CHAPTER-1

INTRODUCTION

When multiple process try to access a shared resource(A resource can be anything on a machine. It can be memory, a peripheral device etc.) Or when multiple processes are executing concurrently, they need an awareness so as to complete their task by using a common/ shared resource. This is achieved by interprocess communication.

Computer graphics provides one of the most natural means of communicating within a computer, since our highly developed 2D and 3D pattern-recognition abilities allow us to perceive and process pictorial data rapidly and effectively. Interactive computer graphics is the most important means of producing pictures since the invention of photography and television. It has the added advantage that, with the computer, we can make pictures not only of concrete real world objects but also of abstract, synthetic objects, such as mathematical surfaces and of data that have no inherent geometry, such as survey results.

Computer graphics started with the display of data on hardcopy plotters and cathode ray tube screens soon after the introduction of computers themselves.

It has grown to include the creation, storage, and manipulation of models and images of objects. These models come from a diverse and expanding set of fields, and include physical, mathematical, engineering, architectural, and even conceptual structures, natural phenomena, and so on. Computer graphics today is largely interactive. The user controls the contents, structure, and appearance of the objects and of their displayed images by using input devices, such as keyboard, mouse, or touch-screen. Due to close relationships between the input devices and the display, the handling of such devices is included in the study of computer graphics. The advantages of the interactive graphics are many in number.

Graphics provides one of the most natural means of communicating with a computer, since our highly developed 2D and 3D patter-recognition abilities allow us to perceive and process data rapidly and efficiently. In many design, implementation, and construction processes today, the information pictures can give is virtually indispensable.

CHAPTER-2

OPENGL

2.1: Overview

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive 3D applications.

OpenGL is designed as a streamlined, Hardware- describing models of three-dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, you must build up your desired model from a small set of geometric primitives-points, lines, and polygons.

A sophisticated library that provides these features could certainly be built on top of OpenGL, The OpenGL Utility Library (GLU) provide 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, you must work through whatever windowing system controls the particular hardware you're using. Similarly, OpenGL doesn't provide high-level commands for many of the modelling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation. Also, there is a higher-level, object-oriented tool kit, open inventor which is built atop OpenGL, and is available separately for many implementations of OpenGL.

In some implementations (such as with the X Window systems) , OpenGL is designed to work even if the computer that displays the graphics you create isn't the computer that runs your graphics program.

This might be the case if you 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 your 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 the 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.

2.2: A Smidgen of OpenGL Code

Because you can do so many things with OpenGL graphics system, an OpenGL program can be complicated. However, the basic structure of useful program can be simple: Its task are initialized certain states that controls how OpenGL renders and to specify objects to be rendered.

Before you look at some OpenGL code, lets go over few terms. Rendering, which you have already seen used, is the process by which a computer creates images from models. these models, or objects, are constructed from geometric primitives – points, lines, and polygons – that are specified by their vertices.

The final rendered image consist of pixels drawn on the screen; a pixel is the smallest visible element the display hardware can put on the screen. Information about the pixels (for instance, what color they're supposed to be) is organized in memory into bit planes.

A bit plane is an area of memory that holds one bit of information for every pixel on the screen; the bit might indicate how red a particular pixel is supposed to be, for example,

The bit planes are themselves organized into a frame buffer, which holds all the information that the graphics display needs to control the color and intensity of all the pixels on the screen.

2.3: OpenGL Command Syntax

As you might have 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 use all capital letters, and use underscores to separate words (like GL_COLOR_BUFFER _ BIT).

You might also have noticed some seemingly extraneous letters appended to some command names (for example, the 3f I n glColor3f() and glVertex3f()). Its 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 you can use different types of arguments, In particular, the 3 parts of the suffix indicates that three arguments are given; another version of the color command takes four arguments. The first 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 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 implementations of OpenGL are shown in the Table along with the corresponding OpenGL type definitions.

The particular implementation of OpenGL that you are using might not follow this scheme exactly; an implementation in C++ or ADA.

Table : Command Suffixes and Arguments Data Types

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 unsigned long**	**Glint, GLenum,GLbitfield**

Finally, OpenGL defines the typedef GLvoid. This is most often used for OpenGL commands that accept pointers to arrays of values.

- **2.4: COLOR**

On a color computer screen, the hardware causes each pixel on the screen to emit different amounts of red, green, and blue light. These are called the R, G, and B values. They are often packed together (Sometimes with a fourth value, called alpha, or A), and the packed value is called RGB (RGBA) value.

The color information at each pixel can be stored either in RGBA mode, in which the R, G, B, and possibly a values are kept for each pixel. Each color index an entry in a table that defines a particular set of R, G, and B values. Such a table is called a color map.

On any graphics system, each pixel has the same amount of memory for storing its color, all the memory for all the pixels is called the color buffer. The size of a buffer is usually measured in bits, so an 8-bitbuffer could store 8 bits of data (256 possible different colors)for each pixel. The size of the possible buffers varies from machine to machine.

The R, G, and B values can range from 0.0(none) to 1.0(full intensity). For example, R=0.0, G=0.0, and B=1.0 represents the brightest possible blue.

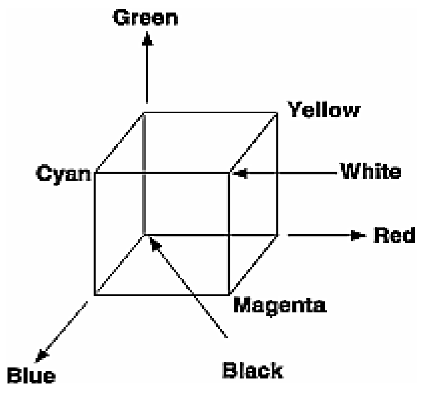
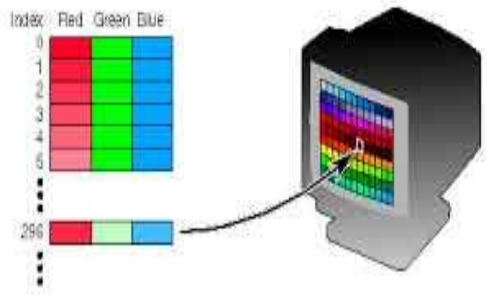


Fig 2.4.1: Color Cube in Black and White



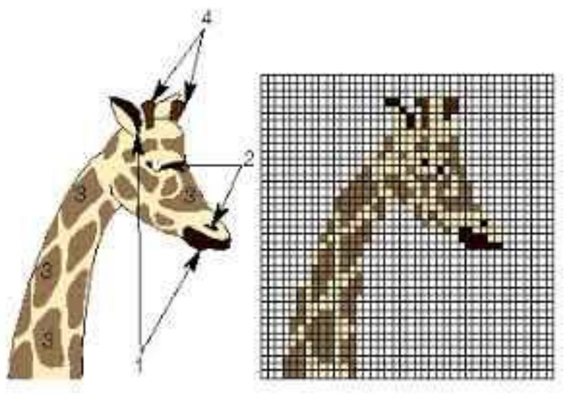


Fig 2.4.2: Color Cube **Fig 2.4.3: Size Angles**

2.5: RGBA Display Mode

In RGBA mode, the hardware sets aside a certain number of bit-planes for each of the R, G, B, and A Components as shown in figure. The RGB values are typically stored as integers rather than floating-point numbers, and they are scaled to the number of bits for the storage and retrieval.

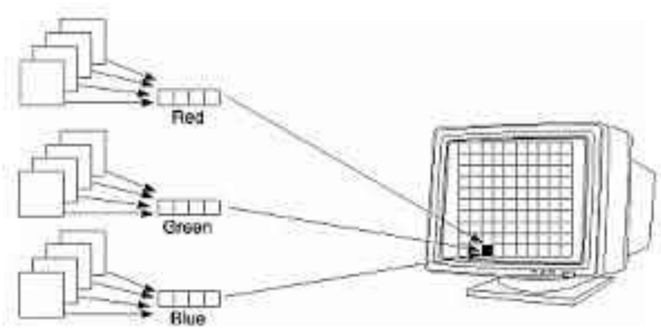


Fig 2.5.1: RGB values from the bit-planes

The number of distinct colors that can be displayed at a single pixel depends on the number bit-planes and the capacity of the hardware to interpret those bit-planes. The number of distinct colors can’t exceed 2n, where n is the number of bit-planes.

Color-Index Display mode

With color-index mode, OpenGL uses a color map (or lookup table), which is similar to using a palette to mix paints to prepare for a paint-by-number scene. A painter’s palette provides spaces to mix paints together, similarly, a computers color map provides indices where the primary red, green, and blue values can be mixed as shown in the figure.

Painter filling in a paint-by-number scene chooses a color from the color palette and fills the corresponding numbered regions with that color. A computer stores the color index in the bit-planes each pixel. Then those biplane values reference the color map, and the screen is painted with the corresponding red, green, and blue values from the color map, as shown in Figure.

In color-index mode, the number of simultaneously available colors is limited by the size of the color map and the number of bit-planes available. The size of the color map is always a power of 2 and the typical sizes range from 256(2^ (8)) 4096(2^ (12) where the exponent is the number of bit-planes being used.

CHAPTER- 3

EXPERIMENTAL SETUP

3.1: Software specifications:

1. Software type	:	Microsoft visual studio 2022.
2. Implementation Language	:	OpenGL Using C++ Language.
3. Operating system required	:	Windows with Dos.
4. Toolkit	:	GLUT Toolkit, VC++.

3.2: Hardware specifications:

1. Processor	:	Min IntelX86.
2. Memory Requirement	:	Min 16MB.
3. Display Type	:	CGA/VGA/EGA at 640X480 Resolution.
4. Hard Disk Space	:	At least 300MB.
5. Keyboard	:	Low Profile, dispatch able Type.
6. I/O Parts	:	Mouse, Monitor.

CHAPTER-4

IMPLEMENTATION

4.1: OpenGL

We have implemented our graphics package in Open GL using C++ language, that provides all the inbuilt functions for basic drawing of objects such as line drawing, rectangle, square and coloring the objects, moving an object etc.

Open GL using C++ language provides functions for initializing the graphics mode. The function for this purpose is:

Void init(void)

Init initialize the graphics system by loading a graphics driver from disc then putting the system into graphics mode. Its also resets all the graphics setting (color palette, current position, view port etc) to their defaults, then resets graph result to zero.

Some of the in-built functions used are:.

Function Details

glClear() : It clears the window

glColor3f() : It establishes what color to use drawing objects.

glBegin() : It specifies the type of primitive that the vertices define glVertex() : It specifies the position of a vertex in 2,3 or 4D

Void keyboard() : Its is the callback for events generated by pressing key.glutPostRedisplay() : Its requests that the display callback be executed after the current callback returns.

glClearColor() :It establishes what color the window will be cleared to.

glMatrixMode() : It specifies which matrix will be affected by subsequent transformations.

glLoadIdentity() : It sets the current transformation matrix to an identity matrix.

glutMainLoop()	: It cause the program to enter an event-processing loop.
glutInitWindowSize()	: It specifies the initial height and width of the window in pixels.
glutInitWindowPosition()	: It specifies the initial position of the top-left corner of the window in pixels.
glutCreateWindow()	: It creates a window on the display.
glutDisplayFunc()	: It registes the display function func that is executed when the window needs to be redraw.

4.2: SOURCE CODE

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

#define BREITE 0.1f
#define STANGENBREITE 0.025f
#define SLICES 32
#define INNERSLICES 16
#define LOOPS 1
#define FPS 64   /* more looks nicer, uses more cpu power */
#define FEM 1000.0/FPS
#define FSEM 0.001f   /* speed (bigger is faster)*/

struct config {
     GLfloat gap;
     GLfloat pinradius;
     GLfloat pinheight;
};

struct action {
     char fromstack;
     char tostack;
     struct action* next;
};
typedef struct action action;

struct actions {
     action* head;
     action* tail;
};
typedef struct actions actions;
struct disk {
     char color;
     GLfloat radius;
     struct disk* next;
     struct disk* prev;
};
typedef struct disk disk;

struct stack {
     disk* bottom;
     disk* top;
};
typedef struct stack stack;

int disks = 3;  //initial number of disks
GLfloat rotX, rotY, zoom, offsetY = 1.5, speed;
GLUquadricObj* quadric;
```

```
GLfloat pos;
GLboolean fullscreen;
stack pin[3];
float pinheight[3];
struct config config;
actions actqueue;
action* curaction;
disk* curdisk;
int duration;
char seconds[24] = "Time: 0s";
int draw, maxdraws;

//function prototypes
void moveDisk(int param);
void hanoiinit(void);
void reset();
void Display(void);
void hanoi(actions* queue, const int n, const char pin1, const
char pin2, const char pin3);
void push(stack* pin, disk* item);
disk* pop(stack* pin);
void drawDisk(GLUquadricObj** quadric, const GLfloat outer,
const GLfloat inner);
void drawPin(GLUquadricObj** quadric, const GLfloat radius,
const GLfloat height);
void drawAllPins(GLUquadricObj** quadric, const GLfloat radius,
const GLfloat height, const GLfloat gap);
void drawBitmapString(const GLfloat x, const GLfloat y, const
GLfloat z, void* font, char* string);
void drawBitmapInt(const GLfloat x, const GLfloat y, const
GLfloat z, void* font, const int number);




void hanoi(actions* queue, const int n, const char pin1, const
char pin2, const char pin3)
{
     action* curaction;
     if (n > 0)
     {
          hanoi(queue, n - 1, pin1, pin3, pin2);
          /* push action into action queue */
          curaction = (action*)malloc(sizeof(action));
          curaction->next = NULL;
          curaction->fromstack = pin1;
          curaction->tostack = pin3;
          if (queue->head == NULL)
               queue->head = curaction;
          if (queue->tail != NULL)
```

```
                queue->tail->next = curaction;
            queue->tail = curaction;
            hanoi(queue, n - 1, pin2, pin1, pin3);
        }
}

/** push item to pin */
void push(stack* pin, disk* item) {
        item->next = NULL;
        if (pin->bottom == NULL) {
            pin->bottom = item;
            pin->top = item;
            item->prev = NULL;
        }
        else {
            pin->top->next = item;
            item->prev = pin->top;
            pin->top = item;
        }
}

/** pop item from pin */
disk* pop(stack* pin) {
        disk* tmp;
        if (pin->top != NULL) {
            tmp = pin->top;
            if (pin->top->prev != NULL) {
                pin->top->prev->next = NULL;
                pin->top = tmp->prev;
            }
            else {
                pin->bottom = NULL;
                pin->top = NULL;
            }
            return tmp;
        }
        return NULL;
}

void drawDisk(GLUquadricObj** quadric, const GLfloat outer,
const GLfloat inner)
{
        glPushMatrix();
        glRotatef(-90.0, 1.0, 0.0, 0.0);
        gluCylinder(*quadric, outer, outer, BREITE, SLICES,
LOOPS);
        gluQuadricOrientation(*quadric, GLU_INSIDE);
        if (inner > 0)
            gluCylinder(*quadric, inner, inner, BREITE,
INNERSLICES, LOOPS);
```

```
     gluDisk(*quadric, inner, outer, SLICES, LOOPS);
     gluQuadricOrientation(*quadric, GLU_OUTSIDE);
     glTranslatef(0.0, 0.0, BREITE);
     gluDisk(*quadric, inner, outer, SLICES, LOOPS);
     gluQuadricOrientation(*quadric, GLU_OUTSIDE);
     glPopMatrix();
}

void drawPin(GLUquadricObj** quadric, const GLfloat radius,
const GLfloat height)
{
     glPushMatrix();
     glRotatef(-90.0, 1.0, 0.0, 0.0);
     gluCylinder(*quadric, radius, radius, BREITE / 2, SLICES,
LOOPS);
     gluQuadricOrientation(*quadric, GLU_INSIDE);
     gluDisk(*quadric, 0.0, radius, SLICES, LOOPS);
     gluQuadricOrientation(*quadric, GLU_OUTSIDE);
     glTranslatef(0.0, 0.0, BREITE / 2);
     gluDisk(*quadric, 0.0, radius, SLICES, LOOPS);
     gluCylinder(*quadric, STANGENBREITE, STANGENBREITE,
height, INNERSLICES, LOOPS);
     glTranslatef(0.0, 0.0, height);
     gluDisk(*quadric, 0.0, STANGENBREITE, INNERSLICES, LOOPS);
     glPopMatrix();
}

void drawAllPins(GLUquadricObj** quadric, const GLfloat radius,
const GLfloat height, const GLfloat gap)
{
     glPushMatrix();
     drawPin(quadric, radius, height);
     glTranslatef(-gap, 0.0, 0.0);
     drawPin(quadric, radius, height);
     glTranslatef(gap * 2, 0.0, 0.0);
     drawPin(quadric, radius, height);
     glPopMatrix();
}

void drawBitmapString(const GLfloat x, const GLfloat y, const
GLfloat z, void* font, char* string)
{
     char* c;
     glRasterPos3f(x, y, z);
     for (c = string; *c != '\0'; c++)
          glutBitmapCharacter(font, *c);
}

void drawBitmapInt(const GLfloat x, const GLfloat y, const
GLfloat z, void* font, const int number)
```

```
{
	char string[17];
	printf(string, "%d", number);
	drawBitmapString(x, y, z, font, string);
}

void populatePin(void)
{
	int i;
	disk* cur;
	GLfloat radius = 0.1f * disks;
	for (i = 0; i < disks; i++)
	{
		cur = (disk*)malloc(sizeof(disk));
		cur->color = (char)i % 6;
		cur->radius = radius;
		push(&pin[0], cur);
		radius -= 0.1;
	}
	duration = 0;
	draw = 0;
}

void clearPins(void)
{
	int i;
	disk* cur, * tmp;
	free(curdisk);
	curdisk = NULL;
	for (i = 0; i < 3; i++)
	{
		cur = pin[i].top;
		while (cur != NULL)
		{
			tmp = cur->prev;
			free(cur);
			cur = tmp;
		}
		pin[i].top = NULL;
		pin[i].bottom = NULL;
	}
}

void hanoiinit(void)
{
	GLfloat radius;
	speed = FSEM * FEM;
	radius = 0.1f * disks;
	config.pinradius = radius + 0.1f;
	config.gap = radius * 2 + 0.5f;
```

```
     config.pinheight = disks * BREITE + 0.2f;
     maxdraws = (2 << (disks - 1)) - 1;   //calculate minimum
number of moves
     populatePin();
     /* calculate actions; initialize action list */
     actqueue.head = NULL;
     hanoi(&actqueue, disks, 0, 1, 2);
     curaction = actqueue.head;
     curdisk = pop(&pin[(int)curaction->fromstack]);
     pos = 0.001;
}

void reset(void)
{
     clearPins();
     populatePin();
     /* reset actions */
     curaction = actqueue.head;
     curdisk = pop(&pin[(int)curaction->fromstack]);
     pos = 0.001;

}

void hanoicleanup(void)
{
     action* acur, * atmp;
     clearPins();
     acur = actqueue.head;
     do {
          atmp = acur->next;
          free(acur);
          acur = atmp;
     } while (acur != NULL);
     gluDeleteQuadric(quadric);
}
void setColor(const int color)
{
     switch (color) {
     case 0:
          glColor3f(1.0, 0.0, 0.0);
          break;
     case 1:
          glColor3f(0.0, 1.0, 0.0);
          break;
     case 2:
          glColor3f(1.0, 1.0, 0.0);
          break;
     case 3:
          glColor3f(0.0, 1.0, 1.0);
          break;
```

```
	case 4:
		glColor3f(1.0, 0.0, 1.0);
		break;
	case 5:
		glColor3f(0.0, 0.0, 0.0);
		break;
	}
}

void Init(void)
{
	const GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
	const GLfloat mat_shininess[] = { 50.0 };
	const GLfloat light_position[] = { 0.0, 1.0, 1.0, 0.0 };
	glShadeModel(GL_SMOOTH);
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);  /* draw 
polygons filled */
	glClearColor(1.0, 1.0, 1.0, 1.0);   /* set screen clear 
color */
	glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  /* 
blending settings */
	glCullFace(GL_BACK);     /* remove backsides */
	glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
	glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
	glLightfv(GL_LIGHT0, GL_POSITION, light_position);
	glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
	glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);
	glEnable(GL_COLOR_MATERIAL);
	glEnable(GL_DEPTH_TEST);
	quadric = gluNewQuadric();
	gluQuadricNormals(quadric, GLU_SMOOTH);
}

/*Is called if the window size is changed */
void Reshape(int width, int height)
{
	glViewport(0, 0, (GLint)width, (GLint)height);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(60.0, 1.0, 1.0, 75.0);
	glMatrixMode(GL_MODELVIEW);
}		}
		void Key(unsigned char key, int x, int y)
{
	switch (key)
	{
	case '1':
		disks = 1;
```

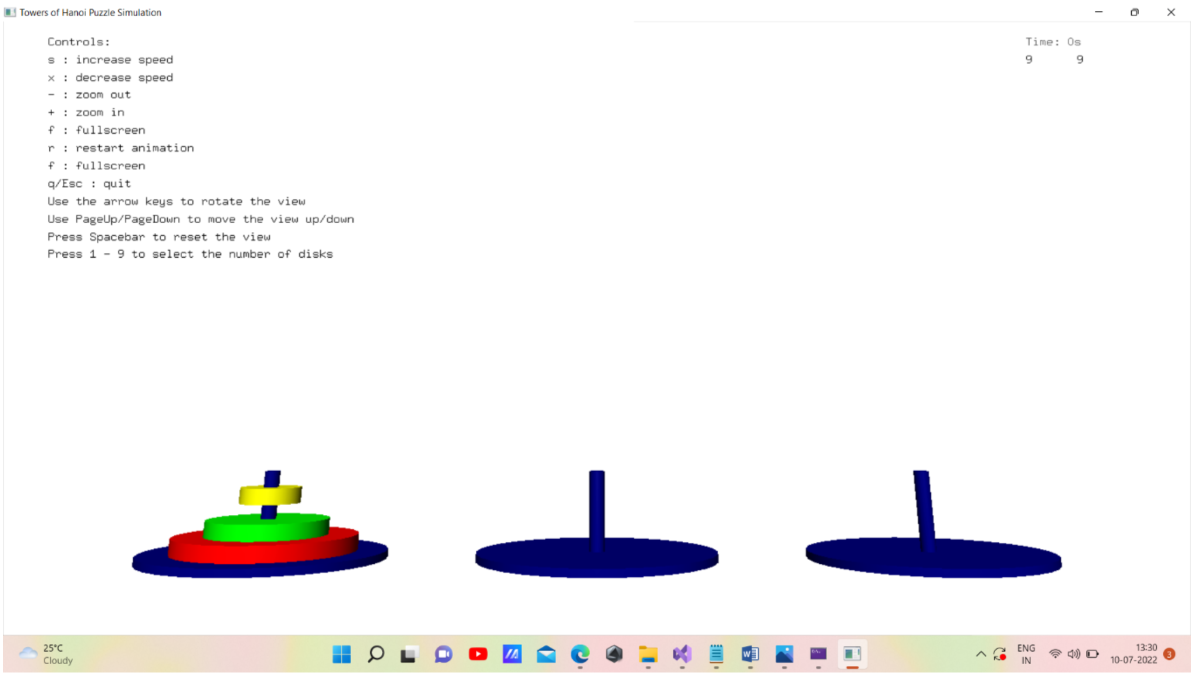
```
          hanoiinit();
          reset();
          offsetY = 0.9;
          zoom = -1.3;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          reset();
          break;
     case '2':
          disks = 2;
          hanoiinit();
          reset();
          offsetY = 1.1;
          zoom = -0.8;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          reset();
          break;
     case '3':
          disks = 3;
          hanoiinit();
          reset();
          offsetY = 1.3;
          zoom = -0.3;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          break;
     case '4':
          disks = 4;
          hanoiinit();
          reset();
          offsetY = 1.5;
          zoom = 0.8;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          break;
     case '5':
          disks = 5;
          hanoiinit();
          reset();
          offsetY = 1.7;
          zoom = 1.3;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          break;
     case '6':
          disks = 6;
          hanoiinit();
          reset();
          offsetY = 1.9;
          zoom = 1.8;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          break;
     case '7':
          disks = 7;
          hanoiinit();
          reset();
          offsetY = 2.1;
          zoom = 2.3;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          break;
     case '8':
          disks = 8;
          hanoiinit();
          reset();
          offsetY = 2.3;
          zoom = 2.8;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          break;
     case '9':
          disks = 9;
          hanoiinit();
          reset();
          offsetY = 2.5;
          zoom = 3.3;
          gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0,
0.0, 1.0, 0.0);
          break;
     case 27:
     case 'q':
          exit(EXIT_SUCCESS);
          break;
     case ' ':
          rotX = 0.0;
          rotY = 0.0;
          zoom = 0.0;
          offsetY = 1.5;
          speed = FSEM * FEM;
          break;
     case '+':
          zoom -= 0.1;
          break;
     case '-':
          zoom += 0.1;
          break;
     case 'r':
          reset();
          break;
     case 'f':
          if (fullscreen == 0)
          {
               glutFullScreen();
               fullscreen = 1;
          }
          else
          {
               glutReshapeWindow(800, 600);
               glutPositionWindow(50, 50);
               fullscreen = 0;
          }
          break;
     case 's':
          speed += 0.005;
          break;
     case 'x':
          speed -= 0.005;
          if (speed < 0.0)
               speed = 0.0;
          break;
     }
     glutPostRedisplay();
}

void SpecialKey(int key, int x, int y)
{
     switch (key)
     {
     case GLUT_KEY_UP:
          rotX -= 5;
          break;
     case GLUT_KEY_DOWN:
          rotX += 5;
          break;
     case GLUT_KEY_LEFT:
          rotY -= 5;
          break;
     case GLUT_KEY_RIGHT:
          rotY += 5;
          break;
     case GLUT_KEY_PAGE_UP:
          offsetY -= 0.1;
          break;
     case GLUT_KEY_PAGE_DOWN:
          offsetY += 0.1;
          break;
     }
     glutPostRedisplay();
```

```
}
int main(int argc, char* argv[])
{
	hanoiinit();
	atexit(hanoicleanup);
	glutInit(&argc, argv);
	//command line arguments for setting the number of disks
	if (argc > 1)
		disks = (int)argv[1];
	glutInitWindowPosition(0, 0);
	glutInitWindowSize(800, 600);
	glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE);
	if (glutCreateWindow("Towers of Hanoi Puzzle Simulation")
== GL_FALSE)
		exit(EXIT_FAILURE);
	Init();
	glutReshapeFunc(Reshape);
	glutKeyboardFunc(Key);
	glutSpecialFunc(SpecialKey);
	glutMouseFunc(mouse);
	glutDisplayFunc(Display);
	glutTimerFunc((unsigned)FEM, moveDisk, 0);
	glutTimerFunc(1000, timer, 0);
	glutMainLoop();
	return EXIT_SUCCESS;
}
```

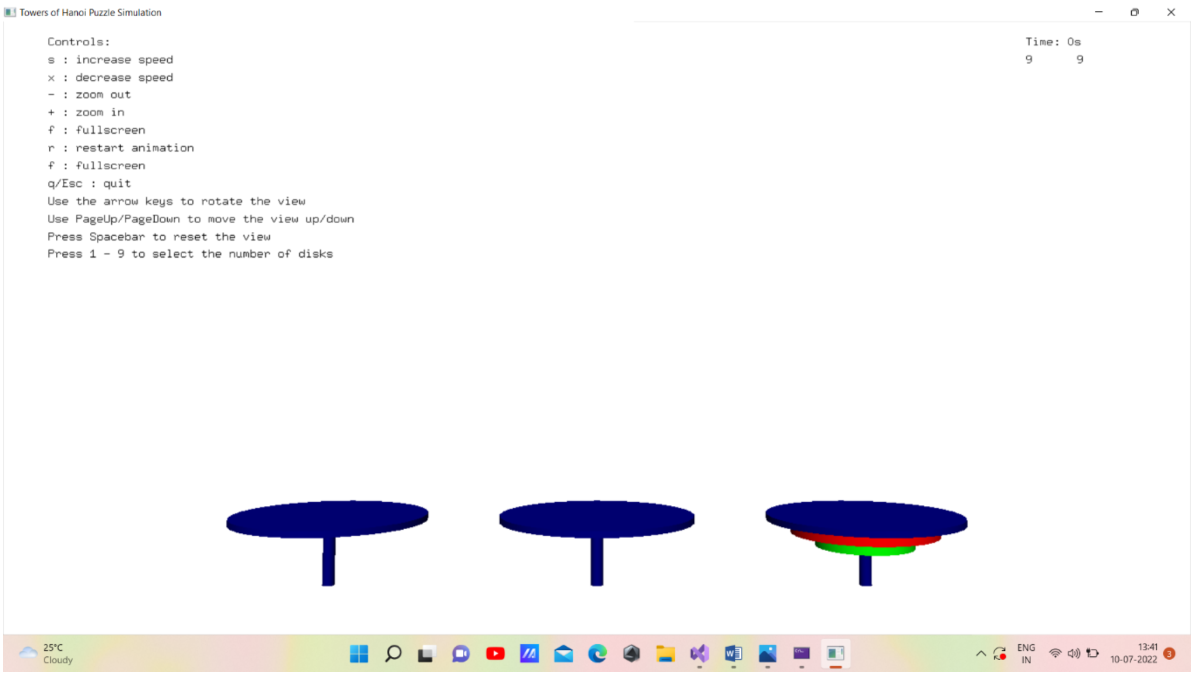
CHAPTER-5

SNAP SHOTS



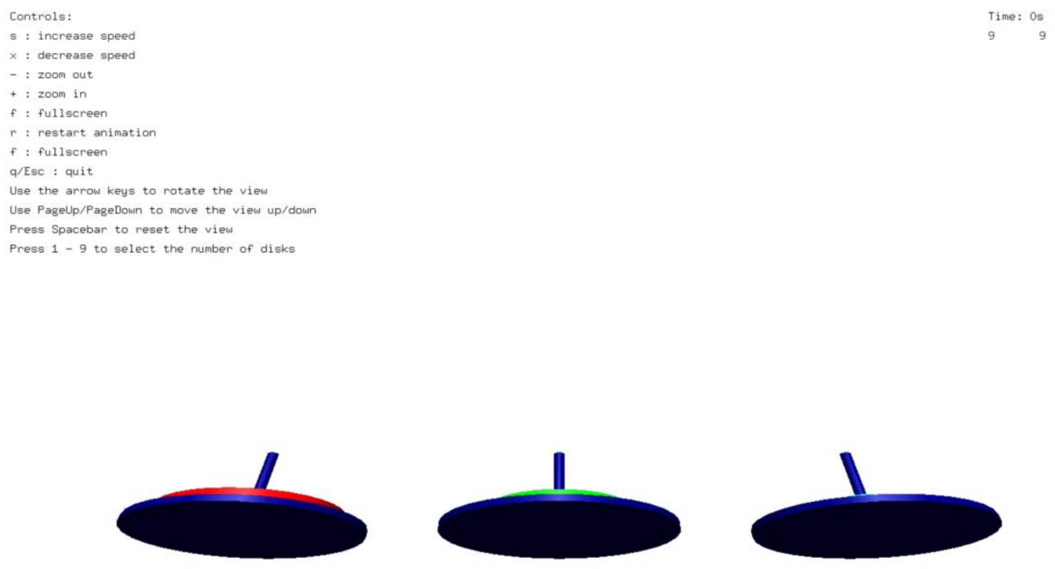
```
Controls:                                                        Time: 0s
s : increase speed                                               9      9
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks
```

Fig 5.1: Main Page With Game Instructions.



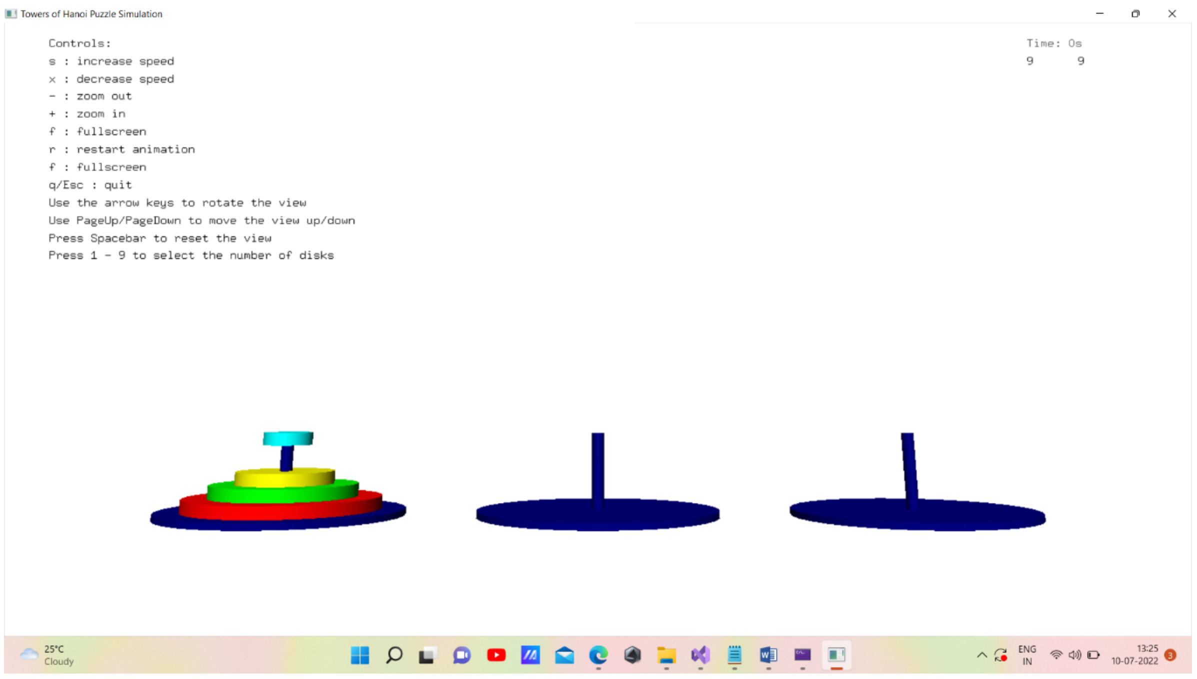
```
Controls:                                                        Time: 0s
s : increase speed                                               9      9
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks
```

Fig 5.2: Towers Rotated Downward using 'Down' Navigation Key.



```
Controls:                                                        Time: 0s
s : increase speed                                               9      9
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks
```

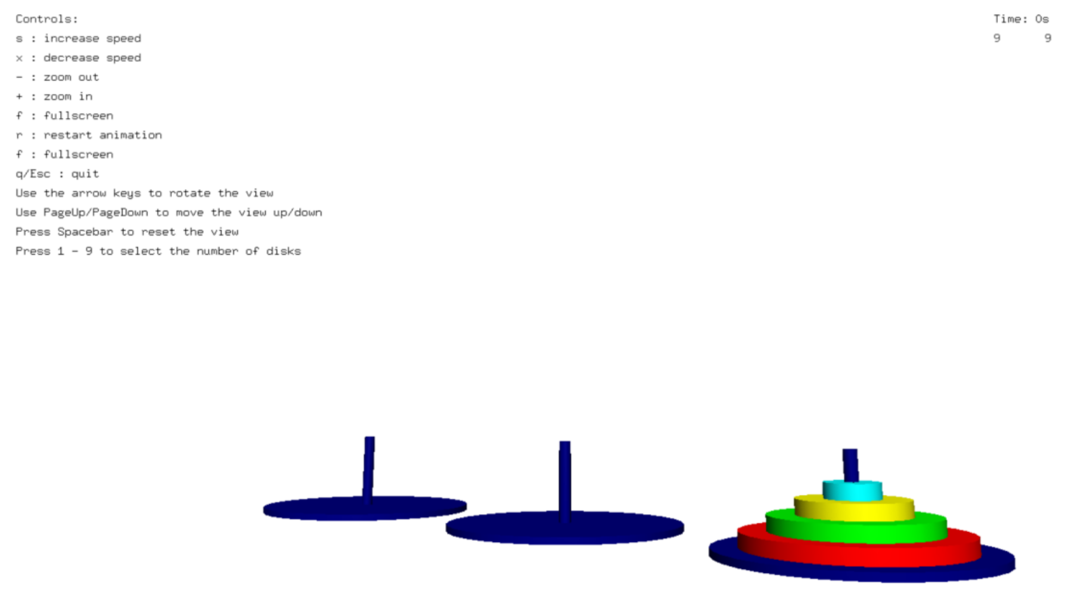
Fig 5.3: Towers Rotated upward using ‘Up’ Navigation Key.



```
Towers of Hanoi Puzzle Simulation

Controls:                                                        Time: 0s
s : increase speed                                               9      9
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks
```

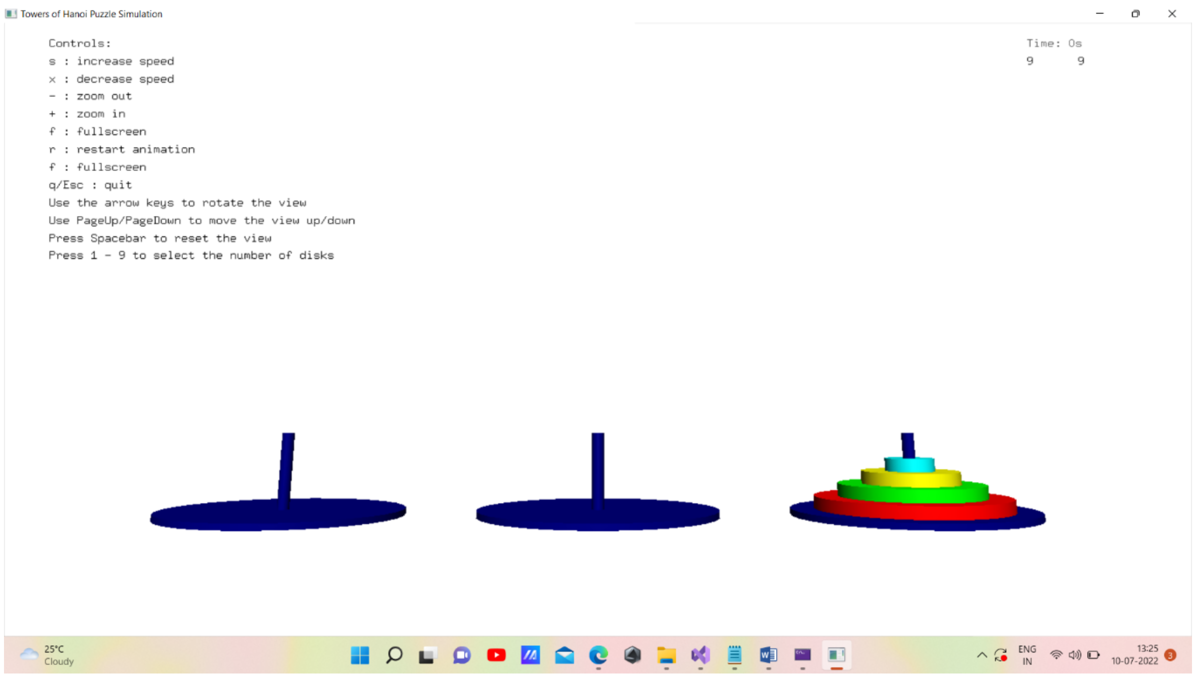
Fig 5.4: Selected Four Disk



```
Controls:
s : increase speed
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks
```

```
Time: 0s
9     9
```

Fig 5.5: Towers Rotated Clockwise using 'Left' Navigation Key



Towers of Hanoi Puzzle Simulation

```
Controls:
s : increase speed
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks
```

```
Time: 0s
9     9
```

Fig 5.6: Result

CHAPTER-6

CONCLUSION

By implementing this project I got to know how to use some of the built in functions effectively and how to interact efficiently and easily. I got a good exposure of how animation and simulations are developed, by working on this project.

One of the major accomplishments in the specification of OpenGL was the isolation of window system dependencies from OpenGL's rendering model. The result is that OpenGL is window system independent. Window system operations such as the creation of a rendering window and the handling of window system events are left to the native window system to define. Necessary interactions between OpenGL and the window system such as creating and binding an OpenGL context to a window are described separately from the OpenGL specification in a window system dependent specification.

This is to conclude that the project I undertook has been worked upon with sincere effort and has been completed successfully. The requirements and goals of the project have been achieved. The desktop application has been thoroughly tested and can now be implemented in the real world. By this project I hope to bring satisfaction to the users and meet their expectations.

CHAPTER-7

BIBLIOGRAPHY

[1] Edward Angel, "Interactive Computer Graphics", 5th edition, Pearson Education, 2005

[2] "Computer Graphics", Addison-Wesley 1997 James D Foley, Andries Van Dam, Steven K Feiner, John F Hughes.

[3] F.S.Hill and Stephen M.Kelly, "Computer Graphics using OpenGl ", 3rd edition

[4] http://www.opengl.org

[5] www.openglprojects.in

[6] http://www.openglprogramming.com/

[7] http://www.google.com/