LAB MANUAL

Computer Graphics using OpenGL
Department of CSE & IT, JUIT.

Chief Course Coordinator: Dr Rajesh Siddavatam
Associate Professor, CSE& IT

Lab Co-ordinators: Ms Meenakshi S Arya ,
Mr Ravikant Verma
Lecturers, CSE& IT

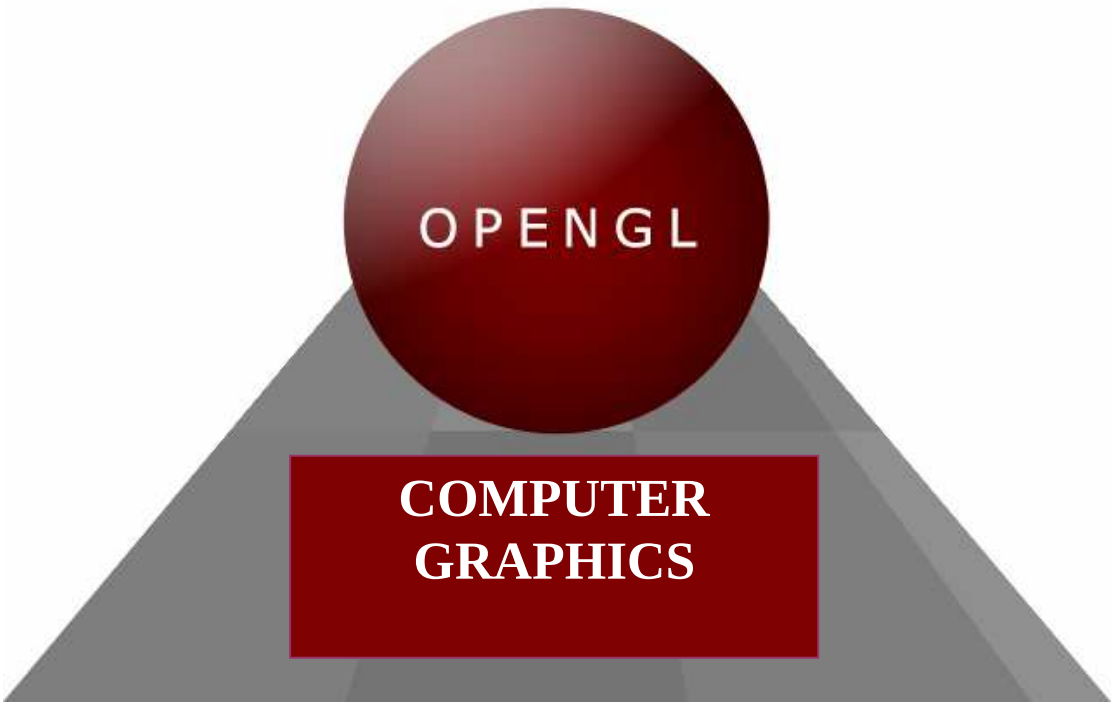


Table of contents

Chapter 5: Drawing Algorithms

- Line Drawing Algorithms
  - Line Slope Intercept
  - DDA
  - Bresenham

- Circle Drawing Algorithms:
  - Direct equation.
  - Midpoint Algorithm.

Chapter 6: 2D Transformation

- Creating Menu
- Rotation , Scale and Translation

Chapter 7: OpenGL Transformation

- OpenGL Transform Matrix
- Example: GL_MODELVIEW Matrix
- Example: GL_PROJECTION Matrix

Chapter 8: 3D Shapes in Open GL

- Cube
- Sphere
- Cone
- Torus
- Teapot

Setting up the VC++ environment for Open GL.

1. Check whether VC++ (Microsoft Visual Studio 6.0) is setup on your machine/system.

2. Open the glut folder from the server.

3. Copy glut.dll, glu32.dll and glut32.dll into C:\WINDOWS\system32.

4. Copy glut32.lib,glut.lib into C:\Program Files\Microsoft Visual Studio\VC98\ into the lib folder.

5. Copy glut.h into C:\Program Files\Microsoft Visual Studio\VC98\ into the INCLUDE folder.

1 Introduction to OpenGL

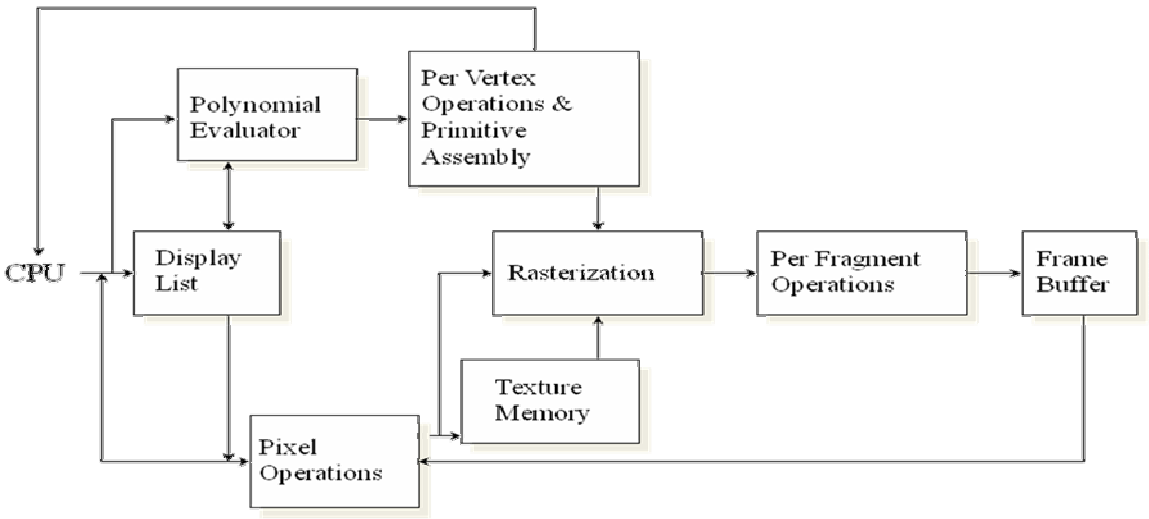
What is OpenGL?

It is a window system independent, operating system independent graphics rendering API which is capable of rendering high-quality color images composed of geometric and image primitives. OpenGL is a library for doing computer graphics. By using it, you can create interactive applications which render high-quality color images composed of 3D geometric objects and images.

As OpenGL is window and operating system independent. As such, the part of your application which does rendering is platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform you're working on. Summarizing the above discussion, we can say OpenGL is a software API to graphics hardware.

- Designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms.
- Procedural interface.
- No windowing commands.
- No high-level commands.

OpenGL Architecture



This important diagram represents the flow of graphical information, as it is processed from CPU to the frame buffer. There are two pipelines of data flow. The upper pipeline is for geometric, vertex-based primitives. The lower pipeline is for pixel-based, image primitives. Texturing combines the two types of primitives together.

OpenGL as a Renderer

OpenGL is a library for rendering computer graphics. Generally, there are two operations that you do with OpenGL:

- draw something
- change the state of how OpenGL draws

OpenGL has two types of things that it can render: geometric primitives and image primitives. *Geometric primitives* are points, lines and polygons. *Image primitives* are bitmaps and graphics images (i.e. the pixels that you might extract from a JPEG image after you've read it into your program.) Additionally, OpenGL links image and geometric primitives together using *texture mapping*, which is an advanced topic we'll discuss this afternoon.

The other common operation that you do with OpenGL is *setting state.* "Setting state" is the process of initializing the internal data that OpenGL uses to render your primitives. It can be as simple as setting up the size of points and color that you want a vertex to be, to initializing multiple mipmap levels for texture mapping.

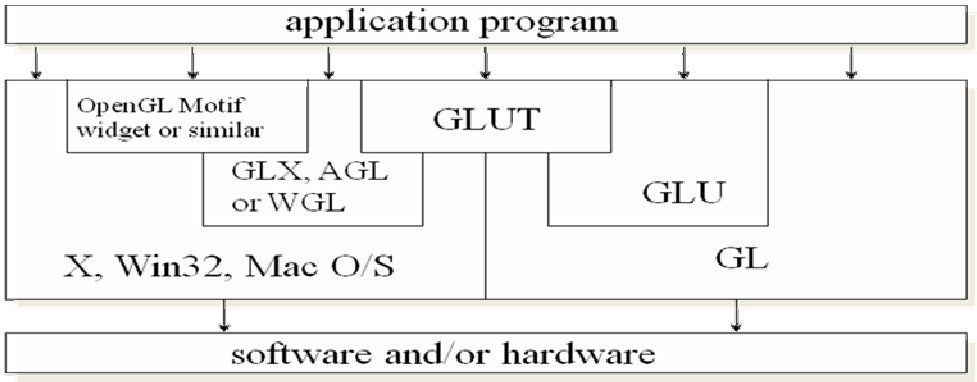
OpenGL and Related APIs

OpenGL is window and operating system independent. To integrate it into various window systems, additional libraries are used to modify a native window into an OpenGL capable window. Every window system has its own unique library and functions to do this. Some examples are:

- GLX for the X Windows system, common on Unix platforms
- AGL for the Apple Macintosh
- WGL for Microsoft Windows
-

OpenGL also includes a utility library, GLU, to simplify common tasks such as: rendering quadric surfaces (i.e. spheres, cones, cylinders, etc.), working with NURBS and curves, and concave polygon tessellation.

Finally to simplify programming and window system dependence, we'll be using the freeware library, GLUT. GLUT, written by Mark Kilgard, is a public domain window system independent toolkit for making simple OpenGL applications. It simplifies the process of creating windows, working with events in the window system and handling animation.



2 GLUT (OpenGL Utility Toolkit)

Introduction

GLUT (pronounced like the glut in gluttony) is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming. GLUT provides a portable API so you can write a single OpenGL program that works across all PC and workstation OS platforms.

GLUT is designed for constructing small to medium sized OpenGL programs. While GLUT is well-suited to learning OpenGL and developing simple OpenGL applications, GLUT is not a full-featured toolkit so large applications requiring sophisticated user interfaces are better off using native window system toolkits. GLUT is simple, easy, and small.

The GLUT library has C, C++ (same as C), FORTRAN, and Ada programming bindings. The GLUT source code distribution is portable to nearly all OpenGL implementations and platforms. The current version is 3.7. Additional releases of the library are not anticipated. GLUT is not open source. Mark Kilgard maintains the copyright. There are a number of newer and open source alternatives.

The toolkit supports:

- Multiple windows for OpenGL rendering
- Callback driven event processing
- Sophisticated input devices
- An 'idle' routine and timers
- A simple, cascading pop-up menu facility
- Utility routines to generate various solid and wire frame objects
- Support for bitmap and stroke fonts

Design Philosophy

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 API (like the OpenGL API) is stateful. Most initial GLUT state is defined and the initial state is reasonable for simple programs.

The GLUT routines also take relatively few parameters. No pointers are returned. The only pointers passed into GLUT are pointers to character strings (all strings passed to GLUT are copied, not referenced) and opaque font handles.

The GLUT API is (as much as reasonable) window system independent. For this reason, GLUT does not return *any* native window system handles, pointers, or other data structures. More subtle window system dependencies such as reliance on window system dependent fonts are avoided by GLUT; instead, GLUT supplies its own (limited) set of fonts.

For programming ease, GLUT provides a simple menu sub-API. While the menuing support is designed to be implemented as pop-up menus, GLUT gives window system leeway to support the menu functionality in another manner (pull-down menus for example).

Two of the most important pieces of GLUT state are the *current window* and *current menu*. Most window and menu routines affect the *current window* or *menu* respectively. Most callbacks implicitly set the *current window* and *menu* to the appropriate window or menu responsible for the callback. GLUT is designed so that a program with only a single window and/or menu will not need to keep track of any window or menu identifiers. This greatly simplifies very simple GLUT programs.

GLUT is designed for simple to moderately complex programs focused on OpenGL rendering. GLUT implements its own event loop. For this reason, mixing GLUT with other APIs that demand their own event handling structure may be difficult. The advantage of a builtin event dispatch loop is simplicity.

GLUT contains routines for rendering fonts and geometric objects, however GLUT makes no claims on the OpenGL display list name space. For this reason, none of the GLUT rendering routines use OpenGL display lists. It is up to the GLUT programmer to compile the output from GLUT rendering routines into display lists if this is desired.

GLUT routines are logically organized into several sub-APIs according to their functionality. The sub-APIs are:

Initialization.
: Command line processing, window system initialization, and initial window creation state are controlled by these routines.

Beginning Event Processing.
: This routine enters GLUT's event processing loop. This routine never returns, and it continuously calls GLUT callbacks as necessary.

Window Management.
: These routines create and control windows.

Overlay Management.
: These routines establish and manage overlays for windows.

Menu Management.
: These routines create and control pop-up menus.

Callback Registration.
: These routines register callbacks to be called by the GLUT event processing loop.

Color Index Colormap Management.
: These routines allow the manipulation of color index colormaps for windows.

State Retrieval.
: These routines allows programs to retrieve state from GLUT.

Font Rendering.
: These routines allow rendering of stroke and bitmap fonts.

Geometric Shape Rendering.

These routines allow the rendering of 3D geometric objects including spheres, cones, icosahedrons, and teapots.

Miscellaneous window management functions

Installing OpenGL Libraries

Installation of OpenGL differs from operating system to another and from compiler to another, because each system like Linux, Win, or Mac has different way of sorting system files, the same issue with compilers, but since we are using Microsoft Visual Studio 98 C++ then we are only going to demonstrate how to install OpenGL on Windows system. To install the OpenGL library all you have to do are the following steps:

Get the OpenGL library or the OpenGL Utility Toolkit Library files, and you can find GLUT on the following links:

a. http://www.xmission/~nate/glut/glut-3.7.6-bin.zip
b. http://www.sultan-eid.com/files/glut-3.7.6-bin.rar

NOTE: Including the glut library will also provide the main GL functions which means you don't have to include the both libraries, you can only include the GLUT and the program will successfully works.

By extracting the compressed files you will find the following:

a. glut32.dll.
b. glut32.lib
c. glut.h
d. README-win32.
e.

In order to write a C application using GLUT you'll need three files:

- glut.h - This is the file you'll have to include in your source code. The common place to put this file is in the *gl* folder which should be inside the include folder of your system.
- glut.lib (SGI version for Windows) and glut32.lib (Microsoft's version) - This file must be linked to your application so make sure to put it your *lib* folder.
- glut32.dll (Windows) and glut.dll (SGI version for Windows) - choose one according to the OpenGL you're using. If using Microsoft's version then you must choose glut32.dll. You should place the dll file in your system folder.

To get the above mentioned files in the correct location, copy each file to the following indicated folder:

- glut32.dll -> c:/windows/system32
- glut32.lib ->c:/program files/Microsoft visual 98 /VC98/Lib
- glut.h ->c:/program files/Microsoft visual 98 /VC98/include

It's important to consider that we are working on Microsoft Visual Studio 98 using C++, in case you are using different compiler or different version of Microsoft Visual Studio, the installation will take different paths.

Now you've completely installed OpenGL library into your machine and you are ready to go and start programming using different graphical functions. In order to write applications with GLUT you should have the latest version. The GLUT distribution comes with lots and lots of examples so after you read through the basics in here you'll have plenty of material to go on.

But before we start to move our fingers lets take a look at the syntax of OpenGL functions.

OpenGL: Conventions

Here’s the basic structure that we’ll be using in our applications. This is generally what you’d do in your own OpenGL applications.
The steps are:

1) Choose the type of window that you need for your application and initialize it.

2) Initialize any OpenGL state that you don’t need to change every frame of your program. This might include things like the background color, light positions and texture maps.

3) Register the *callback* functions that you’ll need. Callbacks are routines you write that GLUT calls when a certain sequence of events occurs, like the window needing to be refreshed, or the user moving the mouse. The most important callback function is the one to render your scene, which we’ll discuss in a few slides.

4) Enter the main event processing loop. This is where your application receives events, and schedules when callback functions are called.

GLUT window and screen coordinates are expressed in pixels. The upper left hand corner of the screen or a window is (0,0). X coordinates increase in a rightward direction; Y coordinates increase in a downward direction. Note: This is inconsistent with OpenGL's coordinate scheme that generally considers the lower left hand coordinate of a window to be at (0,0) but is consistent with most popular window systems. Integer identifiers in GLUT begin with one, not zero. So window identifiers, menu identifiers, and menu item indexes are based from one, not zero. The functions in OpenGL always start with a specific prefix which is "gl" and a capitalized first character of each word in the name of function syntax, which indicates that this function belongs to the OpenGL library.

In GLUT's ANSI C binding, for most routines, basic types (`int`, `char*`) are used as parameters. In routines where the parameters are directly passed to OpenGL routines, OpenGL types (`GLfloat`) are used. The header files for GLUT should be included in GLUT programs with the following include directive:

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

Because a very large window system software vendor (who will remain nameless) has an apparent inability to appreciate that OpenGL's API is independent of their window system API, portable ANSI C GLUT programs should not directly include `<GL/gl.h>` or

<GL/glu.h>. Instead, ANSI C GLUT programs should rely on <GL/glut.h> to include the necessary OpenGL and GLU related header files.

The ANSI C GLUT library archive is typically named libglut.a on Unix systems. GLUT programs need to link with the system's OpenGL and GLUT libraries (and any libraries these libraries potentially depend on). A set of window system dependent libraries may also be necessary for linking GLUT programs. For example, programs using the X11 GLUT implementation typically need to link with Xlib, the X extension library, possibly the X Input extension library, the X miscellaneous utilities library, and the math library.

Examples on OpenGL function formats:

Functions have prefix **gl** and initial capital letters for each word

glClearColor(), glEnable(), glPushMatrix() …

OpenGL data types

GLfloat, GLdouble, GLint, GLenum, …

glutInit

`glutInit` is used to initialize the GLUT library.

Usage

```
void glutInit(int *argcp, char **argv);
```

`argcp`

A pointer to the program's *unmodified* `argc` variable from `main`. Upon return, the value pointed to by `argcp` will be updated, because `glutInit` extracts any command line options intended for the GLUT library.

`argv`

The program's *unmodified* `argv` variable from `main`. Like `argcp`, the data for `argv` will be updated because `glutInit` extracts any command line options understood by the GLUT library.

Description

`glutInit` will initialize the GLUT library and negotiate a session with the window system. During this process, `glutInit` may cause the termination of the GLUT program with an error message to the user if GLUT cannot be properly initialized. Examples of this situation include the failure to connect to the window system, the lack of window system support for OpenGL, and invalid command line options.

`glutInit` also processes command line options, but the specific options parse are window system dependent.

glutInitWindowPosition, glutInitWindowSize

`glutInitWindowPosition` and `glutInitWindowSize` set the *initial window position* and *size* respectively.

Usage

```
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
```
`width`
: Width in pixels.

`height`
: Height in pixels.

`x`
: Window X location in pixels.

`y`
: Window Y location in pixels.

Description

Windows created by `glutCreateWindow` will be requested to be created with the current *initial window position* and *size*.

glutInitDisplayMode

`glutInitDisplayMode` sets the *initial display mode*.

Usage

```
void glutInitDisplayMode(unsigned int mode);
```
`mode`
: Display mode, normally the bitwise *OR*-ing of GLUT display mode bit masks. See values below:

`GLUT_RGBA`
: Bit mask to select an RGBA mode window. This is the default if neither `GLUT_RGBA` nor `GLUT_INDEX` are specified.

`GLUT_RGB`
: An alias for `GLUT_RGBA`.

`GLUT_INDEX`
: Bit mask to select a color index mode window. This overrides `GLUT_RGBA` if it is also specified.

`GLUT_SINGLE`
: Bit mask to select a single buffered window. This is the default if neither `GLUT_DOUBLE` or `GLUT_SINGLE` are specified.

`GLUT_DOUBLE`
: Bit mask to select a double buffered window. This overrides `GLUT_SINGLE` if it is also specified.

`GLUT_ACCUM`
: Bit mask to select a window with an accumulation buffer.

`GLUT_ALPHA`
: Bit mask to select a window with an alpha component to the color buffer(s).

`GLUT_DEPTH`
: Bit mask to select a window with a depth buffer.

`GLUT_STENCIL`
: Bit mask to select a window with a stencil buffer.

`GLUT_MULTISAMPLE`
: Bit mask to select a window with multisampling support. If multisampling is not available, a non-multisampling window will automatically be chosen. Note: both the OpenGL client-side and server-side implementations must support the `GLX_SAMPLE_SGIS` extension for multisampling to be available.

`GLUT_STEREO`

Bit mask to select a stereo window.

`GLUT_LUMINANCE`

Bit mask to select a window with a ``luminance" color model. This model provides the functionality of OpenGL's RGBA color model, but the green and blue components are not maintained in the frame buffer. Instead each pixel's red component is converted to an index between zero and `glutGet(GLUT_WINDOW_COLORMAP_SIZE)-1` and looked up in a per-window color map to determine the color of pixels within the window. The initial colormap of `GLUT_LUMINANCE` windows is initialized to be a linear gray ramp, but can be modified with GLUT's colormap routines.

Description

The *initial display mode* is used when creating top-level windows, subwindows, and overlays to determine the OpenGL display mode for the to-be-created window or overlay.

Note that `GLUT_RGBA` selects the RGBA color model, but it does not request any bits of alpha (sometimes called an *alpha buffer* or *destination alpha*) be allocated. To request alpha, specify `GLUT_ALPHA`. The same applies to `GLUT_LUMINANCE`.

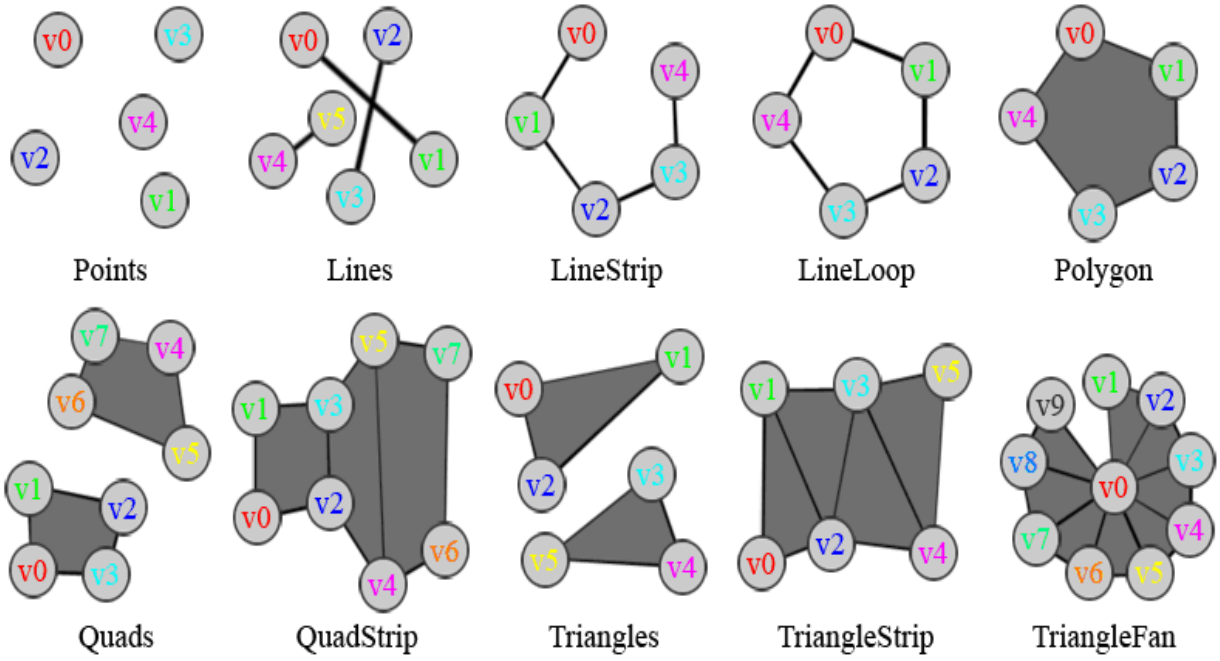
First Single OpenGL Program:

```
/* simple.c second version */
/* This program draws a white rectangle on a black background.*/

#include <glut.h> /* glut.h includes gl.h and glu.h*/
void display()
{
        /* clear window */
        glClear(GL_COLOR_BUFFER_BIT);
        /* draw unit square polygon */
        glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
        glEnd();
        /* flush GL buffers */
        glFlush();
}
void init() // initialize colors
{
        /* set clear color to black */
        glClearColor(0.0, 0.0, 0.0, 0.0);
        /* set fill color to white */
        glColor3f(1.0, 1.0, 1.0);
}

void main(int argc, char** argv)
{
/* Initialize mode and open a window in upper left corner of
```

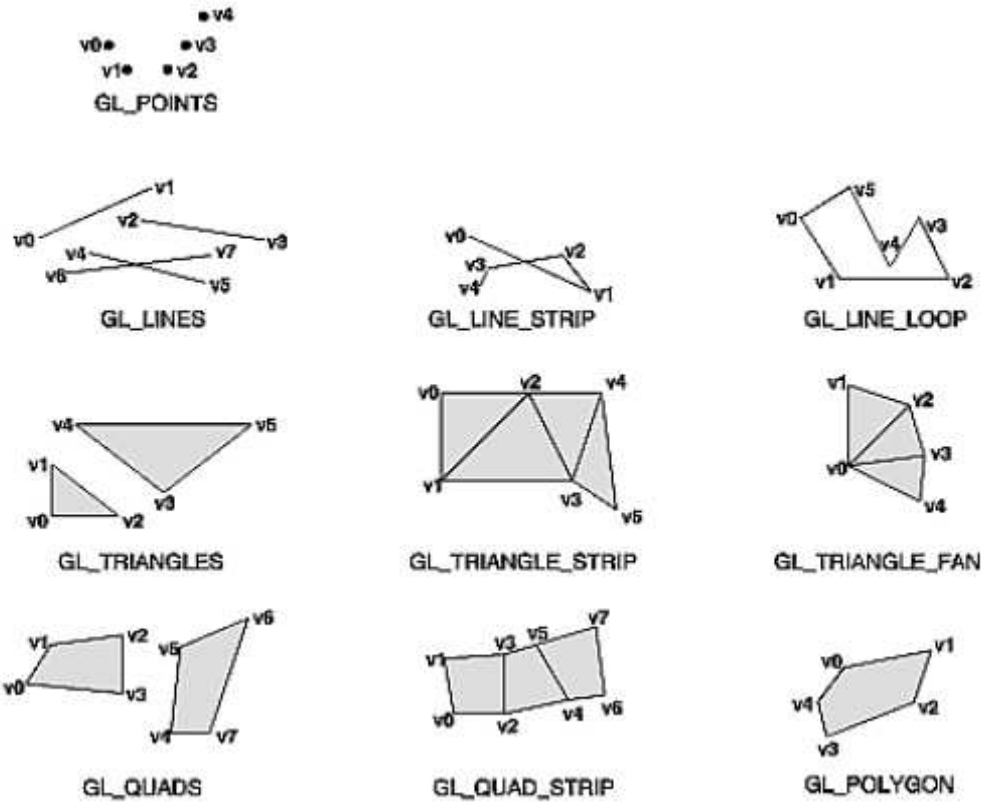
```
        /* screen */
        /* Window title is name of program (arg[0]) */
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500); // Set window Size
        glutInitWindowPosition(0, 0); //Set Window Position
        glutCreateWindow("simple"); //Create Window and Set title
        glutDisplayFunc(display); //Call the Displaying function
        init(); //Initialize Drawing Colors
        glutMainLoop(); //Keep displaying until program is closed.
}
```

3 Drawing Primitives

Geometric Primitive Types in OpenTK.OpenGL (defined Clockwise)



Value	Meaning
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon



A primitive "object" can be anything from a 3D point to a line to a triangle to an n-sided polygon. Each primitive has at least one vertex. With points, vertex is just that - the point. A line has only 2 vertices - its starting point and its ending point. With polygons, there should be more than 2 vertices since polygons are surfaces defined by more or equal to 3 vertices residing on the same plane. A triangle is, for instance, a polygon with 3 vertices. This all should be obvious to you at this point, if you're serious about 3D graphics. Note that a 3D cube cannot be considered a primitive. Generally, primitives restrict themselves to triangles. A four-sided polygon can generate a quad but that quad will still be made out of 2 polygons. Points and lines can also be considered primitives.

glVertex

The main function (and probably the most used OpenGL function) is function named glVertex. This function defines a point (or a vertex) in your 3D world and it can vary from receiving 2 up to 4 coordinates.

glVertex2f(100.0f, 150.0f); defines a point at x = 100, y = 150, z = 0; this function takes only 2 parameters, z is always 0. glVertex2f can be used in special cases and won't be used a lot unless you're working with pseudo-2D sprites or triangles and points that always have to be constrained by the depth coordinate.

glVertex3f(100.0f, 150.0f, -25.0f); defines a point at x = 100, y = 150, z = -25.0f; this function takes 3 parameters, defining a fully 3D point in your world.

glVertex4f(100.0f, 150.0f, -25.0f, 1.0f); this is the same as glVertex3f, the only difference is in the last coordinate that specifies a scaling factor. The scaling factor is set to 1.0f by default. It can be used to make your 3D points look thicker than one pixel.

glBegin and **glEnd**

glVertex alone won't draw anything on the screen, it merely defines a vertex, usually of a more complex object. To really start displaying something on the screen you will have to use two additional functions. These functions are

glBegin(int *mode*); and **glEnd**(*void*);

glBegin and **glEnd** delimit the vertices of a primitive or a group of like primitives. What this means is that everytime you want to draw a primitive on the screen you will first have to call glBegin, specifying what kind of primitive it is that you want to draw in the ***mode*** parameter of glBegin, and then list all vertices one by one (by sequentially calling glVertex) and finally call glEnd to let OpenGL know that you're done drawing a primitive. The parameter mode of the function glBegin can be one of the following:

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUADS
GL_QUAD_STRIP
GL_POLYGON

These flags are self-explanatory. As an example the code given below to shows how to draw some primitives.

```
// this code will draw a point located at [100, 100, -25]
glBegin(GL_POINTS);
glVertex3f(100.0f, 100.0f, -25.0f);
glEnd( );

// next code will draw a line at starting and ending coordinates
specified by glVertex3f
glBegin(GL_LINES);
glVertex3f(100.0f, 100.0f, 0.0f); // origin of the line
glVertex3f(200.0f, 140.0f, 5.0f); // ending point of the line
glEnd( );

// the following code draws a triangle
glBegin(GL_TRIANGLES);
glVertex3f(100.0f, 100.0f, 0.0f);
glVertex3f(150.0f, 100.0f, 0.0f);
```

```
glVertex3f(125.0f, 50.0f, 0.0f);
glEnd( );



// this code will draw two lines "at a time" to save
// the time it takes to call glBegin and glEnd.

glBegin(GL_LINES);
glVertex3f(100.0f, 100.0f, 0.0f); // origin of the FIRST line
glVertex3f(200.0f, 140.0f, 5.0f); // ending point of the FIRST line
glVertex3f(120.0f, 170.0f, 10.0f); // origin of the SECOND line
glVertex3f(240.0f, 120.0f, 5.0f); // ending point of the SECOND line
glEnd( );
```

glVertex is not constrained to be the only function that can be used inside glBegin and glEnd. Here is the full listing of all functions that can be used inside glBegin and glEnd

glVertex
glColor
glIndex
glNormal
glTexCoord
glEvalCoord
glEvalPoint
glMaterial
glEdgeFlag

```
/* hello.c   This is a simple, introductory OpenGL program */
#include <GL/glut.h>
void display(void)
{
glClear (GL_COLOR_BUFFER_BIT);/* clear all pixels  */
glColor3f (1.0, 1.0, 1.0);  /* draw white polygon (rectangle)*/
                      /*with corners at (0.25,0.25,0.0) and
                      /*(0.75, 0.75, 0.0)  */
glBegin(GL_POLYGON);
glVertex3f (0.25, 0.25, 0.0);
glVertex3f (0.75, 0.25, 0.0);
glVertex3f (0.75, 0.75, 0.0);
glVertex3f (0.25, 0.75, 0.0);
glEnd();
glFlush ();   /* start processing buffered OpenGL routines  */
}
void init (void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);/* select clearing color       */
glMatrixMode(GL_PROJECTION);   /* initialize viewing values  */
glLoadIdentity();
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
```

```
/* Declare initial window size, position, and display mode (single buffer
and RGBA). Open window with "hello” in its title bar. Call
initialization routines. Register callback function to display graphics.
Enter main loop and process events. */
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (250, 250);
glutInitWindowPosition (100, 100);
glutCreateWindow ("hello");
init ();
glutDisplayFunc(display);
glutMainLoop();
return 0;
}           /* ANSI C requires main to return int. */
```

Chapter 4: Handling Events

Events	Callback functions	Parameters
Mouse	glutMouseFunc(myMouse)	int button, int state, int x, int y
Mouse Motion	glutMotionFunc(myMotion)	int x , int y
Keyboard	glutKeyboardFunc(myKeyboard)	unsigned char theKey, int mouseX, int mouseY

ANIMATION USING MOUSE

Animation

One of the most exciting things you can do on a graphics computer is draw pictures that move. Whether you're an engineer trying to see all sides of a mechanical part you're designing, a pilot learning to fly an airplane using a simulation, or merely a computer-game aficionado, it's clear that animation is an important part of computer graphics. In a movie theater, motion is achieved by taking a sequence of pictures and projecting them at 24 per second on the screen. Each frame is moved into position behind the lens, the shutter is opened, and the frame is displayed. The shutter is momentarily closed while the film is advanced to the next frame, then that frame is displayed, and so on. Although you're watching 24 different frames each second, your brain blends them all into a smooth animation. (The old Charlie Chaplin movies were shot at 16 frames per second and are noticeably jerky.)

In fact, most modern projectors display each picture twice at a rate of 48 per second to reduce flickering. Computer-graphics screens typically refresh (redraw the picture) approximately 60 to 76 times per second, and some even run at about 120 refreshes per second.

Clearly, 60 per second is smoother than 30, and 120 is marginally better than 60. Refresh rates faster than 120, however, are beyond the point of diminishing returns, since the human eye is only so good.The key reason that motion picture projection works is that each frame is complete when it is displayed. Suppose you try to do computer animation of your million-frame movie with a program like this:

```
open_window();
for (i = 0; i < 1000000; i++) {
clear_the_window();
draw_frame(i);
wait_until_a_24th_of_a_second_is_over();
}
```

If you add the time it takes for your system to clear the screen and to draw a typical frame, this program gives more and more disturbing results depending on how close to 1/24 second

it takes to clear and draw. Suppose the drawing takes nearly a full 1/24 second. Items drawn first are visible for the full 1/24 second and present a solid image on the screen; items drawn toward the end are instantly cleared as the program starts on the next frame. They present at best a ghostlike image, since for most of the 1/24 second your eye is viewing the cleared background instead of the items that were unlucky enough to be drawn last. The problem is that this program doesn't display completely drawn frames; instead, you watch the drawing as it happens. Most OpenGL implementations provide double-buffering - hardware or software that supplies two complete color buffers. One is displayed while the other is being drawn. When the drawing of a frame is complete, the two buffers are swapped, so the one that was being viewed is now used for drawing, and vice versa. This is like a movie projector with only two frames in a loop; while one is being projected on the screen, an artist is desperately erasing and redrawing the frame that's not visible. As long as the artist is quick enough, the viewer notices no difference between this setup and one where all the frames are already drawn and the projector is simply displaying them one after the other.

With double-buffering, every frame is shown only when the drawing is complete; the viewer never sees a partially drawn frame. A modified version of the preceding program that does display smoothly animated graphics might look like this:

```
open_window_in_double_buffer_mode();
for (i = 0; i < 1000000; i++) {
clear_the_window();
draw_frame(i);
swap_the_buffers();
}
```

The Refresh That Pauses

For some OpenGL implementations, in addition to simply swapping the viewable and drawable buffers, the **swap_the_buffers()** routine waits until the current screen refresh period is over so that the previous buffer is completely displayed. This routine also allows the new buffer to be completely displayed, starting from the beginning. Assuming that your system refreshes the display 60 times per second, this means that the fastest frame rate you can achieve is 60 frames per second (*fps*), and if all your frames can be cleared and drawn in under 1/60 second, your animation will run smoothly at that rate. What often happens on such a system is that the frame is too complicated to draw in 1/60 second, so each frame is displayed more than once. If, for example, it takes 1/45 second to draw a frame, you get 30 fps, and the graphics are idle for 1/30-1/45=1/90 second per frame, or one-third of the time.

In addition, the video refresh rate is constant, which can have some unexpected performance consequences. For example, with the 1/60 second per refresh monitor and a constant frame rate, you can run at 60 fps, 30 fps, 20 fps, 15 fps, 12 fps, and so on (60/1, 60/2, 60/3, 60/4, 60/5, ...). That means that if you're writing an application and gradually adding features (say it's a flight simulator, and you're adding ground scenery), at first each feature you add has no effect on the overall performance - you still get 60 fps. Then, all of a sudden, you add one new feature, and the system can't quite draw the whole thing in 1/60 of a second, so the animation slows from 60 fps to 30 fps because it misses the first possible buffer-swapping time. A similar thing happens when the drawing time per frame is more than 1/30 second - the animation drops from 30 to 20 fps. If the scene's complexity is close to any of the magic times (1/60 second, 2/60 second, 3/60 second, and so on in this example), then because of random variation, some frames go slightly over the time and some slightly under. Then the frame rate is irregular, which can be visually disturbing. In this case, if you can't simplify the

scene so that all the frames are fast enough, it might be better to add an intentional, tiny delay to make sure they all miss, giving a constant, slower, frame rate. If your frames have drastically different complexities, a more sophisticated approach might be necessary.

Motion = Redraw + Swap

The structure of real animation programs does not differ too much from this description. Usually, it is easier to redraw the entire buffer from scratch for each frame than to figure out which parts require redrawing. This is especially true with applications such as three-dimensional flight simulators where a tiny change in the plane's orientation changes the position of everything outside the window.

In most animations, the objects in a scene are simply redrawn with different transformations the viewpoint of the viewer moves, or a car moves down the road a bit, or an object is rotated slightly. If significant recomputation is required for non-drawing operations, the attainable frame rate often slows down. Keep in mind, however, that the idle time after the **swap_the_buffers()** routine can often be used for such calculations.
OpenGL doesn't have a **swap_the_buffers()** command because the feature might not be available on all hardware and, in any case, it's highly dependent on the window system.

If you are using the GLUT library, you'll want to call this routine:
void glutSwapBuffers(void);

Example 4-1 illustrates the use of **glutSwapBuffers()** in an example that draws a spinning square as shown in Figure 4.1.

The following example also shows how to use GLUT to control an input device and turn on and off an idle function.

In this example, the mouse buttons toggle the spinning on and off.

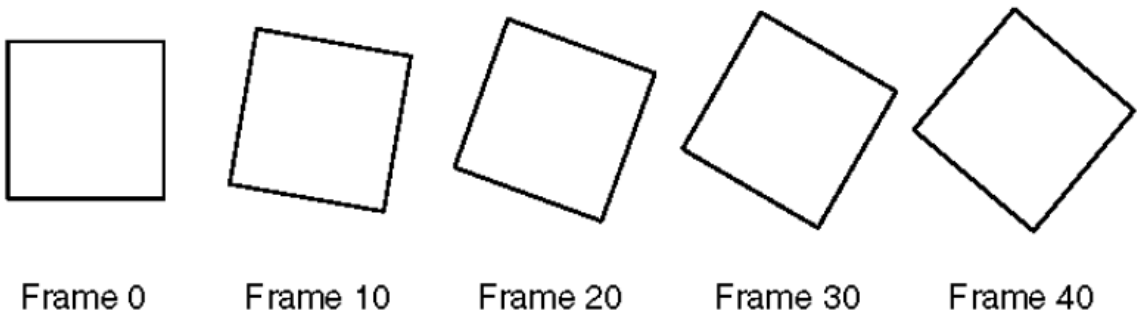


Figure 4.1 : Double-Buffered Rotating Square

Example 4-1 : Double-Buffered Program:

```
#include <GL/glut.h>
#include <stdlib.h>
static GLfloat spin = 0.0;
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glPushMatrix();
glRotatef(spin, 0.0, 0.0, 1.0);
glColor3f(1.0, 1.0, 1.0);
glRectf(-25.0, -25.0, 25.0, 25.0);
glPopMatrix();
glutSwapBuffers();
}
void spinDisplay(void)
{
spin = spin + 2.0;
if (spin > 360.0)
spin = spin - 360.0;
glutPostRedisplay();
}
void reshape(int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
switch (button) {
case GLUT_LEFT_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(spinDisplay);
break;
case GLUT_MIDDLE_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(NULL);
break;
```

```
default:
break;
}
}
/*
* Request double buffer display mode.
* Register mouse input callback functions
*/
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize (250, 250);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();
return 0;

}
```

5 Drawing Algorithms

Line Drawing Algorithms

Line Slope Intercept (LSI) Algorithm

```
// Program to draw a line using Line Slope Intercept Algorithm

#include <iostream.h>
#include <glut.h>

GLdouble x1, y1, x2, y2; // Decleration of the start and end POINTS

void LineSI(void) // Line Slope Intercept (LSI) function
{
          GLdouble dx = x2-x1;
          GLdouble dy = y2-y1;
          GLdouble m = dy/dx;
          GLdouble b = y1-m*x1;
          glClear(GL_COLOR_BUFFER_BIT);
          glBegin(GL_POINTS);
          if(m<=1)
          {
                for(GLdouble x = x1; x <= x2 ; x++)
                {
                      GLdouble y = m*x+b;
                      glVertex2d(x,y);
                }
          }
          else
          {
                for(GLdouble y = y1 ; y <= y2 ; y++)
                {
                      GLdouble x = (1/m)*y+b;
                      glVertex2d(x,y);
                }
          }
          glEnd();
          glFlush();
}

void Init()
{
          glClearColor(1.0,1.0,1.0,0);
          glColor3f(0.0,0.0,0.0);
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          gluOrtho2D(0 , 640 , 0 , 480);
}




void main(int argc, char **argv)
```

```
{
          cout<<"Enter Two Points for Draw LineSlopeIntercept:\n";
          cout<<"\nEnter Point1( x1 , y1):";
          cin>>x1>>y1;
          cout<<"\nEnter Point2( x2 , y2):";
          cin>>x2>>y2;


          glutInit(&argc,argv);
          glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
          glutInitWindowSize(640,480);
          glutInitWindowPosition(0,0);
          glutCreateWindow("LineSlopeIntercept");
          Init();
          glutDisplayFunc(LineSI);
          glutMainLoop();

}
```

Digital Differential Analyzer (DDA) Line Algorithm

```
// Program to draw a line using DDA Algorithm

#include <iostream.h>
#include <math.h>
#include <glut.h>

GLdouble X1, Y1, X2, Y2;

void LineDDA(void)
{
          GLdouble dx=X2-X1 , dy=Y2-Y1,steps;
          float xInc,yInc,x=X1,y=Y1;
          steps=(abs(dx)>abs(dy))?abs(dx):abs(dy);
          xInc=dx/(float)steps;
          yInc=dy/(float)steps;
          glClear(GL_COLOR_BUFFER_BIT);
                glBegin(GL_POINTS);
                      glVertex2d(x,y);
                      for(int k=0;k<steps;k++)
                      {
                            x+=xInc;
                            y+=yInc;
                            glVertex2d(x,y);
                      }
                glEnd();
          glFlush();
}

void Init()
{
          glClearColor(1.0,1.0,1.0,0);
          glColor3f(0.0,0.0,0.0);
          glViewport(0 , 0 , 640 , 480);
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          gluOrtho2D(0 , 640 , 0 , 480);
```

```
}

void main(int argc, char **argv)
{
          cout<<"Enter Two Points for Draw LineDDA:\n";
          cout<<"\nEnter Point1( X1 , Y1):";
          cin>>X1>>Y1;
          cout<<"\nEnter Point2( X2 , Y2):";
          cin>>X2>>Y2;


          glutInit(&argc,argv);
          glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
          glutInitWindowSize(640,480);
          glutInitWindowPosition(0,0);
          glutCreateWindow("LineDDA");
          Init();
          glutDisplayFunc(LineDDA);
          glutMainLoop();
}
```

Bresenham's Line Algorithm

Bresenham's line algorithm properties:

- Only uses incremental integer calculations.
- Can be adapted to display circles and other curves.
- Basic idea find next pixel from current one

```
// Program to draw a line using Bresenham's Algorithm

#include <iostream.h>
#include <math.h>
#include <glut.h>

GLdouble X1, Y1, X2, Y2;
void LineBres (void)
{
          glClear(GL_COLOR_BUFFER_BIT);
          int dx = abs (X1 - X2), dy = abs (Y1 - Y2);
          int p = 2 * dy - dx;
          int twoDy = 2 * dy, twoDyDx = 2 * (dy - dx);
          int x, y, xEnd;

          if (X1 > X2)
          {
                x = X2;
                y = Y2;
                xEnd = X1;
          }




else
```

```
	{
		x = X1;
		y = Y1;
		xEnd = X2;
	}

	glBegin(GL_POINTS);
		glVertex2d(x,y);
		while (x < xEnd)
		{
			x++;
			if (p < 0)
				p += twoDy;
			else
			{
				y++;
				p += twoDyDx;
			}
			glVertex2d(x,y);
		}
	glEnd();

	glFlush();
}



void Init()
{
	//glClearColor(1.0,1.0,1.0,0);
	//glColor3f(0.0,0.0,0.0);
	glViewport(0 , 0 , 640 , 480);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0 , 640 , 0 , 480);
}


int main(int argc, char **argv)
{
	cout<<"Enter Two Points for Draw LineBresenham:\n";
	cout<<"\nEnter Point1( X1 , Y1):";
	cin>>X2>>Y2;
	cout<<"\nEnter Point2( X2 , Y2):";
	cin>>X2>>Y2;

	glutInit(&argc,argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
	glutInitWindowSize(640,480);
	glutInitWindowPosition(0,0);
	glutCreateWindow("LineBresenham");
	Init();
	glutDisplayFunc(LineBres);

	glutMainLoop();
	return 0;
}
```

Circle Drawing Algorithms

Direct equation

```
// Draws a circle with a radius R and centered at (Xc,Yc)using the
// direct substitution in the circle equation y = Yc +/- sqrt( R^2 -
//(x-Xc)^2 )

#include <iostream.h>        // for cin, cout
#include <math.h>            // for the "abs" function
#include <glut.h>      // include the OpenGL library



GLdouble R, Xc, Yc;    // the Radius and the center point

// draw a circle using the direct method
void CircleDirect(void)
{
     GLdouble x,y;
     glClear(GL_COLOR_BUFFER_BIT);
     glBegin(GL_POINTS);
           for(x = -R; x <= R; x++)
           {
                 y= sqrt(R * R - (x * x));
                 glVertex2i(Xc+x,Yc+y);        // The upper half
                 glVertex2i(Xc+x,Yc-y);        // the lower half
           }
     glEnd();
     glFlush();


}
//setup the windows parameters
void Init()
{
     glClearColor(1.0,1.0,1.0,0);  // set the background color
(white)
     glColor3f(0.0,0.0,0.0);             // set the foreground color
(black)
     glViewport(0 , 0 , 640 , 480);      // set the viowport
dimensions (all the window)
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     gluOrtho2D(0 , 640 , 0 , 480);      // set the window dimension
(640x480)
}

//The main function
void main(int argc, char** argv)
{
     //Read the circle radius and the center
     cout<<"Enter the radius R :\n";
     cout<<"\nEnter R: ";
     cin>>R;
     cout<<"\nEnter center( Xc , Yc): ";
     cin>>Xc>>Yc;
```

```
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
      glutInitWindowSize(640,480);
      glutInitWindowPosition(100, 150);
      glutCreateWindow("A Circle with a direct method");
      Init();
      glutDisplayFunc(CircleDirect);
      glutMainLoop();
}
```

Midpoint Algorithm

```
// Draws a circle with a radius R and centered at (Xc,Yc) using the
// Midpoint algorithm

#include <glut.h>
#include <iostream.h>

int Xc, Yc, R;  // circle center and radius

void myInit(void)
 {
    glClearColor(1.0,1.0,1.0,0);
    glColor3f(1.0, 0.0, 0.0);
           glPointSize(1.0);
           glMatrixMode(GL_PROJECTION);
           glLoadIdentity();
           gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}

//Draws the 8 symmetric points
void Circle8Points(int x, int y)
 {
            glVertex2i(Xc + x, Yc  + y);
            glVertex2i(Xc - x, Yc  + y);
            glVertex2i(Xc + x, Yc  - y);
            glVertex2i(Xc - x, Yc  - y);
            glVertex2i(Xc + y, Yc  + x);
            glVertex2i(Xc - y, Yc  + x);
            glVertex2i(Xc + y, Yc  - x);
            glVertex2i(Xc - y, Yc  - x);

 }

//Implement the Midpoint algorithm
void CircleMidpoint(void)
 {
            int x=0;
            int y=R;
            int p=1-R;       //initial value of the decision parameter

            glClear(GL_COLOR_BUFFER_BIT);

            glBegin(GL_POINTS);
                 glVertex2i(Xc,Yc);            //Draw circle center for
illustration
                 Circle8Points(x,y);/* Plot the First point */
```

```
                while(x<y)
                {
                      x++;
                      if(p<0)
                            p +=2 * x + 1;
                      else
                      {
                            y--;
                            p += 2 * (x -y) + 1;
                      }


Circle8Points(x,y);
                }
            glEnd();
            glFlush();
 }
//Main function
void main(int argc, char** argv)
{
           cout << " Enter the circle center ( Xc , Yc ):";
           cin >> Xc >> Yc ;
           cout << " Enter the Radius R= ";
           cin >> R;

           glutInit(&argc, argv);
           glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
           glutInitWindowSize(640,480);
           glutInitWindowPosition(100, 150);
           glutCreateWindow("Circle Midpoint");
           myInit();
           glutDisplayFunc(CircleMidpoint);
           glutMainLoop();
}
```

6 Transformation

Geometric image transformation functions use mathematical transformations to crop, pad, scale, rotate, transpose or otherwise alter an image array to produce a modified view of an image. A transformation thus is the process of mapping points to other locations. Common transformations are Translation, Scaling and Rotation.

When an image undergoes a geometric transformation, some or all of the pixels within the source image are relocated from their original spatial coordinates to a new position in the output image. When a relocated pixel does not map directly onto the center of a pixel location, but falls somewhere in between the centers of pixel locations, the pixel's value is computed by sampling the values of the neighboring pixels.

Basic 2D Transforms

Scaling

This transform can change length and possibly direction of a vector

$$scale(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

A vector with Cartesian coordinates (*x,y*) is transformed as

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x & 0 \\ 0 & s_y y \end{bmatrix}$$

Rotation

In matrix form, the equivalent transformation that takes *a* to *b* is

$$rotate(\varphi) = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}$$

For example a matrix that rotates vectors by π/4 radians (45 degrees) is

$$\begin{bmatrix} \cos\pi/4 & -\sin\pi/4 \\ \sin\pi/4 & \cos\pi/4 \end{bmatrix} = \begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix}$$

Shearing

A shear is something that pushes things sideways
The horizontal and vertical shear matrices are

$$shear_x(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \qquad shear_y(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

Transformations

```
// Program to implement basic transformations

#include <glut.h>

void display()
{
      /* clear window */

      glFlush();
            glClear(GL_COLOR_BUFFER_BIT);
}

void OptionsMenu (GLint selectedOpt)
{
      switch (selectedOpt)
      {
      case 1:
                  /* draw unit square polygon */
      glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_POLYGON);
      glVertex2f(100, 150);
      glVertex2f(150, 200);
      glVertex2f(200, 300);
      glVertex2f(400, 100);
      glEnd();
      /* flush GL buffers */
      glFlush();

            break;
      case 2:
                  glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_LINES);
      glVertex2f(100, 100);
      glVertex2f(200, 200);
            glEnd();
      /* flush GL buffers */
      glFlush();

            break;
      case 3:
            glRotatef(45,450,450,450);
            break;
```

```
case 4:
            glScalef(1.5,1.5,0);
            break;




      case 5:
            glTranslatef(2,2,0);
      default: break;
      }


}
void init()
{

    glClearColor(1.0,1.0,1.0,0.0);       // set white background color
    glColor3f(0.0f, 0.0f, 0.0f);          // set the drawing color
      glPointSize(1.0);              // a 'dot' is 4 by 4 pixels
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0.0, 800.0, 0.0, 800.0);
}


void main(int argc, char** argv)
{
/* Initialize mode and open a window in upper left corner of
      /* screen */
      /* Window title is name of program (arg[0]) */
      glutInit(&argc,argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
      glutInitWindowSize(800, 800);
      glutInitWindowPosition(0, 0);
      glutCreateWindow("simple");
      glutDisplayFunc(display);
      glutCreateMenu (OptionsMenu);
      glutAddMenuEntry("Polygon ",1);
      glutAddMenuEntry("Line ",2);
            glutAddMenuEntry("Rotate ",3);
                  glutAddMenuEntry("Scale",4);
                        glutAddMenuEntry("Translate ",5);
      glutAttachMenu (GLUT_RIGHT_BUTTON);
      init();
      glutMainLoop();
}
```

Menu Driven Program

```
//A menu driven program

#include <glut.h> /* glut.h includes gl.h and glu.h*/
void display()
{
      /* clear window */

      glFlush();
}

void OptionsMenu (GLint selectedOpt)
{
      switch (selectedOpt)
      {
      case 1:
                  /* draw unit square polygon */
            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();
      /* flush GL buffers */
      glFlush();

            break;
      case 2:
                  glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_LINES);
      glVertex2f(-0.5, -0.5);
      glVertex2f(-0.5, 0.5);
            glEnd();
      /* flush GL buffers */
      glFlush();

            break;
      default: break;
      }


}
void init()
{
      /* set clear color to black */
      glClearColor(0.0, 0.0, 0.0, 0.0);
      /* set fill color to white */
      glColor3f(1.0, 1.0, 1.0);
}


void main(int argc, char** argv)
{
/* Initialize mode and open a window in upper left corner of
      /* screen */
      /* Window title is name of program (arg[0]) */
```

```
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(0, 0);
        glutCreateWindow("simple");
        glutDisplayFunc(display);
        glutCreateMenu (OptionsMenu);
        glutAddMenuEntry("Polygon ",1);
        glutAddMenuEntry("Line ",2);
        glutAttachMenu (GLUT_RIGHT_BUTTON);
        init();
        glutMainLoop();
}
```

7 OpenGL Transformation

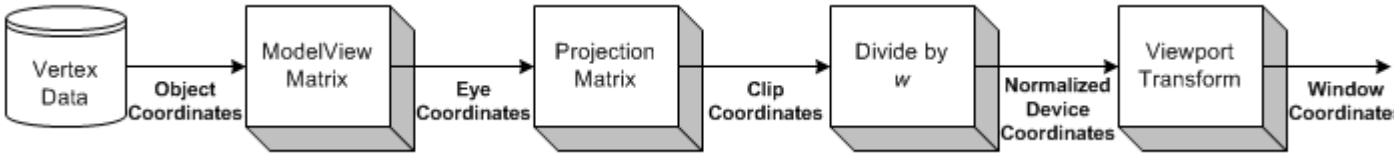
Related Topics: OpenGL Pipeline, OpenGL Projection Matrix, Homogeneous Coordinates

Overview

- OpenGL Transform Matrix
- Example: GL_MODELVIEW Matrix
- Example: GL_PROJECTION Matrix

Overview

Geometric data such as vertex positions and normal vectors are transformed via **Vertex Operation** and **Primitive Assembly** operation in OpenGL pipeline before raterization process.



OpenGL vertex transformation

Object Coordinates

It is the local coordinate system of objects and is initial position and orientation of objects before any transform is applied. In order to transform objects, use glRotatef(), glTranslatef(), glScalef().

Eye Coordinates

It is yielded by multiplying GL_MODELVIEW matrix and object coordinates. Objects are transformed from object space to eye space using GL_MODELVIEW matrix in OpenGL. **GL_MODELVIEW** matrix is a combination of Model and View matrices ($M_{\mathbf{view}} \cdot M_{\mathbf{model}}$). Model transform is to convert from object space to world space. And, View transform is to convert from world space to eye space.

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelView} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

Note that there is no separate camera (view) matrix in OpenGL. Therefore, in order to simulate transforming the camera or view, the scene (3D objects and lights) must be transformed with the inverse of the view transformation. In other words, OpenGL

defines that the camera is always located at (0, 0, 0) and facing to -Z axis in the eye space coordinates, and cannot be transformed. *See more details of*

GL_MODELVIEW matrix in ModelView Matrix.

Normal vectors are also transformed from object coordinates to eye coordinates for lighting calculation. Note that normals are transformed in different way as vertices do. It is mutiplying the tranpose of the inverse of GL_MODELVIEW matrix by a normal vector.

$$\begin{pmatrix} nx_{eye} \\ ny_{eye} \\ nz_{eye} \\ nw_{eye} \end{pmatrix} = \left(M_{modelView}{}^{-1}\right)^{T} \cdot \begin{pmatrix} nx_{obj} \\ ny_{obj} \\ nz_{obj} \\ nw_{obj} \end{pmatrix}$$

Clip Coordinates

It is after applying eye coordinates into **GL_PROJECTION** matrix. Objects are clipped out from the viewing volume (frustum). Frustum is used to determine how objects are projected onto screen (perspective or orthogonal) and which objects or portions of objects are clipped out of the final image.

GL_PROJECTION matrix in Projection Matrix.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

Normalized Device Coordinates (NDC)

It is yielded by dividing the clip coordinates by *w*. It is called *perspective division*. It is more like window (screen) coordinates, but has not been translated and scaled to screen pixels yet. The range of values is now normalized from -1 to 1 in all 3 axes.

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

Window Coordinates (Screen Coordinates)

It is yielded by applying normalized device coordinates (NDC) to viewport transformation. The NDC are scaled and translated in order to fit into the rendering screen. The window coordinates finally are passed to the raterization process of OpenGL pipeline to become a fragment. **glViewport()** command is used to define the rectangle of the rendering area where the final image is mapped. And, **glDepthRange()** is used to determine the *z* value of the window coordinates. The window coordinates are computed with the given parameters of the above 2 functions;

glViewport(x, y, w, h);

glDepthRange(n, f);

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{w}}{2} x_{ndc} + (\mathrm{x} + \frac{\mathrm{w}}{2}) \\ \frac{\mathrm{h}}{2} y_{ndc} + (\mathrm{y} + \frac{\mathrm{h}}{2}) \\ \frac{\mathrm{f}-\mathrm{n}}{2} z_{ndc} + \frac{\mathrm{f}+\mathrm{n}}{2} \end{pmatrix}$$

The viewport transform formula is simply acquired by the linear relationship between NDC and the window coordinates;

$$\begin{cases} -1 & \to \mathrm{x} \\ 1 & \to \mathrm{x} + \mathrm{w} \end{cases} \quad \begin{cases} -1 & \to \mathrm{y} \\ 1 & \to \mathrm{y} + \mathrm{h} \end{cases} \quad \begin{cases} -1 & \to \mathrm{n} \\ 1 & \to \mathrm{f} \end{cases}$$

OpenGL Transformation Matrix

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

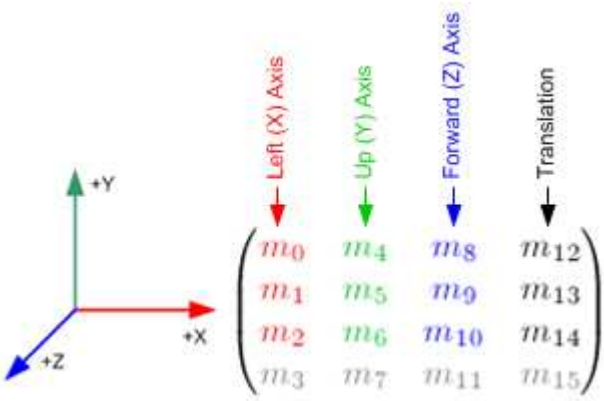
OpenGL Transform Matrix

OpenGL uses 4 x 4 matrix for transformations. Notice that 16 elements in the matrix are stored as 1D array in column-major order. You need to transpose this matrix if you want to convert it to the standard convention, row-major format.

OpenGL has 4 different types of matrices; **GL_MODELVIEW**, **GL_PROJECTION**, **GL_TEXTURE**, and **GL_COLOR**. You can switch the current type by using **glMatrixMode()** in your code. For example, in order to select GL_MODELVIEW matrix, use **glMatrixMode(GL_MODELVIEW)**.

Model-View Matrix (GL_MODELVIEW)

GL_MODELVIEW matrix combines viewing matrix and modeling matrix into one matrix. In order to transform the view (camera), you need to move whole scene with the inverse transformation. **gluLookAt()** is particularly used to set viewing transform.



4 columns of GL_MODELVIEW matrix

The 3 matrix elements of the rightmost column (m_{12}, m_{13}, m_{14}) are for the translation transformation, **glTranslatef()**. The element m_{15} is the homogeneous coordinate. It is specially used for projective transformation.

3 elements sets, (m_0, m_1, m_2), (m_4, m_5, m_6) and (m_8, m_9, m_{10}) are for Euclidean and affine transformation, such as rotation **glRotatef()** or scaling **glScalef()**. Note that these 3 sets are actually representing 3 orthogonal axes;

- (m_0, m_1, m_2) : +X axis, *left* vector, (1, 0, 0) by default
- (m_4, m_5, m_6) : +Y axis, *up* vector, (0, 1, 0) by default
- (m_8, m_9, m_{10}) : +Z axis, *forward* vector, (0, 0, 1) by default

We can directly construct GL_MODELVIEW matrix from angles or lookat vector without using OpenGL transform functions. Here are some useful codes to build GL_MODELVIEW matrix:

- Angles to Axes
- Lookat to Axes

Note that OpenGL performs matrices multiplications in reverse order if multiple transforms are applied to a vertex. For example, If a vertex is transformed by M_A first, and transformed by M_B second, then OpenGL performs M_B x M_A first before multiplying the vertex. So, the last transform comes first and the first transform occurs last in your code.

$$v' = M_B \cdot (M_A \cdot v) = (M_B \cdot M_A) \cdot v$$

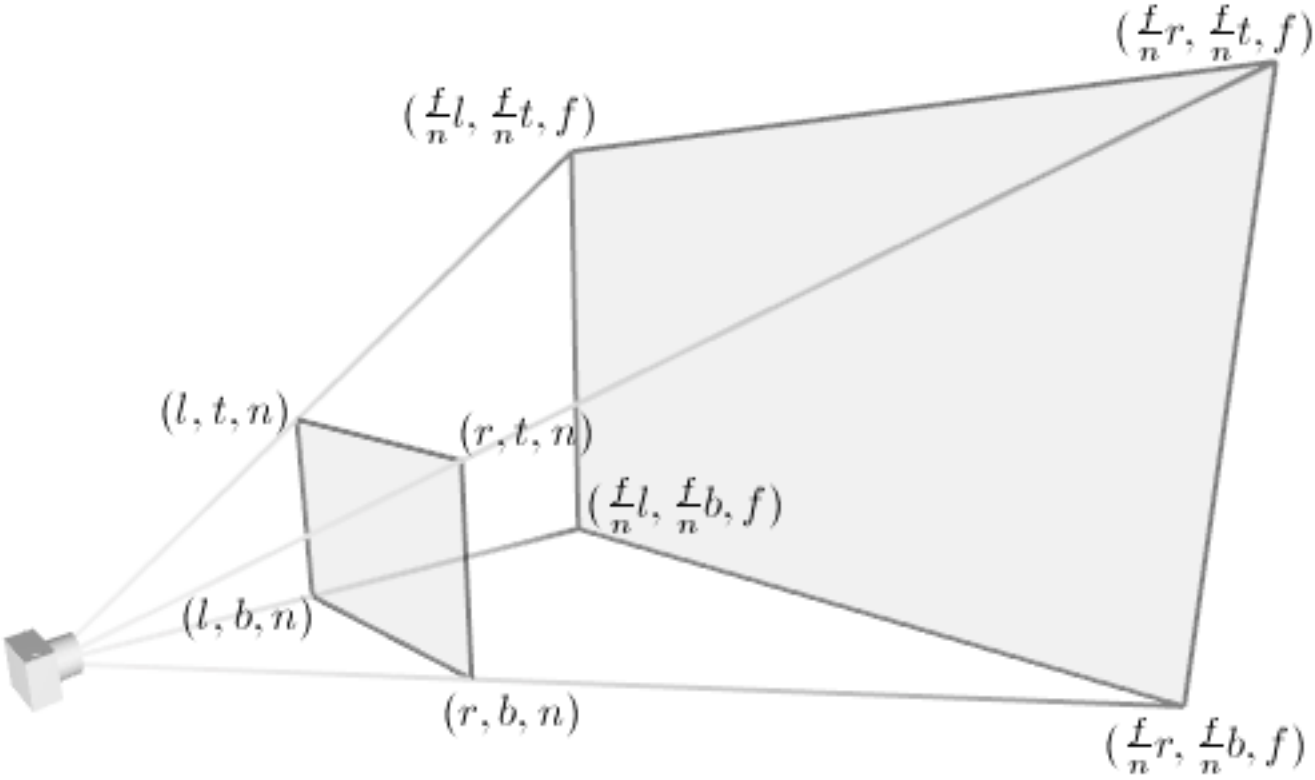
```
// Note that the object will be translated first then rotated
glRotatef(angle, 1, 0, 0);   // rotate object angle degree around X-axis
glTranslatef(x, y, z);       // move object to (x, y, z)
drawObject();
```

Projection Matrix (GL_PROJECTION)

GL_PROJECTION matrix is used to define the frustum. This frustum determines which objects or portions of objects will be clipped out. Also, it determines how the 3D scene is projected onto the screen.

OpenGL provides 2 functions for GL_PROJECTION transformation. **glFrustum()** is to produce a perspective projection, and **glOrtho()** is to produce a orthographic (parallel)

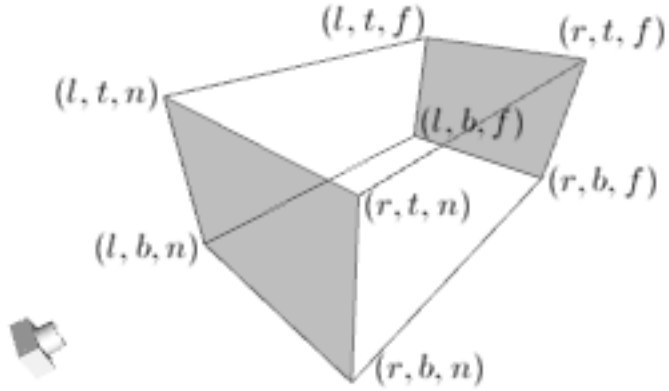
projection. Both functions require 6 parameters to specify 6 clipping planes; *left*, *right*, *bottom*, *top*, *near* and *far* planes. 8 vertices of the viewing frustum are shown in the following image.



OpenGL Perspective Viewing Frustum

The vertices of the far (back) plane can be simply calculated by the ratio of similar triangles, for example, the left of the far plane is;

$$\frac{far}{near} = \frac{left_{far}}{left}, \quad left_{far} = \frac{far}{near} \cdot left$$



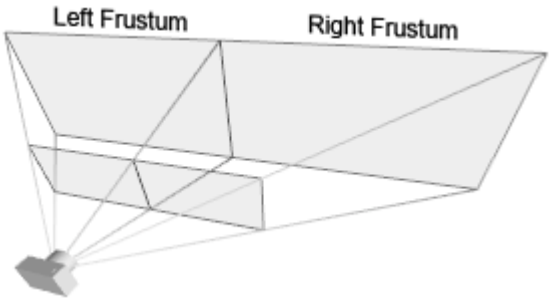
OpenGL Orthographic Frustum

For orthographic projection, this ratio will be 1, so the *left*, *right*, *bottom* and *top* values of the far plane will be same as on the near plane.

You may also use gluPerspective() and gluOrtho2D() functions with less number of parameters. **gluPerspective()** requires only 4 parameters; vertical field of view (FOV), the aspect ratio of width to height and the distances to near and far clipping planes. The

equivalent conversion from gluPerspective() to glFrustum() is described in the following code.

```
// This creates a symmetric frustum.
// It converts to 6 params (l, r, b, t, n, f) for glFrustum()
// from given 4 params (fovy, aspect, near, far)
void makeFrustum(double fovY, double aspectRatio, double front, double
back)
{
    const double DEG2RAD = 3.14159265 / 180;

    double tangent = tan(fovY/2 * DEG2RAD);   // tangent of half fovY
    double height = front * tangent;          // half height of near plane
    double width = height * aspectRatio;      // half width of near plane

    // params: left, right, bottom, top, near, far
    glFrustum(-width, width, -height, height, front, back);
}
```



An example of an asymmetric frustum

However, you have to use glFrustum() directly if you need to create a non-symmetrical viewing volume. For example, if you want to render a wide scene into 2 adjoining screens, you can break down the frustum into 2 asymmetric frustums (left and right). Then, render the scene with each frustum.

Texture Matrix (GL_TEXTURE)

Texture coordinates (*s*, *t*, *r*, *q*) are multiplied by GL_TEXTURE matrix before any texture mapping. By default it is the identity, so texture will be mapped to objects exactly where you assigned the texture coordinates. By modifying GL_TEXTURE, you can slide, rotate, stretch, and shrink the texture.

```
// rotate texture around X-axis
glMatrixMode(GL_TEXTURE);
glRotatef(angle, 1, 0, 0);
```

Color Matrix (GL_COLOR)

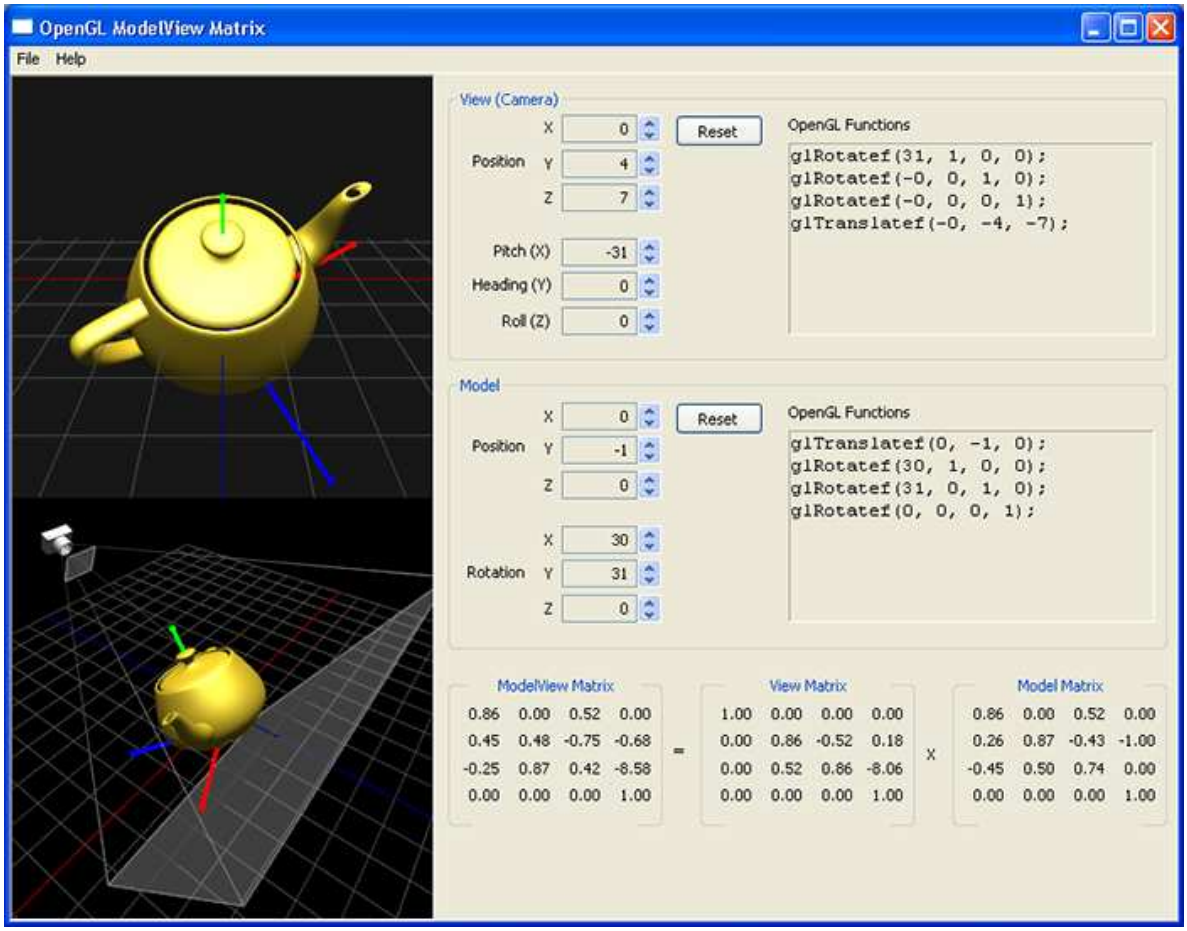
The color components (*r*, *g*, *b*, *a*) are multiplied by GL_COLOR matrix. It can be used for color space conversion and color component swaping. GL_COLOR matrix is not commonly used and is required **GL_ARB_imaging** extension.

Other Matrix Routines

glPushMatrix() :
push the current matrix into the current matrix stack.
glPopMatrix() :

pop the current matrix from the current matrix stack.
glLoadIdentity() :
set the current matrix to the identity matrix.
glLoadMatrix{fd}(*m*) :
replace the current matrix with the matrix *m*.
glLoadTransposeMatrix{fd}(*m*) :
replace the current matrix with the row-major ordered matrix *m*.
glMultMatrix{fd}(*m*) :
multiply the current matrix by the matrix *m*, and update the result to the current matrix.
glMultTransposeMatrix{fd}(*m*) :
multiply the current matrix by the row-major ordered matrix *m*, and update the result to the current matrix.
glGetFloatv(GL_MODELVIEW_MATRIX, *m*) :
return 16 values of GL_MODELVIEW matrix to *m*.

Example: ModelView Matrix

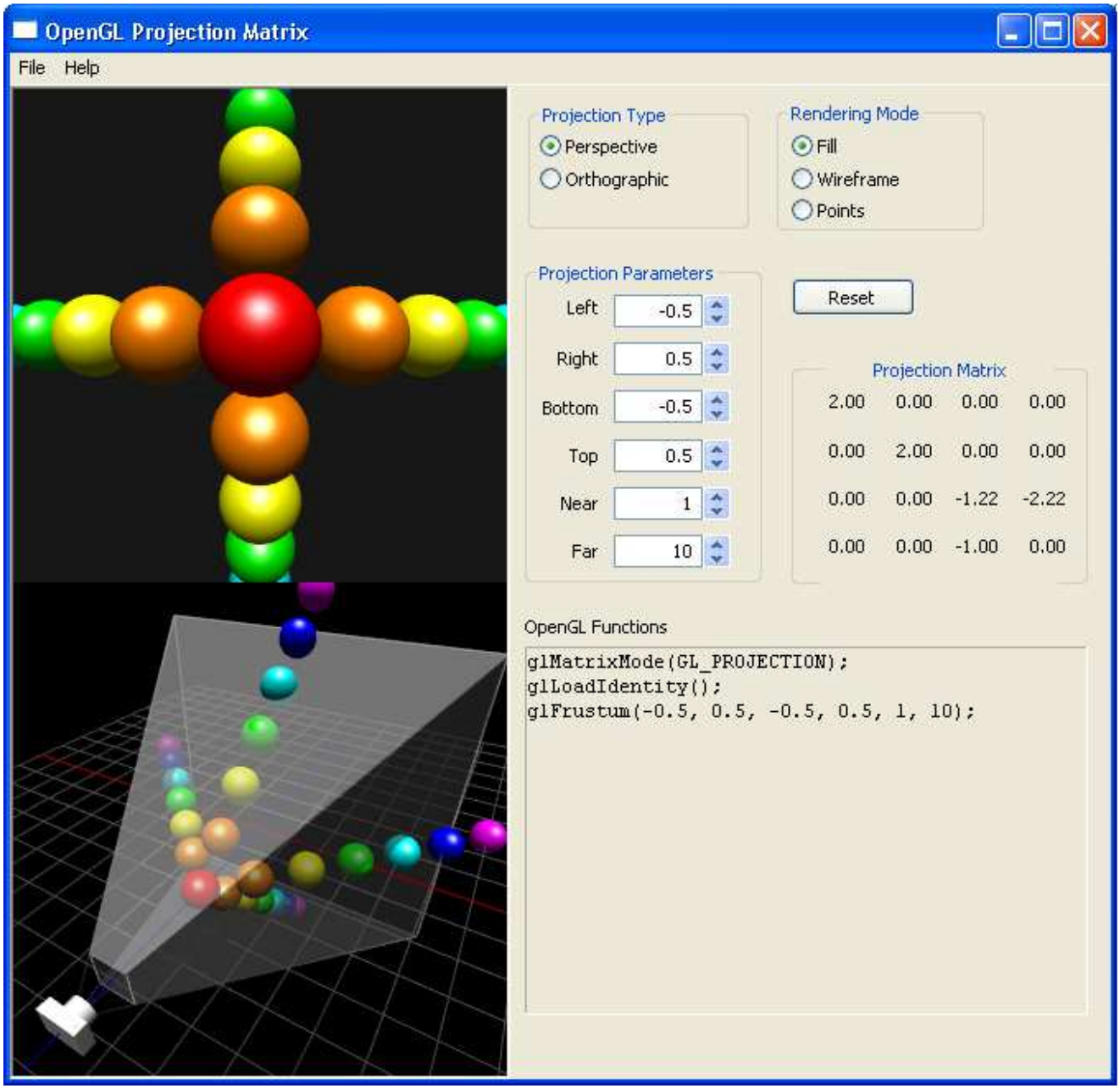


This demo application shows how to manipulate GL_MODELVIEW matrix with glTranslatef() and glRotatef().

Note once again, OpenGL performs multiple transformations in reverse order, therefore, viewing transform comes first before modeling transform in your code. And, if you want to rotate then translate an object, put glTranslatef() first then glRotatef().

```
...
// initialze ModelView matrix
glLoadIdentity();

// ModelView matrix is product of viewing matrix and modeling matrix
// ModelView_M = View_M * Model_M
// First, transform the camera (viewing matrix) from world space to eye
space
// Notice all values are negated, because we move the whole scene with the
// inverse of camera transform
glRotatef(-cameraAngle[0], 1, 0, 0); // pitch
glRotatef(-cameraAngle[1], 0, 1, 0); // heading
glRotatef(-cameraAngle[2], 0, 0, 1); // roll
glTranslatef(-cameraPosition[0], -cameraPosition[1], -cameraPosition[2]);

// transform the object
// This modeling transform will modify modeling matrix,
// convert object coords to world coords.
glPushMatrix();
glTranslatef(modelPosition[0], modelPosition[1], modelPosition[2]);
glRotatef(modelAngle[0], 1, 0, 0);
glRotatef(modelAngle[1], 0, 1, 0);
glRotatef(modelAngle[2], 0, 0, 1);

// draw objects
draw();

glPopMatrix();
...
```

Example: Projection Matrix



This demo application is to show how to manipulate the projection transformation with glFrustum() or glOrtho().

8. 3D Shapes in Open GL

Introduction

All is perfectly fine if we want to draw standard 2D shapes in OpenGL, but the next step up is 3D shapes, and it is these shapes that will really make your application look impressive. The best part about 3D shapes, is that they are made up of 2D shapes, just drawn in 3D space. While OpenGL provides methods for easily rendering 2D shapes, it doesn't provide any methods for shapes such as cubes, spheres, pyramids, etc. But all is not lost, you have two choices. The first choice is to create the shapes yourself, work out the vertices, determine which vertices you want to use to make faces, and hand code it all in. The next choice, not including 3D modeling applications and model loaders, is to use GLUT for simple 3D shapes. GLUT comes with the ability to render some extremely basic 3D shapes such as a cube, sphere (without texture coordinates), cone, torus/donut and the all famous teapot. GLUT also lets us render this in both their wireframe version, or their regular filled version. GLUT also comes with several other shapes, but they are not all that common. Lets take a look at the code required to render these more common shapes:

Cube

```
1. glutWireCube(double size);
2. glutSolidCube(double size);
```

This will create a cube with the same width, height and depth/length, each the length of the size parameter specified. This cube in GLUT also comes with surface normals but not texture coordinates.

Sphere

```
1. glutWireSphere(double radius, int slices, int stacks);
2. glutSolidSphere(double radius, int slices, int stacks);
```

The calls to create a sphere in GLUT require you to give a radius, which determines the size of the sphere, and the number of stacks and slices. The stacks and slices determine the quality of the sphere, and are the number of divisions in vertical and horizontal directions. The sphere does come with surface normals, but does not come with texture coordinates.

Cone

```
1. glutWireCone(double radius, double height, int slices, int stacks);

2. glutSolidCone(double radius, double height, int slices, int stacks);
```

If you want to create a cone, you would use the above GLUT calls. These calls are almost identical to that of the sphere code, we have a radius and the stacks and slices. But it also takes another parameter which defines the height of the cone. Also note that the radius of the cone, refers to the radius of the base of the cone. The cone will come with surface normals, but does not come with texture coordinates.

Torus

```
1. glutWireTorus(double inner_radius, double outer_radius, int sides, i
   nt rings);
2. glutSolidTorus(double inner_radius, double outer_radius, int sides,
   int rings);
```

A torus looks exactly like a donut. It has an inner radius, which specifies the size of the hole in the middle, an outer radius, which specifies the outer side of the torus from the centre (not the inner radius onwards), the sides specifies the number of sides in each radial section and finally, the rings specify how many radial divisions are used for the torus.

Teapot

```
1. glutWireTeapot(double size);
2. glutSolidTeapot(double size);
```

The all famous teapot was first created in 1975 by Martin Newell and is widely used for testing in computer graphics because it is round, has saddle points, can project a shadow onto itself and looks decent when it is untextured. All you have to do to create the teapot is specify the size for the teapot. The teapot comes with surface normals and texture coordinates which means it is perfect for testing bump mapping, environment mapping, and many other effects.