JSS MAHAVIDYAPEETHA, MYSURU

JSS Academy of Technical Education

JSS Campus, Uttarahalli Kengeri Main Road, Bengaluru – 560060

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the mini project work entitled **"AQUARIUM - Shark Hunt**" is a benefited work carried out by **Deepak Naidu** and **Girish Kumar DV** bearing USN **1JS19CS049** and **1JS19CS057** bonafide student of **JSS Academy of Technical Education** in the partial fulfillment for the award of the **Bachelor of Engineering** in **Computer Science & Engineering**of the **Visvesvaraya Technological University**, Belgaum, during the year 2021-22. It is certified that all corrections / suggestions indicated for Internal Assessment have been incorporated in thereport deposited in the departmental library. The mini project report has been approved as it satisfies the academic requirements in respect of mini-Project work prescribed for the said degree.

Mr. Vikhyath K B
Asst.Professor, Dept. of CSE

Dr. Prasad M R
Asst.Professor, Dept. of CSE

Dr. Naveen N C
Prof & HOD, Dept. of CSE

Name of the Examiners	**Signature with date**
1……...	**……**
2……...	**……**

ABSTRACT

Graphics provides one of the most natural means of communication 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 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.

This project is on "AQUARIUM – Shark Hunt". Computer Graphics using Open GL Functions. It is a user interactive program where in the user can view the required display by making use of the input devices like Keyboard and Mouse. The project AQUARIUM-Shark Hunt is the project in which there are many fishes and has many plants. A shark is the main object and it is provided with hunting functionality. The objects are drawn by using GLUT functions. This project has been developed using Microsoft Visual Studio Code in Windows 10 environment with Open GL package(freeglut on Windows).

ACKNOWLEDGEMENT

We, take this opportunity to thank one and all involved in helping me build this project. Firstly, we would like to thank the college for providing me an opportunity to work on this project.

We thank the management of **JSS Academy of Technical Education Bengaluru** for providing all the resources required for the project.

We wish to acknowledge my sincere gratitude to our **Principal, Dr. Bhimasen Soragaon** for his constant encouragement and for providing us with all the facilities required for the accomplishment of this project.

The project would not have been possible if not for the constant support of our Professor and Head of Computer Science Department, **Dr. N C Naveen.**

We also, am highly grateful to the guidance offered by **Mr. Vikhyath K B and Dr. Prasad M R, Asst. Professors Department of Computer Science and Engineering,** who have been very generous in assisting and supporting, to do this project named **"AQUARIUM - Shark Hunt"**, which formally started as just a rough idea and now has resulted in the form of this project.

We also would like to thank all the other teaching and non-teaching staff members who had extended their hand for support and co-operation while bringing up this project.

Deepak Naidu – 1JS19CS049

Girish Kumar DV -1JS19CS057

CONTENTS

Sl.No	Chapter Name	Page No.
	ACKNOWLEDGMENT	**I**
	ABSTRACT	**II**
	CONTENTS	**III**
	LIST OF FIGURES	**IV**
	LIST OF TABLES	**V**
1.	**INTRODUCTION**	**1**
1.1	About OpenGL	2
1.2.1	OpenGL commands and primitives	7
1.2.2	OpenGL rendering pipeline	9
1.2.3	OpenGL -GLUT and OpenGL Utility Libraries	12
2.	**REQUIRMENTS ANALYSIS**	**16**
2.1	Requirements of the project	16
2.2	Resource requirements	16
2.2.1	Software requirements	16
2.2.2	Hardware requirements	17
3.	**DESIGN PHASE**	**19**
3.1	Algorithm	19
3.2	Flow Diagram	21
4.	**IMPLEMENTATION**	**23**
4.1	Implementation of OpenGL built in functions	23
4.2	Implementation of user defined functions	27
4.3	Source Code	28
5.	**TESTING AND SNAPSHOTS**	**40**
5.1	Test Cases	40
5.2	Snapshots	41
	FUTURE ENHANCEMENT	**VI**
	CONCLUSION	**VII**
	REFERENCES	**VIII**

LIST OF FIGURES

Fig No.	Figure Name	Page No.
Fig 1.1.	Order of Operations	08
Fig 5.1.	Snapshot - Initial Screen	42
Fig 5.2.	Snapshot - Inside the project	42
Fig 5.3.	Snapshot - Different menu options	43
Fig 5.4.	Snapshot – Shark hunting mode	43
Fig 5.5.	Snapshot – Shark hunting options	44

LIST OF TABLES

Table No.	Table Name	Page No.
Table 1.1	Datatypes accepted by OpenGL	08
Table 2.1	Hardware requirements for visual studio C++ 2010 Express	18
Table 5.1	Test Cases	41

INTRODUCTION

Computer graphics is a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Although the term often refers to the study of three-dimensional computer graphics, it also encompasses two-dimensional graphics and image processing. It is divided into two broad classes. It is also called passive graphics. Here the user has no control over the pictures produced on the screen. Interactive graphics provides extensive user-computer interaction. It provides a tool called "motion dynamics" using which the user can move objects. A broad classification of major subfields in computer graphics might be:

1. Geometry: study of different ways to represent and process surfaces.
2. Animation: study of different ways to represent and manipulate motion.
3. Rendering: study of algorithms to reproduce light transport
4. Imaging: study of image acquisition or image editing.
5. Topology: study of the behavior of spaces and surfaces.

Geometry

The subfield of geometry studies the representation of three-dimensional objects in a discrete digital setting. Because the appearance of an object depends largely on its exterior, boundary representations are most commonly used. Two dimensional surfaces are a good representation for most objects, though they may be non-manifold. Since surfaces are not finite, discrete digital approximations are used.

Animation

The subfield of animation studies descriptions for surfaces (and other phenomena) that move or deform over time. Historically, most work in this field has focused on parametric and data-driven models, but recently physical simulation has become more popular as computers have become more powerful computationally.

Rendering

Rendering generates images from a model. Rendering may simulate light transport to create realistic images or it may create images that have a particular artistic style in non-photorealistic rendering. The two basic operations in realistic rendering are transport (how much light passes from one place to another) and scattering (how surfaces interact with light).

Imaging

Digital imaging or digital image acquisition is the creation of digital images, such as of a physical scene or of the interior structure of an object.

Topology

Topology is the mathematical study of shapes and topological spaces.

1.1 About OpenGL

OpenGL(**O**pen **G**raphics **L**ibrary**)** is an application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. OpenGL stands for Open Graphics Library. It is a specification of an API for rendering graphics, usually in 3D. OpenGL implementations are libraries that implement the API defined by the specification. OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that is used to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL (**Open G**raphics **L**ibrary) is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics the interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, and flight simulation. It is also used in video games, where itcompetes with Direct3D on Microsoft Windows platforms. OpenGL is managed by the non-profit technology consortium, the Khronos Group**.**

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, user must work through whatever windowing system controls the particular hardware that the user is using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow the user to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation.

OpenGL serves two main purposes:

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

OpenGL's basic operation is to accept primitives such as points, lines and polygons, and convert them into pixels. This is done by a graphics pipeline known as the OpenGL state machine. Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives. Prior to the introduction of OpenGL 2.0, each stage of the pipeline performed a fixed function and was configurable only within tight limits. OpenGL 2.0 offers several stages that are fully programmable using GLSL.

OpenGL is a low-level, procedural API, requiring the programmer to dictate the exact steps required to render a scene. These contrasts with descriptive APIs, where a programmer only needs to describe a scene and can let the library manage the details of rendering it. OpenGL's low-level design requires programmers to have a good knowledge of the graphics pipeline, but also gives a certain amount of freedom to implement novel rendering algorithms.

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

1. Evaluation, if necessary, of the polynomial functions which define certain inputs, like NURBS surfaces, approximating curves and the surface geometry.

2. Vertex operations, transforming and lighting them depending on their material. Also clipping non visible parts of the scene in order to produce the viewing volume.

3. Rasterization or conversion of the previous information into pixels. The polygons are represented by the appropriate color by means of interpolation algorithms.

4. Per-fragment operations, like updating values depending on incoming and previously stored depth values, or color combinations, among others.

5. Lastly, fragments are inserted into the Frame buffer.

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

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

Although the OpenGL specification defines a particular graphics processing pipeline, platform vendors have the freedom to tailor a particular OpenGL implementation to meet unique system cost and performance objectives. Individual calls can be executed on dedicated hardware, run as software routines on the standard system

CPU, or implemented as a combination of both dedicated hardware and software routines. This implementation flexibility means that OpenGL hardware acceleration can range from simple rendering to full geometry and is widely available on everything from low-cost PCs to high-end workstations and supercomputers. Application developers are assured consistent display results regardless of the platform implementation of the OpenGL environment.

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

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

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, the user must work through whatever windowing system controls the particular hardware that the user is using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow the user to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules.

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

- Construct shapes from geometric primitives, thereby creating mathematical descriptions of objects. (OpenGL considers points, lines, polygons, images, and bitmaps to be primitives.)

- Arrange the objects in three-dimensional space and select the desired vantage point for viewing the composed scene.
- Calculate the color of all the objects. The color might be explicitly assigned by the application, determined from specified lighting conditions, obtained by pasting a texture onto the objects, or some combination of these three actions.
- Convert the mathematical description of objects and their associated color information to pixels on the screen. This process is called rasterization.

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

In this situation, the computer on which the program runs and issues OpenGL drawing commands is called the client, and the computer that receives those commands and performs the drawing is called the server.

The format for transmitting OpenGL commands (called the *protocol*) from the client to the server is always the same, so OpenGL programs can work across a network even if the client and server are different kinds of computers. If an OpenGL program isn't running across a network, then there's only one computer, and it is both the client and the server.

1.2.1 OpenGL Commands and Primitives

OpenGL draws *primitives*—points, line segments, or polygons—subject to several selectable modes. You can control modes independently of each other; that is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the frame buffer). Primitives are specified, modes are set, and other OpenGL operations are described by issuing commands in the form of function calls. Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an

endpoint of a line, or a corner of a polygon where two edges meet. Data (consisting of vertex coordinates, colors, normal, texture coordinates, and edge flags) is associated with a vertex, and each vertex and its associated data are processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be *clipped* so that a particular primitive fits within a specified region; in this case, vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state-querying commands return data that's consistent with complete execution of all previously issued OpenGL commands.

OpenGL commands use the prefix **gl** and initial capital letters for each word making up the command name (**glClearColor ()**, for example). Similarly, OpenGL defined constants begin with GL_, use all capital letters, and use underscores to separate words (like GL_COLOR_BUFFER_BIT). Some seemingly extraneous letters appended to some command names (for example, the **3f** in **glColor3f () and glVertex3f ()**) can also be seen. It's true that the **Color** part of the command name **glColor3f ()** is enough to define the command as one that sets the current color. However, more than one such command has been defined so that the user can use different types of arguments. In particular, the **3** parts of the suffix indicates that three arguments are given; another version of the **Color** command takes four arguments. The **f** part of the suffix indicates that the arguments are floating-point numbers. Having different formats allows OpenGL to accept the user's data in his or her own data format.

Some OpenGL commands accept as many as 8 different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are shown in Table 1.1, along with the corresponding OpenGL type definitions. The particular implementation of OpenGL that are used might not follow this scheme exactly; an implementation in C++ or ADA, for example, wouldn't need to implementation in C++ or ADA, for example, wouldn't need to.

Suffix	**Data Type**	**Typical Corresponding C-Language Type**	**OpenGL Type Definition**
B	8-bit integer	Signed char	GLbyte

S	16-bit integer	Short	GLshort
I	32-bit integer	int or long	GLint, GLsizei
F	32-bit floating-point	Float	GLint, GLclampf
D	62-bit floating-point	Double	GLdouble, GLclampd
Ub	8-bit unsigned integer	Unsigned char	GLubyte, GLboolean
Us	16-bit unsigned integer	Unsigned short	GLushort
Ui	32-bit unsigned integer	Unsigned int or long	GLuint, GLenum,GLbitfield

Table 1.1 - Datatypes accepted by OpenGL

1.2.2 OpenGL Rendering Pipeline

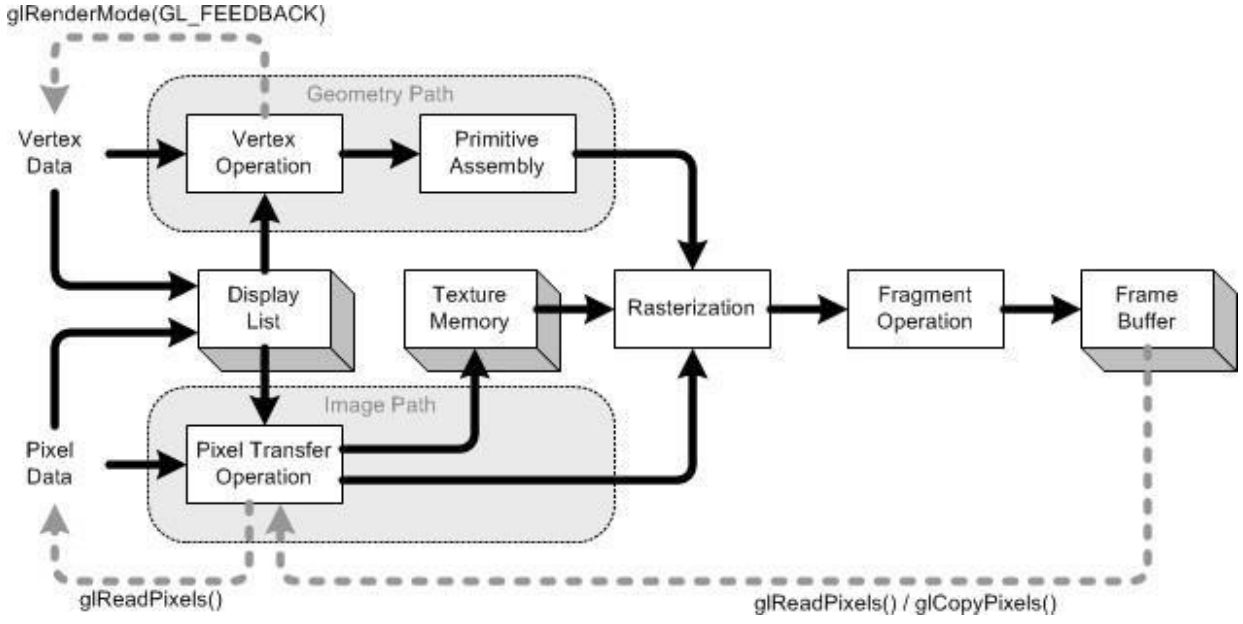

Figure 1.1: Order of Operations

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

Display Lists

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

Evaluators

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

Per-Vertex Operations

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

calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

Primitive Assembly

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

Pixel Operations

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

Texture Assembly

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

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

Rasterization

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

Fragment Operations

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

1.2.3 OpenGL -GLUT and OpenGL Utility Libraries

There are numerous Windowing system and interface libraries available for OpenGL as well as Scene graphs and High-level libraries build on top of OpenGL

About GLUT

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

windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL Programming.

Other GLUT-like Window System Toolkits

Libraries that are modeled on the functionality of GLUT providing support for things like: windowing and events, user input, menus, full screen rendering, performance timing

About GLX, GLU & DRI

GLX is used on Unix OpenGL implementation to manage interaction with the X Window System and to encode OpenGL onto the X protocol stream for remote rendering. GLU is the OpenGL Utility Library. This is a set of functions to create texture mipmaps from a base image, map coordinates between screen and object space, and draw quadric surfaces and NURBS. DRI is the Direct Rendering Infrastructure for coordinating the Linux kernel, X window system, 3D graphics hardware and an OpenGL-based rendering engine.

GLX, GLU and DRI

GLX Library

GLX 1.3 is used on Unix OpenGL implementation to manage interaction with the X Window System and to encode OpenGL onto the X protocol stream for remote rendering. It supports: pixel buffers for hardware accelerated offscreen rendering; read-only drawables for preprocessing of data in an offscreen window and direct video input; and FBConfigs, a more powerful and flexible interface for selecting frame buffer configurations underlying an OpenGL rendering window.

GLU Library

The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. GLU routines use the prefix **glu**. This is a set of functions to create texture mipmaps from a base image, map coordinates between screen and object space, and draw quadric surfaces and NURBS.

GLU 1.2 is the version of GLU that goes with OpenGL 1.1. GLU 1.3 is available and includes new capabilities corresponding to new OpenGL 1.2 features.

Direct Rendering Infrastructure (DRI)

In simple terms, the DRI enables hardware-accelerated 3D graphics on Linux. More specifically, it is a software architecture for coordinating the Linux kernel, X window system, 3D graphics hardware and an OpenGL-based rendering engine.

Higher Level Libraries built on OpenGL

Leading software developers use OpenGL, with its robust rendering libraries, as the 2D/3D graphics foundation for higher-level APIs. Developers leverage the capabilities of OpenGL to deliver highly differentiated, yet widely supported vertical market solutions. Open Inventor, IRIS Performer, OpenGL Optimizer, OpenGL Volumizer, OpenGL Shader, Scene Graph APIs.

- Open Inventor by VSG

  Open Inventor by VSG is the commercial, current evolution of Open Inventor and provides an up-to-date, highly-optimized, full-featured implementation of the popular object-oriented scene graph API for C++, .NET and Java. Applications powered by Open Inventor by VSG also benefit from powerful extensions such as VolumeViz LDM for very large volume data, MeshViz XLM for high-performance mesh support, or ScaleViz for multi-GPUs and immersive VR.
- Coin

  Coin is a 3D graphics library. Its C++ API is based on the Open Inventor 2.1 API and OpenGL. For Win32, Linux, IRIX and Mac OS X.
- Gizmo 3D

  Gizmo3D is a high-performance OpenGL-based 3D Scene Graph and effect visualization C++ toolkit for Linux, WIN32 and IRIX used in Game or VisSim development. It is similar to other scene graph engines such as Cosmo3D/Performer/Fahrenheit/Inventor/VisKit/VTree but is a multi-platform compatible API with very high performance. It also adds some state-of-the-art functionality for effect presentations and graph interaction.
- OpenSceneGraph

OSG is a open source high performance 3D graphics toolkit, used by application developers in fields such as visual simulation, games, virtual reality, scientific visualization and modelling. Written entirely in Standard C++ and OpenGL it runs on all Windows platforms, OSX, Linux, IRIX, Solaris and FreeBSD operating systems.

- OpenRM Scene Graph

  OpenRM an open-source development environment used to build portable 2D/3D/stereohigh performance graphics, virtual reality, visualization applications and games for Unix/X11 and Win32 platforms. It is a scene graph API that uses OpenGL as a graphics platform and graphics hardware acceleration. A scene graph model is a useful way to organize data for rendering in a way that is particularly efficient for graphics display engines (in most cases).

- Quesa 3D

  Quesa is a high-level 3D graphics library, released as Open Source under the LGPL, which implements Apple's QuickDraw 3D API on top of OpenGL. It supports both retained and immediate mode rendering, an extensible file format, plug-in renderers, a wide range of high-level geometries, hierarchical models, and a consistent and object-orientated API. Quesa currently supports Mac OS, Linux, and Windows - ports to Be and Mac OS X are in progress.

CHAPTER 2

REQUIREMENTS ANALYSIS

The requirement analysis specifies the requirements of the computer graphic system is needed to develop a graphic project.

2.1 Requirements of the project

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

2.2 Resource Requirements

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

- Hardware Requirements
- Software Requirements

2.2.1 Software Requirements

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

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

Interface with other software components or products, including other systems, utility software, databases, and operating systems.

OpenGL libraries for required are

- GLUT library
- STDLIB library

The application is very self-contained. Robust error handling will be implemented and code will be object-oriented to allow for easier maintenance and feature additions.

This model runs using Microsoft Visual Studio C++ Express version 10 and runs on Windows XP/07/08.

2.2.2 Hardware Requirements

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

- Processor: Intel® Pentium 2.6+ GHz
- Hard disk Capacity: 50 GB on a single drive/partition
- RAM: 4 GB
- CPU Speed: 2.6+ GHz
- Keyboard: Standard
- Mouse: Standard

Visual Studio Hardware Requirements

- Some specific hardware requirements are needed for Visual Studio C++ Express 2010 which include the Processor, RAM, Hard disk space, required operating system, Video and Mouse. They can be listen as follows:

Requirement	**Professional**
Processor	1.6 GHz processor or higher
RAM	1 GB of RAM (1.5 GB if running on a virtual machine)
Available Hard Disk Space	20 GB of available hard disk space
Operating System	Visual Studio 2010 can be installed on the following operating systems:

	• Windows XP (x86) with Service Pack 3 - all editions except Starter Edition • Windows Vista (x86 & x64) with Service Pack 2 - all editions except Starter Edition • Windows 7 (x86 and x64) • Windows Server 2003 (x86 & x64) with Service Pack 2 - all editions. Users must install MSXML6 if it is not already present. • Windows Server 2003 R2 (x86 and x64) - all editions • Windows Server 2008 (x86 and x64) with Service Pack 2 - all editions • Windows Server 2008 R2 (x64) - all editions
Video	DirectX 9-capable video card that runs at 1024 x 768 or higher display resolution
Mouse	Microsoft mouse or compatible pointing device

Table 2.1: Hardware Requirements for Visual Studio C++ 2010 Express

- Performance has not been tuned for minimum system configuration. Increasing the RAM above the recommended system configuration will improve performance, specifically when there are running multiple applications, working with large projects, or doing enterprise-level development.
- When the Visual Studio installer is started, the default installation location is the system drive, which is the drive that boots the system. However, the user can install the application on any drive. Regardless of the application's location, the installation process installs some files on the system drive. Consequently, make sure that the required amount of space, as listed in this table, is available on the system drive regardless of the application's location. Make sure that the required additional space, as listed in this table, is available on the drive on which it is in the application is installed.

CHAPTER 3
DESIGN PHASE

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

3.1 Algorithm:

The entire design process can be explained using a simple algorithm. The algorithm gives a detailed description of the design process of 'Shark hunt'.
The various steps involved in the design of 'Shark Hunt' are as shown below:Step 1: Start

Step 2: Set initial Display Mode.

Step 3: Set initial Window Size.

Step 4: Create Window "Shark Hunt".

Step 5: Display function

Display the contents of the Initial Window

Step 6: displayFish() function

Set clear to the Color and Depth Buffers
Display Initial Window by calling display function.
Display aquarium
Display fishes.
Display Frog, pebbles and leaves
and bubbles. Call Swap Buffer
function.

Step 7: Keyboard function

Assign ‘P’ to Enter.

Check function.

Step 8 : Menu Creation

Create all menus and sub menus and pass the respective functions.

Step 9: Init function

Set Clear Color.

Set Ortho function.

Step 10: Stop

3.2 Flow Diagram:

Flow diagram is a collective term for a diagram representing a flow or set of dynamic relationships in a system. The term flow diagram is also used as synonym of the flowchart and sometimes as counterpart of the flowchart.

Flow diagrams are used to structure and order a complex system, or to reveal the underlying structure of the elements and their interaction. Flow diagrams are used in analyzing, designing, documenting or managing a process or program in various fields

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

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

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

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

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

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

CHAPTER 4

IMPLEMENTATION

The implementation stage of this model involves the following phases.

- Implementation of OpenGL built in functions.
- User defined function Implementation.

4.1 Implementation of OpenGL Built in Functions

1. glutInit ( )

glutInit is used to initialize the GLUT library.

glutInit will initialize the GLUT library and negotiate a session with the window system.

2. glutInitDisplayMode ( )

glutInitDisplayMode sets the initial display mode.

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

3. glutCreateWindow ( )

glutCreateWindow creates a top-level window.

glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name.

4. glutDisplayFunc ( )

glutDisplayFunc sets the display callback for the current window.

glutDisplayFunc sets the display callback for the current window. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called. The display callback is called with no parameters. The entire normal plane region should be redisplayed in response to the callback.

5. glutKeyboardFunc ( )

glutKeyboardFunc sets the keyboard callback for the current window. glutKeyboardFunc sets the keyboard callback for the *current window*. When a user types into the window, each key press generating an ASCII character will generate a keyboard callback. The key

callback parameter is the generated ASCII character. The x and y callback parameters indicate the mouse location in window relative coordinates when the key was pressed.

6. glutMainLoop ( )

glutMainLoop enters the GLUT event processing loop.

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

7. glMatrixMode ( )

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

8. gluOrtho2D ( )

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

9. glTranslate ( )

glTranslate — multiply the current matrix by a translation matrix.

glTranslate produces a translation by x y z. The current matrix is multiplied by this translation matrix, with the product replacing the current matrix, as if glMultMatrix were called with the following matrix for its argument:1 0 0 x 0 1 0 y 0 0 1 z 0 0 0 1

10.glutWireSphere ( )

glutWireSphere render a wireframe sphere. Renders a sphere centered at the modeling coordinates origin of the specified radius. The sphere is subdivided around the Z axis into slices and along the Z axis into stacks.

11. glutIdleFunc ()

glutIdleFunc sets the global idle callback. glutIdleFunc sets the global idle callback to be func so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received. If enabled, the idle callback is continuously called when events are not being received.

12. glutReshapeFunc ()

glutReshapeFunc sets the reshape callback for the *current window*. glutReshapeFunc sets the reshape callback for the *current window*. The reshape callback is triggered when a

window is reshaped. A reshape callback is also triggered immediately before a window's first display callback after a window is created or whenever an overlay for the window is established.

13. glutBitmapCharacter ()

glutBitmapCharacter renders a bitmap character using OpenGL. Without using any display lists, glutBitmapCharacter renders the character in the named bitmap font.

14. gluLookAt ()

gluLookAt — define a viewing transformation. gluLookAt creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.

15. glRotatef( )

glRotate — multiply the current matrix by a rotation matrix.glRotate produces a rotation of angle degrees around the vector x y z . The current matrix is multiplied by a rotation matrix with the product replacing the current matrix

16. glOrtho( )

glOrtho — multiply the current matrix with an orthographic matrix.glOrtho describes a transformation that produces a parallel projection. The current matrix is multiplied by this matrix and the result replaces the current matrix

17. glutSwapBuffers ()

glutSwapBuffers swaps the buffers of the *current window* if double buffered. Performs a buffer swap on the *layer in use* for the *current window*. Specifically, glutSwapBuffers promotes the contents of the back buffer of the *layer in use* of the *current window* to become the contents of the front buffer.

18. glutPostRedisplay( )

glutPostRedisplay marks the *current window* as needing to be redisplayed. Mark the normal plane of *current window* as needing to be redisplayed. The next iteration through

glutMainLoop, the window's display callback will be called to redisplay the window's normal plane.

19. gluPerspective( )

gluPerspective — set up a perspective projection matrix.gluPerspective specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in gluPerspective should match the aspect ratio of the associated viewport.

20. glViewPort( )

glViewport — set the viewport.glViewport specifies the affine transformation of x and y from normalized device coordinates to window coordinates.

21. glMaterialfv()

The glMaterialfv function specifies material parameters for the lighting model.

22. glColorMaterial()

glColorMaterial — cause a material color to track the current color.
glColorMaterial specifies which material parameters track the current color. When GL_COLOR_MATERIAL is enabled, the material parameter or parameters specified by *mode*, of the material or materials specified by *face*, track the current color at all times.

23. glLightfv()

glLight — set light source parameters. glLight sets the values of individual light source parameters. *light* names the light and is a symbolic name of the form GL_LIGHT i, where i ranges from 0 to the value of GL_MAX_LIGHTS - 1. *pname* specifies one of ten light source parameters, again by symbolic name. *params* is either a single value or a pointer to an array that contains the new values.

4.2 Implementation of User Defined Functions:

1. Shark()

This function is written to design the shark shape, size and color. We have used GL_TRIANGLE_FAN and GL_POLYGON macro calls to define the shape of the shark and the angular curvature of the body is given by the sine of angle in radians.

2. fish1() & fish2()

This function is written to design the fish shape, size and color. We have used GL_TRIANGLES and GL_POLYGON macro calls to define the shape of the fishes. Random colors are looped to display the fish

3. drawstring ()

The drawstring function is called inorder to display text in the first two screens. It includes some built-in functions like glRasterPos2i(x, y) and glutBitmapCharacter(font, string[i]) that creates the text required using an array initialised as glutBitmapCharacter() can print only one string at a time.Rather than using the function many times, an array is used.

4. screen()

This function is used to display all the contents of the first window using the display_string() function.

5. displayShark()

This function is used to display the shark along with the fishes and frog in the tank.

6. huntShark()

This function is used to display the shark in hunting mode.

7. Menu()

This function is used to create a menu for selecting the various options that are offered to be selected to the user. This is the main menu which contains other sub menu called 'Hunt options'.

This creates a menu when the user clicks the right mouse button on the display window which will show all the options available and user can select any option as per his choice and the respective task will be done.

8. subMenu ()

This function is a sub menu which basically gives the user various hunting options that can be implemented by the shark.

4.3 Source Code

```
#include<stdio.h>
#include<stdlib.h>
#include<GL/freeglut.h>
#include<math.h>
float  s=400,ss=140, a=0,b=0,aa=-70,bb=0,flag=0,flag3=0, flag4=0;
float x=100,y=0,r=0.5,y11=0,y21=0,y31=0,y41=0,y51=0,y61=0,y71=0,y81=0,y91=0,y10=0,y12=0,y13=0,y14=0,x0=0,xo=0,
angle=0.0, pi=3.142;
int moving = 1;
int temp =0;
void create_menu(void);
void menu(int);
void display();
void drawFrog();
void displayShark();
void displayFish();
void mov(void);

void drawstring(float x,float y,float z,char *string, int f)
{
   char *c;
   glRasterPos3f(x,y,z);
   for(c=string;*c!='\0';c++){
      if(f==0)
         glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18,*c);
      else
         glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24,*c);
   }
}

void init()
{
glClearColor(0.0,0.6,0.9,0.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,500,0,500);
}

void screen()
{
   glClear(GL_COLOR_BUFFER_BIT);
```

```
            glColor3f(1.0,0.0,0.0);
drawstring(255,425,0.0,"JSS ACADEMY OF TECHNICAL EDUCATION, BENGALURU",1);
            glColor3f(0.0,0.0,0.0);
drawstring(245,385,0.0,"DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING",1);
            glColor3f(0.6,1.0,0.0);
drawstring(210,350,0.0,"A MINI PROJECT ON COMPUTER GRAPHICS AND VISUALIZATION",1);
            glColor3f(1.0,1.0,0.0);
drawstring(440,300,0.0,"AQUARIUM",1);
            glColor3f(1.0,0.0,0.0);
  drawstring(135,205,0.0," STUDENT NAME & USN",0);
            glColor3f(0.0,0.0,0.0);
drawstring(140,170,0.0,"DEEPAK NAIDU - 1JS19CS049",0);
drawstring(140,135,0.0,"GIRISH KUMAR DV - 1JS19CS057",0);
            glColor3f(1.0,0.0,0.0);
drawstring(710,205,0.0,"Under the Guidance of",0);
            glColor3f(0.0,0.0,0.0);
drawstring(720,170,0.0,"Mr. VIKHYATH KB",0);
   glColor3f(0.0,0.0,0.0);
drawstring(720,135,0.0,"Asst. professor CSE",0);
            glColor3f(1.0,1.0,0.0);
drawstring(406,95,0.0,"Academic Year 2021-22",0);
drawstring(400,65,0.0,"Press 'P' to enter aquarium",0);

glFlush();
 glutSwapBuffers();
}

void displayHuntMsg1(){
   glClear(GL_COLOR_BUFFER_BIT);
            glColor3f(1.0,0.0,0.0);
drawstring(255,425,0.0,"Hunted 1 Fish!",1);
   glFlush();
   glutSwapBuffers();
}

void displayHuntMsg2(){
   glClear(GL_COLOR_BUFFER_BIT);
            glColor3f(1.0,0.0,0.0);
drawstring(255,425,0.0,"Hunted 2 Fishes!",1);
   glFlush();
   glutSwapBuffers();
}

void reshape(GLint w, GLint h)
{
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if(h>w)
            gluOrtho2D(0, 500, ((float)h/(float)w)*(0), ((float)h/(float)w)*500);
else
       gluOrtho2D(((float)w/(float)h)*(0), ((float)w/(float)h)*(520), 0, 500);
glMatrixMode(GL_MODELVIEW);
glutPostRedisplay();
}

void fish1()
{


glColor3f(1.0,0.0,0.0);
  glBegin(GL_POLYGON);
   glVertex2f(270+a,350+aa);

   glVertex2f(300+a,325+aa);
```

```
   glVertex2f(370+a,350+aa);
   glVertex2f(300+a,375+aa);
  glEnd();

glBegin(GL_POLYGON);

    glVertex2f(360+a,350+aa);
    glColor3ub( rand()%1, rand()%1000, rand()%1000 );
   glVertex2f(405+a,370+aa);
   glVertex2f(395+a,350+aa);
   glVertex2f(405+a,327+aa);
  glEnd();

  glBegin(GL_TRIANGLES);

  glColor3ub( rand()%1, rand()%1000, rand()%1000 );
    glVertex2f(302+a,373+aa);
   glColor3f(1.0,0.0,0.0);
 glVertex2f(340+a,409+aa);
 glVertex2f(320+a,360+aa);
  glEnd();

glBegin(GL_TRIANGLES);

glColor3ub( rand()%1, rand()%1000, 0 );
    glVertex2f(302+a,328+aa);
glColor3f(1.0,0.0,0.0);
 glVertex2f(340+a,300+aa);
 glVertex2f(320+a,340+aa);
  glEnd();


 glColor3f(0.0,0.0,0.0);
  glPointSize(4.0);
  glBegin(GL_POINTS);
  glVertex2f(280+a,355+aa);
  glEnd();
}

void fish2()
{
 glColor3f(1.0,0.0,0.0);
  glBegin(GL_POLYGON);
     glVertex2f(70+b,145+bb);
  glColor3ub(rand()%500, rand()%500,0);
   glVertex2f(20+b,120+bb);
    glVertex2f(30+b,145+bb);
   glVertex2f(20+b,170+bb);


  glEnd();
  glColor3f(1.0,0.0,0.0);
    glBegin(GL_POLYGON);
   glVertex2f(65+b,145+bb);

   glVertex2f(125+b,170+bb);

   glVertex2f(165+b,145+bb);
   glVertex2f(125+b,120+bb);
  glEnd();

  glBegin(GL_TRIANGLES);
  glColor3ub(rand()%500, rand()%500,0);
    glVertex2f(126+b,168+bb);
```

```
   glColor3f(1.0,0.0,0.0);
   glVertex2f(110+b,155+bb);
    glVertex2f(85+b,195+bb);
 glEnd();
 glBegin(GL_TRIANGLES);
  glColor3ub(rand()%500, rand()%500,0);
    glVertex2f(126+b,122+bb);
   glColor3f(1.0,0.0,0.0);
glVertex2f(110+b,136+bb);
glVertex2f(85+b,95+bb);
  glEnd();




 glColor3f(0.0,0.0,0.0);
  glPointSize(4.0);
  glBegin(GL_POINTS);
  glVertex2f(150+b,149+bb);
  glEnd();

}

void pbSmall(){

    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<360.0; angle+=.1){
        glColor3ub(46,47,47);
        y =(sin(angle*pi/180)*30);
        x =(cos(angle*pi/180)*50);
        glVertex2f(x+150,y-5);
    }
    glEnd();
}

void pb(){

    pbSmall();
    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<360.0; angle+=.1){
        glColor3ub( 142,20,30);
        y =(sin(angle*pi/180)*45);
        x =(cos(angle*pi/180)*75);
        glVertex2f(x+250,y-10);
    }
    glEnd();
}

 void pb1(){

    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<360.0; angle+=.1){
        glColor3ub( 20,20,20);
        y =(sin(angle*pi/180)*35);
        x =(cos(angle*pi/180)*65);
        glVertex2f(x+370,y-5);
    }
    glEnd();
}

 void pb2(){

    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<360.0; angle+=.1){
        glColor3ub(36, 60, 2);
```

```
        y =(sin(angle*pi/180)*40);
        x =(cos(angle*pi/180)*75);
        glVertex2f(x+610,y+10);
    }
    glEnd();
}

void circleFunc(float yc,int d,int x1,int y1){
    for(angle=0; angle<360.0; angle+=.1)
{
        y =yc+(sin(angle*pi/180)*d);
        x =(cos(angle*pi/180)*d);
        glVertex2f(x+x1,y-y1);
}
}

float checkBubPos(float y,float c){
    if(y<500){
        return y+c;
    }
    else{
        return 0;
    }
}

void circle(int x){

    //to draw circles
    glBegin(GL_POINTS);
    circleFunc(y21,15,x+10,30);
    circleFunc(y31,5,x,60);
    circleFunc(y41,7,x+10,90);
    circleFunc(y51,12,x,120);
    circleFunc(y61,15,x+10,150);
    circleFunc(y71,5,x,180);
    circleFunc(y81,3,x+10,210);
    circleFunc(y81,15,x,240);
    circleFunc(y91,12,x+10,270);
    circleFunc(y10,10,x,300);
    circleFunc(y12,16,x+10,330);
    circleFunc(y13,15,x,360);
    circleFunc(y14,10,x+10,400);
glEnd();

y11 = checkBubPos(y11,2.0);
y21 = checkBubPos(y21,3.0);
y31 = checkBubPos(y31,4.5);
y41 = checkBubPos(y41,7.0);
y51 = checkBubPos(y51,6.5);
y61 = checkBubPos(y61,18.0);
y71 = checkBubPos(y71,17.5);
y81 = checkBubPos(y81,8.0);
y91 = checkBubPos(y91,7.5);
y10 = checkBubPos(y10,10.0);
y12 = checkBubPos(y12,11.0);
y14 = checkBubPos(y14,8.0);
y12 = checkBubPos(y12,9.0);
y13 = checkBubPos(y13,1.0);
glutPostRedisplay();

}

void plant3L(int x1, int x2, int m){
    int dis = x2-x1;
    dis = dis/3;
```

```
   glColor3ub( 40,170,5);
   glBegin(GL_POLYGON);
   glVertex2f(x1,0);
   glVertex2f(x1-10,m-10);
   glColor3f(0.0,0.5,0.0);
   glVertex2f(x1+dis,15);
   glVertex2f(x1+((x2-x1)/2),m);
   glVertex2f(x2-dis,15);
   glVertex2f(x2+12,m+10);
   glVertex2f(x2,0);
   glEnd();
}

void plant2L(int x1, int x2, int h1, int h2){

   glColor3ub( 60,170,15);
   glBegin(GL_POLYGON);
   glVertex2f(x1,0);
   glVertex2f(x1-20,h1);
   glColor3f(0.0,0.3,0.0);
   glVertex2f(x2,0);
   glVertex2f(x1+15,h2);
   glEnd();
}

void plant()
{
   plant2L(50,40,40,60);
   plant2L(95,85,50,60);
   plant2L(120,110,45,65);
   plant2L(70,60,60,40);
   plant2L(140,130,43,60);
   plant2L(950,940,40,50);
   plant2L(870,860,50,60);
   plant2L(470,460,40,50);
   plant2L(490,480,55,65);
   plant2L(520,510,40,50);
   plant2L(540,530,50,60);
   plant2L(700,690,40,50);
   plant2L(730,720,55,65);
   plant2L(795,785,40,50);

   plant3L(10,20,60);
   plant3L(160,170,50);
   plant3L(310,320,70);
   plant3L(435,445,60);
   plant3L(750,760, 55);
   plant3L(820,830,80);
   plant3L(900,910, 55);
}

void drawLeafLeft(int x, int y){

   glLineWidth(5);
   glColor3ub( 60,120,15);
   glBegin(GL_LINES);
   glVertex2f(x,y);
   glVertex2f(x-10,y);
   glEnd();

   glColor3ub( 60,170,15);
   glBegin(GL_POLYGON);
   glVertex2f(x-10,y);
   glVertex2f(x-17,y+18);
   glVertex2f(x-50,y+13);
```

```
    glColor3f(0.1,0.5,0.0);
    glVertex2f(x-70,y);
    glVertex2f(x-50,y-13);
    glVertex2f(x-17,y-18);
    glEnd();
}

void drawLeafRight(int x, int y){

    glLineWidth(5);
    glColor3ub( 60,120,15);
    glBegin(GL_LINES);
    glVertex2f(x,y);
    glVertex2f(x+10,y);
    glEnd();

    glColor3ub( 60,170,15);
    glBegin(GL_POLYGON);
    glVertex2f(x+10,y);
    glVertex2f(x+17,y+18);
    glVertex2f(x+50,y+13);
    glColor3f(0.1,0.5,0.0);
    glVertex2f(x+70,y);
    glVertex2f(x+50,y-13);
    glVertex2f(x+17,y-18);
    glEnd();

}

void drawSineWave(){
    glPointSize(8.0);
    glBegin(GL_POINTS);
    for(angle=0; angle<500.0; angle=angle+0.9){
        glColor3ub(46,110,7);
        x =(sin(angle*pi/90)*15);
        y = angle;
        glVertex2f(x+960,y);
    }
    glEnd();
    drawLeafLeft(960,80);
    drawLeafLeft(969,210);
    drawLeafLeft(945,330);
    drawLeafLeft(957,460);
    drawLeafRight(945,140);
    drawLeafRight(957,280);
    drawLeafRight(971,410);
}

void mov(void){
    if(a>=-500)
        a=a-7;
    else
        a=700;

    if(a<-500){
        aa=aa-100;
    }

    if(aa<-251){
        aa=100;
    }

    if(b<=1000)
        b=b+8.0;
    else
```

```
        b=-500;

    if(b>1000){
        bb=bb+150;
    }
    if(bb>251){
        bb=50;
    }

    if(s>=-700)
        s=s-9;
    else
        s=700;

    if(s<-700){
        ss=ss-250;
    }

    if(ss<-251){
        ss=80;
    }
}

void HuntShark(){
        //For Shark Body
    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<320; angle+=2){
        glColor3ub(1,0,0);
        y =(sin(angle*pi/319)*30);
        x =angle;
        glVertex2f(x+280+s,y+285+ss);
    }
    glColor3ub(200,200,200);
    glVertex2f(x+280+s,y+265+ss);
    for(angle=298; angle>35; angle-=2){
        x =angle;
        y = -(sin(angle*pi/319)*20);
        glVertex2f(x+280+s,y+265+ss);
    }

    glVertex2f(300+s,265+ss);
    glEnd();
    glColor3ub(138,3,3);
    glBegin(GL_POLYGON);
    glVertex2f(302+s,263+ss);
    glVertex2f(330+s,276+ss);
    glVertex2f(297+s,268+ss);
    glEnd();

    //For Shark Tail
    glColor3ub(115,116,117);
    glBegin(GL_POLYGON);
    glVertex2f(598+s,285.6+ss);
    glVertex2f(640+s,330+ss);
    glColor3ub(rand()%40+190,rand()%40+190,rand()%40+190);
    glVertex2f(628+s,275+ss);
    glVertex2f(640+s,220+ss);
    glVertex2f(598+s,265+ss);
    glEnd();

    glColor3ub(100,106,107);
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(410+s,313+ss);
    glColor3ub(150,150,150);
    glVertex2f(460+s,360+ss);
```

```
    glVertex2f(475+s,313+ss);
    glEnd();

    glColor3ub(115,116,117);
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(420+s,260+ss);
    glVertex2f(450+s,210+ss);
    glColor3ub(rand()%40+190,rand()%40+190,rand()%40+190);
    glVertex2f(460+s,260+ss);
    glEnd();

    glColor3ub(115,116,117);
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(555+s,297+ss);
    glVertex2f(580+s,305+ss);
    glVertex2f(580+s,290+ss);
    glEnd();

    glColor3ub(115,116,117);
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(555+s,257+ss);
    glVertex2f(580+s,247+ss);
    glColor3ub(rand()%40+190,rand()%40+190,rand()%40+190);
    glVertex2f(580+s,261+ss);
    glEnd();

    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<360.0; angle+=.1){
        glColor3ub(0,0,0);
        y =(sin(angle*pi/180)*5);
        x =(cos(angle*pi/180)*5);
        glVertex2f(x+332+s,y+287+ss);
    }
    glEnd();

    glColor3f(1.0,1.0,1.0);
    glPointSize(4.0);
    glBegin(GL_POINTS);
    glVertex2f(331.8+s,287+ss);
    glEnd();
}

void displayHunt1(){
    glClear(GL_COLOR_BUFFER_BIT);
    if(moving==1){
        mov();
    }
    pb1();
    pb2();
    pb();
    plant();
    fish1();
    drawSineWave();
    HuntShark();
    drawFrog();

    glColor3f(1.0,1.0,1.0);
    glPointSize(2.0);
    circle(30);
    circle(930);


    if(flag3==0){
        screen();
    }
```

```
    glFlush();
    glutSwapBuffers();
}

void displayHunt2(){
    glClear(GL_COLOR_BUFFER_BIT);
    if(moving==1){
        mov();
    }
    pb1();
    pb2();
    pb();
    plant();
    drawSineWave();
    HuntShark();
    drawFrog();

    glColor3f(1.0,1.0,1.0);
    glPointSize(2.0);
    circle(30);
    circle(930);


    if(flag3==0){
        screen();
    }

    glFlush();
    glutSwapBuffers();
}

void subMenu(int opt){
    switch(opt){
        case 1:displayHunt1();
        glutDisplayFunc(displayHunt1);
        break;
        case 2:displayHuntMsg2();
        glutDisplayFunc(displayHunt2);
        break;
        default: glutDisplayFunc(displayShark);
    }
}

void create_menu(void){

    int sm = glutCreateMenu(subMenu);
    glutAddMenuEntry("Hunt 1 fish",1);
    glutAddMenuEntry("Hunt 2 fishes",2);
    glutCreateMenu(menu);
    glutAddMenuEntry("Fish", 1);
    glutAddMenuEntry("Shark", 2);
    glutAddMenuEntry("Back", 3);
    glutAddMenuEntry("Exit", 4);
    glutAddSubMenu("Hunt options",sm);
    glutAttachMenu(GLUT_LEFT_BUTTON);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
}

void speed()
{
a=a-.0;
s=s-10;
b=b+10;
}
```

```
void slow()
{
a=a-0.0001;
}

void key(unsigned char k,int x,int y)
{
if(k=='i')
      glutIdleFunc(speed);

if(k=='d'){
      glutIdleFunc(slow);
}

if(k=='p'){
      flag3=1;
      flag4=1;
}

}

void drawFrog(){

   glPushMatrix();
   glTranslatef(600,55,0);
   glRotatef(50, 0, 0, 1);
   glBegin(GL_TRIANGLE_FAN);
   for(angle=0; angle<360.0; angle+=.1){
      glColor3ub(46,127,7);
      y =(sin(angle*pi/180)*15);
      x =(cos(angle*pi/180)*7);
      glVertex2f(x,y);
   }
   glEnd();
   glPopMatrix();

   glPushMatrix();
   glTranslatef(630,55,0);
   glRotatef(-50, 0, 0, 1);
   glBegin(GL_TRIANGLE_FAN);
   for(angle=0; angle<360.0; angle+=.1){
      glColor3ub(46,127,7);
      y =(sin(angle*pi/180)*15);
      x =(cos(angle*pi/180)*7);
      glVertex2f(x,y);
   }
   glEnd();
   glPopMatrix();

   glBegin(GL_TRIANGLE_FAN);
   for(angle=0; angle<360.0; angle+=.1){
      glColor3ub(46,150,7);
      y =(sin(angle*pi/180)*21);
      x =(cos(angle*pi/180)*13);
      glVertex2f(x+615,y+65);
   }
   glEnd();
   glBegin(GL_TRIANGLE_FAN);
   for(angle=0; angle<360.0; angle+=.1){
      glColor3ub(46,130,7);
      y =(sin(angle*pi/180)*10);
      x =(cos(angle*pi/180)*17);
      glVertex2f(x+615,y+85);
   }
```

```
    glEnd();
    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<360.0; angle+=.1){
        glColor3ub(255,255,255);
        y =(sin(angle*pi/180)*7);
        x =(cos(angle*pi/180)*5);
        glVertex2f(x+605,y+92);
    }
    glEnd();
    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<360.0; angle+=.1){
        glColor3ub(255,255,255);
        y =(sin(angle*pi/180)*7);
        x =(cos(angle*pi/180)*5);
        glVertex2f(x+625,y+92);
    }
    glEnd();

    glBegin(GL_TRIANGLE_FAN);
    for(angle=180; angle<360.0; angle+=.1){
        glColor3ub(255,255,255);
        y =(sin(angle*pi/180)*7);
        x =(cos(angle*pi/180)*7);
        glVertex2f(x+615,y+82);
    }
    glEnd();

    glColor3f(0.0,0.0,0.0);
    glPointSize(6.0);
    glBegin(GL_POINTS);
    glVertex2f(605,92);
    glVertex2f(625,92);
    glEnd();

    glColor3ub(46,115,7);
    glBegin(GL_POLYGON);
    glVertex2f(605,70);
    glVertex2f(600,42);
    glVertex2f(604,42);
    glVertex2f(611,70);
    glEnd();

    glColor3ub(46,115,7);
    glBegin(GL_POLYGON);
    glVertex2f(620,70);
    glVertex2f(627,42);
    glVertex2f(631,42);
    glVertex2f(626,70);
    glEnd();


}

void Shark(){

    //For Shark Body
    glBegin(GL_TRIANGLE_FAN);
    for(angle=0; angle<320; angle+=2){
        glColor3ub(115,116,117);
        y =(sin(angle*pi/319)*30);
        x =angle;
        glVertex2f(x+280+s,y+285+ss);
    }
    glColor3ub(200,200,200);
    glVertex2f(x+280+s,y+265+ss);
```

```
for(angle=298; angle>35; angle-=2){
   x =angle;
   y = -(sin(angle*pi/319)*20);
   glVertex2f(x+280+s,y+265+ss);
}

glVertex2f(300+s,265+ss);
glEnd();
glColor3ub(138,3,3);
glBegin(GL_POLYGON);
glVertex2f(302+s,263+ss);
glVertex2f(330+s,276+ss);
glVertex2f(297+s,268+ss);
glEnd();

//For Shark Tail
glColor3ub(115,116,117);
glBegin(GL_POLYGON);
glVertex2f(598+s,285.6+ss);
glVertex2f(640+s,330+ss);
glColor3ub(rand()%40+190,rand()%40+190,rand()%40+190);
glVertex2f(628+s,275+ss);
glVertex2f(640+s,220+ss);
glVertex2f(598+s,265+ss);
glEnd();

glColor3ub(100,106,107);
glBegin(GL_TRIANGLE_FAN);
glVertex2f(410+s,313+ss);
glColor3ub(150,150,150);
glVertex2f(460+s,360+ss);
glVertex2f(475+s,313+ss);
glEnd();

glColor3ub(115,116,117);
glBegin(GL_TRIANGLE_FAN);
glVertex2f(420+s,260+ss);
glVertex2f(450+s,210+ss);
glColor3ub(rand()%40+190,rand()%40+190,rand()%40+190);
glVertex2f(460+s,260+ss);
glEnd();

glColor3ub(115,116,117);
glBegin(GL_TRIANGLE_FAN);
glVertex2f(555+s,297+ss);
glVertex2f(580+s,305+ss);
glVertex2f(580+s,290+ss);
glEnd();

glColor3ub(115,116,117);
glBegin(GL_TRIANGLE_FAN);
glVertex2f(555+s,257+ss);
glVertex2f(580+s,247+ss);
glColor3ub(rand()%40+190,rand()%40+190,rand()%40+190);
glVertex2f(580+s,261+ss);
glEnd();

glBegin(GL_TRIANGLE_FAN);
for(angle=0; angle<360.0; angle+=.1){
   glColor3ub(0,0,0);
   y =(sin(angle*pi/180)*5);
   x =(cos(angle*pi/180)*5);
   glVertex2f(x+332+s,y+287+ss);
}
glEnd();
```

```
    glColor3f(1.0,1.0,1.0);
    glPointSize(4.0);
    glBegin(GL_POINTS);
    glVertex2f(331.8+s,287+ss);
    glEnd();

}

void display(void){

    glClear(GL_COLOR_BUFFER_BIT);
    if(moving==1){
        mov();
    }
    pb1();
    pb2();
    pb();
    plant();
    fish1();
    drawSineWave();
    fish2();
    drawFrog();

    glColor3f(1.0,1.0,1.0);
    glPointSize(2.0);
    circle(30);
    circle(930);


    if(flag3==0){
        screen();
    }

    glFlush();
    glutSwapBuffers();
}

void displayShark(void){

    glClear(GL_COLOR_BUFFER_BIT);
    if(moving==1){
        mov();
    }
    pb1();
    pb2();
    pb();
    plant();
    fish1();
    drawSineWave();
    fish2();
    Shark();
    drawFrog();

    glColor3f(1.0,1.0,1.0);
    glPointSize(2.0);
    circle(30);
    circle(930);


    if(flag3==0){
        screen();
    }

    glFlush();
```

```
    glutSwapBuffers();
}

void displayFish(void){

    glClear(GL_COLOR_BUFFER_BIT);
    if(moving==1){
        mov();
    }
    pb1();
    pb2();
    pb();
    plant();
    fish1();
    drawSineWave();
    fish2();
    drawFrog();

    glColor3f(1.0,1.0,1.0);
    glPointSize(2.0);
    circle(30);
    circle(930);


    if(flag3==0){
        screen();
    }

    glFlush();
    glutSwapBuffers();
}

void menu(int val){
    switch (val) {
        case 1:
            glutDisplayFunc(displayFish);
            break;
        case 2:
            glutDisplayFunc(displayShark);
            moving = 1;
            break;
        case 3:
            moving = 1;
            flag3=0;
            flag4=0;
            screen();
            break;
        case 4: exit(0);
            break;

    }
}

int main(int argc,char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowPosition(0,0);
    glutInitWindowSize(1200,600);
    glutCreateWindow("AQUARIUM");
    init();
    glutReshapeFunc(reshape);
    glutKeyboardFunc(key);
    glutDisplayFunc(display);
    create_menu();
```

```
    glutMainLoop();
}
```

CHAPTER 5
TESTING AND SNAPSHOTS

5.1 Testing

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

5.1 Test Cases

SLNO	Name of the test	Expected output	Result	Remarks
1.	With the press of the key 'P'	Ocean theme is Shown.	Displays fishes in ocean theme.	Pass
2.	With right click of mouse button	Menu should appear	Displays Menu	Pass
3.	When 'Hunt 1 Fish' is clicked	1 fish should be less in tank.	1 fish is dead, hence out of tank.	Pass
4.	When 'Hunt 2 Fish' is clicked	Both fishes disappear from tank.	Both fishes disappear. Only shark remains.	Pass
5.	When 'back' option is clicked.	Display initial details screen.	Displays initial details screen.	Pass

Table 5.1-Test cases

5.2 Snapshots

In computer file systems, a snapshot is a copy of a set of files and directories as they were at a particular point in the past. The term was coined as an analogy to that in photography. One approach to safely backing up live data is to temporarily disable write access to data during the backup, either by stopping the accessing applications or by using the locking API provided by the operating system to enforce exclusive read access. This is tolerable for low-availability systems.

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

Snapshots are the pictures representing different phases of the program execution. The figure below shows the initial window of the project ‘Shark Hunt'

Fig 5.1: Snapshot-Initial Screen

The above snapshot displays the Initial screen of the mini project which describes the topic and provides an interactive message that goes to the next screen.

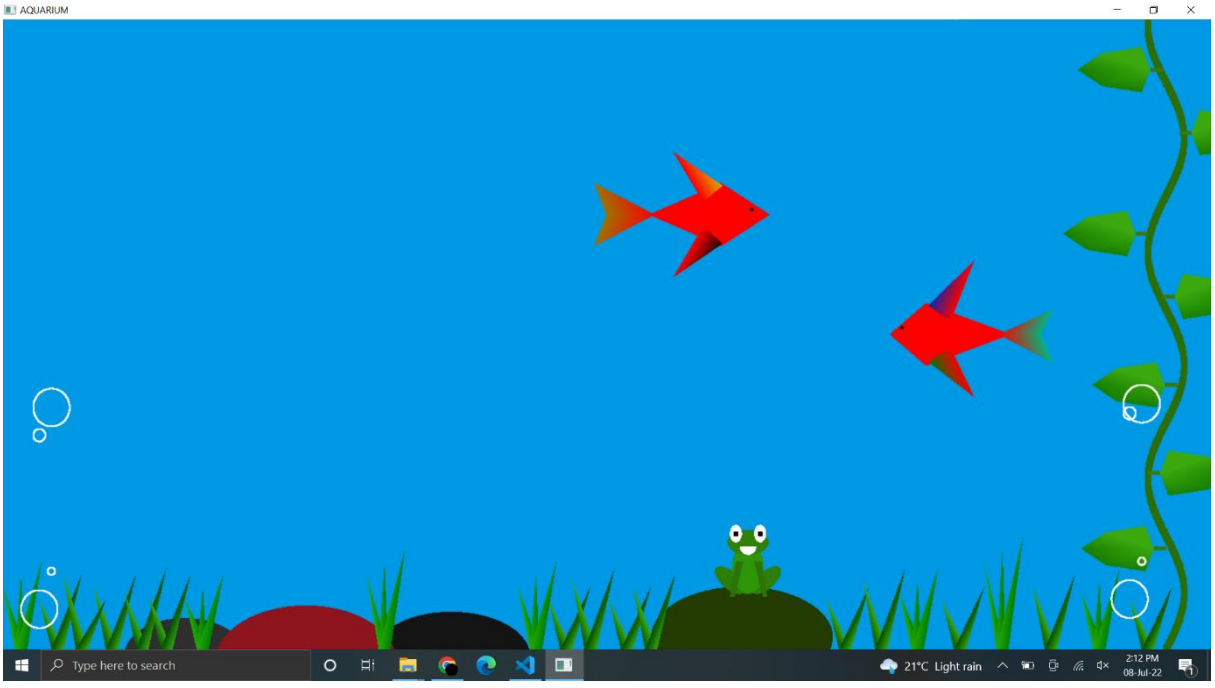

Fig 5.2: Snapshot- Inside the project

The above snapshot displays the aquarium with 2 fishes and frog rendered inside.

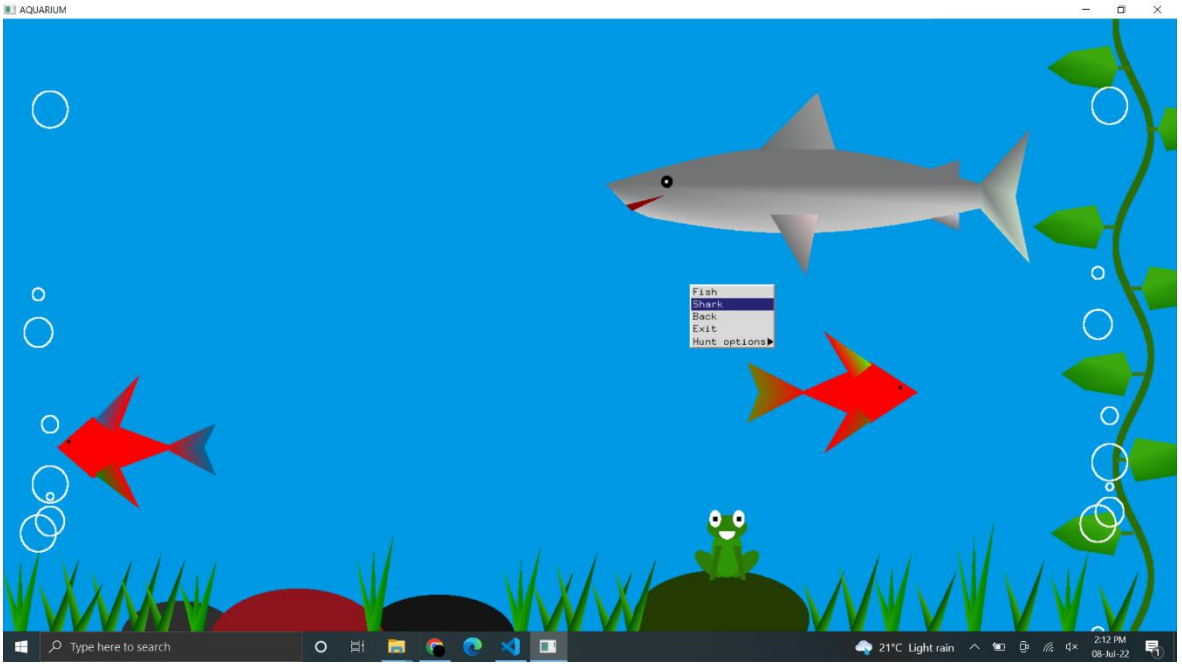

Fig 5.3: Snapshot-Aquarium menu

The above snapshot displays the screen which includes the menu from which user can chose what to render on screen.

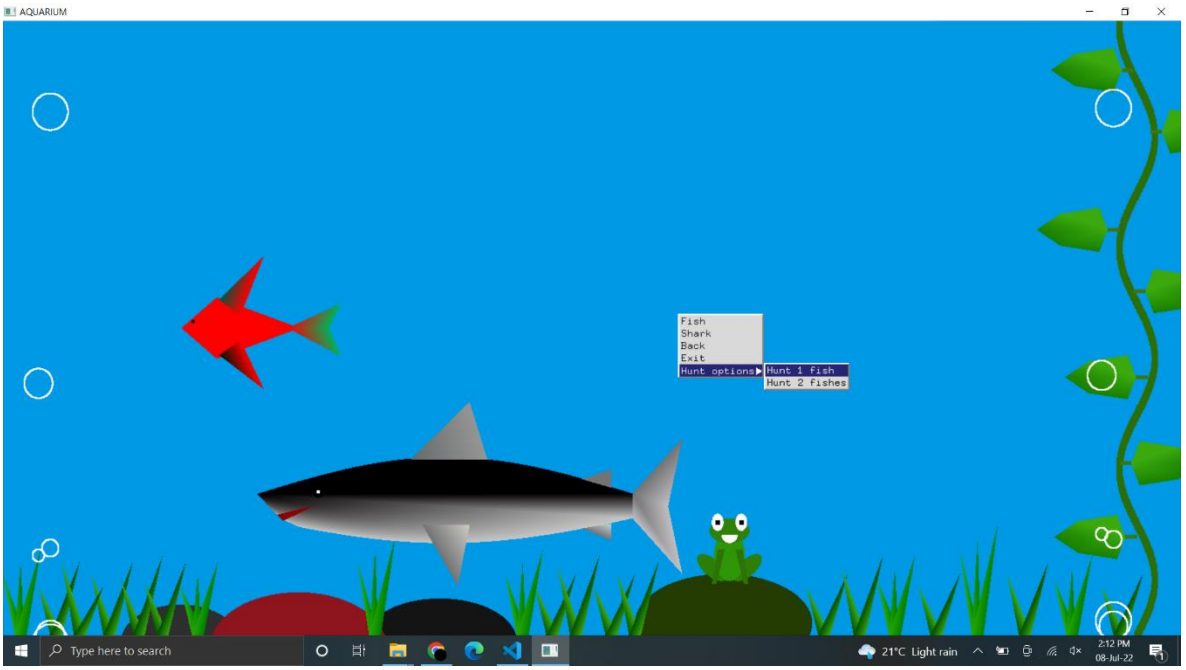

Fig 5.4: Snapshot-Shark Hunting Mode

The above snapshot displays the shark in hunting mode, user can choose to hunt 1 fish and 1 fish will be removed from the tank.

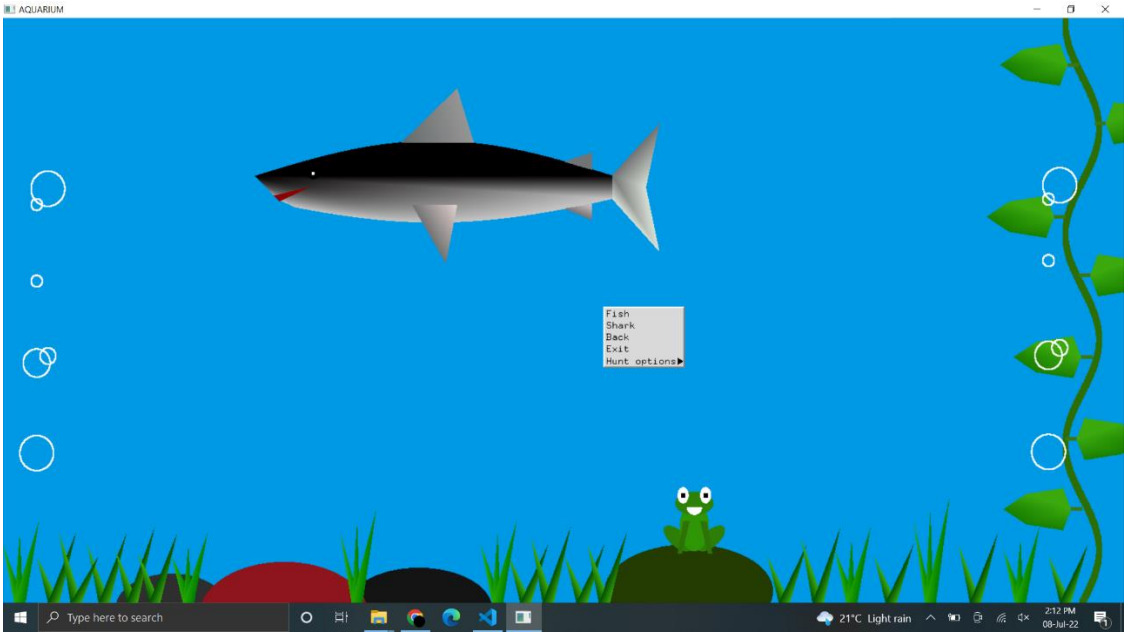

Fig 5.5: Snapshot-Shark Hunting mode

The above snapshot displays the tank after shark has hunted both fishes. The back button in the menu leads back to the initial screen.

CONCLUSION

It has been an interesting journey through the development of this project. At the beginning we used our limited knowledge to implement only the basic features. However, through the months of development, new issues and bugs led to new ideas which led to newer methods of implementation which, in turn, led to us learning even more features of the OpenGL and apply more creative and efficient ways to perform the older functions. New methods helped us in adding flexibility to the various parts of the program, making the further addition of newer features easier and less time consuming which, again, led to the possibility of adding even more features. This sequential chain reaction of progress and ideas has enabled to learn so much through the months of working on this project and we have done our best to add as many features as we could and provide a user interface that is easy and intuitive to use.

Before concluding, it is worth mentioning that this project would never have been possible without the tremendous amount of encouragement by the staff and guides of our department.

We are content with the outcome of this project and are hopeful that it meets the requirements expected and we wish that it may inspire others to be creative and critical in the field of computer graphics and have a newfound appreciation for the Open Graphics Library.

FUTURE ENHANCEMENT

Future scope of this very engaging project is vast as a lot of animations can be further added to and could be made more interactive with the user.

To list out a few enhancements :

- The hunting of the shark can be shown graphically.
- Waves effect of the water can be shown to simulate an ocean.
- We can simulate multiple sharks and perform more activities like stunts.
- We can simulate the fishes trying to escape
- We can replace fishes with humans and show how sharks hunt humans.

REFERENCES

The following books have been referred to complete this project.

[1] Interactive Computer Graphics - A Top down Approach Using OpenGL by Edward Angel.

[2] Computer Graphics using OpenGL by F.S. Hill,Jr and Stephen M. Kelly

[3] OpenGL Tutorials.