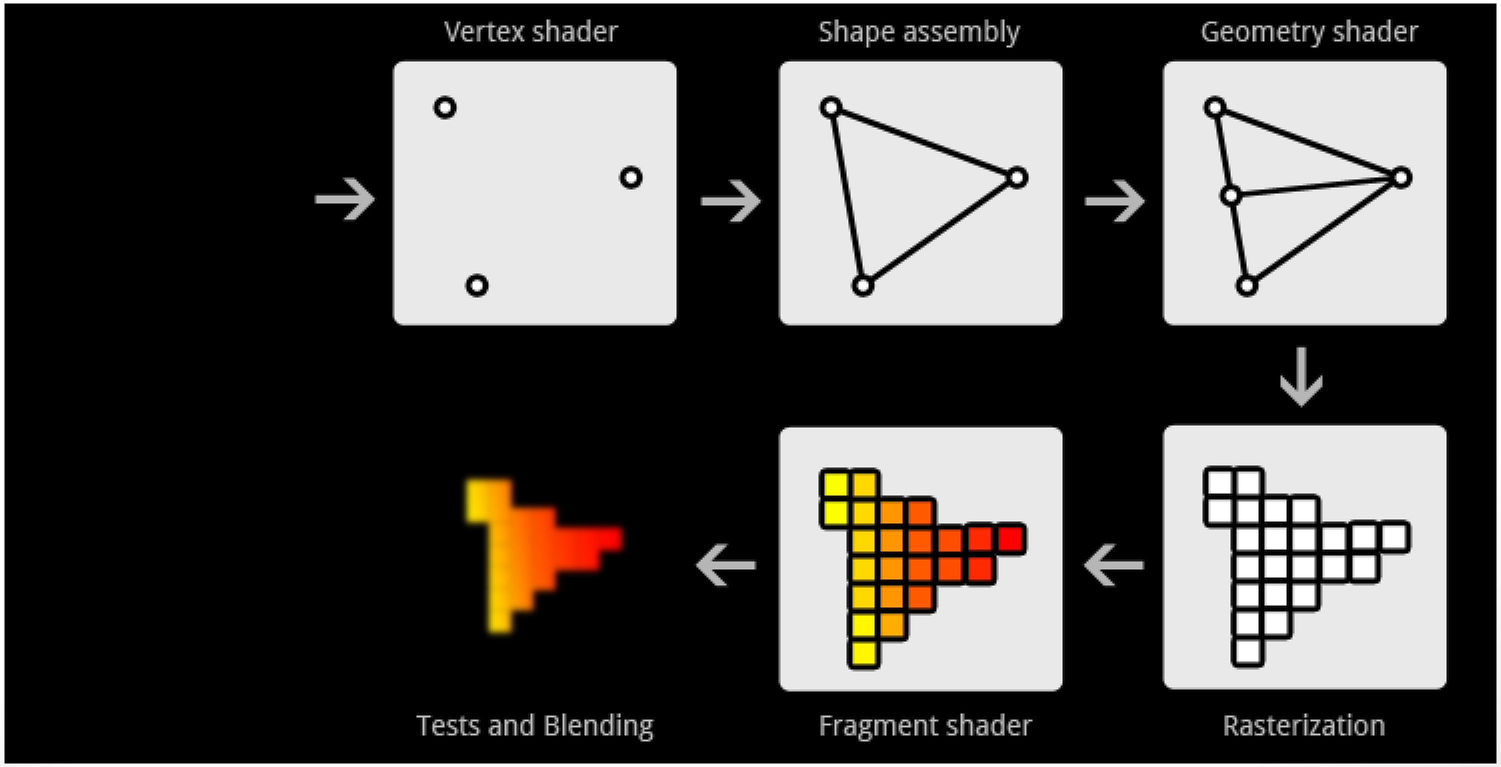


Connected lines

```
/* Draws a 'C' */
glBegin(GL_LINE_STRIP);
        glVertex2f(0.5f, 0.5f);
        glVertex2f(-0.5f, 0.5f);
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(0.5f, -0.5f);
glEnd();
```

polygons using OpenGL

- The graphics pipeline covers all of the steps that follow each other up on processing the input data to get to the final output image. I'll explain these steps with help of the following illustration.



polygons using OpenGL

- It all begins with the *vertices*, these are the points from which shapes like triangles will later be constructed. Each of these points is stored with certain attributes and it's up to you to decide what kind of attributes you want to store. Commonly used attributes are 3D position in the world and texture coordinates.
- The *vertex shader* is a small program running on your graphics card that processes every one of these input vertices individually. This is where the perspective transformation takes place, which projects vertices with a 3D world position onto your 2D screen! It also passes important attributes like color and texture coordinates further down the pipeline.

polygons using OpenGL

- After the input vertices have been transformed, the graphics card will form triangles, lines or points out of them. These shapes are called *primitives* because they form the basis of more complex shapes. There are some additional drawing modes to choose from, like triangle strips and line strips. These reduce the number of vertices you need to pass if you want to create objects where each next primitive is connected to the last one, like a continuous line consisting of several segments.

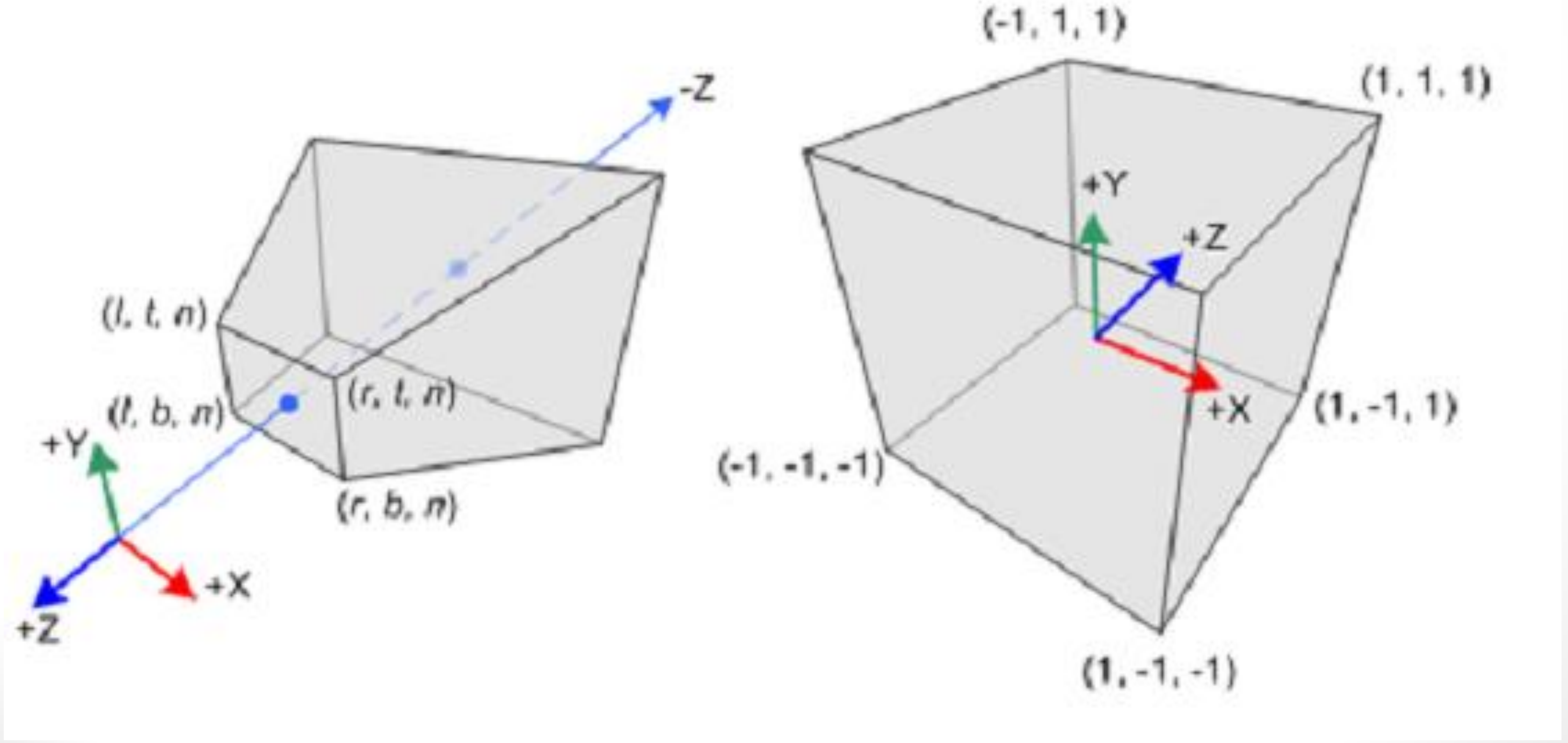
polygons using OpenGL

```
glBegin(GL_POLYGON);
            glVertex2f(0.90 , 0.50);
        glVertex2f(0.50 , 0.90);
        glVertex2f(0.10 , 0.50);
        glVertex2f(0.50 , 0.10);
    glEnd();
```

Viewing and Lighting

- As far as OpenGL is concerned, there is no camera. More specifically, the camera is always located at the eye space coordinate (0.0, 0.0, 0.0). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation by placing it on the MODELVIEW matrix. This is commonly referred to as the viewing transformation.

Viewing and Lighting



Chapter 10 Finished

Best Of luck for your Board Exam