1.2 PROJECT DESCRIPTION

Worm is two stage game in which food pops up randomly on the screen which is eaten by the worm. Task is to direct the worm using W, S, A, D keys for up, down, left and right directions respectively. The worm moves and eats the food as soon as the head of the worm touches the food. With each food the worm eats, it grows bigger by one matrix. The game ups by one level as soon as the player reaches a score of 10. Now the worm is unable to pass through walls as a protective wall emerges at the boundary.
OpenGL is the standard library used for implementing the game in C++ programming language.

CHAPTER 2

OPENGL

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment. Any visual computing application requiring maximum performance-from 3D animation to CAD to visual simulation-can exploit high-quality, high-performance OpenGL capabilities. These capabilities allow developers in diverse markets such as broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2D and 3D graphics.

OpenGL (**O**pen **G**raphics **L**ibrary) is a standard specification defining a cross-language, cross platform API for writing applications that produce 2D and3D computer Graphics interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc (SGI) in 1992 and is widely used in CAD, virtual Reality, scientific visualization, information visualization, and flight simulation. It is also used in video games, where it competes with Direct3D on Microsoft Windows platforms. OpenGL is managed by the non-profit technology consortium, the Khronos Group.

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. This contrast 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.

Advantages

- **Industry standard: -** An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard.
- **Stable: -** OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes. Backward compatibility requirements ensure that existing applications do not become obsolete.
- **Reliable and portable:** - All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or window
- **Evolving:** - Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. In this way, innovations appear in the API in a timely fashion, letting application developers and hardware vendors incorporate new features into their normal product release cycles.
- **Scalable:** - OpenGL API-based applications can run on systems ranging from consumer electronics to PCs, workstations, and supercomputers. As a result, applications can scale to any class of machine that the developer chooses to target.

- **Easy to use:** - OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features.
- **Well-documented:** - Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain.
- **Evolving:** - Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism.
- **Scalable:** - OpenGL API-based applications can run on systems ranging from consumer electronics to PCs, workstations, and supercomputers. As a result, applications can scale to any class of machine that the developer chooses to target.
- **Easy to use:** - OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware.
- **Well-documented:** - Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain.

Applications of OpenGL

The development of computer graphics has been driven both by the user community and by advances in hardware and software. The application of computers graphics are many and varied; we can however, divide them in to four major areas: -

- Display of Information
- Design
- Simulation and animation
- User interfaces

2.1 GLUT

GLUT is a complete API written by Mark Kilgard which lets you create windows and handle the messages. It exists for several platforms, that means that a program which uses GLUT can be compiled on many platforms without (or at least with very few) changes in the code.

2.2 Working of OpenGL

OpenGL bases on the state variables. There are many values, for example the color, that remain after being specified. That means, you can specify a color once and draw several polygons, lines or whatever with this color then. There are no classes like in DirectX. However, it is logically structured. Before we come to the commands themselves, here is another thing.

To be hardware independent, OpenGL provides its own data types. They all begin with "GL". For example GLfloat, GLuint and so on. There are also many symbolic constants, they all begin with "GL_", like GL_POINTS, GL_POLYGON. Finally the commands have the prefix "gl" like glVertex3f(). There is a utility library called GLU, here the prefixes are "GLU_" and "glu". GLUT commands begin with "glut", it is the same for every library.

A very important thing is to know, that there are two important matrices, which affect the transformation from the 3d-world to the 2d-screen: The projection matrix and the modelview matrix. The projection matrix contains information, how a vertex – let's say a "point" in space – shall be mapped to the screen. This contains, whether the projection shall be isometric or from a perspective, how wide the field of view is and so on. Into the other matrix you put information, how the objects are moved, where the viewer is and so on.

2.3 Usage of GLUT

GLUT provides some routines for the initialization and creating the window (or full screen mode, if you want to). Those functions are called first in a GLUT application:

In your first line you always write glutInit (&argc, argv). After this, you must tell GLUT, which display mode you want – single or double buffering, color index mode or RGB and so on. This is done by calling glutInitDisplayMode(). The symbolic constants are connected by a logical OR, so you could use glutInitDisplayMode (GLUT_RGB | GLUT_SINGLE). In later tutorials we will use some more constants here.

After the initialization you call glCreateWindow() with the window name as parameter.

Then you can (and should) pass some methods for certain events. The most important ones are "reshape" and "display". In reshape you need to (re)define the field of view and specify a new area in the window, where OpenGL is allowed to draw to.

Display should clear the so called color buffer – let's say this is the sheet of paper – and draw our objects.

You pass the methods by glut*Func(), for example glutDisplayFunc(). At the end of the main function you call glutMainLoop(). This function doesn't return, but calls the several functions passed by glut*Func.

2.4 OPENGL libraries

- The OpenGL Utility Library (GLU), which contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing tessellation, and rendering surfaces.
- The OpenGL Extension to the X Window System (GLX) provides a means of creating an OpenGL context and associating it with an X Window System window.
- The OpenGL Programming Guide Auxiliary Library provides routines for initializing and opening windows, handling X events and dealing with common complex shapes, such as cubes, spheres, and cylinders.

2.5 OPENGL PIPELINE ARCHITECTURE

The OpenGL architecture is structured as a state-based pipeline. Fig 2.5.1 is a simplified diagram of this pipeline. Commands enter the pipeline from the left.

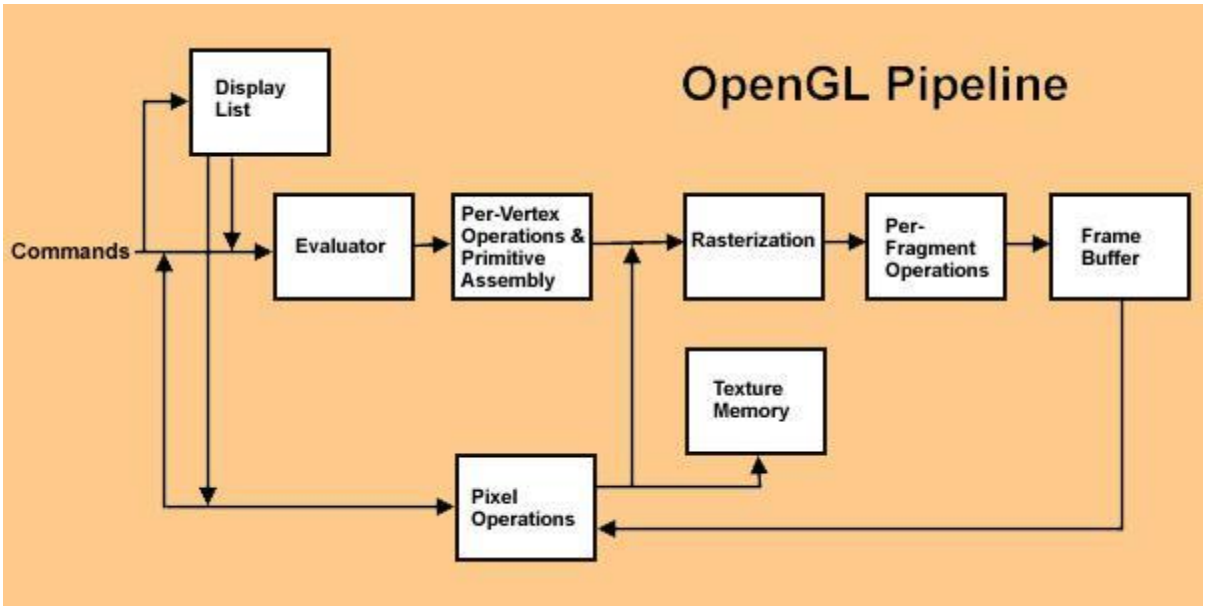

Fig2.5.1: Pipeline architecture of OpenGL

Commands may either be accumulated in display lists, or processed immediately through the pipeline. Display lists allow for greater optimization and command reuse, but not all commands can be put in display lists.

- The first stage in the pipeline is the evaluator. This stage effectively takes any polynomial evaluator commands and evaluates them into their corresponding vertex and attributes commands.

- The second stage is the per-vertex operations, including transformations, lighting, primitive assembly, clipping, projection, and viewport mapping.

- The third stage is rasterization. This stage produces fragments, which are series of framebuffer addresses and values, from the viewport-mapped primitives as well as bitmaps and pixel rectangles.

- The fourth stage is the per-fragment operations. Before fragments go to the framebuffer, they may be subjected to a series of conditional tests and modifications, such as blending or z-buffering.

Parts of the framebuffer may be fed back into the pipeline as pixel rectangles. Texture memory may be used in the rasterization process when texture mapping is enabled.

2.5.1 Vertices and Primitives

Most objects (with the exception of pixel rectangles and bitmaps), use Begin/End primitives. Each Begin/End primitive contains a series of vertex data, and may optionally contain normals, texture coordinates, colors, edge flags, and material properties.

Commands not associated with primitives are not allowed within Begin/End blocks. This allows increased optimization of primitive processing. Vertices may be specified in 2D, 3D, or 4D. 2D coordinates are promoted to 3D by assigning a Z value of zero. 4D homogeneous coordinates are reduced to 3D by dividing *x*, *y*, and *z* by the *w* coordinate (if non-zero).

Optional vertex attributes are picked up from state if not specified per-vertex. The normal is a 3D vector perpendicular to the surface being described, and is used during these vertex and attribute commands, as well as many other OpenGL commands, accept arguments either explicitly or through pointers. They also accept a variety of data types as arguments, such as ints, floats, doubles, bytes, unsigned ints and bytes, etc.

The model-view matrix, texture matrix, and projection matrix each affect the vertex and its attributes, and may be easily manipulated via transformations such as rotation, scaling, and translation.

Lighting parameters, such as material properties, light source properties, and lighting model parameters affect lighting on a per-vertex basis.

2.5.2 Clipping and Projection

Once a primitive has been assembled, it is subject to arbitrary clipping via user definable clip planes. An OpenGL implementation must provide at least six, and they may be turned on and off independently by the user.

Points are either clipped in our out, depending on whether they fall inside or outside the half-space defined by the clip planes. Lines and polygons, however, may either be 100% clipped, 100% unclipped, or they may fall partially within the clip space. In this latter case, new vertices are automatically place on the clip boundary between pre-existing vertices. Vertex attributes are interpolated.

After clipping, vertices are transformed by the projection matrix (either perspective or orthogonal), and then clipped to the frustum (view space), following the same process as above. Finally, the vertices are mapped to the viewport (screen space).

2.5.3 Rasterization

Rasterization converts the above viewport-mapped primitives into fragments. Fragments consist of pixel location in framebuffer, color, texture coordinates, and depth (z buffer). Depending on the shading mode, vertex attributes are either interpolated across the primitive to all fragments (smooth shading), or all fragments are assigned the same values based on one vertex's attributes (flat shading).

Rasterization is affected by the point and line widths, the line stipple sequence, and the polygon stipple pattern. Antialiasing may be enabled or disabled for each primitive type. If enabled, the alpha color value (if in RGBA mode) or color index (if in CI mode) are modified to reflect sub-pixel coverage.

Pixel rectangles and bitmaps are also rasterized, but they bypass the lighting and geometrical transformations. They are groups of values heading for the framebuffer. They can be scaled, offset, and mapped via lookup tables. The rasterization process produces a rectangle of fragments at a location controlled by the current raster position state variable. The size may be affected by the pixel zoom setting.

Bitmaps are similar to pixel rectangles, but the data is binary, only producing fragments when on. This is useful for drawing text in 3D space as part of a scene.

2.5.4 Framebuffer

Fragments produced by rasterization go to the framebuffer where they may be displayed. The framebuffer is a rectangular array of *n* bitplanes. The bitplanes are organized into several logical buffers -- Color, Depth, Stencil, and Accumulation.

The color buffer contains the fragment's color info.

The depth buffer contains the fragment's depth info, typically used for z-buffering hidden surface removal.

The stencil buffer can be associated with fragments that pass the conditional tests described below and make it into the frame buffer. It can be useful for multiple-pass algorithms.

The accumulation buffer is also used for multiple-pass algorithms. It can average the values stored in the color buffer. Full-screen antialiasing can be achieved by jittering the viewpoint. Depth of Field can be achieved by jittering the view angle. Motion blur can be achieved by stepping the scene in time.

Stereo and double-buffering may be supported under OpenGL, depending on the implementation. These would further divide the framebuffer into up to 4 sections -- front and back buffer, left and right. Other auxiliary buffers may be available on some implementations. Any buffers may be individually enabled or disabled for writing. The depths and availabilities of buffers may vary, but must meet the minimum requirement of OpenGL.

CHAPTER 3

SYSTEM SPECIFICATION

3.1 SOFTWARE REQUIREMENTS

Minimum Software Requirements	
Operating System	The Visual Studio 2005 can be installed on the following platforms: Windows XP
Other	Windows comes with OpenGL, and Visual Studio comes with the OpenGL libraries, but neither of them comes with GLUT .GLUT 3.7.6 has to be included. Prior to installing the latest Windows service packs and critical updates have to be updated.

3.2 HARDWARE REQUIREMENTS

Minimum Hardware Requirements	
Processor	133-MHz Intel Pentium-class processor
Memory	128 MB of RAM, 256 MB recommended
Hard Disk	110 MB of hard disk space required, 40 MB additional hard disk space required for installation (150 MB total)
Display	800 x 600 or higher-resolution display with 256 colors

CHAPTER 4

IMPLEMENTATION

4.1 INBUILT FUNCTIONS

4.1.1 BASIC FUNCTIONS

glEnable, glDisable Function

The glEnable and glDisable functions enable or disable OpenGL capabilities.

SYNTAX:

void dlEnable, glDisable(GLenum cap);

PARAMETERS:

- cap - A symbolic constant indicating an OpenGL capability.
- glEnable(GL_DEPTH_TEST);
- glDisable(GL_DEPTH_TEST);
-

glColor3f Function

Sets the current color.

SYNTAX:

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.

glBegin, glEnd Function

The glBegin and glEnd function delimit the vertices of a primitive or a group of like primitives.

SYNTAX:

void glBegin, glEnd(GLenum mode);

PARAMETERS:

- mode - The primitive or primitives that will be created from vertices presented between glBegin and the subsequent glEnd. The following are accepted symbolic constants and their meanings.

GL_LINES-

Treats each pair of vertices as an independent line segment. Vertices 2n-1 and 2n define line n.N/2 lines are drawn.

GL_QUADS-

Treats each group of four vertices as an independent quadrilateral. Vertices 4n-3, 4n-2, 4n-1 and 4n defined quadrilaterals are drawn.

glVertex3f Function

Specifies a vertex.

SYNTAX: Void glVertex3f(GLfloat x,GLfloat y,GLfloat z);

PARAMETERS:

- X- Specifies the x-coordinate of a vertex.
- Y- Specifies the y-coordinate of a vertex.
- Z- Specifies rhe z-coordinate of a vertex.

EXAMPLE- glVertex3f(-3.0,3.0,4.0);

4.1.2 TRANSFORMATION FUNCTIONS

glRotatef Function

the glRotate and glRotatef functions multiply the current matrix by a rotation matrix.

SYNTAX:

void glRotate(GLfloat angle, GLfloat x, GLfloat y, 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.
- glRotatef(xrot,1.0,0.0,0.0);

glTranslate Function

The glTranslate and glTranslatef functions multiply the current matrix by a translation matrix.

SYNTAX:

void glTranslate(x,y,z);

PARAMETERS:

- X,Y,Z-The x,y, and z coordinates of a translation vector.
- glTranslate(0.0,0.0,-1.0);

4.1.3 FUNCTIONS USED TO DISPLAY

glClear Function

The glClear function clears buffers to preset values.

SYNTAX:

glClear(GLbitfield mask);

PARAMETERS:

- mask- Bitwise OR operations of masks that indicate the buffers to be cleared. The four masks are as follows.

Value	Meaning
GL_COLOR_BUFFER_BIT	The buffers currently enable for colorwriting
GL_DEPTH_BUFFER_BIT	The depth buffer.

Eg: glclear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

glMatrixMode Function

The gLMatrix Mode function specifies which matrix is the current matrix.

SYNTAX :

Void gLMatrix Model(GLenum mode);

PARAMETERS :

- Mode – The matrix stack that is the target for subsequent matrix operations. The mode parameter can assume one of three values:

Value	Meaning
GL_MODEL VIEW	Applies sub sequent matrix operations to the Modelview matrix stack.

Eg: gLMatrix Mode(GL_MODELVIEW) :

glLoadIdentity Function

The glLadIdentity function replaces the current matrix with the identity matrix.

SYNTAX :

Void glLoadIdentity(void);

PARAMETERS :

This function has no parameters.

glClear Color

glclearColor used for clearing the color buffers.

SYNTAX :

Void glclearColor(GLclampf r, GLclampf g,
GLclampf b, GLclampf

a)Variables of type GLclampf are floating point numbers between 0.0 and 1.0.

glFlush

Forces execution of GL commands in finite time.

SYNTAX:

Void glFlush(void);

glutPostRedisplay

This function requests the display callback be executed after the current callback returns.

SYNTAX :

Void glutPostRedisplay(void);

4.1.4 FUNCTIONS USED TO SET THE VIEWING VALUE

glOrtho

This function defines orthographic viewing volume with all parameters measured from the centre of projection.

Multiply the current matrix by a perspective matrix.

SYNTAX:

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

PARAMETERS :

- Left, right – Specify the coordinates for the left and right vertical clipping planes.
- Bottom, top – Specify the coordinates for the bottom and top and horizontal clipping plane.
- near Val, far Val – Specify the coordinates for the bottom and top horizontal clipping planes.
- nearVal, farVal – Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.

4.1.5 CALL BACK FUNCTIONS

glutDisplayFunc Function

glutDisplayFunc sets the display callback for the current window.

SYNTAX:

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

PARAMATERS:

- func - The new display callback function.

EG: glutDisplayFunc(display):

glutReshapeFunc Function

glutReshapeFunc Function sets the reshape callback for the current window.

SYNTAX:

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

PARAMATERS:

- Func - The new Idle callback function.

Eg: glutReshapeFunc(reshape);

4.1.6 MAIN FUNCTION

glutInit Function

glutInit is used to initialize the GLUT library.

SYNTAX :

gluInit(int *argcp, char **argv);

PARAMATERS:

* argcp – A pointer to the program's unmodified argc variable from main. Upon return, the value pointer to by argcp will be updated, because gluInit extracts any command line options intended for the GLUT library.

* argv – The program's unmodified argv variable from main. Like argcp, the data for argv will be updated becaue glutInit extracts any command line options understood by the GLUT library.

Eg: glutInit(&argc,argv);

glutInitDisplayMode Function

glutInitDisplayMode sets the initial display mode.

SYNTAX :

Void glutInitDisplayMode(unsigned int mode);

PARAMATERS:

* mode – Display mode, normally the bitwise OR-ing of GLUT display mode bit masks. See values below:

GLUT_RGB: An alias for GLUT_RGBA.

GLUT_DOUBLE: Bit mask to select a double buffered window. This overrides

GLUT_SINGLE if it is also specified.

GLUT_DEPTH: Bit mak to select a window with a depth buffer.

Eg:

glutInitDisplayMode(GLUT_RGB/GLUT_DEPTH/GLUT_DOUBLE);

gluInitWindowPosition, glutInitWindowSize Functions

glutInitWindowPosition and glutInitWindowsize set the initial window position and size respectively.

SYNTAX :

Void glutInitWindowSize(int width, int height);

Void glutInitWindowPosition(int x, int y);

PARAMATERS:

* width – Width in pixels.
* height – Height in pixels.
* x – Window X location in pixels.
* y – Window Y location in pixels.

Eg: glutInitWindowsSize(300,300);

- **glutCreateWindow Function**

glutCreateWindow creates a top-level window.

SYNTAX:

int glutCreateWindow(char *name)'

PARAMATERS:

* name – ASCII character string for use as window name. Eg: glutCreateWindow("Snake game"),

- **glutMainLoop Function**

glutMainLoop enters the GLUT event processing loop.

SYNTAX:

Void glutMainLoop(void),

Eg: glutMainLoop();

4.1.7 Text Displaying Functions

glutBitmapCharacter

It renders the character with ASCII code char at the current raster position using the Bitmap font.

SYNTAX:

Void glutBitmapCharacter(void* font, int char)

PARAMATERS:

- Font – GLUT_BITMAP_HELVETICA

GlRasterPos

glRasterPos specify the raster position for pixel operations

PARAMETERS:

* x, y, z, w- Specify the x, y, z, and w object coordinates (if present) for the raster position.

glutBitmapCharacter

glutBitmapCharacter renders a bitmap character using OpenGL.

SYNTAX:

Void glutBitmapCharacter(void *font, int character);

PARAMATERS:

*font- GLUT_BITMAP_8_BY_13
GLUT_BITMAP_9_BY_15
GLUT_BITMAP_TIMES_ROMAN_10
GLUT_BITMAP_TIMES_ROMAN_24

4.1.8 INTERACTIVE FUNCTIONS

glutKeyboardFunc Function

Registers the keyboard callback function func. The callback function returns the ASCII copde of the key pressed and the position of the mouse.

SYNTAX :

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

PARAMATERS:

- Func – The new keyboard callback function.

glutMousefunc Functoin

Registers the mouse callback function func. The callback function returns the buttion GLUT_LEFT_BUTTON,

GLUT_RIGHT_BUTTION,

GLUT_MIDDLE_BUTTION),

the state of the button after the event (GLUT_UP,GLUT_DOWN) and the position of the mouse with respect to the top left corner of the window.

SYNTAX:

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

CONCLUSION

A **Worm** Graphics package has been developed Using OpenGL.The illustration of graphical principles and OpenGl features are included and application program is efficiently developed.

The aim in developing this program was to design a simple program using Open GL 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.

The Main idea of the program is to create an interactive game, fun to play.