CHAPTER 1
INTRODUCTION

Computer Graphics is the creation and manipulation of picture with the aid of computers. It is divided into two broad classes. It is also called passive graphics. Here the user has no control over the pictures produced on the screen. Interactive graphics provides extensive user-computer interaction. It provides a two-way communication between computer and user. It provides a tool called “motion dynamics” using which the user can move objects. It is also able to produce audio feedback to make the simulated environment more realistic. C language helps to implement different graphics objects which are interactive and non-interactive.

1.1 About

Computer Graphics is one of the most powerful and interesting facets of computers. There is a lot we can do in graphics apart from drawing figures of various shapes. All video games, animation, multimedia predominantly works using computer graphics. There are so many applications of computer graphics, which make it significant. Computer graphics is concerned with all aspects of producing pictures or images using a computer.

1.2 About OpenGL

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that is used to specify the objects and operations needed to produce interactive three-dimensional applications

OpenGL (Open Graphics Library) is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics the interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, and flight simulation. It is also used in video games , where it competes with Direct3D on Microsoft Windows platforms. OpenGL is managed by the non-profit technology consortium , the Khronos Group. OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, user must work through whatever windowing system controls the particular hardware that the user is using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow the user to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. A sophisticated library that

provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation. OpenGL serves two main purposes:

- To hide the complexities of interfacing with different 3D accelerators, by presenting the programmer with a single, uniform API.
- To hide the differing capabilities of hardware platforms, by requiring that all implementations support the full OpenGL feature set (using software emulation if necessary).

OpenGL's basic operation is to accept primitives such as points, lines and polygons, and convert them into pixels. This is done by a graphics pipeline known as the OpenGL state machine. Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives. Prior to the introduction of OpenGL 2.0, each stage of the pipeline performed a fixed function and was configurable only within tight limits. OpenGL 2.0 offers several stages that are fully programmable using GLSL. OpenGL is a low-level, procedural API, requiring the programmer to dictate the exact steps required to render a scene. These contrasts with descriptive APIs, where a programmer only needs to describe a scene and can let the library manage the details of rendering it. OpenGL's low-level design requires programmers to have a good knowledge of the graphics pipeline, but also gives a certain amount of freedom to implement novel rendering algorithms. OpenGL has historically been influential on the development of 3D accelerators, promoting a base level of functionality that is now common in consumer-level hardware:

- Rasterised points, lines and polygons as basic primitives.
- Simplified version of the Graphics Pipeline Process; excludes a number of features like blending, VBOs and logic ops
- A transform and lighting pipeline
- Texture mapping
- Alpha Blending

A brief description of the process in the graphics pipeline could be:

1. Evaluation, if necessary, of the polynomial functions which define certain inputs, like NURBS surfaces, approximating curves and the surface geometry.
2. Vertex operations, transforming and lighting them depending on their material. Also clipping non visible parts of the scene in order to produce the viewing volume.
3. Rasterisation or conversion of the previous information into pixels. The polygons are represented by the appropriate color by means of interpolation algorithms.
4. Per-fragment operations, like updating values depending on incoming and previously stored depth values, or color combinations, among others.
5. Lastly, fragments are inserted into the Frame buffer.

Several libraries are built on top of or beside OpenGL to provide features not available in OpenGL itself. Libraries such as GLU can always be found with OpenGL implementations, and others such as GLUT and SDL have grown over time and provide rudimentary cross platform windowing and mouse functionality and if unavailable can easily be downloaded and added to a development environment. Simple graphical user interface functionality can be found in libraries like GLUI or FLTK. Still other libraries like GLAux (OpenGL Auxiliary Library) are deprecated and have been superseded by functionality commonly available in more popular libraries, but code using them still exists, particularly in simple tutorials. Other libraries have been created to provide OpenGL application developers a simple means of managing OpenGL extensions and versioning. Examples of these libraries include GLEW (the OpenGL Extension Wrangler Library) and GLEE (the OpenGL Easy Extension Library).

In addition to the aforementioned simple libraries, other higher level object oriented scene graph retain mode libraries exist such as PLIB, OpenSG, OpenSceneGraph, and OpenGL Performer. These are available as cross platform free/open source or proprietary programming interfaces written on top of OpenGL and systems libraries to enable the creation of real-time visual simulation applications. Other solutions support parallel OpenGL programs for Virtual Reality, scalability or graphics clusters usage, either transparently like Chromium or through a programming interface like Equalizer.

Although the OpenGL specification defines a particular graphics processing pipeline, platform vendors have the freedom to tailor a particular OpenGL implementation to meet unique system cost and performance objectives. Individual calls can be executed on dedicated hardware, run as software routines on the standard system CPU, or implemented as a combination of both dedicated hardware and software routines. This implementation flexibility means that OpenGL hardware acceleration can range from simple rendering to full geometry and is widely available on everything from low- cost PCs to high-end workstations and supercomputers. Application developers are assured consistent display results regardless of the platform implementation of the OpenGL environment.

Using the OpenGL extension mechanism, hardware developers can differentiate their products by developing extensions that allow software developers to access additional performance and technological innovations.

Many OpenGL extensions, as well as extensions to related APIs like GLU, GLX, and WGL, have been defined by vendors and groups of vendors. The OpenGL Extension Registry is maintained by SGI and contains specifications for all known extensions, written as modifications to the appropriate specification documents. The registry also defines naming conventions, guidelines for creating new extensions and writing suitable extension specifications, and other related documentation.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, the user must work through whatever windowing system controls the

particular hardware that the user is using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow the user to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules.

The color plate gives an idea of the kinds of things that can be done with the OpenGL graphics system. The following list briefly describes the major graphics operations which OpenGL performs to render an image on the screen.

- Construct shapes from geometric primitives, thereby creating mathematical descriptions of objects. (OpenGL considers points, lines, polygons, images, and bitmaps to be primitives.)
- Arrange the objects in three-dimensional space and select the desired vantage point for viewing the composed scene.
- Calculate the color of all the objects. The color might be explicitly assigned by the application, determined from specified lighting conditions, obtained by pasting a texture onto the objects, or some combination of these three actions.
- Convert the mathematical description of objects and their associated color information to pixels on the screen. This process is called rasterization.

During these stages, OpenGL might perform other operations, such as eliminating parts of objects that are hidden by other objects. In addition, after the scene is rasterized but before it's drawn on the screen, the user can perform some operations on the pixel data if needed. In some implementations (such as with the X Window System), OpenGL is designed to work even if the computer that displays the graphics that is created isn't the computer that runs the graphics program. This might be the case if a user work in a networked computer environment where many computers are connected to one another by a digital network.

In this situation, the computer on which the program runs and issues OpenGL drawing commands is called the client, and the computer that receives those commands and performs the drawing is called the server. The format for transmitting OpenGL commands (called the protocol) from the client to the server is always the same, so OpenGL programs can work across a network even if the client and server are different kinds of computers. If an OpenGL program isn't running across a network, then there's only one computer, and it is both the client and the server.

1.2.1 OpenGL Command Syntax

As it might have been observed from the simple program in the previous section, OpenGL commands use the prefix gl and initial capital letters for each word making up the command name (recall glClearColor(), for example). Similarly, OpenGL defined constants begin with GL_, use all capital letters, and use underscores to separate words (like GL_COLOR_BUFFER_BIT).The user might also have noticed some seemingly extraneous letters appended to some command names (for example, the 3f in glColor3f () and glVertex3f (). It's true that the Color part of the command name glColor3f () is enough to define the command as one that sets the current color.

However, more than one such command has been defined so that the user can use different types of arguments. In particular, the 3 part of the suffix indicates that three arguments are given; another version of the Color command takes four arguments. The f part of the suffix indicates that the arguments are floating-point numbers. Having different formats allows OpenGL to accept the user's data in his or her own data format. Some OpenGL commands accept as many as 8 different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are shown in Table 1.1, along with the corresponding OpenGL type definitions. The particular implementation of OpenGL that are used might not follow this scheme exactly; an implementation in C++ or Ada, for example, wouldn't need to.

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
B	8-bit integer	signed char	GLbyte
S	16-bit integer	Short	GLshort
I	32-bit integer	int or long	GLint, GLsizei
F	32-bit floating- point	Float	GLfloat, GLclampf
D	64-bit floating- point	Double	GLdouble, GLclampd
Ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
Us	16-bit unsigned integer	unsigned short	GLushort
Ui	32-bit unsigned integer	Unsigned int or long	GLuint,GLenum, GLbitfield

1.2.2 OpenGL as a State Machine

OpenGL is a state machine. It can be put into various states (or modes) that then remain in effect until it is changed. As it is already seen, the current color is a state variable. The user can set the current color to white, red, or any other color, and thereafter every object is drawn with that color until the current set color to something else. The current color is only one of many state variables that OpenGL maintains. Others control such things as the current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes, pixel-packing conventions, positions and characteristics of lights, and material properties of the objects being drawn. Many state variables refer to modes that are enabled or disabled with the command glEnable () or glDisable (). Each state variable or mode has a default value, and at any point the user can query the system for each variable's current value.

Typically, one of the six following commands are used to do this: **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**,**glGetIntegerv()**, **glGetPointerv()**, or **glIsEnabled()**. Which of these commands is selected depends on what data type the answer needs to be given in. Some state variables have a more specific query command (such as **glGetLight*()**, **glGetError()**, or **glGetPolygonStipple()**). In addition, the user can save a collection of state variables on an attribute stack with **glPushAttrib()** or **glPushClientAttrib()**, temporarily modify them, and later restore the values with **glPopAttrib()** or **glPopClientAttrib()**.

For temporary state changes, the user should use these commands rather than any of the query commands, since they're likely to be more efficient.

1.2.3 OpenGL Rendering Pipeline

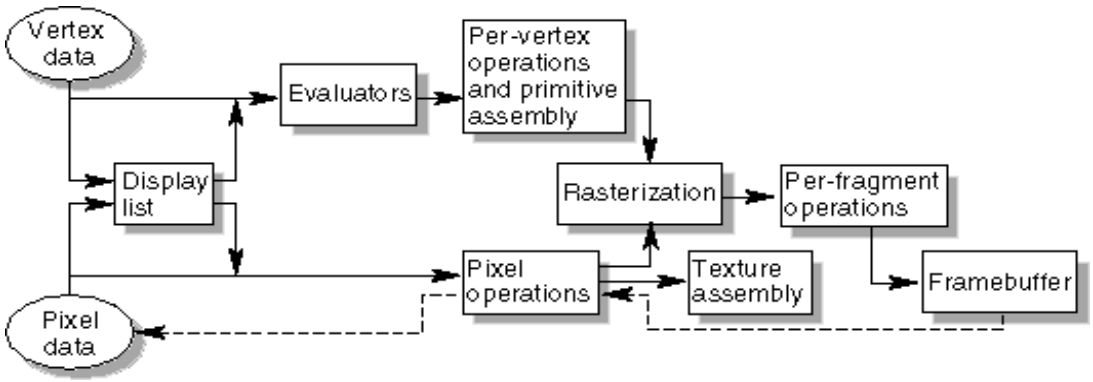


Figure 1.1: Order of Operations

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering, as shown in the figure below, is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do. The following diagram shows the Henry Ford assembly line approach, which OpenGL takes to processing data. Geometric data (vertices, lines, and polygons) follow the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per fragment operations) before the final pixel data is written into the frame buffer. Given below are the key stages in the OpenGL rendering pipeline.

Display Lists

All data, whether it describes geometry or pixels, can be saved in a *display list* for current or later use. (The alternative to retaining data in a display list is processing the data immediately - also known as *immediate mode*.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.

Evaluators

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points.

Per-Vertex Operations

For vertex data, next is the "per-vertex operations" stage, which converts the vertices into primitives. Some vertex data are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on the screen. If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

Primitive Assembly

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines. The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

Pixel Operations

While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. If pixel data is read from the frame buffer, pixel-transfer operations are performed. Then these results are packed into an appropriate format and returned to an array in system memory. There are special pixel copy operations to copy data in the framebuffer to other parts of the framebuffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the frame buffer.

Texture Assembly

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that it can be easily switched among them. Some OpenGL implementations may have special resources to accelerate texture performance. There may

be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

Rasterization

Rasterization is the conversion of both geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the frame buffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support anti-aliasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

Fragment Operations

Before values are actually stored into the frame buffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled. The first operation which may be encountered is texturing, where a Texel is generated from texture memory for each fragment and applied to the fragment. Then fog calculations may be applied, followed by the scissor test, the alpha test, the stencil test, and the depth-buffer test. Failing an enabled test may end the continued processing of a fragment's square. Then, blending, dithering, logical operation, and masking by a bitmask may be performed. Finally, the thoroughly processed fragment is drawn into the appropriate buffer.

1.2.4 OpenGL – Related Libraries

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow the user to simplify the programming tasks, including the following:

The 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, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. GLU routines use the prefix **glu**.

For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix **glX**. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. All WGL routines use the prefix **wgl**. For IBM

OS/2, the PGL is the Presentation manager to OpenGL interface, and its routines use the prefix **pgl**.

The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs. GLUT routines use the prefix **glut**.

Open Inventor is an object-oriented toolkit based on OpenGL which provides objects and methods for creating interactive three-dimensional graphics applications. Open Inventor, which is written in C++, provides prebuilt objects and a built-in event model for user interaction, high-level application components for creating and editing three-dimensional scenes, and the ability to print objects and exchange data in other graphics formats. Open Inventor is separate from OpenGL.

The Display Callback

glutDisplayFunc (void (*func)(void)) is the first and most important event callback function. Whenever GLUT determines the contents of the window need to be redisplayed, the callback function registered by **glutDisplayFunc ()** is executed.

Therefore, the user should put all the routines needed to redraw the scene in the display callback function. If the program changes the contents of the window, sometimes the user will have to call **glutPostRedisplay**(void), which gives **glutMainLoop()** a nudge to call the registered display callback at its next opportunity.

Running the Program

The very last thing to do is call **glutMainLoop** (void). All windows that have been created are now shown, and rendering to those windows is now effective. Event processing begins, and the registered display callback is triggered. Once this loop is entered, it is never exited!

Handling Input Events

User can use these routines to register callback commands that are invoked when specified events occur.

- **glutReshapeFunc** (void (*func)(int w, int h)) indicates what action should be taken when the window is resized.
- **GlutKeyboardFunc** (void (*func)(unsigned char key, int x, int y)) and **glutMouseFunc**(void (*func)(int button, int state,int x, int y)) allow the user to link a keyboard key or a mouse button with a routine that's invoked when the key or mousebutton is pressed or released.

- **glutMotionFunc** (void (*func)(int x, int y)) registers a routine to call back when the mouse is moved while a mouse button is also pressed.

Managing a Background Process

User can specify a function that's to be executed if no other events are pending - for example, when the event loop would otherwise be idle - with **glutIdleFunc**(void (*func) (void)). This routine takes a pointer to the function as its only argument. Pass in NULL (zero) to disable the execution of the function.

CHAPTER 2
REQUIREMENT ANALYSIS

The requirement analysis specifies the requirements needed to develop a graphic project.

2.1 Requirements of the project

A graphics package that attracts the attention of the viewers is to be implemented. The package should provide a platform for user to perform animation.

2.2 Resource Requirements

The requirement analysis phase of the project can be classified into:

- Software Requirements
- Hardware Requirements

Software Requirements

This document will outline the software design and specification of our application. The application is a Windows based C++ implementation with OpenGL. Our main UI will be developed in C++, which is a high level language. This application will allow viewing of GDS II files, assignment of Color and Transparency to layers within the VLSI object, as well as printing of the rendered object. There will be the ability to save these color/transparency palettes for a given GDS file as well as the foundry used to create the file. These palettes can then be used with future GDS files to save time assigning colors/transparencies to layers.

Operator/user interface characteristics from the human factors point of view

- Mouse Interface allows the user to point and click on GDS objects.
- Standard Win Forms keyboard shortcuts to access menus
- OpenGL VLSI viewer
- The interface will use OpenGL to display GDS file.
- Any mouse and keyboard that works with Win Forms.
- Program will use window's print APIs to interface with printer.

OpenGL libraries are required are:

- GLUT library
- STDLIB library

Hardware Requirements

There are no rigorous restrictions on the machine configuration. The model should be capable of working on all machines capable of supporting recent versions of Microsoft Visual Studio.

- Processor: Intel® Pentium 4 CPU
- Hard disk Capacity: 80 GB
- RAM: 1 GB
- CPU Speed: 2.9 GHz
- Keyboard: Standard
- Mouse: Standard

CHAPTER 3

DESIGN PHASE

Design of any software depends on the architecture of the machine on which that software runs, for which the designer needs to know the system architecture. Design process involves design of suitable algorithms, modules, subsystems, interfaces etc.

3.1 Algorithm

The entire design process can be explained using a simple algorithm. The algorithm gives a detailed description of the design process of 'dynamic sorting algorithm visualization'.

The various steps involved in the design of 'dynamic sorting algorithm visualization' are as shown below:

Step 1. Start

Step 2. Set Initial Display Mode

Step 3. Set Initial Window Size to (1000, 600)

Step 4. Set Initial Window Position to (0, 0)

Step 5. Create Window "Dynamic Sorting Visualizer"

Step 6. Display()

- Set clear to clear the buffer
- call front function
- call display_text function
- call glFlush function

Step 7. keyboard

- Assign 'r' to randomize the numbers
- Assign 's' to start sorting the numbers
- Assign 'p' to pause the sorting process
- Assign 'c' to change the sorting algorithm
  - call insertionsort()
  - call bubblesort()
  - call ripplesort()
  - call selectionsort()
- Assign 'Esc' to exit

Step 8. Initialize()

- Set ClearColor
- Set MatrixMode
- Set LoadIdentity
- Set gluOrtho2D

Step 9. Stop

3.2 Flow Diagram

A flow diagram is a common type of chart that represents an algorithm or process showing the steps as boxes of various kinds, and their order by connecting these with arrows. Flow diagrams are used in analyzing, designing, documenting or managing a process or program in various fields.

Flow diagrams used to be a popular means for describing computer algorithms. They are still used for this purpose; modern techniques such as UML activity diagrams can be considered to be extensions of the flow diagram.

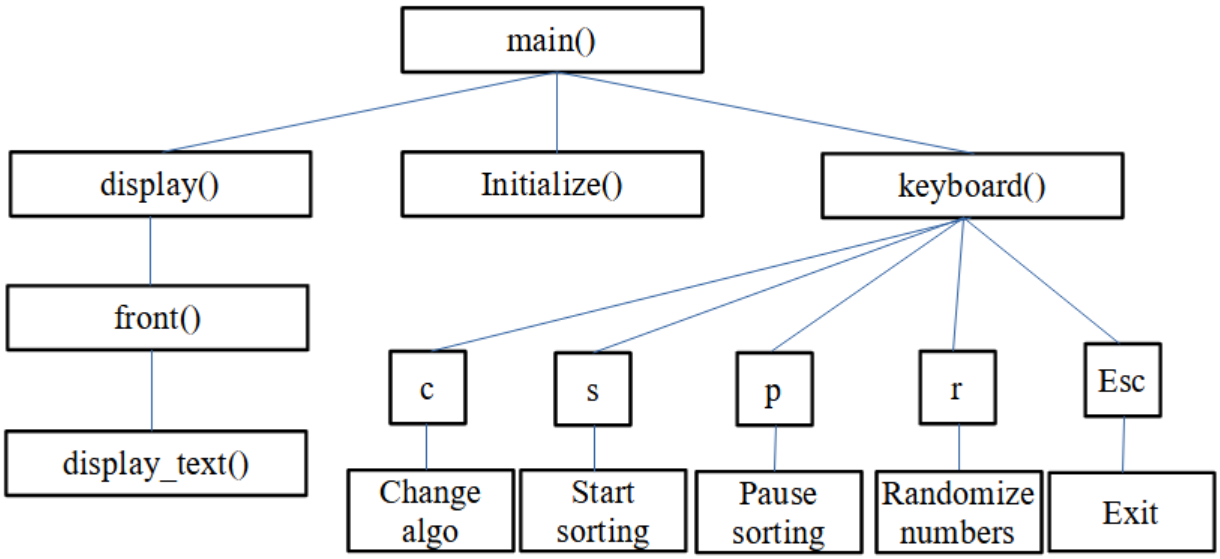
However, their popularity decreased when, in the 1970s, interactive computer terminals and third-generation programming languages became the common tools of the trade, since algorithms can be expressed much more concisely and readably as source code in such a language. Often, pseudo-code is used, which uses the common idioms of such languages without strictly adhering to the details of a particular one.

There are many different types of flow diagrams. On the one hand there are different types for different users, such as analysts, designers, engineers, managers, or programmers. On the other hand those flow diagrams can represent different types of objects. Sterneckert (2003) divides four more general types of flow diagrams:

- Document flow diagram, showing a document flow through system
- Data flow diagram, showing data flows in a system
- System flow diagram showing controls at a physical or resource level
- Program flow diagram, showing the controls in a program within a system

Flow diagrams show how data is processed at different stages in the system. Flow models are used to show how data flows through a sequence of processing steps. The data is transformed at each step before moving on to the next stage. These processing steps of transformations are program functions when data flow diagrams are used to document a software design.

The following Flow Diagram shows the processing of different operations in this project.



CHAPTER 4

IMPLEMENTATION

The implementation stage of this model involves the following phases:

1. Implementation of OpenGL built in functions.
2. User defined function Implementation.

Implementation of OpenGL Built In Functions

glutInit( )

glutInit is used to initialize the GLUT library.
Usage: void glutInit(int *argcp, char **argv);
Description: glutInit will initialize the GLUT library and negotiate a session with the window system.

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.
GLUT_RGB- An alias for GLUT_RGBA.
GLUT_SINGLE-Bit mask to select a single buffered window.
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_DEPTH- Bit mask to select a window with a depth buffer.
Description: The initial display mode is used when creating top-level windows, sub windows, and overlays to determine the OpenGL display mode for the to-be-created window or overlay.

glutCreateWindow( )

glutCreateWindow creates a top-level window.
Usage: int glutCreateWindow(char *name);
Name - ASCII character string for use as window name.
Description: glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name. Implicitly, the current window is set to the newly created window. Each created window has a unique associated OpenGL context.

glutDisplayFunc( )

glutDisplayFunc sets the display callback for the current window.
Usage: void glutDisplayFunc(void (*func)(void));
Func - The new display callback function.
Description: glutDisplayFunc sets the display callback for the current window. When GLUT determines that the normal plane for the window needs to be re-displayed, the display callback for the window is called. Before the callback, the current window is set to the window needing to be redisplayed and the layer in use is set to the normal plane. The display callback is called with no parameters. The entire normal plane region should be re-displayed in response to the callback.

glutKeyboardFunc()

glutKeyboardFunc sets the keyboard callback for the current window.
Usage: void glutKeyboardFunc (void(*func)(unsigned char key, int x, int y))
Description: glutKeyboardFunc sets the keyboard callback for the current window. When a user press any key on the keyboard, it generates a callback. The x and y callback parameters indicate the window relative coordinates when the key is pressed.

glutMainLoop( )

glutMainLoop enters the GLUT event processing loop.
Usage: void glutMainLoop(void);
Description: glutMainLoop enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

glMatrixMode( )

The two most important matrices are the model-view and projection matrix. At any time, the state includes values for both of these matrices, which are initially set to identity

matrices. There is only a single set of functions that can be applied to any type of matrix. Select the matrix to which the operations apply by first set in the matrix mode, a variable that is set to one type of matrix and is also part of the state.

gluOrtho2D( )

It is used to specify the two dimensional orthographic view. It takes four parameters; they specify the leftmost corner and rightmost corner of viewing rectangle.

Implementation of User Defined Functions:

1. front()

   Used for displaying the welcome screen.

2. display_text()

   Used to display the sorting page.

3. Notsorted()

   returns 1 if not sorted.

4. insertionsort()

   sorts numbers using insertion sort algorithm.

5. ripplesort()

   sorts numbers using ripple sort algorithm.

6. bubblesort()

   sorts numbers using bubble sort algorithm.

7. Selectionsort()

   sorts numbers using selection sort algorithm.

8. Makedelay()

   timer function, takes care of sort selection.

9. keyboard()

   assigns operation to keys pressed.

CHAPTER 5

TESTING

5.1 TESTING

Testing is the process of executing a program to find the errors. A good test has the high probability of finding a yet undiscovered error. A test is vital to the success of the system. System test makes a logical assumption that if all parts of the system are correct, then goal will be successfully achieved.

5.2 TEST CASES

SL.NO	Name of the test	Expected output	Result	Remarks
1.	‘c’ key pressed	Change of sorting algorithm	Sorting algorithm changed successfully	Pass
2.	‘s’ key pressed	Start sorting the numbers	Successfully sorted the numbers	Pass
3.	‘p’ key pressed	Pause the sorting process	Successfully paused sorting process	Pass
4.	‘r’ key pressed	Generate random numbers	Successfully generated random numbers	Pass
5.	‘Esc’ key pressed	End the execution	Successfully ended the execution	Pass

CHAPTER 6

CODE SNIPPETS

Main function:

```
int main(int argc,char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(1000,600);
   glutInitWindowPosition(0,0);
   glutCreateWindow("Dynamic Sorting Visualizer");
   Initialize();
   glutDisplayFunc(display);
   glutKeyboardFunc(keyboard);
   glutTimerFunc(1000,makedelay,1);
   glutMainLoop();
   return 0;
}
```

Display function:

```
void display()
{
   int ix,temp;
   glClear(GL_COLOR_BUFFER_BIT);

   if(k==0)
   front();
   else{
   display_text();
   char text[10];
   for(ix=0;ix<MAX;ix++)
   {
      printf("%d,%d,%d\n",R,G,B);
      glColor3f(R, G, B);
      glBegin(GL_POLYGON);
         glVertex2f(10+(700/(MAX+1))*ix,50);
         glVertex2f(10+(700/(MAX+1))*(ix+1),50);
         glVertex2f(10+(700/(MAX+1))*(ix+1),50+a[ix]*4);
         glVertex2f(10+(700/(MAX+1))*ix,50+a[ix]*4);
      glEnd();
```

```
        glColor3f(0, 0, 0);
           glBegin(GL_LINE_LOOP);
           glVertex2f(10+(700/(MAX+1))*ix,50);
           glVertex2f(10+(700/(MAX+1))*(ix+1),50);
           glVertex2f(10+(700/(MAX+1))*(ix+1),50+a[ix]*4);
           glVertex2f(10+(700/(MAX+1))*ix,50+a[ix]*4);
        glEnd();

        int_str(a[ix],text);
        //printf("\n%s",text);
        glColor3f(0,0,0);
bitmap_output(12+(700/(MAX+1))*ix,35,text,GLUT_BITMAP_TIMES_ROMAN_10);
    }

        if(swapflag || sorting==0)
        {
           glColor3f(0,0,0);
           glBegin(GL_POLYGON);
           glVertex2f(10+(700/(MAX+1))*j,50);
           glVertex2f(10+(700/(MAX+1))*(j+1),50);
           glVertex2f(10+(700/(MAX+1))*(j+1),50+a[j]*4);
           glVertex2f(10+(700/(MAX+1))*j,50+a[j]*4);
           glEnd();
           swapflag=0;
        }
    }
    glFlush();
}
```

front function:

```
void front()
{
    glColor3f(0.0,0.0,1.0);
    bitmap_output(290, 565, "WELCOME!",GLUT_BITMAP_TIMES_ROMAN_24);
    /*glBegin(GL_LINE_LOOP);
    glVertex2f(285, 560);
    glVertex2f(395, 560);
    glEnd();*/
    bitmap_output(320, 525, "TO",GLUT_BITMAP_TIMES_ROMAN_24);
    /*glBegin(GL_LINE_LOOP);
```

```
   glVertex2f(325, 521);
   glVertex2f(360, 521);
   glEnd();*/

                bitmap_output(150, 475, "DYNAMIC SORTING ALGORITHM
VISUALIZER",GLUT_BITMAP_TIMES_ROMAN_24);
   glBegin(GL_LINE_LOOP);
   glVertex2f(145, 470);
   glVertex2f(520, 470);
   glEnd();
              bitmap_output(192, 350, "MADE BY: PRAJWAL KULKARNI
(1KS19CS070)",GLUT_BITMAP_HELVETICA_18);

   glColor3f(1.0,0.0,0.0);
   glBegin(GL_QUADS);
   glVertex2f(520,120.0);glVertex2f(520,170);glVertex2f(796,170);glVertex2f(796,120.0);
   glEnd();
   glColor3f(0.0,1.0,0.0);
                              bitmap_output(530, 125, "Press Enter to
continue.....",GLUT_BITMAP_HELVETICA_18);
}
```

keyboard function:

```
void keyboard (unsigned char key, int x, int y)
{
   if(key==13)
   k=1;
   if (k==1 && sorting!=1)
   {
      switch (key)
      {
         case 27 : exit (0); // 27 is the ascii code for the ESC key
         case 's' : sorting = 1; break;
         case 'r' : Initialize(); break;
         case 'c' : sort_count=(sort_count+1)%SORT_NO; break;
      }
   }
   if(k==1 && sorting==1)
      if(key=='p')
         sorting=0;
}
```

insertion sort function:

```
void insertionsort()
{
   int temp;

   while(i<MAX)
   {
   if(flag==0){j=i; flag=1;}
   while(j<MAX-1)
   {
      if(a[j]>a[j+1])
      {
         swapflag=1;
         temp=a[j];
         a[j]=a[j+1];
         a[j+1]=temp;

         goto A;
      }
      j++;
      if(j==MAX-1){flag=0;}
   }
      i++;
   }
   sorting=0;
   A:
      i=j=0;
}
```

CHAPTER 7
SNAPSHOTS

In computer file systems, a snapshot is a copy of a set of files and directories as they were at a particular point in the past. The term was coined as an analogy to that in photography.

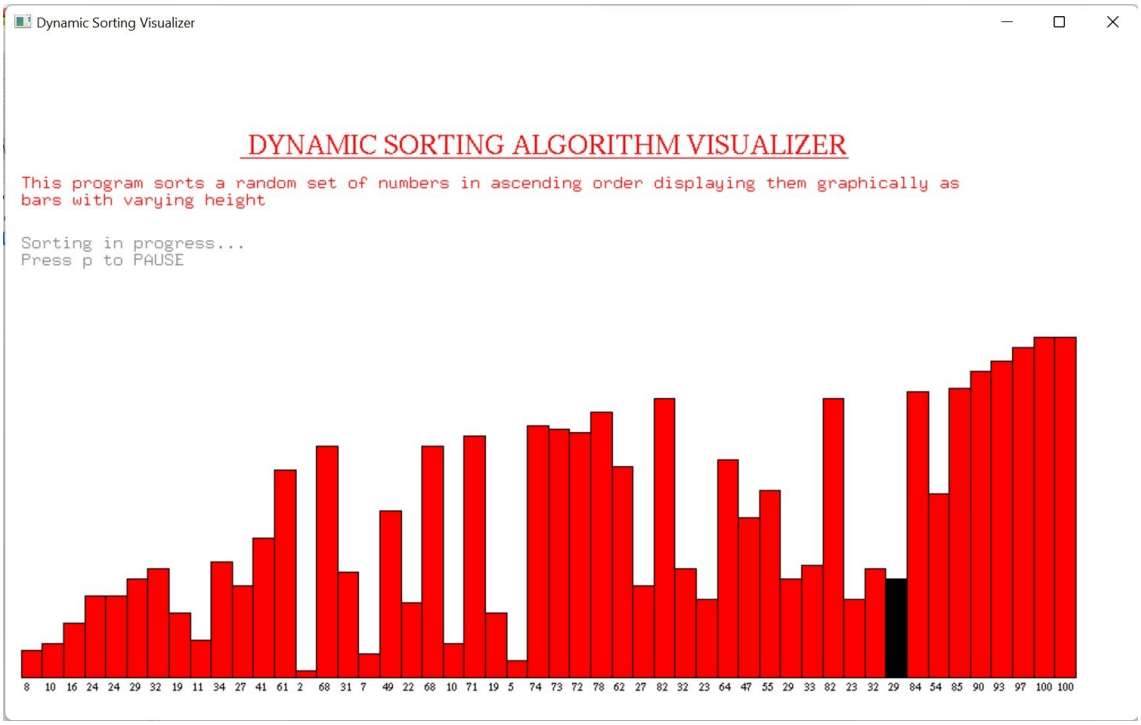
One approach to safely backing up live data is to temporarily disable write access to data during the backup, either by stopping the accessing applications or by using the locking API provided by the operating system to enforce exclusive read access. This is tolerable for low-availability systems (on desktop computers and small workgroup servers, on which regular downtime is acceptable). High-availability 24/7 systems, however, cannot bear service stoppages.

To avoid downtime, high-availability systems may instead perform the backup on a snapshot—read-only copies of the data set frozen at a point in time—and allow applications to continue writing to their data. Most snapshot implementations are efficient and can create snapshots in O (1). In other words, the time and I/O needed to create the snapshot does not increase with the size of the data set, whereas the same for a direct backup is proportional to the size of the data set.

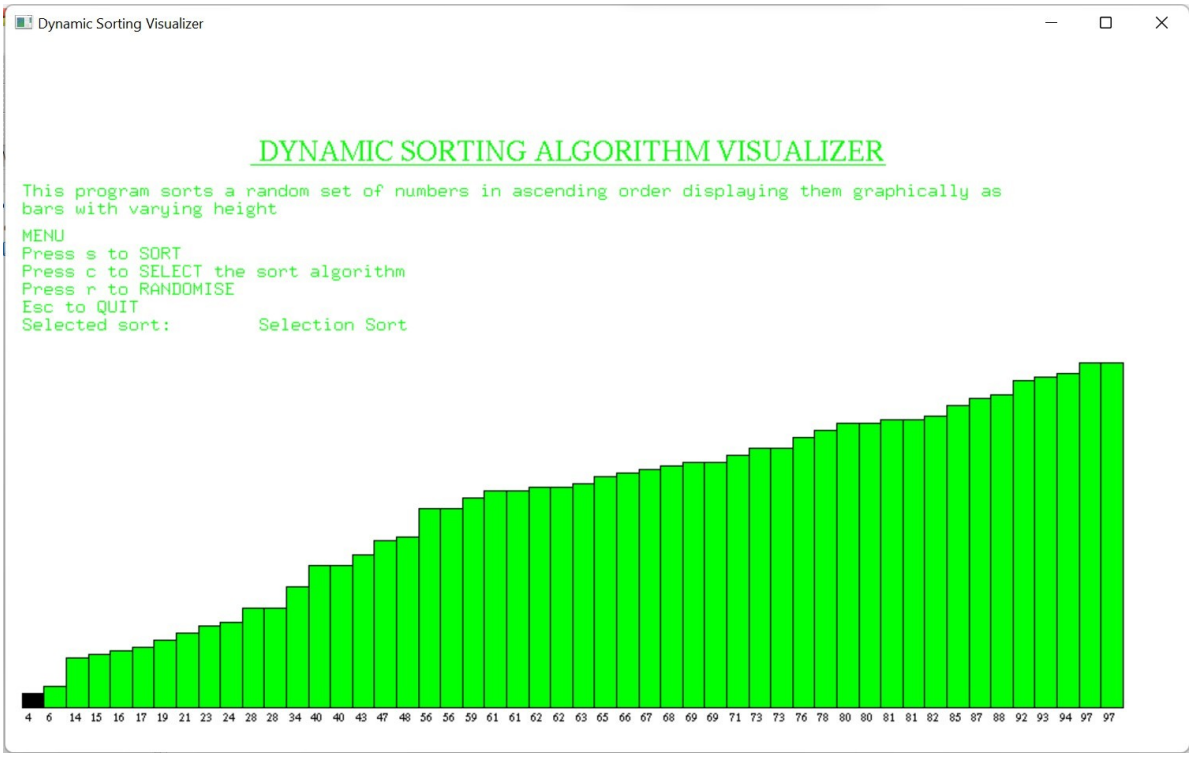
Snapshots are the pictures representing different phases of the program execution. The figure below shows the initial window of the project 'dynamic sorting algorithm visualization’

Welcome screen:

Sorting:



sorted:


Dynamic Sorting Visualizer
DYNAMIC SORTING ALGORITHM VISUALIZER
This program sorts a random set of numbers in ascending order displaying them graphically as bars with varying height
MENU
Press s to SORT
Press c to SELECT the sort algorithm
Press r to RANDOMISE
Esc to QUIT
Selected sort: Selection Sort
4 6 14 15 16 17 19 21 23 24 28 28 34 40 40 43 47 48 56 56 59 61 61 62 62 63 65 66 67 68 69 69 71 73 73 76 78 80 80 81 81 82 85 87 88 92 93 94 97 97

CHAPTER 8
CONCLUSION

This mini-project is about the design and implementation of the '**DYNAMIC SORTING ALGORITHM VISUALIZATION**' package developed using OpenGL. The very purpose of developing this project is to exploit the strength of OpenGL graphic capabilities and to illustrate how useful it is to develop project involving graphics.

This project helped a lot to learn about the proper utilization of various graphics library function that are defined in GLUT, GLU and GL. This project also helped to understand the theoretical concepts in practical applications.

CHAPTER 9
REFERENCES

The following books have been referred to complete this project:

1. **Interactive Computer Graphics – A Top Down Approach Using OpenGL** by Edward Angel.
2. **Computer Graphics – Principles & practice** by James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes.
3. **StackOverflow**