BY: Mukunda Paudel
Paudelmuku@gmail.com

COMPUTER GRAPHICS

Disclaimer

*This document is part of teaching materials for COMPUTER GRAPHICS under the Tribhuvan University syllabus for Bachelors of Science in Computer Science and Information Technology (**BSc. CSIT**).This document does not cover all aspect of learning COMPUTER GRAPHICS, nor are these be taken as primary source of information. As the core textbooks and reference books for learning the subject has already been specified and provided to the students, students are encouraged to learn from the original sources because this document cannot be used as a substitute for prescribed textbooks..*

*Various text books as well as freely available material from internet were consulted for preparing this document. Contents in This document are **copyrighted** to the instructor and authors of original texts where applicable.*

Unit 10: Introduction to OpenGL

OpenGL

- OpenGL is a low-level graphics library specification.
- It makes available to the programmer a small set of geometric primitives - points, lines, polygons, images, and bitmaps.
- OpenGL provides a set of commands immediately executed functions that allow the specification of geometric objects in 2-D or 3-D, using the provided primitives, together with commands that control how these objects are rendered (drawn).
- 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)

Libraries

OpenGL provides a powerful but primitive set of rendering command, and all higher-level drawing must be done in terms of these commands. There are several libraries that allow us to simplify our programming tasks, including the following:

- ***OpenGL Utility Library (GLU)*** contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections and rendering surfaces.
- ***OpenGL Utility Toolkit (GLUT)*** is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window APIs.

Since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), the **OpenGL Utility Toolkit (GLUT)** has been created to aid in the development of more complicated three-dimensional objects such as a sphere, a torus, and even a teapot. GLUT may not be satisfactory for full-featured OpenGL applications, but it is a useful starting point for learning OpenGL.

GLUT is designed to fill the need for a window system independent programming interface for OpenGL programs. The interface is designed to be simple yet still meet the needs of useful OpenGL programs. Removing window system operations from OpenGL is a sound decision because it allows the OpenGL graphics system to be retargeted to various systems including

powerful but expensive graphics workstations as well as mass-production graphics systems like video games, set-top boxes for interactive television, and PCs.

GLUT simplifies the implementation of programs using OpenGL rendering. The GLUT application programming interface (API) requires very few routines to display a graphics scene rendered using OpenGL. The GLUT routines also take relatively few parameters.

Callback Functions in OpenGL

- A callback function is a function which the library (GLUT) calls when it needs to know how to process something.
  For e.g. when glut gets a key down event it uses the glutKeyboardFunc callback routine to find out what to do with a key press.
- 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.
  - ***Menu callback*** is set by the *glutCreateMenu* call described already.
  - ***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.

A callback function is basically a function pointer that you can set that GLFW (Graphics Library Framework) 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);
```

Description:

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.

```
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);
}
```

Description:

In this (newly created) key_callback function checks,

- ✓ 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.

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.
- ❖ For instance, pure Red is represented as (1, 0, 0)
- ❖ Full Blue is (0, 0, 1).
- ❖ Full Green is (0,1,0)
- ❖ White is the combination of all three, denoted (1, 1, 1)
- ❖ 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).

OpenGL has a large collection of functions that can be used to specify colors for the geometry that we draw.

These functions have names of the form ***glColor****, where the "*" stands for a suffix that gives the number and type of the parameters.

For example,

- The function ***glColor3f*** has three parameters of type **float**.
  - The parameters give the red, green, and blue components of the color as numbers in the range 0.0 to 1.0.
  - In fact, values outside this range are allowed, even negative values. When color values are used in computations, out-of-range values will be used as given. When a color actually appears on the screen, its component values are clamped to the range 0 to 1. That is, values less than zero are changed to zero, and values greater than one are changed to one.
- You can add a fourth component to the color by using **glColor4f().**
  - The fourth component, known as alpha, is not used in the default drawing mode, but it is possible to configure OpenGL to use it as the degree of transparency of the color, similarly to the use of the alpha component in the 2D graphics APIs that we have looked at. You need two commands to turn on transparency:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

  - The first command enables use of the alpha component. It can be disabled by calling *glDisable(GL_BLEND)*.
  - When the *GL_BLEND* option is disabled, alpha is simply ignored. The second command tells how the alpha component of a color will be used.
- If you would like to use integer color values in the range 0 to 255, you can use ***glColor3ub() or glColor4ub*** to set the color.
  - In these function names, “ub” stands for “unsigned byte.”
  - ***Unsigned byte*** is an eight-bit data type with values in the range 0 to 255.
- followings are some examples of commands for setting drawing colors in OpenGL Using any of these functions sets the value of a “current color,” which is part of the OpenGL state.

1. **glColor3f(0,0,0);** // Draw in black.

2. **glColor3f(1,1,1);** // Draw in white.

3. **glColor3f(1,0,0);** // Draw in full-intensity red.

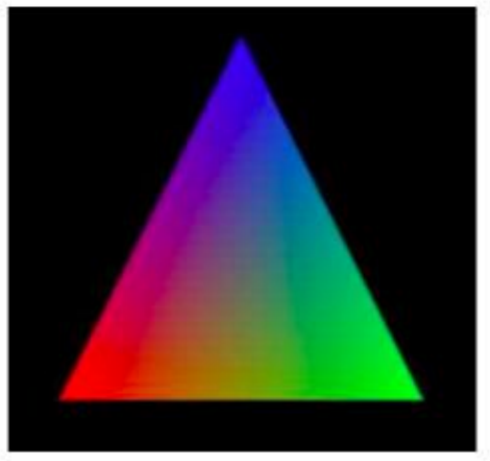
4. **glColor3ub(1,0,0);** // Draw in a color just a tiny bit different from // black. (The suffix, "ub" or "f", is important!)

5. **glColor3ub(255,0,0);** // Draw in full-intensity red.

6. **glColor4f(1, 0, 0, 0.5);** // Draw in transparent red, but only if OpenGL has been configured transparency. By default this is the same as drawing in plain red.

For example,

Here is a triangle in which the three vertices are assigned the colors red, green, and blue.



The triangle can be drawn with the commands:

```
glBegin(GL_TRIANGLES);
glColor3f( 1, 0, 0 ); // red
glVertex2f( -0.8, -0.8 );
glColor3f( 0, 1, 0 ); // green
glVertex2f( 0.8, -0.8 );
glColor3f( 0, 0, 1 ); // blue
glVertex2f( 0, 0.9 );
glEnd();
```

- When drawing a primitive, you do **not** need to explicitly set a color for each vertex, as was done here. If you want a shape that is all one color, you just have to set the current color once, before drawing the shape (or just after the call to *glBegin()*.
  - **For example**, we can draw a solid yellow triangle with

```
glColor3ub(255,255,0); // yellow
glBegin(GL_TRIANGLES);
glVertex2f( -0.5, -0.5 );
glVertex2f( 0.5, -0.5 );
glVertex2f( 0, 0.5 );
glEnd();
```

Drawing Primitives

```
glBegin (mode);
commands…
glEnd;
```

Mode can be..:

GL_POINTS: Individual points

GL_LINES: Pairs of vertices used to draw segments

GL_TRIANGLES: Triple of vertices used to draw triangles

GL_QUADS: Quandruples of vertices used to draw quadrilaterals

GL_POLYGON: Boundary of a polygon

Commands between glBegin() and glEnd():

glVertex*(): Specifies vertex coordinates

glColor*(): Specifies color

Drawings pixels

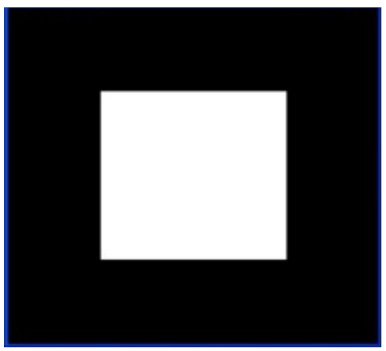
- OpenGL provides only the lowest level of support for drawing strings of characters and manipulating fonts.
- The commands **glRasterPos*()** and **glBitmap()** position and draw a single bitmap on the screen.
- In addition, through the display-list mechanism, you can use a sequence of character codes 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(0.25,0.25);
glVertex2f(0.75,0.75);
glEnd();
```

/* 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();
```



/*Draw Circle*/

```
#define PI 3.14159
GLint CirclePoints = 100;
glBegin(GL_LINE_LOOP);
for(i=0; i<CirclePointss; i++)
{
angle = 2 * PI * i / CirclePoints;
glVertex2f (cos(angle), sin(angle));
}
glEnd();
```

/*Draw Polygon*/

```
glBegin(GL_POLYGON);
 glColor3f (1.0, 0.0, 0.0);
glVertex3f (0.25, 0.25, 0.0);
glVertex3f (0.75, 0.25, 0.0);
glColor3f (0.0, 0.0, 1.0);
glVertex3f (0.75, 0.75, 0.0);
glVertex3f (0.25, 0.75, 0.0);
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.
- OpenGL pipelining is the sequence of steps that OpenGL takes when rendering objects.

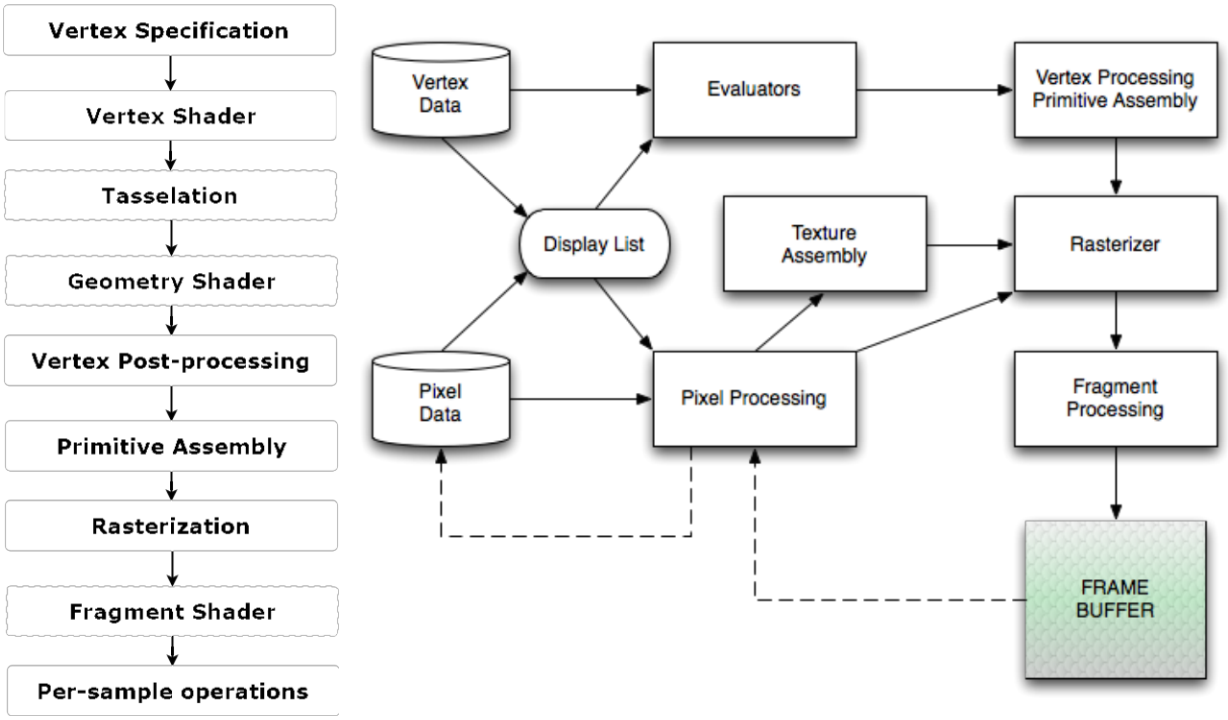


Figure a) Sequential Execution of OpenGL Pipeline **b):** OpenGL Rendering Pipeline

Vertex Specification: List an ordered list of vertices that define the boundaries of the primitive. Along with this, one can define other vertex attributes like color, texture coordinates etc.

Vertex Shader: Vertex Shader is a program that manipulate the vertex data. The ultimate goal of vertex shader is to calculate final vertex position of each vertex. Vertex shaders are executed once for every vertex (e.g. in case of a triangle it will execute 3-times) that the GPU processes.

Tessellation: Optional stage. In this stage primitives are tessellated i.e. divided into smoother mesh of triangles.

Geometry Shader: Optional stage. The work of Geometry Shader is to take an input primitive and generate zero or more output primitive.

Vertex Post Processing: This is a fixed function stage i.e. user has a very limited to no control over these stages. The most important part of this stage is **clipping**. Clipping discards the area of primitives that lie outside the viewing volume.

Primitive Assembly: This stage collects the vertex data into an ordered sequence of simple primitives (lines, points or triangles).

Rasterization: An important step in pipeline. The output of rasterization is a fragments.

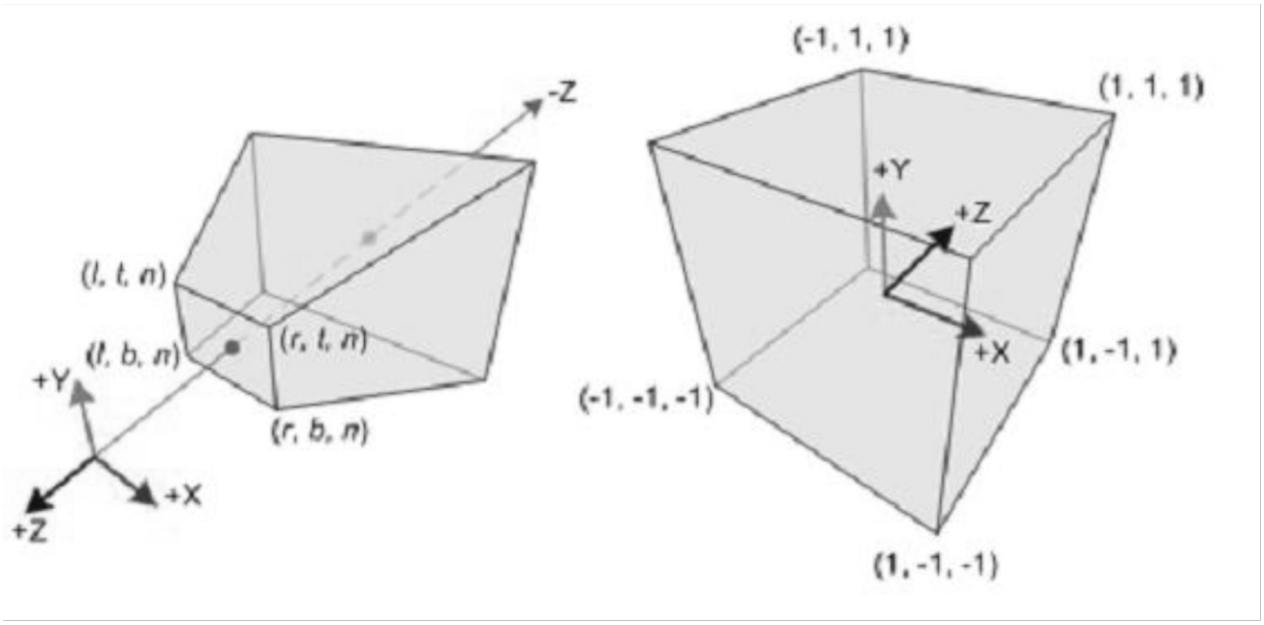
Fragment Shader: The fragment shader runs for each fragment in the geometry. The job of the fragment shader is to determine the final color for each fragment.

Per-sample Operations: There are few tests that are performed based on user has activated them or not. Some of these tests for example are Pixel ownership test, Scissor Test, Stencil Test, Depth Test.

Finally, Geometric data (vertices, line, and polygons) follow a path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images and bitmaps) are treated differently for part of the process. Both types of data undergo the same final step before the final pixel data is written to the frame buffer.

Viewing and Lightening

- 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.



Learn more about OpenGL lightening on https://learnopengl.com/Lighting/Basic-Lighting

** End of Chapter**

Thank you and Good Luck.