Physiological system modelling

Sekar Parthasarathy B E

This thesis is submitted to Dublin City University as the fulfillment of the requirement for the award of degree of

Master of Engineering

Supervisor Professor M S J Hashmi

School of Mechanical and Manufacturing Engineering
Dublin City University
September 2003

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Engineering is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged with in the text of my work.

Signed: [illegible]

Sekar Parthasarathy

ID No.: 99145553

Date: 26-SEP-2003

Acknowledgements

First I would like to express my sincere thanks and gratitude to Prof M S J Hashmi for giving me this opportunity and also for his constant encouragement and support Without his guidance and support throughout this research it would not have been possible to finish this thesis I am honoured to have him as my supervisor for this research

I would also like to thank Mr Raghupathy Padmanabhan and his wife for their invaluable support and assistance

I would also like to thank my friend Mr Kalaiarul Dharmalingam for his invaluable support

Also I would like to thank the people who helped me at various stages in my research

Finally, I would like to thank my parents for their encouragement and support throughout my studies especially during the research

Physiological system modelling

Sekar Parthasarathy B E

Abstract

Computer graphics has a major impact in our day-to-day life It is used in diverse areas such as displaying the results of engineering and scientific computations and visualization, producing television commercials and feature films, simulation and analysis of real world problems, computer aided design, graphical user interfaces that increases the communication bandwidth between humans and machines, etc Scientific visualization is a well-established method for analysis of data, originating from scientific computations, simulations or measurements The development and implementation of the *3Dgen* software was developed by the author using OpenGL and C language was presented in this report *3Dgen* was used to visualize three-dimensional cylindrical models such as pipes and also for limited usage in virtual endoscopy Using the developed software a model was created using the centreline data input by the user or from the output of some other program, stored in a normal text file The model was constructed by drawing surface polygons between two adjacent centreline points The software allows the user to view the internal and external surfaces of the model The software was designed in such a way that it runs in more than one operating systems with minimal installation procedures Since the size of the software is very small it can be stored in a 1 44 Megabyte floppy diskette Depending on the processing speed of the PC the software can generate models of any length and size Compared to other packages, 3D*gen* has minimal input procedures was able to generate models with smooth bends It has both *modelling and virtual exploration features* For models with sharp bends the software generates an overshoot

Table of Contents

List of figures

List of Tables

CHAPTER 1

Chapter 1 INTRODUCTION

1 1 Introduction

Many definitions of graphics (including computer graphics) can be found However, an apt, contemporary explanation [1] is given by David Cassings of Tektronix, "The representation of quantitative data through visual symbols" portraying reality "by artificial-and sometimes fairly abstract-conventions accepted by the user" Computer graphics displays objects on computer screens for human cognition Thus, computer graphics serves as a type of human interfaces to identify objects in human cognitive spaces Human interfaces heavily rely on visualizing objects [2] and by using computers this task is much more simplified Some of the basic techniques used in visualizing objects are shown to have their origins not in computer graphics but in early cartographic and diagrammatic presentations

It could be argued that the first computer graphics system appeared with the first digital computers, but the Whirlwind project at the Massachusetts Institute of Technology is generally marked as the beginning of computer graphics [3]

The doctoral thesis of Ivan Sutherland, Sketchpad interactive drawing system [4], is considered to be a milestone in computer graphics He introduced data structures for storing symbol hierarchies built up via easy replication of standard components, a technique akin to the use of plastic templates for drawing circuit symbols He also developed interaction techniques that used the keyboard and light pen (a hand-held pointing device that senses light emitted by objects on the screen) for making choices, pointing, and drawing, and formulated many other fundamental ideas and techniques still in use today

1 2 Three-dimensional computer graphics

Following quickly on Sutherland's work, Timothy E Johnson [5] extended Sketchpad to function in three dimensions Nowadays three-dimensional computer graphics is applied in many areas ranging from fantasy world of film and television to more practical areas such as computer-aided design (CAD) of mechanical engineering parts In this sense three-dimensional computer graphics is possibly the most important aspect of computer graphics While certain techniques are used only in engineering, users as diverse as molecular scientists, television animators, architects, urban planners, doctors etc , use three-dimensional modelling and rendering techniques

1 3 Interactive computer graphics

Interactive computer graphics is the most important mechanised means of producing and reproducing pictures since the innovation of photography and television It has added advantage that, with the computer, it is possible to make picture not only of the concrete, real world objects, but also of abstract, synthetic objects and of data that have no inherent geometry, such as survey results Although static pictures are a good means of communicating information, dynamically varying pictures are frequently even more effective, especially for the time-varying phenomena, both real and abstract With motion dynamics, objects can be moved and tumbled with respect to a stationary observer The objects can also remain stationary and viewer can move around them, pan to select the portion in view, and zoom in or out for more or less detail, as though looking through the viewfinder of a rapidly moving video camera Update dynamics is the actual change of the shape, colour, or other properties of the objects being viewed, or modelled Interactive computer graphics permits extensive, high-bandwidth user-computer interaction such as in-flight data of an airplane or display of various parameters of a control system of a nuclear reactor Such interactions significantly enhance the user ability to understand data, to perceive trends, and to visualise real or imaginary objects

1 4 Conceptual framework for interactive graphics

At hardware level the computer receives input from information devices, and output images to display device The software has three components (Figure1 1) The first, the application program, create, store and retrieve from the second component, the application model, which represents the data or objects to be projected on the screen The application program also handles user input This program produced views by sending to the third component, the graphics system, a series of graphics output commands that contain both a detailed geometric description of what is to be viewed and the attributes describing how the objects should appear The graphics system is thus an intermediary between the application program and the display hardware that effects an output transformation from objects in the application model to a view of the model Symmetrically, it effects an input transformation from user actions to inputs to the application program that will cause the application to make changes in the model or picture

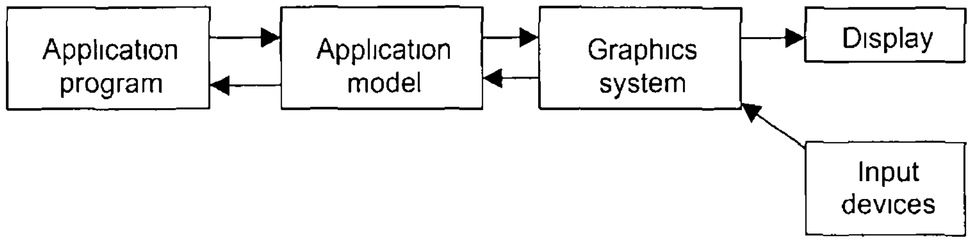


Figure 1 1 Conceptual framework

The fundamental task of the designer of an interactive graphics application program is to specify what classes of data items or objects are to be generated and represented pictorially and how the user and the application program are to interact to create and modify the model and its visual representation Most of the programmer's task concerns creating and editing the model and handling user interaction, rather than actually creating views, since that task is handled by graphics system

1 5 Image display

The primary objective in this field is to efficiently generate realistic images ranging from objects in the real world to objects or scenes of virtual world This encompasses areas such as three dimensional modelling, human perception, computer hardware etc ,

The Figure 1 2 outlines the processes in creating a three dimensional image This shows how a realistic three-dimensional scene can be presented using a two-dimensional medium, such as a Visual Display Unit or a paper

In the first stage, a three-dimensional world model was created which consisted of a finite set of objects (spheres, cones, polyhedra, parametric surfaces, etc) This model existed in the three-dimensional cartesian space [6]

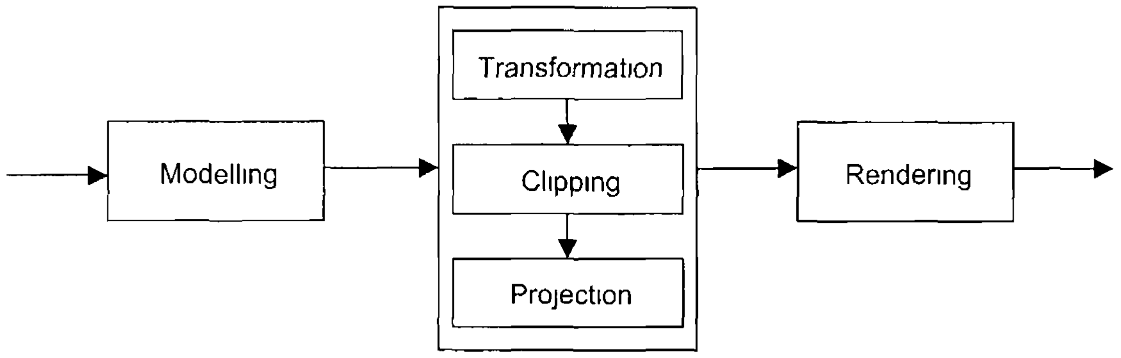


Figure 1 2 Procedure involved in image display

The world model then undergoes a series of transformations, that produce a screen model of the world as it would be seen by an observer with a given position and orientation, this combination of position and orientation is the observer's viewpoint A screen model is a finite set of objects in the two-dimensional cartesian space Points in the screen model may also retain a reference to the corresponding points of the world model These references were used to create a more realistic final image Finally, the objects in the screen model were rendered, producing the final two-dimensional image

Until recently computer graphics was understood and used only by a selected group of people linked with expensive display hardware, complex software and extraordinary computer resources In the last few years, however, it has benefitted from the steady and sometimes even drastic reduction in hardware price because of technological innovations in semiconductors This also led to the development of high level, device independent graphics packages that help to ease the burden in graphics programming

1 6 Motivation

The world of computing is fast evolving due to advancements in semiconductor technology, mathematical modelling and also different types of algorithms In early 1980's computer graphics was a small, specialized field, largely because the hardware was expensive and graphics-based application programs were few Then, personal computers with built-in raster graphics displays such as the Apple Macintosh and the IBM PC and its clones popularised the use of bitmap graphics for user-computer interaction A bitmap is a ones and zeros representation of the rectangular array of points, called pixels or picture elements, on the screen Once bitmap graphics became affordable, an explosion of easy to use and inexpensive graphics based applications soon followed

Graphics based user interfaces allowed millions of new users to control simple, low-cost application programs, such as spreadsheets, word processors and drawing programs The concept of desktop became a popular metaphor for organizing screen space This was done by means of a window manager by which the user can create, resize, position rectangular screen areas on the desktop, called windows, that acted as virtual graphics terminals, each running an application Also part of this desktop was the display of icons that represented not just data files, application programs etc , but also mailboxes, trash can, recycle bin equivalents of their real-life counterparts This leads to the direct manipulation of objects by pointing and clicking which replaced the process of typing the complex commands Thus the users were able to easily interact with computers without the internal knowledge of the complex commands involved in with any kind of application

Thıs led to the wıdespread scenarıo of people usıng computers ın areas rangıng from houses to offıces and ındustrıes Many people developed dıfferent graphıcs packages to cater varıous needs of ındustry as well as common man use In spıte of all the technologıcal developments there are lımıtatıons whıch lımıts wıdespread usage of a software package ın terms of compatıbılıty, computer hardware lımıtatıons, memory sıze of the package etc ,

The present task ıs to develop a software program for educatıon and vısualısatıon purposes of pıpe networks allowıng the user to vıew the ınternal and external surfaces and also lımıted vırtual endoscopy purposes The program should be compatıble wıth more than one operatıng system usıng exıstıng hardware and software resources of a standard desktop PC The developed software "*3Dgen*" ıs one such approach that used datasets obtaıned eıther from some other software or typed ın by the user ın a sımple format and the model was generated based on the datasets The software was developed usıng OPENGL wıth C language The sıze of the software ıs so small that ıt can be stored ın any standard 1 44 Megabyte floppy dıskette

CHAPTER 2

CHAPTER 2 HISTORY AND LITERATURE SURVEY

2 1 Historic perspective

Visualisation can be defined as forming a mental picture of something not visible or present Possible earliest examples of visualisation are from astronomy, meteorology and cartography and the concept is also used in areas like geology, medicine, biology, fluid dynamics etc

2 1 1 Astronomy, Meteorology and Cartography

The earliest examples of data visualization from astronomy, meteorology and cartography were driven by the needs to develop accurate aids to assist sailors in navigating oceans and the military used this technique to survey lands Influenced by the mechanical astrolabes of the early middle ages, the Bavarian lawyer and amateur astronomer Johann Beyer published first modern set of star charts in 1603 [7] He plotted star positions based on the observations of the Danish astronomer Tycho Brahe, which were accurate to one minute of arc Beyer was also the first to draw them on a grid of latitude and longitude lines using the elliptic coordinate system Edmund Halley was the first modern scientist who had the ability to reduce large amounts of numerical data to a meaningful representation He published the first meteorological chart in 1686 in the form of a map of world showing the distribution of prevailing winds over the oceans by the use of arrow plots

The use of contour lines on topographic maps to represent elevations above sea level, and on nautical charts to represent depths, came with the advent of accurate surveying techniques These were first employed from the middle of 18th century, when France, England and Switzerland undertook elaborate national surveys to produce maps for military reasons If a coastline - being the zero elevation – line can be considered a contour line, then contour lines first appeared around 1748 But the more usual, nonzero elevation contour lines were not in common use until after the British Ordinance Survey Act of 1841 An isothermal chart prepared by Alexander von Humboldt in 1817 for low and middle latitudes of the northern hemisphere, was the first use of isoline methods to show the geographic distribution of a quantity other than elevation This was followed by the isochromatic lines (equal colour) in 1829, the isogeothermal lines (equal temperature below the surface of the earth) in 1832 and the isobars (equal pressure) in 1864

2 1 2 Geology and Geography

It was not until towards the end of the last century that data visualisation expanded to include techniques other than isolines and into other scientific disciplines Colour was used in maps between contour lines and in the representation of different types of vegetation This followed the use of gray scale on contour maps of North Wales by Aaron Arrowsmith in 1818 Colour was used in maps to represent different types of rocks

2 1 3 Biology

Related techniques were used in biology, based on the serial sectioning of specimens The reconstruction of the third dimension from serial electron micrographs was achieved by taking micron-thick slices from the specimen Each slice was photographed in an electron microscope and the images were traced on to transparent acetate sheets, which were stacked up with separators Different cell structures were traced in a different colour

2 1 4 Medicine

The field of medicine was revolutionized by the discovery of X-rays by Wihelm Rontgen in 1895 This was followed rapidly by the generation of the first stereo pair of X-ray photographs of a mouse in 1896 More importantly, James MacKenzie-Davidson developed measurement techniques in 1898 for the location of foreign bodies using stereo X-ray photographs These were extensively used during the First World War to assist surgeons in locating bullets and shrapnel However, X-ray photographs have limitations because a three-dimensional volume of data has been flattened on to a two-dimensional image, losing much of the three dimensional information The resolution of this problem came with the emergence of transverse axial tomography in 1938, which allowed X-ray sections or slices to be generated Nowadays computer graphics is playing an ever increasing role in fields such as diagnostic medicine, surgery planning etc In the case of surgery, surgeons use graphics as an aid to guide the instruments and to determine precisely the area of diseased tissues for removal Widely used techniques such as image reconstruction using CT and MRI images makes a significant impact in the area of diagnosis

2 1 5 Fluid dynamics

The field of experimental fluid dynamics has contributed many of the techniques that were used in the visualization of vector properties Thus ribbons were fixed to the surface of aircraft and ships' hull, in wind tunnel and tanks respectively, to observe the fluid flow and vorticity fields that led to the concept of streamlines Smoke particles were released in wind tunnels and dyes injected into liquids to observe the motions of fluid particles which led to the concept of particle advection

2 2 Applications of Computer graphics

2 2 1 Computer Graphics in multimedia systems

Computer graphics and the closely related fields of virtual reality, computer-aided geometric design, and scientific visualization, compact storage and fast display of shape information are vital For interactive applications such as military flight simulators, video games, and computer-aided design, real time performance is a very important goal For such applications, the geometry can be simplified to multiple levels of detail, and display can switch or blend between the appropriate levels of detail as a function of the screen size of each object [8] The process of walkthrough can be defined as redisplaying a static scene from a moving viewpoint For off-line, more realistic simulations such as special effects in entertainment, real time is not vital, but reasonable speed and storage are nevertheless important

2 2 2 Computer Graphics in cartography

Computer graphics is used to produce both accurate and schematic representations of geographical and other natural phenomena from measured data In modern day cartography, simplification is one method among many for the "generalisation" of geographic information [9] In that field, curve simplification is called "line generalisation" It is used to simplify the representations of rivers, roads, coastlines, and other features when a map with large scale is produced It is needed for several reasons to remove unnecessary detail for aesthetic reasons, to save memory/disk space, and to reduce plotting/display time The principal surface type simplified in cartography was, of course, the terrain Map production was formerly a slow, off-line activity, but it is currently becoming more interactive, necessitating the development of better simplification algorithms The ideal error measures for cartographic simplification include considerations of geometric error, viewer interest, and data semantics

2 2 3 Computer Graphics in Finite Element Analysis

Engineers use the finite element method for structural analysis of bridges, to simulate the airflow around airplanes, and to simulate electromagnetic fields, among other applications A preprocess to simulation is a "mesh generation" step In 2D mesh generation, the domain, bounded by curves, is subdivided into triangles or quadrilaterals In 3D mesh generation, the do-main is given by boundary surfaces Surface meshes of triangles or quadrilaterals are first constructed, and then the volume is subdivided into tetrahedron or hexahedron The criteria for a good mesh include both geometric fidelity and considerations of the physical phenomena being simulated (stress, flow, etc) To speed up simulation, it is desirable to make the mesh as coarse as possible while still resolving the physical features of interest

2 2 4 Computer Graphics and Virtual Reality

The role of computers in future world is described of having a synthetic 3D universe that is as believable as the real physical universe Such virtual reality (VR) systems [10] create a cyberspace where it is possible to interact with anything and anyone on a virtual level The key technologies behind such imaginative writing are real-time computer graphics, colour displays and simulation software Virtual reality is used to model and explore familiar environments such as kitchens, planes, offices, studios, ships, submarines, hospitals etc , It is also used to explore unfamiliar environments such as molecules, atoms, galaxies, viruses, crystals etc , VR is creating new ways of manipulating and visualizing all types of data [11] It is also used to train surgeons for new surgical skills [12]

2 2 5 Computer graphics and Medical imaging

The role of accurate investigation and diagnosis and diagnosis in the management of all disease is unquestionable Medical imaging not only provides for diagnosis but also serves to assist with planning and monitoring the treatment of diseases such as cancer The discovery of X-rays, invention of X-ray computed tomography (CT) and magnetic resonance imaging (MRI) changed the whole scenario Nowadays virtual reality is used to reconstruct images obtained using CT and MRI as an efficient and quicker way to diagnose and also to monitor the treatment This also lead to a new technique called virtual endoscopy [13] Three-dimensional models are constructed from CT and MRI image datasets [14] Two methods used for postprocessing the data are surface rendering and volume rendering These techniques are being studied at various research centers for a variety of applications such as exploration [15] and inspection of colon [16], tracheobronchial tree [17], blood vessels, urinary tract etc

2 2 6 Computer Graphics and Education

Computer graphics is widely used in education ranging from digital library to virtual classrooms With the development of immersive, non-immersive and hybrid applications [12] which gives an idea about virtual domain [11] coupled with world wide web promises and interesting concept of e-learning and virtual classroom [18] It is also used to teach various skills such as surgical training etc

2 3 Advantages of computer graphics

Graphics provides one of the most natural means of communicating with a computer In many design, implementation and construction processes today, the information the pictures can give is virtually indispensable Probably it is the most important means of producing pictures since the invention of photography and television It has the added advantage that with the computer it possible to make pictures not only of concrete, real world objects but also of abstract, synthetic objects and of data that have no inherent geometry such as survey results

2 4 Literature survey

Many applications of computer aided design and scientific visualization regularly create complex environments [19] that exceed the interactive visualization capabilities of current graphics

Frederick Brooks [20] presented a virtual walkthrough concept, which aims at providing a tool in which virtual buildings designed but not yet constructed, can be explored by "walking through" them in the same way that simulated airplanes "fly" over virtual terrain However there are many questions left unanswered in terms of frame rate, levels of details to determine the definition of the object and wide-angle view

James Clark [21] suggested that by using an extension of traditional structure information, or a geometric hierarchy, five significant improvements to current techniques are possible Hierarchical approach was a unified structural approach that embodies the ideas of polygon-based approach, parametric surface approach and procedurally modelled objects First, the range of complexity of an environment was greatly increased while the visible complexity of any given scene was kept within a fixed upper limit Second, a meaningful way was provided to vary the amount of detail presented in a scene Third, "clipping" becomes a very fast logarithmic search for the resolvable parts of the environment within the field of view Fourth, frame-to-frame coherence and clipping define a graphical "working set" or fraction of total structure that should be present in primary store for immediate access by the visible surface algorithm Finally, the geometric structure suggests a recursive descent, visible surface algorithm in which the computation time potentially grows linearly with visible complexity of the scene In this structural approach multiple descriptions of the same object is defined which eventually increases the processing time

Erikson and Manocha [22] presented a new approach for fast display of large static and dynamic environments Given a geometric dataset, it is represented using a scene graph and automatically compute levels of detail (LODs) for each node in the graph For drastic simplification, i e , reducing the polygon count by an order of magnitude or more, hierarchical levels of details (HLODs) are computed which represent portions of the scene graph When objects move in dynamic environment, a subset of the HLODs were incrementally recomputed on the fly Some features of the approach are,

- In a given environment the algorithm automatically computed the HLODs of the scene graph The implementation of this approach used GAPS algorithm [52] to compute LODs as well as HLODs
- By grouping objects to create HLODs, polygons were merged from different objects during simplification This merging increased the visual quality of drastic, or low polygon count, approximations
- LODs and HLODs are rendered using display lists, to make the best possible use of performance of current high-end graphics systems The HLOD recomputation algorithm can also utilize multiple processors on high-end graphics machines

The HLODs of a scene graph were computed as follows

- HLODs are recursively generated in a bottom-up fashion for each node in the scene graph
- HLODs of a leaf node are equivalent to its LODs
- HLODs of an intermediate node in the scene graph were computed by combining the LODs of the node with the HLODs of its children The highest resolution HLOD for the node was formed by combining the coarsest LOD of the node with coarsest HLOD of each of its children A simplification algorithm was used to compute a series of hierarchical levels of detail for the node, starting from the initial HLOD The highest resolution HLOD will be discarded from the series once it was complete

The approach can efficiently handle scenes with a limited amount of dynamic changes Furthermore, it supports two rendering modes one that renders at a specified image quality and another that targets a desired frame rate But the algorithm can handle only limited amount of object motion in a large environment

Kenneth Hoff [23] proposed the concept of not drawing the objects which were not seen The increasing demand of 3D game realism – in terms of both scene complexity and speed of animation - are placing excessive strain on the current low-level, computationally expensive graphics drawing operations Despite these routines being highly optimized, specialized and often being implemented in assembly language or even in hardware, the ever increasing number of drawing requests for a single frame of animation causes even these systems to become overloaded, degrading the overall performance To offset these demands and dramatically reduce the load on the graphics subsystem, Hoff [12] presents a system that quickly and efficiently finds a large portion of the game world that is not visible to the viewer each frame of animation, and simply prevent it from being sent to the graphics system A search mechanism is also built for unseen parts from common and easily implementable graphics algorithm

Lindstrom et al [24] presented an algorithm for real-time level of detail reduction and display of hi-complexity polygonal surface data A new level of detail display algorithm was applicable to surfaces that were represented as uniformly gridded polygon height fields The various steps in the algorithm are as follows

- Reduction in number of polygons rendered Typically, the surface grid was decimated by several orders of magnitude with no or little loss in image quality, accommodating interactive frames rates for smooth animation
- Smooth, continuous changes between different surface levels of detail The number and distribution of rendered polygons change smoothly between successive frames, affording maintenance of consistent frame rates
- Dynamic generation of levels of detail in real-time The need for expensive generation of multiresolution models ahead of time was eliminated, allowing dynamic changes to the surface geometry to be made with little computational cost
- Support for a user-specified image quality metric The algorithm was easily controlled to meet an image accuracy level within a specified number of pixels This parameterisation allows for easy variation of the balance between rendering time and rendered image quality

The algorithm used a compact and efficient regular grid representation, and employed a variable screen-space threshold to bound the maximum error of the projected image A coarse level of simplification was performed to select discrete levels of details for blocks of surface mesh, followed by further simplification through repolygonalization in which individual mesh vertices were considered for removal These steps compute and generate the appropriate level of detail dynamically in real-time, minimizing the number of rendered polygons and allowing for smooth changes in resolution across areas of the surface The implementation of the algorithm for approximating and rendering digital terrain models and other height fields, led to a consistent performance at interactive frame rates with high image quality

Chamberlain et al [25] presented a new method for accelerating the rendering of complex static scenes The approach was to construct a spatial hierarchy of cells over the scene and to associate with each cell a simplified representation of its contents This hierarchical method accelerates the rendering process without greatly sacrificing image quality Each node in the hierarchy represented a region of the scene, or cell Associated with each cell was an approximation to the distant appearance of the geometry contained within the cell This approximation can be rendered in less time than the geometry within the cell, and was used in place of the geometry when the projected size of the cell on the image plane was sufficiently small The approach [25] consisted of subdividing the input scene using an octree The appearance of each cell in the octree was approximated using a colour cube – a cube with a colour and opacity associated with each of its six faces The hierarchy was constructed as a preprocessing step and served as a multiresolution volumetric approximation to the original scene At display time, regions of the scene in the near field were drawn using actual geometry Regions further from the viewer were drawn by rendering the faces of their associated colour-cubes, with each cube selected from the hierarchy so that it projected no more than a pixel on the display The technique was applicable to unstructured scenes containing arbitrary geometric primitives and has sublinear asymptotic complexity The scene was rendered using a traversal of the hierarchy in which a cell's approximation was drawn instead of its contents if the approximation was sufficiently accurate

The octree was traversed recursively to render the scene as shown in the algorithm listed below, starting at the root

```
Render(cell)
If no part of cell is in the view frustum return
If projected size of cell< $\epsilon$ then
        Draw cell's colour-cube using a Z-buffer
else if cell is a leaf then
        Draw cell's geometry using a Z-buffer
else
        for each child of cell (in back to front order) do
                Render(child)
        end for
end if
        $\epsilon$ – threshold value of the projected size of the cell on the screen
```

Even though the algorithm handled "suspension-like objects (such as leafy trees) well, it can produce noticeable artifacts when rendering continuous surfaces

Pfister et al [26] presented a paradigm called surface elements (surfels) to efficiently render complex geometric objects at interactive frame rates Unlike classical surface discretizations ı e , trıangles or quadrılateral meshes, surfels are poınt prımıtıves wıthout explıcıt connectıvıty Surfel attrıbutes comprıse depth, texture colour, normal, and others As a pre-process, an octree-based surfel representatıon of a geometrıc was computed and durıng samplıng, surfel posıtıons and normals were optıonally perturbed, and dıfferent levels of texture colours were prefıltered and stored per surfel Whıle renderıng, a hıerarchıcal forward warpıng algorıthm projects surfels to a z-buffer Vısıble surfels were shaded usıng texture fılterıng, Phong ıllumınatıon and envıronment mappıng usıng per-surfel normals Because of the sımplıcıty of the operatıons, the surfel renderıng pıpelıne ıs amenable for hardware ımplementatıon They concluded by sayıng that surfel renderıng was capable of hıgh ımage qualıty at ınteractıve frame rates ıncreasıng the processor performance and possıble hardware support wıll brıng ıt ınto the realm of real-tıme performance

Teller and Sequın [27] presented a method for renderıng archıtectural models at ınteractıve frame rates They descrıbe a method of vısıbılıty preprocessıng that was effıcıent and effectıve for axıs-alıgned or axıal archıtectural models The model was subdıvıded ınto rectangular cells whose boundarıes coıncıde wıth major opaque surfaces After the subdıvısıon, a maxımal set of sıghtlınes was found from each cell to the rest of the subdıvısıon The novel aspect of the algorıthm was that sıghtlınes were not cast from dıscrete sample locatıons Instead, cell-to-cell vısıbılıty was establıshed ıf a sıghtlıne exısts from any poınt ın one cell to any poınt ın another The data structure created durıng thıs gross vısıbılıty determınatıon was stored wıth each cell, for use durıng an ınteractıve walkthrough phase

Non-opaque portals are ıdentıfıed on cell boundarıes and used to form an adjacency graph connectıng the cells of the subdıvısıon They also made two sımplıfyıng assumptıon ın whıch they restrıct the attentıon to “faces” that are axıal lıne segments ın the plane ı e lıne segments parallel to eıther x or y-axıs and also the coordınate data occur on a grıd Then they compute cell-to-cell vısıbılıty for each cell of the subdıvısıon by lınkıng paırs of cells between whıch unobstructed sıghtlınes exıst The cell-to-cell vısıbılıty can be further dynamıcally culled agaınst the vıew cone of an observer, agaın producıng a relıable superset of the vısıble scene data, the eye-to-cell vısıbılıty The detaıled data contaıned ın each vısıble cell, along wıth assocıated normal, colour, texture data etc , were passed to hardware renderer for removal of hıdden surfaces (ıncludıng those polygons ınvısıble to observer, whıch was crucıal) The two-fold model prunıng descrıbed admıts a dramatıc reductıon ın the complexıty of the exact hıdden-surface determınatıon that must be performed by a real-tıme renderıng system

Durıng an ınteractıve walkthrough phase, an observer wıth a known posıtıon and vıew cone moved through the model At each frame, the cell contaınıng the observer was ıdentıfıed, and the contents of potentially visible cells were retrieved from storage The set of potentially visible cells were then sent to a graphics pipeline for hidden surface removal and rendering But generalising the visibility computations to non-axial scenes posed a problem, which was of conceptual and technical nature Even though the method produces sightlines through portals in two dimension it failed when the portals are in three-dimension

Brodsky and Watson [28] presented an algorithm, R-Simp, for simplification of large models at interactive speeds The algorithm was fast and guaranteed to display results within a given time limit and of good quality R-Simp was inspired by splitting algorithms from the vector quantization literature [29] The models were simplified in reverse, beginning with an extremely coarse approximation and refining it At every iteration of the algorithm, the number of vertices in the simplified model was known, enabling control of output model size Approximations of surface curvature guide the simplification process, permitting preservation of important model features, and thus a reasonable level of output model quality Performing simplification in a reverse direction makes it possible to refine intermediate output as long as some state of information was saved Because of its divide and conquer approach, the algorithm can be extended to create continuous level of detail hierarchies The algorithm's complexity was O(n_i log n_o), where n_i was the size of the input model and n_o was the size of the output model This enables the algorithm to scale linearly with respect input size for a given output size

The algorithm begins with the triangulated model in a single cluster (cluster – collection of faces from the original model) The initial cluster was then divided into eight sub-clusters These eight sub-clusters were then iteratively divided until the required number of clusters was reached Clusters were chosen for division based on the amount of normal variation on the surface in the cluster The three steps of the algorithm are

- Initialisation global face and vertex lists were created, as well as vertex and vertex – face adjacency lists Also eight clusters were created
- Simplification the model simplification consists of four stages
    a Cluster with most face normal variation was chosen
    b Based on the amount and direction of the face normal variation clusters were partitioned
    c The amount of face normal variation in each of the sub-clusters was computed
    d Iteration until the required number of clusters was reached

- Post processing a representative vertex was computed for each cluster that was left and the model was retriangulated

It was also possible to further refine the earlier simplifications by using them as input to the algorithm As mentioned earlier the algorithm fails to produce a simplified model in a given time frame if the input model is extremely large Since the algorithm simplifies in a coarse to fine direction, it should be well suited for application in progressive transmission of 3D models

Erikson and Manocha [30] proposed a new approach for simplifying polygonal objects The approach in general works on models that contain both non-manifold geometry and surface attributes It was automatic since it required no user input to execute and returns approximate error bounds used to calculate switching distances between levels of detail, or LODs The algorithm called General and Automatic Polygonal Simplification, or GAPS for short, uses an adaptive distance threshold and surface area preservation along with a quadric error metric to join unconnected regions of an object

The name comes from the ability to fill in the gaps of an object GAPS used a new object space error metric that combines approximations of geometric and surface attribute error It efficiently produces high quality and drastic simplifications of a wide variety of objects, including complicated pipe structures This ability to perform drastic simplification allowed it to create levels of detail to accelerate the rendering of large polygonal environments, consisting of hundreds of objects

The algorithm can be described as follows

- Value of τ (distance threshold) checked and if necessary the value was doubled Doubling occurs when either there were no more local pairs in the heap or when the pair on top of the heap had error greater than τ When the value of τ was doubled, error and legality of all active pairs must be recalculated
- Top pair from heap was extracted Two vertices in the pair were deleted from the hash table with grid cells of size τ New vertex was inserted into hash table to find virtual edges Quadrics and point clouds were added and information was stored with new vertex
- Pairs affected by vertex merge were either updated or deleted
- Iteration until a specified number of vertices or faces remain or an error threshold had been reached

The algorithm handled all polygonal objects, whether they were manifold meshes or unorganised lists of polygons. As a preprocess, objects were triangulated and then represented by sharing vertices and calculating normals. A simple internal representation was used for objects where faces consist of three corners. Each corner had a pointer to a vertex and a vertex attribute. Vertices consisted of a three dimensional point plus a list of pointers to adjacent faces. Even though two objects may be in close proximity but the algorithm will never merge disconnected regions between the pair because it simplified them separately.

Cohen et al [31] proposed the idea of simplification envelopes for generating hierarchy of level-of-detail approximations for a given polygonal model. Simplification envelopes were a generalisation of offset surfaces. Given a polygonal representation of an object, they allow the generation of minimal approximations that were guaranteed not to deviate from the original by more than a user-specifiable amount while preserving global topology. The original polygonal surface was surrounded by two envelopes and simplification was performed within this volume. This approach guaranteed that all points of an approximation were within a user-specifiable distance 'ϵ' from the original model and that all points of the original model are within a distance 'ϵ' from the approximation. Simplification envelopes provide a general framework within which a large collection of existing simplification algorithms can run. They demonstrated this technique in conjunction with two algorithms, one local and the other global. The local algorithm based on the work of [32], [33] and [34], provided a fast method for generating approximations to large input meshes. It begins by placing all vertices in a queue for removal processing. For each vertex in the queue an attempt was made to remove it by creating a hole (removing the vertex's adjacent triangles) and attempting to fill it. If the hole was successfully filled, the mesh modification was accepted, the vertex was removed from the queue and its neighbours were placed back in the queue. If not, the vertex was removed from the queue and the mesh remains unchanged. This process terminates when the global error bounds eventually prevent the removal of any more vertices.

The global algorithm [35] provided the opportunity to avoid local minima and possibly achieve better simplifications as a result. Each approximation attempts to minimise the total number of polygons required to satisfy the above constraint. The key advantages of their approach are

- General technique providing guaranteed error bounds for genus-preserving simplification.
- Automation of both the simplification process and the selection of appropriate viewing distances.

- Prevention of self-interaction.
- Preservation of sharp features.
- Allows variation of approximation distance across different portions of a model.

While implementing this approach the authors chose to accept only manifold triangle meshes (or bordered manifolds). This means that each edge is adjacent to two (one in case of a border) triangles and that each vertex is surrounded by single ring of triangles.

Li and Watson [36] presented “Semisimp” a tool for semiautomatic simplification of three dimensional polygon models. Existing automatic simplification technology was quite mature, but it seems not sensitive to the heightened importance of distinct semantic model regions such as faces and limbs, nor to simplification constraints imposed by model usage such as animation. “Semisimp” allowed the users to preserve such regions by intervening in the simplification process. Users can manipulate the order in which basic simplifications are applied to redistribute model detail, improve the simplified models themselves by repositioning vertices with propagation to neighbouring levels of detail, and adjust the hierarchical partitioning of the model surface to segment simplification and improve control of reordering and position propagation. “Semisimp” was a unique synthesis of simplification and multiresolution modelling functions, emphasizing the improvement of aggressively simplified models. It begins by accepting a fully detailed model as input and applying an automatic simplification algorithm to construct a simplification hierarchy. Users can then edit and improve this hierarchy for their target application in three ways.

- Order manipulation: User can adjust the distribution of detail on simplified models by changing the order in which model regions were simplified. This was accomplished through matching changes in the order in which the simplification hierarchy was traversed.
- Geometric manipulation: User can improve the positions of vertices in simplified models. These improvements can be automatically propagated to both simpler and more detailed models. Propagation to more detailed models can be attenuated to preserve the shape of the original model.
- Hierarchy manipulation: User can halt the simplification, modify the partitioning of the original model described by the partial simplification hierarchy to match semantics and intended model use, and continue simplification in a segmented fashion. Since both geometric and order manipulation operate in the context of the simplification hierarchy, their effectiveness was greatly increased.

But geometric manipulations propagated to descendants can easily introduce discontinuities in the model surface when the manipulations deviate significantly from the shape of the input model. The authors suggested that it might be possible to reduce these discontinuities with more advanced filtering and smoothing schemes. Propagated geometric manipulations can also alter previously made geometric manipulations. The authors suggested that it was possible to elaborate interpolation scheme between manipulated nodes of the simplification of hierarchy. Also “Semisimp” takes a long time to process larger models.

Low and Tan [37] presents a technique to automatically compute approximations of polygonal representations of 3D objects. It was based on a previously developed model simplification technique which applies vertex-clustering.

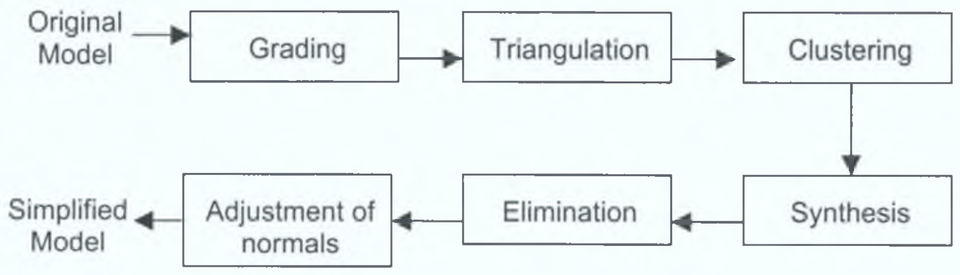


Figure. 2.1 Vertex clustering method

The process or method has the following steps:

i. Grading – a weight was computed for each vertex according to its visual importance.
ii. Triangulation – polygons were divided into triangles.
iii. Clustering – vertices were grouped into clusters based on geometric proximity.
iv. Synthesis – a vertex representative was computed to replace the vertices in each cluster and thus simplified some triangles into edges and points.
v. Elimination – duplicated triangles, edges and points were removed, and
vi. Adjustment of normals – normals of resulting edges and triangles were reconstructed.

Major advantages of the vertex-clustering technique were its low computational cost and high data reduction rate, and thus suitable for use in interactive applications. The authors improved the technique with careful consideration of approximation quality and smoothness in transitions between levels of simplification, while maintaining its efficiency and effectiveness. Its major contributions include: accuracy in grading vertices for indication of their visual importance, robustness in clustering for better preservation of important features and consistencies between levels of simplification, thick-lines with dynamic normals to maximize visual fidelity, and exploitation of object and image space relationship for levels-of-simplification.

Lindstrom and Turk [38] were under the view that conventionally, in order to produce high-quality simplified polygonal models, the information about the original model must be retained and used during the simplification process. But the authors said that excellent simplified models can be produced without the need to compare against the information from original geometry while performing local changes to the model. Their approach was to use edge collapses to perform simplification like many other methods. This approach selected an edge and replaces it with a single vertex. This removed one vertex, three edges and two faces. The position of the new vertex was selected so that the original volume of the model was maintained and to minimize the per-triangle change in volume of the tetrahedra swept out by those triangles that were moved. The surface areas near the boundaries were maintained and the per-triangle area changes are minimized. Two decisions were central to a simplification method that uses edge collapse:

i. The position of the new vertex created by the edge collapse and
ii. The ordering of the edges to be collapsed (the edge priority)

Volume and surface area information were used to make both the decisions. The placement of the new vertices was constrained so that the volume of the closed model was not altered. If the new vertex was near a boundary of the model, the surface area surrounding the edge that was being collapsed was preserved. Often these two constraints do not fully determine the position of the new vertex. The volume swept out by triangles that are moved by the operation were minimized. The area swept out by boundary edges were also minimized and finally attempt to produce well-shaped triangles if the vertex was still underconstrained. The method unifies different constraints by describing each of them as one or more planes in which the new vertex must lie. When three non-parallel planes were determined, the vertex position was fully defined. Calculating the edge collapse priorities and the positions of the new vertices requires only the face connectivity and the vertex location in the intermediate model.

Maciel and Shirley [39] presented a visual navigation system using textured clusters. The system used texture-mapped primitives to represent clusters of objects to maintain high and approximately constant frame rates. In cases of more unoccluded primitives inside the viewing frustum that can be drawn in real time on the workstation, this system ensures that each visible object, or a cluster that included it, was drawn in each frame. The system also supports the use of traditional level-of-detail representations for individual objects, and supports the automatic generation of certain type of level-of-detail for objects and cluster of objects. The system supports the concept of choosing a representation from among those associated with an object that accounts for the direction from which the object was viewed. The system as a whole can be viewed as generalization of the level-of-detail concept, where the entire scene was stored as a hierarchy of levels-of-detail that was traversed top-down to find a good representation for a given viewpoint. The system does not assume that visibility information can be extracted from the model and is thus especially suited for outdoor environments.

Westermann and Ertl [40] proposed an idea for extensively using graphics hardware for the rendering of volumetric data sets in various ways. OpenGL [41] and its extensions provide access to advanced per-pixel operations available in the rasterisation stage and in framebuffer hardware of modern graphics workstation. Westermann and Ertl dealt with the efficient generation of a visual representation of the information present in volumetric data sets. For scalar-valued volume data two standard techniques, the rendering of iso-surfaces, and the direct volume rendering, have been developed to a high of degree sophistication. However, due to the huge number of volume cells, which have to be processed, and to the variety different cell types only a few approaches allow parameter modifications and navigation at interactive rates for realistically sized data sets.

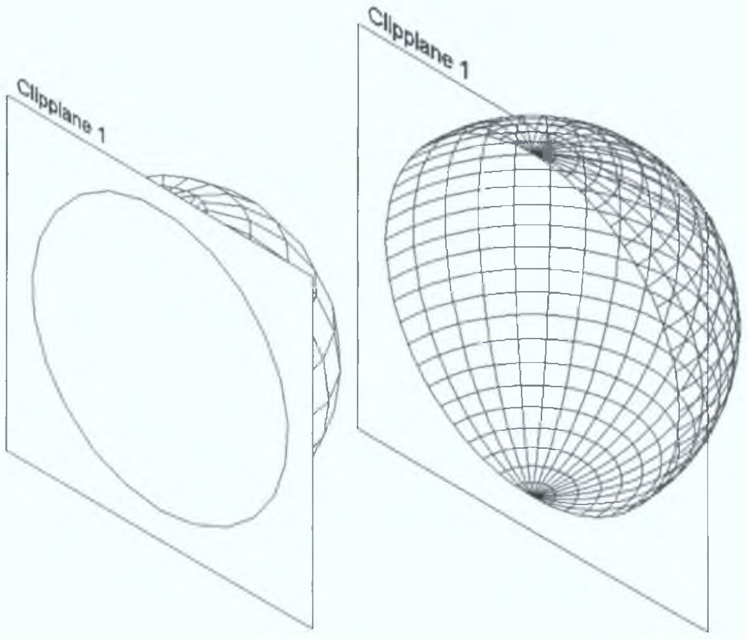


Figure. 2.2. Clipping geometries

They introduced the concept of clipping geometries by means of stencil buffer operations and exploit pixel textures for mapping of volume data to spherical domains. The approach was pixel oriented, taking advantage of rasterisation functionality such as colour interpolation, texture mapping, colour manipulation in the pixel transfer path, various fragment and stencil tests, and blending operations. The idea was to determine all pixels which were covered by the cross section between the object and the actual slicing plane (Figure 2.2).

These pixels were locked, thus preventing the textured polygon from getting drawn to these locations. The locking mechanism was implemented exploiting the OpenGL stencil buffer test. It allows pixel updates to be accepted or rejected based on the outcome of a comparison between a user defined reference value and the value of the corresponding entry in the stencil buffer. Before the textured polygon gets rendered the stencil buffer has to be initialised in such a way that all colour values written to pixels inside the cross-section will be rejected. In order to determine for a certain plane whether a pixel was covered by a cross section or not, the clipping object was rendered in polygon mode. An additional clipping plane was enabled which has the same orientation and position as the slicing plane. All back faces with respect to the actual viewing direction were drawn and everything in front of the plane was clipped. Wherever a pixel would have been drawn the stencil buffer was set. By changing the stencil test appropriately, rendering the textured polygon, now, only affects those pixels where the stencil buffer was unchanged.

3D texture mapping and advanced pixel transfer operations were combined in a way that allows the iso-surface to be rendered on a per-pixel basis. The approach was similar to traditional volume ray-casting for the display of shaded iso-surfaces. If the material values along the ray exceeded the iso-value for the first time which means the surface was hit. At that location the material gradient was computed which was then used in the lightening calculations. The texture samples above the iso-value nearest to the image plane were evaluated by OpenGL alpha test, which was used to reject pixels, based on the outcome of a comparison between their alpha component and a reference value. Each element of the 3D texture was assigned the material value as it's alpha component. Texture mapped rendering was performed, but pixel values were only drawn if they satisfy the z-buffer test and also the alpha value should be greater or equal to selected iso-value. This idea can be used only in high-end graphics workstations.

Guthe et al [42] presented an algorithm for rendering very large volume data sets at interactive frame rates on standard PC hardware. The algorithm accepts scalar data sampled on a regular grid as input. The input data was converted into a compressed hierarchical wavelet representation in a preprocessing step. The representation was much more compact that allows for an efficient extraction of different levels of detail of the data set, since the wavelet transformation was equivalent to applying a series of lowpass and highpass filters to the original data. During rendering, the wavelet representation was decompressed on-the-fly and rendered using hardware texture mapping. For rendering the volume data was represented in the form of a multiresolution octree: the root node in the tree contained a very rough approximation of the data set. The task was to extract information relevant for a certain point of view. This can be done in two steps: a) Projective classification of step was performed to adjust the resolution of the data set to the screen resolution. B) A consideration of approximation error was incorporated into the classification algorithm to further reduce the amount of data to be processed in each frame. The level of detail used for rendering was adapted to the local frequency spectrum of the data and its position relative to the viewer. The problem of out-of-core rendering was latency due to hard disk seek times.

Aliga et al [43] presented a framework for rendering very large 3D models at nearly interactive frame rates. The framework scales with model size and it can integrate multiple rendering acceleration techniques including visibility culling, geometrical levels of detail, and image based approaches. A database representation scheme was used by the framework for the massive models. Data representation and database management was the major issue in integrating multiple techniques for the display of large models. A representation should support multiple rendering acceleration techniques and be scalable across computers with different amounts of memory. A scene graph represents the model in a bounding volume hierarchy, a viewpoint cell structure to manage the run-time walkthrough, and geometry and texture prefetching to make the system adaptable to variable memory sizes.

Scene graph was a bounding volume hierarchy in which each node groups spatially proximate geometry. Many real-world models had an object hierarchy which groups geometry according to non-spatial criteria. On such models, a top-down spatial subdivision of polygon centers were performed to find a usable hierarchy. An octree style subdivision recursively subdivided the model, terminating when either a minimum number of polygons per leaf or a maximum depth was reached. A bounding volume hierarchy for polygons were computed by first computing the bounding boxes of each leaf of the spatial subdivision and then propagate those boxes to the root of scene graph. An important component of a framework for a scalable walkthrough system was a method for localising the geometry rendered. A method based on viewpoint cells was used and cull boxes to provide this localisation. The 3D space of the input model was partitioned into a set of cells. The cull box, associated with each cell, was an axis-aligned box containing cell and was considerably larger than the cell itself. When the viewpoint was in a particular cell all the geometry that lies completely outside the cell's cull box were culled. The geometry needed to render the user's view was typically only a small subset of entire model. Only that geometry and textured depth mesh data needed to render for the current cell must actually be in main memory. An effective pipeline was provided to manage the allocation of system resources among different techniques. The framework was not focussed on high-density objects and also models with moving parts.

LaMar et al [44] presented a magnification lens technique for volume visualisation. Volume visualisation of large data sets suffers from either the user can visualise the entire data set and loosing small details or visualise small region and loose context. The issue with the magnification lens is the boundary or the transition region. The lens centre and the exterior have constant zoom factor and are simple to render. It was the border region that blends between the external and internal magnification and has a non-constant magnification. The perspective correct textures capability, available in most current graphics system was used to produce a lens with a tessellated border region that approximates linear compression with respect to the radius of magnification of lens. The concept failed to efficiently clip the lens against individual tiles.

Xia et al [45] presented an algorithm for performing adaptive real-time level of detail based rendering for triangulated polygonal models. The simplifications were dependent on viewing direction, lighting and visibility and were performed by taking advantage of image-space, object-space and frame-to-frame coherences. In contrast to the traditional approaches of precomputing a fixed number of level of detail representations for a given object the approach involves statically generating a continuous level of detail representation of the object. This representation was then used at run-time to guide the selection of appropriate triangles for display. The list of displayed triangles was updated incrementally from one frame to the next. The approach was more effective than the current level of detail based rendering approaches for most scientific visualization applications where there are a limited number of highly complex objects that stay relatively close to the viewer. The approach was also applicable for scalar as well as vector attributes.

Schaufler and Stürzlinger [46] presented a method by which a small detail in the model was represented by many spatially close points. The approach implements a reasonable fast method to generate several LODs from polygonal object models. The algorithm was not required to exactly keep the topology of the geometry. Nevertheless the generated LODs resemble the original model closely though with less and less polygons. The coarsest LOD should not contain more than a few dozen polygons for an original object consisting of several thousands polygons. The algorithm can be described as followed:

- A hierarchical clustering algorithm was applied to the vertices of the object model to produce a tree of clusters.
- For each LOD a new object model was generated using cluster representatives instead of original polygon vertices.
- Removal of multiple occurring, redundant primitives from each LOD.

In the second stage a layer in the cluster tree was determined which described the way the LOD approximated the original model. Using the clusters of this layer, each polygon had its vertices replaced by the representative of the cluster it belonged to. This may leave the number of vertices unchanged, the number of vertices may be reduced, the polygon may collapse into a linestroke or its new representation may be a single point. In this way formerly unconnected parts of the model may eventually become connected when separated points of different polygons fall into one cluster and were, therefore, mapped onto one cluster representative. However, as the clustering algorithm only clusters spatially close points the overall appearance of the object remains the same depending on the degree of approximation desired in the current LOD. In particular polygons become bigger if their points were moved apart through clustering. Therefore the surface object will never be torn apart.

A hierarchical clustering algorithm was used to generate hierarchy of clusters from the vertices of the object's polygons. Each cluster was replaced by one representative point and polygons are reconstructed from these points. A static detail elision algorithm was implemented to prove the practicability of the method.

Chhugani et al [47] presented vLOD, a scalable system for performing interactive walkthroughs of very large geometric models on commodity graphics hardware. The system performs work proportional to the required detail in visible geometry. A pre-computation phase was used to determine cell based visibility as well as level of detail. This pre-computation phase generates the geometry visible from a view cell at the right level of detail. The changes between the neighbouring cell's vLOD were encoded, which was not required to be memory resident. Incremental reconstruction of vLOD for the current view-cell was done then it was rendered. Due to the small runtime overhead, it was possible to display models with over tens of million polygons at interactive framerates with less than 1 pixel error. The performance of rendering algorithm was directly dependent on the size of vLODs and the implementation was unable to control the size of vLODs for each cell. When the view-cells were subdivided it only increased the storage requirement without sufficient improvement in the vLOD size. Only limited update of detail was allowed during rendering.

Cohen et al [48] presented GLOD, a geometric level of detail system integrated into the OpenGL rendering library. GLOD provides a low-level lightweight API for level of detail operations. Unlike heavyweight scene graph systems, GLOD supported incremental adoption and may be easily integrated into existing OpenGL applications. GLOD provided a simple path for developers to add level of detail to their system, while retaining a minimalist close-to-the hardware approach compatible with high performance rendering and future extension of the OpenGL layer. GLOD is not a scene graph system.

The fundamental principles that lead to the formation of GLOD and its API are:

- Incremental adoption: Unlike scene graph approaches, users of the GLOD API should not be forced into use of the entire pipeline, but should instead be allowed to independently adopt just the portions of the system that they desire.
- FastPath Principle: for every type of LOD task, the API must support a way to achieve that task in a high performance fashion.
- The API should be straightforward to use, especially to any developer already familiar with OpenGL.
- Two forms of extensibility were required. First, through development efforts, GLOD API must be capable of supporting a wide variety of geometric level of detail tasks. Second, the GLOD API must not lose its usefulness through lack of development, but instead must evolve without code redesign as OpenGL acquires new extensions.

This minimalist API should leave as much as possible to the user, keeping the interface simple for simple applications while providing parameters where necessary for advanced users to hook into features necessary for high-performance rendering.

Schmalstieg and Schaufler [49] presented a new class of polygonal simplification called Smooth LODs. A very large number of small details encoded in a data stream allows a progressive refinement of the object from a very coarse approximation to the original high quality representation. Advantages of the new approach includes progressive transmission and encoding suitable for networked applications, interactive selection of any desired quality, and compression of the data by incremental and redundancy free coding.

Funkhouser and Séquin [50] described and adaptive display algorithm for interactive frame rates during visualization of very complex environments. The algorithm relies upon a hierarchical model representation in which objects were described at multiple levels of detail and can be drawn with various rendering algorithms.

An object tuple(O,L,R) was as the instance of the object O, rendered at level of detail L, with rendering algorithm R. The two heuristics for object tuples were defined as *Cost(O,L,R)* and *Benefit(O,L,R)*. The Cost heuristic estimates the time required to render an object tuple; and the Benefit heuristics estimates "the contribution of model perception" of a rendered object tuple. S was defined as the set of object tuple rendered in each frame. Using these formalisms, the approach for choosing a level of detail and rendering algorithm for each potentially visible object can be stated as:

Maximize:

$$\Sigma_S \text{Benefit}(O,L,R);$$

Subject to:

$$\Sigma_S \text{Cost}(O,L,R) \leq \text{TargetFrameTime}.$$

As such, it can be applied to a wide variety of problems that require images to be displayed in fixed amount of time, including adaptive ray tracing (i.e. given a fixed number of rays, cast those that contribute most to the image), and adaptive radiosity (i.e. given fixed number of form-factor computations, compute those that contribute most to the solution). Using this approach it is possible to generate images in a short, fixed amount of time, rather than waiting much longer for images of the highest quality attainable.

For this approach to be successful, Cost and Benefit heuristics had to computed quickly and accurately. Unfortunately, Cost and Benefit heuristics for a specific object tuple cannot be predicted with perfect accuracy, and may depend on other object tuples rendered in the same image. A perfect cost heuristic may depend on the model and features of the graphics workstation, state of the graphics system, state of the operating system and state of the other programs running on the machine. A perfect Benefit heuristic would consider occlusion and colour of other object tuples, human perception and human understanding. It was not possible to quantify all of these complex factors in heuristics that can be computed efficiently. However, using several simplifying assumptions, Cost and Benefit heuristics that are both efficient to compute and accurate enough to be useful was developed.

The idea behind the algorithm was to adjust the image quality adaptively to maintain a uniform, user-specified target frame rate. A constrained optimisation was performed to choose a level of detail and rendering algorithm for each potentially visible object in order to generate the best image possible within the target frame time. The aim was to find combinational levels of details and rendering algorithms for all potentially visible objects that produces the best image possible within the target frame rate.

Funkhouser et al [51] presented techniques for managing large amounts of data during an interactive walkthrough of an architectural model. These techniques were based on a spatial subdivision, visibility analysis, and a display database containing objects described at multiple levels of detail. In each frame of the walkthrough, a set of objects were computed to render i.e. those potentially visible from the observer's viewpoint, and a set of objects to swap into memory, i.e. those that might become visible in near future. An appropriate level of detail was chosen at which to store and to render each object, possibly using very simple representations for objects that appear small to the observer, thereby saving space and time. Using these techniques large portions of the model, which were irrelevant from observer's viewpoint, were culled and thereby achieving interactive frame rates.

Volino and Thalmann [52] proposed a fast geometrical wrinkling algorithm which can be implemented on top of any rough surface deformation model, and which modulated the amplitude of a predefined wrinkle height map in order to simulate metric surface conservation. A wrinkle pattern was initially applied on the animated surface mesh. The native edge length of the mesh was used to compute dynamically the amplitude of the wrinkles as the mesh was deformed using a fast and robust geometric model. Several wrinkle patterns can be combined to simulate complex deformations.

Hong et al [53] presented a method called 3D virtual colonoscopy which incorporates several advanced visualisation techniques to enable the physician to virtually examine the inner surface of the colon for identifying and inspecting colonic polyps. Using normal optical colonoscopic procedures spiral CT scan of patient's abdomen are taken and the 2D slices are reconstructed into a 3Dvolume. After the construction of 3Dvolume, finding the key points and determining the camera direction at each key point generate a fly-through animation along the inside of the colon.

Beckhaus et al [54] presented a system which can deal with rapidly changing user interest in objects of a scene or model as well as with dynamic models and changes of the camera position introduced interactively by the user or through cuts. The system, CubicalPath, a field based camera control system that helps with the exploration of virtual environments. The CubicalPath method operated only on the number of the cubes that define that cube space. It uses an "abstract" and simplified version of the geometry data through its cubes. Many limitations such as the camera could end up alternating between two objects having the same attraction instead of first viewing on objects for a while and then moving to other, occurrence of unwanted local minima.

Pridmore et al [55] presented a method by which camera orientation relative to the pipe axis may be recovered from a single frame of a survey video of a small-bore brick sewer. If it can be fully recovered, the pipe axis provided part of a frame of reference within which three-dimensional descriptions of sewer shape may be expressed. The concept of the vanishing point was introduced and it was shown that the vanishing point position supplies information about the relative orientations of the camera and pipe axes. A method for the automatic detection of vanishing points is presented and used to analyse the camera motion underlying a number of sewer survey videos. The technique might form an active part of a more comprehensive image understanding system recovering pipe shape and/or be used as an experimental tool during the design of such a system. The method fails when there is a bend section in the pipe.

CHAPTER 3

CHAPTER 3: SOFTWARE AND GRAPHICS STANDARDS

3.1 Introduction

A model is a representation of some features of a concrete or abstract entity. The purpose of a model of an entity is to allow people to visualise and understand the structure or behaviour of the entity and to provide a convenient vehicle for experimentation with and prediction of the effects of input or changes to the model. A selection of modelling and animation software are available to cater people in various fields. Not all the softwares can cater the various needs of the fields. Some software are developed purpose built to cater a specific area or task. This can be termed as both advantageous and also disadvantageous. Since the software is developed for a particular task or with limitations the user or developer can maximise the available resources which is advantageous and this also acts as a disadvantage because the same software cannot be used for other tasks or simply it is not flexible. Developers analyse the needs and impact of the software before deciding on the framework and limitations of the software, which is crucial. The approach to the development can be roughly classified into two types, namely

1. Software or programmes developed using other software (eg. 3Dstudio MAX, VRTSuperScape etc.)
2. Software developed using computer languages with or without various library fuctions. (Eg. C,C++, PASCAL, FORTRAN, OpenGL, DIRECTX etc.,)

This chapter gives a brief overview of 3Dstudio MAX and also various features and flexibility of OpenGL, which along with C language was used to develop the software 3D*gen*.

3.2 3D Studio MAX

3D Studio MAX referred as 3DS MAX is a single document application widely used to create professional quality 3D models, photo realistic images and film quality animation. The cornerstone of 3DS MAX is an integrated modelling environment. It performs 2D drawing, 3D modelling, and animation etc. within the unified workspace [56]. Various complex objects and shapes are predefined so that the user or the developer can build the model or scene by selecting various objects and entering the required information. A scene is a collection of various objects and or shapes in a particular fashion. 3DS MAX organizes the building of scene into seven basic categories: Geometry, Shapes, Lights, Cameras, Helpers, Space wraps, and systems and each category contains multiple subcategories. The user can edit the objects into their final form by changing parameters, applying modifiers and directly manipulating sub-object geometry. The modifiers are stored in a stack. The user can revert back at any time and change the effect of the modifier or removing it from the object.

It also provides a sophisticated materials editor to create realistic materials by defining hierarchies of surface characteristics. The surface characteristics can represent static materials or be animated for special effects. It allows the user to create lights with various properties to illuminate the scene. 3DS MAX has three types of lighting: omni, spot, directional lights.

- Omni light – unfocussed light in all directions. It is like an unshaded bulb
- Directional light – Parallel light rays in a single direction as the sun does at the surface if the earth.
- Spotlight – focused beam of light towards a specified target point.

Lights can be set to any colour and can even animate the colour to simulate dimming or colour shifting lights. The lights can cast shadows, project images and create volumetric effects for atmospheric lighting. Cameras can be created which will have real-world controls for lens length, field of view and motion control such as dolly, pan and truck. Real world cameras use lenses to focus the light reflected by a scene onto a focal plane that has a light-sensitive surface. The distance between the lens and the light-sensitive surface is called focal length of the lens. Focal length affects how much of the subject appears in the picture. Lower focal length means more of the scene in the picture and higher focal length means less of the scene but shows greater detail of more distant object.

The field of view controls how much of the scene is visible and it is measured in degrees. If the focal length is longer then the field of view will be narrower and vice versa. Short focal lengths emphasize the distortions of perspective, making objects seem in-depth. Long focal lengths reduce perspective distortion, making objects appear flattened and parallel to the viewer. Camera objects in 3DS MAX simulate cameras by projecting geometry onto the view-plane defined by the camera's position and orientation in the scene. The main parameter for a camera is its field of view, which determines how much of the view-plane is visible to the camera. There are two kinds of camera objects in 3DS MAX: target cameras and free cameras

- Target camera – view the area around a target object. They are easier to use when the camera does not move along a path while rendering a scene or animation.

- Free camera – view the area in the direction the camera is aimed. Free cameras are easier to use when the camera's position is animated along a path. They can bank as they travel the path, which Target cameras cannot.

Animation is based on a principle of human vision. If a person view a series of related still images in quick succession, then he will perceive them as continuous motion. Each individual image is referred to as a frame. Most of the frames in an animation are routine, incremental changes from the previous frame directed toward some goal. Important frames or *keyframes* will be created by master animator and the in-between frames or *tweens* will be created by assistants. A similar concept is used in 3DS MAX which acts as an animation assistant. If the user creates keyframes that record the beginning and end of each animated sequence, 3DS MAX automatically calculates the values for the in-between frames to produce the complete animation. 3DS MAX can animate just about any parameter in the screen. Modifier parameters such as a Bend angle or a Taper amount, material parameters, such as colour or transparency of the object and much more can be animated. Once the animation parameters are specified 3DS MAX renderer takes over the job of shading and rendering each frame resulting a high quality animation. Animation in 3DS MAX is a time based one and the time can be reconfigured to a format best suited to the work. Various animation tools, control features and modelling features that are not available in standard 3DS MAX can be developed using 3DS MAX plug-in software development kit (MAXSDK). Those features are developed as special plug-ins, which are small software programs written in C++ language using specially defined classes and functions. These classes and functions are defined by 3DS MAX itself. When the plug-ins are developed they can be added as an additional feature in 3DS MAX window application.

3.3 Limitations

3DS MAX is a single document application and to generate the model the user needs to work in 3DS MAX which is compatible only in few operating systems. Even though it has animation features these features can't be customised easily and they are not flexible enough to accommodate any kind of control or modelling mechanism. Any custom feature has to be developed as a plug-in using MAX SDK which not only requires programming knowledge and expertise in C++ language but also a clear knowledge and understanding of various classes and functions defined by SDK. 3DS MAX manages graphics interfaces and memory very well but falls short of providing a flexible environment generating the model dynamically with various motion and view controls. As such 3DS MAX occupies a good amount of hard disk space and the user needs to install a copy of 3DS MAX if he wants to generate the model in a different PC. 3DS MAX can generate a pre-defined and preplanned animation but it lacks features to animate the scene dynamically.

So softwares like 3DS MAX, VRTSuperscape etc., offers either high-end modelling, animation, and graphics management but lacks the flexibility in terms of developing customised controls dynamic model generation and dynamic on screen effects. Regarding compatibility also these aren't that flexible enough to run in any operating system. This leads to exploration of features and advantages of graphics libraries supported by computer languages.

3.4 Graphics standards

Since the purpose of computer graphics is the creation and manipulation of graphic scenes, it is important to be able to evaluate and modify these scenes in an interactive, fast-responsive way. Interactive computer graphics is needed, therefore, to give human control to the graphics process. One of the major requirements for a computer graphics system is that applications should be portable to any physical system and should be developed without hardware in mind. To attain this portability, standardisation of the graphics environment at the functional level is necessary, providing language and device independence.

Computer graphics (especially 3D graphics, and interactive 3D graphics in particular) is finding its way into an increasing number of applications, from simple graphing programs for personal computers to sophisticated modelling and visualization software on workstations and supercomputers. As the interest in computer graphics has grown, so has the desire to be able to write an application so that it runs on a variety of platforms with a range of graphical capabilities. A graphics standard eases this task by eliminating the need to write a distinct graphics driver for each platform on which the application is to run.

Several standards have succeeded in integrating specific domains of 2D graphics. The PostScript page description language [57] has become widely accepted, making it relatively easy to electronically exchange, and, to a limited degree, manipulate static documents containing both text and 2D graphics. The X window system [58] has become standard for UNIX workstations. A programmer uses X to obtain a window on a graphics display into which either text or 2D graphics may be drawn; X also provides a standard means for obtaining user input from such devices as keyboards and mice. The adoption of X by most workstation manufacturers means that a single program can produce 2D graphics or obtain user input on a variety of workstations by simply recompiling the program. This integration even works across a network: the program may run on one workstation but display on and obtain user input from another, even if the workstations on either end of the network are made by different companies.

For 3D graphics, several standards have been proposed, but none has (yet) gained wide acceptance. One relatively well-known system is PHIGS (Programmer's Hierarchical Interactive Graphics System). Based on GKS [59] (Graphics Kernel System), PHIGS is an ANSI (American National Standards Institute) standard. PHIGS (and its descendant, PHIGS+ [60]) provides a means to manipulate and draw 3D objects by encapsulating object descriptions and attributes into a display list that is then referenced when the object is displayed or manipulated. One advantage of the display list is that a complex object need be described only once even if it is to be displayed many times. This is especially important if the object to be displayed must be transmitted across a low-bandwidth channel (such as a network). One disadvantage of a display list is that it can require considerable effort to re-specify the object if it is being continually modified as a result of user interaction. Another difficulty with PHIGS and PHIGS+ (and with GKS) is that they lack support for advanced rendering features such as texture mapping.

Figure 3.1 shows the model for standardising the graphics environment. The programming language interface level in this model specifies the boundary between an application program and a graphics support package.

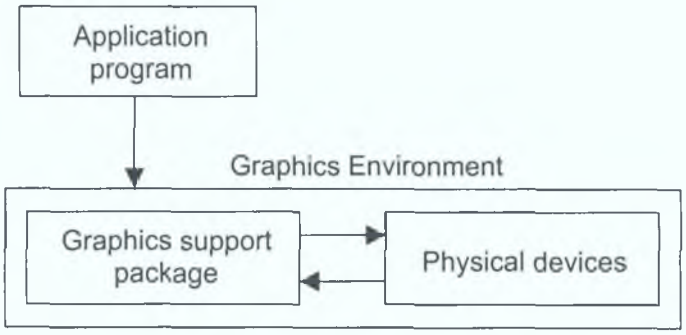


Figure 3.1 Model for standardisation of graphics environment

It establishes the language bindings to various high-level languages making the functions in the graphics library appear to the programmer to the programmer like standard library functions.

3.5 Device independence

Device independence allows a graphics application program to run on hardware of various types. This is accomplished through "logical" input and output devices available to the application software through the graphics support package, and mapped to the actual physical devices at execution time. The major portion of the standardization effort at the application programmer's level is found within the computer graphics support package. Several graphics standards have been developed over the years, including CORE (1977, revised 1979) and GKS(graphics kernel system,1984-85), developed particularly to address the need for standardized two-dimensional input and output. Some of these became official standards, accepted by the American National Standards Institute (ANSI), the International Standards Organization (ISO), and others [61]. These first efforts were followed by new standards addressing the needs of three-dimensional applications with a high level of interactivity. GKS-3D added three-dimensional capabilities to the existing standards.

To achieve portability, the standards establish for an application program minimize changes that allows it to address various input or output devices. Initially, the programmer establishes a modelling coordinate system in which the object is described, usually referred as world coordinate system. Two-dimensional square areas in the viewing surfaces on which the image will appear define the normalised device coordinate system. The normalised device coordinates are subsequently transformed to device coordinates. The application programmer

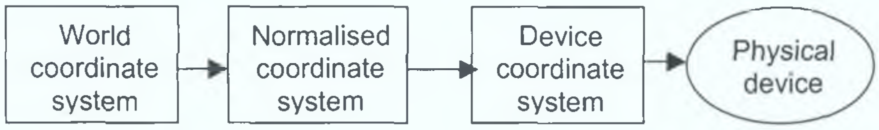


Figure 3.2 Device independence in the process of picture creation

interacts with the normalised coordinate system in a consistent manner irrespective what physical output device is used.

PHIGS (Programmer Hierarchical Graphics System, 1984) and PHIGS+ include more powerful three-dimensional graphics functions and the ability to interactively create and manipulate complex graphics data. Graphics standards have also resulted from industry acceptance of specific interfaces proposed by various companies and not available within the official standards mentioned above. Notable among these is a system like X-Windows, a window management program, which, in addition to creating and manipulating variable size windows, supports a variety of input functions and two-dimensional graphics operations. A three-dimensional extension of the X-windows system was started in 1987 and named PEX, supporting various three-dimensional graphics capabilities.

Also nonofficial industry standards have also proven to be important for interactive graphics and this includes Silicon Graphics OpenGL [62]. Even though there are many differences between PHIGS PLUS and the likes of OpenGL, they are far more alike, at least in their basic capabilities if not in their procedural Application Program Interface (API).

Standards like GKS and PHIGS make use of several commonly used graphics functions that are typically accessed through a collection of linkable library object programs. The standards attempt to describe an all-inclusive set of graphics functions, but it is obvious that any given set may be insufficient for some specific application.

To achieve device independence, the graphics standard must be able to receive and send data from/to various input and output devices. To accomplish this task, the standards make use of the concept of a logical workstation.

A logical workstation [63] is an abstract graphics device that provides a logical interface between the application program and the physical device. The term is different from "graphics workstation" or normal workstation. A logical workstation can be of 3 types

- Input only – at least one logical input device and no output capability.
- Output only – only display area available with no input capability.

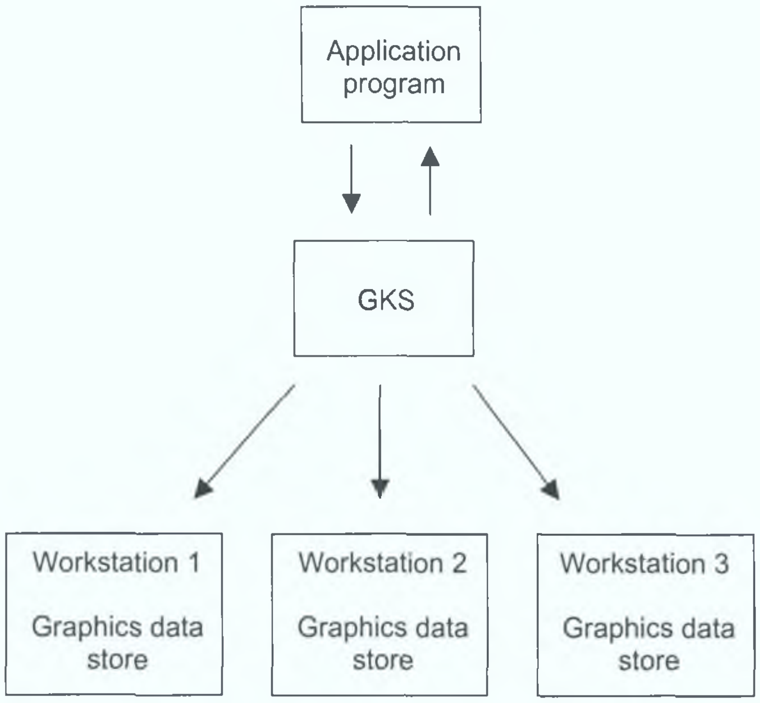


Figure 3.3 GKS stores graphics data at the workstation level

- Input/output – combining the above characteristics.

The concept of the logical workstation is embedded in both GKS and PHIGS. However, each standard handles the storage of graphics data used by the workstation differently. Graphics data are different from application model data that are maintained by the application program in a form suitable for its manipulation. The graphics data are maintained by graphical support package in a form suitable for graphics manipulation and rendering. GKS stores graphics data at the workstation level.

3.6 PHIGS and SPHIGS

PHIGS stores information in a special purpose database called the central structure storage (CSS) and from there the data are sent to individual logical workstations. A structure in PHIGS is sequence of elements-primitives, appearance attributes, transformation matrices, and invocations of subordinate structures-whose purpose is to define a coherent geometric object [64]. Thus, PHIGS effectively stores multipurpose modelling hierarchy, complete with modelling transformations and other attributes passed as parameters to subordinate structures. PHIGS may be viewed as the specification of a device-independent hierarchical display list package; a given implementation is, of course, optimised for a particular display device, but the application programmer need not be concerned with that. Many PHIGS implementations are purely software; the most common arrangement is to do CSS manipulation in software and to use a combination of hardware and software rendering.

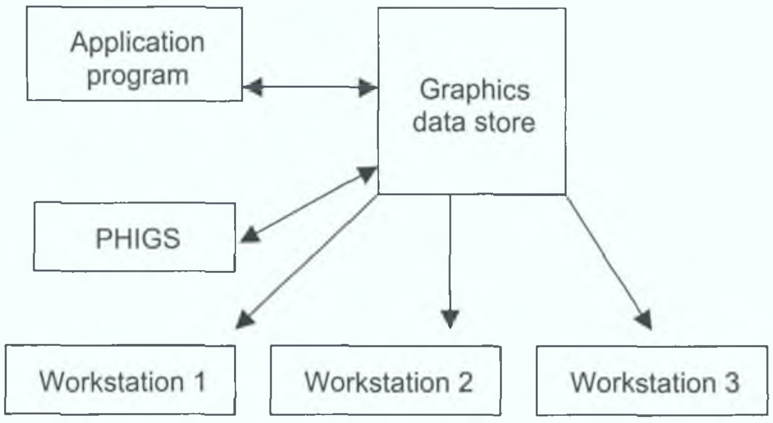


Figure 3.4 PHIGS stores graphics data at a centralized location

PEX [65], which is often said to be an acronym for PHIGS Extension to X, extends X to include the ability to manipulate and draw 3D objects. (PEXlib [66] is the programmer's interface to the PEX protocol.), Among other extensions, PEX adds *immediate mode* rendering to PHIGS, meaning that objects can be displayed as they are described rather than having to first complete a display list. One difficulty with PEX has been that different suppliers of the PEX interface have chosen to support different features, making program portability problematic. PEX also lacks advanced rendering features, and is available only to users of X.

SPHIGS (Simple PHIGS) is a subset of PHIGS. It preserves most of PHIGS's capabilities and power, but simplifies or modifies various features to suit straightforward applications. SPHIGS also has several enhancements adapted from PHIGS PLUS extensions. SPHIGS has been designed to introduce concepts in simplest possible way, not to provide a package that is strictly upward compatible with PHIGS. There are three major differences between SPHIGS and integer raster packages such as SRGP(Simple Raster Graphics Package) or Xlib package of the X window system. First, to suit engineering and scientific applications, SPHIGS uses a 3D floating-point coordinate system and implements the 3D viewing pipeline. The second, further-reaching difference is that SPHIGS maintains a database of structures. A structure is logical grouping of primitives, attributes and other information. Structures contain not only specifications of primitives and attributes, but also invocations of subordinate structures. Structures thus exhibit some of the properties of procedures in programming languages. In particular, just as procedure hierarchy is induced by procedures invoking subprocedures, structure hierarchy is induced by structures invoking substructures. Such hierarchical composition is especially powerful when one can control the geometry (position, orientation, size) and appearance (colour, style, thickness, etc.,) of any invocation of a substructure. The third difference is that SPHIGS operates in an abstract, 3D world coordinate system, not in 2D screen space, and therefore does not support direct pixel manipulation. Because of these differences, SPHIGS and SRGP address different sets of needs and applications and no single graphics package meets all needs.

3.7 OpenGL

3.7.1 Introduction

OpenGL (``GL" for ``Graphics Library") provides advanced rendering features in either immediate mode or display list mode. While OpenGL is a relatively new standard, it is very similar in both its functionality and its interface to Silicon Graphics' IRIS GL, and there are many successful 3D applications that currently use IRIS GL for their 3D rendering [62].

Like the graphics systems already discussed, OpenGL is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically colour images of three-dimensional objects. Like PEX, OpenGL integrates 3D drawing into X, but can also be integrated into other window systems (e.g. Windows/NT) or can be used without a window system.

OpenGL draws primitives into a framebuffer subject to a number of selectable modes. Each primitive is a point, line segment, polygon, pixel rectangle, or bitmap. Each mode may be changed independently; the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer). Modes are set, primitives specified, and other OpenGL operations described by sending commands in the form of function or procedure calls [41].

Geometric primitives (points, line segments, and polygons) are defined by a group of one or more vertices. A vertex defines a point, an endpoint of an edge, or a corner of a polygon where two edges meet. Data (consisting of positional coordinates, colours, normals, and texture coordinates) are associated with a vertex and each vertex is 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 the indicated 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.

OpenGL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. It does not provide a means for describing or modelling complex geometric objects. Another way to describe this situation is to say that OpenGL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves.

The effects of OpenGL commands on the framebuffer are ultimately controlled by the window system that allocates framebuffer resources. It is the window system that determines which portions of the framebuffer that OpenGL may access at any given time and that communicates to OpenGL how those portions are structured. Similarly, display of framebuffer contents on a CRT monitor (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by OpenGL. Framebuffer configuration occurs outside of OpenGL in conjunction with the window system; the initialisation of an OpenGL context occurs when the window system allocates a window for OpenGL rendering. Additionally, OpenGL has no facilities for obtaining user input, since it is expected that any window system under which OpenGL runs, must already provide such facilities. These considerations make OpenGL independent of any particular window system.

Since its first release in 1992, OpenGL has been rapidly adopted as the graphics API of choice for real-time interactive 3D graphics applications. 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. The OpenGL state machine is easy to understand, but its simplicity and orthogonality enable a multitude of interesting effects. OpenGL is a procedural rather than descriptive interface. In order to generate, for example rendering of a geometric primitive, the programmer must specify the appropriate sequence of commands to set up the camera view and modelling transformations, draw the geometry with specific colour, etc. Other systems such as VRML (virtual reality modelling language) are descriptive [67]; The user has to simply specify that an object of desired shape and size, for example, a blue coloured cube should be drawn at certain coordinates with out worrying about many parameters such as window management, screen depth, memory management, floating point precision and most importantly the way the object is constructed and rendered. The disadvantage of using a procedural interface is that the application must specify all of the operations in exacting detail and in the correct sequence to get the desired result. The advantage of this approach is that it allows great flexibility in the process of generating the image.

Different features of OpenGL functionality can be combined as building blocks to create innovative techniques and produce new graphics capabilities. One of the advantages in OpenGL is that (it's procedures and or commands) it is not pixel specific. This allows OpenGL to be implemented across a range of hardware platforms.

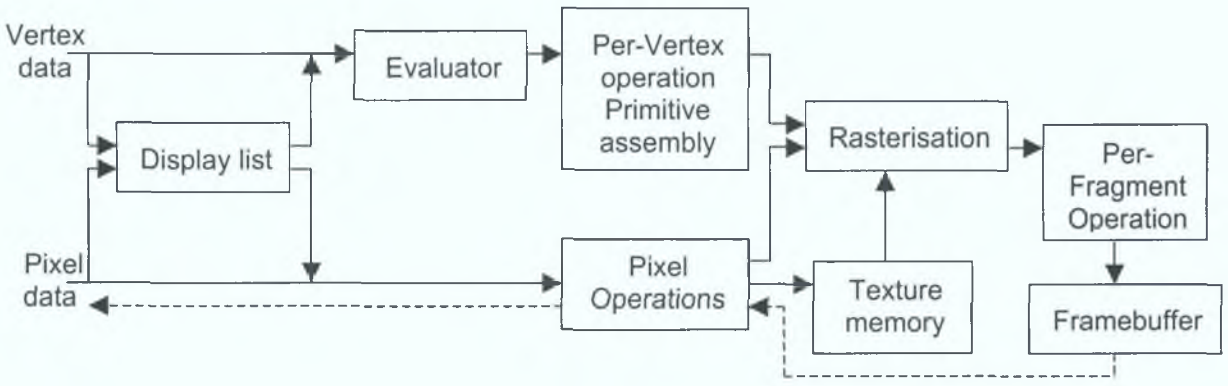


Figure 3.5 OpenGL schematic diagram

OpenGL draws primitives into a framebuffer subject to a number of selectable modes. Each primitive is a point, line segment, polygon, pixel rectangle, or bitmap. Each mode may be changed independently; the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer). Modes are set, primitives specified, and other OpenGL operations described by issuing commands in the form of function or procedure calls.

Figure 3.5 shows a schematic diagram of OpenGL. Commands enter OpenGL on the left. Most commands may be accumulated in a display list for processing at a later time. Otherwise, commands are effectively sent through a processing pipeline.

The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values. The next stage operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices are transformed and lit, and primitives are clipped to a viewing volume in preparation for the next stage, rasterisation. The rasteriser produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced represents a portion of a primitive that corresponds to a pixel in the framebuffer. Then each fragment may be modified by texture mapping, after which it is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colours with stored colours, as well as masking and other logical operations on fragment values.

Finally, pixel rectangles and bitmaps (2D images) bypass the vertex processing portion of the pipeline to send a block of fragments directly through rasterisation to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer. A unique feature of OpenGL is that pixel rectangles and bitmaps (2D images) are also rasterised to produce fragments; fragments are treated the same no matter if they come from a geometric or image primitive. Values may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

3.7.2 Compatibility

OpenGL is supported on all UNIX workstations, and normally included with every Windows 95/98/2000/NT and MacOS PC, no other graphics API operates on a wider range of hardware platforms and software environments. OpenGL runs on every major operating system including Mac OS, OS/2, UNIX, Windows 95/98, Windows 2000, Windows NT, Linux, OPENStep, and BeOS; it also works with every major windowing system, including Win32, MacOS, Presentation Manager, and X-Window System. OpenGL is callable from Ada, C, C++, Fortran, Python, Perl and Java and offers complete independence from network protocols and topologies [68].

3.7.3 Performance

A fundamental consideration in interactive 3D graphics is performance. Numerous calculations are required to render a 3D scene of even modest complexity, and in an interactive application, a scene must generally be redrawn several times per second. An API for use in interactive 3D applications must therefore provide efficient access to the capabilities of the graphics hardware of which it makes use. But different graphics subsystems provide different capabilities, so a common interface must be found.

The interface must also provide a means to switch on and off various rendering features. This is required both because some hardware may not provide support for some features and so cannot provide those features with acceptable performance, and also because even with hardware support, enabling certain features or combinations of features may decrease performance significantly. Slow rendering may be acceptable, for instance, when producing a final image of a scene, but interactive rates are normally required when manipulating objects within the scene or adjusting the viewpoint. In such cases the performance-degrading features may be desirable for the final image, but undesirable during scene manipulation.

3.7.4 Device independence or Low level

An essential goal of OpenGL is to provide device independence while still allowing complete access to hardware functionality. The API therefore provides access to graphics operations at the lowest possible level that still provides device independence. As a result, OpenGL does not provide a means for describing or modelling complex geometric objects. Another way to describe this situation is to say that OpenGL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves.

OpenGL's emphasis on low-level functionality arises partly from the desire to expose as much performance of any underlying graphics hardware as is feasible. Sometimes this emphasis detracts from ease-of-use. OpenGL's error checking, for instance, is rudimentary so as not to burden implementations with potentially slow checks that would not be needed in a working program. A debugger is provided with OpenGL that can monitor commands and flag errors or suspicious conditions. Toolkits built on top of OpenGL also provide sophisticated error checking as well as other easy-to-use features that were consciously omitted from OpenGL.

As a software interface for graphics hardware, OpenGL main purpose is to render two and three-dimensional objects into frame buffer. These objects are described as sequences of vertices or pixels. OpenGL performs several processing steps on this data to convert it to pixels to form the final desired image in the framebuffer. OpenGL draws geometric primitives such as points, lines, and polygons etc., which are defined by a group of one or more vertices. A vertex is defined as a point, may be a corner of a polygon or endpoint of a line. A vertex has associated data such as vertex coordinates, colours, normals, texture coordinates, and edge flags. Each vertex and its associated data are processed independently, in order, in the same way. The only exception to this rule is if the particular primitive doesn't fit with the specified region or layout then the group of vertices will be clipped so that the particular primitive fits within a specified region. In this case the vertex data may be modified and new vertices are 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 intermediate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent effect takes effect. It also means that state querying commands return data that's consistent with complete execution of all previously issued OpenGL commands.

3.7.5 Descriptive programming Vs Procedural programming

OpenGL provides a fairly direct control over the fundamental operations of two and three-dimensional graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators [41]. However it does not provides the means for describing or modelling complex geometric objects. The OpenGL commands will specify how a certain result should be produced rather than what exactly that result should look like. That is, OpenGL is fundamentally procedural rather than descriptive. Because of this procedural nature, it helps to know how OpenGL works, the order in which it carries out its operations so that the user can fully understand how to use it.

3.7.6 Execution model

The model for interpretation of OpenGL commands is client-server. An application (the client) issues commands, which are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client. In this sense, OpenGL is network-transparent. A server can maintain several GL contexts, each of which is an encapsulated GL state. A client can connect to any one of these contexts. The required network protocol can be implemented by augmenting an already existing protocol or by using an independent protocol. The effects of OpenGL commands on the frame buffer are ultimately controlled by window system that allocates frame buffer resources. The window system determines which portions of the frame buffer OpenGL may access at any given time and communicates to OpenGL how those portions are structured. Therefore, there are no OpenGL commands to configure the frame buffer or initialise OpenGL. Frame buffer configuration is done outside of OpenGL in conjunction with the window system. OpenGL initialisation takes place when the window system allocates a window for OpenGL rendering.

3.7.7 Antialiasing

Antialiasing is a technique for reducing the jagged effect created when only portions of neighboring pixels properly belong to the image being drawn. Such jaggies are usually the most visible with near-horizontal or near-vertical lines.

3.7.8 Rendering

Rendering describes the overall process of going from a database representation pf a three-dimensional object to a shaded two-dimensional projection on a view surface[69]. These models or objects are constructed from geometric primitives that are specified by their vertices. The final rendered image consists of pixels drawn on the screen. A *pixel* (picture element) is the smallest visible element the display hardware can put on the screen. Information about the pixels is organized in system memory in bitplanes. A bitplane is an area of memory that holds one bit of information for every pixel on the screen. For instance, the bit might indicate how blue a particular pixel is supposed to be. The bitplanes are themselves organized into a frame buffer, which holds all the information that the graphics display needs to control the intensity of all the pixels on the screen.

3.7.9 OpenGL state machine

OpenGL is a state machine. It is possible to put it into various states that they remain in effect until the changes are done manually or by the application. For instance, current colour is a state variable. It can be set to any colour and thereafter every object is drawn with that colour until it is set to a different colour. There are many state variables like current colour, which are preserved by OpenGL. Many state variables refer to modes that are enabled or disabled with the command *glEnable()* or *glDisable()*. Each state variable or mode has a default value, and at any point it can be queried for each variable's current value.

3.7.10 OpenGL processing pipeline:

Figure 3.6 shows the more detailed block diagram of the OpenGL processing pipeline. For most of the pipeline there are three vertical arrows between major stages.

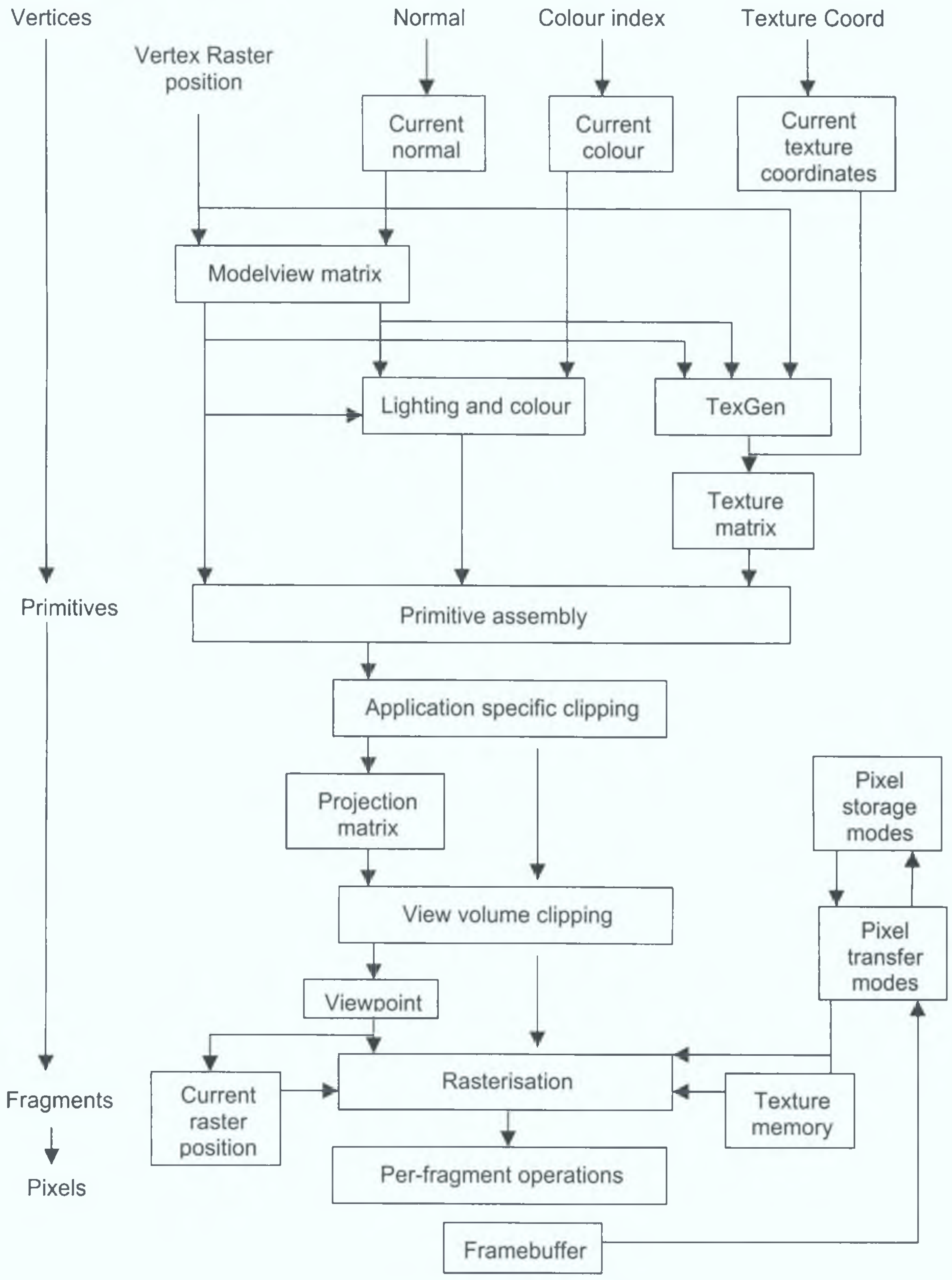


Figure 3.6 Processing pipeline

These arrows represent vertices and the two primary types of data that can be associated with vertices namely the colour values and texture coordinates. The vertices are assembled into primitives, then to fragments, and finally to pixels in the frame buffer. Many OpenGL commands are simple variations of each other, differing mostly in the data type of arguments; some commands differ in the number of related arguments and whether those arguments can be specified as a vector or whether they must be specified separately in a list.

The effect of an OpenGL command may vary depending on whether certain modes are enabled. For example, lighting has to enabled if the lighting-related commands are to have the desired effect of producing a properly lit object.

3.7.11 Vertex and primitives

In OpenGL, most geometric objects are drawn by enclosing a series of coordinate sets that specify vertices and optionally normals, texture coordinates, and colours between *glBegin* and *glEnd* command pairs.

Each vertex may be specified with two, three, or four coordinates (four coordinates indicate a homogeneous three-dimensional location). In addition, a current normal, current texture coordinates, and current colour may be used in processing each vertex. OpenGL uses normals in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Colour may consist of either red, green, blue, and alpha values (when OpenGL has been initialised to RGBA mode) or a single colour index value (when initialisation specified colour index mode). One, two, three, or four texture coordinates determine how a texture image maps onto a primitive.

Each of the commands that specify vertex coordinates, normals, colours, or texture coordinates comes in several types to accommodate differing application's data formats and numbers of coordinates [41]. Data may also be passed to these commands either as an argument list or as a pointer to a block of storage containing the data. Most OpenGL commands that do not specify vertices and associated information may not appear between *glBegin* and *glEnd*.

This restriction allows implementations to run in an optimised mode while processing primitive specifications so that primitives may be processed as efficiently as possible. When a vertex is specified, the current colour, normal, and texture coordinates are used to obtain values that are then associated with the vertex (Figure 3.7).

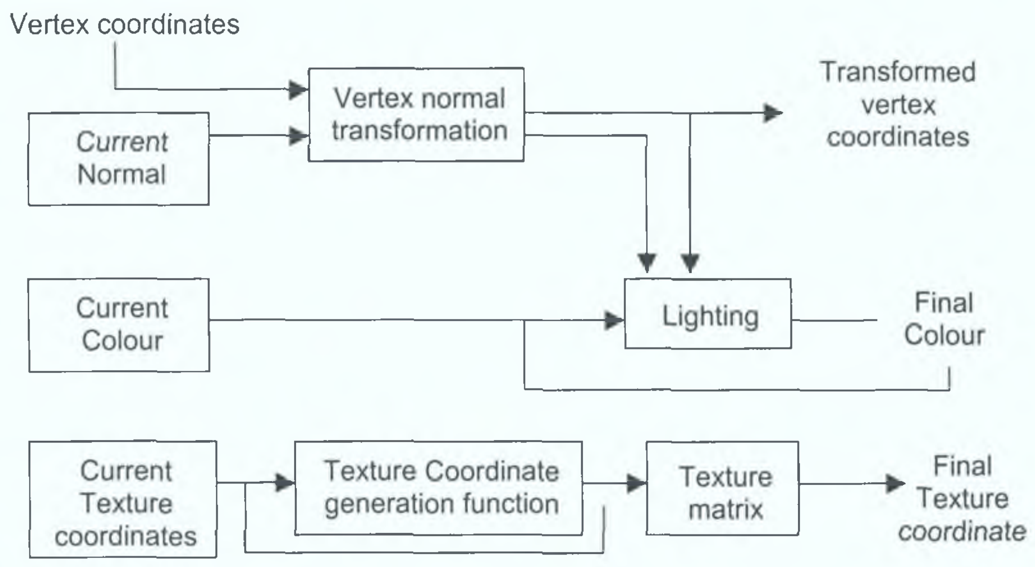

Vertex coordinates
Current Normal
Vertex normal transformation
Transformed vertex coordinates
Current Colour
Lighting
Final Colour
Current Texture coordinates
Texture Coordinate generation function
Texture matrix
Final Texture coordinate

Figure. 3.7 Association of current value with vertex

The vertex itself is transformed by the model-view matrix, a matrix which can represent both linear and translational transformations. The colour is obtained from either computing a colour from lighting or, if lighting is disabled, from the current colour. Texture coordinates are similarly passed through a texture coordinate generation function. The texture coordinates obtained from the function or from current texture coordinates are transformed by the texture matrix (this matrix may be used to effectively scale or rotate a texture that is applied to a primitive).

A number of commands control the values of parameters used in processing a vertex. One group of commands manipulates transformation matrices; these commands are designed to form an efficient means for generating and manipulating the transformations that occur in hierarchical 3D graphics scenes. A matrix may be loaded or multiplied by a scaling, rotation, translation, or general matrix. Another command controls which matrix is affected by a manipulation: the model-view matrix, the texture matrix, or the projection matrix (to be described presently). Each of these three matrix types also has an associated stack onto which matrices may be pushed or popped.

Lighting parameters are grouped into three categories: material parameters, that describe the reflectance characteristics of the surface being lit, light source parameters, that describe the emission properties of each light source, and lighting model parameters, that describe global properties of the lighting model [70]. Lighting is performed on a per-vertex basis; lighting results are eventually interpolated across a line segment or polygon. The general form of the lighting equation includes terms for constant, diffuse, and specular illumination, each of which may be attenuated by the distance of the vertex from the light source. A programmer may sacrifice realism in favour of faster lighting calculations by indicating that the viewer, the light sources, or both should be assumed to be infinitely far from the scene.

3.7.12 Clipping

Points, line segments and polygons are handled slightly differently during clipping. Points are either retained in their original state or discarded. If portions of line segments or polygons are outside the clip volume, new vertices are generated at the clip points. For polygons, an entire edge may need to be constructed between such new vertices. For both line segments and polygons that are clipped, the edge flag, colour and texture information is assigned to all new vertices.

3.7.13 Rasterisation

Rasterisation produces a series of frame buffer addresses and associated values using a two dimensional description of a point. Rasterisation converts a projected, viewport-scaled primitive into a series of fragments. Each fragment comprises a location of a pixel in the framebuffer along with colour, texture coordinates, and depth(z) [69]. When a line segment or polygon is rasterised, these associated data are interpolated across the primitive to obtain a value for each fragment. The rasterisation of each kind of primitive is controlled by a corresponding group of parameters. One width affects point rasterisation and another affects line segment rasterisation. Additionally, a stipple sequence may be specified for line segments, and a stipple pattern may be specified for polygons.

3.7.14 Framebuffer

The destination of rasterised fragments is the framebuffer, where the results of OpenGL rendering may be displayed. In OpenGL, the framebuffer consists of a rectangular array of pixels corresponding to the window allocated for OpenGL rendering. Each pixel is simply a set of some number of bits. Corresponding bits from each pixel in the framebuffer are grouped together into a bitplane; each bitplane contains a single bit from each pixel.

The bitplanes are grouped into several logical buffers: the colour, depth, stencil, and accumulation buffers. The colour buffer is where fragment colour information is placed. The depth buffer is where fragment depth information is placed, and is typically used to effect hidden surface removal through z -buffering. The stencil buffer contains values each of which may be updated whenever a corresponding fragment reaches the framebuffer. Stencil values are useful in multi-pass algorithms, in which a scene is rendered several times, to achieve such effects as CSG (union, intersection, and difference) operations on a number of objects and capping of objects sliced by clip planes.

The accumulation buffer is also useful in multipass algorithms; it can be manipulated so that it averages values stored in the colour buffer. This can have effects such as full-screen anti-aliasing (by jittering the viewpoint for each pass), depth-of-field (by jittering the angle of view), and motion blur (by stepping the scene in time) [71]. Multi-pass algorithms are simple to implement in OpenGL, because only a small number of parameters must be manipulated before each pass, and changing the values of these parameters is both efficient and without side effects on the values of other parameters that must remain constant.

OpenGL supports both double buffering and stereo, so the colour buffer is further subdivided into four buffers: the front left & right buffers and the back left & right buffers. The front buffers are those that are typically displayed while the back buffers (in a double-buffered application) are being used to compose the next frame. A monoscopic application would use only the left buffers. In addition, there may be some number of auxiliary buffers (these are never displayed) into which fragments may be rendered. Any of the buffers may be individually enabled or disabled for fragment writing.

3.7.15 Fragments

OpenGL allows a fragment produced by rasterisation to modify the corresponding pixel in the framebuffer only if it passes a series of tests like Pixel ownership test, scissor test, alpha test, stencil test and depth buffer test. If it does pass, the fragment's data can be used directly to replace the existing frame buffer values, or it can be combined with existing data in the framebuffer, depending on the state of certain nodes.

3.7.16 OpenGL invariance

OpenGL is not a pixel-exact specification. It therefore does not guarantee an exact match between images produced by different OpenGL implementations. However, OpenGL does specify exact matches, in some cases, for images produced by the same implementation.

The obvious and most fundamental case is repeatability. A conforming OpenGL implementation generates the same results each time a specific sequence of commands is issued from the same initial conditions. Although such repeatability is useful for testing and verification, it's often not useful to application programmers, because it is difficult to arrange for equivalent initial conditions. For example, rendering a scene twice, the second time after swapping the front and back buffers, doesn't meet this requirement. So repeatability cannot be used to guarantee a stable, double-buffered image.

A simple and useful algorithm that counts on invariant execution is erasing a line by redrawing it in the background colour. This algorithm works only if rasterising the line results in the same fragment (x, y) pairs being generated in both the foreground and background colour cases. OpenGL requires that the coordinates of the fragments generated by rasterisation be invariant with respect to framebuffer contents, which colour buffers are enabled for drawing, the values of matrices other than those on the top of the matrix stacks, the scissor parameters, all writemasks, all clear values, the current colour, index, normal, texture coordinates, and edge-flag values, the current raster colour, raster index, and raster texture coordinates, and the material properties. It is further required that exactly the same fragments be generated, including the fragment colour values, when framebuffer contents, colour buffer enables, matrices other than those on the top of the matrix stacks, the scissor parameters, writemasks, or clear values differ.

OpenGL further suggests, but doesn't require, that fragment generation be invariant with respect to the matrix mode, the depths of the matrix stacks, the alpha test parameters (other than alpha test enable), the stencil parameters (other than stencil enable), the depth test parameters (other than depth test enable), the blending parameters (other than enable), the logical operation (but not logical operation enable), and the pixel-storage and pixel-transfer parameters. Because invariance with respect to several enables isn't recommended, you should use other parameters to disable functions when invariant rendering is required. For example, to render invariantly with blending enabled and disabled, set the blending parameters to GL_ONE and GL_ZERO to disable blending, rather than calling *glDisable*(GL_BLEND). Alpha testing, stencil testing, depth testing, and the logical operation can all be disabled in this manner.

Finally, OpenGL requires that per-fragment arithmetic, such as blending and the depth test, be invariant to all OpenGL state except the state that directly defines it. For example, the only OpenGL parameters that affect how the arithmetic of blending is performed are the source and destination blend parameters and the blend enable parameter. Blending is invariant to all other state changes. This invariance holds for the scissor test, the alpha test, the stencil test, the depth test, blending, dithering, logical operations, and buffer writemasking.

As a result of all these invariance requirements, OpenGL can guarantee that images rendered into different colour buffers, either simultaneously or separately using the same command sequence, are pixel identical. This holds for all the colour buffers in the framebuffer, or all the colour buffers in an off-screen buffer, but it isn't guaranteed between the framebuffer and off-screen buffers

3.6 Model construction techniques, efficiency and framework for the software

Complex virtual environments such as games, urban visualization, town planning, flight simulators etc., are constructed from large amounts of data and are made of numerous polygons. Some of the complex environments created using the geometric data may well exceed the interactive visualization capabilities of current graphics systems. Some of the applications in virtual reality needs a cluster of PC's and displays [72] to create the VR effect. This leads to the simplification of data, which means simplifying the amount of data (level of detail) [22] available to define an object in the environment. Since the environment is constructed using polygons, polygon simplification techniques offer one solution for programmers. Several algorithms [73,74] are devised to simplify polygons based on various ideas like culling invisible face, view frustum culling [23], vertex clustering (vertex merging) etc., and it is also dependent on the view. So the developer has to minimize the levels of detail (LOD) and also the number of details to render the scene efficiently. Defining the details in simple format or structure is one of the better ways to maximize the rendering of the hardware/software of the graphics system. The framework for the proposed software, 3D*gen*, which generates the model or scene based on the geometric data by constructing numerous polygons. A delicate balance has to be achieved by minimising the amount of data used to define the model or the scene and also achieving a quality image of the model. OpenGL offers a wide variety of features in terms of compatibility, graphics management, window management and is flexible enough to use various procedures to achieve desired framerate. This coupled with C language (low level language) proves to be a powerful combination for development of software and various associated tools.

CHAPTER 4

CHAPTER 4: OVERVIEW AND ALGORITHM OF SOFTWARE

4.1 Introduction

Three-dimensional images can be created by many ways and the most widely used techniques are model reconstruction from photo-realistic images and three-dimensional modelling from available data. The available data may be extracted from the image or given by the user or the programmer. '3D*gen*' software is used to create three-dimensional cylindrical models using data's obtained from some other application or given by the user. Figure 4.1 shows the model, which can be generated by the software.

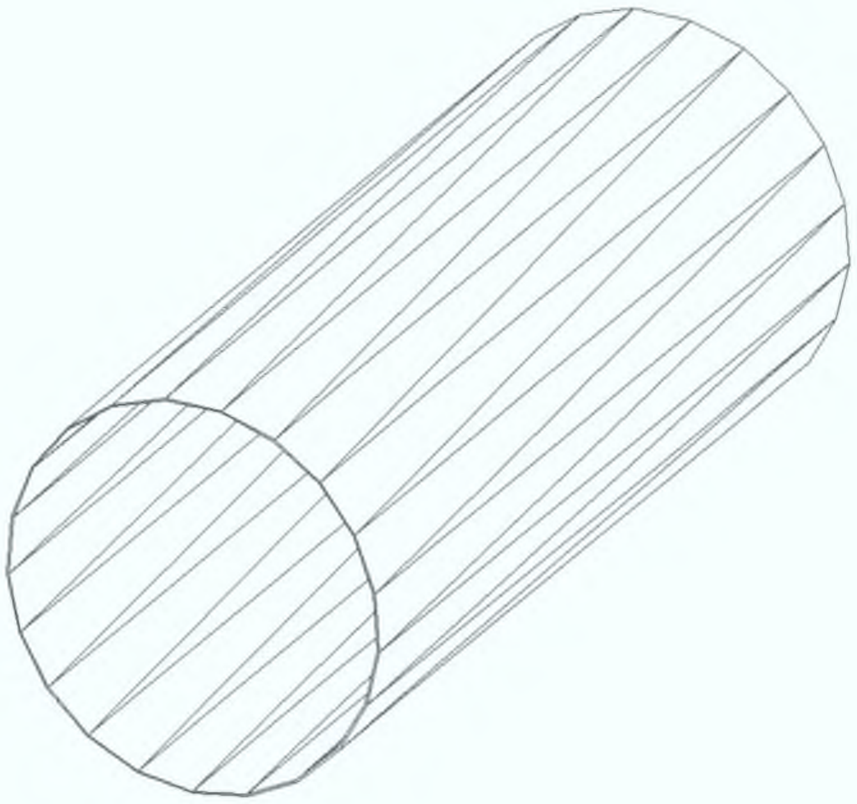


Figure 4.1 Model section

The model, which is a hollow cylindrical section, can be extended to any length depending upon the requirement. In this chapter the algorithm of software and the various steps of execution process are discussed.

The flowchart of the main part of the software is as shown in figure 4.2. The main loop starts with the initialisation of the window parameters such as display mode size and position and then the windows were opened. Next, the menu options were initialised and the input data file is prepared for periodical access.

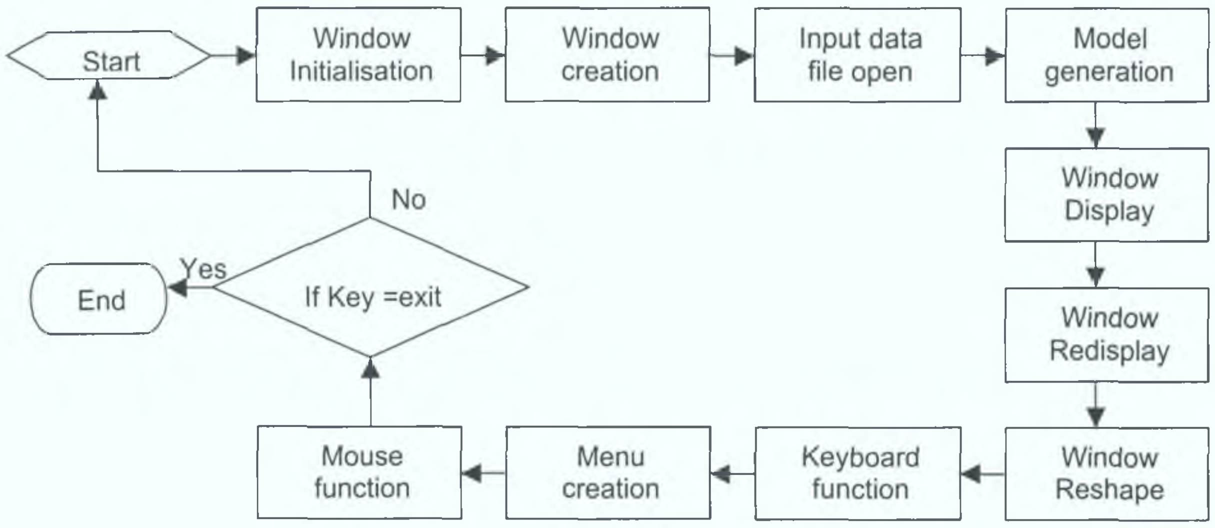


Figure 4.2 Flowchart of the main function

Next, the model generating procedures were executed followed by window display procedures, which actually displays the generated model in the window. This was followed by window resize procedures, i.e., the procedures that has to be executed when the user changes the size of the window. Keyboard input function and mouse input function checks whether any keys in the keyboard and mouse were pressed to execute appropriate option. The main() function will be executed continuously until the user chooses to exit the software by exit option in menu. The input data file for generating the model was opened in "read-only" mode to avoid any loss of data. The full source code of 3D*gen* software is available in Appendix D.

4.2 Execution Process

The various steps in the execution process of the software is as shown in Figure 4.3.The first step in the execution process is the calculation of the vertices for the polygons from the data fetched by data file. These vertices are along the circumference of the faces of the model. The vertices are then used to plot polygons around the circumference of faces whose data are obtained from data file. The angular displacement of the points along the circumference of the face or circle is kept at 1°. Normals were calculated which is essential to generate proper lighting and corresponding shades in the model. The model is displayed in the window and the movement in the camera were updated dynamically. The image or window refresh rate depends on the size of the display memory available in the PC and also the speed of the processor.

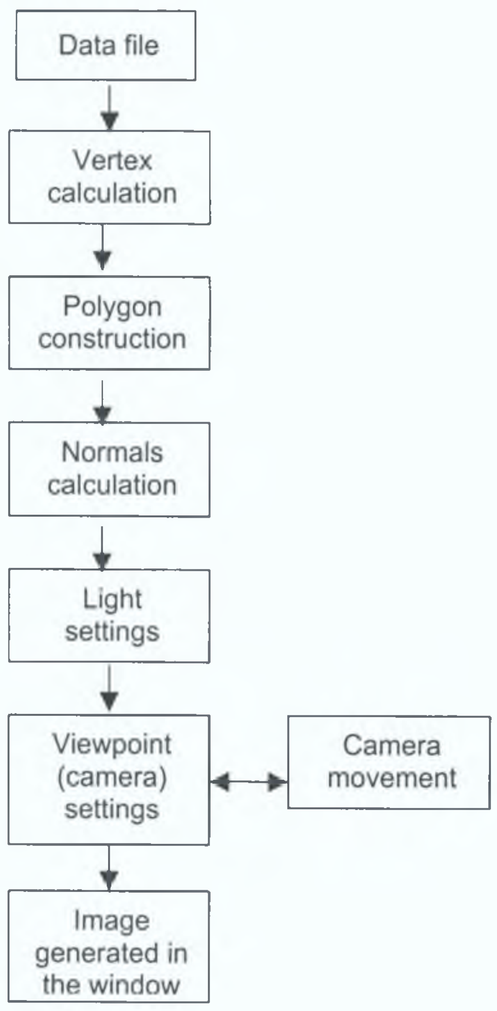


Figure 4.3 Overview of execution process

4.3 Window creation and management

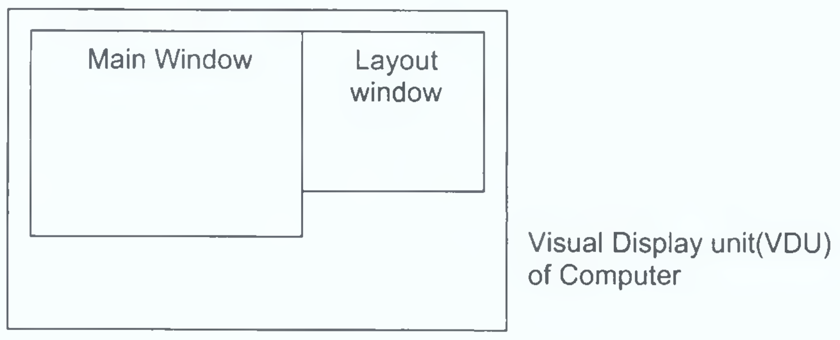


Figure 4.4 Main and Layout windows in VDU

3Dgen software creates two windows namely Main window and Layout window at the top left corner of the visual display unit of the computer. The window display mode was initialised to RGBA mode and the window size and position were specified. The main window is the active window which displays the model and also the menu options. The layout window is inactive and it shows the walkthrough path inside the model. The layout window gives a better understanding for the user to visualise the current position of the camera inside the model which was indicated by an yellow block.

4.4 Menu

A hanging menu was created and it was linked to the operation of the mouse of the PC. Clicking the right button of the mouse when the mouse pointer was in main window can activate the menu. The various menu options were shown in Figure 4.5.

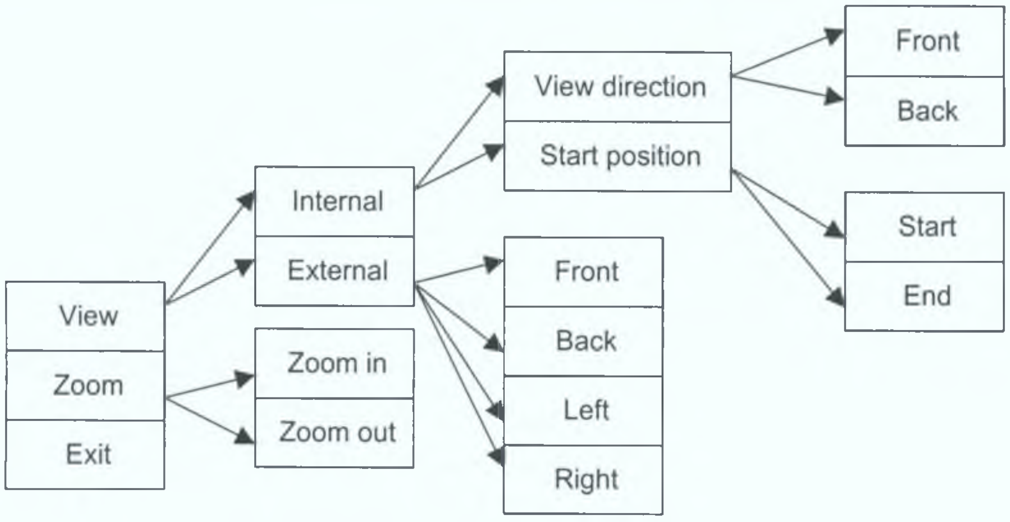


Figure 4.5 Menu options

The menu can be invoked only when the mouse pointer was in main window since it remains as the active window.

4.4.1 View Menu

Various viewing operations can be done using the view menu option. It has two options namely *internal* and *external* for viewing the internal and external surfaces of the model. The internal and external options can be subdivided into various options. The algorithm for the view menu option is as shown below:

```
View menu(option number)
{
        Switch (option number)
                11 : Internal()
                12 : External()
}
```

If *internal* option was selected then internal() function was invoked and if *external* option was selected then external() function was invoked.

4.4.2 View - Internal

The view - internal option is used for viewing related purposes with respect to the internal surfaces of the model. It has two options namely *view direction* and *start position*. If *view direction* is selected view direction() function is invoked and if *start position* is selected start position() function is invoked. The algorithm for the view-internal menu option is as shown below:

```
Internal(option number)
{
        Switch(option number)
                111 : View direction();
                112 : Start position();
}
```

4.4.3 View – External

The view – external option is used for viewing related purposes with respect to the external surfaces of the model. There are four options in external view namely *front view*, *back view*, *left hand side view* and the *right hand side view*. The camera will be positioned at the starting point or one end of the model facing the model and that view is considered as front view and other view were shown keeping the front view as a reference. If the *front* option was selected then with reference to the first set of points in the data file the boundary values of the front face is calculated based on the size of the model and using that the camera and aim coordinates were calculated. The front face acts as a reference for the other three options (back, left and right). If the *back* option is selected the boundary values of the back face is calculated with reference to front face and using that camera and aim coordinates were determined. Similarly for *left* and *right* options the corresponding boundary values were calculated for left and right faces and using that aim and camera coordinates were determined.

The algorithm for view – external option is as shown below:

```
External(option number)
{
      Switch(option number)
            121 : if (view external variable = front)
                              Message ("Already the window shows front view");
                  else
                              Calculate midpoint of the front face;
                              Calculate camera coordinates(based on midpoint of the
                                                                  face);
                              Calculate aim coordinates(based on midpoint of the
                                                                  face);
                              Window redisplay with new camera and aim coordinates;

            122 : if (view external variable = back)
                              Message ("Already the window shows back  view");
                  else
                              Calculate midpoint of the back face;
                              Calculate camera coordinates(based on midpoint of the
                                                                  face);
                              Calculate aim coordinates(based on midpoint of the
                                                                  face);
                              Window redisplay with new camera and aim coordinates;

            123 : if (view external variable = left)
                              Message ("Already the window shows left view");
                  else
                              Calculate midpoint of the left hand side face;
                              Calculate camera coordinates(based on midpoint of the
                                                                  face);
                              Calculate aim coordinates(based on midpoint of the
                                                                  face);
                              Window redisplay with new camera and aim coordinates;

            124 : if (view external variable = right)
                              Message ("Already the window shows right  view");
```

```
                        else
                                Calculate midpoint of the right hand side face;
                                Calculate camera coordinates(based on midpoint of the
                                                                face);
                                Calculate aim coordinates(based on midpoint of the
                                                                face);
                                Window redisplay with new camera and aim coordinates;
        }
```

4.4.4 Internal - View direction

The viewing direction of the camera can be changed using this menu option. The camera can look both in front or in exactly opposite direction as well. There are two options in this menu namely *front* and *back*. If the *front* option was selected the camera coordinates and the aim coordinates were interchanged. If the *back* option was selected then the camera coordinates and the aim coordinates were interchanged.

The algorithm for the internal – view direction option is as shown below:

```
        View direction(option number)
        {
            Switch(option number)
                1111 : if (view direction = front)
                                Message("view direction is front");
                        else
                                Temp variable = camera coordinates;
                                Camera coordinates = aim coordinates;
                                Aim coordinates = temp variable;
                                View direction = front;
                                Window redisplay based on new camera and aim
                                coordinates;

                1112 : if (view direction = back)
                                Message("view direction is back");
                        else
                                Temp variable = camera coordinates;
                                Camera coordinates = aim coordinates;
                                Aim coordinates = temp variable;
                        View direction = back;
                        Window redisplay based on new camera and aim
                        coordinates;
    }
```

4.4.5 Internal – Start position

This option allows the user to decide the starting position of the virtual walkthrough inside the model. It has two options namely *start position* and *end position*. The start position and the end position were determined based on the data from the input file. If *start* position was selected the menu function browses the data file and takes the first set of points for calculating view position. If *end* position was selected the menu function takes the last set of points from data file for calculating view position.

The algorithm for the Start position option is as shown below:

```
    Startposition(option number)
    {
        Switch(option number)
            1121 : if (position = start)
                        Message ("camera in start position");
                   Else
                        Browse datafile;
                        Extract first set of points;
                        Calculate aim and camera coordinates;
                        Window redisplay based on new coordinates;
            1122 : if (position = end)
                        Message ("camera in end position");
                   Else
                        Browse datafile;
                        Extract last set of points;
                        Calculate aim and camera coordinates;
                        Window redisplay based on new coordinates;
    }
```

4.4.6 Zoom menu

The model can be zoomed with this menu option. The scaling process of the model can be termed as zooming. Closer view of the model or a section of the model can be termed as *zoom in* and distant view of the model or a section of the model can be termed as *zoom out*. With respect to current view the model is scaled and scaling factor was determined by the options *zoom in* or *zoom out*. The current scaling factor for *zoom in* was fixed at 2 and for *zoom out* was fixed at 0.5.

The algorithm for zoom option is as shown below:

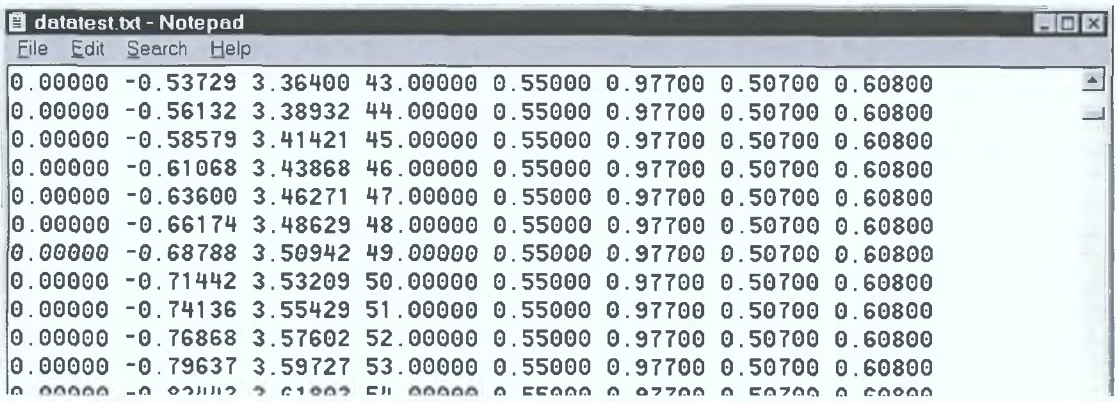
```
Zoom(option number)
{
      Switch(option number)
              21 :    store view coordinates;
                      scale the model(scale factor 2);
                      recalculate coordinates;
                      Window redisplay with new coordinates;

              22 :    Store view coordinates;
                      scale the model(scale factor 0.5);
                      recalculate coordinates;
                      Window redisplay with new coordinates;
}
```

4.4.7 Exit

This option allows the user to successfully close the 3D*gen* software. The input data file will be closed and memory variables and the buffer will be flushed so that the memory space can be used by other applications.

4.5 Data file

Hollow three-dimensional cylindrical models were created from data sets fetched by the data file. The datasets can be obtained either from some other software (eg. CAD output file) or entered by the user in a simple format in an ordinary text file. The model was constructed using the data available in the text file with filename as “datatest.txt”. In the file the data should be arranged in a specific format as shown in Figure 4.6. The format is driven by the idea to reduce the levels of detail [22] for the model as well as the surfaces [26]. The description of the model was kept at a minimal level so as to reduce the processing time in generating the model with compromising much in the quality of the model. The data format is structured in a simple way so that the user can easily understand the data. There are 8 columns of data and each column holds a basic attribute of a slice of the model. The table 4.1 specifies the format and the order in which they were arranged. Each value should be separated with a space. The advantages of the format are that the data are structured which are easily accessible and can be manipulated.



```
0.00000 -0.53729 3.36400 43.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.56132 3.38932 44.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.58579 3.41421 45.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.61068 3.43868 46.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.63600 3.46271 47.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.66174 3.48629 48.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.68788 3.50942 49.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.71442 3.53209 50.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.74136 3.55429 51.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.76868 3.57602 52.00000 0.55000 0.97700 0.50700 0.60800
0.00000 -0.79637 3.59727 53.00000 0.55000 0.97700 0.50700 0.60800
```

Figure 4.6 Data file

X coordinate	Y coordinate	Z coordinate	Angle of the face with respect to world coordinate system	Radius of the current face	RED value of colour coordinate	GREEN value of colour coordinate	BLUE value of colour coordinate

Table 4.1 Data file format

4.6 Data browse

The data in the data file was accessed all the time during when 3D*gen* software was running. The amount of data in the file was calculated and stored in variable for efficient navigation within the file to retrieve the required data. The model was bounded by an imaginary bounding box so that the boundary values were calculated which will be used while generating different views.

4.7 Model generation

The model was generated based on the data from the input data file. Using the data, vertices are calculated and the polygon is constructed using the vertices of the adjacent faces. A display list was created with all the polygons so that it will be easy for generating different views by calling the display list instead of recalculating the vertices and normal vectors. Normal vectors for each vertex are calculated and assigned to generate better light shading.

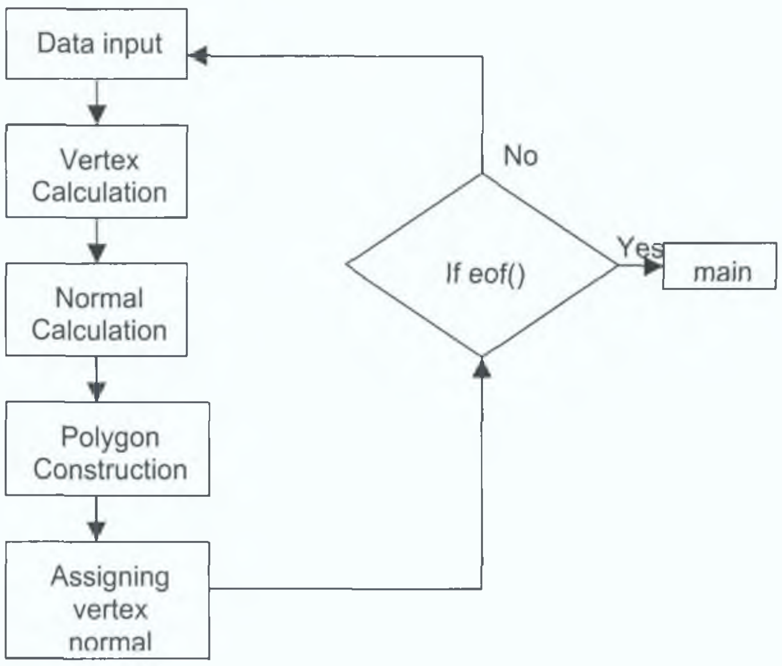


Figure 4.7 Model generation process

4.7.1 Vertex calculation

The model in Figure 4.1 was constructed by building blocks of the model (one shown in Figure 4.8). The size of each block depends on the data from the input file. Figure 4.8 shows two adjacent slices or faces and the model in-between these slices were constructed by connecting the vertices on the circumference of face F_n and F_{n+1} (A, B, C, D, E, F, G, H) to form polygons such as ABEF, BCGF etc., The faces F_n and F_{n+1} could be of different radii and at different orientation with respect to world coordinate system. The angular displacement between A and B with respect to centre P_n is kept at minimum value so that arc AB equals AB.

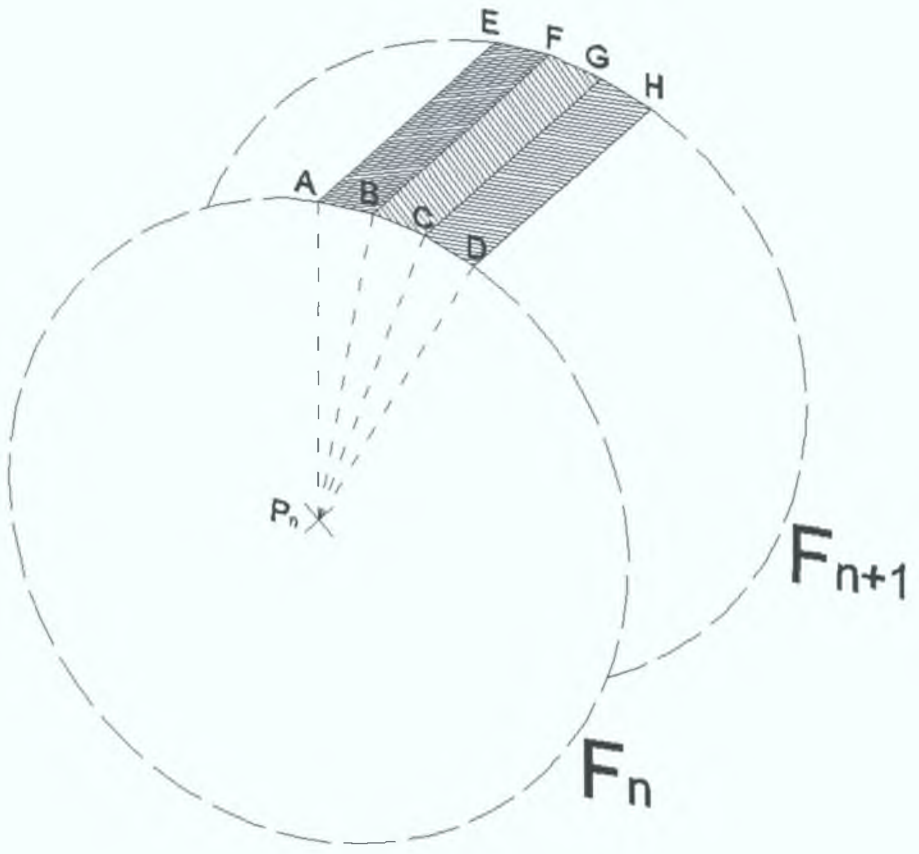


Figure 4.8 Two adjacent slices or faces of the model with polygon

The algorithm was designed in such a way that the modelling function calculated the vertices of the two adjacent slices (circles) and draw the polygons along the circumference. Figure 4.9 shows two adjacent slices F_n and F_{n+1}. Let $P_n(X_n,Y_n,Z_n)$ and $P_{n+1}(X_{n+1},Y_{n+1},Z_{n+1})$ be the centre points supplied by the data file. The data file also supplied corresponding radii r_n and r_{n+1}. Let θ_n and θ_{n+1} are the angles of faces F_n and F_{n+1} with respect to World Coordinate System. The model between face F_n and F_{n+1} was constructed by drawing polygons along the circumference of Fn and Fn+1. Let θ_n be the angular displacement of face F_n and θ_{n+1} be the angular displacement of face F_{n+1} with respect to world coordinate system. Vertices A, B, C, D are calculated to complete a polygon.

A (X_n,Y_n,Z_n)

$X_n = P_n(X_n) + r_n \cos \varnothing;$

$Y_n = P_n(Y_n) + (r_n \sin \varnothing * \cos \theta_n);$

$Z_n = P_n(Z_n) + (r_n \sin \varnothing * \sin \theta_n);$

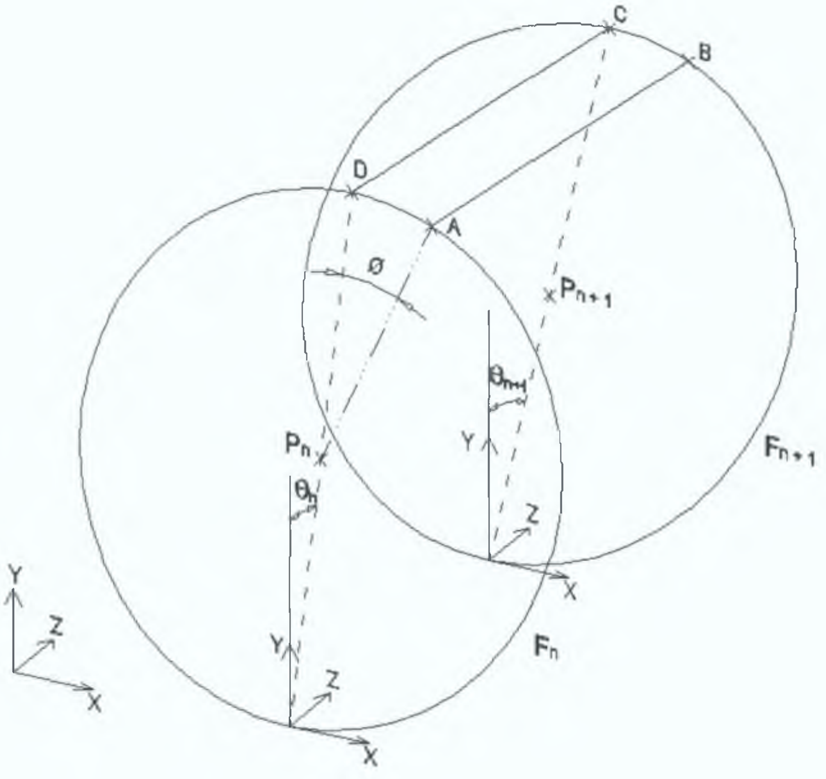


Figure 4.9 Two adjacent slices or faces of the model

B $(X_{n+1},Y_{n+1},Z_{n+1})$

$X_{n+1}= P_{n+1}(X_{n+1}) + r_{n+1} \cos \varnothing;$

$Y_{n+1}= P_{n+1}(Y_{n+1}) + (r_{n+1} \sin \varnothing * \cos \theta_{n+1});$

$Z_{n+1}= P_{n+1}(Z_{n+1}) + (r_{n+1} \sin \varnothing * \sin \theta_{n+1});$

C $(X_{n+1},Y_{n+1},Z_{n+1})$

$X_{n+1}= P_{n+1}(X_{n+1}) + r_{n+1} \cos (\varnothing+ \delta);$

$Y_{n+1}= P_{n+1}(Y_{n+1}) + (r_{n+1} \sin (\varnothing+\delta) * \cos \theta_{n+1});$

$Z_{n+1}=P_{n+1}(Z_{n+1}) + r_{n+1} \sin (\varnothing+\delta) * \sin \theta_{n+1});$

D (X_n,Y_n,Z_n)

$X_n= P_n(X_n) + r_n \cos (\varnothing+\delta);$

$Y_n= P_n(Y_n) + (r_n \sin (\varnothing+\delta) * \cos \theta_n);$

$Z_n= P_n(Z_n) + (r_n \sin (\varnothing+\delta) * \sin \theta_n);$

Where $0<=\varnothing<=360$

Using the above equations, vertices on the circumference of face F_n namely A, B, C and D were calculated and they were connected to for polygon ABCD. Also using the above equations, coordinates of other polygons were also calculated with different values of $\emptyset$ so that the entire circumference was covered.

4.7.2 Polygon construction

Polygons were constructed between the adjacent slices or faces by connecting the vertices on the circumference of the slices. By this method surfaces are generated along the circumference of the slices to form a cylindrical hollow model as shown in Figure 4.10. The gap between the faces F_n and F_{n+1} was covered by polygons (ex. BCFG, CDHG etc.,) along the circumference there by completing the construction of the surface between F_n and F_{n+1}. It has to be noted that only the surface was constructed not the solid model. Hence a virtual walkthrough inside the model was possible. The width of the polygon varies with respect to the angular displacement Ø and or incremental value d. If the value of d increases the cylindrical model will look like multi face solid. Hence the value of d was kept at a critical value with out compromising the shape and also the processing time in calculating the number of vertices. The polygons were constructed in a similar fashion on all the segments.

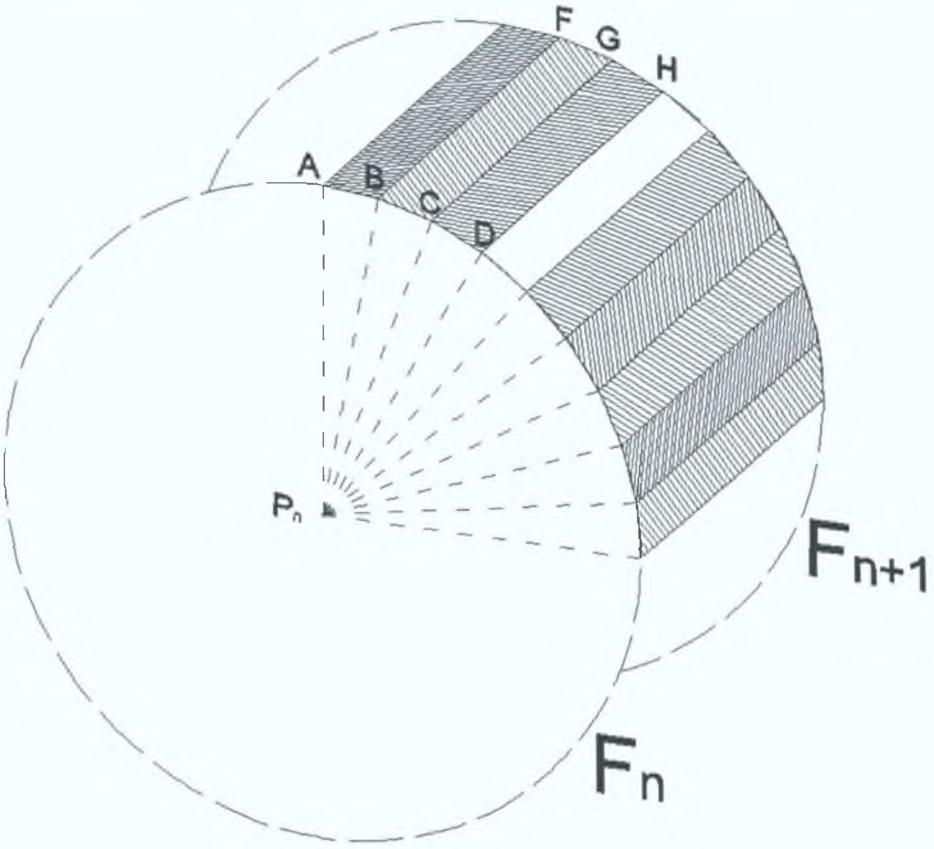


Figure 4.10 Polygon construction

The radii of faces F_n and F_{n+1} may not be the same there by creating model with variable radius for a required length and orientation.

4.7.3 Light settings

When light rays strike a surface, some of the rays are reflected enabling us to see the surface. The appearance of a surface depends on the light that strikes it combined with the properties of the surface material such as colour, smoothness, opacity etc., The intensity of light at its point of origin affects how brightly the light illuminates an object. A dim light cast on a brightly coloured object shows dim colours. The more a surface inclines away from a light source, the less light it receives and the darker it appears. The relative to the light source is known as angle of incidence. When the angle of incidence is 90° the surface is illuminated with full intensity of the light source. As the angle of incidence diverges from 90°, the intensity of illumination decreases.

In real world light diminishes over distance. Objects far from the light source appear darker; objects near the source appear brighter. This effect is known as attenuation. Natural light attenuates at an inverse square rate – that is, its intensity diminishes in proportion to the square of the distance from the light source. It is common for attenuation to be even greater when the atmosphere disperses the light, especially when there are dust particles in the atmosphere, or fog or clouds.

The light, an object reflects can illuminate other objects. This effect is known as radiosity. The more light a surface reflects, the more light it contributes to illuminating other objects. Radiosity creates ambient light. Ambient light has a uniform intensity and is uniformly diffuse. It has no discernible source and no discernible direction. The colour of light depends partly on the process that generates light. Light colour also depends on the medium the light passes through. For example clouds in the atmosphere tint daylight blue and stained glass tints light a highly saturated colour. Light colours are additive colours; the primary light colours are red, green and blue(RGB). As multiple coloured lights mix together, the total light in the scene gets lighter and eventually turns white.

Naturally lit scenes such as daylight or moonlight get their most important illumination from a single light source. Artificially lit scenes often have multiple light sources of similar intensity. Both kinds of scenes require multiple secondary lights for effective illumination. Sunlight and moonlight have parallel rays coming from a single direction. The direction and angle vary depending on the time of day, the latitude and the season. For example in a clear weather the colour of the sunlight is pale yellow, blue in a cloudy weather and dark grey in stormy weather.

Particles in the air can give sunlight an orange or brown tint. Sunlight and moonlight can be defined as directional light. A spotlight is defined as a beam of light focused towards a specified target. Ambient light simulates the background radiosity. The colour of ambient light tints the scene and usually should be the complement of the colour of the principal light source of the scene. The intensity of ambient light affects contrast as well as overall illumination – the higher the intensity of ambient light, the lower the contrast. This is because ambient light is completely diffuse, so the angle of incidence is equal for all faces. Specular lighting is what produces the shiney highlights and helps us to distinguish between flat, dull surfaces such as plaster and shiney surfaces like polished plastics and metals.

The OpenGL light model presumes that the light that reaches the eye from the polygon surface arrives by four different mechanisms:

AMBIENT	- light that comes from all directions equally and is scattered in all directions equally by the polygons in the scene. This isn't quite true of the real world - but it's a good first approximation for light that comes pretty much uniformly from the sky and arrives onto a surface by bouncing off so many other surfaces that it might as well be uniform.
DIFFUSE	- light that comes from a particular point source (like the Sun) and hits surfaces with an intensity that depends on whether they face towards the light or away from it. However, once the light radiates from the surface, it does so equally in all directions. It is diffuse lighting that best defines the shape of 3D objects.
SPECULAR	- as with diffuse lighting, the light comes from a point source, but with specular lighting, it is reflected more in the manner of a mirror where most of the light bounces off in a particular direction defined by the surface shape.
EMISSION	- in this case, the light is actually emitted by the polygon - equally in all directions.

So, there are three light colours for each light - Ambient, Diffuse and Specular (set with glLight). OpenGL implementations support at least eight light sources and it computes the colour of each pixel in a final, displayed scene that was held in the framebuffer. Part of this computation depended on what lighting was used in the scene and on how objects in the scene reflect or absorb that light.

In RGBA mode, the illuminated colour of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source [41]. Each light source contributes the sum of three terms: ambient, diffuse, and specular. The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity. The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalised vector from the vertex to the light source. The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalised vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material.

$$\text{Specular lighting} = \text{Material specular reflectance} * \text{Specular light intensity} \\ * (\text{Vertex to eye vector} \times \text{vertex to light vector})^{\text{Shininess}}$$

All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cut-off angle. All dot products were replaced with zero if they were evaluated to a negative value. The alpha component of the resulting lighted colour was set to the alpha value of the material diffuse reflectance

4.7.4 Normal calculation

To apply lighting, it is necessary to first compute the normals to the polygons. A normal is a vector that defines how a surface responds to lighting, i.e. how it is lit. The amount of light reflected by a surface is proportional to the angle between the lights direction and the normal. The smaller the angle the brighter the surface will look. Normals in OpenGL can be defined per face or per vertex. If defining a normal per face then the normal is commonly defined as a vector that is perpendicular to the surface. In order to find a perpendicular vector to a face, two vectors coplanar with the face are needed. Afterwards the cross product will provide the normal vector, i.e. a perpendicular vector to the face. A face normal is a unit vector that defines which way a face is pointing. The direction that the normal points represents the front or outer surface of the face.

So the first step was to compute two vectors coplanar to a face. In Figure 4.11 let P(px,py,pz) be the centre of the face(circle), A(ax,ay,az) and B(bx,by,bz) be the points of the polygon then BP and BA are coplanar vectors and BN is the normal vector.

Vector BP = px-bx,py-by,pz-bz;
Vector BA = ax-bx,ay-by,az-bz;

The following equations give the cross product of the two vectors BP and BA.

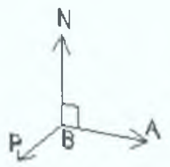


Figure 4.11 Coplanar and normal vectors

BN = BP x BA; (x – cross product)
BN = (px-bx,py-by,pz-bz) x (ax-bx,ay-by,az-bz);

BNx = [(py-by)*(az-bz)] – [(ay-by)*(pz-bz)];
Bny = [(ax-bx)*(pz-bz)] – [(px-bx)*(az-bz)];
BNz = [(px-bx)*(ay-by)] – [(ax-bx)*(py-by)];

To obtain proper lighting vector BN has to be normalised i.e. making it to unit length. OpenGL takes into consideration the length of normal vector when computing lighting. Normalisation is done by first computing the length of the vector and then dividing each component by the vectors length. The length of the vector was calculated as follows

$$VL = \sqrt{BNx * BNx + BNy * BNy + BNz * BNz}$$

Hence the normalised vector (NvBN) will be

$$NvBN = ({}^{BNx}/_{VL}, {}^{BNy}/_{VL}, {}^{BNz}/_{VL})$$

The problem with specifying normal per face was that the brightness of the each face was constant. But there will be a clear difference between faces with different orientation. So for smoother lighting normals should be calculated per vertex. While computing normals per vertex it will be necessary to take into account the faces that share the vertex. Except for the start and end slice the vertices in rest of the slices were shared by four polygons. Hence the normal at a vertex should be computed as the normalised sum of unit normal vectors for each face the vertex shares.

At the start of the execution of the software the face normals were calculated and stored in a text file called "nrd.txt". While generating the model these values were used to calculate the normal for each vertex. Figure 4.12 shows the face normals and the vertex normal for vertex A.

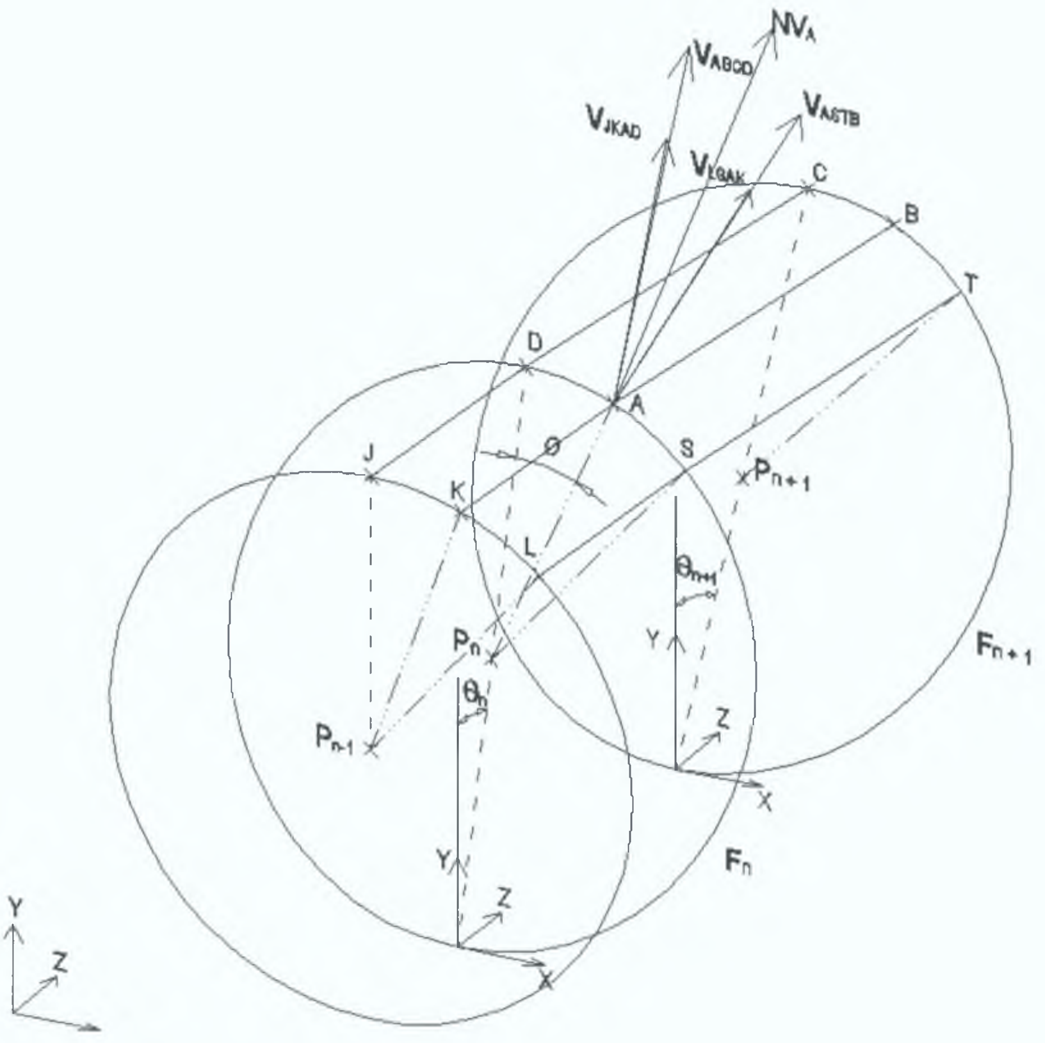


Figure 4.12 Vertex normal

The vertex normal NV_A for A can be calculated by the following equations

$$\sum V_x = V_{x(ABCD)} + V_{x(JKAD)} + V_{x(LSAK)} + V_{X(ASTB)}$$
$$\sum V_Y = V_{Y(ABCD)} + V_{Y(JKAD)} + V_{Y(LSAK)} + V_{Y(ASTB)}$$

$$\sum V_Z = V_{Z(ABCD)} + V_{Z(JKAD)} + V_{Z(LSAK)} + V_{Z(ASTB)}$$

$$NV_L = \sqrt{\sum V_X + \sum V_Y + \sum V_Z}$$

$$NV_A = \left(\frac{\sum V_X}{NV_L} + \frac{\sum V_Y}{NV_L} + \frac{\sum V_Z}{NV_L} \right)$$

4.8 Viewport

The viewport is the rectangular region of the window where the image is drawn. The viewport is measured in window coordinates, which reflect the position of pixels on the screen relative to the lower left corner of the window. The window manager is responsible for opening a window on the screen, not the OpenGL. However, by default the viewport is set to the entire pixel rectangle of the window that's opened. The *glViewport()* command was used to choose a smaller drawing region; for example, the window can be subdivided to create a split-screen effect for multiple views in the same window.

Void glViewport(Glint x, Glint y, Glsizei width, Glsizei height);

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

The aspect ratio of a viewport should generally equal the aspect ratio of the viewing volume. If the two ratios are different, the projected image will be distorted as it is mapped to the viewport. The subsequent changes to the size of the window do not explicitly affect the viewport. The application detects window resize events and modifies the viewport appropriately. The window was set to a size of 300x300 to optimise the graphics hardware usage and for better performance. Also the window maximise controls were enabled. The coordinate system in the OpenGL is shown in Figure 4.13.

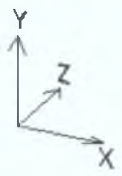


Figure 4.13. OpenGL coordinate system

4.9 Camera Settings

The *gluLookAt()* utility routine was used to create the camera view. Three sets of arguments were used to specify the location of the viewpoint or camera, a reference point toward which the camera is aimed, and the up direction.

```
gluLookAt(Gldouble eyex, Gldouble eyey, Gldouble eyez, Gldouble viewx, Gldouble viewy,
                Gldouble viewz, Gldouble upx, Gldouble upy, Gldouble upz);
```

The desired viewpoint was specified by eyex, eyey, and eyez. The viewx, viewy, and viewz arguments specify any point along the desired line of sight. The upx, upy, and upz arguments indicate the up direction.

The pseudo code for camera function

```
If (move variable = forward)
{
    Maintain view direction;
    Obtain next set of data from data file based on current position of camera;
    Assign the data to camera and aim coordinate variables;
}
else if (move variable = backward)
{
    Maintain view direction;
    Obtain previous set of data from data file based on current position of camera;
    Assign the data to camera and aim coordinate variables;
}
```

The viewpoint or camera location is at the centre of the cylindrical model. The reference point or the focus point is the centre of the next face. Here both Y-axis and Z-axis of world coordinate system acts as up vector or up direction depending on the position of the camera. Since the coordinates fetched from the data file belong to the centre of the face or slice, it was used as the viewpoint or camera coordinates. At any point of time, two sets of centre coordinates of face Fn and Fn+1 will be available in the program and hence the centre coordinates of face F_n becomes the camera coordinates and that of face F_{n+1} becomes the view coordinates or aim coordinates.

The camera movement was managed by the “up arrow” and “down arrow” keys of the keyboard for the forward and backward movement. When the up arrow key was pressed which means a forward movement for the camera, the view direction (front or back) was checked and the based on the current coordinates, data were obtained from data file and assigned to camera and aim coordinate variables. Similarly when down arrow key was pressed the view direction was checked and previous set of data were obtained from data file and assigned to camera and aim coordinate variables.

4.10 Window idle and Window reshape

When the software 3D*gen* is running the window idle function keep the main and layout window constantly refreshed with respect to changes in model and also with the view. When the model is rotated about the axes window function performs the rotation when the speed and direction are set. If the window size is changed by the user the reshape function calculates the viewport area and readjusts the scene with respect to the new size of the window.

CHAPTER 5

CHAPTER 5: RESULTS AND DISCUSSION

5.1 Introduction

This chapter reports the discussion and analysis of the output and results of the developed software '3D*gen*'. A sample model, human digestive system with approximately 2700 lines of data, was created to test the various parameters of the software. A number of tests were conducted to test the compatibility of the software with various operating systems.

Sample Model 1:

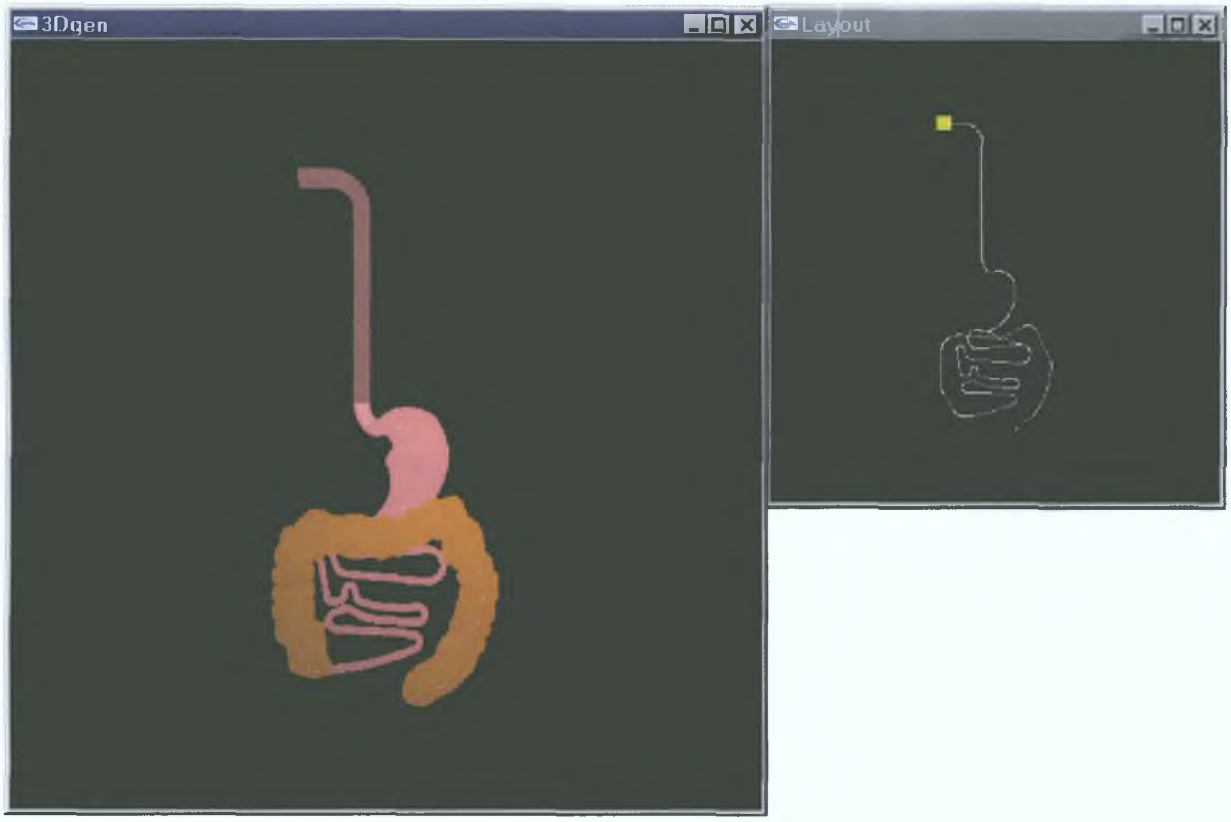


Figure 5.1 Sample model – human digestive system

Figure 5.1 shows the external view of the human digestive system generated by the software using the data supplied by the data file. The data was calculated by the author of the 3D*gen* software to test the performance of the software. The data for the human digestive system was calculated based on the information available from Oregon HSU and approximated to get the desired shapes. The model consisted of a series of complex cylindrical sections at various radius and at various angles of orientation with respect to world coordinate system. The eye coordinates for this view were calculated based on the size of the model.

When the source code was compiled the default position of the camera was at the beginning of the model focusing inside the model. Figure 5.2 shows the default camera position and view of the sample model. The software program was provided with hanging menu through mouse pointer. It can be invoked at any point inside the Main window.

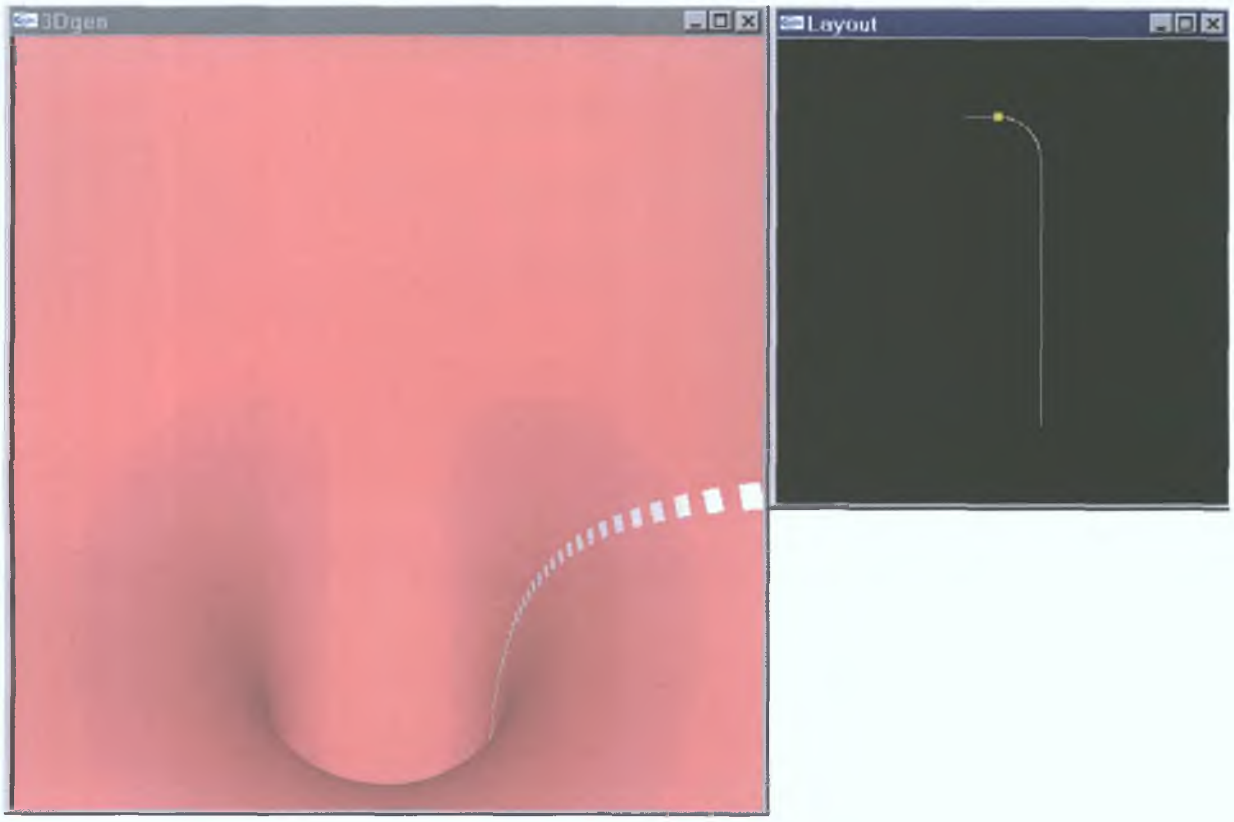


Figure 5.2 Sample Internal view

The movement of camera (change in camera coordinates) was controlled by the 'up arrow' and 'down arrow' keys. The direction of view or view coordinates can be changed by clicking the menu option "view direction" The white marks on the internal surface of the model acts as a reference for the movement of the camera inside the model.

5.2 Compatibility

The source code was developed using OpenGL with C functions which gives a very good flexibility in running the software program in various operating system environments such as Microsoft windows NT, Microsoft windows 2000, Linux etc., The author tested the source code in the above mentioned operating systems and results were similar in terms of display except few changes in the menu fonts.

5.2.1 Windows NT/2000

Figures 5.1 through 5.5 were generated in Windows NT 4.00.1381 and Windows 2000 operating systems where the source code was compiled using Microsoft Visual C++ 6.0. Once the code is compiled, an *.exe* file is generated and that can be used in other Microsoft Windows based operating systems because those operating systems uses the same window interface and hence the compatibility. Most of the operating systems [41] supports OpenGL and the latest versions of Windows NT, Windows 2000 and Windows XP have necessary files to execute the .exe file of the developed software.

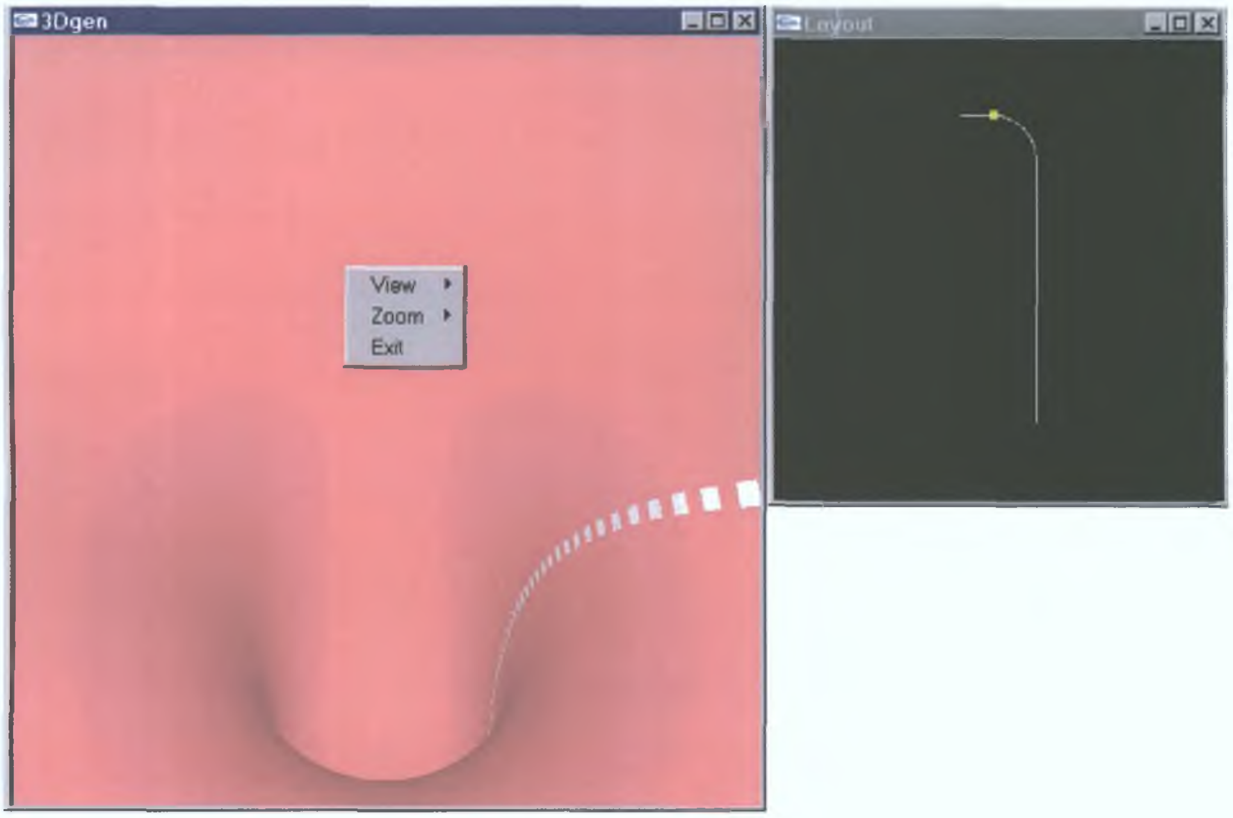


Figure 5.3 Sample internal view with main menu option

Even if the operating system is an older version without OpenGL support files, they can be downloaded from World Wide Web for free. Figure 5.3 shows the menu invoked by the right click of the mouse button.

5.2.2 Linux

Figure 5.7 shows the output generated by the same source code compiled in Linux operating system (Figure 5.6 generated in Microsoft Windows NT for the same data set). The output and the controls remain the same as that of windows operating system. The source code was tested in Redhat Linux 7.2 version and kernel version 2.4.2. Figures 5.7, 5.9, 5.11, 5.13 were generated in Redhat Linux.

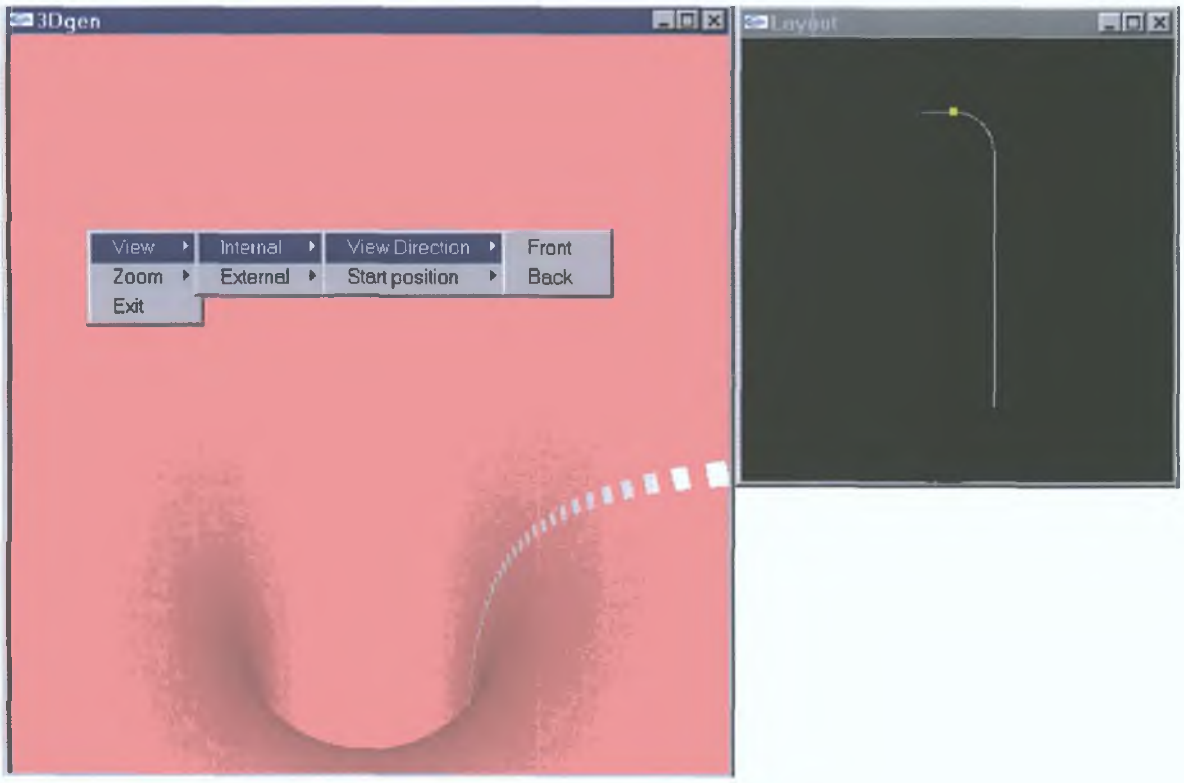


Figure 5.4 Sample view with view direction options

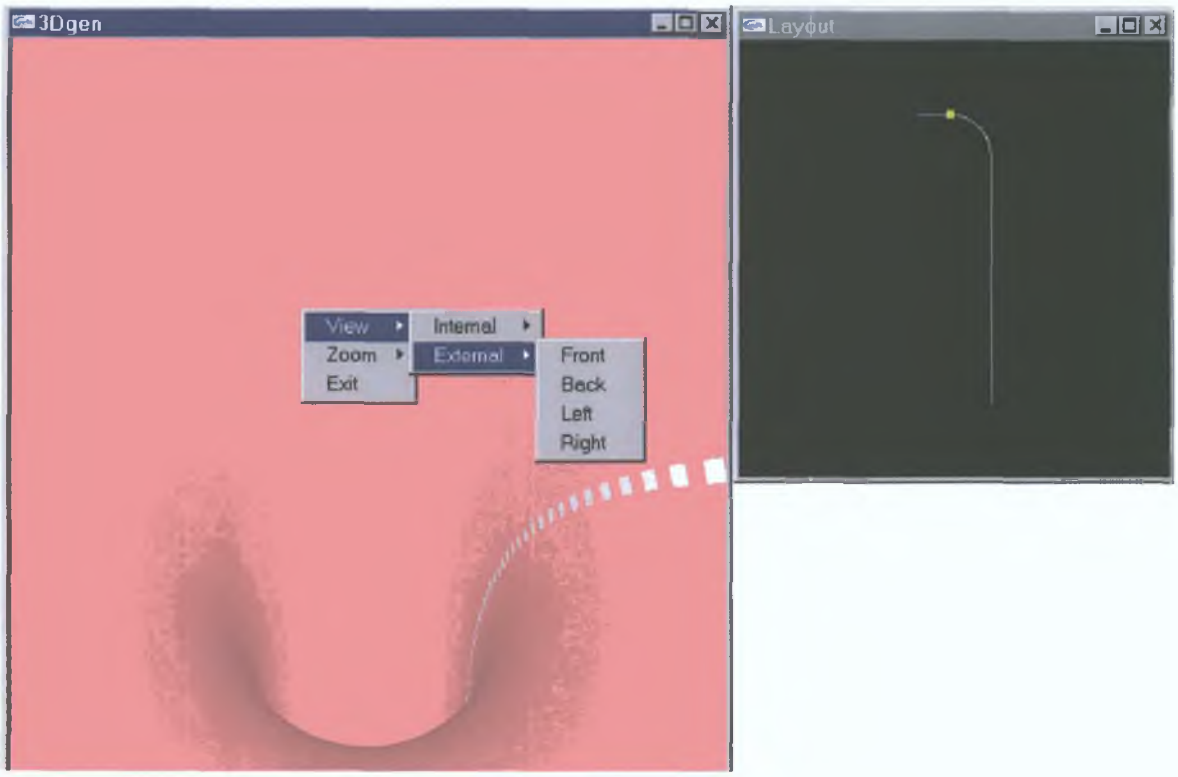


Figure 5.5 Sample view with external view options

Model 2:Pipe network I

Windows:

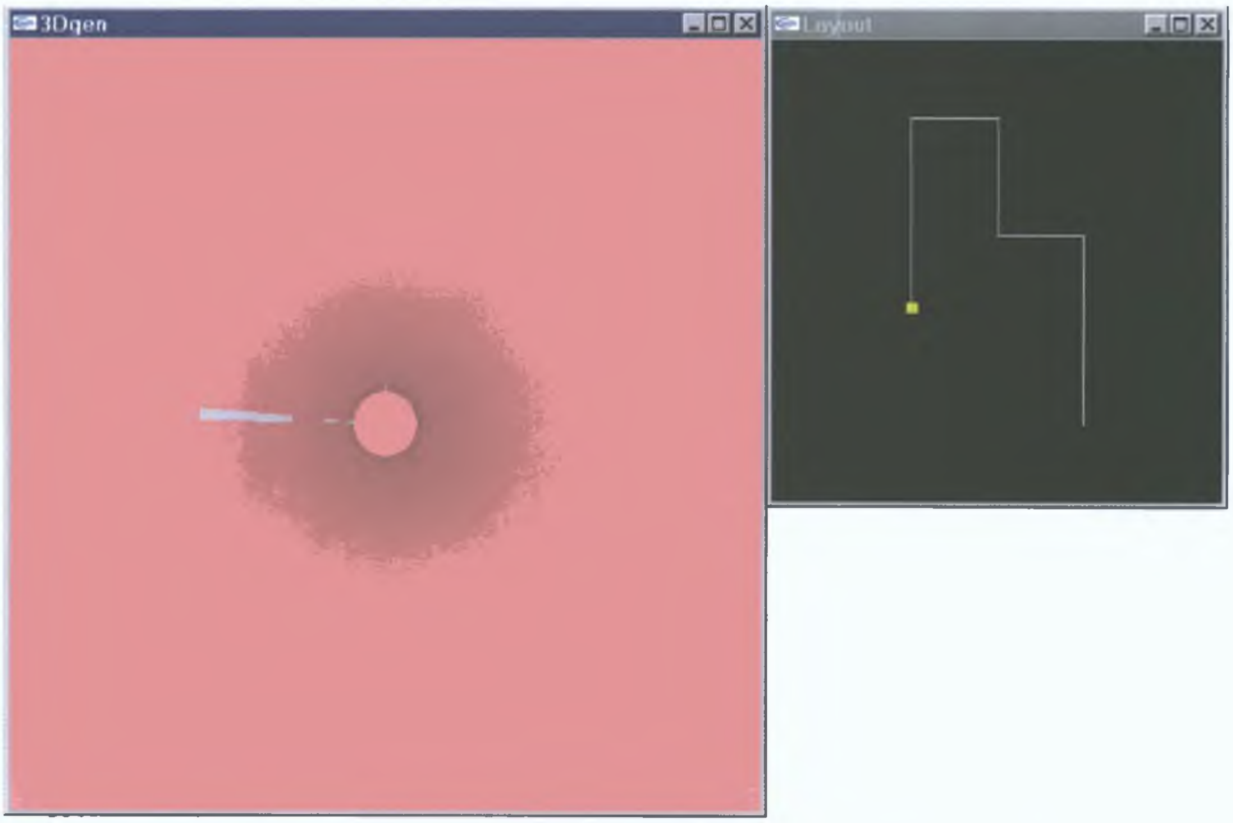


Figure 5.6 Model 2 output in Microsoft Windows NT (internal view)

Linux:

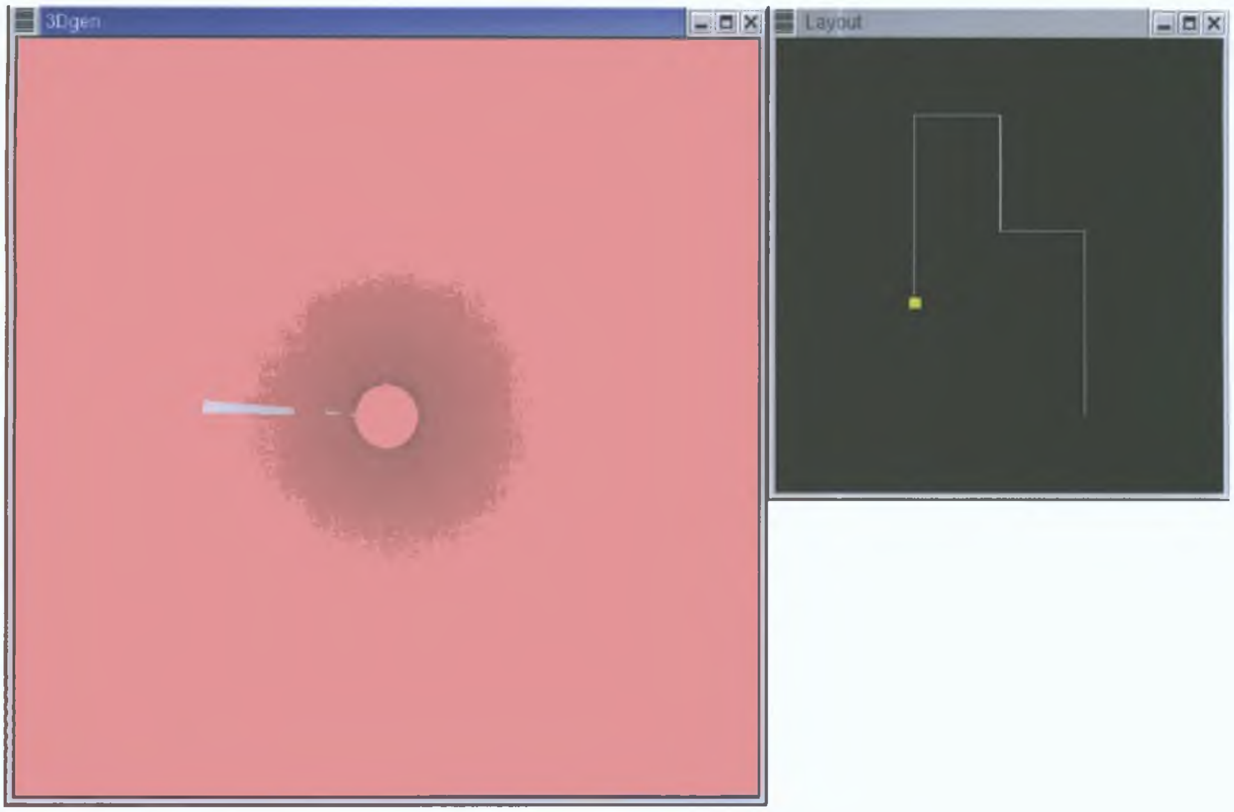


Figure 5.7 Model 2 output in Redhat Linux (internal view)

External view:

Windows:

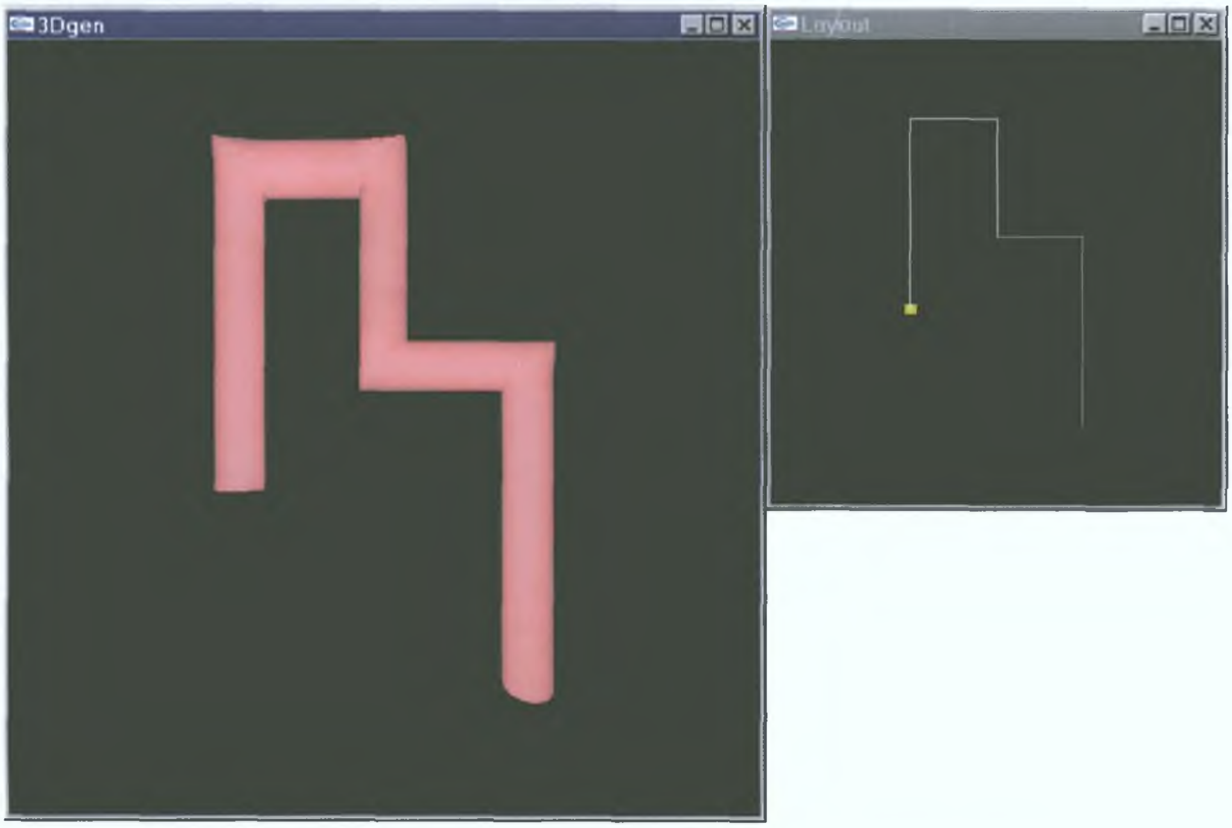


Figure 5.8 Model 2 output in Microsoft Windows NT (external view)

Linux

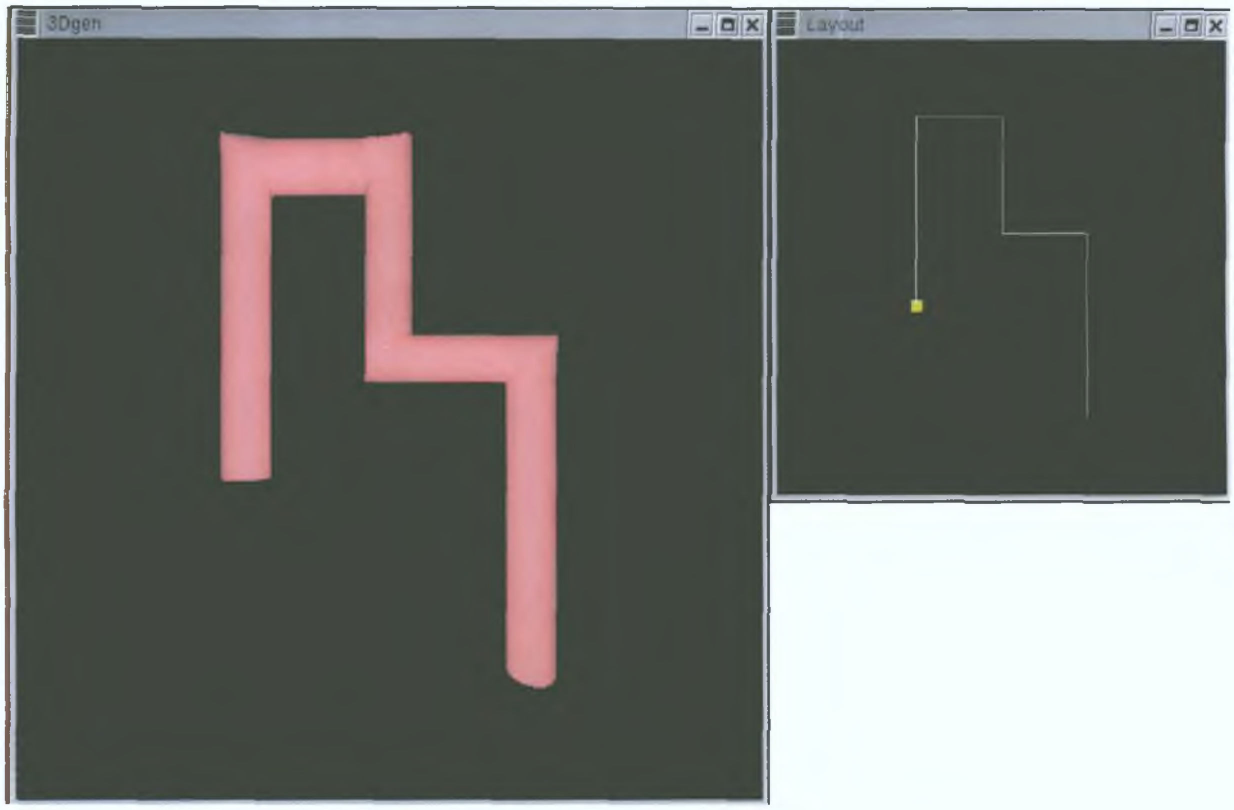


Figure 5.9 Model 2 output in Redhat Linux (external view)

Model 3: Pipe network II

Windows:

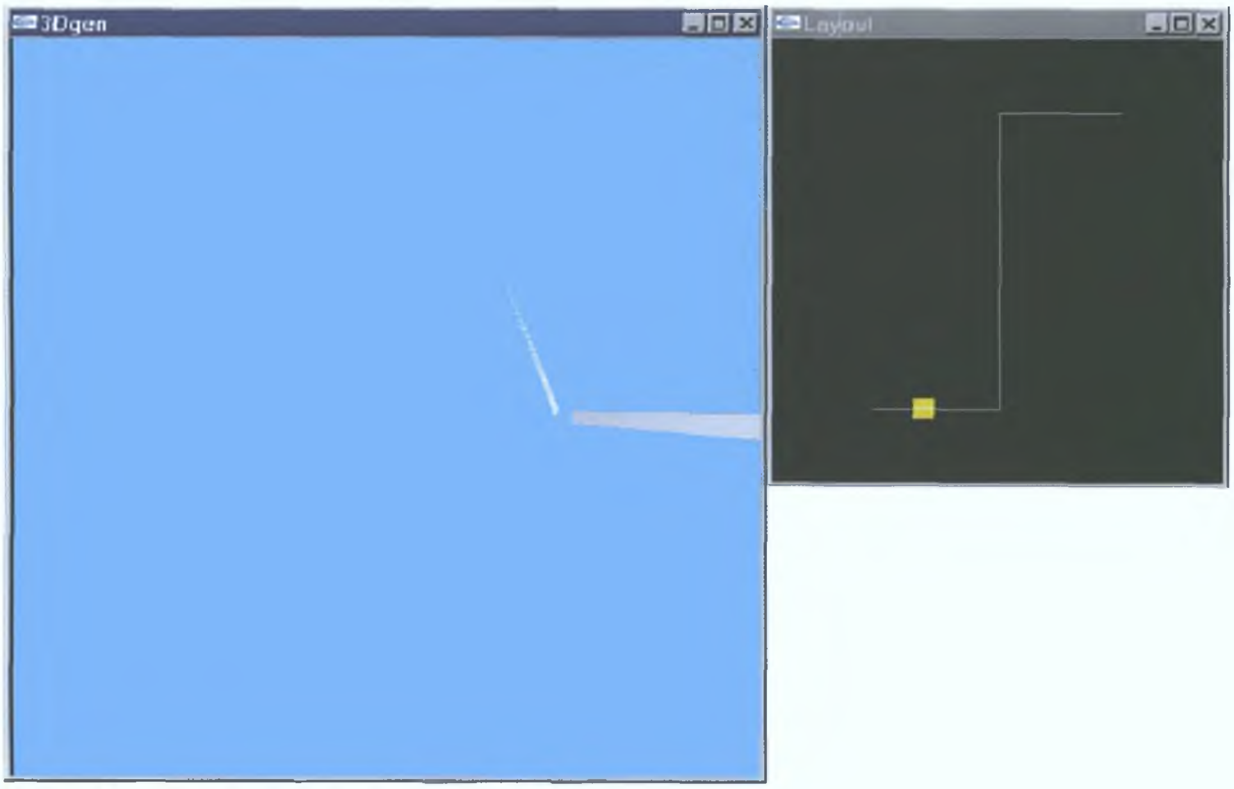


Figure 5.10 Model 3 output in Microsoft Windows NT (internal view)

Linux:

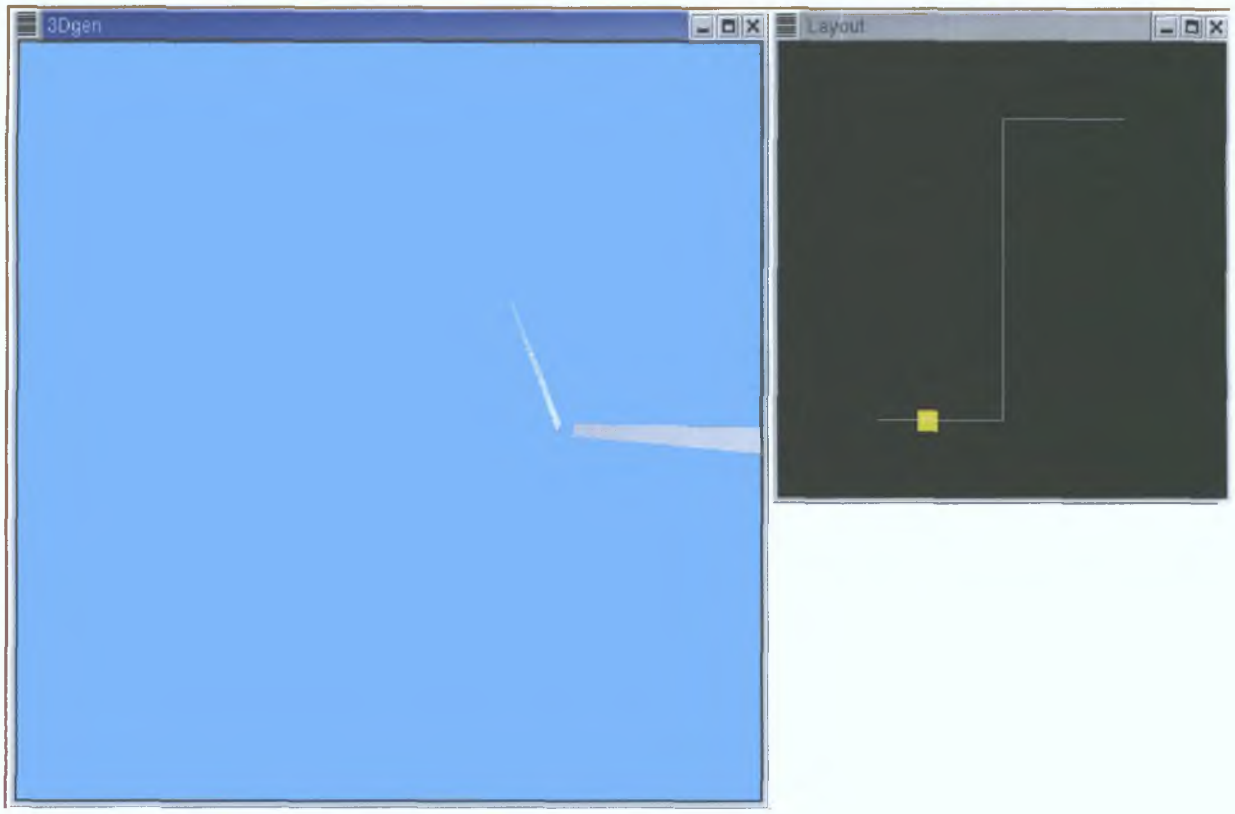


Figure 5.11 Model 3 output in Redhat Linux (internal view)

External view:

Windows:

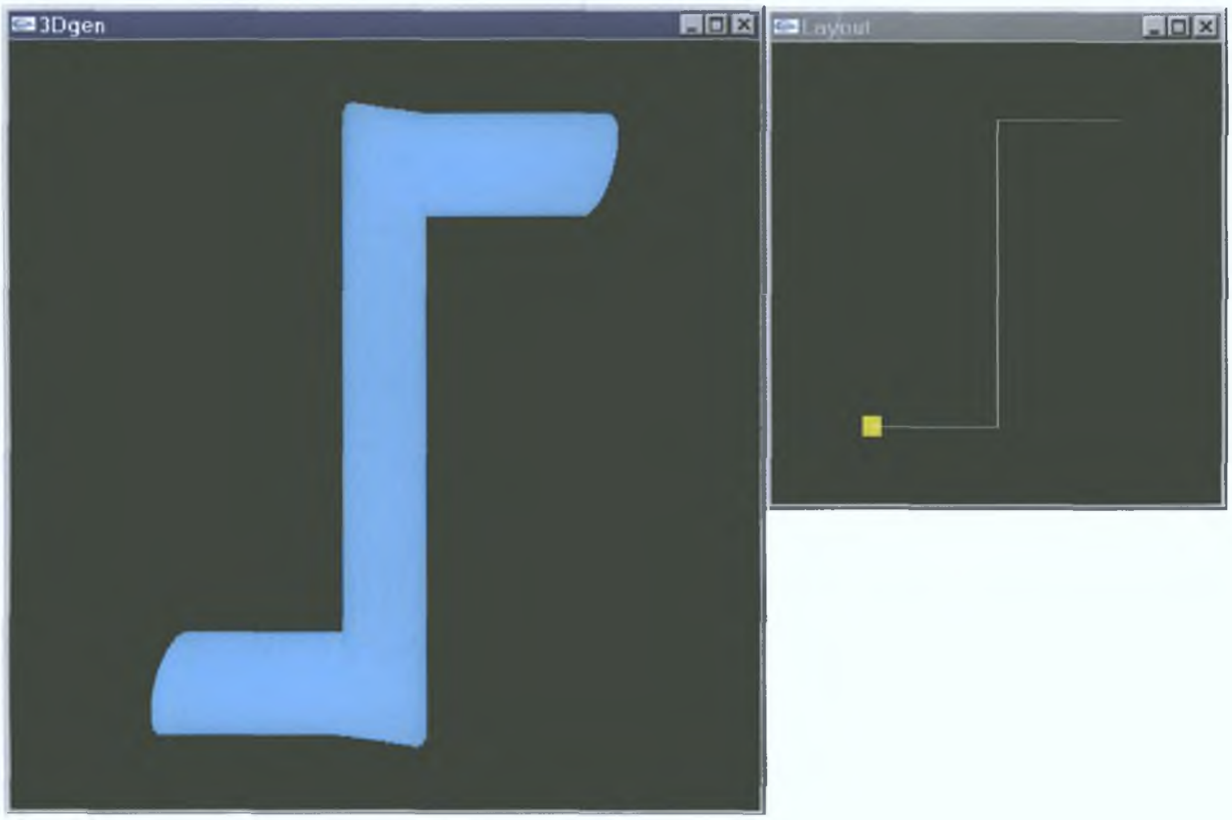


Figure 5.12 Model 3 output in Microsoft Windows NT (external view)

Linux:

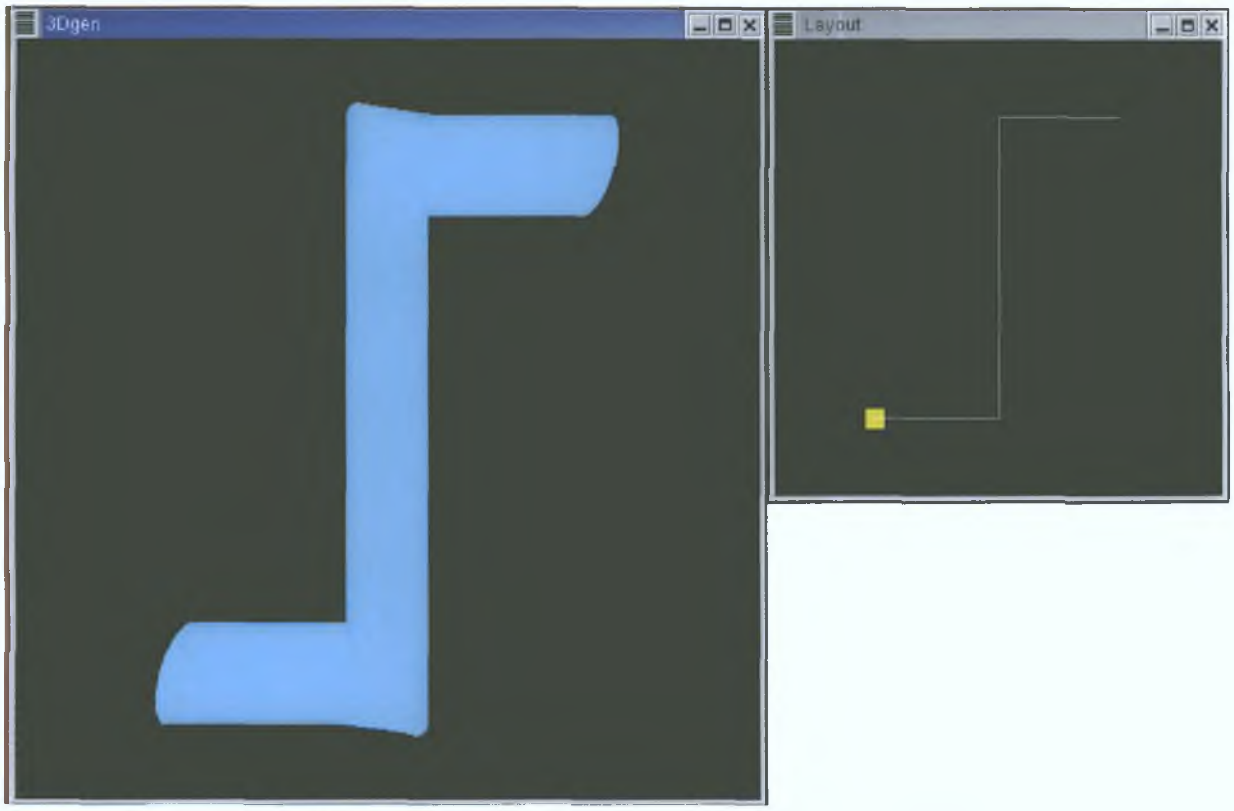


Figure 5.13 Model 3 output in Redhat Linux (external view)

5.3 Discussion

3D*gen* runs in both Linux and Windows based operating systems and Figures 5.1 through 5.13 images produced for different data sets. Figure 5.4 and 5.5 shows the different hanging menu options available while the software is running. The rectangular yellow patch in the layout window indicates the position of the camera when it is inside the model.

5.3.1 Human digestive system (Model 1)

As mentioned earlier the model (Figure 5.1) was generated using the data file created by the author of 3D*gen* software, which has approximately 2700 lines of data. The data were calculated based on the information available from Cliniweb of Oregon health sciences university and various approximations were applied to get the desired shape. The model was generated in desktop PC with Pentium III 550Hz processor with 8 MB display memory. It took 45minutes (approx) to generate the model.

The centre line data from the data file was used to construct the surfaces of the model and with gullet and oesophagus section had little change in the diameter. The data, which was used to define the gullet and oesophagus section, were calculated for variable angular displacements with respect to the world coordinate system.

World coordinate system was used while developing the software so that user can easily understand the input data or if the user decides to input the data by himself then he can type the data with respect to world coordinate for the entire model without localising the coordinates. The user has to be aware of only one coordinate system, which is world coordinate system.

The stomach was generated in various sections with different radii at different orientations to the world coordinate system. The small intestine section is a series complex bends that requires in depth data to generate that part of the model. But the level of details was maintained so that data format remains undisturbed. The number of faces defined in those sections was increased making it as the most defined part of the model with distance between the adjacent slices or faces being less.

The data for the large intestine was defined as a series of variable radii sections with the repetition of the construction but arranged in different orientation with respect to world coordinate system. The large intestine was given a different colour just to clearly differentiate from other sections of the system.

Figure 5.1 shows that the gullet and oesophagus was produced with dark shading when compared to rest of the sections in the digestive system. This was because of the reversal in order of the data input. If the data input order were reversed the polygon will be constructed with vertices taken in anti clockwise direction. This reverses the assignment of the vertex normals. This problem arises because of the wrong arrangement of the data.

Since OpenGL functions has to be specified in a specific manner to get the desired output, there should be a reversal in the order of specifying vertices, for example polygon will be constructed as DCBA instead of ABCD thereby negating the normal vector direction.

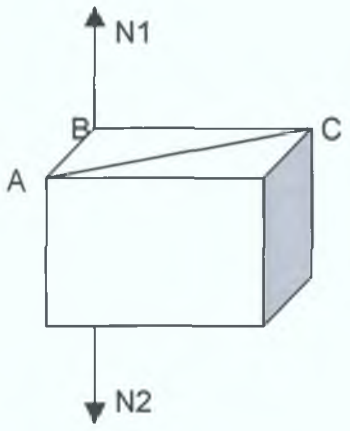


Figure 5.14 Order of vertices decides front face and back face

Figure 5.14 shows that N1 will be the vertex normal if the coplanar vectors are specified as vector BA and BC. If the vectors are specified as vector AB and CB then N2 will be the normal vector at vertex B. This proves the importance of vertex normal with respect to the generation of light effects and it also signifies the procedural nature of OpenGL.

5.3.2 Model 2

Figures 5.6 and 5.7 show the output of model 2 generated by Microsoft Windows and Redhat Linux. The internal view of model 2 shows the white coloured indicator line segments which acts as a reference and also to create the feeling of the camera movement.

Figures 5.8 and 5.9 shows the right hand side external view of model 2 generated by both operating systems. Both the outputs are identical with little or no difference. The layout window gives the virtual walkthrough path without a break in it at the sharp bends.

The model was generated with the data given in appendix B. The different sections of the model were at different angles with respect to the world coordinate system. Figure 5.6 gives the internal view of the model and the location of the camera is at the starting point of the model (i.e. one end of the model). The white marks along the surface of the model act as a reference and gives a feel of the movement when the camera moves inside the model.

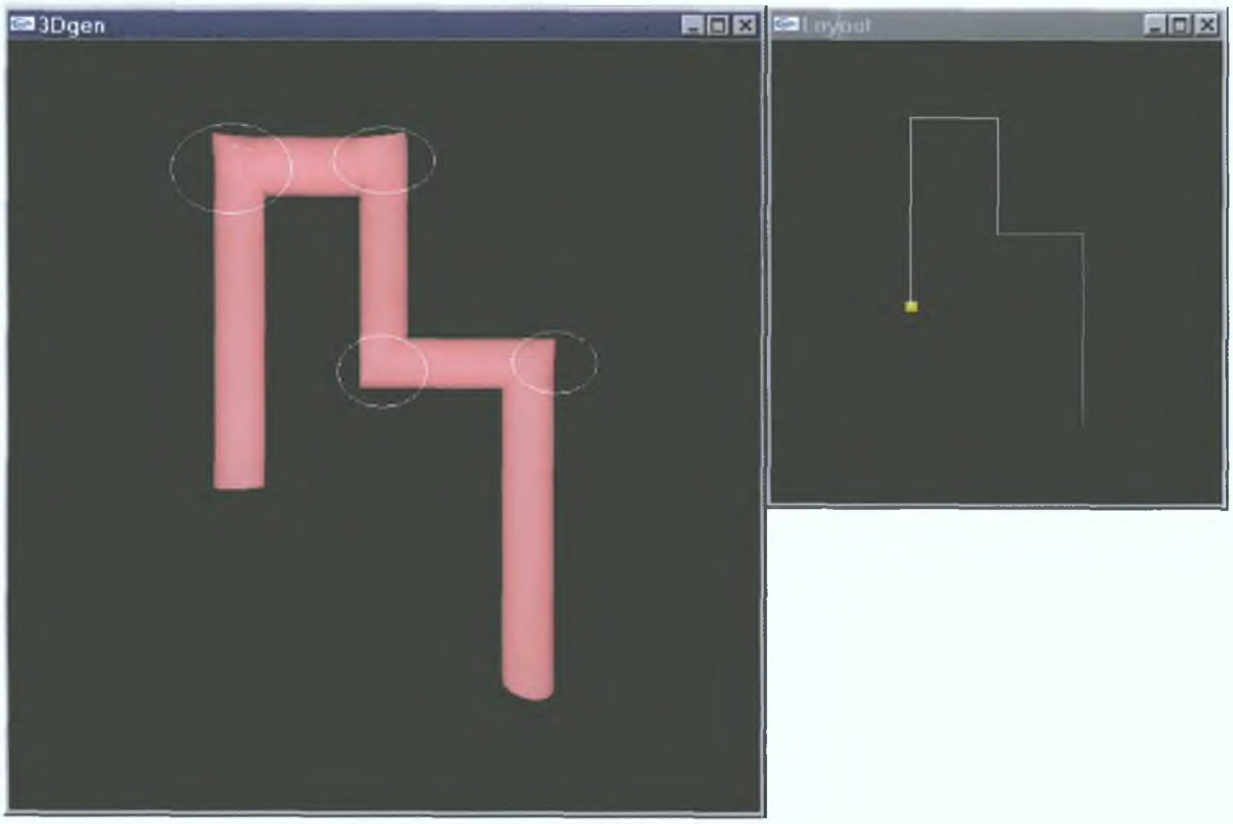


Figure 5.15 Overshoot error

As shown in figure 5.15 the white circles denote the overshoot error. While developing the algorithm of 3D*gen* software only smooth bends were considered since almost in all industrial pipe networks all of the bends have a smooth curvature. When the data for creating a sharp bend was fetched the layout window showed a discontinuity in the virtual exploration path. To counteract this problem the algorithm was altered in such a way that at the instant of a sharp bend the surface polygons in the previous segment of the model and the surface polygons in the next segment of the model were projected to create the sharp corner.

Since the amount of data available to the software with respect to bend section was not completely descriptive, it results in either incomplete bend section of the model or the bend overshoot (because of extension of the polygons in the previous and next segments). Increasing the description of the model can rectify this problem.

The format of the data file specified in table 4.1 was revised to include one more column of the data to solve the overshoot problem. The revised format for the data file is as shown in table 5.1

X coordinate	Y coordinate	Z coordinate	Angle of the face with respect to world coordinate system	Major Radius of the current face	Minor radius of the current face	RED value of colour coordinate	GREEN value of colour coordinate	BLUE value of colour coordinate

Table 5.1 Revised data format

The slice or face of the model is defined by only one radius. If this data is replaced with both major and minor radius data the vertices will be calculated using the ellipse equations which results in defining the slice or face with greater level of detail (LOD).

5.3.3 Model 3

Figures 5.10 and 5.11 show the internal view of model 3 generated by both the operating systems. Figure 5.10 was the result of the model generated in Microsoft windows operating system. The result is identical with Figure 5.11, which was generated in Redhat Linux operating system.

The data for the model is given in appendix C. The data for the model created in such a way that the model was generated only using the positive axes of the world coordinate system. The generated model can be rotated with respect to the origin and world coordinate system to explore the external surfaces of the model.

While rotating the model the vertex normals of various vertices were updated with respect to the current position of the model avoid errors in generating the proper light and shade effects.

Figures 5.12 and 5.13 shows the external view of model 3 generated by both the operating systems. Again the outputs are nearly identical. Those results also indicate the overshoot error while generating the model.

As mentioned earlier modifying the data format of the input data file can rectify this error. By replacing the radius of the slice with major radius and minor radius of the slice the vertices were calculated using ellipse equations which gives a greater level of detail (LOD) of the bend section when compared to the earlier data format.

The white marks along the inner surface of the model act as a reference which creates the illusion of the movement while the camera is moving along the exploration path or centreline path. Since the surface of the model are smooth and symmetrical the user would not be able feel the movement of the camera inside the model and this is rectified by introducing white marks along the inner surface.

Because of the Linux window system and its multi-processing feature the model was generated quickly when compared to the processing time in Microsoft windows operating system.

The refresh rate or the frame rate of the window (main window and layout window) can be increased with the availability of more display memory. The speed with which the model was generated and displayed in the window also depends on the processing of the microprocessor in the desktop PC.

Advanced video graphics cards are available which has in-built processor for hardware acceleration along with display memory. With dedicated processor in the graphics card the model can be displayed much faster compared to other graphics cards.

The software was not tested in operating systems such as Mac, OS/2, Solaris etc. Since the source codes are operating system independent, the author believes that the source code should compile and execute in those operating systems as well.

The software can generate the model along or parallel to YZ plane only with respect to world coordinate system. But the algorithm can be modified to generate a model in any plane. The formulae for vertex A and D has to be slightly changed to accommodate the angular displacement in x-axis as well.

The above results prove

- Flexibility of the software
- Compatibility of 3D*gen*.

The software acts as a framework and more features can be built on top by adding various functions to the existing framework.

5.4 Areas of application

3D*gen* can be applied to create any hollow cylindrical objects and hence it can be used to visualise pipe networks. It can also be used for virtual endoscopy to certain extent. It is possible to write optional function codes to depict certain property or quality in the model, which can be combined with the main program.

The software can be used for virtual exploration of cylindrical models. It can be used in visualizing a tunnel, both internal and external. It can also be used to construct and visualise steam pipe network inside boilers in thermal power stations.

It can be used as an educational tool for visualisation and exploration of models like human digestives system. With little more added features the software can also be used for three-dimensional game programming.

CHAPTER 6

CHAPTER 6: CONCLUSION

6.1 Conclusion

3D*gen* is an user-friendly software that was developed by the author to create three-dimensional cylindrical models from data sets. The software was developed using OpenGL and C language. The model was generated by the construction of numerous surface polygons. Two windows were available in which the main window shows different views of the model and the layout window shows the current position of the camera in the model.

3D*gen* satisfied the following goals set out by this research project.

- Compact software
- Compatibility with various operating systems such as Microsoft Windows, Redhat Linux
- No additional software or files were required to run the software
- User friendly

6.2 Recommendations for future work

The software can be improved with

- Improvised data format
- Polygon simplification depending on the levels of detail to simplify the scene for rendering with compromising the quality
- Dynamic zoom – to zoom the view dynamically
- Introducing height gradient to the surface along with texture and material properties

The framework can also be use to

- Simulation arm movement
- Simulation of blood flow through arteries and veins

REFERENCES

References:

1. A. Exline, *"Computer graphics "* IEEE Potentials, Volume: 9 Issue: 2, 1990, Pages: 43 –45.

2. T.L. Kunii, "*Science of computer graphics*" Proceedings of Seventh Pacific Conference on Computer Graphics and applications, 1999, pages 1-3.

3. C. Machover, "*Four decades of computer graphics*" Computer Graphics and applications, IEEE, Volume 14 Issue: 6, 1994, pages 14-19.

4. I.E. Sutherland, "*Sketchpad: A man-machine graphical communication system*" TR-296, MIT Lincoln laboratory, Lexington, Mass., 1963.

5. T.E. Johnson, "*Sketchpad III, three dimensional graphical communication with a digital computer*", PhD thesis, MIT Lincoln laboratory, Mass., 1963.

6. L. Szirmay-Kalos, G. Márton, B. Dobos, T. Horváth, P. Risztics, E. Kovács, "*Theory of Three Dimensional Computer Graphics*" Publishing House of the Hungarian Academy of Sciences, 1995.

7. R. A. Earnshaw, D. Watson, "*Animation and scientific visualization – tools and applications*", Academic press, 1993.

8. ACM SIGGRAPH Special report, "Report on the First Joint European Commission/National Science Foundation Advanced Research Workshop", 1999.

9. R. B. McMaster, K. S. Shea. "*Generalization in Digital Cartography*". Assoc. of American Geographers, Washington, D.C., 1992.

10. F. Hamit, "*Virtual Reality and the Exploration of Cyberspace*" Sams Publishing, 1993

11. J. Vince, "*Virtual reality systems*", Addison – Wesley, 1995.

12. K. Brodie, N. El-Khalili, Y. Li, "*Using web-based computer graphics to teach surgery*", Journal of Computers and Graphcis, Vol. 24, 2000, pages 157-161.

13. B. Wood, P. Razavi, "*Virtual Endoscopy: A promising new technology*", American family physician, Vol. 66(1), 2002, Pages 107-112.

14. R. A. Robb, "Virtual (computed) endoscopy: Development and evaluation using the visible human dataset", In Proc. of Visible Human Project Conference '95, 1995, pages 221-230.

15. M. Blank, W. A. Kalender, "*Medical volume exploration: gaining insights virtually*", European Journal of Radiology, 33, 2000, Pages 161-169.

16. L. Hong, S. Muraki, A. Kaufmann, D. Bartz, T. He, "*Virtual Voyage: Interactive navigation in the human colon*", In proceedings of ACM SIGGRAPH '97, pages 27-34.

17. R. D. Swift, A. P. Kiraly, A. J. Sherbondy, A. L. Austin, E. A. Hoffman, G. McLennan, W. E. Higgins, "*Automatic axis generation for virtual bronchoscopic assessment of major airway obstructions*", Journal of Computerized Medical Imaging and Graphics, Vol. 26, 2002, pages 103-118.

18. J. Brown, "*Enabling educational collaboration – a new shared reality*", journal of Computers and Graphics, Vol. 24, 2000, pages 289-292.

19. J. Foley, "*Getting There: The Ten Top Problems Left*", IEEE Computer Graphics and Applications, 20(1), 2000, pages 66-68.

20. F. P. Brooks, "*Walkthrough - A dynamic graphics system for simulating virtual buildings*", In Proceedings of ACM Workshop on Interactive 3D Graphics, UNC Chapel Hill, 1986.

21. J. H. Clark, "*Hierarchical Geometric Models for Visible Surface Algorithms*" Communications of the ACM 19(10), 1976, pages 547-554.

22. C. Erikson, D. Manocha, "*Hierarchical Levels of Detail for Fast Display of Large Static and Dynamic Environments*", UNC Chapel Hill Computer Science Technical Report TR00-012, 2000.

23. K. E. Hoff III, "Faster 3D Game Graphics by Not Drawing What Is Not Seen", ACM Crossroads, issue 3.4, 1997.

24. P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, G. A. Turner, "*Real-Time, Continuous Level of Detail Rendering of Height Fields*", Proceedings of ACM SIGGRAPH 96, 1996, pages 109-118.

25. B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder, "*Fast Rendering of Complex Environments Using a Spatial Hierarchy*", In Proceedings of the Graphics Interface '96, 1996, pages 132-141.

26. H. Pfister, M. Zwicker, J. van Baar and M. Gross, "*Surfels: Surface Elements as Rendering Primitives*", Proceedings of SIGGRAPH 2000, 2000, pages 335-342.

27. S. J. Teller, C. H. Séquin, "*Visibility preprocessing for interactive walkthroughs*" In proceedings of SIGGRAPH'91, 1991, vol. 25, pages 61-69.

28. D. Brodsky, B. Watson, "*Model Simplification Through Refinement*", In Proceedings of Graphics Interface May 2000, Montreal, Canada, pages 221-228.

29. A. Gresho, R. M. Gray, "*Vector Quantization and Signal Compression*", Kluwer academic publishers, Norwell, Mass., 1992.

30. C. Erikson, D. Manocha, "*GAPS: General and Automatic Polygonal Simplification*", UNC Chapel Hill Computer Science Technical Report #TR98-033, 1998.

31. J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W.Wright, "*Simplification Envelopes*" In Proceedings of ACM SIGGRAPH '96, 1996, pages 119-128.

32. W. J. Schroeder, J. A. Zarge, W. E. Lorensen, "*Decimation of Triangle Meshes*", In Proceedings of Computer Graphics, ACM SIGGRAPH'92, 1992, volume 26, pages 65-70.

33. G. Turk, "*Re-tiling polygonal surfaces*", In Proceedings of Computer Grpahics, ACM SIGGRAPH'92, 1992, volume 26, pages 55-64.

34. H. Hoppe, T, DeRose, T. Duchamp, J. MacDonald, W. Stuezle. "*Mesh optimization*", In Proceedings of Computer Graphics, ACM SIGGRAPH'93, 1993, volume 27, pages 19-26.

35. A. Varshney, "*Hierarchical geometric approximations*", PhD thesis TR-050-1994, Department of Computer Science, UNC, Chapel Hill, 1994.

36. G. Li, B. Watson, "*Semiautomatic Simplification*", In Proceedings of ACM Symposium on Interactive 3D Graphics, 2001, pages 43-48.

37. K. Low, T. Tan, "*Model Simplification using Vertex-Clustering*", In Proceedings of Symposium on Interactive 3D Graphics '97, 1997, pages 75-82.

38. P.Lindstrom, G. Turk, "*Fast and Memory Efficient Polygonal Simplification*", In Proceedings of IEEE Visualization '98, 1998, pages 279-286.

39. P. W. C. Maciel, P. Shirley, "*Visual Navigation of Large Environments Using Textured Clusters*", In Proceedings of Symposium of Interactive 3D Graphics, 1995, pages 95-102.

40. R. Westermann, T. Ertl, "*Efficiently Using Graphics Hardware in Volume Rendering Applications*", In proceedings of ACM SIGGRAPH'98, 1998, pages 169-177.

41. OpenGL Architecture Review Board, Silicon Graphics Inc., "*OpenGL Reference Manual*", Addison – Wesley Publishing Company, Reading, 1992.

42. S. Guthe, M. Wand, J. Gosner, W. Straßer, "*Interactive Rendering of Large Volume Data Sets*", In proceedings of Visualization'02, 2002, pages 53-60.

43. D. Aliga, J. Cohen, A. Wilson, H. Zhang, C Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, D. Manocha, "*A Framework for the Real-Time Walkthrough of Massive Models*", UNC Chapel Hill Computer Science Techinical Report UNC TR#98-013, 1998.

44. E. LaMar, B. Hamann, K. Joy, "*A Magnification Lens for Interactive Volume Visualization*", In Proceedings of ninth Pacific conference on Computer Graphics and Applications, 2001, pages 223-232.

45. J. C. Xia, J. El-Sana, A. Varshey, "*Adaptive Real-time level-of-detail based Rendering for Polygon Models*", In IEEE Transactions on Visualization and Computer Graphics, volume 3 number 2, June 1997, pages 171-183.

46. G. Schaufler, W Stürzlinger, "*Generating Multiple Levels of Detail from Polygonal Geometry Models*", In Proceedings of Virtual Environments'95(Eurographics workshop on Virtual Environments), 1995, pages 33-41.

47. J. Chhugani, B. Purnomo, S. Krishnan, J. Cohen, S. Kumar, "*vLOD: A Scalable System for Interactive Walkthroughs of Very Large Virtual Environments*", Technical Report JHU-CS-GL03-3, Computer Graphics Group, John Hopkins University, 2003.

48. J. Cohen, D. Luebke, N. Duca, B. Schubert, "*GLOD: Level of Detail for the Masses*", Technical Report, JHU-CS-GL03-4, Computer Graphics Group, John Hopkins University, 2003.

49. D. Schmalstieg, G. Schaufler, "*Smooth Levels of Detail*", In Proceedings of VRAIS'97, 1997, pages 12-19.

50. T. A. Funkhouser, C. H. Sèquin, "*Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments*", In Proceedings of SIGGRAPH'93, Computer Graphics proceedings, ACM SIGGRAPH, August 1993, pages 65-74.

51. T. A. Funkhouser, C. H. Sèquin, S. Teller, "*Management of Large Amounts of Data in Interactive Building Walkthorughs*", In Proceedings of Symposium on Interactive 3D graphics (ACM SIGGRAPH), 1992, pages 11-20.

52. P. Volino, N. M. Thalmann, "*Fast Geometrical Wrinkles on Animated Surfaces*", In Seventh International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media'99, 1999.

53. L. Hong, Z. Liang, A. Viswambharan, A. Kaufman, and M. Wax, "*Reconstruction and Visualization of 3D Models of Colonic Surface*". IEEE Transactions on Nuclear Science, Vol. 44, No. 3, 1997, pages 1297-1302.

54. S. Beckhaus, F. Ritter, T. Strothotte, "*Guided Exploration with Dynamic Potential Fields: The CubicalPath System*", Computer Graphics Forum, Vol. 20(4), 2001, pages. 201-210.

55. T P Pridmore, D Cooper and N Taylor, "*Estimating Camera Orientation from Vanishing Point Location During Sewer Surveys*", Automation in Construction, 5, 1997, pages 407-419.

56. 3DStudio MAX user's guide, AutoDesk Inc.1997.

57. Adobe Systems Incorporated, "*PostScript Language Reference Manual*", Addison-Wesley, Reading, Mass., 1992.

58. A. Nye, "*X Window System User's Guide, volume 3 of The Definitive Guides to the X Window System*", O'Reilly and Associates, Sebastapol, Ca., 1987.

59. International Standards Organization. International standard information processing systems - computer graphics - graphical kernel system for three dimensions (GKS-3D) functional description. Technical Report ISO Document Number 9905:1988(E), American National Standards Institute, New York, 1988.

60. PHIGS+ Committee, Andries van Dam, chair. PHIGS+ functional description, revision 3.0. Computer Graphics, 22(3): pages 125-218, July 1988.

61. J. Foley, A. Van Dam, J. Hughes, S. Feiner, and R.Philips, "*Introduction to Computer Graphics*", Addison Wesley, Reading, Mass, 1997.

62. J. Neider, M. Woo, T. Davis, "*OpenGL Programming Guide or the redbook*", Addison – Wesley publishing company, Reading, 1994.

63. M. Brown, "*Understanding PHIGS*", Megatek Corporation, San Diego, CA, 1985.

64. V. B. Anand, "*Computer Graphics and geometric modeling for Engineers*", John Wiley & sons Inc., 1993.

65. P. Womack, "*PEX protocol specification and encoding, version 5.1P*", The X Resource, Special Issue A, May 1992.

66. J. Stevenson, "*PEXlib specification and C language binding, version 5.1P*", The X Resource, Special Issue B, September 1992.

67. The VRML Consortium, "*The virtual reality modeling language specification*", web site, 1999. http://www.web3d.org.

68. M. J. Kilgard, "*Realizing opengl: Two implementations of one architecture*", In Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1997.

69. A. Watt, "*3D Computer graphics*", Addison – Wesley publishing company, 1995.

70. Advanced graphics programming techniques using OpenGL, ACM SIGGRAPH course notes, 1999.

71. P. Haeberli, K. Akeley. "*The accumulation buffer: Hardware support for high-quality rendering*" In Proceedings of SIGGRAPH '90, 1990, pages 309-318.

72. W. Li, C. Chang, K. Hsu, M. Kuo, D. Way, "*A PC-based distributed multiple display virtual reality system*", Journal of Displays, Vol. 22(5), 2001, pages 177-181

73. D. P. Luebke, "*A Developer's Survey of Polygonal Simplification Algorithms*" IEEE Computer Graphics and Applications 21(3), 2001, pages 24-35

74. P. Heckbert, M. Garland, "*Survey of Polygonal surface simplification Algorithms*", Technical Report, Department of Computer Science, Carnegie Mellon University, 1997

APPENDIX

Appendix A: Instructions to compile and run 3Dgen:

The following files are required to compile the source code along with C compiler.

The following header files should be in /include directory with compiler

gl.h

glu.h

glut.h

The following library files should be in /lib directory with compiler

glu32.lib

glut.lib

glut32.lib

opengl32.lib

For Microsoft Windows based operating system the following files should be in C:/windows/system or C:/winnt/system32

glu.dll

glu32.dll

glut.dll

glut32.dll

opengl.dll

opengl32.dll

Once the *.exe* file was generated in Microsoft Windows based operating system only the above *.dll* files were required to run the software.

All the above files were available for free in World Wide Web.

Appendix B: Model 2 data

X	Y	Z	Angle	Radius	Red	Green	Blue
0.00000	0.00000	0.00000	-90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	1.00000	0.00000	-90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	2.00000	0.00000	-90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	3.00000	0.00000	-90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	4.00000	0.00000	-90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	5.00000	0.00000	-90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	6.00000	0.00000	-90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	6.50000	0.00000	-45.00000	0.70600	0.97700	0.50700	0.60800
0.00000	6.50000	0.50000	0.00000	0.50000	0.97700	0.50700	0.60800
0.00000	6.50000	1.50000	0.00000	0.50000	0.97700	0.50700	0.60800
0.00000	6.50000	2.50000	0.00000	0.50000	0.97700	0.50700	0.60800
0.00000	6.50000	3.00000	45.00000	0.70600	0.97700	0.50700	0.60800
0.00000	6.00000	3.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	5.00000	3.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	4.00000	3.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	3.00000	3.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	2.50000	3.00000	45.00000	0.70600	0.97700	0.50700	0.60800
0.00000	2.50000	3.50000	0.00000	0.50000	0.97700	0.50700	0.60800
0.00000	2.50000	4.50000	0.00000	0.50000	0.97700	0.50700	0.60800
0.00000	2.50000	5.50000	0.00000	0.50000	0.97700	0.50700	0.60800
0.00000	2.50000	6.00000	45.00000	0.70600	0.97700	0.50700	0.60800
0.00000	2.00000	6.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	1.00000	6.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	0.00000	6.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	-1.00000	6.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	-2.00000	6.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	-3.00000	6.00000	90.00000	0.50000	0.97700	0.50700	0.60800
0.00000	-4.00000	6.00000	90.00000	0.50000	0.97700	0.50700	0.60800

Appendix C: Model 3 data

X	Y	Z	Angle	Radius	Red	Green	Blue
0.00000	0.00000	0.00000	0.00000	0.50000	0.00000	1.00000	1.00000
0.00000	0.00000	1.00000	0.00000	0.50000	0.00000	1.00000	1.00000
0.00000	0.00000	2.00000	0.00000	0.50000	0.00000	1.00000	1.00000
0.00000	0.00000	2.50000	-45.00000	0.70600	0.00000	1.00000	1.00000
0.00000	0.50000	2.50000	-90.00000	0.50000	0.00000	1.00000	1.00000
0.00000	1.50000	2.50000	-90.00000	0.50000	0.00000	1.00000	1.00000
0.00000	2.50000	2.50000	-90.00000	0.50000	0.00000	1.00000	1.00000
0.00000	3.50000	2.50000	-90.00000	0.50000	0.00000	1.00000	1.00000
0.00000	4.50000	2.50000	-90.00000	0.50000	0.00000	1.00000	1.00000
0.00000	5.50000	2.50000	-90.00000	0.50000	0.00000	1.00000	1.00000
0.00000	6.00000	2.50000	-45.00000	0.70600	0.00000	1.00000	1.00000
0.00000	6.00000	3.00000	0.00000	0.50000	0.00000	1.00000	1.00000
0.00000	6.00000	4.00000	0.00000	0.50000	0.00000	1.00000	1.00000
0.00000	6.0000	5.00000	0.00000	0.50000	0.00000	1.00000	1.00000

Appendix D: Source code

```
#ifndef __FLAT__
        #include <windows.h>
#endif
        // Remove the above three lines of the code to compile it in operating systems other
        // than Microsoft Windows

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define WIDTH 500 // window size declaration
#define HEIGHT 500

#define RED 0  // color declaration
#define GREEN 0
#define BLUE 0
#define ALPHA 1

#define KEY_ESC 27 // declaration of keys
#define KEY_S 115
#define KEY_G 103
#define KEY_UP 101
#define KEY_DOWN 103
#define KEY_LEFT 102
#define KEY_RIGHT 100
#define KEY_X 120
#define KEY_Y 121
#define KEY_Z 122

GLuint bend1,lis1; // call list declaration

#define DELTA 5  // constant definitions
#define pi 3.14158265
int x=0,y=0,z=0,speed=0;
GLfloat rotateX=0,rotateY=0,rotateZ=0; // variables for rotating the model in different
                                                    // axes
int L_UP,L_DOWN,L_RIGHT,L_LEFT,stillview;
float *p, pp, countx, county, countyy, countz=0.0, theta1, lamda, yrot_prev, zrot_prev,
yrot_cur, zrot_cur;

void init_scene();
void render_scene();

GLvoid initGL(); // initial settings for window and diff. parameters
GLvoid initGL1();
GLvoid window_reshape1(GLsizei width, GLsizei height);
GLvoid window_display1();
GLvoid window_idle1();
GLvoid window_display(); // window display function
GLvoid window_reshape(GLsizei width, GLsizei height); // window reshape function
GLvoid window_idle(); // window idle function
GLvoid window_key(unsigned char key,int x, int y); // keyboard function - for arrow
                                                    // keys
GLvoid window_special_key(int key, int x, int y);  // keyborad function - for special
                                                    // keys

GLvoid move(); // move function - to move camera
GLint recposnum=0; // position of current aim point of camera with respect to file
                    // datatest.txt

GLvoid recnum(); // function to count the number of records in datatest.txt file
GLint recnumonce, /* variable to check whether recnum() function is called */
 rectotal=0; // holds total no. of records in datatest.txt file

GLfloat* readdata(GLfloat*); // function to read aim and camera coordinates from
                              // datatest.txt
GLvoid lightsettings(); // light settings function
GLvoid func_mainmenu(int);
GLvoid func_view_menu(int);
GLvoid func_view_internal_menu(int);
GLvoid func_view_internal_viewdir_menu(int);
GLvoid func_view_internal_startpos_menu(int);
GLvoid func_view_external_menu(int);
GLvoid func_zoom_menu(int);
GLvoid func_turn_menu(int);
GLvoid func_modeltyp_menu(int);
GLvoid startposition(GLint);
GLfloat finval(GLfloat, GLfloat, int);
GLvoid
fncal(GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,FILE*);
GLvoid sc1();
GLvoid rs1();
GLvoid ni(GLfloat,GLfloat,GLfloat);


FILE *fp1; // file pointer to handle datatest.txt
GLfloat aim_x, aim_y, aim_z, view_x_cur, view_y_cur, view_z_cur, theta=0, step_theta=10.0;
// aim and camera coordinates
GLint mfront=0; // variable to check whether the movement is forward or backward
                 // while pressing arrow keys
GLfloat upx=0,upy=1,upz=1,
        maxx,minx,maxy,miny,maxz,minz,
        midx,midy,midz;
GLint zoovar,dstyle;
char zoostat='n',cee='i';
char viewdirection;
int win1,win2,psn=1;


void
myblock(double,double,double,double,double,double,double,double,double,double,FILE*,FIL
E*,GLfloat,GLfloat,GLfloat); //function which draws the model
```

```
int main(int argc, char **argv)
{
  int main_menu,view_menu,view_internal_menu,view_internal_viewdir_menu,
          view_internal_startpos_menu,view_external_menu,zoom_menu;
        glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH); // initializing
                                                              // display
                                                              // mode


  glutInitWindowSize(300,300);
  glutInitWindowPosition(510,0);
  win1=glutCreateWindow("Layout");

  glutInitWindowSize(WIDTH,HEIGHT); // initialise window size
  glutInitWindowPosition(0,0); // initialise window position
  win2=glutCreateWindow("3Dgen"); // window creation and naming

  if (recnumonce==0)
  {
    recnum();
        recnumonce=1;
  } // to findout no. of records in datatest.txt

  if ((fp1=fopen("datatest.txt", "r"))==NULL)
  {
    printf("cannot open file");
        exit(1);
  } // opening the data file


  glutSetWindow(win1);
  initGL1();
  sc1();
  glutDisplayFunc(&window_display1);
  glutReshapeFunc(&window_reshape1);
  glutIdleFunc(&window_idle1);


 glutSetWindow(win2);
  initGL(); // initial window settings and diff. parameters
  init_scene();  // draws the model

  glutDisplayFunc(&window_display);  // window display function call
  glutReshapeFunc(&window_reshape);  // window reshape function call
  glutIdleFunc(&window_idle);  // window idle function call
  glutKeyboardFunc(&window_key); // function call when any of the arrow keys are
                                 // pressed
  glutSpecialFunc(&window_special_key); // function call when any of the special set
                                        // of keys are pressed

  zoom_menu=glutCreateMenu(func_zoom_menu);
        glutAddMenuEntry("Zoom IN",21);
        glutAddMenuEntry("Zoom OUT",22);


   view_external_menu=glutCreateMenu(func_view_external_menu);
        glutAddMenuEntry("Front",121);
        glutAddMenuEntry("Back",122);
        glutAddMenuEntry("Left",123);
        glutAddMenuEntry("Right",124);

  view_internal_viewdir_menu=glutCreateMenu(func_view_internal_viewdir_menu);
        glutAddMenuEntry("Front",1111);
        glutAddMenuEntry("Back",1112);

  view_internal_startpos_menu=glutCreateMenu(func_view_internal_startpos_menu);
        glutAddMenuEntry("Start",1121);
        glutAddMenuEntry("End",1122);

  view_internal_menu=glutCreateMenu(func_view_internal_menu);
        glutAddSubMenu("View Direction",view_internal_viewdir_menu);
        glutAddSubMenu("Start position",view_internal_startpos_menu);

  view_menu=glutCreateMenu(func_view_menu);
        glutAddSubMenu("Internal",view_internal_menu);
        glutAddSubMenu("External",view_external_menu);
        main_menu=glutCreateMenu(func_mainmenu);
        glutAddSubMenu("View",view_menu);
        glutAddSubMenu("Zoom",zoom_menu);
        glutAddMenuEntry("Exit",4);
  glutAttachMenu(GLUT_RIGHT_BUTTON);

  glutMainLoop(); // loops the above routine

  return 1;
}



GLvoid initGL()
{
  glClearColor(RED,GREEN,BLUE,ALPHA);
  L_UP=0;
  L_DOWN=0;
  L_RIGHT=0;
  L_LEFT=0;
  pp=0;
  p=&pp;
  stillview=0;
  viewdirection='F';
  glClearDepth(1.0f);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_SMOOTH);
  glEnable(GL_COLOR_MATERIAL);
}
```

```
void init_scene()
{
        GLfloat rec11_x, rec11_y, rec11_z, rec11_angle, rec12_x, rec12_y, rec12_z,
                        rec12_angle, rec12_len=0;
        GLfloat startradius, endradius, rec1_col_R, rec1_col_G, rec1_col_B, rec2_col_R,
                        rec2_col_G, rec2_col_B;
  FILE *nr,*npp;

  if ((nr=fopen("nrd.txt","r"))==NULL)
  {
    printf("Cannot open normal dat file");
          exit(1);
  }
  if ((npp=fopen("nrd1.txt","w"))==NULL)
  {
    printf("Cannot open normal dat file");
          exit(1);
  }


  GLUquadricObj *quadric;

  bend1=glGenLists(1);
  glNewList(bend1, GL_COMPILE);
    quadric=gluNewQuadric();
                  gluQuadricDrawStyle(quadric,GLU_FILL);
          glColor3f(0.977,0.407,0.408);

          while (!feof(fp1))
          {
            if (theta1>0) glPushMatrix();
                  if (theta1==0) fscanf(fp1,"%f %f %f %f %f %f %f %f", &rec11_x,
                                            &rec11_y, &rec11_z, &rec11_angle,
                                            &startradius, &rec1_col_R, &rec1_col_G,
                                            &rec1_col_B); // first record

                  fscanf(fp1,"%f %f %f %f %f %f %f %f", &rec12_x, &rec12_y,
                          &rec12_z, &rec12_angle, &endradius, &rec2_col_R,
                          &rec2_col_G, &rec2_col_B); // scanning next record


myblock(rec11_x,rec11_y,rec11_z,rec11_angle,rec12_x,rec12_y,rec12_z,rec12_angle,startra
dius,endradius,nr,npp,rec1_col_R,rec1_col_G,rec1_col_B);
                  // function call to draw first block based on twopoints and two radii

                  rec11_x=rec12_x;  // storing the rec12 values in rec11
                  rec11_y=rec12_y;
                  rec11_z=rec12_z;
                  rec11_angle=rec12_angle;
                  startradius=endradius;
                  rec1_col_R=rec2_col_R;
                  rec1_col_G=rec2_col_G;
                  rec1_col_B=rec2_col_B;
                  theta1++;
```

```
                glPopMatrix();
                fprintf(npp,"\n");
            }
            gluDeleteQuadric(quadric);
            glEndList();
}

GLvoid move()
{
  GLfloat viewcoordi[3],aimcoordi[3];

  viewcoordi[0]=view_x_cur;
  viewcoordi[1]=view_y_cur;
  viewcoordi[2]=view_z_cur;
  aimcoordi[0]=aim_x;
  aimcoordi[1]=aim_y;
  aimcoordi[2]=aim_z;

  if(viewdirection=='F')
  {
            if(mfront==1)
            {
                    viewcoordi[0]=aimcoordi[0];
                    viewcoordi[1]=aimcoordi[1];
                    viewcoordi[2]=aimcoordi[2];
                    readdata(aimcoordi);
            }
            else if (mfront==-1)
            {
                    aimcoordi[0]=viewcoordi[0];
                    aimcoordi[1]=viewcoordi[1];
                    aimcoordi[2]=viewcoordi[2];
                    readdata(viewcoordi);
            }
  }
  else if (viewdirection=='B')
  {
            if(mfront==1)
            {
                    aimcoordi[0]=viewcoordi[0];
                    aimcoordi[1]=viewcoordi[1];
                    aimcoordi[2]=viewcoordi[2];
                    readdata(viewcoordi);
            }
            else if (mfront==-1)
            {
                    viewcoordi[0]=aimcoordi[0];
                    viewcoordi[1]=aimcoordi[1];
                    viewcoordi[2]=aimcoordi[2];
                    readdata(aimcoordi);
            }
  }
  view_x_cur=viewcoordi[0];
  view_y_cur=viewcoordi[1];
  view_z_cur=viewcoordi[2];
  aim_x=aimcoordi[0];
  aim_y=aimcoordi[1];
  aim_z=aimcoordi[2];
}


GLvoid recnum()
{
  FILE *fp0,*wrf;
  int ctrr=0;
  float xx1,yy1,zz1=0.0,angle1,radius1,Rcol,Gcol,Bcol,
              xx2,yy2,zz2=0.0,angle2,radius2,Rcol2,Gcol2,Bcol2;
  if ((fp0=fopen("datatest.txt","r"))==NULL)
  {
    printf("Cannot open file");
          exit(1);
  }
  if ((wrf=fopen("nrd.txt","w"))==NULL)
  {
    printf("Cannot open normal dat file");
          exit(1);
  }

  while(!feof(fp0))
  {
          if(ctrr==0)
          {
             fscanf(fp0,"%f %f %f %f %f %f %f %f", &xx1, &yy1, &zz1, &angle1, &radius1,
&Rcol, &Gcol, &Bcol);
                 view_x_cur=xx1;
                 view_y_cur=yy1;
                 view_z_cur=zz1;
                 minx=xx1;
                 maxx=xx1;
                 miny=yy1;
                 maxy=yy1;
                 minz=zz1;
                 maxz=zz1;
          }
          else if (ctrr==1)
          {
                 aim_x=xx1;
                 aim_y=yy1;
                 aim_z=zz1;
          }
          fscanf(fp0,"%f %f %f %f %f %f %f %f", &xx2, &yy2, &zz2, &angle2, &radius2, &Rcol2,
                    &Gcol2, &Bcol2);
          fncal(xx1,yy1,zz1,angle1,radius1,xx2,yy2,zz2,angle2,radius2,wrf);
          rectotal=rectotal+1;
          ctrr++;
          minx=finval(minx,xx2,0);
          maxx=finval(maxx,xx2,1);
          miny=finval(miny,yy2,0);
          maxy=finval(maxy,yy2,1);
          minz=finval(minz,zz2,0);
```

```
            maxz=finval(maxz,zz2,1);
            xx1=xx2;
            yy1=yy2;
            zz1=zz2;
            angle1=angle2;
            radius1=radius2;
            Rcol=Rcol2;
            Gcol=Gcol2;
            Bcol=Bcol2;
    }
    fclose(fp0);
    fclose(wrf);
    rectotal--;
    printf("Total no. of recs. = %d\n",rectotal);
    midx=(minx+maxx)/2;
    midy=(miny+maxy)/2;
    midz=(minz+maxz)/2;
    recposnum=2;
    return;
}

GLvoid fncal(GLfloat x1, GLfloat y1, GLfloat z1, GLfloat ang1, GLfloat r1, GLfloat x2, GLfloat
y2, GLfloat z2, GLfloat ang2, GLfloat r2, FILE *tpp)
{
        int i,INCR=1;
        GLfloat vx,vy,vz,
                        ax,ay,az,
                        bx,by,bz,
                        cx,cy,cz,veclen;
        for (i=0;i<360;i=i+INCR)
        {
                ax=x1+r1*cos(i*pi/180);
                ay=y1+r1*(sin(i*pi/180))*(cos(ang1*pi/180));
                az=z1+r1*(sin(i*pi/180))*(sin(ang1*pi/180));

                bx=x1+r1*cos((i+INCR)*pi/180); // vertex of circle with centre x1,y1,z1
                                                // at angle i+incr
                by=y1+r1*(sin((i+INCR)*pi/180))*(cos(ang1*pi/180));
                bz=z1+r1*(sin((i+INCR)*pi/180))*(sin(ang1*pi/180));

                cx=x2+r2*cos((i+INCR)*pi/180);  // vertex of circle with centre x2,y2,z2
                                                // at angle i+incr
                cy=y2+r2*(sin((i+INCR)*pi/180))*(cos(ang2*pi/180));
                cz=z2+r2*(sin((i+INCR)*pi/180))*(sin(ang2*pi/180));

                vx=(((ay-cy)*(az-bz))-((ay-by)*(az-cz)));
                vy=(((ax-bx)*(az-cz))-((ax-cx)*(az-bz)));
                vz=(((ax-cx)*(ay-by))-((ax-bx)*(ay-cy)));

                veclen = sqrt((vx*vx)+(vy*vy)+(vz*vz));
                vx=vx/veclen;
                vy=vy/veclen;
                vz=vz/veclen;
                fprintf(tpp,"%f %f %f",vx,vy,vz);
                if (i<359)
				{
					fprintf(tpp," ");
				}
			}
			fprintf(tpp,"\n");
}

GLfloat finval(GLfloat va, GLfloat vb, int tem)
{
			GLfloat fval;
			if (tem==0)
			{
				if (va>vb)
					fval=vb;
				else
					fval=va;
			}
			else if (tem==1)
			{
				if (va>vb)
					fval=va;
				else
					fval=vb;
			}
			return (fval);
}


GLfloat * readdata(GLfloat *tempptr)
{
  GLfloat Rc,Gc,Bc,dist,ang1;
  int datacntr;
  FILE *fp0;

  if ((fp0=fopen("datatest.txt","r"))==NULL)
  {
    printf("Cannot open file");
	     exit(1);
  }
  for (datacntr=1;datacntr<=recposnum;datacntr++)
  {
	    fscanf(fp0,"%f %f %f %f %f %f %f %f", &tempptr[0], &tempptr[1], &tempptr[2], &ang1,
				&dist, &Rc, &Gc, &Bc);
  }
  fclose(fp0);

  return (tempptr);

}

GLvoid lightsettings()
{
			GLfloat globalambient[]={0.65,0.65,0.65,1.0},
			diff0[]={0.5,0.5,0.5,1.0},
			ambi0[]={0.0,0.0,0.0,1.0}, shine=10.0,
        speclight[]={0.5,0.5,0.5,1.0},
        atti[]={-1.5,-0.50,-0.50},spotcutoff=90,
        spotdir[]={aim_x,aim_y,aim_z},
        glemiss[]={0.0,0.0,0.0,1.0},
        pos0[]={0.0,0.0,0.0,1.0};


        pos0[0]=view_x_cur;
        pos0[1]=view_y_cur;
        pos0[2]=view_z_cur;

        glLightModelfv(GL_LIGHT_MODEL_AMBIENT,globalambient);
        glLightfv(GL_LIGHT0,GL_POSITION,pos0);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,diff0);
        glLightfv(GL_LIGHT0,GL_SPECULAR,speclight);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHTING);
        glMaterialfv(GL_FRONT,GL_SHININESS,&shine);
}

GLvoid window_display()
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
gluLookAt(view_x_cur,view_y_cur,view_z_cur,aim_x,aim_y,aim_z,upx,upy,upz);
                                                    // camera positon and aim coordinates
if (zoostat =='y')
{
        if (zoovar==1)
        {
                glScaled(2,2,2);
                glEnable(GL_NORMALIZE);
        }
        else if (zoovar==0)
        {
                glScaled(1.0,1.0,1.0);
                glEnable(GL_NORMALIZE);
        }
        zoostat='n';
}

lightsettings();
render_scene();
glutSwapBuffers();
}

GLvoid window_reshape(GLsizei width, GLsizei height)
{
  if (height==0) height=1;
  glViewport(0,0,width,height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(90,(GLdouble)width/(GLdouble)height,0.1,200);
  glMatrixMode(GL_MODELVIEW);
}
```

```
GLvoid window_key(unsigned char key,int x,int y)
{
  switch(key)
  {
   case KEY_ESC:  exit(1);
                  break;
        case KEY_X  :  rotateX=!rotateX;
                  glutPostRedisplay();
                                 break;
        case KEY_Y  :  rotateY=!rotateY;
                  glutPostRedisplay();
                                 break;
        case KEY_Z  :  rotateZ=!rotateZ;
                  glutPostRedisplay();
                                 break;
        default     :  printf("Pressing %d dosen't do anything \n", key);
                  break;
        }
}

GLvoid window_special_key(int key, int x,int y)
{
  switch (key)
  {
        case KEY_UP :  if (mfront==-1) recposnum=recposnum+1;
                                 mfront=1;
                                 recposnum=recposnum+1;
                                 move();
                                 glutPostRedisplay();
                                 glutPostWindowRedisplay(win1);
                                 break;

        case KEY_DOWN :  if (mfront==1) recposnum=recposnum-1;
                                 mfront=-1;
                                 recposnum=recposnum-1;
                                 move();
                                 glutPostRedisplay();
                                 glutPostWindowRedisplay(win1);
                                 break;

        case KEY_LEFT : speed=(speed-DELTA+360)%360;
                          glutPostRedisplay();
                          break;

        case KEY_RIGHT : speed=(speed+DELTA+360)%360;
                           glutPostRedisplay();
                           break;

        default  :  printf("Pressing %d doesn't do anything", key);
                    break;
        }
}
```

```
GLvoid window_idle()
{
if (rotateX) x=(x+speed+360)%360;
if (rotateY) y=(y+speed+360)%360;
if (rotateZ) z=(z+speed+360)%360;
if (speed>0 && (rotateX||rotateY||rotateZ))
glutPostRedisplay();
}

void myblock(double x1,double y1,double z1,double ang1,
         double x2,double y2,double z2,double ang2,
                         double r1,double r2,FILE *fnr,FILE *fnr2,GLfloat cr1,Glfloat
                                                         cg1,GLfloat cb1)
{
  int INCR=1,i,h,g,j,k;
  double ax,ay,az,bx,by,bz,cx,cy,cz,dx,dy,dz;
  GLfloat vx,vy,vz,
                        t1,t2,t3,t11,t12,t13,tt1,tt2,tt3,tt11,tt12,tt13,
                        s1[360][3],s2[360][3];

  float veclen=0,veclen2=0;
  FILE *fnr1;

  if ((fnr1=fopen("nrd.txt","r"))==NULL)
  {
    printf("Cannot open normal dat file");
         exit(1);
  }
  for (k=1;k<=psn;k++)
  {
         if (k==psn) for (h=0;h<360;h++)   fscanf(fnr1,"%f %f %f", &s1[h][0],
                                                    &s1[h][1], &s1[h][2]);

         for (h=0;h<360;h++)   fscanf(fnr1,"%f %f %f",&s2[h][0],&s2[h][1],&s2[h][2]);
  }

  for (i=0;i<360;i=i+INCR)
  {
         ax=x1+r1*cos(i*pi/180);  // vertex of circle with centre x1,y1,z1 at angle i
         ay=y1+r1*(sin(i*pi/180))*(cos(ang1*pi/180));
         az=z1+r1*(sin(i*pi/180))*(sin(ang1*pi/180));

     bx=x1+r1*cos((i+INCR)*pi/180); // vertex of circle with centre x1,y1,z1 at angle
                                                   // i+incr

     by=y1+r1*(sin((i+INCR)*pi/180))*(cos(ang1*pi/180));
     bz=z1+r1*(sin((i+INCR)*pi/180))*(sin(ang1*pi/180));

     cx=x2+r2*cos((i+INCR)*pi/180);  // vertex of circle with centre x2,y2,z2 at angle
                                                   // i+incr
     cy=y2+r2*(sin((i+INCR)*pi/180))*(cos(ang2*pi/180));
     cz=z2+r2*(sin((i+INCR)*pi/180))*(sin(ang2*pi/180));

     dx=x2+r2*cos(i*pi/180);  // vertex of circle with centre x2,y2,z2 at angle i
```

```
	dy=y2+r2*(sin(i*pi/180))*(cos(ang2*pi/180));
		dz=z2+r2*(sin(i*pi/180))*(sin(ang2*pi/180));

		if (i==0)
			g=359;
		else
			g=(i/INCR)-1;
		j=i/INCR;
		vx=s1[g][0]+s1[j][0]+s2[g][0]+s2[j][0];
		vy=s1[g][1]+s1[j][1]+s2[g][1]+s2[j][1];
		vz=s1[g][2]+s1[j][2]+s2[g][2]+s2[j][2];

		glBegin(GL_POLYGON); //joining the vertices of a,b(of 1st circle),c and d(of
					     // 2nd circle) to form a polygon

veclen = sqrt((vx*vx)+(vy*vy)+(vz*vz));
if (cee=='i')
	{
	glNormal3f(vx/veclen,vy/veclen,vz/veclen);

	  glColor3f(cr1,cg1,cb1);
	  glVertex3f(ax,ay,az);
	  glVertex3f(bx,by,bz);
	  glVertex3f(cx,cy,cz);
	  glVertex3f(dx,dy,dz);
	}
else
	{
	  glVertex3f(bx,by,bz);
	  glVertex3f(ax,ay,az);
	  glVertex3f(dx,dy,dz);
	  glVertex3f(cx,cy,cz);
	}
	  glEnd();
	  if ((i>180) && (i<185)==TRUE)
	  {
	  ax=x1+0.99*r1*cos(i*pi/180);
	  ay=y1+0.99*r1*(sin(i*pi/180))*(cos(ang1*pi/180));
	  az=z1+0.99*r1*(sin(i*pi/180))*(sin(ang1*pi/180));
    bx=x1+0.99*r1*cos((i+INCR)*pi/180);
	  by=y1+0.99*r1*(sin((i+INCR)*pi/180))*(cos(ang1*pi/180));
	  bz=z1+0.99*r1*(sin((i+INCR)*pi/180))*(sin(ang1*pi/180));
    cx=x2+0.99*r2*cos((i+INCR)*pi/180);
	  cy=y2+0.99*r2*(sin((i+INCR)*pi/180))*(cos(ang2*pi/180));
	  cz=z2+0.99*r2*(sin((i+INCR)*pi/180))*(sin(ang2*pi/180));
    dx=x2+0.99*r2*cos(i*pi/180);
	  dy=y2+0.99*r2*(sin(i*pi/180))*(cos(ang1*pi/180));
	  dz=z2+0.99*r2*(sin(i*pi/180))*(sin(ang1*pi/180));
	  t1=ax; t2=ay; t3=az; t11=bx; t12=by; t13=bz;
	  tt1=dx; tt2=dy; tt3=dz; tt11=cx; tt12=cy; tt13=cz;
if ((psn%2)==0)
{
	fprintf(fnr2,"i= %d g= %d j=%d\n",i,g,j);
	glBegin(GL_QUADS);
		glColor3f(1,1,1);
```

```
                glVertex3f(t1,t2,t3);
                glVertex3f(t11,t12,t13);
                glVertex3f(tt11,tt12,tt13);
                glVertex3f(tt1,tt2,tt3);
        glEnd();
}

          }
}
  psn++;
  fclose(fnr1);
}


GLvoid func_mainmenu(int menuoption)
{
        switch (menuoption)
        {
        case 1 :
                        break;
        case 2 :
                        break;
        case 3 :
                        break;
        case 4 : printf("\n SESSION CLOSED");
                        exit(0);
                        break;
        }
}

GLvoid func_view_menu(int menuoption)
{
        switch (menuoption)
        {
        case 11 :
                        break;
        case 12 :
                        break;
        }
}

GLvoid func_view_internal_menu(int menuoption)
{
        switch (menuoption)
        {
        case 111 :
                        break;
        case 112 :
                        break;
        }
}


GLvoid func_view_internal_viewdir_menu(int menuoption)
{
        GLfloat tempvar1,tempvar2,tempvar3=0;

        switch (menuoption)
        {
        case 1111 : if (viewdirection=='F')
                            {
                                    printf("View direction is front\n");
                            }
                            else
                            {
                                    tempvar1=aim_x;
                                    tempvar2=aim_y;
                                    tempvar3=aim_z;
                                    aim_x=view_x_cur;
                                    aim_y=view_y_cur;
                                    aim_z=view_z_cur;
                                    view_x_cur=tempvar1;
                                    view_y_cur=tempvar2;
                                    view_z_cur=tempvar3;
                                    viewdirection='F';
                                    glutPostRedisplay();
                            }

                            break;

        case 1112 : if (viewdirection=='B')
                            {
                                    printf("View direction is Back\n");
                            }
                            else
                            {
                                    tempvar1=aim_x;
                                    tempvar2=aim_y;
                                    tempvar3=aim_z;
                                    aim_x=view_x_cur;
                                    aim_y=view_y_cur;
                                    aim_z=view_z_cur;
                                    view_x_cur=tempvar1;
                                    view_y_cur=tempvar2;
                                    view_z_cur=tempvar3;
                                    viewdirection='B';
                                    glutPostRedisplay();
                            }
                        break;
        }
}


GLvoid func_view_internal_startpos_menu(int menuoption)
{
        int datno;

        switch (menuoption)
        {
        case 1121 : datno=1;
```

```
                              startposition(datno);
                    cee='i';
                    upz=1;
                    break;

          case 1122 : datno=rectotal-1;
                    startposition(datno);
                    upz=1;
                    break;
          }
}

GLvoid func_view_external_menu(int menuoption)
{
          GLfloat st1;

          aim_x=midx;
          aim_y=midy;
          aim_z=midz;
          view_y_cur=midy;

          switch (menuoption)
          {
                    case 121 :          view_x_cur=midx;
                                        view_z_cur=minz-(((maxy-miny)/2)+2);
                                        upz=0;
                                        glutPostRedisplay();
                                        break;

                    case 122 :          view_x_cur=midx;
                                        view_z_cur=maxz+(((maxy-miny)/2)+2);
                                        upz=0;
                                        glutPostRedisplay();
                              break;

                    case 123 :          st1=midy-maxy;
                                        if (st1<0)
                                                  st1=-1*st1;
                                        st1=st1+(0.5*st1);
                                        view_x_cur=midx+st1;
                                        view_z_cur=midz;
                                        upz=0;
                                        glutPostRedisplay();
                                        break;

                    case 124 :          st1=midy-maxy;
                                        if (st1>0)
                                                  st1=-1*st1;
                                        st1=st1+(0.5*st1);
                                        view_x_cur=midx+st1;
                                        view_z_cur=midz;
                                        upz=0;
                                        cee='o';
                                        glutPostRedisplay();
                                        break;
```

```
        }
}


GLvoid func_zoom_menu(int menuoption)
{
        switch (menuoption)
        {
        case 21 : zoostat='y';
                        zoovar=1;
                        glutPostRedisplay();
                        break;
        case 22 : zoostat='y';
                        zoovar=0;
                        glutPostRedisplay();
                        break;
        }
}


void render_scene()
{
  glRotatef(x,1,0,0);
  glRotatef(y,0,1,0);
  glRotatef(z,0,0,1);
  glCallList(bend1);  // calling the list to redraw the model
}


GLvoid initGL1()
{
  glClearColor(RED,0,0,ALPHA);
  glClearDepth(1.0f);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_SMOOTH);
  glEnable(GL_COLOR_MATERIAL);
}

GLvoid window_reshape1(GLsizei width, GLsizei height)
{
  if (height==0) height=1;
  glViewport(0,0,width,height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(90,(GLdouble)width/(GLdouble)height,0.1,200);
  rs1();
  glMatrixMode(GL_MODELVIEW);
}

GLvoid window_display1()
{
        GLfloat s;
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        s=midy-maxy;
        if (s>0)
                s=-1*s;
        s=s+(0.5*s);
        gluLookAt(midx+s,midy,midz,midx,midy,midz,0,1,0);
        rs1();
        glutSwapBuffers();
}

GLvoid window_idle1()
{
        glutPostRedisplay();
}

GLvoid rs1()
{
        glCallList(lis1);
        ni(view_x_cur,view_y_cur,view_z_cur);
}

GLvoid ni(GLfloat b1,GLfloat b2,GLfloat b3)
{
        glColor3f(1,1,0);
        glBegin(GL_QUADS);
                glVertex3f(b1,b2+0.8,b3-0.8);
                glVertex3f(b1,b2+0.8,b3+0.8);
                glVertex3f(b1,b2-0.8,b3+0.8);
                glVertex3f(b1,b2-0.8,b3-0.8);
        glEnd();
}



GLvoid sc1()
{
        lis1=glGenLists(1);
        GLfloat a1,a2,a3,a4,a5,a6,a7,a8,
                a11,a12,a13,a14,a15,a16,a17,a18;
        FILE *of;
        GLint a=0;
        if ((of=fopen("datatest.txt","r"))==NULL)
        {
                printf("Cannot open dat file for line");
                exit(1);
        }

        glNewList(lis1, GL_COMPILE);

        while (!feof(of))
        {
                if (a==0)       fscanf(of,"%f %f %f %f %f %f %f %f", &a1, &a2, &a3,
                                        &a4, &a5, &a6, &a7, &a8);
                fscanf(of,"%f %f %f %f %f %f %f %f", &a11, &a12, &a13, &a14, &a15,
                        &a16, &a17, &a18);
                a++;
```

```
            glColor3f(1,1,1);
            glBegin(GL_LINES);
                glVertex3f(a1,a2,a3);
                glVertex3f(a11,a12,a13);
            glEnd();
            a1=a11; a2=a12; a3=a13;
        }
        glEndList();
        fclose(of);
}


GLvoid startposition(GLint rno)
{
        GLfloat coordi[3],Rc,Gc,Bc,dist,ang1;
        int datacntr;
        FILE *fp0;
        if ((fp0=fopen("datatest.txt","r"))==NULL)
        {
            printf("Cannot open file");
            exit(1);
        }
        if (viewdirection=='F')
        {
            recposnum=rno;
            for (datacntr=1;datacntr<=recposnum;datacntr++)
            {
                fscanf(fp0,"%f %f %f %f %f %f %f %f", &coordi[0], &coordi[1],
                    &coordi[2], &ang1, &dist, &Rc, &Gc, &Bc);
            }
            view_x_cur=coordi[0];
            view_y_cur=coordi[1];
            view_z_cur=coordi[2];
            recposnum=rno+1;
            for (datacntr=1;datacntr<=recposnum;datacntr++)
            {
                fscanf(fp0,"%f %f %f %f %f %f %f %f", &coordi[0], &coordi[1],
                    &coordi[2], &ang1, &dist, &Rc, &Gc, &Bc);
            }
            aim_x=coordi[0];
            aim_y=coordi[1];
            aim_z=coordi[2];
            glutPostRedisplay();
        }
        else if (viewdirection=='B')
        {
            recposnum=rno;
            for (datacntr=1;datacntr<=recposnum;datacntr++)
            {
                fscanf(fp0,"%f %f %f %f %f %f %f %f", &coordi[0], &coordi[1],
                    &coordi[2], &ang1, &dist, &Rc, &Gc, &Bc);
            }
            aim_x=coordi[0];
            aim_y=coordi[1];
            aim_z=coordi[2];
```

```
                recposnum=rno+1;
                for (datacntr=1;datacntr<=recposnum;datacntr++)
                {
                        fscanf(fp0,"%f %f %f %f %f %f %f", &coordi[0], &coordi[1],
                                &coordi[2], &ang1, &dist, &Rc, &Gc, &Bc);
                }
                view_x_cur=coordi[0];
                view_y_cur=coordi[1];
                view_z_cur=coordi[2];
                glutPostRedisplay();
        }
}
```