RV COLLEGE OF ENGINEERING®
BENGALURU – 560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

" 2D-Snake Game Using OpenGL "

COMPUTER GRAPHICS LAB (16CS73)

OPEN ENDED EXPERIMENT REPORT

VII SEMESTER

2020-2021

Submitted by

RISHAB V ARUN 1RV17CS122

Under the Guidance of

Prof. Mamatha T
Department of CSE, R.V.C.E.,
Bengaluru - 560059

ABSTRACT

A 2d graphics-based snake game is a great start for a student who starts learning computer graphics & visualization. The development of the game has large scope to learn computer graphics from scratch. We used OpenGL utility toolkit to implement the algorithm, written it in C language.

The game developed in this project is one of the most well known games. The snake game in which the objective is to grow the snake as long as possible by eating food/apple while making sure that the snake's head does not make contact with its body or the boundary of the game window. This game can be implemented with OpenGL functions to understand the application of OpenGL in developing applications.

RV COLLEGE OF ENGINEERING®, BENGALURU -560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the **Open-Ended Experiment** titled "2D-Snake Game Using OpenGL" has been carried out by **Rishab V Arun (1RV17CS122),** bonafide student of RV College of Engineering, Bengaluru, has submitted in partial fulfillment for the **Internal Assessment of Course: COMPUTER GRAPHICS LAB (16CS73)** during the year 2020-2021. It is certified that all corrections/suggestions indicated for the internal Assessment have been incorporated in the report.

Prof. Mamatha T
Faculty Incharge,
Department of CSE,
R.V.C.E., Bengaluru –59

Dr. Ramakanth Kumar P
Head of Department,
Department of CSE,
R.V.C.E., Bengaluru–59

ACKNOWLEDGEMENT

While presenting my project on 2D Snake game using OpenGL, I feel that it is my duty to acknowledge the help rendered by various people for the successful completion of this project. I would like to convey my thanks to the principal & HOD(CSE) Dr. Ramakanth Kumar P (RVCE, Bangalore) for being kind enough to provide me an opportunity to do a project in this institution. I would greatly mention the enthusiastic influence provided by Prof. Mamatha T, my project guide, for her ideas and co-operation showed on me during the venture and making this project a great success. I am very much pleasured to express my sincere gratitude to the friendly co-operation showed by all the staff members of Computer Science Department, RVCE.

RV COLLEGE OF ENGINEERING® , BENGALURU - 560059
(Autonomous Institution Affiliated to VTU)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DECLARATION

I, **Rishab V Arun (1RV17CS122)** student of Seventh Semester B.E., Computer Science and Engineering, R.V. College of Engineering, Bengaluru hereby declare that the mini-project titled **"2D-Snake Game Using OpenGL "** has been carried out by me and submitted in partial fulfillment for the **Internal Assessment of Course: COMPUTER GRAPHICS LAB (16CS73) - Open-Ended Experiment** during the year 2020-2021. I do declare that matter embodied in this report has not been submitted to any other university or institution for the award of any other degree or diploma.

Place: Bengaluru **Rishab V Arun**

Date:

CONTENTS

Chapter 1

INTRODUCTION

Computer Graphics

Computer graphics is one of the most exciting and rapidly growing computer fields. It is also an extremely effective medium for communication between man and computer; a human being can understand the information content of a displayed diagram or perspective view much faster than he can understand a table of numbers or text containing the same information. Thus computer graphics is being used more extensively.

There is a lot of development in hardware and software required to generate images, and nowadays the Cost of hardware and software is dropping rapidly. Due to this, interactive computer graphics is becoming available to more and more people.

Computer graphics started with the display of data on hardcopy plotters and cathode ray tube (CRT) screens soon after the introduction of computers themselves. It has grown to include the creation, storage and manipulation of models 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 objects and their displayed images by using input devices, such as a keyboard, mouse, or touch sensitive panel on the screen. The handling of such devices is included in the study of computer graphics, because of the close relationship between the input devices and the display.

Computer Graphics Architecture

Early graphics system used general purpose computers with the standard von Neumann architecture, such computers are characterized by a single processing unit that processes a single instruction at a time. The display in these systems was based on a calligraphic CRT display that included the necessary circuitry to generate the line segment connecting two points. The job of first computer was to run the application program and to compute the endpoints of the line segment in the image. This information had to be sent to the display at a rate high enough to avoid flickers on the display.

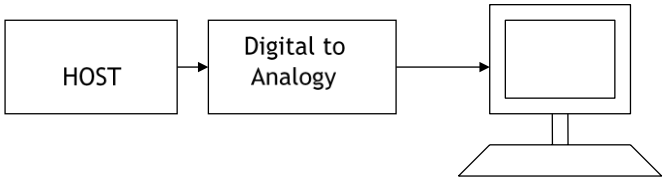


Figure 1.1 Early Graphics System

To reduce the burden of basic systems we use the following support applications.

1. Display processor
2. Pipeline architecture
3. Graphics pipeline
4. Vertex processing
5. Clipping and primitive assembly
6. Rasterization
7. Fragment processing

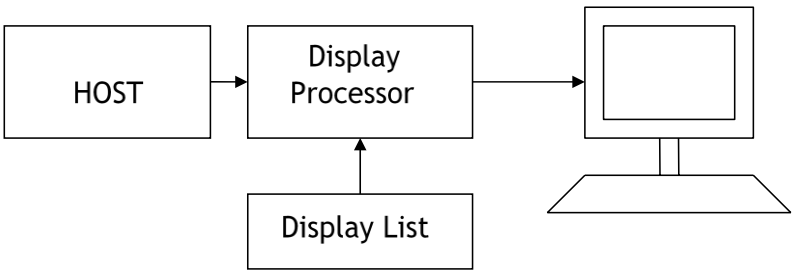


Figure1.2 Display Processor Architecture

Application of Computer Graphics

The development of computer graphics has been driven both by the needs of user community and by advances in hardware and software.
The four major areas are:

1. Display of information

Classical graphics techniques are used as a medium to convey information among people. In ancient times Greeks were able to convey their architectural ideas graphically even though the relevant mathematics was not developed.

Today the same type of information is generated by architectures mechanical designers and drafts people using computer based drafting system.

2. Design

Professionals such as engineering and architecture are concerned with design. Starting with a set of specifications, engineers and architects seek cost effective and esthetic solution that satisfies the specifications. Design is an iterative process.

3. Simulation and Animation

Once graphics system evolved to be capable of generating sophisticated images in real time, engineers and researchers began to use them as simulators. Graphical flight simulator as proved to increase safety and reduce training expenses, Use of graphics for animation in television, motion pictures and advertising industry.

4. User interfaces

Our interaction with computers has become dominated by visual paradigm that includes windows, icons, menus and a pointing device such as a mouse. More recently millions of people have become internet users. Their access is through graphical network browsers such as Firefox and internet explorer.

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 three-dimensional applications.
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, you must work through whatever windowing system controls the particular hardware you're using.
Similarly, OpenGL doesn't provide high-level commands for 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) provides many of the modeling 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 toolkit, Open Inventor, which is built atop OpenGL, and is available separately for many implementations of OpenGL.
OpenGL (Open Graphics Library) is a standard specification defining a crosslanguage, cross-platform API for writing applications that produce 2D and 3D. 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.

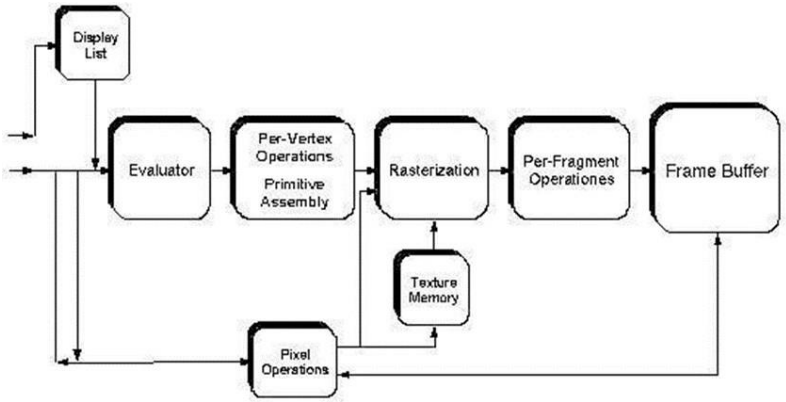


Figure1.3 Order of operation in OpenGL rendering pipeline

2D-Snake game

This mini project under Computer Graphics & Visualization Laboratory is an implementation of a 2D Snake game using the OpenGL Graphics Library and GLUT Toolkit.

Scope

The player needs to control the snake using the arrow keys in the keyboard. He is able to adjust the speed of the game using ‘a’ and ‘s’ keys. He can play the game in grid mode or no-grid mode by toggling using the ‘g’ key. The player’s objective is to score as much points as possible while making sure that the snake is alive.

Objective

The objective of the game is to score as much points as possible by making the snake eat apple/food, while making sure that the snake does not die. The snake can die if the following to conditions occur:

1. The snake head collides with the boundary
2. The snake head collides with its body

The player has the control to tune the speed of the snake using keyboard keys. He also has the option whether to play the game in grid mode or no-grid mode. When the game ends after the snake dies, the player’s score is displayed in a dialog box.

Keyboard function: -

- Pressing UP key moves snake up and down if press DOWN key.
- Pressing LEFT key moves snake left and right if press RIGHT key.
- ‘a’ and ‘s’ key to decrease and increase the speed of the snake respectively.
- ‘g’ key to toggle display of the grid.
- ‘q’ or ‘ESC’ key to quit.

Chapter 2

BASIC OpenGL COMMANDS & FUNCTIONS

Commands

void glBegin(GLenum mode)

Marks the beginning of a vertex-data list that describes a geometric primitive. The type of primitive is indicated by mode, which can be any of the values shown in

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

void glEnd(void);

Marks the end of a vertex-data list.

Viewing

The viewing transformation

void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

Defines a viewing matrix and multiplies it to the right of the current matrix. The desired viewpoint is specified by eyex, eyey, and eyez. The centerx, centery, and centerz arguments specify any point along the desired line of sight, but typically they're some point in the center of the scene being looked at. The upx, upy, and upz arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume).

Modeling transformation

Translate

void glTranslate{fd}(TYPEx, TYPE y, TYPEz);

Multiplies the current matrix by a matrix that moves (translates) an object by the given x, y, and z values (or moves the local coordinate system by the same amounts).

Rotate

void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);

Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation in degrees.

Scale

void glScale{fd}(TYPEx, TYPE y, TYPEz);

Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each x, y, and z coordinate of every point in the object is multiplied by the corresponding argument x, y, or z. With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or reflected by the x, y, and z factors, and the associated object is transformed with them.

Projection transformation

Perspective projection

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

Creates a matrix for a perspective-view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the parameters: (left, bottom, -near) and (right, top, -near) specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane; near and far give the distances from the viewpoint to the near and far clipping planes. They should always be positive.

void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);

Creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. fovy is the angle of the field of view in the x-z plane; its value must be in the range [0.0,180.0]. aspect is the aspect ratio of the frustum, its width divided by its height. near and far values the distances between the viewpoint and the clipping planes, along the negative z-axis. They should always be positive.

Orthographic projection

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (left, bottom, -near) and (right, top, -near) are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewport window, respectively. (left, bottom, -far) and (right, top, -far) are points on the far clipping plane that

are mapped to the same respective corners of the viewport. Both near and far can be positive or negative.

Viewing volume clipping

Viewport transformation

void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);

Defines a pixel rectangle in the window into which the final image is mapped. The (x, y) parameter specifies the lower-left corner of the viewport, and width and height are the size of the viewport rectangle. By default, the initial viewport values are (0, 0, winWidth, winHeight), where winWidth and winHeight are the size of the window.

Manipulating the matrix stacks

void glPushMatrix(void);

Pushes all matrices in the current stack down one level. The current stack is determined by glMatrixMode(). The topmost matrix is copied, so its contents are duplicated in both the top and second-from-the-top matrix. If too many matrices are pushed, an error is generated.

void glPopMatrix(void);

Pops the top matrix off the stack, destroying the contents of the popped matrix. What was the second-from-the-top matrix becomes the top matrix. The current stack is determined by glMatrixMode(). If the stack contains a single matrix, calling glPopMatrix() generates an error.

OpenGL Lighting function

A Hidden-Surface Removal Survival Kit

A depth buffer works by associating a depth, or distance, from the view plane (usually the near clipping plane), with each pixel on the window. Initially, the depth values for all pixels are set to the largest possible distance (usually the far clipping plane) using the glClear() command with GL_DEPTH_BUFFER_BIT. Then the objects in the scene are drawn in any order.

To use depth buffering, you need to enable depth buffering. This has to be done only once. Before drawing, each time you draw the scene, you need to clear the depth buffer and then draw the objects in the scene in any order.

To performs hidden-surface removal, in the main() function

```
glutInitDisplayMode (GLUT_DEPTH | .....);
….. glEnable(GL_DEPTH_TEST);
...
```

Real world and OpenGL Lighting

The OpenGL lighting model considers the lighting to be divided into four independent components: emissive, ambient, diffuse, and specular. ***ambient*** illumination is light that's been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions.

The ***diffuse*** component is the light that comes from one direction, so it's brighter if it comes squarely down on a surface than if it barely glances off the surface. Once it hits a surface, however, it's scattered equally in all directions, so it appears equally bright, no matter where the eye is located. ***Specular*** light comes from a particular direction, and it tends to bounce off the surface in a preferred direction. A well-collimated laser beam bouncing off a highquality mirror produces almost 100 percent specular reflection. Shiny metal or plastic has a high specular component, and chalk or carpet has almost none. You can think of specularity as shininess.

Creating light source

The command used to specify all properties of lights is **glLight*().** void glLight{if}(GLenum light, GLenum pname, TYPEparam); void glLight{if}v(GLenum light, GLenum pname, TYPE *param);

Creates the light specified by light, which can be GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7. The characteristic of the light being set is defined by pname, which specifies a named parameter param indicates the values to which the pname characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued light characteristics.

examples: -

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
. . .
. . .
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Enabling lighting

With OpenGL, you need to explicitly enable (or disable) lighting.

```
glEnable(GL_LIGHTING);
```

To disable lighting, call **glDisable()** with GL_LIGHTING as the argument.

Blending, Antialiasing and Fog

"Blending" tells you how to specify a blending function that combines color values from a source and a destination. The final effect is that parts of your scene appear translucent.

"Antialiasing" explains this relatively subtle technique that alters colors so that the edges of points, lines, and polygons appear smooth rather than angular and jagged.

"Fog" describes how to create the illusion of depth by computing the color values of an object based on its distance from the viewpoint. Thus, objects that are far away appear to fade into the background, just as they do in real life.

Bitmaps and Fonts

OpenGL provides only the lowest level of support for drawing strings of characters and manipulating fonts. The commands glRasterPos*() and glBitmap() position and draw a single bitmap on the screen.

Current raster position

void glRasterPos{234}{sifd}(TYPE x, TYPE y, TYPE z, TYPE w);

void glRasterPos{234}{sifd}v(TYPE *coords);

Sets the current raster position. The x, y, z, and w arguments specify the coordinates of the raster position. If the vector form of the function is used, the coords array contains the coordinates of the raster position. If glRasterPos2*() is used, z is implicitly set to zero and w is implicitly set to one; similarly, with glRasterPos3*(), w is set to one.

Examples: - glRasterPos2i(20, 20);

Drawing the bitmap

void glBitmap(GLsizei width, GLsizei height, GLfloat xbo, GLfloat ybo, GLfloat xbi, GLfloat ybi, const GLubyte *bitmap);

Draws the bitmap specified by bitmap, which is a pointer to the bitmap image. The origin of the bitmap is placed at the current raster position. If the current raster position is invalid, nothing is drawn, and the raster position remains invalid. The width and height arguments indicate the width and height, in pixels, of the bitmap. The width need not be a multiple of 8, although the data is stored in unsigned characters of 8 bits each.

Texture and mapping

Texture mapping allows you to glue an image of a brick wall (obtained, perhaps, by scanning in a photograph of a real wall) to a polygon and to draw the entire wall as a single polygon. Texture mapping ensures that all the right things happen as the polygon is transformed and rendered.

Steps in Texture Mapping

To use texture mapping, you perform these steps

1. Create a texture object and specify a texture for that object.
2. Indicate how the texture is to be applied to each pixel.
3. Enable texture mapping.
4. Draw the scene, supplying both texture and geometric coordinates.

Chapter 3

BASICS OF GLUT: The OpenGL UTILITY TOOLKIT

This section describes a subset of Mark Kilgard's OpenGL Utility Toolkit (GLUT), which is fully documented in his book, *OpenGL Programming for the X Window System* (Reading, MA: Addison-Wesley Developers Press, 1996). GLUT has become a popular library for OpenGL programmers, because it standardizes and simplifies window and event management. GLUT has been ported atop a variety of OpenGL implementations, including both the X Window System and Microsoft Windows NT.

This appendix has the following major sections:

- "Initializing and Creating a Window"
- "Handling Window and Input Events"
- "Loading the Color Map"
- "Initializing and Drawing Three-Dimensional Objects"
- "Managing a Background Process"
- "Running the Program"

1. Initializing and Creating a Window

Before a programmer can open a window, he/she must specify its characteristics: Should it be single-buffered or double-buffered? Should it store colors as RGBA values or as color indices? Where should it appear on your display? To specify the answers to these questions, call **glutInit**(), **glutInitDisplayMode**(), glutInitWindowSize(), and **glutInitWindowPosition**() before you call **glutCreateWindow**() to open the window.

*void **glutInit**(int argc, char **argv);*

glutInit() should be called before any other GLUT routine, because it initializes the GLUT library. ***glutInit()*** will also process command line options, but the specific options are window system dependent. For the X Window System, -iconic, -geometry, and display are examples of command line options, processed by ***glutInit()***. (The parameters to the ***glutInit()*** should be the same as those to ***main()***.)

*void **glutInitDisplayMode**(unsigned int mode);*

Specifies a display mode (such as RGBA or color-index, or single- or double-buffered) for windows created when ***glutCreateWindow()*** is called. You can also specify that the window have an associated depth, stencil, and/or accumulation buffer. The mask argument is a bitwise ORed combination of GLUT_RGBA or GLUT_INDEX, GLUT_SINGLE or GLUT_DOUBLE, and any of the buffer-enabling flags: GLUT_DEPTH, GLUT_STENCIL, or GLUT_ACCUM. For example, for a doublebuffered, RGBA-mode window with a depth and stencil buffer, use GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL. The default value is GLUT_RGBA | GLUT_SINGLE (an RGBA, single-buffered window).

*void **glutInitWindowSize**(int width, int height);*

*void **glutInitWindowPosition**(int x, int y);*

Requests windows created by ***glutCreateWindow()*** to have an initial size and position. The arguments (x, y) indicate the location of a corner of the window, relative to the entire display. The width and height indicate the window's size (in pixels). The initial window size and position are hints and may be overridden by other requests.

*int **glutCreateWindow**(char *name);*

Opens a window with previously set characteristics (display mode, width, height, and so on). The string name may appear in the title bar if your window system does that sort of thing. The window is not initially displayed until ***glutMainLoop()*** is entered, so do not render into the window until then. The value returned is a unique integer identifier for the window. This identifier can be used for controlling and rendering to multiple windows (each with an OpenGL rendering context) from the same application.

2. Handling Window and Input Events

After the window is created, but before programmer enter the main loop, he/she should register callback functions using the following routines.

void **glutDisplayFunc**(void (*func)(void));

Specifies the function that's called whenever the contents of the window need to be redrawn. The contents of the window may need to be redrawn when the window is initially opened, when the window is popped and window damage is exposed, and when **glutPostRedisplay()** is explicitly called.

void **glutReshapeFunc**(void (*func)(int width, int height));

Specifies the function that's called whenever the window is resized or moved. The argument func is a pointer to a function that expects two arguments, the new width and height of the window. Typically, func calls **glViewport()**, so that the display is clipped to the new size, and it redefines the projection matrix so that the aspect ratio of the projected image matches the viewport, avoiding aspect ratio distortion. If **glutReshapeFunc()** isn't called or is deregistered by passing NULL, a default reshape function is called, which calls **glViewport**(0, 0, width, height).

void **glutKeyboardFunc**(void (*func)(unsigned int key, int x, int y);

Specifies the function, func, that's called when a key that generates an ASCII character is pressed. The key callback parameter is the generated ASCII value. The x and y callback parameters indicate the location of the mouse (in window-relative coordinates) when the key was pressed.

void **glutMouseFunc**(void (*func)(int button, int state, int x, int y));

Specifies the function, func, that's called when a mouse button is pressed or released. The button callback parameter is one of GLUT_LEFT_BUTTON,

GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. The state callback parameter is either GLUT_UP or GLUT_DOWN, depending upon whether the mouse has been released or pressed. The x and y callback parameters indicate the location (in windowrelative coordinates) of the mouse when the event occurred.

void **glutMotionFunc**(void (*func)(int x, int y));

Specifies the function, func, that's called when the mouse pointer moves within the window while one or more mouse buttons is pressed. The x and y callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

void **glutPostRedisplay**(void);

Marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by **glutDisplayFunc()** will be called.

3. Loading the Color Map

If programmer is using color-index mode, he/she might be surprised to discover there's no OpenGL routine to load a color into a color lookup table. This is because the process of loading a color map depends entirely on the window system. GLUT provides a generalized routine to load a single color index with an RGB value, **glutSetColor()**.

void **glutSetColor**(GLint index, GLfloat red, GLfloat green, GLfloat blue);

Loads the index in the color map, index, with the given red, green, and blue values. These values are normalized to lie in the range [0.0,1.0].

4. Initializing and Drawing Three-Dimensional Objects

Many programs use three-dimensional models to illustrate various rendering properties. The following drawing routines are included in GLUT to avoid having to reproduce the code to draw these models in each program. The routines render all their graphics in immediate mode. Each three-dimensional model comes in two flavors: wireframe without surface normals, and solid with shading and surface normals. Use the solid version when you're applying lighting. Only the teapot generates texture coordinates.

void **glutWireSphere**(GLdouble radius, GLint slices, GLint stacks); void **glutSolidSphere**(GLdouble radius, GLint slices, GLint stacks);

void **glutWireCube**(GLdouble size); void **glutSolidCube**(GLdouble size);

void **glutWireTorus**(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

void **glutSolidTorus**(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings); void **glutWireIcosahedron**(void); void **glutSolidIcosahedron**(void);

void **glutWireOctahedron**(void); void **glutSolidOctahedron**(void);

void **glutWireTetrahedron**(void); void **glutSolidTetrahedron**(void); void **glutWireDodecahedron**(GLdouble radius);

void **glutSolidDodecahedron**(GLdouble radius); void **glutWireCone**(GLdouble radius, GLdouble height, GLint slices, GLint stacks); void **glutSolidCone**(GLdouble radius, GLdouble height, GLint slices, GLint stacks); void **glutWireTeapot**(GLdouble size); void **glutSolidTeapot**(GLdouble size);

5. Managing a Background Process

You 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(). This is particularly useful for continuous animation or other background processing.

void **glutIdleFunc**(void (*func)(void));

Specifies the function, func, to be executed if no other events are pending. If NULL (zero) is passed in, execution of func is disabled.

6. Running the Program

After all the setup is completed, GLUT programs enter an event processing loop, glutMainLoop().

void **glutMainLoop**(void);

Enters the GLUT processing loop, never to return. Registered callback functions will be called when the corresponding events instigate them.

Chapter 4

SYSTEM REQUIREMENTS

1. Software Requirements

- Operating System: Windows 10/ MACOS
- Language: C
- Tool: Codeblocks
- Library: OpenGL (glut 3.7.6)

2. Hardware Requirements

- Processor: Intel Processor 100 MHz / Pentium Processor 100 MHz /AMD
- RAM: 256 MB or more

Chapter 5

ANALYSIS & DESIGN

1. ANALYSIS

The objective of the game is to score as much points as possible by making the snake eat apple/food, while making sure that the snake does not die. The snake can die if the following to conditions occur:

1. The snake head collides with the boundary
2. The snake head collides with its body

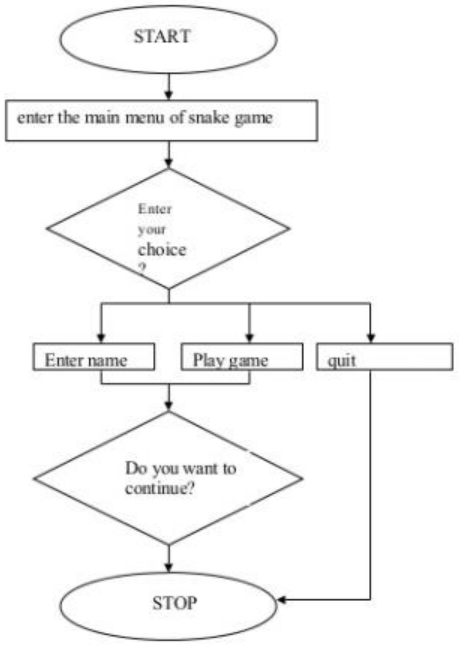
The player has the control to tune the speed of the snake using keyboard keys. He also has the option whether to play the game in grid mode or no-grid mode. When the game ends after the snake dies, the player's score is displayed in a dialog box

.

The basic feature of the 2D game was analyzed to be: -

1. On the start of the game the player is allowed to take control of the snake using arrow keys.
2. Calculating the score of the player as the snake eats food.
3. A game over window which will show score of the player when the snake dies.
4. Game speed controls and toggling grid using the keyboard.

2. DESIGN:



The snake game is designed for the user to play multiple times. It includes background music and added sound effects when the snake eats an apple and when the snake dies.

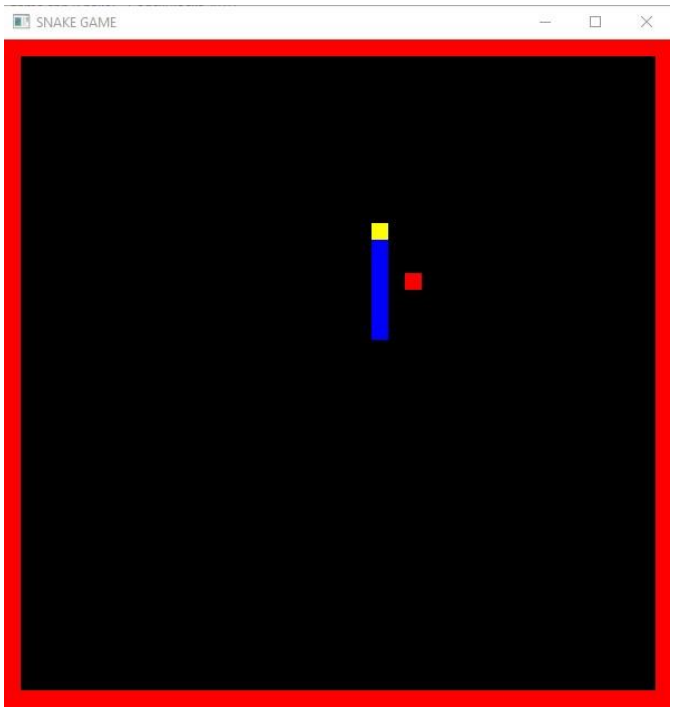
The snake is represented by an array of points which is updated after each frame. To check for collisions with the boundary it is checked if the value of the snake body box matches with that of another point in the snake body or a boundary point.

The direction of the taken by the keyboard is used to update the array that represents the snake in each frame. The score of the player is updated when the snake eats the apple. The final score will be displayed when the game ends.

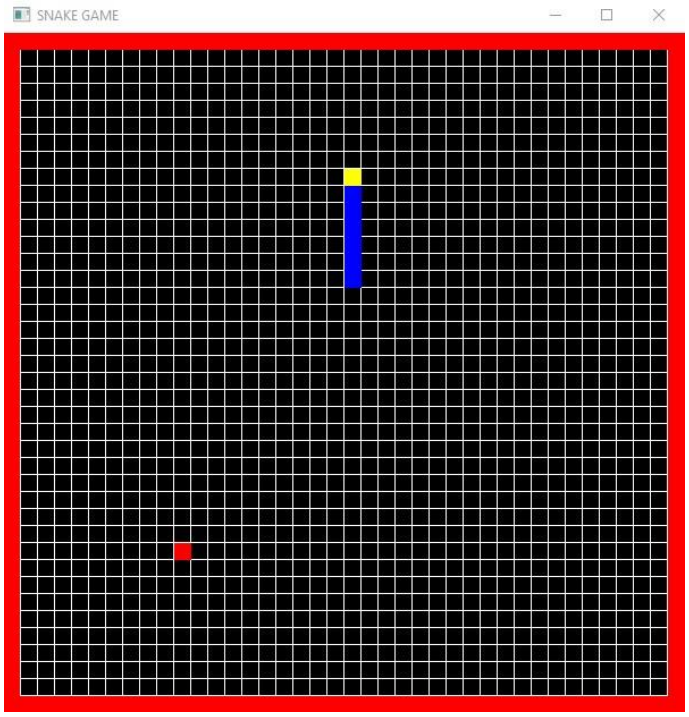
Chapter 6

SNAPSHOT

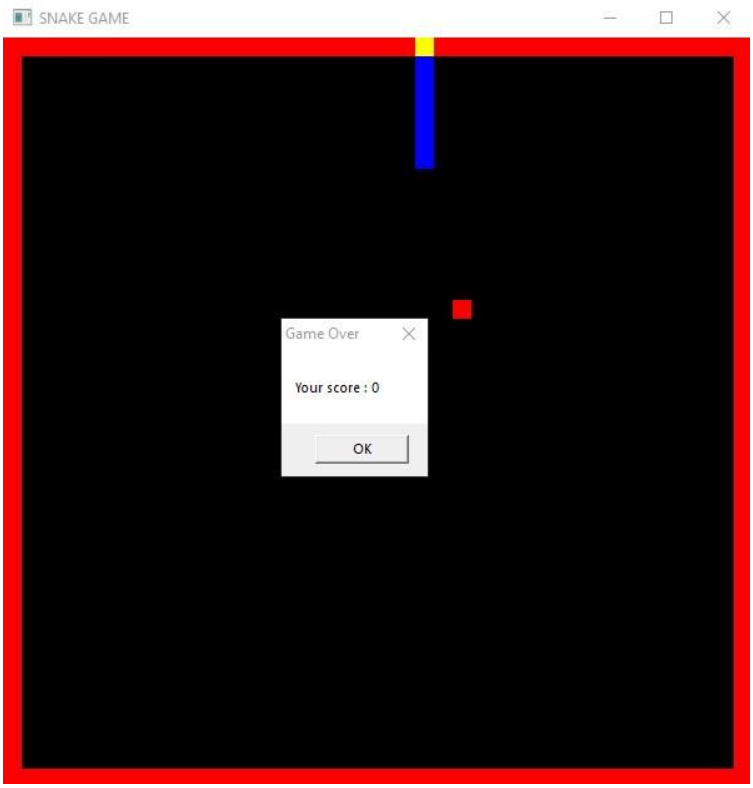
1. Game without grid mode



2. Game with grid mode



3. Game over


SNAKE GAME
Game Over
Your score : 0
OK

Chapter 7

CONCLUSION

I have attempted to design and implement “2D snake game”. OpenGl supports enormous flexibility in the design and the use of OpenGl graphics programs. The presence of many built in classes methods take care of much functionality and reduce the job of coding as well as makes the implementation simpler.

The project was started with the designing phase in which we figured the requirements needed, the layout design, then comes the detail designing of each function after which, was the testing and debugging stage. We have tried to implement the project making it as user-friendly and error free as possible.

BIBLIOGRAPHY:

1.Computer Graphics – Principals And Practice (Foley, Van Dam, Fenier and Hughes) helped me to understand graphics generation algorithms, user interface and dialogue design

2.OpenGL Programming Guide (Addison-Wesley Publishing Company) helped me to get through all OpenGL functions and Commands and understandings of all aspects of them.

3.www.cplusplus.com: - provided references regarding all C++ functions and their uses.

4.www.stackoverflow.com: - help to get rid of all types of error occurred regarding uses of OpenGL functions.

5.www.lighthouse3d.com: - OpenGL tutorial for implementing the OpenGL functions in Source code.

APPENDIX A: SOURCE CODE

```
#include <GL/glut.h>
#include <iostream>
#include <ctime>
#include "game.h"
#include <sstream>
#include <windows.h>
#include <mmsystem.h>
void unit(int,int);
int random(int,int);

bool length_inc = false;
bool seedflag = false;
bool grid = false;
extern int score;
extern bool game_over;
bool food=false;
int rows=0,columns=0;
int sDirection = RIGHT;
int foodx,foody;
int posx[MAX+1]={4,3,2,1,0,-1,-1};
int posy[MAX+1]={10,10,10,10,10,10,10};
int length=7;

void initGrid(int x,int y)
{
   columns=x;
   rows=y;
}

void draw_grid()
{

   for(int i =0;i<columns;i++)
   {
      for(int j=0;j<rows;j++)
          {unit(i,j);}
   }
}

void draw_snake()
{
```

```
    for(int i =length-1;i>0;i--)
    {
       posx[i]=posx[i-1];
       posy[i]=posy[i-1];
    }
    for(int i=0;i<length;i++)
    {
       glColor3f(0.0,0.0,1.0);
       if(i==0)
       {
          glColor3f(1.0,1.0,0.0);
          switch(sDirection)
          {
          case UP:
             posy[i]++;
             break;
          case DOWN:
             posy[i]--;
             break;
          case RIGHT:
             posx[i]++;
             break;
          case LEFT:
             posx[i]--;
             break;
          }
          if(posx[i]==0||posx[i]==columns-1||posy[i]==0||posy[i]==rows-1)
             game_over=true;
          else if(posx[i]==foodx && posy[i]==foody)
          {
             PlaySound("C:\\Users\\LENOVO\\Desktop\\CG
lab\\Snake\\EatSound.wav", NULL, SND_ASYNC);
             food=false;
             score++;
             if(length<=MAX)
                length_inc=true;
             if(length==MAX)
                MessageBox(NULL,"You Win","Congratulations",0);
          }
          for(int j=1;j<length;j++)
          {
             if(posx[j]==posx[0] && posy[j]==posy[0])
                game_over=true;
          }
```

```
        }
        glBegin(GL_QUADS);
            glVertex2d(posx[i],posy[i]); glVertex2d(posx[i]+1,posy[i]);
glVertex2d(posx[i]+1,posy[i]+1); glVertex2d(posx[i],posy[i]+1);
        glEnd();
    }
    if(length_inc)
    {
        length++;
        length_inc=false;
    }
}

void draw_food()

{
    if(!food)
    {
        foodx=random(2,columns-2);
        foody=random(2,rows-2);
        food=true;
    }
    glBegin(GL_POLYGON);
        glVertex2d(foodx,foody); glVertex2d(foodx+1,foody);
glVertex2d(foodx+1,foody+1); glVertex2d(foodx,foody+1);
    glEnd();
}

void unit(int x,int y)
{
    glLoadIdentity();
    if(x==0||x==columns-1||y==0||y==rows-1)
    {
        glColor3f(1.0,0,0);
        glLineWidth(.0);
        glBegin(GL_POLYGON);
            glVertex2d(x,y); glVertex2d(x+1,y);
            glVertex2d(x+1,y); glVertex2d(x+1,y+1);
            glVertex2d(x+1,y+1); glVertex2d(x,y+1);
            glVertex2d(x,y+1); glVertex2d(x,y);
        glEnd();
    }
    else if(grid)
    {
```

```
        glColor3f(1.0,1.0,1.0);
        glLineWidth(1.0);
        glBegin(GL_LINE_LOOP);
            glVertex2d(x,y); glVertex2d(x+1,y);
            glVertex2d(x+1,y); glVertex2d(x+1,y+1);
            glVertex2d(x+1,y+1); glVertex2d(x,y+1);
            glVertex2d(x,y+1); glVertex2d(x,y);
        glEnd();
    }

}
int random(int _min,int _max)
{
    if(!seedflag)
    {
        srand(time(NULL));
        seedflag=true;
    }
    else
        seedflag=false;
    return _min+rand()%(_max-_min);
}
```