

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

BELGAVI, KARNATAKA -590 018



A Mini-Project Report on

“THREE-DIMENSIONAL MODEL OF SOLAR SYSTEM”

*Submitted in partial fulfillment for the Computer Graphics Laboratory with
Mini-Project (18CSL67) course of Sixth Semester of Bachelor of Engineering
in Computer Science & Engineering during the academic year 2022-23.*

By

Team Code: **CGP2023-C05**

CHAITHRASHREE S

4MH20CS016

BHOOMIKA S

4MH21CS401

: Under the Guidance of:

Prof. Bhavya M R

Assistant Professor

Department of CS&E

MIT Mysore



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MAHARAJA INSTITUTE OF TECHNOLOGY MYSORE

Belawadi, S.R. Patna Taluk, Mandya Dist-571477.

Accredited By:



2022-23

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
MAHARAJA INSTITUTE OF TECHNOLOGY MYSORE**



~~ CERTIFICATE ~~



*Certified that the mini-project work entitled “**THREE-DIMENSIONAL MODEL OF SOLAR SYSTEM**” is a bonafide work carried out by **CHAITHRASHREE S(4MH20CS016)** & **BHOOMIKA S (4MH21CS401)** for the Computer Graphics Laboratory with Mini-Project (18CSL67) of Sixth Semester in Computer Science & Engineering under Visvesvaraya Technological University, Belagavi during academic year 2022-23.*

It is certified that all corrections/suggestions indicated for Internal Assignment have been incorporated in the report. The report has been approved as it satisfies the course requirements.

Signature of Guide

Prof. Bhavya M R

Assistant Professor

Dept. of CS&E

MIT Mysore

Signature of the HOD

Dr. Shivamurthy R C

Professor & HOD

Dept. of CS&E

MIT Mysore

External viva

Name of the Examiners

Signature with date

1).....

2)

~~~~ ACKNOWLEDGEMENT ~~~~

It is the time to acknowledge all those who have extended their guidance, inspiration, and their wholehearted co-operation all along our project work.

We are also grateful to **Dr. B G Naresh Kumar**, principal, MIT Mysore and **Dr. Shivamurthy R C**, HOD, CS&E, MIT Mysore for having provided us academic environment which nurtured our practical skills contributing to the success of our project.

We wish to place a deep sense of gratitude to all Teaching and Non-Teaching staffs of Computer Science and Engineering Department for wholehearted guidance and constant support without which Endeavour would not have been possible.

Our gratitude will not be complete without thanking our parents and our friends, who have been a constant source of support and aspirations

CHAITHRASHREE S

BHOOMIKA S

~~~ ABSTRACT ~~~

In this project, we strive to obtain a 3-Dimensional Model of the Solar System. The principle behind the working of the project is that, we use spheres to create the objects and use images for the textures. We achieve this by taking an image of .JPEG or .JPG form and use certain tools to obtain a True vision Graphics Adapter (.tga) File. This file format is widely used for 3D model texture mapping. Resizing is done internally when this process occurs. we specify the width and the depth scaling during the conversion.

We use this file to generate textures for the objects. The user-defined functions handle the image loading and displaying of the texture is obtained using API of OpenGL. We use OpenGL for rendering of the object. We achieve this by storing the textures in an array. This array contains all the textures required for rendering the model. We make use of Lighting API of OpenGL to make the model seem better. Menu list is created using the OpenGL API to display individual objects separately. The model is provided with certain operations that can be performed by the viewer, which is executed using the OpenGL API for keyboard operations. With all the data on hand, we create a 3-Dimensional Model of the Solar System.

We make use of C with OpenGL for entire coding purpose along with some features of Windows. The OpenGL Utility is a Programming Interface. We use light functions to add luster, shade and shininess to graphical objects. The toolkit supports much more functionalities like multiple window rendering, callback event driven processing using sophisticated input devices etc.

~~~~~ CONTENTS ~~~~~

1. INTRODUCTION	01
1.1 Aim of the Project	01
1.2 Overview of the Project.....	01
1.3 Outcome of the Project.....	02
 2. DESIGN AND IMPLEMENTATION	 03
2.1 OpenGL technology.....	03
2.2 Flow Chart	05
2.3 OpenGL API's Used with Description	06
2.4 Source Code	18
 3. RESULT ANALYSIS	 29
3.1 Snap Shots	29
3.2 Discussion	32
 4. CONCLUSION AND FUTURE WORK	 33
4.1 Conclusion	33
4.2 Future Enhancement	33
 5. REFERENCES	 34

CHAPTER - 1

INTRODUCTION

1.1 Aim

Our project aims at displaying the 3D model of the solar system in OpenGL. Using built in functions provided by graphics library and integrating it with the C implementation of list it was possible to visually represent the objects in a 3D plane.

1.2 Overview

Computer graphics is the process of making the design, 2D, 3D and animation of an object. Computer graphics can do many things, including modeling, simulation and visualization of an object or a problem. Modeling is a representation of how people describe or explain an object, system, or a concept, which is usually manifested by simplification or idealization. This can be represented by physical models (mockups, prototypes), the model image (design drawings, computer images), or mathematical formulas.

Solar system is a set of celestial bodies bound by gravitational forces. The movement of celestial bodies like the sun, stars, planets and the other will be more easily understood if taught through visualization movement through computer animation. This visualization shows the solar system planetary motion, or we can call it a revolution, that is, when the planets move around the sun, and remain in orbit each using OpenGL API to represent the solar system as a visual. OpenGL support this modeling capability as OpenGL has additional features to better produce something more realistic. OpenGL allows us to create a graph that can be run on any operating system only minor adjustments.

The principle behind the working of the project is that, we use spheres to create the objects and use images for the textures. We achieve this by taking an image of .JPEG or .JPG form and use certain tools to obtain a True vision Graphics Adapter (.tga) File. This file format is widely used for 3D model texture mapping. Resizing is done internally when this process occurs; we specify the width and the depth scaling during the conversion.

We use this file to generate textures for the objects. The user-defined functions handle the image loading and displaying of the texture is obtained using API of OpenGL. We use OpenGL for rendering of the object. We achieve this by storing the textures in an array. This array contains all the textures required for rendering the model. We make use of Lighting API of OpenGL to make the model seem better. Menu list is created using the OpenGL API to display individual objects separately. The model is provided with certain operations that can be performed by the viewer, which is executed using the OpenGL API for keyboard operations. With all the data on hand, we create a 3-Dimensional Model of the Solar System.

1.3 Outcome

The development of technology has increased very rapidly and can be found in almost all areas of human life and in the field of education. Nowadays, education learning is using technology in order to perform the delivery of material will become more interesting and easier memorable. The use of technology in education can be implemented with a visualization on a subject of study. The study of the solar system for example, by using the visualization of objects in the solar system would facilitate teachers for the delivery of content. Visualization of the solar system is modified in graphic or animation to display a collection of celestial objects that consist a large star called the sun, and all objects that are bound by the force of gravity. The objects are eight planets that make a revolution or rotation of the sun and remain in orbit respectively. This project is developed using Code Blocks and this project is implemented by making extensive use of library functions offered by graphics package of OpenGL

CHAPTER – 2

DESIGN AND IMPLEMENTATION

2.1 OpenGL Technology

OpenGL is a graphics application programming interface (API) which was originally developed by Silicon Graphics. OpenGL is not in itself a programming language, like C++, but functions as an API that can be used as a software development tool for graphics applications.

The term Open is significant in that OpenGL is operating system independent. GL refers to graphics language. OpenGL also contains a standard library referred to as the OpenGL Utilities (GLU). GLU contains routines for setting up viewing projection matrices and describing complex objects with line and polygon approximations. OpenGL gives the programmer an interface with the graphics hardware. OpenGL is a lowlevel, widely supported modelling and rendering software package, available on all platforms. It can be used in a range of graphics applications, such as games, CAD design, modelling. OpenGL is the core graphics rendering option for many 3D games, such as Quake 3. The providing of only low-level rendering routines is intentional because this gives the programmer a great control and flexibility in his applications. These routines can easily be used to build highlevel rendering and modelling libraries.

The OpenGL Utility Library (GLU) does exactly this, and is included in most OpenGL distributions! OpenGL was originally developed in 1992 by Silicon Graphics, Inc., (SGI), as a multi-purpose platform independent graphics API. Since 1992 all of the development of 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 are using. Visualization Programming Pipeline:

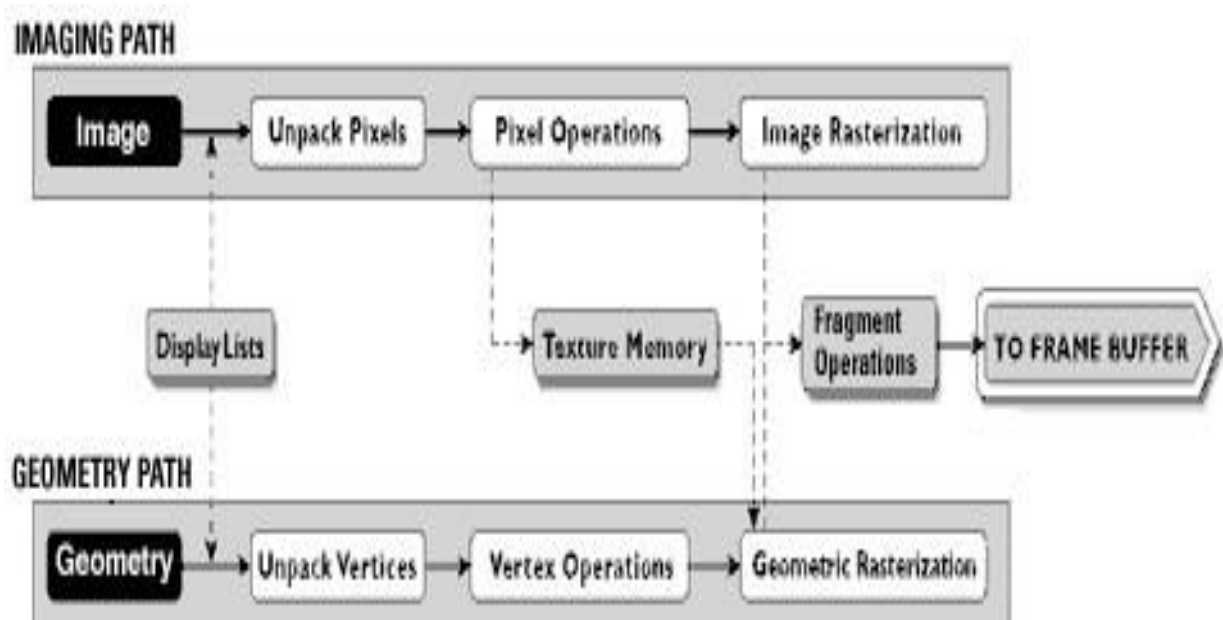


Figure. 2.1: OpenGL Visualization Programming Pipeline

OpenGL operates on image data as well as geometric primitives. Simplifies Software Development, Speeds Time-to-Market.

Routines simplify the development of graphics software—from rendering a simple geometric point, line, or filled polygon to the creation of the most complex lighted and texture mapped NURBS curved surface. OpenGL gives software developers access to geometric and image primitives, display lists, modeling transformations, lighting and texturing, anti-aliasing, blending, and many other features. Every conforming OpenGL implementation includes the full complement of OpenGL functions.

The well-specified OpenGL standard has language bindings for C, C++, FORTRAN, Ada, and Java. All licensed OpenGL implementations come from a single specification and language binding document and are required to pass a set of conformance tests.

2.2 Flow Chart

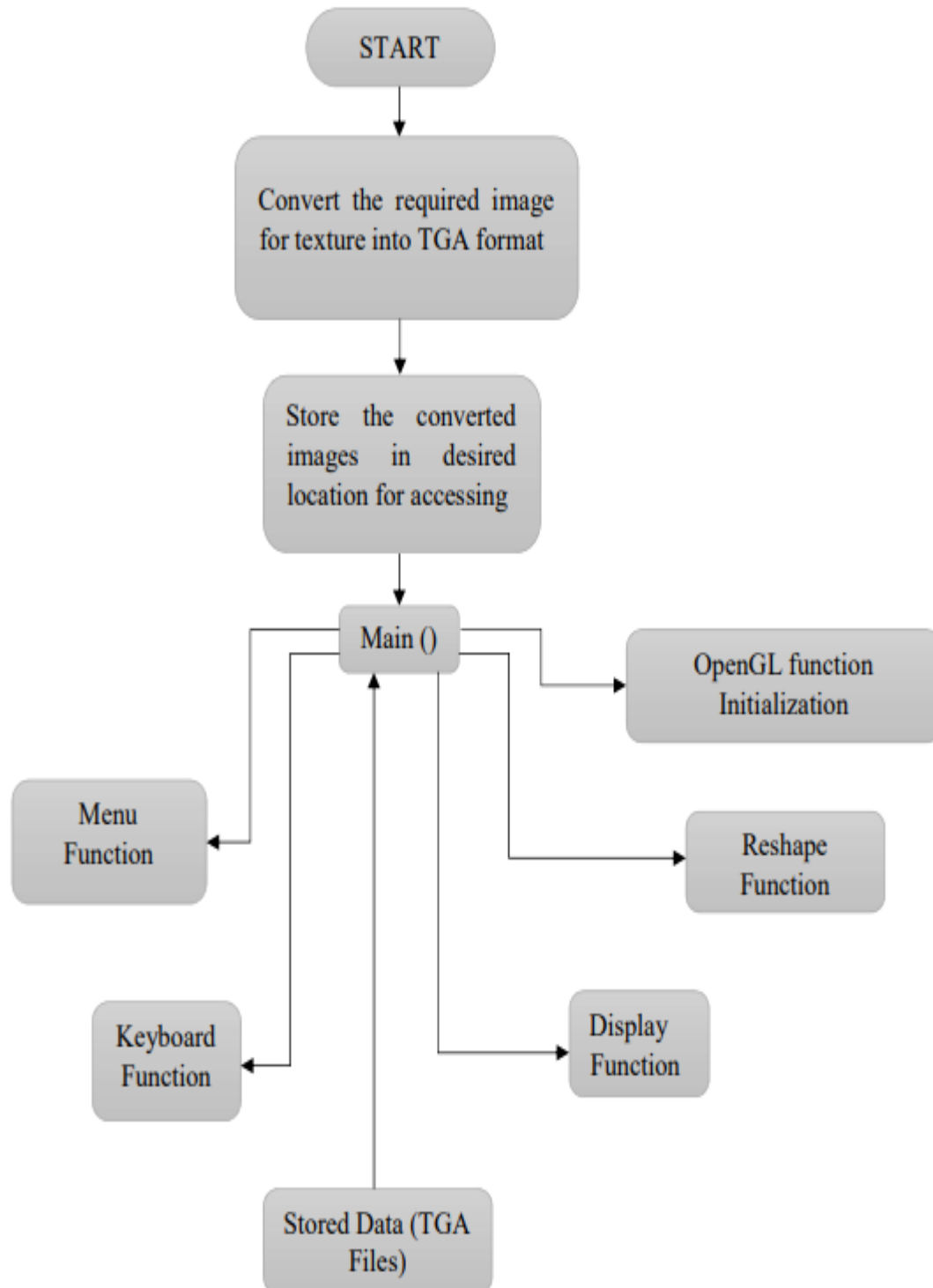


Fig:2.2 The flow diagram for displaying the 3D Model of Solar System

OpenGL is a software tool for developing the graphics objects. OpenGL library called GLUT i.e. Graphics Library Utility toolkit supports graphics system with the necessary modelling and rendering techniques. The Lighting system is a technique for displaying graphic objects on the monitor and displaying the light effects. It provides the following functionalities.

- **Initialization**

This function is the initial stage of the system where the system initializes the various aspects of the graphics system based on the user requirements, which include Command line processing, window system initialization and also the initial window creation state is controlled by these routines.

- **Event Processing**

This routine enters GLUT's event processing loop. This routine never returns, and it continuously calls GLUT callback as and when necessary. This can be achieved with the help of the callback registration functions. These routines register callbacks to be called by the GLUT event processing loop.

2.3 OpenGL API'S Used with Description

This project is developed using CodeBlocks and this project is implemented by making extensive use of library functions offered by graphics package of OpenGL, a summary of those functions follows:

- **glBegin() :**

Specifies the primitives that will be created from vertices presented between glBegin and subsequent glEnd. GL_POLYGON, GL_LINE_LOOP etc.

- **glEnd(void):**

It ends the list of vertices.

- **glPushMatrix():**

void glPushMatrix (void);

`glPushMatrix` pushes the current matrix stack down by one level, duplicating the current matrix.

- **`glPopMatrix()` :**

`void glPopMatrix (void);`

`glPopMatrix` pops the top matrix off the stack, destroying the contents of the popped matrix. Initially, each of the stacks contains one matrix, an identity matrix.

- **`glTranslate()` :**

`void glTranslate(GLdouble x, GLdouble y, GLdouble z);`

Translation is an operation that displaces points by a fixed distance in a given direction. Parameters `x`, `y`, `z` specify the `x`, `y`, and `z` coordinates of a translation vector. Multiplies current matrix by a matrix that translates an object by the given `x`, `y` and `z`-values.

- **`glClear()` :**

`void glClear(GLbitfield mask);`

`glClear` takes a single argument that is the bitwise or of several values indicating which buffer is to be cleared. `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_ACCUM_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT`. Clears the specified buffers to their current clearing values.

- **`glClearColor()` :**

`void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);`

Sets the current clearing color for use in clearing color buffers in RGBA mode. The red, green, blue, and alpha values are clamped if necessary to the range `[0,1]`. The default clearing color is `(0, 0, 0, 0)`, which is black.

- **`glMatrixMode()` :**

`void glMatrixMode(GLenum mode);`

It accepts three values `GL_MODELVIEW`, `GL_PROJECTION` and `GL_TEXTURE`. It specifies which matrix is the current matrix. Subsequent transformation commands affect the specified matrix.

- **`glutInitWindowPosition()` :**

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

This API will request the windows created to have an initial position. The arguments x, y indicates the location of a corner of the window, relative to the entire display.

- **glLoadIdentity() :**

```
void glLoadIdentity(void);
```

It replaces the current matrix with the identity matrix.

- **glutInitWindowSize() :**

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

The API requests windows created to have an initial size. The arguments 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.

- **glutInitDisplayMode():**

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

Specifies the display mode, normally the bitwise OR of GLUT display mode bit masks. This API specifies a display mode (such as RGBA or color-index, or single or double-buffered) for windows.

- **glutSwapBuffers() :**

```
void glutSwapBuffers(void);
```

- 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. The contents of the back buffer then become undefined. The update typically takes place during the vertical retrace of the monitor, rather than immediately after glutSwapBuffers is called.
- An implicit glFlush is done by glutSwapBuffers before it returns. Subsequent OpenGL commands can be issued immediately after calling glutSwapBuffers, but are not executed until the buffer exchange is completed.
- If the layer in use is not double buffered, glutSwapBuffers has no effect.

- **glutCreateWindow() :**

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

The parameter name specifies any name for window and is enclosed in double quotes. This opens a window with the set characteristics like display mode, width, height, and so on. The string name will appear in the title bar of the window system. The value returned is a unique integer identifier for the window. This identifier can be used for controlling and rendering to multiple windows from the same application.

- **glutDisplayFunc() :**

`void glutDisplayFunc(void (*func)(void))`

Specifies the new display callback function. The API specifies the function that's called whenever the contents of the window need to be redrawn. All the routines need to be redrawn the scene are put in display callback function.

- **glVertex3f**

`void glVertex3f(GLfloat x, GLfloat y, GLfloat z);`

- x Specifies the x-coordinate of a vertex.
- y Specifies the y-coordinate of a vertex.
- z Specifies the z-coordinate of a vertex.

The glVertex function commands are used within glBegin/glEnd pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only x and y are specified, z defaults to 0.0 and w defaults to 1.0. When x, y, and z are specified, w defaults to 1.0.

- **glColor3f**

`void glColor3f(GLfloat red, GLfloat green, GLfloat blue);`

PARAMETERS:

- Red: The new red value for the current color.
- Green: The new green value for the current color.
- Blue: The new blue value for the current color.

Sets the current color.

- **glRotate():**

`void glRotate(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);`

PARAMETERS:

-
- angle: The angle of rotation, in degrees.
 - x: The x coordinate of a vector.
 - y: The y coordinate of a vector.
 - z: The z coordinate of a vector.

The `glRotated` and `glRotatef` functions multiply the current matrix by a rotation matrix.

- **`glGenTextures()`:**

```
void glGenTexture (GLsizei n, GLuint textures);
```

PARAMETERS:

- n: Specifies the number of texture names to be generated.
- Textures: Specifies an array in which the generated texture names are stored.

`glGenTextures` returns 'n' texture names in textures. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenTextures`.

- **`glBindTexture()`:**

```
void glBindTexture (GLenum target, GLuint texture);
```

PARAMETERS:

- target: Specifies the target to which the texture is bound. Must be one of `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_1D_ARRAY`, `GL_TEXTURE_2D_ARRAY`, `GL_TEXTURE_RECTANGLE`, `GL_TEXTURE_CUBE_MAP`, `GL_TEXTURE_CUBE_MAP_ARRAY`, `GL_TEXTURE_BUFFER`, `GL_TEXTURE_2D_MULTISAMPLE` or `GL_TEXTURE_2D_MULTISAMPLE_ARRAY`.
- texture: Specifies the name of a texture.

- **`glutInit()`:**

```
glutInit(int *argcp, char **argv);
```

PARAMETERS:

-
- `argc` : A pointer to the program's unmodified `argc` variable from main. Upon return, the value pointed to by `argc` will be updated, because `glutInit` extracts any command line options intended for the GLUT library.
 - `argv` : The program's unmodified `argv` variable from main. Like `argc`, the data for `argv` will be updated because `glutInit` extracts any command line options understood by the GLUT library.
 - `glutInit(&argc,argv);`
 - `glutInit` is used to initialize the GLUT library.

- **`glutMainLoop ()`:**

```
void glutMainLoop (void);  
  
glutMainLoop();
```

`glutMainLoop` enters the GLUT event processing loop.

- **`glLightfv()`:**

```
void glLightfv(GLenum light, GLenum pname, GLfloat *params);
```

The `glLightfv` function returns light source parameter values.

PARAMETERS:

- **Light:** The identifier of a light. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form `GL_LIGHTi` where `i` is a value: 0 to `GL_MAX_LIGHTS`
 - **Pname:** A light source parameter for light. The following symbolic names are accepted:
 - **GL_DIFFUSE:** The `params` parameter contains four integer or floating-point values that specify the diffuse RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default diffuse intensity of light zero is (1.0, 1.0, 1.0, 1.0).
 - **GL_SPECULAR:** The `params` parameter contains four integer or floating-point values that specify the specular RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular intensity
-

is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default specular intensity of light zero is (1.0, 1.0, 1.0, 1.0).

- **GL_AMBIENT:** The params contains four integer or floating-point values that specify the ambient RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial ambient light intensity is (0.0, 0.0, 0.0, 1.0).

```
glLightfv(GL_LIGHT1, GL_POSITION, pos);
```

- **glEnable():**

```
void glEnable(GLenum cap);
```

Ex: glEnable(GL_CULL_FACE);

PARAMETERS:

- cap: A symbolic constant indicating an OpenGL capability.
- The glEnable enables OpenGL capabilities.

- **glDisable():**

```
void glDisable (GLenum cap);
```

Ex: glDisable (GL_CULL_FACE);

PARAMETERS:

- cap: Specifies a symbolic constant indicating a GL capability.
- The glDisable disables OpenGL capabilities.

- **GLUquadric* gluNewQuadric(void);**

- 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. The contents of the back buffer then become undefined.
- The update typically takes place during the vertical retrace of the monitor, rather than immediately after glutSwapBuffers is called.

- An implicit `glFlush` is done by `glutSwapBuffers` before it returns. Subsequent OpenGL commands can be issued immediately after calling `glutSwapBuffers`, but are not executed until the buffer exchange is completed.
- If the layer in use is not double buffered, `glutSwapBuffers` has no effect.

- **`gluQuadricNormals()`:**

```
void gluQuadricNormals (GLUquadric* quad, GLenum normal);
```

PARAMETERS:

- Quad: Specifies the quadrics object (created with `gluNewQuadric`).
- Normal: Specifies the quadrics object (created with `gluNewQuadric`).
- `gluQuadricNormals` specifies what kind of normals are desired for quadrics rendered with quad. The legal values are as follows:
- `GLU_NONE`: No normals are generated.
- `GLU_FLAT`: One normal is generated for every facet of a quadric.
- `GLU_SMOOTH`: One normal is generated for every vertex of a quadric. This is the initial value.

- **`gluQuadricTexture()`:**

```
void gluQuadricTexture (GLUquadric* quad, GLboolean texture);
```

PARAMETERS:

- Quad: Specifies the quadrics object (created with `gluNewQuadric`).
- Texture: Specifies a flag indicating if texture coordinates should be generated.
- `gluQuadricTexture` specifies if texture coordinates should be generated for quadrics rendered with quad. If the value of texture is `GLU_TRUE`, then texture coordinates are generated, and if texture is `GLU_FALSE`, they are not. The initial value is `GLU_FALSE`.
- The manner in which texture coordinates are generated depends upon the specific quadric rendered.

- **gluLookAt():**

```
void gluLookAt (GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,  
                GLdouble centerX, GLdouble centerY, GLdouble centerZ,  
                GLdouble upX, GLdouble upY, GLdouble upZ);
```

PARAMETERS:

- eyeX, eyeY, eyeZ : Specifies the position of the eye point.
- centerX, centerY, centerZ : Specifies the position of the reference point.
- upX, upY, upZ: Specifies the direction of the up vector.
- gluLookAt creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.

- **gluSphere():**

```
void gluSphere(GLUquadric* quad, GLdouble radius, GLint slices, GLint stacks);
```

PARAMETERS:

- Quad: Specifies the quadrics object (created with gluNewQuadric).
- Radius: Specifies the radius of the sphere.
- Slices: Specifies the number of subdivisions around the z axis (similar to lines of longitude).
- Stacks: Specifies the number of subdivisions along the z axis (similar to lines of latitude).
- gluSphere draws a sphere of the given radius centered around the origin. The sphere is subdivided around the z axis into slices and along the z axis into stacks (similar to lines of longitude and latitude).

- **glViewport():**

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

PARAMETERS:

- x,y: Specify the lower left corner of the viewport rectangle, in pixels. The initial value is (0,0).
- width, height: Specify the width and height of the viewport. When a GL context is first attached to a window, width and height are set to the dimensions of that window.

- **glFrustum():**

```
void glFrustum(GL double left, GLdouble right, GLdouble bottom, GLdouble top,  
               GLdouble nearVal, GLdouble farVal);
```

PARAMETERS:

- left, right: Specify the coordinates for the left and right vertical clipping planes.
- bottom, top: Specify the coordinates for the bottom and top horizontal clipping planes.
- nearVal, farVal : Specify the distances to the near and far depth clipping planes. Both distances must be positive.
- glFrustum describes a perspective matrix that produces a perspective projection.

- **glutTimerFunc():**

void glutTimerFunc(unsigned int msec, void (*func)(int value), value);

- glutTimerFunc registers the timer callback func to be triggered in at least msec milliseconds. The value parameter to the timer callback will be the value of the value parameter to glutTimerFunc. Multiple timer callbacks at same or differing times may be registered simultaneously.
- The number of milliseconds is a lower bound on the time before the callback is generated. GLUT attempts to deliver the timer callback as soon as possible after the expiration of the callback's time interval.
- There is no support for canceling a registered callback. Instead, ignore a callback based on its value parameter when it is triggered.

- **glCreateMenu():**

int glutCreateMenu (void (*func)(int value));

PARAMETERS:

- func : The callback function for the menu that is called when a menu entry from the menu is
 - selected. The value passed to the callback is determined by the value for the selected
 - menu entry.
- glutCreateMenu creates a new pop-up menu and returns a unique small integer identifier. The range of allocated identifiers starts at one. The menu identifier range is separate from the window identifier range. Implicitly, the current menu is set to the newly created menu. This menu identifier can be used when calling glutSetMenu.

-
- When the menu callback is called because a menu entry is selected for the menu, the current menu will be implicitly set to the menu with the selected entry before the callback is made.

- **glutAddMenuEntry():**

```
void glutAddMenuEntry (char *name, int value);
```

PARAMETERS:

- name: ASCII character string to display in the menu entry.
- value: Value to return to the menu's callback function if the menu entry is selected.

glutAddMenuEntry adds a menu entry to the bottom of the current menu. The string name will be displayed for the newly added menu entry. If the menu entry is selected by the user, the menu's callback will be called passing value as the callback's parameter.

- **glutAddSubMenu():**

```
void glutAddSubMenu (char *name, int menu);
```

PARAMETERS:

- Name: ASCII character string to display in the menu item from which to cascade the sub-menu.
- Menu: Identifier of the menu to cascade from this sub-menu menu item.

glutAddSubMenu adds a sub-menu trigger to the bottom of the current menu. The string name will be displayed for the newly added sub-menu trigger. If the sub-menu trigger is entered, the sub-menu numbered menu will be cascaded, allowing sub-menu menu items to be selected.

- **glutAttachMenu ():**

```
void glutAttachMenu (int button);
```

PARAMETERS:

Button: The button to attach a menu or detach a menu.

`glutAttachMenu` attaches a mouse button for the current window to the identifier of the current menu. By attaching a menu identifier to a button, the named menu will be popped up when the user presses the specified button. `button` should be one of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, and `GLUT_RIGHT_BUTTON`. Note that the menu is attached to the button by identifier, not by reference.

- **`glutReshapeFunc()`:**

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

PARAMETERS:

Func: The new reshape callback function.

- `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. The width and height parameters of the callback specify the new window size in pixels. Before the callback, the current window is set to the window that has been reshaped.
- If a reshape callback is not registered for a window or `NULL` is passed to `glutReshapeFunc` (to deregister a previously registered callback), the default reshape callback is used. This default callback will simply call `glViewport(0,0, width,height)` on the normal plane (and on the overlay if one exists).
- If an overlay is established for the window, a single reshape callback is generated. It is the callback's responsibility to update both the normal plane and overlay for the window (changing the layer in use as necessary).
- When a top-level window is reshaped, subwindows are not reshaped. It is up to the GLUT program to manage the size and positions of subwindows within a top-level window. Still, reshape callbacks will be triggered for subwindows when their size is changed using `glutReshapeWindow`.

- **`glutKeyboardFunc()`:**

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

PARAMETERS:

Func: The new keyboard callback function.

-
- `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 state of modifier keys such as Shift cannot be determined directly; their only effect will be on the returned ASCII data.
 - The `x` and `y` callback parameters indicate the mouse location in window relative coordinates when the key was pressed. When a new window is created, no keyboard callback is initially registered, and ASCII key strokes in the window are ignored. Passing `NULL` to `glutKeyboardFunc` disables the generation of keyboard callbacks.
- **`glutPostRedisplay ()`:**
 - `void glutPostRedisplay (void);`
 - 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.
 - Multiple calls to `glutPostRedisplay` before the next display callback opportunity generates only a single redisplay callback. `glutPostRedisplay` may be called within a window's display or overlay display callback to re-mark that window for redisplay.
 - Logically, normal plane damage notification for a window is treated as a `glutPostRedisplay` on the damaged window.

2.4 Source Code

The header files are:

```
#include <windows.h>
#include <GL/glut.h>
#include <math.h>
#include "tgaload.c"
```

The code for creating the orbits for the planets:

```
void generateOrbit(void)
{
    for (int i = 0; i <= 360; i++)
        //Mercury
```

```
x1[i][0] = sin(i*3.1416/180) *2.4;
x1[i][1] = cos(i*3.1416/180) *2.4;
//Venus
x2[i][0] = sin(i*3.1416/180) *3.2;
x2[i][1] = cos(i*3.1416/180) *3.2;
//Earth
x3[i][0] = sin(i*3.1416/180) *4.2;
x3[i][1] = cos(i*3.1416/180) *4.2;
//Moon
x10[i][0] = sin(i*3.1416/180) *0.6;
x10[i][1] = cos(i*3.1416/180) *0.6;
//Mars
x4[i][0] = sin(i*3.1416/180) *5.5;
x4[i][1] = cos(i*3.1416/180) *5.5;
//Jupiter
x5[i][0] = sin(i*3.1416/180) *7;
x5[i][1] = cos(i*3.1416/180) *7;
//Saturn
x6[i][0] = sin(i*3.1416/180) *8.8;
x6[i][1] = cos(i*3.1416/180) *8.8;
//Uranus
x7[i][0] = sin(i*3.1416/180) *10.5;
x7[i][1] = cos(i*3.1416/180) *10.5;
//Neptune
x8[i][0] = sin(i*3.1416/180) *12;
x8[i][1] = cos(i*3.1416/180) *12;
//Pluto
```

```
x9[i][0] = sin(i*3.1416/180) *13.2;
x9[i][1] = cos(i*3.1416/180) *13.2;
}
}
```

The Reshape Function:

```
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glFrustum (-2.0, 2.0, -2.0, 2.0, 1.5, 30.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}
```

The code for drawing and rotating the planets:

```
void planets ()
{ //SUN
    glPushMatrix ();
    gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
    glTranslatef (Xvalue, 0.0, Yvalue);
    glRotatef (Angle/25, 0.0, 0.0, 1.0);
    glBindTexture (GL_TEXTURE_2D, texture_id[0] );
    gluSphere(sphere,2,100,100);
    glPopMatrix ();
//MERCURY
    glPushMatrix ();
    gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
```

```
if (MoveMerc==360)
    MoveMerc = 0;

glTranslatef(x1[MoveMerc][1], x1[MoveMerc][0], 0.0);

glRotatef (Angle, 0.0, 0.0, 1.0);

glBindTexture (GL_TEXTURE_2D, texture_id[1] );

gluSphere (sphere, 0.25, 100, 100);

glPopMatrix ();

//VENUS

glPushMatrix ();

gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

if (MoveVenus == 720)
    MoveVenus = 0;

glTranslatef(x2[MoveVenus/2][1], x2[MoveVenus/2][0], 0.0);

glRotatef (Angle, 0.0, 0.0, -1.0);

glBindTexture (GL_TEXTURE_2D, texture_id[2] );

gluSphere (sphere, 0.28, 100, 100);

glPopMatrix ();

//EARTH

glPushMatrix ();

gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

if (MoveEarth==1080)
    MoveEarth = 0;

glTranslatef(x3[MoveEarth/3][1], x3[MoveEarth/3][0], 0.0);

glRotatef (Angle*2, 0.1, 0.3, 0.7);

glBindTexture (GL_TEXTURE_2D, texture_id[3]);

gluSphere (sphere, 0.3, 100, 100);

//MOON
```

```
    if (MoveMoon == 360)
        MoveMoon = 0;

    glTranslatef(x10[MoveMoon][1], x10[MoveMoon][0], 0.0); //glTranslated(-0.3,-0.3,0.3);
    glBindTexture (GL_TEXTURE_2D, moon_id [0]);
    gluSphere (sphere, 0.1, 100, 100);
    glPopMatrix();

//MARS

    glPushMatrix ();
    gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
    if (MoveMars == 1440)
        MoveMars = 0;

    glTranslatef(x4[MoveMars/4][1], x4[MoveMars/4][0], 0.0);
    glRotatef (Angle, 0.0, 0.0, 1.0);
    glBindTexture (GL_TEXTURE_2D, texture_id[4] );
    gluSphere (sphere, 0.28, 100, 100);
    glPopMatrix ();

//JUPITER

    glPushMatrix ();
    gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
    if (MoveJup == 1800)
        MoveJup = 0;

    glTranslatef(x5[MoveJup/5][1], x5[MoveJup/5][0], 0.0);
    glRotatef (Angle, 0.0, 0.0, 0.1);
    glBindTexture (GL_TEXTURE_2D, texture_id[5] );
    gluSphere (sphere, 0.7, 100, 100);
    glPopMatrix ();

//SATURN
```

```
glPushMatrix ();
gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
if (MoveSaturn==2160)
    MoveSaturn = 0;
glTranslatef(x6[MoveSaturn/6][1], x6[MoveSaturn/6][0], 0.0);
glRotatef (Angle, 0.0, 0.0, 0.1);
glBindTexture (GL_TEXTURE_2D, texture_id[6] );
gluSphere (sphere, 0.50, 100, 100);
int i = 0;
glBindTexture(GL_TEXTURE_2D, texture_id[10]);
glBegin(GL_QUAD_STRIP);// Saturn Rings
for (i=0; i <= 360; i++)
{
    glVertex3f(sin(i*3.1416/180) *0.58, cos(i*3.1416/180) *0.58, 0);
    glVertex3f(sin(i*3.1416/180) *0.61, cos(i*3.1416/180) *0.61, 0);
}
for (i = 0; i <= 360; i++)
{
    glVertex3f(sin(i*3.1416/180)*0.65, cos(i*3.1416/180)*0.65, 0 );
    glVertex3f(sin(i*3.1416/180)*0.72, cos(i*3.1416/180)*0.72, 0 );
}
glEnd();
glRotatef(Angle, 0.5, 0.2, 1.5);
glPopMatrix();
//URANUS
glPushMatrix();
gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
```

```
if (MoveUr == 2520)
    MoveUr = 0;

glTranslatef(x7[(MoveUr/7)][1], x7[(MoveUr/7)][0], 0.0);
glRotatef(Angle, 0.0, 0.0, -0.1);
glBindTexture ( GL_TEXTURE_2D, texture_id[7] );
gluSphere(sphere, 0.4, 100, 100);
glBindTexture(GL_TEXTURE_2D, texture_id[11]);
glBegin(GL_QUAD_STRIP);
for(int i=0; i <= 360; i++)
{
    glVertex3f(sin(i*3.1416/180)*0.58, cos(i*3.1416/180)*0.58, 0 );
    glVertex3f(sin(i*3.1416/180)*0.61, cos(i*3.1416/180)*0.61, 0 );
}
glEnd();
glPopMatrix();

//NEPTUNE
glPushMatrix();
gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
if (MoveNep == 2880)
    MoveNep = 0;

glTranslatef(x8[MoveNep/8][1], x8[MoveNep/8][0], 0.0);
glRotatef(Angle, 0.0, 0.0, 0.1);
glBindTexture ( GL_TEXTURE_2D, texture_id[8] );
gluSphere(sphere, 0.33, 100, 100);
glPopMatrix();

//PLUTO
glPushMatrix();
```

```
gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);  
if (MovePlu == 3240)  
    MovePlu = 0;  
glTranslatef(x9[MovePlu/9][1], x9[MovePlu/9][0], 0.0);  
glRotatef(Angle, 0.0, 0.0, 0.1);  
glBindTexture ( GL_TEXTURE_2D, texture_id[9] );  
gluSphere(sphere, 0.2, 100, 100);  
glPopMatrix();  
}
```

Code for displaying the 3D Model:

```
void myDisplay(void)  
{ glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
  glColor3f (1.0, 1.0, 1.0);  
  glLoadIdentity();  
  if (value != 0 && value != 15)  
  { glPushMatrix();  
    gluLookAt (worldX, worldY, worldZ, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);  
    glTranslated(Xvalue, 0.0, Yvalue);  
    glRotatef(Angle, 0.0, 0.0, 1.0);  
    glDisable(GL_LIGHTING);  
    glBindTexture(GL_TEXTURE_2D, texture_id[value]);  
    gluSphere(sphere, 0.2, 100, 100);  
    glBegin(GL_QUAD_STRIP);  
    if (value == 6 || value == 7)  
    {  
      if (value == 6)  
        glBindTexture(GL_TEXTURE_2D, texture_id[10]);  
    }  
  }  
}
```

```
else glBindTexture(GL_TEXTURE_2D, texture_id[11]);
for(int i=0; i <= 360; i++)
{
    glVertex3f(sin(i*3.1416/180)*4.38, cos(i*3.1416/180)*4.38, 0 );
    glVertex3f(sin(i*3.1416/180)*5.11, cos(i*3.1416/180)*5.11, 0 );
}
}
if (value == 6)
{
    for (int i = 0; i <= 360; i++)
    {
        glVertex3f(sin(i*3.1416/180)*5.5, cos(i*3.1416/180)*5.5, 0 );
        glVertex3f(sin(i*3.1416/180)*5.92, cos(i*3.1416/180)*5.92, 0 );
    }
}
glEnd();
glEnable(GL_LIGHTING);
glPopMatrix();
}
else if (value == 15)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    Sun();Mercury();Venus();EarthAndMoon();Mars();
    Jupiter();Saturn();Uranus();Neptune();Pluto();
}
glutSwapBuffers();
}
```

The code for creating a menu list:

```
void menu (int num)
{
    if(num == 0)
        exit(0);
    else value = num;
    glutPostRedisplay();
}

void CreateMenuItem(void)
{
    submenu_id = glutCreateMenu(menu);
    glutAddMenuEntry("Mercury",1);
    glutAddMenuEntry("Venus",2);
    glutAddMenuEntry("Earth",3);
    glutAddMenuEntry("Mars",4);
    glutAddMenuEntry("Jupiter",5);
    glutAddMenuEntry("Saturn",6);
    glutAddMenuEntry("Uranus",7);
    glutAddMenuEntry("Neptune",8);
    glutAddMenuEntry("Pluto",9);
    menu_id = glutCreateMenu(menu);
    glutAddSubMenu("Display Planet",submenu_id);
    glutAddMenuEntry("Quit", 0);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
}
```

The Keyboard Function:

```
void keys(unsigned char key, int x, int y)
{
    switch (key)
    {
        case 'q': exit(0);          break;
    }
}
```

```
    case 'y': worldY -= 1.0f;    break;
    case 'Y': worldY += 1.0f;    break;
    case 'z': worldZ -= 1.0f;    break;
    case 'Z': worldZ += 1.0f;    break;
    case 'x': worldX -= 1.0f;    break;
    case 'X': worldX += 1.0f;    break;
    case ' ': if (!paused) Sleep(2000); break;
    case '+': revspeed=5.5;      break;
    case '-': revspeed-=0.5;     break;
    case 'r': value = 15; glutPostRedisplay(); break;
    default: break;

        }glutPostRedisplay();
}
```

For loading the texture from TGA format image:

```
tgaLoad("C:/Chaithra/OpenGLProjects/Solar_System/bin/Debug/Sun.tga",&temp_image,
TGA_FREE | TGA_LOW_QUALITY);
```

CHAPTER - 03

RESULT ANALYSIS

3.1 Snapshots:

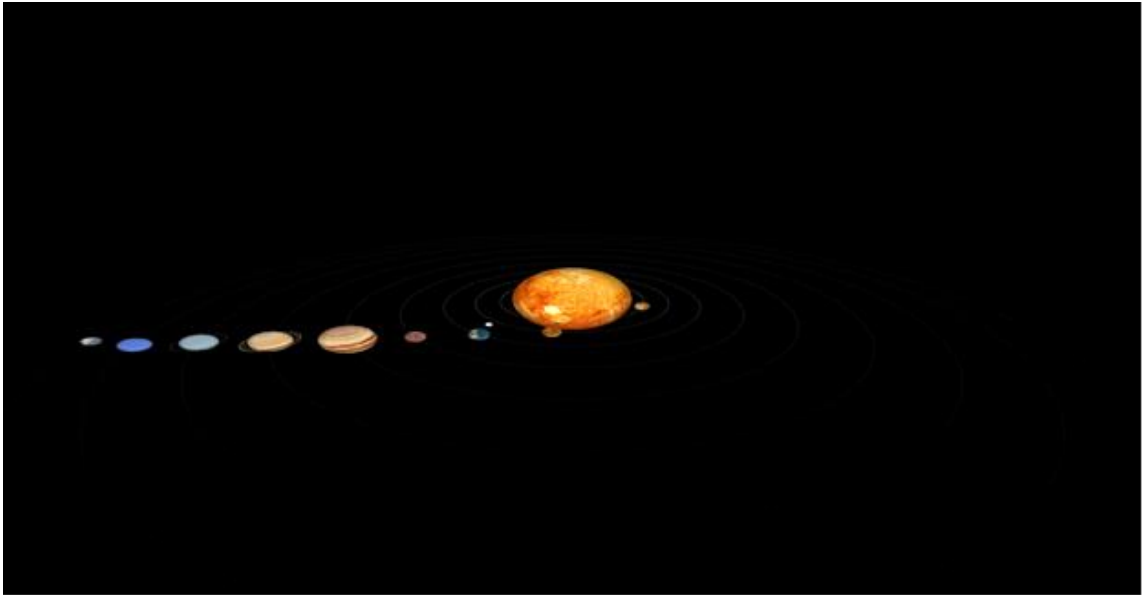


Fig 3.1 Initial View of Model

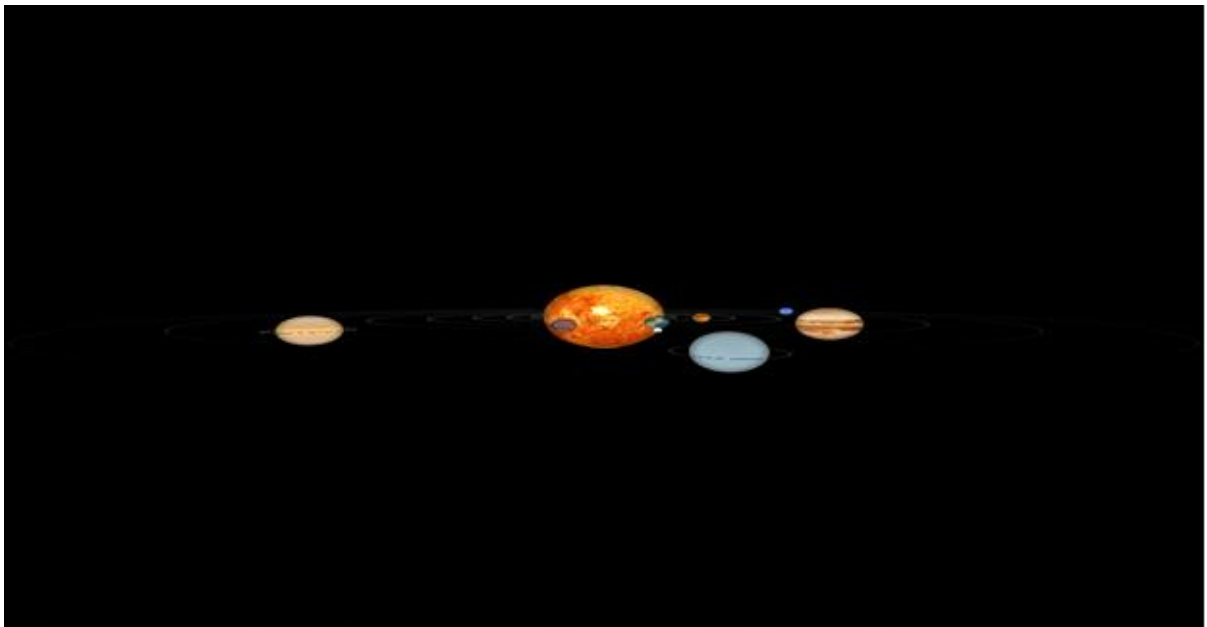
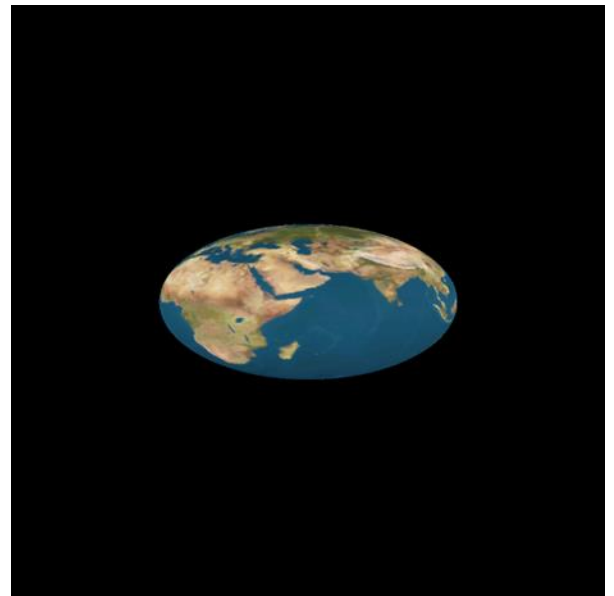


Fig 3.2 Cross sectional view



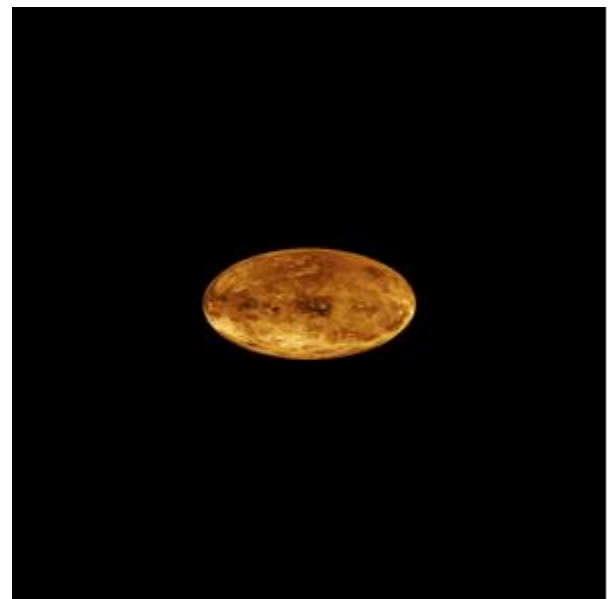
Mercury



Earth



Venus



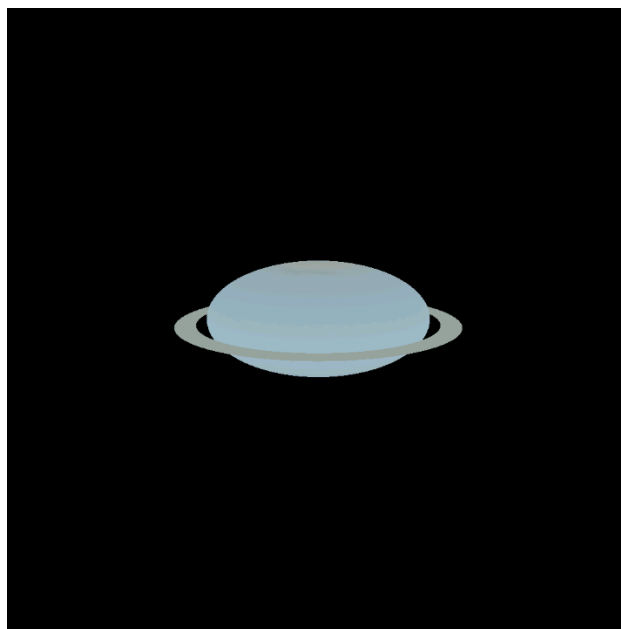
Mars



Jupiter



Saturn



Uranus

3.2 Discussion

Based on the analysis of results that has been obtained, we can conclude that:

- ❖ Animation circulation of planets around the sun interface design is the design of the visual and interpretive models designed.
- ❖ Application of learning visualization of the solar system was designed using API OpenGL that can facilitate computer users to know the objects of the solar system and the solar system information more easily
- ❖ The development of imagination and the ability to process a multimedia device that is imaginary with supporting software utilization growing.
- ❖ We also hope the necessity of adding multimedia animation, and texture for better and have a variety of materials, so that users can more enjoyed.

CHAPTER – 04

CONCLUSION AND FUTURE WORK

4.1 Conclusion

- ❖ Our project aims at displaying the 3D model of the solar system in OpenGL. Using built in functions provided by graphics library and integrating it with the C implementation of list it was possible to visually represent the objects in a 3D plane.
- ❖ The described project demonstrates the power of Viewing which is implemented using different modes of viewing. The lighting and texture functions of OpenGL library add effect to the objects in animation.
- ❖ The aim in developing this program was to design a simple program using OpenGL application software by applying the skills we learnt in class, and in doing so, to understand the algorithms and the techniques underlying interactive graphics better.
- ❖ The designed program will incorporate all the basic properties that a simple program must possess.
- ❖ The program is user friendly as the only skill required in executing this program is the knowledge of graphics.

4.2 Future Enhancement

- We can try to simulate the solar system with precise orbits and revolutions.
- More functionalities, like displaying information about the planets, can be added.
- Orbits can be made elliptical, as is the original solar system instead of circular.
- Other celestial objects like moons, stars, asteroids and other such objects can also be included.

CHAPTER – 05

REFERENCES

Books:

- The Red Book –OpenGL Programming Guide,6th edition.
- Interactive Computer Graphics-A Top - Down Approach Using OpenGL, Edward Angel, Pearson-5th edition.

Websites:

- <http://planetpixelemporium.com/>
- <http://www.glprogramming.com/red/>
- <https://open.gl/textures>
- <https://www.khronos.org/registry/OpenGL-Refpages/gl4/>