CHAPTER 1 : INTRODUCTION

1.1 COMPUTER GRAPHICS :

Typically, the term Computer Graphics refers to several different interpretations:

- Representation and manipulation of image data by a computer.
- The sub-field of Computer Science which studies methods for digitally synthesizing visual content.

The phrase "Computer Graphics" was coined in 1960 by William Fetter, a graphic designer for Boeing. The field of computer graphics developed with the emergence of computer graphics hardware.

Although the term 'Computer Graphics' often refers to three – dimensional computer graphics, it also encompasses two-dimensional graphics and image processing.

As an academic discipline, computer graphics studies the manipulation of visual and geometric information using computational techniques. It focuses on mathematical and computational foundations of image generation and processing rather than purely aesthetic issues. Computer graphics is often differentiated from the field of visualization although the two fields have many similarities.

Concepts Involved:

- Rendering : It is defined as generating a 3D model by means of computer programs.
- 3D projection : It is a method of mapping three dimensional points to a two dimensional plane.
- 3D modelling : It is process of developing a mathematical, wireframe representation of any 3D object called a '3D model', via a specialized software.
- Tessellation: It is a process of filling up given 2D plane or 3D space with repetitive figures.

1.2 OpenGL

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 streamlined, hardware-independent interface to be implemented on many different hardware problems. Similarly, OpenGL doesn't provide high-level commands for describing models of three dimensional objects. With OpenGL, you must build up your desired model from a small set of geometric primitives – points, lines, polygons etc.

Libraries in OpenGL:

The earliest versions of OpenGL were released with a companion library called GLU, the OpenGL utility library. The GLU specification was last updated in 1998 and latest version uses the GLUT libraries.

OpenGL pipeline:

The rendering pipeline is the sequence of steps that OpenGL takes when rendering objects. The openGL rendering pipeline works in the following order.

- Prepare vertex array data and then render it.
- Vertex processing
    - Each vertex is acted upon by a Vertex shader. Each vertex in the stream is processed in turn into output vertex.
    - Optional primitive tessellation stages.
    - Optional Geometry shader primitive processing. The output is a sequence of primitives.
- Vertex Post-processing, the outputs of the last stage are adjusted or shipped to different locations.
- Transform Feedback occurs next.
- Primitive clipping, the perspective divide and the viewport transform to window space.
- Primitive assembly lining.
- Scan conversion and primitive parameter interpolation, which generates a number of fragments.
- A Fragment Shader processes each fragment. Each fragment generates a number of outputs.
- Per-Sample_Processing:
    - Scissor Test
    - Stencil Test
    - Depth Test
    - Blending
    - Logical Operation
    - Write Mask

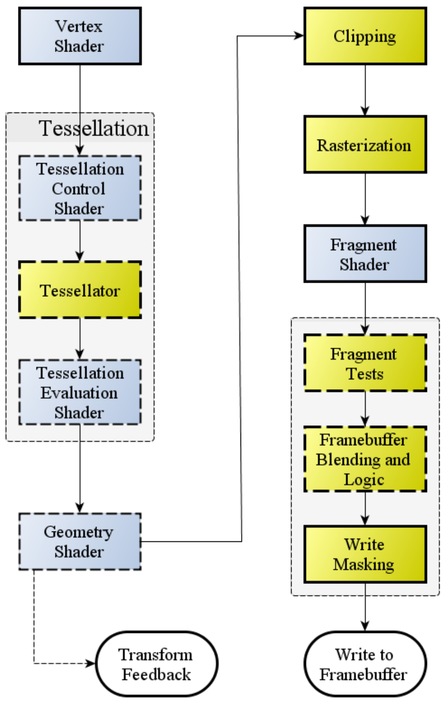


Figure 1.1 OpenGL pipeline

1.3 Overview of the project

3D world Simulations are designed to accurately model the flow of real life scenario in a graph-based network and model the development of perspective vision. They form a complex system that is open. Shows emergent phenomena, non-linear relationships and can contain feedback loops. An accurate and realistic 3D simulation could aid in developing more realistic game engines.

1.4 Problem Statement

Problem was regarding the depth information of the various objects including houses, trees stars which has been solved using the in-built depth specific algorithms and the associated display function calls.

Controlling the movement and the orientation of the camera has been done with interactive functions including keyboard and mouse functions.

1.5 Design

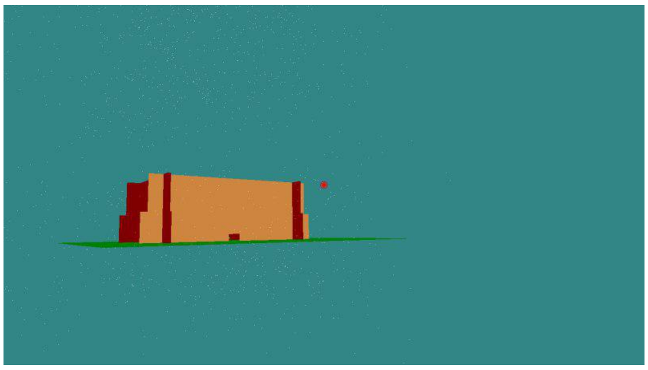


Figure 1.2: A 3d world simulation.

3D world Simulations are designed to accurately model the flow of real life scenario in a graph-based network and model the development of perspective vision. They form a complex system that is open. Shows emergent phenomena, non-linear relationships and can contain feedback loops. An accurate and realistic 3D simulation could aid in developing more realistic game engines.

There are two key components involved in the simulation of 3D world. The first is in controlling the movement as well as the orientation of the camera and the second one is in controlling the movement of the world objects, which make up the virtual world.

CHAPTER 2: SYSTEM REQUIREMENTS SPECIFICATION

2.1 Hardware Requirements

- OpenGL drivers
- Processors compatible : Intel Core 2 Duo CPU P8400, Intel Core i-series
- Graphics Memory: Works best on 1 GB or higher. Compatible with 800 MB and above.
- Display Adapter : Intel 4 series Express chipset family
- Refresh rate : Ideal 65 fps, supports 60Hz
- RAM : 2 GB or higher
- Processor speed : 2.4 GHz

2.2 Software Requirements

- Operating System: Microsoft Windows XP or higher.
- IDE : Microsoft Visual Studio Express 8 or higher.
- GLUT environment : V3.7.6 or higher.
- End user Requirement : Knowledge of operating a system.

Operating System/ files	**Windows**
glut.h	C:\program files\Microsoft SDKs\windows\v8.0A\gl\
glut32.lib	C:\program files\Microsoft SDKs\windows\v8.0A\lib\
glut32.h	C:\WINDOWS\system32

CHAPTER 3: IMPLEMENTATION

3.1 C HEADERS:

1. Stdio.h :
   It is the standard input and output library used for textual user interface.
2. string.h
   It is the standard string library that enables string functions defined by GNU.
3. math.h
   It is standard maths library that enables math functions defined by GNU

3.2 GLUT HEADERS AND FUNCTIONS:

1. glut.h : It is the glut environment header that enables graphic communication. It includes gl.h, glu.h, and glx.h automatically but it is safe to include separately to expand horizon of multiplatform compatibility.
2. glutInit(int argc, char **argv): Initialize GLUT and OpenGL. Pass in command-line arguments.
3. glutInitDisplayMode(unsigned int mode): Set GLUT's display mode. The mode is the logical-or of one or more of the following:
   Select one of the following three:
   GLUT RGB use RGB colors
   GLUT RGBA use RGBA colors
   GLUT INDEX use color-mapped colors (not recommended)
   Select one of the following two:
   GLUT SINGLE use single-bufferring or GLUT DOUBLE allow double-buffering (for smooth animation).
   Select the following for hidden surface removal:
4. GLUT DEPTH allow depth-buffering
5. There are a number of other options, which we have omitted. This last option makes depth-buffering possible. It is also necessary to enable the operation (see glEnable() below).
6. glutInitWindowSize(int width, int height): Set the window size. Also see: glutInitWindowPosition().
7. glutCreateWindow(char *title): Create window with the given title (argv[0] sets the name to the program's name).
8. glutMainLoop(void): This starts the main event loop. Control returns only through one of the callback functions given below.
9. glutDisplayFunc(void (*)(void)): Call the given function to redisplay everything (required).
10. glutReshapeFunc(void (*)(int width, int height)): Call the given function when window is resized (and originally created). See the function glViewport(), which typically should be called after each resizing.

11. glutIdleFunc(void (*)(void)): Call the given function with each refresh cycle of the display.
12. glutKeyboardFunc(void (*)(unsigned char key, int x, int y)): Call the given function when a keyboard key is hit. (This only handles ASCII characters. There is a different callback for function and arrow keys.)
13. glutPostRedisplay(void): Request that the image be redrawn.
14. glutSwapBuffers(void): Swap the front and back buffers (used in double buffering).
15. glClearColor(GLclampf R, GLclampf G, GLclampf B, GLclampf A): Specifies the background color to be used by glClear(), when clearing the buffer.
16. glClear(GLbitfield mask): Clears one or more of the existing buffers. The mask is a logical or of any of the following: GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_ACCUM_BUFFER or GL_STENCIL_BUFFER BIT.
17. glFlush(void): OpenGL normally saves or “buffers" drawing commands for greater efficiency. This forces the image to be redrawn, in case the next update will be far in the future.
18. glBegin(GLenum mode)...glEnd(void): Define an object, where mode may be any of:
    GL_POINTS: set of points
    GL_LINES: set of line segments (not connected)
    GL_LINE_STRIP: chain of connected line segments
    GL_LINE_LOOP: closed polygon (may self-intersect)
    GL_TRIANGLES: set of triangles (not connected)
    GL_TRIANGLE_STRIP: linear chain of triangles
    GL_TRIANGLE_FAN: fan of triangles (joined to one point)
    GL_QUADS: set of quadrangles (not connected)
    GL_QUAD_STRIP: linear chain of quadrangles
    GL_POLYGON: a convex polygon
19. glVertex*(...): Specify the coordinates of a vertex.
20. glNormal*(...): Specify the surface normal for subsequent vertices. The normal should be of unit length after the modelview transformation is applied. Call glEnable(GL_NORMALIZE) to have OpenGL do normalization automatically.
21. glColor*(...): Specify the color of subsequent vertices. This is normally used if lighting is not enabled.
22. glLineWidth(GLfloat width): Sets the line width for subsequent line drawing. The argument is the width of the line (in pixels). (Note that not all widths are necessarily supported.)
23. glRasterPos*(...): Set the current raster position (in 3D window coordinates) for subsequent glutBitmapCharacter() and other pixel copying operations.
24. glViewport(GLint x, GLint y, GLsizei width, GLsizei height): Sets the current viewport, that is, the portion of the window to which everything will be clipped. Typically this is the entire window: glViewport(0,0, win width, win height), and hence this should be called whenever the window is reshaped.
25. glMatrixMode(GLenum mode): Set the current matrix mode to one of GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE. The default is GL_MODELVIEW.
26. glLoadIdentity(void): Set the current matrix to the identity.
27. glPushMatrix(void): Make a copy of the current matrix and push it onto the stack.
28. glPopMatrix(void): Pop the top of the current matrix.

29. gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz):
    All arguments are of type double. This generates a transformation which maps world coordinates into camera coordinates. This is typically called before any drawing is done (just after glLoadIdentity()). The camera position is specified by the eye location, looking at the given center point, with the given up-directional vector. It composes this with the current (typically modelview) transformation.
30. glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z):
    Generate a rotation transformation about the specified vector (x, y, z) and compose it with the current transformation.
31. glTranslatef(GLfloat x, GLfloat y, GLfloat z):
    Generate a translation by vector (x; y; z) and compose it with the current transformation.
32. gluPerspective(GLfloat angle, GLfloat aspectRatio, GLgloat near, GLfloat far):
    In order to obtain a perspective viewing model.Similar to glFrustum() (It sets near = -1 and far = +1.)

CHAPTER 4 : ALGORITHM AND IMPLEMENTATION

4.1 ALGORITHM

Input: Key-press events from the user
Output: Multiple as discussed below.

Step 1 : Front screen showing project details.

Step 2: If Enter is pressed, Menu is shown

Step 3: Pressing H opens the Help Screen

Step 4: Pressing T starts the Virtual Simulation 1

Step 5: Pressing Y starts the Virtual Simulation 2

Step 6: Use the down arrow key to move backwards

Step 7: Use the up arrow key ti move forward

Step 8: Drag while pressing the left mouse button to change the camera orientation

Step 13: When the user is idle, the 3D world continues on undergoing various transformations.

4.2 IMPLEMENTATION

Main Function

This main function is capable of handling the arguments given in the argument list at the command prompt as we have used variable ‘argc’ for total number of arguments and ‘argv’ for the array of argument list.

Main function initializes the Display Mode, Window Size and position. Then it invokes the display function within glutDisplayFunc() as the call back function.

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

 // init GLUT and create window
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DEPTH| GLUT_SINGLE | GLUT_RGBA);
 glutInitWindowPosition(100,100);
 glutInitWindowSize(320,320);
 glutCreateWindow("Lighthouse3D - GLUT Tutorial");

 // register callbacks
 glutDisplayFunc(renderScene);
 glutReshapeFunc(changeSize);
 glutIdleFunc(renderScene);

 glutIgnoreKeyRepeat(1);
 glutKeyboardFunc(keys);
 glutSpecialFunc(pressKey);
 glutSpecialUpFunc(releaseKey);

 // here are the two new functions
 glutMouseFunc(mouseButton);
 glutMotionFunc(mouseMove);

 // timer func
 glutTimerFunc(10000,update,0);

 // OpenGL init
 glEnable(GL_DEPTH_TEST);

 glutFullScreen();
 init();
 // enter GLUT event processing cycle
 glutMainLoop();

 return 1;
}
```

Keyboard Function

This Interactive function is invoked at the main program within glutKeyboardFunc() and it is repeatedly called during the execution of the program and hence handles the keyboard interrupts from the users.
When a key is pressed, the ASCII code of the key and the position of the screen pointer at the time of interruption will be sent to the function.

```
 void keys(unsigned char key ,int x , int y){
  if(key == 'e' || key == 'E')
  exit(0);

  if(key == 13){

  flag = 2;
  renderScene();
  }

  if(key == 'h' || key == 'H'){
  flag =3;
  renderScene();
  }

  if(key == 't' || key == 'T'){
  flag = 4;
  renderScene();
  }

  if(key == 'y' || key == 'Y'){
  flag = 5;
  renderScene();
  }
  }

void pressKey(int key, int xx, int yy) {

 switch (key) {
 case GLUT_KEY_UP : deltaMove = deltaMoveConstant; break;
 case GLUT_KEY_DOWN : deltaMove = -deltaMoveConstant; break;

 }
}
void releaseKey(int key, int x, int y) {

 switch (key) {
 case GLUT_KEY_UP :
 case GLUT_KEY_DOWN : deltaMove = 0;break;
 }}
```

Display Text on Screen

The setFont() function is used to set the type of the font we are using. The drawstring() function takes the position of the string to be displayed on the screen in X Y Z coordinates and the string to display.

```
void *currentfont;
void setFont(void *font)
{
        currentfont=font;

}
void drawstring(float x,float y,float z,char *string)
{
         char *c;
         glRasterPos3f(x,y,z);
         for(c=string;*c!='\0';c++)
        {
                glColor3f(0.0,0.0,0.0);
                glutBitmapCharacter(currentfont,*c);
        }
}
```

Display Functions

We have used five display functions in this program. The first display function is called in the main program using glutDisplayFunc() call back function. This display function controls the displaying of the Welcome screen. The second display function controls the display of the application menu where as the third display function displays the help screen. The fourth and the fifth display functions correspond to the simulation 1 and simulation 2 in the project which control the display of the 3D world.

```
/* display function */
void renderScene(void){

 if(flag == 1)
 display1();
 else if(flag == 2)
 display2() ;
 else if(flag == 3)
 display3();
 else if(flag == 4)
 display5();
 else if(flag == 5)
 display4();
 assert();
 }
/* to display 1st simulation screen*/
void display5(void){


 if (deltaMove)
```

```
computePos(deltaMove);

if(red > 0.0 && green > 0.0 && blue > 0.0 && day)
{
glClearColor(red,green, blue,1.0);
red-=0.0002;
green-=0.0002;
blue-=0.0002;

}
else
day = false;

if(red < 0.749 && blue < 0.847 && green < 0.847 && !day ){

glClearColor(red,green,blue,1.0);
red+= 0.0002 ;
green += 0.00022616822;
blue += 0.00022616822;
}
else
day = true;

// Clear Color and Depth Buffers
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Reset transformations
glLoadIdentity();
// Set the camera
gluLookAt( x, 2.5f, z,
x + lx, ly+ 2.5f , z+lz,
0.0f, 1.0f, 0.0f);

// Draw ground

//glRotatef(anglee,0,0,1);
glColor3f(0.0f, 0.5f, 0.0f);
glBegin(GL_QUADS);
glVertex3f(-100.0f, 0.0f, -100.0f);
glVertex3f(-100.0f, 0.0f, 100.0f);
glVertex3f( 100.0f, 0.0f, 100.0f);
glVertex3f( 100.0f, 0.0f, -100.0f);
glEnd();

// draw Castle
glPushMatrix();
glTranslatef(0,0,15);
drawCastle();
glPopMatrix();

// draw pointer
glPushMatrix();
```

```
glLoadIdentity();
glColor3f(1.0,0.0,0.0);
glTranslatef(0,0,0.0);
glScalef(0.01,0.01,2);
glutWireSphere(0.1,10,12);
glPopMatrix();

glPushMatrix();

glPushMatrix();
glTranslatef(0.0,0.0,-15);
//glRotatef(iangles,0,1,0);
// draw House
glPushMatrix();
drawhouse(10.5,-15.0);
drawhouse(-10.5 , -15.0);
drawhouse(10.5,-5.0);
drawhouse(-10.5 , -5.0);
glPopMatrix();


// draw statue
drawStatue();

// draw baton
glPushMatrix();
glTranslatef(0,0,-10);
drawBaton(-10,0);
drawBaton(10,0);
drawBaton(0,-10);
drawBaton(0,10);

glPopMatrix();

glPopMatrix();
glPopMatrix();

// drawStars
glPushMatrix();
glColor3f(1.0,1.0,1.0) ;
glPointSize(1.0);
glTranslatef(0,0,delta);
glBegin(GL_POINTS);
for(int i = 0 ; i < 4000 ; i++ )
glVertex3f(p[i].x,p[i].y,p[i].z);
glEnd();
glPopMatrix();

if(left)
anglee -= 0.1f;
else
anglee += 0.1f;
```

```
if(left){
if(anglee < -3.0f)
left = false;
}
else if(anglee > 3.0f)
left = true;

glFlush();
assertDistance();
//glutSwapBuffers();
}
```

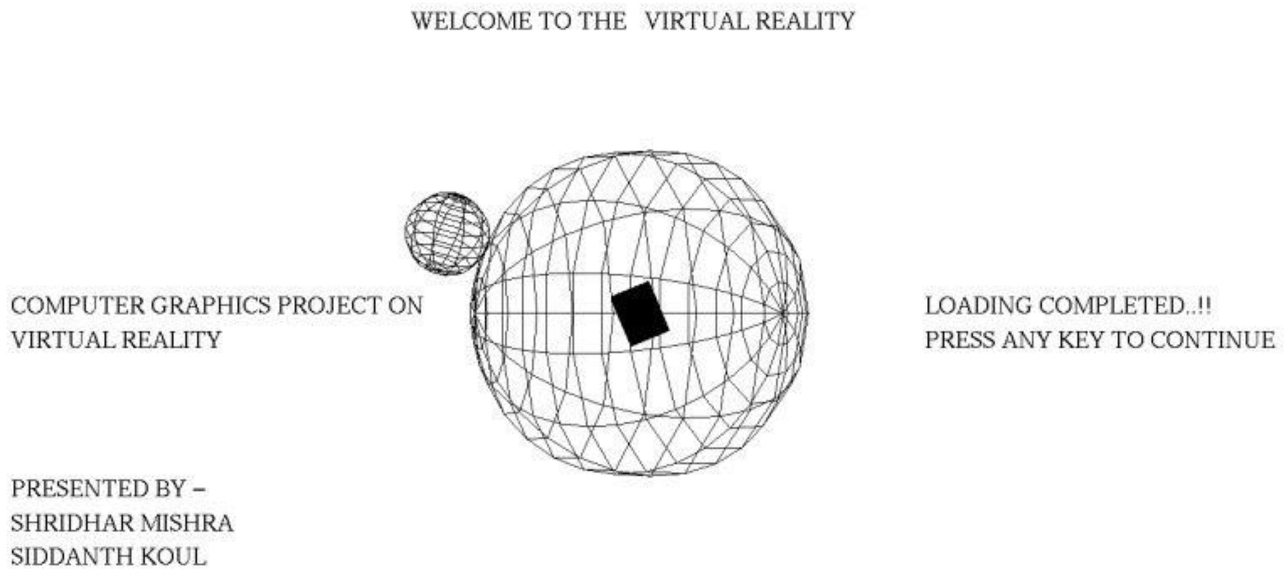
/*to display 2nd simulation screen */

```
void display4(void){
if (deltaMove)
computePos(deltaMove);

if(red > 0.0 && green > 0.0 && blue > 0.0 && day)
{
glClearColor(red,green, blue,1.0);
red-=0.002;
green-=0.002;
blue-=0.002;

}
else
day = false;

if(red < 0.749 && !day ){

glClearColor(red,green,blue,1.0);
red+= 0.002 ;
green += 0.0022616822;
blue += 0.0022616822;
}
else
day = true;

// Clear Color and Depth Buffers
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Reset transformations
glLoadIdentity();
// Set the camera
gluLookAt( x, 1.0f, z,
x + lx, ly+ 1.0f , z+lz,
0.0f, 1.0f, 0.0f);

// Draw ground
```

```
 glColor3f(0.87f, 0.72f, 0.529f);
 glBegin(GL_QUADS);
 glVertex3f(-100.0f, 0.0f, -100.0f);
 glVertex3f(-100.0f, 0.0f, 100.0f);
 glVertex3f( 100.0f, 0.0f, 100.0f);
 glVertex3f( 100.0f, 0.0f, -100.0f);
 glEnd();

// Draw 36 SnowMen

 for(int i = -3; i < 3; i++){
 glPushName(i);
 for(int j=-3; j < 3; j++) {
 glPushMatrix();
 glTranslatef(i*10.0,1.8,j * 10.0);
 glRotatef(anglee +i*j,0,0,1);
 glScalef(0.02,0.02,0.02);
 drawtree();
 glPopMatrix();

 glPushMatrix();
 glTranslatef(i*10.0,1.8,j * 10.0+0.05);
 glRotatef(anglee +i*j,0,0,1);
 glScalef(0.02,0.02,0.02);
 drawtree();
 glPopMatrix();


 }
 glPopName();
 }



 // drawBall1
 glPushMatrix();
 glColor3f(0.5,0,0);
 glTranslatef(0.0,0.0,20.0);
 rollingball(0,0,1 ,distance, 0.0);
 glPopMatrix();

 // drawBall2
 glPushMatrix();
 glTranslatef(20.0,0.0,0.0);
 rollingball(1,0,0 ,0.0, -distance);
 glPopMatrix();

 // drawBall3
 glPushMatrix();
 glTranslatef(0.0,0.0,-20.0);
 rollingball(0,0,1 ,-distance,0.0 );
 glPopMatrix();
```

```
// drawBall4
glPushMatrix();
glTranslatef(-20.0,0.0,0.0);
rollingball(1,0,0 ,0.0, distance);
glPopMatrix();


// drawHouse
glPushMatrix();
drawhouse(2.5,-15.0);
drawhouse(-2.5 , -15.0);
glPopMatrix();

// drawStars
glPushMatrix();
glColor3f(1.0,1.0,1.0) ;
glPointSize(1.0);
glTranslatef(0,0,delta);
glBegin(GL_POINTS);
for(int i = 0 ; i < 4000 ; i++ )
glVertex3f(p[i].x,p[i].y,p[i].z);
glEnd();
glPopMatrix();

if(left)
anglee -= 0.1f;
else
anglee += 0.1f;

if(left){
if(anglee < -3.0f)
left = false;
}
else if(anglee > 3.0f)
left = true;

glFlush();
assertDistance();
//glutSwapBuffers();
}
```

CHAPTER 5 : SCREENSHOTS



WELCOME TO THE VIRTUAL REALITY

COMPUTER GRAPHICS PROJECT ON
VIRTUAL REALITY

LOADING COMPLETED..!!
PRESS ANY KEY TO CONTINUE

PRESENTED BY –
SHRIDHAR MISHRA
SIDDANTH KOUL

Fig 5.1: Front screen showing Project details.

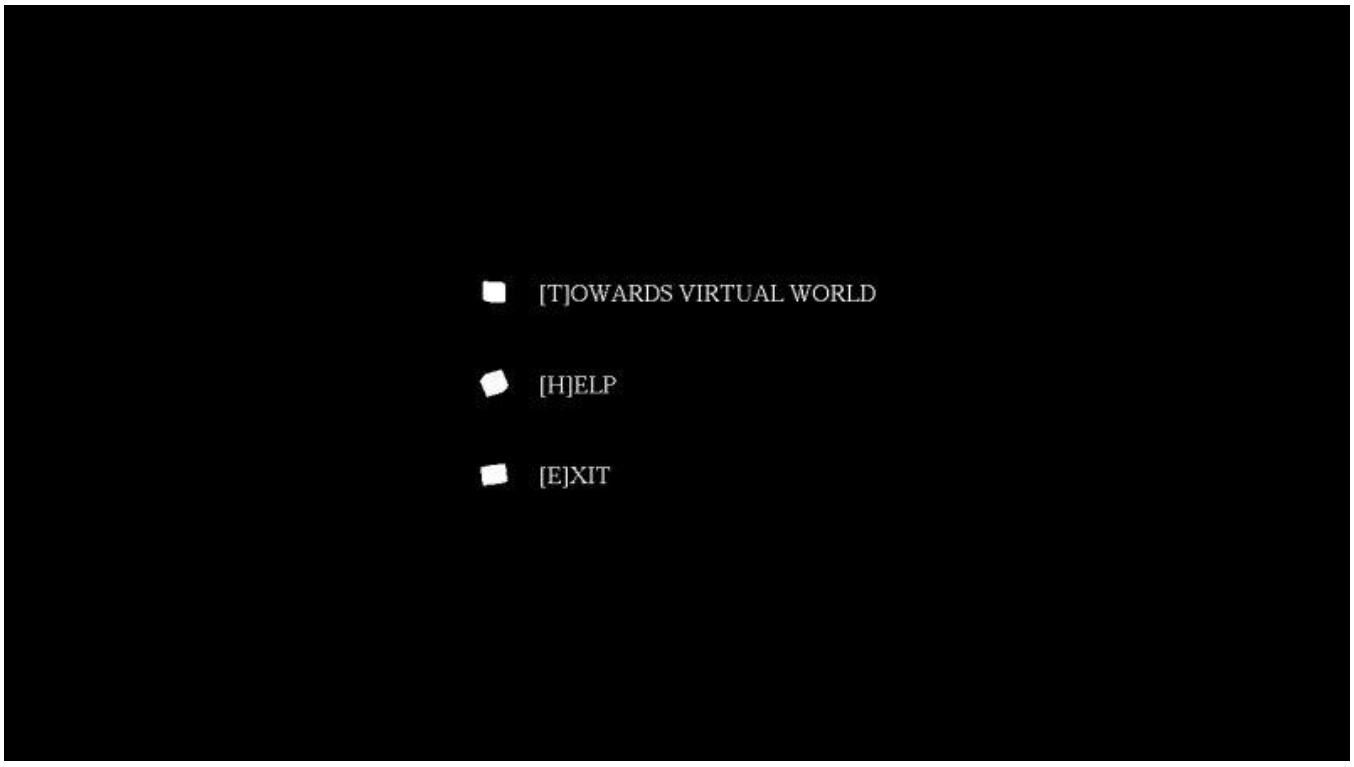
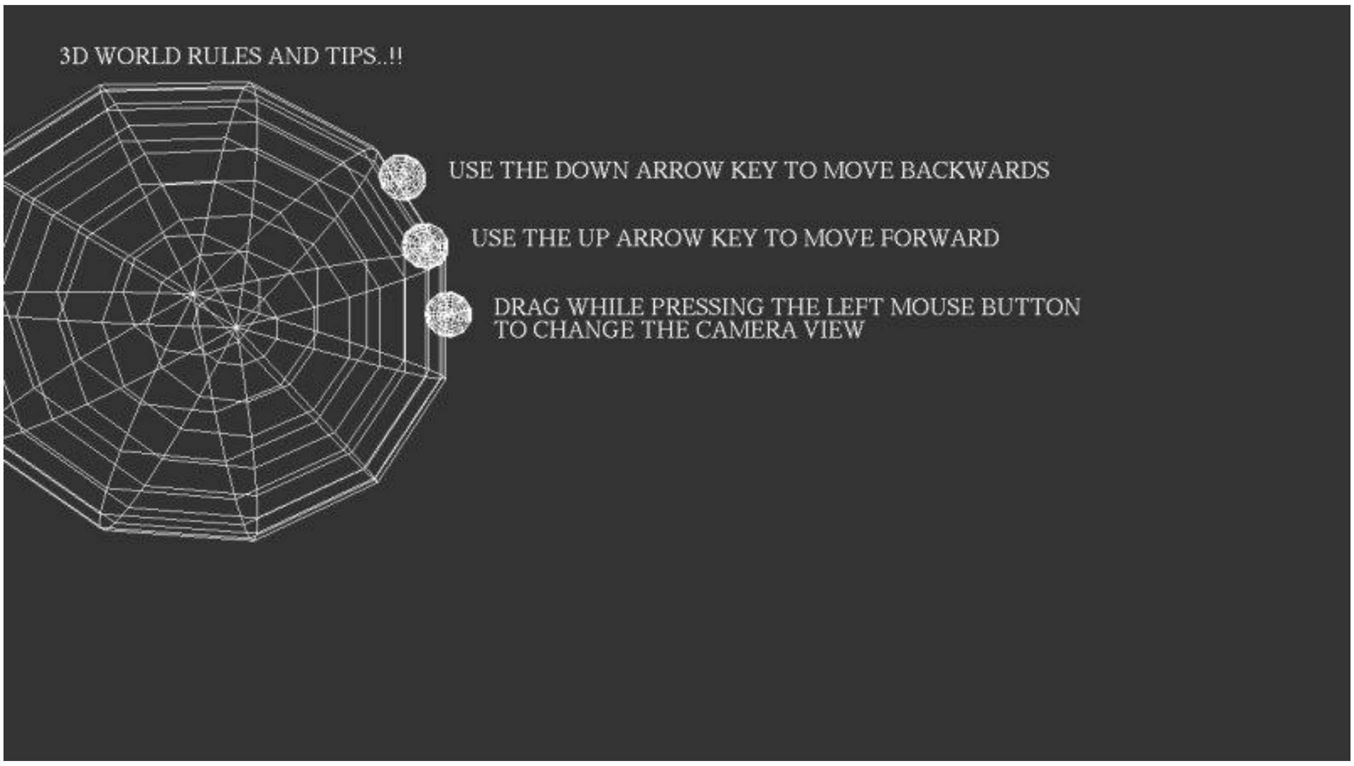


Fig 5.2 : Second screen giving the user menu.



3D WORLD RULES AND TIPS..!!

USE THE DOWN ARROW KEY TO MOVE BACKWARDS

USE THE UP ARROW KEY TO MOVE FORWARD

DRAG WHILE PRESSING THE LEFT MOUSE BUTTON
TO CHANGE THE CAMERA VIEW

Fig 5.3 : Third screen giving instructions to the user.

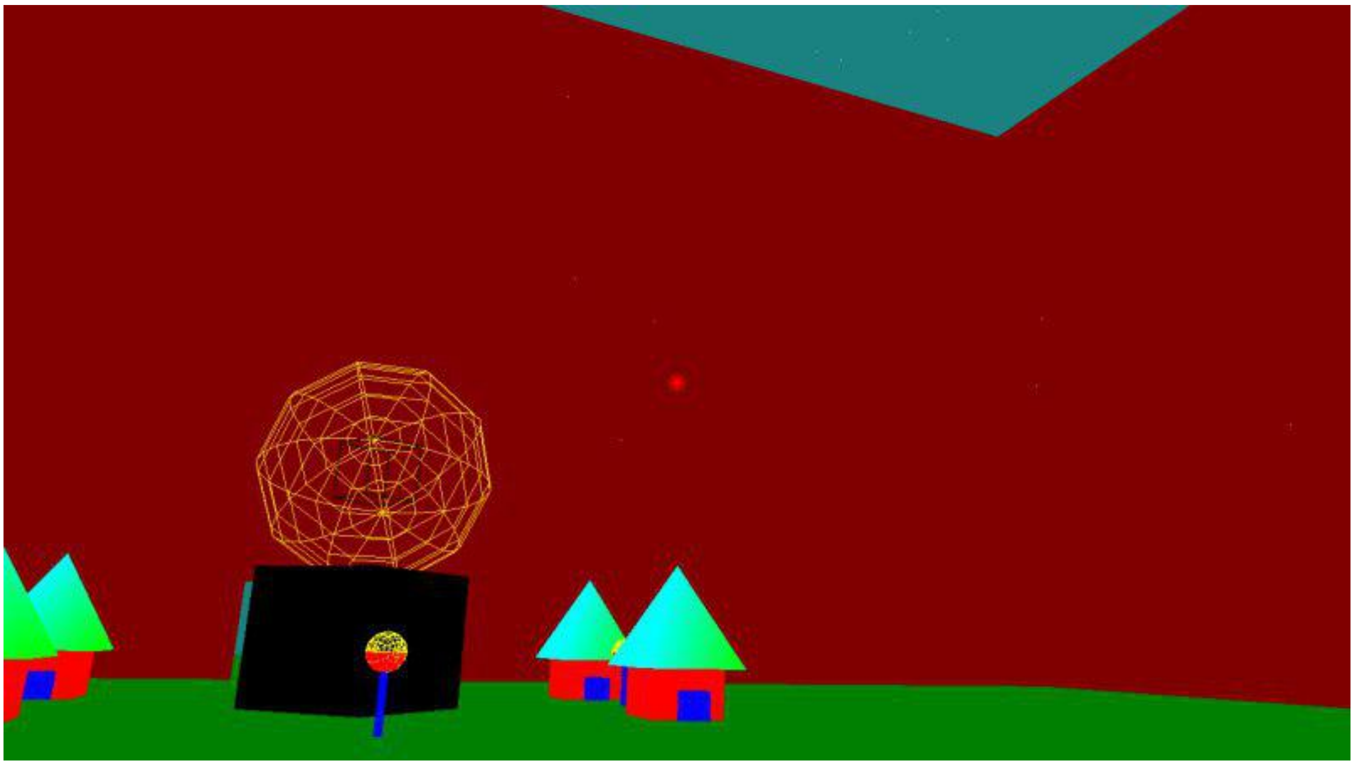


Fig 5.4: Animated screen showing the first 3D simulation

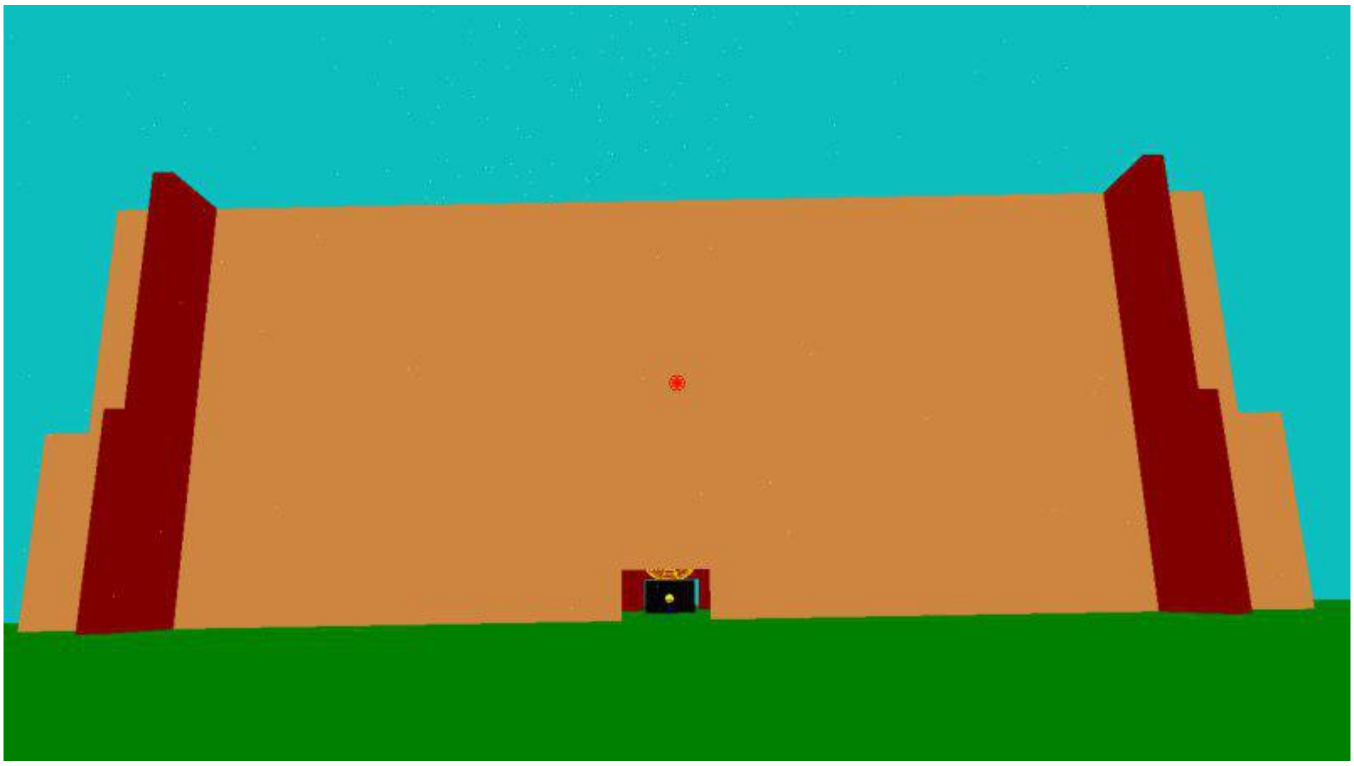


Fig 5.5: Animated screen showing the another view of first 3D simulation

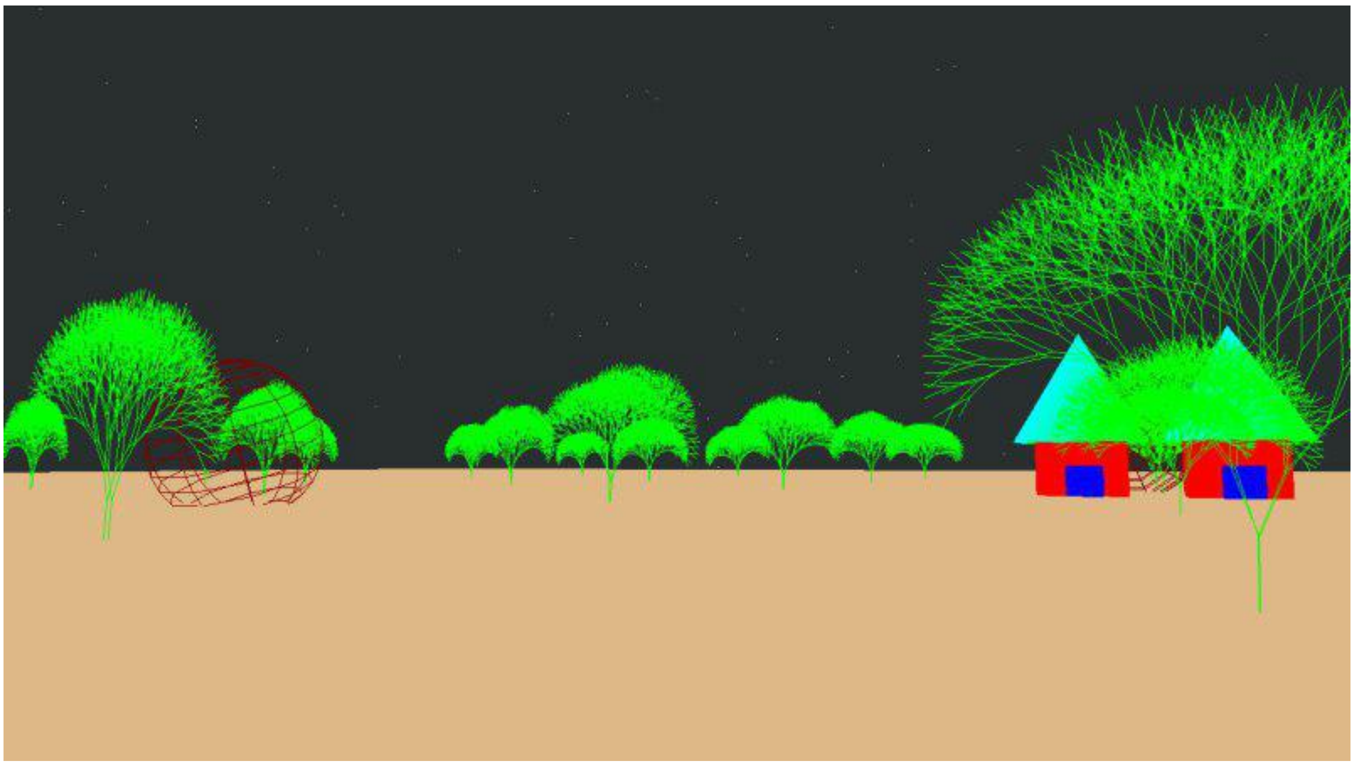


Fig 5.6: Animated screen showing of second 3D simulation

Chapter 6: CONCLUSION

The project **Virtual Reality** was completed and executed successfully. It presents the modelling, analysis and implementation of a 3D world simulation.

The transformation effects of OpenGL have been used satisfactorily to demonstrate the motion of camera and their response to the surrounding 3D world.

The following project can be modified by simulating interactions between the various world objects creating a real life scenario.

BIBLIOGRAPHY

1. Interactive Computer Graphics A Top-Down Approach with OpenGL -Edward Angel, 5th Edition, Addison-Wesley, 2008.
2. Computer Graphics Using OpenGL – F.S. Hill, Jr. 2nd Edition, Pearson Education, 2001.
3. Computer Graphics – James D Foley, Andries Van Dam, Steven K Feiner, John F Hughes, Addison-Wesley 1997.
4. Computer Graphics - OpenGL Version – Donald Hearn and Pauline Baker, 2nd Edition, Pearson Education, 2003.