INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

SOFTWARE REENGINEERING

SYSTEM VISUALIZER

ZUTONG SUN

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

AUGUST 2000

Abstract

Software Reengineering: System Visualizer

Zutong Sun

With recent advent of hardware accelerators for three dimension graphics in appear, the question of how to support them has become a hot topic. OpenGL, designed as a streamlined, hardware-independent interface is the right choice for most graphics programmer who could enjoy real interactive 3D images. In this major report, the author applies the principles of software re-engineering to the System Visualizer and successfully replaces the display module implemented by PEX with OpenGL. The new System Visualizer still retains the original algorithm, data structures and the parser, but incorporates the lighting, texture mapping, alpha blending, antialiasing, animation and hidden surface removal into the display module which the original system missing. OpenGL enables the System Visualizer run on both NT and X platform and removes the limitation of platform dependence.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Dr. Peter Grogono for his guidance and valuable insight throughout this work. His encouragement and patience made this report possible.

I also thank the Computer Science staff for the help they offered over the years. Special thanks to Ms. Halina Monkiewicz for her time and help during my graduate studies.

This work is dedicated to my parents, my mother-in-law, my wife and my son with great love and gratefulness.

Table of Contents

List of Figures

1. Introduction

Today, new developed software is evolving fast and maintaining aging software systems that are constructed to run on a variety of hardware types is becoming a critical task for many organizations. Some of the aging software is programmed in obsolete languages, and suffers from the disorganization that results from prolonged maintenance. As software ages, the task of maintaining it becomes more complex and more expensive.

1.1. Definition

Software Reengineering is the right way to solve the above problems. Software reengineering is the process of understanding existing software and improving it, reconstituting it in a new form and the subsequent implementation of the new form. Software Reengineering can increase or enhance software system functionality, better maintainability, configurability, and reusability. The process involves recovering existing software artifacts and organizing them as a basis for future evolution of the software system.

1.2. Program understanding

This definition divides software reengineering into two sets of activities. The first set of activities are supporting program understanding, such as reverse engineering, so as to understanding the existing system. The second set of activities include full restructuring using new specification and knowledge of the old system obtained from reverse engineering.

Better understanding of a program could help to perform corrective maintenance, reengineer, and keep documentation up to date. In order to minimize the likelihood of errors introduced during the subsequent change process, the software engineer must understand

the system sufficiently well so that changes made to the source code have predictable consequences. The difficulty is to recover a legacy system after many years of operation. Usually, there are three actions that can be taken to understand a program: read documentation; read source codes; and run the program. The requirements and architectural design can be examined by carefully analysis of existing documentation, the source code is usually the primary source of information, but observing the dynamic behavior of an executing program is very useful to improve understanding by showing up program characteristics that cannot be achieved from reading the source code alone.

The strategies for program understanding can be divided into top-down and bottom-up interactive processes. Bottom-up strategy constructs mappings from the implementation domain to the problem domain; software components with similar or related attributes or properties are aggregated to form higher-level conceptual subsystems. Bottom-up strategy is particularly suited for redocumentation purposes and portfolio analyses. Top-down strategy constructs mappings starting from the application domain and moving towards the implementation domain. There may be several intermediate levels of representation, top-down strategy is suited for goal-directed program understanding (e.g., for a given maintenance task).

1.3. Restructuring

Software restructuring is transformation of the system from one representation to another at the same relative abstraction level [Arnold 1993]. In this process, the system's functionality and semantics is preserved. But, the system's internal design and implementation structure is modified so that the system is more efficient. One example is to

rewrite old ``spaghetti" code into structured forms. Software restructuring not only concerns the observable software structure, but is also concerned with people's perceptions of software structure. The idea is to modify software so that the software engineer can understand it and control it easily in the future. Software restructuring potentially reduces software complexity, increased interchangeability of people maintaining software and reduces the amount of time needed for maintenance programmers to become familiar with a system, making the system easier to document and easier to test. Some of the techniques most used are coding style standardization. These approaches modify code to make it easier to understand, often without altering control structure or data structure.

1.4. Reverse Engineering

Program understanding is essential for software reengineering and evolution, reverse engineering technology can significantly aid software understanding [Arnold 1993]. Reverse engineering is the process of understanding different unknown and hidden information about a software system. Because maintenance cannot be performed without a complete understanding of the subject system, reverse engineering plays a very significant role in software maintenance and is an indispensable part of software maintenance. Reverse engineering does not change the existing system. It is a process of examination, not a process of change or replication. The software development process follows from high-level abstraction to more detailed design and concrete implementation. A reverse engineer has to move backward and create an abstract representation of the implementation from the mass of concrete details. Current reverse engineering technology is typically based on program analysis methods such as parsing and data-flow analysis. The domain analysis approach has

been devised as an alternative methodology. Redocumentation and design recovery are most widely discussed among many sub-areas of reverse engineering.

Documentation is often the first place programmers turn to before modifying code. Documentation helps the programmer understand code, plan and perform testing. Unfortunately, documentation often goes out of date and is never referred to. Redocumentation focuses on the redevelopment or adaptation of existing representations of the system. The resulting forms of representation are usually considered alternate views (for example, dataflow, data structure, and control flow) for an intended audience.

Design recovery is the process of recreating design level documentations not only from source code, but also from a combination of code, existing design documentation, personal experience, and general knowledge about problem and application domains. Design recovery reproduces all of the information required for a person to fully understand existing software application.

Most approaches in reverse engineering are semi-automatic in nature. Tools extract information, but people have to guide the tools and decide what information to look for. Efforts are underway for developing more automated tools. But, Reverse engineering tools are badly needed to make the transition over paradigm shifts quicker, easier and cheaper. Since software is greatly influenced by human cognitive process and only human can understand the mental models and intuitions guiding a particular software development, provision should be there for human intervention to guide reverse engineering process.

There will always be old software to be re-modeled, re-designed and re-implemented, good reverse and reengineering tools will always be in demand.

2. Review of OpenGL

OpenGL ("GL" for "Graphics Library") is a powerful software interface to graphics hardware that allows graphics programmers to produce high-quality color images of 3D objects [Woo *et al.*, 1998]. This interfaces enable programmers to build geometric models and, view models interactively in 3D space, control color and lighting, manipulate pixels, and perform such tasks as alpha blending, antialiasing, creating atmospheric effects, and texture mapping. Since its introduction in 1992, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard and has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API). Application developers are assured consistent display results regardless of the platform implementation of the OpenGL environment.

The well-specified OpenGL standard has language bindings for C, C++, Fortran, Ada, and Java™. All licensed OpenGL implementations come from a single specification and language-binding document and are required to pass a set of conformance tests. Applications utilizing OpenGL functions are easily portable across a wide array of platforms for maximized programmer productivity and shorter time-to-market.

2.1. Basic OpenGL Operation

Figure 1 shows a schematic diagram of OpenGL. Commands enter OpenGL on the left and pass through a series of processing stages named the OpenGL rendering pipeline. Most commands may be accumulated in a display list for processing at a later time. Otherwise, commands are effectively sent through a processing pipeline.

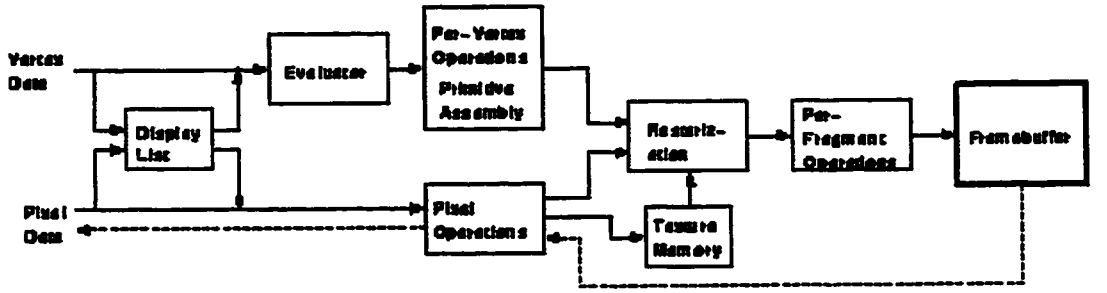

Figure 1. Block diagram of OpenGL

Geometric data (vertices, lines, and polygons) passes the evaluators and per-vertex operations, then undergo the rasterization and per-fragment operations, and finally is written into the framebuffer.

The first stage (Evaluators) provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values, since all geometric primitives are eventually described by vertices, parametric curves and surfaces may be initially represented by control points and polynomial functions. Evaluator provides a method to derive the vertices used to describe the surface from the control points.

The next stage converts the vertices into primitives. 4x4 floating-point matrices transform some vertex data, while spatial coordinates are projected for a position in the 3D world to a position on his screen. The primitives are clipped to a viewing volume in preparation for the next stage, rasterization.

The rasterizer produces a series of framebuffer addresses and fragments using a two-dimensional description of a point, line segment, or polygon. Each fragment square corresponds to a pixel in the framebuffer. A series of operations maybe performed on the fragments before values are stored into 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 colors with stored colors, as well as masking and other logical operations on fragment values.

Finally, Pixel data (pixels, images, and bitmaps) bypass the vertex processing portion of the pipeline to send a block of fragments directly through rasterization to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer.

2.2. Characteristic

- Stable. OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes.

Backward compatibility requirements ensure that existing applications do not become obsolete.

- Evolving. Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. In this way, innovations appear in the API in a timely fashion, letting application developers and hardware vendors incorporate new features into their normal product release cycles.

- Scalable. Although the OpenGL specification defines a particular graphics-processing pipeline, platform vendors have the freedom to tailor a particular OpenGL implementation to meet unique system cost and performance objectives. Individual calls can be executed on dedicated hardware, run as software routines on the standard system CPU, or implemented as a combination of both dedicated hardware and software routines. This implementation flexibility means that OpenGL hardware acceleration can range from simple rendering to full geometry and is widely available on everything from low-cost PCs to high-end workstations and supercomputers. Application developers are assured consistent display results regardless of the platform implementation of the OpenGL environment.

- Easy to use. OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer

lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

- OpenGL routines simplify the development of graphics software—from rendering a simple geometric point, line, or filled polygon to the creation of the most complex lighted and texture-mapped NURBS curved surface. OpenGL gives software developers access to geometric and image primitives, display lists, modeling transformations, lighting and texturing, antialiasing, blending, and many other features.

- Well-documented. Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain.

- Client/Server. OpenGL functions in a client/server environment. That is, the application program producing the graphics may run on a machine other than the one on which the graphics are displayed. The server part of OpenGL can access whatever physical graphics device or frame buffer is available on the machine where the graphics are displayed. This makes OpenGL a software interface to the actual hardware.

2.3. Detailed Description of OpenGL

OpenGL functions (which are called commands) are designed to provide 2D and 3D graphics with the emphasis on 3D. The program is fully functional, including everything that users usually want from 3D graphics. This includes:

- 3D modeling
- transformations
- color
- lighting
- Gouraud shading
- texture mapping
- non-uniform rational B-spline (NURBS) curves and surfaces
- atmospheric fog
- alpha blending
- motion blur

OpenGL creates images from models that are constructed from geometric primitives-points, lines, and polygons that are specified by their vertices and provides direct control over the fundamental operations of 3D and 2D graphics. This includes transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. However, OpenGL does not provide mechanisms to describe the complex models themselves but instead provides mechanism to describe how complex geometric

objects are to be rendered. This means that you can tell the program to draw points, lines, and polygons, and you have to build more complex models upon these. However, there are no special-purpose functions that you can call to create graphs, contour plots, and maps. That is why OpenGL could offer "real" 3D graphics, shading, lighting, texture mapping which other graphics libraries or packages lack.

Geometric primitives 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, colors, normals, and texture coordinates) are associated with a vertex and each vertex is processed independently. There is only one exception that to this rule, 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.

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). In OpenGL, modes are set, primitives are specified, and other OpenGL operations are represented by sending commands in the form of function or procedure calls.

OpenGL also supports advanced rendering features in either *immediate* mode or *display list* mode. With OpenGL, any commands that you issue are executed immediately.

That is, when you tell the program to draw something, it does it right away. You also have the option of putting commands into display lists, which is a non-editable list of OpenGL commands stored for later execution. But you can execute the same display list more than once. For instance, you can use display lists to redraw the graphics whenever the user resizes the window. You can also use a display list to draw the same shape more than once if it repeats as an element of the picture.

2.4. Introduction to the software libraries

Mesa is a 3-D graphics library based on OpenGL. It was written by Brian Paul of the University of Wisconsin at Madison, and it is extremely similar to OpenGL. One major difference between Mesa and OpenGL is that Mesa uses regular X Windows graphics on Unix machines and is therefore not a software interface to any specific hardware graphics devices. Mesa works on most Unix platforms that have X11, including recently added Linux. There are drivers for Microsoft Windows, Mac OS.

3. Review of Kim Thang Vu's Thesis

Today, software engineering consists of constructing a large software application [Sommerville, 1995]. Such a large application requires more people and time to involve and usually is teamwork. Subsequent maintainance and enhancement are also critical. In order to better maintain big software applications that often consists of hundreds of modules, the demand for software tools grows faster. The traditional tools mainly use computer graphics and animation to help illustrate and present computer programs, processes, and algorithms. The limitation is the space of a window, since there is only limited room for a certain number of nodes on the screen. In order to solve some of the limitation associated with two-dimensional display, Kim Thang Vu developed a 3D software visualization tool called System Visualizer [Vu, 1997]. In Vu's thesis, a 3D image can be viewed from different directions. Consequently, a 3D image that may look confusing from one direction may look very clear from another direction. Larger objects that are farther from our eyes appear smaller in the computer screen. Therefore, a 3D image is more comprehensible than 2D image.

3.1. Requirements of System Visualizer

The goal of System Visualizer tool is to be able to display a set of C or C++ modules in a large software system and their uses relationships as three-dimensional structure. This will help a software engineer understand the large software system quickly by looking at the system visually. In Vu's thesis, System Visualizer displays a module as a sphere and a use relationship between two modules as a line between two spheres. Two questions are addressed: The first question is to determine the relationships between all modules of a

system. This is done by assuming a typical software system that is developed by use of C programming language, the software system must have a main module which is a .c file, the .c file contains the main () function and this main module normally has an associated .h header file. In fact, every module of a software system has an associated header file that is included in the .c file for the module. When a module uses another module of the software system, it includes the header file of that module in its header file. Thus, the uses relationships between each module of the software system can be inferred by starting from the main module. The second question addressed in Vu's thesis is how to graphically display theses modules and their relationships as spheres and lines in three-dimensional space. Kim solved this problem by looking at the treelike structure of the relationships between modules of a system, for example, if module A uses modules B and C. Modules B in turn uses modules D, E and F, then module A will be the root of the tree which has two children B and C where C is a leaf of the tree. B in turn has three children D, E and F which all are leaves. Thus, modules and their relationships can be displayed as tree, each module is represented as a sphere and the relationship between two modules is an edge between the spheres. It is noted that in three-dimension space, the children of a module M are put on the surface of a cone with its vertex at M. Therefore the term *cone tree* is used in the Vu's thesis to refer to this kind of tree.

In Vu's thesis, the System Visualizer displays a cone tree of at most 3 levels to avoid very deep cone tree in a limited computer screen. The tool allows the user to select any non-leaf modules and display the cone tree starting from the selected module. The tool also provides different colours for leaf nodes and non-leafs nodes in order to distinguish these two different nodes for the user. Moreover, the tool provides the user with capability to view

the cone tree from six different positions, so that overlapped the spheres in the computer screen can be distinguished.

3.2. Design of System Visualizer

In the design phase of System Visualizer, two major components have been built up. The first component is the *parser* that reads the header files of the software system under study. Then the parser infers module relationships. The second component is the *display module* that uses the information obtained by the parser to build modules and their relationships into three-dimensional image on the computer screen. The module relationships extracted have to be stored into somewhere, so that the display module can retrieved them for drawing. Thus, Vu proposed to use a collection of data structures in the thesis (see figure 2). The data structure is a flat linked list. Since a module may use a list of other modules, the data structure has the module name and a pointer to a list of other modules. It is very easy and efficient to traverse a flat linked list.

By being given a name of the main module M, the parser will first build up a subtree node which module M is the root of the subtree. After extracting module M uses information stored in the data structure, the parser creates a linked list of edges and each edge points to a leaf node that represents a module used by the root. The parser uses a recursive technique to construct the cone tree as follows: For each non-leaf nodes, the parser will build up a subtree which has the non-leaf node as the root and the modules it uses as the leaves. Append the above-created subtree to the linked list of subtrees. If a leaf node is the root of another subtree, then make this leaf node points to that subtree in the linked list of subtrees. Repeat this process for each non-leaf node used by the root of this subtree.

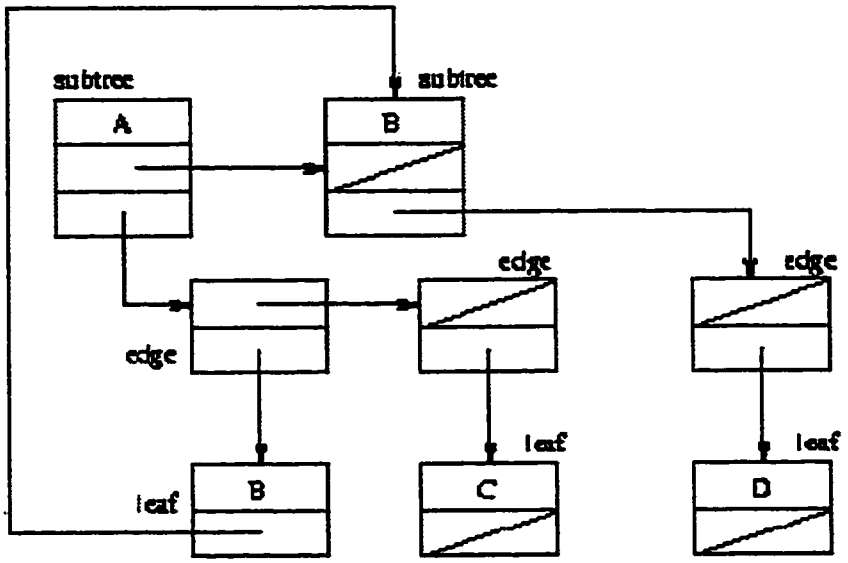

Figure 2. Flat linked list data structure

In order to illustrate the detail of the display module design principal, Kim use an example to explain how to draw a subtree. Considered a subtree with root A module, and module B and C as leaves, the tree can be drawn as figure 3. The sphere C in XYZ coordinate is computed using following formula:

$$x = r \sin\theta \sin\phi$$

$$y = r \cos\phi$$

$$z = r \sin\phi \cos\theta$$

Here r is the distance from the center of sphere C to the origin. ϕ is a fixed angle, which is greater than 90 degrees and θ varies for spheres B and C. In order to draw the module B and C, Vu first draw B and C in the origin, then Vu computes r, θ and ϕ for module B and C, finally move module B and C from the origin to their new XYZ coordinate using the above formula. Vu uses a fixed angle to compute θ for each leaf, here is formula provided to compute θ for a leaf i: which i is from 0 to N – 1.

$$\theta_i = \text{constant angle} + (i \times 360) / N$$

The last task is to draw the line to connect two spheres. Figure 4 demonstrates the drawing process. First the length L of line was drawn along the Y axis starting at the origin, then the line was moved up a distance R where r is the radius of the sphere. Finally the line is rotated down through an ϕ angle and right through an angle θ.

The display module first extracts the information stored into the data structure, starting from the current node as the root, it computes the locations of each sphere in the subtree using the above method. Thus, all the sphere and lines connecting them to build the subtree can be drawn. Finally the built up subtree is moved to its proper location in XYZ coordinates. The process is repeated until there are no non-leaf nodes or maximum number of levels is reached.

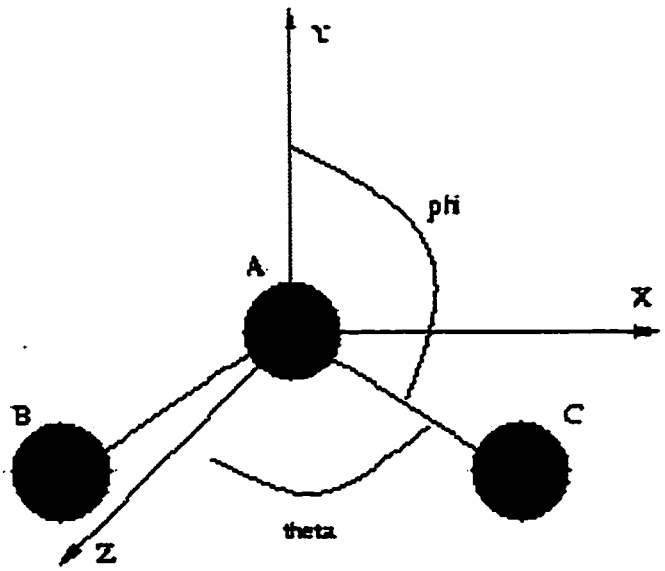

Figure 3. Visualization of a simple tree

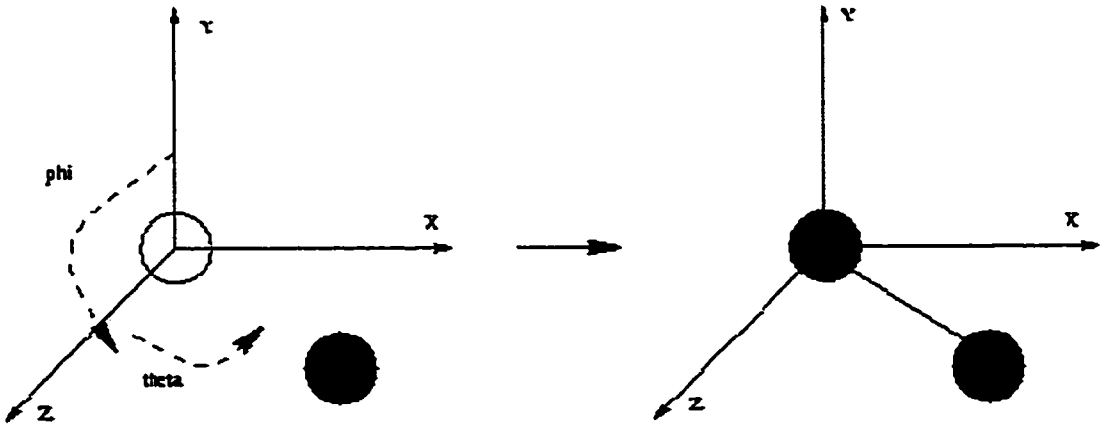

Figure 4. Computing the line between two spheres

In Vu's thesis, different types of modules are painted different colors, the leaf node that its pointer is nil in the data structure is painted blue. The non-leaf node which is appears once is painted yellow, otherwise is painted red. The same subtree can be drawn at the smallest root position once only.

3.3. Implementation of System Visualizer

Corresponding to the design phase of System Visualizer, there are three parts to be implemented for System Visualizer, namely the data structures, the parser and the display module.

3.3.1. Data Structures

There are three major data structures for System Visualizer: subtree, edge and leaf. The root node of a subtree is represented by the subtree data structure that contains the fields: NodeName, nextSubtree, Uses, AppearCount, SmallestRootedLevel, Traversed, and Drawn. Here, the field AppearCount is used to store the count of the number of times that the non-leaf module is to appear in the cone tree; the field SmallestRootedLevel is used to store the information to draw the instance of the subtree which has root at the smallest level. The field drawn is used to mark a subtree as having been drawn once it is drawn at the first time, so that subsequent occurrences of the subtree at this level will not be drawn again. The subtree also contains a pointer Nextsubtree to the next subtree in the subtree linked list and a pointer Uses to a linked list of edges.

The uses relationship between modules of a subtree is represented by the linked list of edges. Two pointers exist in the edges linked list, *nextedge* points the next element of the linked list and leaf pointing to a leaf node of the subtree.

The leaf data structure contains the Nodename that is the name of the module used by the root of the subtree and a pointer sub_tree to the subtree node where the module itself is the root of this subtree.

3.3.2. The parser

The main function of the parser is build_tree (). This function takes the name of the main module A of the application software and calls the function intreelist () to check if the subtree rooted at module A has been built. If the subtree is built up, the build_tree () function just returns. Otherwise, the next step is to call the function buildroot_tree_ds () to construct the subtree rooted at module A, then call the function addsubtree () to append the subtree of A just created to the linked list of subtrees. For each edge E in the linked list of edges pointed to by pointer *Uses* of the subtree of A, the function build_tree () is invoked with passing in module name B, the information stored in the leaf node pointed to by *Leaf* of edge E is extracted and the subtree at module B is built, the pointer sub_tree of the leaf node B is set to point to the subtree rooted at module B.

The implementation of function build_tree () is a little complicated and described as following: first the function takes the name of a module, then open the header file H of the module for reading, while reading every line in the header file H, the function parses every line to get a header file. If the module is a non-leaf module, the function allocates a subtree

node for it, and it also allocates an edge and corresponding Leaf node for every header file included in H.

3.3.3. The display module

The System Visualizer uses PEX that is a three-dimension extension of the X window system to display structure. There are two modes for drawing in PEX, one mode is immediate mode in which the information how to draw picture is not stored somewhere. Every time the picture needs to be redrawn, the information should be re-created. The other mode of drawn is to use PEX Structures. The information to how to draw the picture can stored in the PEX structure, later on, the application can ask PEX to draw the pictures using the information specified in the PEX structure. The display module uses this mode to drawing the cone tree, since the Systems Visualizer needs to redraw the picture many times depending on users input. The following tasks is performed:

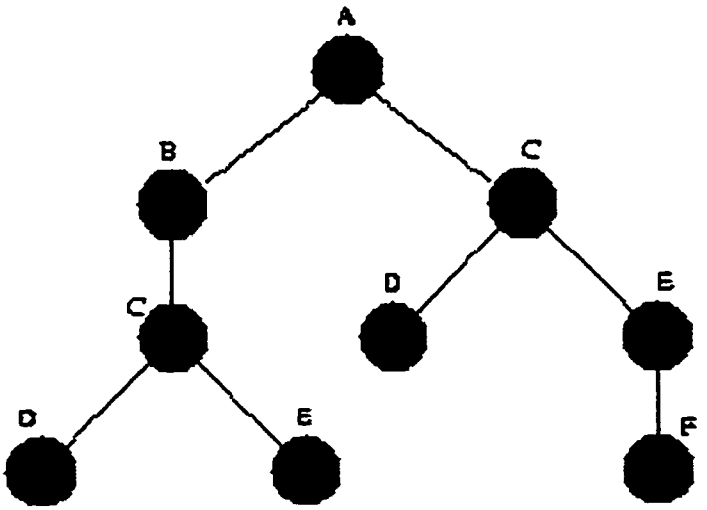

Figure 5. An example of a tree structure with duplicated subtrees

PEX function PEXCreateStructure is invoked to create a PEX structure, and PEXSetViewIndex is invoked to tell PEX to use the view. Function ComputeSmallestRootedLevel is called to compute SmallestRootedLevel for all subtrees to be displayed in the cone tree and function ComputeAppearCount is called to compute AppearCount for all subtrees to be displayed in the cone tree, because the values for both SmallestRootedLevel and AppearCount are computed first to decide how to draw the whole cone tree. According the requirements, the level of subtree can be drawn is limited to MaxLevelToDraw and the root level node can contains the same leaf node as shown in figure 5 C node. The subtree at root node C can only be drawn once at the smallest root level as shown figure 6.

The function BuildSubConeStructure is called recursively to build all subcones of the cone tree to be drawn and store this information into the PEX structure, starting at CurrentLevel that should be 0 and at the first subcone pointed by CurrentConeRoot. After the subcone is built, it should be moved to its CorrectSubconeLocation as (x, y, z) in the XYZ coordinate.

In summary, the System Visualizer achieves some outstanding characteristic of three-dimensional display with the implementation of PEXlib. This includes the ability of rotate a 3D image, as a result, a very crowded cone tree with many nodes can be rotated until the clear picture appear. Also, objects that are closer to our eyes appear larger and objects that are farther from eyes appear smaller with the projection property of PEXlib. Another very useful characteristic is the overlapped portion in three-dimensional space is hidden with the manual control. The result of these is that a much more comprehensive cone tree shows up.

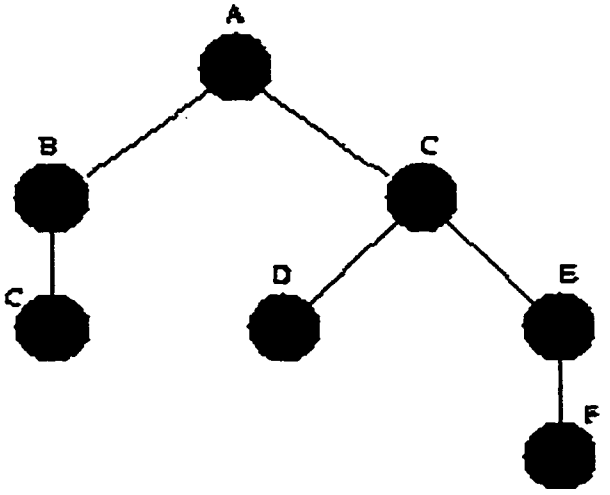

Figure 6. The visualization of the tree structure with duplicated subtrees

4. System Visualizer Design and Implementation By OpenGL

4.1. The difference between PEX and OpenGL

PEX is very tightly coupled to the X Window System, but X is not the only significant window system on the market. For this reason, OpenGL was designed to be window system independent, in order to have a consistent model for three-dimension across the two window systems, we decide to use OpenGL programming language to replace PEX to make the System Visualizer platform independent. From the above review of Vu's thesis, we know that original System Visualizer contains three components, data structures, the *parser* and the *display module* that is implemented using PEX. With OpenGL, we do not need to change the data structures and the parser, the only component we need to replace is the display module, thus, our new system mainly rewrites the display module with OpenGL programming language and integrates with the parser.

PEX and OpenGL both provide a means to store commands for later execution. In PEX, editable structures can be created. Structures may contain calls to execute other structures allowing them to be arranged in a hierarchical fashion. Entire three-dimension models can be constructed out of a hierarchy of structures so that a redraw requires only retraversing the structure hierarchy, as review of Vu's thesis, System Visualizer is implemented in PEX structure.

OpenGL does not support structures in the same way PEX does. However, *display lists* can be constructed which contains sequences of OpenGL commands. Unlike PEX

structures, OpenGL display lists are not editable. Once a display list is created, it is sealed and cannot be changed. The commands in the display list can be optimized for faster execution. Even though display lists cannot be edited, the same effect as editing can be achieved by rewriting display lists called by other display lists.

Display lists and structures both minimize the amount of transfer overhead when running PEX or OpenGL over a network since the commands in a structure or display list can be executed repeatedly by only calling the display list by name. The commands themselves need to be transferred across the network only once. In our final design of this new system, we decide to use the display list to draw the cone tree.

PEX and OpenGL both support basic three-dimension rendering functionality. Both allow three-dimension and two-dimension lines and polygons to be rendered using standard modeling and viewing methods. PEX and OpenGL also supports picking, lighting, depth cueing, and hidden line and surface removal. There are a number of sophisticated rendering features supported by OpenGL but PEX completely lacks, such as alpha blending, texture and environment mapping, antialiasing, accumulation buffer methods, and stencil buffering.

PEX does support features not available in OpenGL. For example, PEX has extensive text support for stroke fonts that are fully transformable in three-dimension.

Double buffering and stereo support are built into OpenGL while PEX relies on proprietary support or not yet nonstandardized X extensions for double buffering and stereo.

4.2. Overall Design by OpenGL

The overall design of our new system is the same as original System Visualizer. First, the parser is built using the function build_tree () with root node name as parameter. The display modules retrieves module relationships information stored in The Parser and use this information to construct and draw the corresponding cone tree. The application draw the root node by calling the function build_a_subtree () which subsequently calls function build_cone_scheme () to build the XYZ coordinate, and finally calls the function build_sub_tree () to recursively build all subcones of the cone tree to be drawn and move them to the appropriate locations in the three-dimension space.

4.3. User Interface

To start up System Visualizer, the executable should be run with the root module name as the first parameter and the directory where all the header files are located as the second parameter. The System Visualizer first parses the header files, then infers module relationships and displays a graphic cone tree that represents each module and their relationships of a software application in a drawing window. Figure 7 is an image displayed by the System Visualizer. By clicking the right button of mouse, four selections pop up: Texture, Start Motion, Stop Motion, and Quit. Select Texture, the background of drawing window will change to another color background; Select the Start Motion, the image will turn around clockwise; Select Stop Motion, the motion stops; Quit is for quitting program. There is another four-keyboard arrow for use, by pressing the ↑ arrow, the image go towards

the window by use of the user as reference point and the image becomes smaller. By pressing ↓ arrow, the image becomes bigger and goes towards user. By pressing ← arrow, the image moves counter-clockwise. By pressing the → arrow, the image moves clockwise, so that the crowded the spheres can be shown up.

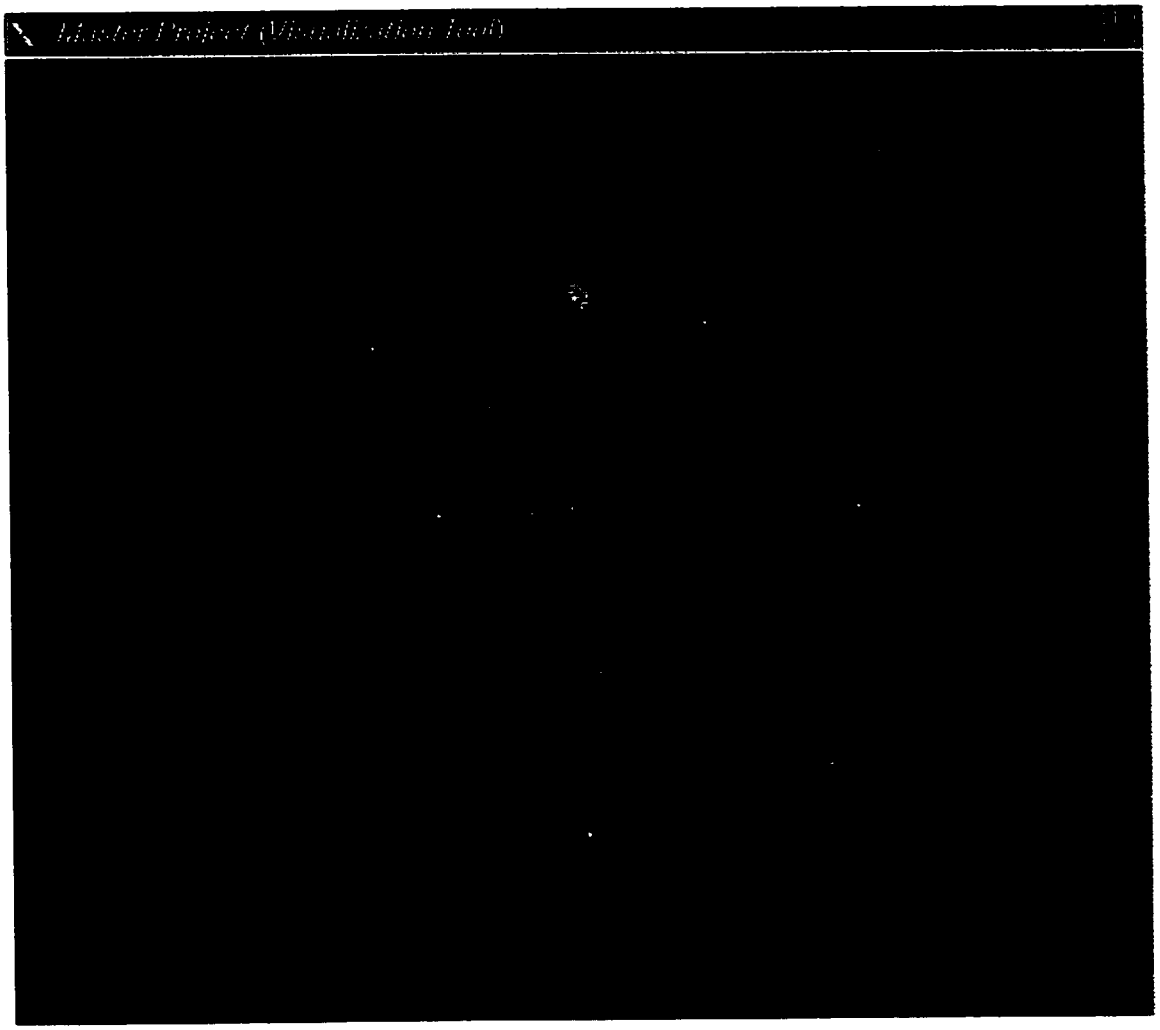

Figure 7. An illustration of the user interface

4.4. Detailed Design, Implement and Integration Process

Since OpenGL is independent of any operating system or window system, it does not contain any commands for opening windows or reading events from the keyboard or mouse. Unfortunately, it is impossible to write a complete graphics program without at least opening a window, and so the GLUT utility toolkit is used to opening windows and detecting input events. GLUT was written Mark Kilgard and includes windowing functions contains multiple windows for OpenGL rendering a simple cascading pop-up menu facility bitmap and stroke fonts event processing. In the beginning of program entry point of main function, the function build_tree () constructs the Parser. The maximum level that is 3 is set up to be displayed on the computer screen. The following four routines is used to perform window management:

```
glutInitWindowSize (int width, int size)
glutInit (&argc, argv)
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH)
glutCreateWindow (name)
```

OpenGL is a state machine, various states or modes should be set up and remain in effect until you change them. The current display mode is a state variable. You can set the current display mode to RGBA color mode which the hardware sets aside a certain number of bitplanes for each of the R, G, B, and A components, then every object is drawn with RGBA mode until you set the current Display Mode to color-index display mode.

We want a window with depth buffer, double buffering and RGBA color mode, so glutInitDisplayMode (GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH) is called, In the following paragraph, we explain the two important OpenGL characteristic of double-buffering and hidden-surface removal.

OpenGL implementations provide double buffering to realize the graphics animation. Double buffering provides two complete color buffers for use in drawing. The frame in one buffer is displayed while the frame in the other buffer is being drawn. When the drawing of a frame is complete the two buffers are swapped so that the one that was being viewed is now used for drawing. One of another important characteristic in three-dimensional visualization is hidden-surface removal, when two objects overlapped parts, the object that are closer to our viewing position should be drawn, the other object behind is obscured and should be eliminated. Hidden-surface removal means the elimination of parts of solid objects that are obscured by others. The easiest way to achieve this is to use the depth buffer. Since the PEX server does not support hidden-surface removal in Vu's thesis, Vu had to do extra work manually to make sure the order in which objects are drawn will not obscured by another object. With OpenGL, we just set up the Display Mode with depth buffer.

The following program executed functions init (). Within the function init (), program draws the cone tree structure by calling function build_a_subtree (). The display list tree is created here to make drawn fast later on. Within the build_a_subtree () function, function init_called_levels () is first called to initialize the called_count to 0, smallest_called_level to 911, sub_tree_drawn 0. Subsequently function compute_called_level

() is called and the value of called_count in the current tree is computed. Both smallest_called_value and call_count should be computed prior to drawn, so that we can decide how to draw the cone tree. This is already discussed in the review of the Vu's thesis. The following paragraph explains how the function works.

The function compute_called_level () takes a pointer to subtree A in which recursively traverses subtree A and all subtrees under subtree A to compute smallest_called_value for each subtree. Starting at CurrentLevel at 0, and stopping at MaxLevelToDraw, if smallest_called_value of A is greater than CurrentLevel (Originally, when subtree A is allocated, smallest_called_level is set to 911), then set it to be CurrentLevel; If CurrentLevel is less than MaxLevelToDraw, then for each non-leaf module X used by A, increases CurrentLevel by 1, recursively call function compute_called_level (), passing the pointer to subtree X. The value of called_count is computed after the value of smallest_called_value is computed, because the algorithm to compute the value of called_count uses the value of smallest_called_value to decide if it should traverse down a subtree.

If the value of called_count in the current tree is greater than 1, we draw a red sphere with GLUT function glutSolidSphere (RADIUS, 50, 50), otherwise we draw a yellow sphere and recursively invoke function build_cone_scheme () and build_sub_tree () until reach the leaf node. The function build_cone_scheme () builds r, ϕ, and θ values for all the nodes within the same level of subtree and put these values into the structure. Subsequent function build_sub_tree () is called to rotate and draw the line, then retrieve the values of r, ϕ, and θ to compute the XYZ coordinate in the three-dimensional space for drawing the sphere.

Finally, OpenGL implementation use the callback mechanism to register the events, the following five functions is registered the callback functions:

```
GlutDisplayFunc (display);
GlutReshapeFunc (reshape);
GlutSpecialFunc (Specialkey);
GlutKeyboardFunc (keyboard);
GlutVisibilityFunc (visible);
```

The callback registered in glutDisplayFunc () is executed whenever the contents of the window need to be redisplayed, thus, we registers display () in glutDisplayFunc () for re-render the scene which is our display list tree. On the other side, we use init () to draw cone tree once during the program initialization.

The very last thing is call glutMainLoop (void) that triggered the event processing, rendering the windows and the registered display callback. Once the program enters the loop, it is never exited until program exits.

4.4. Extra Features

During implementation of the display module by OpenGL, we renders a lit sphere which the original System Visualizer could not do. In order to add lighting to the scene, normal vectors that determine the orientation of the objects relative to the light sources is defined for each vertex of all the objects. The normal for the sphere is defined as part of the glutSolidSphere (). Next, we use glLightfv (GL_LIGHT0, GL_DIFFUSE, white) and glLightfv (GL_LIGHT0, GL_SPECULAR, white) to define light sources. Finally, lighting mode is described, this will decide viewer of the scene is an infinite distance away or local to the scene, whether lighting calculations is performed differently for the front and back surfaces of objects in the scene. In our new system, the default settings--glShadeModel (GL_SMOOTH) for lighting mode was chosen, which is an infinite viewer and one-sided lighting. Because we never see the back of surface of the sphere, one-sided lighting is sufficient. The last thing, we use functions glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, red), glMaterialfv (GL_FRONT, GL_SPECULAR, white), glMaterialfv (GL_FRONT, GL_SHINESS, polished) to define material properties for the objects in the scene. An object's material properties determine how it reflects light, this will give viewer the effect what material it seems to be made of.

In the new system, we have the start motion choice that implemented by gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz). glLookAt is another very useful utility. This utility allow viewer to pan across a landscape, what it does is that

glLookAt defines a viewing matrix derived from an eye point, a reference point indicating the centre of the scene, and an up vector. The matrix maps the reference point to the negative z-axis and the eye point to the origin, so that, when a typical projection matrix is used, the centre of the scene maps to the centre of the viewport. Similarly, the direction described by the up vector projected onto the viewing plane is mapped to the positive y-axis so that it points upward in the viewport.

We also provide the texture mapping technique to the background of the new system. By clicking the right button, Texture option can be selected. The screen background is changed to another colour pattern. The principle of texture mapping is that we can decrease the model complexity and then the computation cost decrease by scanning a photo of the detail and paste it on objects. The first step we do is create texture objects by reading from a file, next we specify how the texture is to be wrapped and how the colours are to be filtered, final step we enable the texture mapping and supply both texture and geometric coordinates for drawing the scene.

In order to have the parts of scene appear translucent, we specify a blending function during the drawing process. We also want to reduce the aliasing of the lines with the antialiasing techniques in the new system. Both blending and antiliasing techniques are simple to use in OpenGL. When blending is enabled, color values of the fragment being processed (source) are combined with the color values of the corresponding currently stored pixel (destination). The function glBlendFunc (source factor, destination factor) specify how the source and destination factors are computed and then the corresponding components are added to give the final, blended RGBA values between 0 and 1. To have antialiasing the

lines, glEnable () must be turned on passing in GL_LINE_SMOOTH. In RGBA mode that is currently applied in our new system, the blending should be first enabled, then with glHint (target, hint) command passing in GL_LINE_SMOOTH_HINT and GL_DONT_CARE. We can control the image quality and speed by specifying hint value, since the more quality of image, the more time OpenGL need to compute. In the current system, we choose GL_DONT_CARE to indicate no preference to both speed and image quality.

5. Conclusion

5.1. Summary

The new System Visualizer successfully removes the limitation of platform from original System Visualizer by use of OpenGL. The new System Visualizer consists of the same three major components as the original system has data structures, the parser and the display module. The new System Visualizer uses the original algorithm to extract module structure of a software application developed in C or C++ and to display system structure in three-dimensional space as the cone tree. Although the System Visualizer only displays a cone tree of maximum three levels, any number of levels cone tree can be displayed by use of the original algorithm. The new System Visualizer not only has all the features of original one, such as hidden surface removal and ability to rotate a 3D image, but also there are new features in the new system which the original system complete missing. The new features include alpha blending, antialiasing, texture mapping, animation and rendering a lit sphere with directional light.

5.2 Future Work

From the user interface point of view, the user interface needs to be more user-friendly. The user can random select the interesting sphere as root note by clicking that sphere, then the deeper use relationship can be shown up as cone tree. From the algorithm point of view, it would be better if the tool can extract the class hierarchy structure

developed in C++, one of example is Rationale Rose which can build up the hierarchy relationship from a given C++ application, but that is not a graphic representation of a system structure. Concerning the limitation of the window screen, one possible way to show entire the system structure is to have multiple windows with scalable size and random positions in the window screen, so that the user can see any node structure at any time by just clicking that window.

References

1. Robert S. Arnold. Software Reengineering, IEEE Computer Society Press, 1993.

2. Mason Woo, Jackie Neider, and Tom Davis. OpenGL Programming Guide, Second Edition, Addison Wesley, 1998.

3. Kim Thang Vu. System Visualizer, Master thesis, Department if Computer Science, Concordia University, March 1997.

4. Ian Sommerville. Software Engineering, Fifth Edition, Addison Wesley, 1995.

5. http://www.opengl.org/About/About.html

6. http://www.sgi.com/software/opengl/glandx/intro/intro.html

7. http://www.opengl.org/Documentation/Papers.html

8. http://trant.sgi.com/opengl/docs/white_papers/oglGraphSys/opengl.html

APPENDIX – System Visualizer Source Code

A.1. Create_tree.h

```
#include <stdio.h>
#include <string.h>
#include <strings.h>

#define MAX_NAME_LEN 10
#define ARRAY_SIZE_FROM_LEAF_LEVEL 10

struct dupl_sub_tree {
  int       count;
  int       from_leaf_level[ARRAY_SIZE_FROM_LEAF_LEVEL];
};

struct tree_t {
  char node_name[MAX_NAME_LEN];
  int       called_count;
  int       smallest_called_level;
  int       sub_tree_drawn;
  int       from_leaf_level;
  struct tree_t *next_tree;
  struct edge_t *uses;
};

struct edge_t {
  struct edge_t *next_edge;
  struct node_t *node;
};

struct node_t {
  char node_name[MAX_NAME_LEN];
  struct tree_t *sub_tree;
};
```

A.2. Create_tree.c

```
#include "create_tree.h"

struct tree_t *the_tree;
extern char dir_path[80];
extern int current_level;

char *module_name(line)
char *line;
{
char *tmp, *start, *module;

  if (!(tmp = strstr(line, "#include"))) {
    return tmp;
  }
  start = tmp + 8;

  if (!(tmp = strchr(start,'"'))) {
    if (!(tmp = strchr(start,'<'))) {
      return tmp;
    }

    start = tmp + 1;
    if (!(tmp = strchr(start,'>'))) {
      return tmp;
    }

    *tmp = 0;
    while (tmp = strtok(start, "/")) {
      start += strlen(tmp) + 1;
      module = tmp;
    }
  }
  else {
    tmp++;
    module = tmp;
    if (!(tmp = strchr(module,'"'))) {
      return tmp;
    }
```

```
    *tmp = 0;
  }

  if (!(tmp = strstr(module, ".h"))) {
    return tmp;
  }
  else {
    *tmp = 0;
    return module;
  }
}

void init_called_levels()
{
struct tree_t *a_sub_tree;

  a_sub_tree = the_tree;
  while (a_sub_tree) {
    a_sub_tree->called_count = 0;
    a_sub_tree->sub_tree_drawn = 0;
    a_sub_tree->smallest_called_level = 911;
    a_sub_tree->from_leaf_level = 0;

    a_sub_tree = a_sub_tree->next_tree;
  }
}

void compute_called_levels(root, called_level, sub_tree_count)
struct tree_t *root;
int         called_level;
struct dupl_sub_tree sub_tree_count;
{
int i;
struct dupl_sub_tree t_sub_tree_count;
struct edge_t *uses;

  if (called_level <= current_level) {
    if (sub_tree_count.count == 0) {
      root->called_count++;
    }
    else {
      for (i = 0; i < sub_tree_count.count; i++) {
        sub_tree_count.from_leaf_level[i]--;
      }

      t_sub_tree_count.count = 0;
      for (i = 0; i < sub_tree_count.count; i++) {
        if (sub_tree_count.from_leaf_level[i] > 0) {
```

```
            t_sub_tree_count.from_leaf_level[t_sub_tree_count.count] =
                sub_tree_count.from_leaf_level[i];
            t_sub_tree_count.count++;
          }
        }

        sub_tree_count.count = t_sub_tree_count.count;
        for (i = 0; i < sub_tree_count.count; i++) {
          sub_tree_count.from_leaf_level[i] =
              t_sub_tree_count.from_leaf_level[i];
        }
      }

      if (root->called_count > 1) {
        if (root->from_leaf_level == 0) {
          root->from_leaf_level = current_level-root->smallest_called_level;
        }

        if (root->from_leaf_level > 0) {
          sub_tree_count.from_leaf_level[sub_tree_count.count] =
              root->from_leaf_level;
          sub_tree_count.count++;
        }
      }

      if (root->smallest_called_level > called_level) {
        root->smallest_called_level = called_level;
      }

      uses = root->uses;
      while (uses) {
        if (uses->node->sub_tree) {
          compute_called_levels(uses->node->sub_tree, called_level+1,
                        sub_tree_count);
        }
        uses = uses->next_edge;
      }
    }
}

int in_tree_list(node_name, node_ptr)
char *node_name;
struct node_t *node_ptr;
{
struct tree_t *a_sub_tree;

  a_sub_tree = the_tree;
  while (a_sub_tree) {
```

```
    if (!strcmp(a_sub_tree->node_name, node_name)) {
      if (node_ptr) node_ptr->sub_tree = a_sub_tree;
      return 1;
    }
    a_sub_tree = a_sub_tree->next_tree;
  }
  return 0;
}

struct tree_t *build_root_tree(node_name)
char *node_name;
{
struct tree_t *a_sub_tree;

  a_sub_tree = (struct tree_t *) malloc(sizeof(struct tree_t));
  strcpy(a_sub_tree->node_name, node_name);
  a_sub_tree->next_tree = 0;
  a_sub_tree->uses = 0;
  return a_sub_tree;
}

struct node_t *ds_build_a_node(node_name)
char *node_name;
{
struct node_t *a_node;

  a_node = (struct node_t *) malloc(sizeof(struct node_t));
  strcpy(a_node->node_name, node_name);
  a_node->sub_tree = 0;
  return a_node;
}

struct tree_t *build_sub_tree_ds(node_name)
char *node_name;
{
FILE *node_fp;
char a_line[132], full_path[132], *sub_node_name;
struct tree_t *a_sub_tree;
struct edge_t *an_edge, *uses=0, *tmp_uses;
struct node_t *a_node;

  sprintf(full_path, "%s/%s.h", dir_path, node_name);
  if (!(node_fp = fopen(full_path,"r"))) {
    printf("Could not fopen %s\n", full_path);
    return 0;
  }

  a_sub_tree = build_root_tree(node_name);
```

```
  while ((sub_node_name = fgets(a_line, 132, node_fp))) {
    if (!(sub_node_name = module_name(a_line))) continue;
    an_edge = (struct edge_t *) malloc(sizeof(struct edge_t));
    if (!(uses)) {
      tmp_uses = uses = an_edge;
    }
    else {
      tmp_uses->next_edge = an_edge;
      tmp_uses = tmp_uses->next_edge;
    }
    an_edge->next_edge = 0;
    a_node = ds_build_a_node(sub_node_name);
    an_edge->node = a_node;
  }

  if (!uses) {
    free(a_sub_tree);
    a_sub_tree = 0;
  }
  else {
    a_sub_tree->uses = uses;
  }

  fclose(node_fp);
  return a_sub_tree;
}

void add_sub_tree(a_sub_tree)
struct tree_t *a_sub_tree;
{
struct tree_t *next_tree;

  if (!the_tree) {
    the_tree = a_sub_tree;
    return;
  }

  next_tree = the_tree;
  while (next_tree->next_tree) {
    next_tree = next_tree->next_tree;
  }
  next_tree->next_tree = a_sub_tree;
  return;
}

void build_tree(node_name, node_ptr)
char *node_name;
struct node_t *node_ptr;
```

```
{
struct tree_t *a_sub_tree;
struct edge_t *an_edge;

  if (!in_tree_list(node_name, node_ptr)) {
    if ((a_sub_tree = build_sub_tree_ds(node_name))) {
      add_sub_tree(a_sub_tree);
      if (node_ptr) node_ptr->sub_tree = a_sub_tree;

      an_edge = a_sub_tree->uses;
      while (an_edge) {
        build_tree(an_edge->node->node_name, an_edge->node);
        an_edge = an_edge->next_edge;
      }
    }
    else {
      if (!the_tree) {
        the_tree = build_root_tree(node_name);
      }
    }
  }
}
```

A.3. texture.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
bwtorgba(unsigned char *b,unsigned char *l,int n) {
   while(n--) {
         l[0] = *b;
         l[1] = *b;
         l[2] = *b;
         l[3] = 0xff;
         l += 4; b++;
   }
}

void
latorgba(unsigned char *b, unsigned char *a,unsigned char *l,int n) {
   while(n--) {
         l[0] = *b;
         l[1] = *b;
         l[2] = *b;
         l[3] = *a;
         l += 4; b++; a++;
   }
}

void
rgbtorgba(unsigned char *r,unsigned char *g,unsigned char *b,unsigned char *l,int n) {
   while(n--) {
         l[0] = r[0];
         l[1] = g[0];
         l[2] = b[0];
         l[3] = 0xff;
         l += 4; r++; g++; b++;
   }
}

void
```

```
rgbatorgba(unsigned char *r,unsigned char *g,unsigned char *b,unsigned char *a,unsigned
char *l,int n) {
   while(n--) {
         l[0] = r[0];
         l[1] = g[0];
         l[2] = b[0];
         l[3] = a[0];
      l += 4; r++; g++; b++; a++;
   }
}

typedef struct _ImageRec {
   unsigned short imagic;
   unsigned short type;
   unsigned short dim;
   unsigned short xsize, ysize, zsize;
   unsigned int min, max;
   unsigned int wasteBytes;
   char name[80];
   unsigned long colorMap;
   FILE *file;
   unsigned char *tmp, *tmpR, *tmpG, *tmpB;
   unsigned long rleEnd;
   unsigned int *rowStart;
   int *rowSize;
} ImageRec;

static void
ConvertShort(unsigned short *array, long length) {
   unsigned b1, b2;
   unsigned char *ptr;

   ptr = (unsigned char *)array;
   while (length--) {
         b1 = *ptr++;
         b2 = *ptr++;
         *array++ = (b1 << 8) | (b2);
   }
}

static void
ConvertLong(unsigned *array, long length) {
   unsigned b1, b2, b3, b4;
   unsigned char *ptr;

   ptr = (unsigned char *)array;
   while (length--) {
         b1 = *ptr++;
```

```
        b2 = *ptr++;
        b3 = *ptr++;
        b4 = *ptr++;
        *array++ = (b1 << 24) | (b2 << 16) | (b3 << 8) | (b4);
    }
}

static ImageRec *ImageOpen(const char *fileName)
{
    union {
        int testWord;
        char testByte[4];
    } endianTest;
    ImageRec *image;
    int swapFlag;
    int x;

    endianTest.testWord = 1;
    if (endianTest.testByte[0] == 1) {
        swapFlag = 1;
    } else {
        swapFlag = 0;
    }

    image = (ImageRec *)malloc(sizeof(ImageRec));
    if (image == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }
    if ((image->file = fopen(fileName, "rb")) == NULL) {
        perror(fileName);
        exit(1);
    }

    fread(image, 1, 12, image->file);

    if (swapFlag) {
        ConvertShort(&image->imagic, 6);
    }

    image->tmp = (unsigned char *)malloc(image->xsize*256);
    image->tmpR = (unsigned char *)malloc(image->xsize*256);
    image->tmpG = (unsigned char *)malloc(image->xsize*256);
    image->tmpB = (unsigned char *)malloc(image->xsize*256);
    if (image->tmp == NULL || image->tmpR == NULL || image->tmpG == NULL ||
        image->tmpB == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(1);
```

```
    }

    if ((image->type & 0xFF00) == 0x0100) {
        x = image->ysize * image->zsize * sizeof(unsigned);
        image->rowStart = (unsigned *)malloc(x);
        image->rowSize = (int *)malloc(x);
        if (image->rowStart == NULL || image->rowSize == NULL) {
            fprintf(stderr, "Out of memory!\n");
            exit(1);
        }
        image->rleEnd = 512 + (2 * x);
        fseek(image->file, 512, SEEK_SET);
        fread(image->rowStart, 1, x, image->file);
        fread(image->rowSize, 1, x, image->file);
        if (swapFlag) {
            ConvertLong(image->rowStart, x/sizeof(unsigned));
            ConvertLong((unsigned *)image->rowSize, x/sizeof(int));
        }
    }
    return image;
}

static void
ImageClose(ImageRec *image) {
    fclose(image->file);
    free(image->tmp);
    free(image->tmpR);
    free(image->tmpG);
    free(image->tmpB);
    free(image);
}

static void
ImageGetRow(ImageRec *image, unsigned char *buf, int y, int z) {
    unsigned char *iPtr, *oPtr, pixel;
    int count;

    if ((image->type & 0xFF00) == 0x0100) {
        fseek(image->file, image->rowStart[y+z*image->ysize], SEEK_SET);
        fread(image->tmp, 1, (unsigned int)image->rowSize[y+z*image->ysize],
            image->file);

        iPtr = image->tmp;
        oPtr = buf;
        while (1) {
            pixel = *iPtr++;
            count = (int)(pixel & 0x7F);
            if (!count) {
```

```
                return;
            }
            if (pixel & 0x80) {
                while (count--) {
                  *oPtr++ = *iPtr++;
                }
            } else {
                pixel = *iPtr++;
                while (count--) {
                  *oPtr++ = pixel;
                }
            }
        }
    } else {
        fseek(image->file, 512+(y*image->xsize)+(z*image->xsize*image->ysize),
            SEEK_SET);
        fread(buf, 1, image->xsize, image->file);
    }
}

unsigned *
read_texture(char *name, int *width, int *height, int *components) {
    unsigned *base, *lptr;
    unsigned char *rbuf, *gbuf, *bbuf, *abuf;
    ImageRec *image;
    int y;

    image = ImageOpen(name);

    if(!image)
        return NULL;
    (*width)=image->xsize;
    (*height)=image->ysize;
    (*components)=image->zsize;
    base = (unsigned *)malloc(image->xsize*image->ysize*sizeof(unsigned));
    rbuf = (unsigned char *)malloc(image->xsize*sizeof(unsigned char));
    gbuf = (unsigned char *)malloc(image->xsize*sizeof(unsigned char));
    bbuf = (unsigned char *)malloc(image->xsize*sizeof(unsigned char));
    abuf = (unsigned char *)malloc(image->xsize*sizeof(unsigned char));
    if(!base || !rbuf || !gbuf || !bbuf)
      return NULL;
    lptr = base;
    for(y=0; y<image->ysize; y++) {
        if(image->zsize>=4) {
            ImageGetRow(image,rbuf,y,0);
            ImageGetRow(image,gbuf,y,1);
            ImageGetRow(image,bbuf,y,2);
            ImageGetRow(image,abuf,y,3);
```

```
            rgbatorgba(rbuf,gbuf,bbuf,abuf,(unsigned char *)lptr,image->xsize);
            lptr += image->xsize;
        } else if(image->zsize==3) {
            ImageGetRow(image,rbuf,y,0);
            ImageGetRow(image,gbuf,y,1);
            ImageGetRow(image,bbuf,y,2);
            rgbtorgba(rbuf,gbuf,bbuf,(unsigned char *)lptr,image->xsize);
            lptr += image->xsize;
        } else if(image->zsize==2) {
            ImageGetRow(image,rbuf,y,0);
            ImageGetRow(image,abuf,y,1);
            latorgba(rbuf,abuf,(unsigned char *)lptr,image->xsize);
            lptr += image->xsize;
        }
      else {
            ImageGetRow(image,rbuf,y,0);
            bwtorgba(rbuf,(unsigned char *)lptr,image->xsize);
            lptr += image->xsize;
   }}
   ImageClose(image);
   free(rbuf);
   free(gbuf);
   free(bbuf);
   free(abuf);

   return (unsigned *) base;
}
```

A.4. Visualizer.c

```
/*#include <GL/gl.h>*/
/*#include <GL/glu.h>*/
#include <GL/glut.h> /* OpenGL Utility Toolkit header */
#include <math.h>    /* for cos(), sin() */
#include <stdlib.h>
#include <stdio.h>
#include "texture.h"
#include "create_tree.h"

/* Some <math.h> files do not define M_PI .. */
#ifndef M_PI
#define M_PI 3.14159265
#endif

/* Atom and ion dimensions, all in Angstroms */
#define DEG_TO_RAD   (M_PI/180)
#define RADIUS 0.22
#define BOND 1.99
#define LENGTH  (RADIUS + BOND + RADIUS)
#define FIRST_ANGLE    30   /*This represents theta in degrees*/
#define LEVEL_1_WIDTH  110 /*This represents phi in degrees*/
#define NO_LEVEL_NODE  20
#define NEXT_LEVEL_DIFF  25
#define ESCAPE          27 /*This is Exc key */


#pragma warning(disable : 4305) /* stops warning about casting float to GLfloat */

extern struct tree_t *the_tree;

/* The atom locations are in spherical coordinates. */
typedef struct{
  double r, t, p; /* radius, theta, phi */
}Sphere_point;

Sphere_point   pos[];
double x, y, z;
```

```
struct tree_t *current_tree;
unsigned int current_level;
char  dir_path[80];
char  err[80];
char  root_node_name[MAX_NAME_LEN];
char  current_node_name[MAX_NAME_LEN];


/* Indice used for display lists */
GLuint tree =1;
GLuint picture =2;

/* Variables used for control the texture mapping*/
static int useTexture= 0;

/* Pointer point to some texture images readed from some particular '*.rgb' files */
GLubyte *pic;

/* Storages for texture arguments*/
static int components =2;
static int width[6]= {1, 1, 1, 1, 1, 1};
static int height[6]= {1, 1, 1, 1, 1, 1};
static GLuint texName[7];


/* Light position and color */
GLfloat lightPosition[4] = {10, 20, 15, 1};
GLfloat lightColor[]={1.0, 1.0, 1.0, 1.0};

/* Vectors for colours and properties of materials. */
const GLfloat red[] = { 1.0, 0.0, 0.0, 1.0 };
const GLfloat blue[] = { 0.0, 0.2, 1.0, 1.0 };
const GLfloat white[] = { 1.0, 1.0, 1.0, 1.0 };
const GLfloat yellow[] = { 1.0, 1.0, 0.0, 1.0 };
const GLfloat polished[] = { 100.0 };
const GLfloat dull[] = { 0.0 };


/* Variable control animation redering mode */
static int moving =0;

/* Title bar name */
static char *windowNameProject= "Master Project (Visualization Tool)";


/*Global variable for viewing transformation */
static GLfloat   ViewAngleY = 0.0;
static GLfloat   DeltaMove = 1.0;
```

```
static GLfloat   DeltaAngle = 15.0;

static GLfloat   x_value=0.0;
static GLfloat   y_value=0.0;
static GLfloat   z_value=10.0;


/*=========================================================
==================*/
void idle (void)
{
  ViewAngleY+=30;
 if(  ViewAngleY >=360)
   {
      ViewAngleY=0;
   }
 glutPostRedisplay();
}


/*=========================================================
==================*/

void visible(int state)
{
 if (state ==GLUT_VISIBLE)
   {
    if (moving)
        glutIdleFunc(idle);
   }
 else
   {
    if (moving)
        glutIdleFunc(NULL);
   }
}

/*=========================================================
================*/
void menu_select(int mode);

/* Construct menu */

void make_menu ()
{
   static int  top;
```

```
   top= glutCreateMenu(menu_select);
   if(useTexture==0) glutAddMenuEntry("Texture", 1);
   else glutAddMenuEntry("No texture", 2);

   glutAddMenuEntry("Start motion",3 /*M_Start_Motion*/);
   glutAddMenuEntry("Stop motion", 4 /*M_Stop_Motion*/);
   glutAddMenuEntry("Quit", 5 /*M_Quit*/);
   glutAttachMenu(GLUT_RIGHT_BUTTON);

}

/*=========================================================
============*/
void menu_select (int mode)
{
 switch (mode)
   {
   case 1:
   case 2:
    if(useTexture ==1)
        {
         useTexture=0;
         glutPostRedisplay();
        }
    else
        {
         useTexture=1;
         glutPostRedisplay();
        }
    make_menu();
    break;

   case 3:
    moving =1;
    glutIdleFunc(idle);
    break;
   case 4:
    moving =0;
    glutIdleFunc(NULL);
    break;
   case 5:
    exit(0);
    break;
   }
}

/*=========================================================
==========*/
```

```
static void usage(void)
{
 printf("\n");
 printf("usage: visualizer node_name directory_name\n");
 printf("\n");
 printf("display modules relationship\n");
 printf("        Esc:            quit\n");
 printf("        -> :            move viewpoint to left\n");
 printf("        <- :            move viewpoint to right\n");
 printf("        up arrow key:   move viewpoint closer\n");
 printf("        down arrow key: move viewpoint further\n");
 printf("        page up key:    move viewpoint up\n");
 printf("        page down key:  move viewpoint down\n");
 printf("\n");
#ifndef EXIT_FAILURE   /* should be defined by ANSI C
                  <stdlib.h> */
#define EXIT_FAILURE 1
#endif
 exit(EXIT_FAILURE);
}


/*==========================================================
============*/

void SpecialKey(int key, int x, int y)
 {
   if (glutGetModifiers() != GLUT_ACTIVE_ALT)
     {
         switch (key) /*<ALT> is NOT being held down.*/
           {
             case GLUT_KEY_LEFT:
                   ViewAngleY -= DeltaAngle;
                glutPostRedisplay();
                break;

             case GLUT_KEY_RIGHT:
                   ViewAngleY += DeltaAngle;
                   glutPostRedisplay();
              break;

             case GLUT_KEY_UP:
              z_value = z_value + DeltaMove;
              glLoadIdentity();
              gluLookAt(x_value, y_value, z_value, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
                   glutPostRedisplay();
              break;
```

```
        case GLUT_KEY_DOWN:
                z_value = z_value - DeltaMove;
                glLoadIdentity();
                gluLookAt(x_value, y_value, z_value, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
                glutPostRedisplay();
                break;

          case GLUT_KEY_PAGE_UP:
              x_value = x_value -DeltaMove;
                y_value = y_value -DeltaMove;
                glLoadIdentity();
                gluLookAt(x_value, y_value, z_value, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
                glutPostRedisplay();
                break;

          case GLUT_KEY_PAGE_DOWN:
              x_value = x_value +DeltaMove;
                y_value = y_value +DeltaMove;
                glLoadIdentity();
                gluLookAt(x_value, y_value, z_value, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
                glutPostRedisplay();
                break;

              }
    }
}

/*=================================================
=============*/
void keyboard (unsigned char key, int x, int y)
{
 if (key ==ESCAPE)
   exit(0);
}


/*=================================================
=======*/

void init(void)
{

  GLfloat values[2];


  glClearColor (0.2f, 0.2f, 0.6f, 1.0f);
```

```
glShadeModel (GL_SMOOTH);
glEnable(GL_DEPTH_TEST);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
glLightfv(GL_LIGHT0, GL_SPECULAR, white);

glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);
glGetFloatv(GL_LINE_WIDTH_RANGE, values);
/* Using Blending wiht Anti-Aliasing */
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable (GL_LINE_SMOOTH);
glEnable (GL_POINT_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
glLineWidth (1.0);

/* Draw the whole tree structure here */
glNewList(tree, GL_COMPILE);
  build_a_subtree();
glEndList();


/* Create texture objects by reading from a file*/
glGenTextures(1, texName);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
pic=(GLubyte *)read_texture("sample2.rgb", &width[0], &height[0], &components);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
gluBuild2DMipmaps(GL_TEXTURE_2D, 4, width[0], height[0], GL_RGBA,
GL_UNSIGNED_BYTE, pic);


glNewList(picture, GL_COMPILE);
  glPushMatrix();
    glBindTexture(GL_TEXTURE_2D, texName[0]);
        glBegin(GL_QUADS);
          glTexCoord2f(0.0, 0.0); glVertex3f(-200.0, -200.0, -200.0);
```

```
        glTexCoord2f(0.0,1.0); glVertex3f(-200.0, -200.0, -200.0);
        glTexCoord2f(1.0, 1.0); glVertex3f( 200.0, 200.0, -200.0);
        glTexCoord2f(1.0, 0.0); glVertex3f( -200.0, 200.0, -200.0);
       glEnd();
   glPopMatrix();
 glEndList();

 /* Initialize menu bar */
 make_menu();

}

/*=================================================
==============*/
/* dsiplay() function uses the global variables to draw the scene */
void display (void)
{
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

 /* Control the light in the redering mode */
 /*glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
  *glPushMatrix();
  *glDisable(GL_LIGHTING);
  *glTranslatef(lightPosition[0], lightPosition[1], lightPosition[2]);
  *glEnable(GL_LIGHTING);
  *glPopMatrix();
  */

 /* Draw the whole tree structure created by display list */
 glPushMatrix();
 glRotatef(ViewAngleY, 0.0, 1.0 ,0.0);
 glCallList(tree);
 glPopMatrix();

 /* Draw a textured background */
 glPushMatrix();
 if(useTexture)
   {
    glEnable(GL_TEXTURE_2D);
    glDisable(GL_LIGHTING);
    glCallList(picture);
   }
 if(useTexture)
   {
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_LIGHTING);
   }
 glPopMatrix();
```

```
 glutSwapBuffers();
 glFlush();

}


/*=====================================================
=============*/
void reshape(int width, int height)
{
 double y = (double) height/(double) width;
 glViewport (0, 0, (GLsizei) width, (GLsizei) height);
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
  y = (double) height / (double) width;
 gluPerspective(60.0, (GLfloat) width/(GLfloat) height, 0.5, 520.0);

 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();

 gluLookAt(0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

}

/*=====================================================
=============*/
build_a_subtree()
{

 struct dupl_sub_tree  sub_tree_count;
 int  i;
 int  pos_size;
 Sphere_point  pos[NO_LEVEL_NODE];
 char *p;


  init_called_levels();
  sub_tree_count.count =0;
  compute_called_levels(current_tree, 0, sub_tree_count);

  if (current_tree->called_count >1)
    {
     /* printf("called_count=%d\n", current_tree->called_count);*/
     /* display a red sphere */
     glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, red);
     glMaterialfv(GL_FRONT, GL_SPECULAR, white);
     glMaterialfv(GL_FRONT, GL_SHININESS, polished);
```

```
    glTranslatef(0.0, 3.0, 0.0);
    glutSolidSphere(RADIUS, 50, 50);

    glPushMatrix();
    glDisable(GL_LIGHTING);
    glTranslatef(0.0,RADIUS, 0.0);
    glScalef(0.003,0.003,0.003);
    glColor3f(1.0, 1.0, 0.0);

    for (p = current_node_name; *p; p++)
       glutStrokeCharacter(GLUT_STROKE_ROMAN, *p);

    glEnable(GL_LIGHTING);
    glPopMatrix();

  }
 else
  {
      /* display a yellow sphere */
    glColor3f(1.0, 1.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow);
      glMaterialfv(GL_FRONT, GL_SPECULAR, white);
      glMaterialfv(GL_FRONT, GL_SHININESS, polished);
    glTranslatef(0.0, 3.0, 0.0);
      glutSolidSphere(RADIUS, 50, 50);

      glPushMatrix();
      glDisable(GL_LIGHTING);
      glTranslatef(0.0,RADIUS, 0.0);
      glScalef(0.003,0.003,0.003);
      glColor3f(1.0, 1.0, 0.0);

      for (p = current_node_name; *p; p++)
       glutStrokeCharacter(GLUT_STROKE_ROMAN, *p);

      glEnable(GL_LIGHTING);
      glPopMatrix();
  }
 /* Recursive function call */
 build_cone_scheme(pos, &pos_size, LEVEL_1_WIDTH, current_tree);
 build_sub_tree(pos,pos_size,current_node_name,1,current_tree, LEVEL_1_WIDTH);
}


/*=================================================
=========*/
build_cone_scheme(pos, pos_size, width, cur_tree)
 Sphere_point   pos[];
```

```
int          *pos_size;
int          width;
struct tree_t  *cur_tree;
{
 int          i;
 struct edge_t  *uses;


 (*pos_size)=0;
 if (cur_tree)
  {
   uses = cur_tree->uses;
   while (uses)
       {
        (*pos_size)++;
        uses = uses->next_edge;
       }
  }

 for (i =0; i< (*pos_size); i++)
  {
      pos[i].r = LENGTH;
      pos[i].t =(FIRST_ANGLE + i*(360/(*pos_size))) * DEG_TO_RAD;
      pos[i].p =width * DEG_TO_RAD;
  }
}

/*==================================================
==================*/
build_sub_tree(pos, pos_size, root_name, level, cur_tree, level_width)

 Sphere_point          pos[];
 int                pos_size;
 char                *root_name;
 int                level;
 struct tree_t         *cur_tree;
 int                level_width;

 {
  int                i;
  int                j;
  int                next_pos_size;
  struct edge_t         *uses;
  Sphere_point          next_pos[NO_LEVEL_NODE];
  char                *p;


  if (!cur_tree) return;
```

```
/*printf("build_sub_tree: root%s no leafs= %d\n", cur_tree->node_name, \
 pos_size);
 *printf("sub_tree_drawn=%d\nsmallest_called_level=%d\nlevel=%d\n",\
 cur_tree->sub_tree_drawn, cur_tree->smallest_called_level, level);
 *printf("called_count=%d\n", cur_tree->called_count);
 */
if (cur_tree->sub_tree_drawn || cur_tree->smallest_called_level+1 !=level)
 return;

cur_tree->sub_tree_drawn = 1;

if(level<current_level+1)
{
uses= cur_tree->uses;

for (i=0; i<pos_size; i++)
 {

      glColor3f(1.0, 0.0, 0.0);

      glPushMatrix();
      glDisable(GL_LIGHTING);
      glRotatef((FIRST_ANGLE+i*(360/pos_size)), 0.0, 1.0, 0.0);
      glRotatef(level_width, 1.0, 0.0, 0.0);
      glTranslatef(0.0, RADIUS, 0.0);

      glBegin(GL_LINES);

      glVertex3f(0.0, 0.0, 0.0);
      glVertex3f(0.0, BOND, 0.0);

      glEnd();
      glEnable(GL_LIGHTING);
      glPopMatrix();

      x = pos[i].r*sin(pos[i].t)*sin(pos[i].p);
      y = pos[i].r*cos(pos[i].p);
      z = pos[i].r*sin(pos[i].p)*cos(pos[i].t);

      if (uses->node->sub_tree)
        {
         if (uses->node->sub_tree->called_count >1)
           {
              /* printf("uses->node->sub_tree->called_count=%d\n",\
                uses->node->sub_tree->called_count);*/

              /* display a red sphere */
              glColor3f(1.0, 0.0, 0.0);
```

```
        glPushMatrix();

        glTranslatef(x, y,z);
        glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, red);
        glMaterialfv(GL_FRONT, GL_SPECULAR, white);
        glMaterialfv(GL_FRONT, GL_SHININESS, polished);

        glutSolidSphere(RADIUS, 50, 50);
        /*display the stroke cahracter*/
        glPushMatrix();
        glDisable(GL_LIGHTING);
        glTranslatef(0.0,RADIUS, 0.0);
        glScalef(0.003,0.003,0.003);
        glColor3f(1.0, 1.0, 0.0);

        for (p = uses->node->node_name; *p; p++)
          glutStrokeCharacter(GLUT_STROKE_ROMAN, *p);
        /*finishing stroke at here */

        glEnable(GL_LIGHTING);
        glPopMatrix();



        glPopMatrix();
    }
  else
    {

        /* display a yellow sphere */
        glColor3f(1.0, 1.0, 0.0);

        glPushMatrix();
        glTranslatef(x, y,z);
        glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow);
        glMaterialfv(GL_FRONT, GL_SPECULAR, white);
        glMaterialfv(GL_FRONT, GL_SHININESS, polished);


        glutSolidSphere(RADIUS, 50, 50);

        /*stroking character at here*/
        glPushMatrix();
        glDisable(GL_LIGHTING);
        glTranslatef(0.0,RADIUS, 0.0);
        glScalef(0.003,0.003,0.003);
        glColor3f(1.0, 1.0, 0.0);
```

```
            for (p =uses->node->node_name; *p; p++)
              glutStrokeCharacter(GLUT_STROKE_ROMAN, *p);

            glEnable(GL_LIGHTING);
            glPopMatrix();
            /*finishing at here*/

            glPopMatrix();

          }
        }
      else
        {
         /* display a blue sphere */

         glColor3f(0.0, 1.0, 1.0);

         glPushMatrix();
         glTranslatef(x, y,z);

         glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, blue);
         glMaterialfv(GL_FRONT, GL_SPECULAR, white);
         glMaterialfv(GL_FRONT, GL_SHININESS, polished);
         glutSolidSphere(RADIUS, 50, 50);

         glPushMatrix();
         glDisable(GL_LIGHTING);
         glTranslatef(0.0,RADIUS, 0.0);
         glScalef(0.003,0.003,0.003);
         glColor3f(1.0, 1.0, 0.0);

         for (p = uses->node->node_name; *p; p++)
           glutStrokeCharacter(GLUT_STROKE_ROMAN, *p);

         glEnable(GL_LIGHTING);
         glPopMatrix();

         glPopMatrix();
        }


      uses = uses->next_edge;
}/*for*/

    uses=cur_tree->uses;

    for (i=0; i<pos_size; i++)
```

```
        {
         /*printf("pos_size=%d\n", pos_size);*/

         x = pos[i].r*sin(pos[i].t)*sin(pos[i].p);
         y = pos[i].r*cos(pos[i].p);
         z = pos[i].r*sin(pos[i].p)*cos(pos[i].t);


         build_cone_scheme(next_pos, &next_pos_size,
                        level_width+NEXT_LEVEL_DIFF, uses->node->sub_tree);

         glPushMatrix();
         glTranslatef(x, y, z);
         build_sub_tree(next_pos, next_pos_size, uses->node->node_name,\
            level+1, uses->node->sub_tree, level_width+NEXT_LEVEL_DIFF);

         glPopMatrix();

         uses= uses->next_edge;
        }
   }
  }



/*==========================================================
==========*/
int main (int argc, char **argv)
{

 int SIZE = 500;
 char *name;

 if (argc !=3)
   {
    printf("Please enter correct parameters!\n");
    usage();
   }
 else
   {
    /* build up the parser */
    the_tree =0;
    strcpy(root_node_name, argv[1]);
    strcpy(dir_path, argv[2]);
    build_tree(root_node_name, 0);
    strcpy(current_node_name, root_node_name);
    current_tree = the_tree;
```

```
    current_level =3;

    /* Initialize */
    glutInitWindowSize(SIZE, SIZE);
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH);
    name = windowNameProject;
    glutCreateWindow(name);

    init();

    /* Register the callback function display */
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutSpecialFunc(SpecialKey);
    glutKeyboardFunc(keyboard);
    glutVisibilityFunc(visible);

    glutMainLoop();
   }

 return 0;      /* ANSI C requires main to return int. */
}
```