SYSTEM VISUALIZER

Kim Thang Vu

A thesis
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

March 1997

0-612-26021-6

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Kim Thang Vu**
Entitled: **System Visualizer**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Chair
______________________ Examiner
______________________ Examiner
______________________ Supervisor

Approved ______________________
Chair of Department or Graduate Program Director

__________ 19 ______ ______________________
Dean of Faculty

Abstract

System Visualizer

Kim Thang Vu

It is a complex and difficult task to maintain and enhance existing large software systems. One of the main reasons is that the task of trying to understand such an existing system is very difficult. Software engineering tools are needed to improve and accelerate understanding. Software tools do this by displaying software in graphical forms. Traditional tools provide only two dimensional display of the software. Two dimensional visualization has certain limitations. This thesis attempts to remove some of the limitations associated with two dimensional displays by providing an initial research into three dimensional visualization of software.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Peter Grogono for his continous guidance and enthusiastic support. It is his valuable input and encouragement that made this thesis possible.

Also, I would like to thank all my brothers and sisters for their encouragement and support.

Finally, I would like to dedicate this work to my mom, my wife and my daughter with great love and gratefulness.

Contents

List of Figures

Chapter 1

Introduction

1.1 Review of Software Engineering

1.1.1 Software Engineering

In the early years of computerization, computer programs were in general small, especially when they are compared to software applications which are being developed now. The programmers who wrote these early programs were also the users of the programs, and they were often experts in the application area concerned. The programs were written to solve problems mostly of a technical nature, and the emphasis was on expressing known algorithms efficiently in some programming language.

Today software applications are different from those early written programs in many aspects. Today software systems are often very large. They are developed by different teams. These teams collaborate over periods which could span several years. The developers are not the future users of the system. Moreover these developers may have little expert knowledge of the application area in question.

Programming techniques have been behind the developments of software both in size and complexity. To many people, programming is still an art, and has never become a craft, let alone a science.

As a result, software is often delivered too late, programs do not behave as the user expects. Programs are rarely adaptable to changed circumstances, and many errors are detected only after the software has been delivered to the customer [10, page 2].

The cost of software is also of crucial importance. The cost is associated with both software development and software maintenance in order to keep the system operational once it has been delivered to the customer.

The productivity in developing the software is closely related with cost. The growth of the population of people working in data processing has not kept up with the increase of the quest for data processing personnel. The gap between demand and supply keeps growing. The result is both a backlog with respect to the maintenance of existing software and a slowing down in the development of new applications.

Cost and productivity are two major problems associated with software. We must do something to resolve these problems. However, these are not the only problems. Society is increasingly dependent on software. The quality of the systems we develop is increasingly determining the quality of our existence. On a larger scale, software bugs could put the world in danger. As an example of this, in June 1980, the United States brought their atomic bombers and nuclear missiles to an increased state of alarm when, because of a computer error, a false alarm indicated that the Soviet Union had started a missile attack [10, page 4]. On a smaller scale, errors in software may also have very unfortunate consequences, such as transaction errors in bank traffic, or a reminder to finally pay a bill of $0.00.

This all marks the enormous importance of the field of software engineering. There is no doubt that better methods and techniques are needed for software development. We need to be able to develop systems that better fit the user needs. More effective methods in software development need to be researched in order to develop software systems which are more reliable and to reduce the cost associated with the development of these systems. Quality and productivity are the two most central themes in the field of software engineering.

Shouldn't it be possible to build software in the way bridges and houses are built, starting from a theoretical basis and using sound and proven design and construction techniques, as in other engineering fields? This is one of the major problems software engineering is trying to tackle.

Software engineering is the systematic approach to the development, operation, maintenance, and retirement of software [10, page 5]. There are some essential characteristics of the field of software engineering:

- Software engineering concerns the construction of large programs. Small programs are generally written by one person in a relatively short period of time. A large software development project refers to multi-person jobs that span more than half a year. The traditional programming techniques and tools are primarily aimed at supporting development of small programs.
- Large problems are difficult to solve. In general, these problems cannot be solved as a whole. The problem often must be split into smaller parts so that each individual part can be understood and therefore can become solvable. The communication between the parts also need to be simple. The total complexity of the problem does not decrease in this way, but it does become manageable.
- Good cooperation between software developers is also very important in constructing large software applications. The problems these applications try to solve are large. We need to have a large number of software developers to work together to solve those problems. There must be clear arrangements for the distribution of work, methods of communication, responsibilities, etc.
- A lot of software models parts of reality, such as processing requests in a library. Reality evolves with time. Therefore the software which models the associated reality has to evolve as well. We have to bear this evolution in mind during development.
- The productivity and efficiency associated with software development is another important aspect of software engineering. The cost and time in developing software is generally high. The same thing also applies to the maintenance of software. Requests for new software applications keep increasing. There are not enough software developers to satisfy all these requests. Therefore, software engineering has to deal with finding better and more efficient methods and tools for the development and maintenance of software.

- Software is developed because people need to use it. Therefore, software has to be able to support its users effectively. The functionality offered should fit users' tasks. It is not sufficient to build the system the right way, we also have to build the right system.

When developing large computer programs, a phased approach is followed. First, the problem is analyzed. Then the system is designed, implemented and tested. Each of these actions compose a phase during the development of software projects. This practice has a lot in common with the engineering of physical products. Software engineering, however, differs from the engineering of physical products in some essential ways.

Software models part of the real world surrounding us, like banking or the reservation of air line seats. The world around us changes over time. So the corresponding software has to change too. It has to evolve together with the changing reality. Much of what we call software maintenance, actually is concerned with ensuring that the software keeps pace with the real world being modeled.

The most important problems that software engineering tries to tackle are the quality of the software being delivered and the productivity of the people working in the field.

1.1.2 Software Engineering Tools

The demand for software grows faster than the increase in software development productivity and available manpower. The result is an ever increasing shortage of personnel. We are less and less able to satisfy the quest for software [10, page 373]. In order to resolve the problem, we must look for techniques that result in significant productivity gains.

Automation is one of the most obvious ways we can do to resolve the problem. The computer itself can be used as a tool to produce software.

The computer has long been employed for the implementation of software. Programmers have a number of tools to use to implement software, such as compilers, linkers and loaders. Also during testing, tools like test drivers and test harnesses have been used for a long time.

The development of tools to support earlier phases of the software life cycle is however a lot more recent. Software tools such as the ones which aid the drawing and validation of data flow diagrams have become available only recently.

Today, there are various software engineering tools which help increase the productivity of software developers and help produce software with better quality. These tools help the developers in various activities during the software development life cycle. Tools may support checking conformance to standards; tools may help to quantify the degree of testing; tools may support progress tracking; and so on.

The application of tools in the software development process is referred to as CASE — Computer Aided Software Engineering. Today, the number of CASE tools is overwhelming.

Software tools can be characterized as follows [10, page 374]:

- Tools which support the requirements specification or global design phase. These tools thus can be called an *analyst workbench.*
- Tools which support the programmer for the implementation of software which includes coding and testing are therefore called *programmer workbench.*
- Tools which support management tasks are called *management workbench.*
- There are also tools intended to support all phases of the software life cycle. These tools are named *Integrated Project Support Environment.*

As we may notice from the above classification, there is a certain relation between a given model of the software life cycle and a corresponding tool collection. Starting from a life cycle model, we can decide on a set of techniques that fit the model. Then we can look for software tools that would support such techniques.

Software tools often render some process models, such as prototyping and evolutionary development. There is an interaction between trends in support environments and trends in process models. Based on trends that have a major impact on support environments, we can distinguish different categories of software engineering tools [10, page 375]:

- Tools which are suited for the support of software development in a specific programming language.
- Tools which are aimed at manipulating program structures. These tools are used in environments which are based on the structure of programming languages.
- There are also various *toolkits*. The support offered by toolkits is independent of a specific programming language. A toolkit merely offers a set of useful bulding blocks. In particular, toolkits tend to contain tools that specifically support the development of large software systems.
- There are tools which are based on certain techniques used in specific phases of the software life cycle. There exist also integrated tool sets which aim at supporting the full spectrum of the software life cycle in a coordinated fashion.

1.2 Thesis Contribution

Perhaps the most difficult software engineering projects involve the development of large software systems. These large team projects, often in maintenance mode, require enhancements involving subtle changes to complex legacy code written over many years. Under these circumstances, programmer productivity is low, changes are more likely to introduce errors, and software projects are often late.

One of the main reasons for these problems is that programmers do not quite understand the software system they are modifying and enhancing. However, we cannot really blame these programmers. Large software systems can contain millions of lines of code. It is a very difficult task to understand such a software system.

It is obvious that software tools are needed to help software engineers, who have to maintain and enhance existing large software systems, understand these large software systems. There is a number of existing software tools which perform this task. These tools mainly use graphical techniques to make software visible by displaying programs, program structures and program behaviours. These traditional tools usually visualize programs in two dimensional space. There are limitations associated with displaying software in two dimensional space. Space is an example of one of these limitations. Given the limited space of a window, there is only room for a certain number of nodes on the screen. A three dimensional display of this number is more effective than a two dimensional display. A 3D image can be viewed from different directions. As a result, a 3D image may look confusing from one direction but may look very clear from another direction. Also in three dimensional space, objects which are closer to our eyes appear larger, and objects which are farther from our eyes appear smaller. Therefore, a three dimensional image is more comprehensible.

The thesis attempts to solve some of the limitations associated with two dimensional display. It has provided a software tool to extract thc structure of a software system, and to visualize the information in three dimensional space. To be more specific, the tool attempts to extract module relationships of a software system, and tries to find a good way to display this information in three dimensional space. Since the tool visualizes the structure of a software system, the tool has been named *System Visualizer*.

1.3 Organization of Thesis

The thesis is made up of seven chapters. The following paragraphs briefly describe the contents of each chapter.

- Chapter 1, Introduction: provides a review of software engineering and software engineering tools. The Chapter then introduces the work provided by the thesis, and briefly describes the organization of the thesis.

- Chapter 2, Visualization: the Chapter first discusses a history of the development of computer outputs. It then talks about visualization and the importance of visualization. The discussion then narrows down to software visualization, and then to the visualization of software structure.

- Chapter 3, Related Work: this Chapter discusses work which has been done in software visualization. Two related articles and a few chapters in a book which talks about software visualization are discussed in Chapter 3.

- Chapter 4, Design: the Chapter first describes the functional and user interface requirements which System Visualizer has to meet. The Chapter then discusses the design of the tool and makes some design decisions.

- Chapter 5, Implementation: discusses the actual implementation of System Visualizer. Some problems which were encountered during the implementation are also discussed.

- Chapter 6, Discussion: provides a discussion of whether System Visualizer has met its design goals and of the aspects the tool has contributed to software engineering.

- Chapter 7, Future Research and Conclusion: the Chapter discusses possible future enhancements which could be made to System Visualizer, and concludes the thesis by summarizing the main points which the thesis has accomplished.

Chapter 2

Visualization

2.1 Development of computer output

In the very early stage of computers, the user had to use punched cards to input programs into the computer. Computers executed in batch mode, and mainly performed computations. The output of the computations went to line printers. Line printers were only able to print the results of computations in text form.

Then terminals were introduced. The terminal was connected to the computer, and the user was able to input programs into the computer by typing program text on the terminal, and save the program into a file. Early terminals were 'dumb', and could only display text line by line. As a result, only line editors were available for the user to use for writing programs. The computer still performed basically computations, and the results of the computations could then go to either line printers or dumb terminals.

Terminals were then improved so that they were able to display text in a full screen mode. Full screen editors became available. The user was then able to use these editors to write programs more effectively. The results of program execution could be output on terminal screen in full screen mode. Therefore, the programmer was able to format the results in a good way so that the user could analyze the results more effectively. These terminals were, however, able to display text only. And also, at any point in time, the results of only one program could be displayed since there was only one window, which was the screen of the terminal, available for display.

Graphic terminals were developed only when fast processors and large memories became available. Terminals were then able to display not only text, but graphics also. Windows were also developed. With the development of windows, a single terminal could then have multiple virtual screens. The outputs from several different programs were then able to go to several windows simultaneously. The early graphic terminals were, however, only able to display simple graphics, and these terminals were still monochrome.

Computers no longer executed in batch mode. Many programs were developed for users' interactions with computers. Users of these programs could then interact with the computer to obtain desired information. To help ease the development of interactive programs, graphical user interface packages have been developed. These packages provide high level primitives for the application software to use. The application software just needs to concentrate on how to provide good interfaces for the user to use.

With the introduction of graphic terminals, the field of computer graphics was born. But, because of the limited graphic display capabilities of the early graphic terminals, and because the field was in its very early stage, computer graphics was not widely recognized.

Today, high resolution, color terminals are available. These modern terminals can display very sophisticated colored images. The speed of displaying is also very fast. With the use of graphical user interface packages, users can also interact with the graphics displayed on these terminals.

The field of computer graphics has also become mature. As a matter of fact computer graphics has become indispensable in countless areas of human activity. High resolution, color displays are used for 'visualization' in endless fields such as aerodynamics, medicine, electrical engineering, mathematics, business, science, education, and others.

Researchers in medicine use graphics generated by the computer to study and analyze medical data. Since processors become very fast, computers have very large memories, and modern terminals can display graphics in realtime, medical doctors can now determine the condition of a patient by looking at the graphics displayed on a terminal which contains the information.

Scientists can track a storm's progress for weather prediction by watching the changes in temperature, in pressure, and in other fields which are visualized on a changing graphic displayed on a terminal.

Mathematicians use computer visualization to study graphs of complicated functions [3, pages 164–167], or to perform research on the field of geometry [4].

With the use of computer aided design tools, which are graphic based, an electrical engineer can design sophisticated electrical systems and an architect can design very complex buildings [3, pages 138–151].

These are just a few examples of how computer visualization is used today in various professions. Software engineers have developed all these sophisticated tools for other professionals to use. But the software engineer himself does not use much visualization for software development.

It is true that there are CASE tools which help the software engineer do software development. But, these CASE tools have not been mature enough to efficiently speed up the development in the field of software engineering.

2.2 Visualization

Visualization enables people to use a natural tool of observation and processing, their eyes as well as their brain, to extract knowledge more efficiently and find insights. Visualization is clearly not only useful to engineers and other professionals dealing with problems in the real world, such as air flow around airplane wings, mechanical stress modelling, and three dimensional reconstructions of brain scans. It has also become a key tool for pure mathematicians, and in a comparatively short time.

Let us consider a simple example. Suppose we would like to study and understand the behaviour of a function, such as the exponential function. For every value of x, there is a corresponding value of the function e^x. It would be a tedious task to study the function by looking at a table of numbers. However, if the function can be visualized by plotting the value of the function against each value of x, then it becomes a lot easier to understand the function.

Now, for a typical engineering example, such as trying to understand the flow of air over the wing of an aircraft. It would be impossible to look at a huge table of numbers, and be able to understand it. Apparently, the only practical way to study this kind of problem is to look at a three dimensional display corresponding to this huge table of numbers.

Today, computers and data gathering devices spew out floods of data every second. Obviously, simply wading through pages and pages of numbers is a tedious route to insight. On the other hand, when data are presented as animated color images, large quantities can be assimilated by the visual cortex, perhaps as much as several gigabytes of data every second.

Engineers, scientists, physicians, and professionals in many other fields need an alternative to numbers and symbols. Visualization provides such an alternative by transforming numbers and symbols into the universal language of visuals, thereby enabling these researchers and practitioners to observe their data, computations, or models in a more meaningful form. This ability to visualize complex data sets, synthetic models, and results from supercomputer simulations is essential in provoking insights, enabling communication of these insights to colleagues, and confirming the integrity of observation.

Visualization means the presentation of large quantities of data in a visual form that can be rapidly assimilated and understood by a person. This is exactly what we do when we plot the exponential function against each value of x. The same applies when a three dimensional display is drawn to represent a huge table of numbers for the case of studying the flow of air over the wing of an aircraft.

Scientific numerical data and their representation are, for the most part, spatial in concept. In other words, they can be mapped by means of a three dimensional representation appropriate to their context. For example, different colored stripes representing temperature or stress can be mapped, in one way or another, directly to the additional representations of these objects to which they apply, whether of a simulation of a missile in flight or a jet exhaust stream.

Information is a more abstract concept. It is used for items that rarely have any correspondence to a physical space. Consider a list of documents related by their subject matter and rated by their relevance to some search criterion. There is nothing intrinsic to such a list that dictates its graphical representation, even though, for example, a document is often said to be of high or even topmost important. Therefore, in order to visualize information, it needs to be mapped to some physical space.

Visualization, often referred to as scientific visualization or visualization in scientific computing, helps us extract useful information from complex or often voluminous data sets through the use of interactive graphics and imaging. It also provides processes for steering the data set and seeing what heretofore may have been invisible, thereby enriching existing investigation methods.

Visualization encompasses computer graphics, image processing, computer vision, computer aided design, signal processing, user interface design, cognitive science, and computational geometry, taking its foundation material from these and related fields and at the same time unifying them.

Scientific visualization aims to devise algorithms and methods that transform massive scientific data sets into pictures and other graphic representations that facilitate comprehension and interpretation. In many scientific domains, analysis of these pictures has motivated further scientific investigation, laboratory experiments, numerical simulations, or remote observations.

Visualization technology has generated much excitement and enthusiasm among its practitioners and has already significantly changed the way scientists do science, the way engineers design, and the way physicians deliver health care. The results of visualization research most directly affect the quantitative and qualitative aspects of users' work, but the general utility of visually displayed information is increasingly affecting other aspects of their work as well.

Supercomputers, satellites, medical scanners, microscopes, radio telescopes, geophysical sensors, information superhighways, and geometric and computational models all generate huge amounts of scientific, engineering, and medical data. The many techniques that help us analyze and comprehend this data by providing visual representations are included in the definition of visualization.

Because tremendous benefits accrue from direct interaction with data sets or simulations, the trend is toward interactive visualization. Interactive visualization means that the user can provide input so that the way the data is visualized will be changed.

Interactive systems are most effective when the results of models or simulations are in a dynamic visual form. Thus, visualization includes techniques that help users interact with visual presentations of scientific data. Visualization also includes the hardware, software, and systems issues required to support the interactive mode.

Visualization today has become a discipline of its own. The importance of visualization research and development has increasingly been recognized. Visualization is now the subject of considerable research and development activity at many academic institutions, as well as in industrial companies around the globe.

2.3 Software Visualization

Large software systems are complex and difficult to maintain. Source listings of production-sized systems may consist of hundreds of thousands of lines spread over hundreds of files. Even a seemingly simple, small project, such as a spreadsheet, is quite complicated. It is time consuming and costly to understand, repair, maintain and enhance code in large systems.

As the software gets older, knowledge of code decays. The design documents are usually out of date, leaving the code as the only guide to system behavior. It is a tedious task to reconstruct the design and system behavior by studying the code.

Software visualization can help software engineers understand, maintain and enhance large software systems more efficiently. Software is intangible. It has no physical shape or size. After it is written, the code is saved into files which are kept on disks. Software visualization tools use graphical techniques to make software visible by displaying programs, program artifacts, and program behavior. The essential idea is that visual representations of software can help make understanding software quicker and easier.

A few basic properties of software systems can be visualized:

- Systems design and architecture: the structure of software systems, such as module relationships, calling relationships between functions, can be visualized using directed graphs. These graphs show the relationships among software entities. For example, a node can represent a procedure, and an edge can represent a calling relationship between the two procedures.
- Systems behavior: graphical representations of data structures and their motions can be used to illustrate the behavior of system's algorithms.
- The code itself: some editors display the code using different colors for different types of code. Comments can be displayed in red. Keywords can be displayed in blue. Built-in types can be displayed in green. Pretty printers also use special fonts and colors to distinguish keywords and so forth.

Software engineers usually work with these large and complex software systems. In order to maintain and enhance these systems, the engineers have to understand the structure of the system, such as function calling relationships. How can software visualization help software engineers perform their tasks quickly and efficiently ?

2.4 Visualizing software structure

Traditionally, software structure, such as calling relationships between functions, is visualized using 'box and arrow' graphs. These graphs usually consist of node and link diagrams which are carefully arranged by sophisticated layout algorithms to show the underlying structure of complicated software systems.

For example, a node can represent a function, and a link between the two nodes can represent a calling relationship between the two functions. A node can be visualized as a circle in two dimensional space, and a link can be visualized as a line between the two circles. Figure 1 is an example of such a two dimensional approach to visualizing software structure:

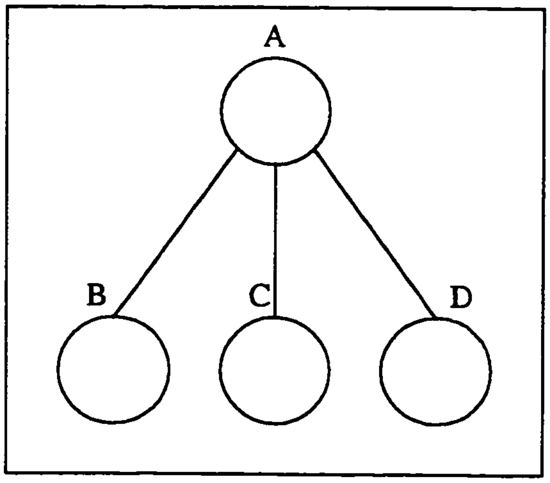


Figure 1: An example of a 2D visualization of software structure

In Figure 1, circles A, B, C, and D represent functions A, B, C, and D respectively. The lines connecting circle A and circles B, C, and D represent calling relationships between function A and functions B, C, and D respectively. In the example, function A calls functions B, C, and D.

The two dimensional approach to visualizing software structure has some limitations:

- There is not enough room for many nodes on the screen. As we can see from Figure 1, with the space of the given window, we can draw at most 4 circles to represent 4 functions. We can fit in more circles if we use smaller circles and a larger window, but the complexity of this kind of display is clearly limited.

- The links tend to cross over, confusing the diagram.
- All nodes are of the same size regardless of their positions.

In order to overcome some of the limitations imposed by 2D visualization, System Visualizer uses a three dimensional approach to visualizing software structure. In particular, the tool was developed to provide a three dimensional display of module interconnections. Figure 2 shows an example of how software structure can be visualized in three dimensional space:

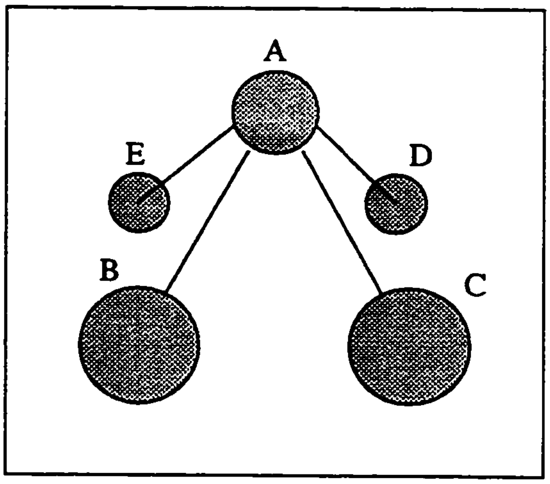


Figure 2: An example of a 3D visualization of software structure

In Figure 2, spheres A, B, C, D, and E represent modules A, B, C, D, and E respectively. The lines connecting sphere A and spheres B, C, D, and E represent the uses relationship between module A and modules B, C, D, and E respectively. In the example, module A uses modules B, C, D, and E.

The three dimensional approach to visualization has some advantages over the two dimensional approach:

- There is room for more nodes in three dimensional space. In Figure 2, the window size is exactly the same as that in Figure 1. Figure 2 displays one node more than Figure 1 does. Moreover, as we can see, there is still room to display more nodes in Figure 2.

- Modules appear larger if they are closer to the viewer. This increases the clarity of the 3D image and helps the viewer interpret the picture better. In Figure 2, modules B and C are closer to the viewer, therefore they appear larger.

- Rotation allows the user to select a view that shows part of the system clearly. A 3D picture may look confusing from one direction, i.e. spheres may overlap each other. However, when the image is rotated as if it is looked at from another direction, then part of the image may become very clear. In Figure 2, the image has been rotated to a position where the whole structure looks very clear.

- Colour coding can be used to distinguish different kinds of modules. For example, leaf modules, i.e. modules which do not use any other modules of a software system, can be colored blue. Non-leaf modules, i.e. modules which use some other modules of the system, can be colored yellow.

Three dimensional visualization clearly has advantages over the traditional two dimensional approach. Unfortunately, there has been very little done on 3D visualization, especially on software visualization. The purpose of this thesis is to try to contribute a part to fill up this lack.

Chapter 3

Related Work

As stated in the previous chapter, there has been very little work done on visualization, especially on software visualization, e.g. visualizing source code, visualizing the design by extracting information from source code, etc. This chapter discusses two articles on visualization. The first article talks about how code can be visualized. The second article discusses a method which visualizes large database structures. The chapter also briefly discusses two chapters in a book which talks about how visualization can help in software development. Finally, the chapter references a few technical reports which talk about the use of visualization in the design and understanding of object oriented programs, and about the needs for 3D graphics in computation visualization.

3.1 Software Visualization in the Large

The article by Ball and Eick [2] presents four different code visualization techniques and then shows how these techniques were applied to the authors' tools to visualize very large software systems developed by Bell Laboratories.

The four visualization techniques are: Line Representation, Pixel Representation, File Summary Representation, and Hierachical Representations.

Line Representation

This technique presents views of code which are color coded at different scales.

In a larger scale, each line of code is presented in a readable format, i.e. just as we read the code using a text editor except that lines are colored differently. In a smaller scale, each line of code is reduced to a single row of pixels, but the indentation, length and coloring of the line are preserved. Colour may be used to represent various properties of the source code, such as its age. Recently-written lines can be colored red, old lines can be colored green, and yellow lines represent intermediate age. Sometimes the line indentation is turned off to emphasize the line colors. The smaller scale is used to view an entire file in a rectangle, while the larger scale is used to read a small portion of the file.

Pixel Representation

The technique represents each line by one, or a small number of, pixels which are also color coded. The pixels are ordered from left to right within a row. There are two different ways to order the pixels. The pixels can be ordered to respect their line numbers, e.g. the first pixel on the left represents line 1, the next pixel represents line 2, and so on. The pixels can also be ordered by their colors to provide mapping from statistics, such as code age, to colors. As an example to illustrate the second way of ordering the pixels, let us assume that line 1 is red, line 2 is yellow, line 3 is green; and these 3 lines are represented by 3 pixels in a graphical row, where the red one is on the left, the yellow one is in the middle and the green one is on the right. Now, if line 4 is yellow, line 6 is red, and line 5 is green, then the pixel representing line 6 would be the first pixel on the left, the one for line 4 would be in the middle and the one for line 5 would be on the right of the next graphical row. As in the case of the Line Representation, this technique can represent a large file in a small window on the screen.

File Summary Representation

In this technique, each file is represented by a rectangle, which could be of different heights, depending on the file size. There are four different rectangle heights which represent four different types of file sizes. Each portion in the rectangle can be colored differently to show different characteristics of the codes within the file. For example, a rectangle may contain a small portion colored purple to represent the amount of code added for new functionality, and a large red portion which shows the larger amount of codes for bug fixing.

Hierachical Representations

A large software system may contain several subsystems which are organized into different directories. Each subsystem, in turn, contains modules which correspond to subdirectories within the directory of the subsystem. Each subdirectory then contains files of the corresponding module.

The whole system can be represented by an entire window. The window is partitioned into rectangles of different sizes to correspond to its subsystems of different sizes. Each subsystem is then partitioned vertically to show its internal directories which correspond to modules of the subsystem. Vertical partitions corresponding to modules could be presented as a bar chart to encode additional statistics, such as the amount of new code development. A subsystem can be zoomed to show files within its subdirectories, where each subdirectory is partitioned horizontally to represent files. Each horizontal partition corresponding to a file within the module could also be presented as a bar chart to encode file-level statistics.

The authors then illustrate the application of these techniques to the visualization of codes based on history and static characteristics, and based on execution behavior.

Code version history

Code age can be viewed using both line representation and pixel representation. The newest lines of code can be colored in red, and the oldest in blue, with a rainbow color scale in between. Therefore, a rectangle representing a file which has a lot of red lines or red rows of pixels, as in the case of line and pixel representations respectively, indicates that a lot of new code have been added. The file summary representation can also be used to show views of code age. The red portion in a rectangle corresponding to a file represents new code, and blue portion represents old code.

The file summary representation can also be used to show bug-fixing code where the red portion can represent the amount of bug-fixing code, and the purple portion represents the amount of new feature development code.

The line representation can also be used to visualize fix-on-fix information. A fix-on-fix occurs when a bug is fixed in the code, and the code is subsequently fixed again because the original repair was faulty. Each vertical rectangle, or column, representing a file contains lines which are color coded to show code age, i.e. red lines represent new code, blue for old code. Now, each column is split in two with half lines on the left representing initial bug fixes, and half lines on the right indicating subsequent fixes to the original fixes. Therefore, half lines means that the bug fixes have worked so far, and whole lines indicate fix-on-fix information.

Differences between releases

The line representation can be used to display the differences between the same file of difference releases. The two files are shown pairwise simultaneously. Four colors are used to show the code. Deleted lines are red, green for added lines, changed lines are yellow, and gray for unchanged lines. Large scale is used to show small portion of the code differences in a readable format, and small scale is used to globally summarizes the differences between two files.

Hierachical representations are used to show differences between all files within a subdirectory representing a module in one release, and all files within another subdirectory representing the same module but in a different release. Each small partition representing a file within a subdirectory contains bar charts of different colors to show deleted, added, changed and unchanged code.

Platform specific information

The pixel representation is used to show platform-specific fragments of the code. Pixels in graphical rows within a column, which represents a file, are colored differently to represent lines of code which are platform-specific. For example, lines of code specific to HP Unix is represented by red pixels, green for the Sun, blue for common code. This visualization shows the amount of code required for each platform.

Conditional nesting complexity

The conditional nesting level of a statement is the number of loops or conditionals which surrounds the statement. The pixel representation is used to indicate the nesting complexity, where a different color is for a different number of nesting levels. Red pixels can represent lines which have the most number of loops or conditionals surrounding the lines. Green pixels are for lines with the least nesting levels. A rainbow color scale is for lines with the number of nesting levels in between.

Execution hot spots

Hot spots in a file are lines of code which were executed more frequently during a user test. Execution hot spots are visualized using the line representation. The color of a line encodes the number of times the line was executed. Red can denote high frequency of execution, and blue can denote the opposite. Line indentation is switched off to make the color patterns more visible.

Dynamic program slices

A dynamic program slice is the code that impacts the execution of a code statement or a procedure. The impacted statement is then called the slice point. A dynamic slice could contain many statements and procedures which may reside within one file or may span over several files. The line representation visualizes dynamic program slices. The point of slice at the mouse can be coded red. Yellow lines can represent statements which directly affects the slice point. Green statements can directly impact the yellow ones. And so on. The large scale is used to show the code portion which contains the point of slice. Smaller scales are used to show the entire dynamic program slice which could span over many files.

In this article, the authors demonstrate that the conventional way of displaying source code on the screen is only one of several possibilities. The capabilities of a high-resolution, color monitor can be used in a variety of ways to provide information about source code rapidly and efficiently. By reducing the visual size of a line of source code to a row of pixels or even a single pixel, the authors are using visualization techniques as an abstraction mechanism. By using color, they make relationships in the source code such as age and maintainability, instantly visible.

3.2 Visualizing the Structure of Large Relational Databases

The article by Antis, Eick and Pyrce [1] visualizes relational databases using seven different views. There is an Overview which displays the entire structure of the database. The other six views show detail information on different characteristics of the database. The authors abandon the use of entity-relationship diagrams because they are not suitable for viewing large relational databases. One of the main reasons for this is that the diagrams become cluttered for large databases, making it difficult to extract information.

The seven views which are going to be discussed are: Overview, Association, Specification, Path, Code, Layout, and Domain.

Overview

The Overview displays all the relations of a database in a window. Relations are represented as horizontal bars. There is a number of columns of bars in the Overview, depending on the number of relations the database has. The length of a bar could represent the number of attributes the corresponding relation has. Or the bar length could represent other statistics of the relation such as the size of a relation tuple, or the maximum number of tuples the relation could have.

The color of each bar encodes some categorial statistic associated with the relation such as the relation's access method, how the relation is stored, how the relation is distributed over processors, or who the owner of the relation is. For example, if the bar color represents the relation's access method, then a blue bar could mean that the relation is accessed using the index method, a green bar means the relation is accessed sequentially, and so on.

Association

The association view shows a relation X with all of its attributes, and all the associated relations in the database specification in a window. Relation X in this view corresponds to the bar in the Overview which the user moves the cursor onto it. Associated relations of X are extracted from predefined queries used in application software. The relation X with all its attributes is in the center of the window, and associated relations are placed around it. The position of associated relations indicates the direction and strength of their association with X. Relations with associations to X, i.e. associations containing the complete key of X, are placed on the left of X. These associations are called *access paths* to relation X since they can be used to directly access tuples of X. Relations Ys with associations from X, i.e. X has associations which contain the complete keys of Ys, are on the right of X. Relations with bidirectional associations, i.e. having both associations to and from X, are on the top. Relations with constraint associations with X are on the bottom. Constraints are associations which do not involve any keys, and therefore cannot be used as access path.

The association view can be expanded to show just relation X and one associated relation Y. All the attributes of relations X and Y are displayed side by side in a window. The linked attributes between X and Y are connected by arrows. Linked attributes are attributes contained in both X and Y. There is a number of small vertical bars in the bottom of the expanded association view. Each bar corresponds to a query in the application software which involves both relations X and Y. The red bar denotes the query which is currently displayed in the Specification view.

Specification

The Specification view shows the text of predefined queries in the application software of the database in a scrolled window. It is linked to the expanded association view. If the mouse touches any of the small vertical bars in the bottom of the expanded association view, the specification view will show the text of the corresponding query in the application software.

Path

The Path view shows all possible shortest paths connecting two relations X and Y. The shortness is measured in terms of the number of joins needed to traverse the path. The relations in the path view are represented by small vertical rectangles in a window. The two rectangles representing relations X and Y are placed at the far left and right side of the window. All relations in between which involve in the paths are placed in between. The relations are connected by arrows to represent shortest paths connecting relations X and Y.

The color of the relations is the same as the color of the corresponding bars in the Overview which encodes a categorial statistic associated with the relation such as the access method, etc. Using the path view, the user can quickly discover the most efficient path which can be used to look up tuples of relation Y from X. If the color encodes how the relation is distributed over processors, then relations which have the same color are stored in the same processor. The most efficient path is the one which involves relations which have the smallest number of tuples and which have the same color, i.e. all relations involved in the path are stored in the same processor, or which have the least number of different colors, i.e. all relations involved are stored in very few different processors.

Code

The Code view shows the source code which references a relation X. The code view organizes the code in hierarchies as a tree of subsystems, modules within subsystems, and files within modules. Each application which references relation X has its own tree. For example, two applications A and B reference the relation X, then Figure 3 illustrates an example of how the code view for this would look like:

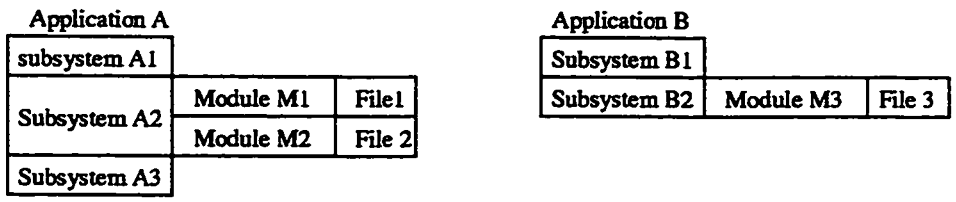


Figure 3: A Code view example

In Figure 3, application A has files 1 and 2 which have code that references relation X. File 1 belongs to module M1, and file 2 belongs to module M2. Both modules M1 and M2 belong to subsystem A2. Application B has file 3 that references relation X. File 3 belongs to module M3 of subsystem B2.

Relation X corresponds to the bar in the Overview on which the user places the cursor. The user can also identify all relations accessed from, say file 2, by clicking on the box which represents file 2 in the code tree. All the relations, which are used by the code in file 2, in the Overview are then highlighted.

Layout

The Layout view shows the physical layout in memory for a tuple of a relation. Each attribute of the relation is represented by a rectangle. The size of the rectangle corresponds to the size of the attribute in the relation. Figure 4 illustrates an example of the layout view.

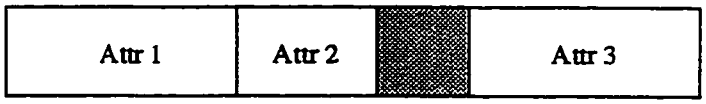


Figure 4: A Layout view example

The gray rectangle in Figure 4 represents the unused portion of the physical data structure for the relation.

Domain

The Domain view is a grid. Each square in the grid represents a domain in the database. The whole grid represents all domains used in the database. A domain is a set of possible values which an existing attribute of a relation can have.

The squares representing the attributes of the selected relation are colored. The purple square represents the domain of the key attribute of the relation. Domains are arranged alphabetically with respect to their names.

All relations which use a particular domain can also be identified. This is accomplished by selecting the square corresponding to the domain of interest in the Domain view. This in turn highlights all relations in the Overview which use the domain.

All the views we have just described may interact, as we have seen. At any point in time, there is one instance of the Overview. But there may be multiple instances of any other views. An object may appear in several different views. The result of manipulating the object in one view will be reflected in all other views.

The second article [1], like the first [2], demonstrates different ways in which visualization techniques can be used to reveal different properties of a complex system, in this case a relational database system. Again, the key factor is that a high-resolution, color monitor has the capability of displaying a large amount of information very rapidly and efficiently, provided that the information is suitably encoded into pixels.

3.3 Visual Programming

There are two chapters in the book by Shu [7] which discusses how programs can be visualized and how visualization can be used in software development.

3.3.1 Visualization of Programs and Execution

Chapter 3 of the book by Shu [7, pages 40–64] presents various ways of visualizing programs and their execution states.

A program can be displayed using 'pretty-printing' method to make it more readable by using different levels of identation, well-placed commenting, and blank-line insertion. Today with high-resolution bit map displays, the 'pretty-printing' method is enhanced to display program source text using multiple fonts, variable point sizes, variable character widths, proportionate character spacing, nonalphanumeric symbols, gray scale tints, rules, and arbitrary spatial location of elements on a page.

Another way of visualizing source code is to graphically represent programs using flow charts and diagrams.

Multiple views of a program and its execution states can also be displayed simultaneously in a window. For example, one view could display program source code, another view could show the graph of the execution flow of the corresponding program source code.

Program algorithms can also be represented through an animated sequence of images. *Interesting events* play an important role in algorithm animation. Interesting events are events which lead to changes in the image being displayed.

3.3.2 Visualization of Software Design

Chapter 4 of the book [7, pages 67–78] discusses two different approaches of how visualization can be used to help in software development.

In the first approach, the users are provided with a set of basic visualization tools. They can use these tools to create various diagrams at their own choice during a software development cycle. There are different categories of diagrams the users can create, such as system requirements diagrams, program function diagrams, program structure diagrams, diagrams of flow control, etc.

The second approach introduces formality into graphical representations of programs. A program design is represented as a hierarchy of interrelated formal dependency diagrams. Diagrams are composed of icons which denote predefined or user defined concepts about program dependencies. Predefined primitives denote objects such as modules, processes. They also denote data dependencies, which involve the declaration, manipulation and sharing of data objects, and control dependencies which deal with asynchronous and synchronous activities. The users can define new concepts using the primitives. Syntactic and semantic constraints are placed on the construction of pictures. When a program is written, it then can be verified whether it meets its pictorial documentation.

3.4 Using Visualization to Foster Object-Oriented Program Understanding

The article by Jerding and Stasko [5] shows a way to visualize object oriented programs and their executions. It then provides a framework to do that.

Classes in an object oriented program are represented as upside-down triangles. Arrows from the bottom of a class to the top of another class represent inheritance. The class hierarchy is represented as a tree structure of upside-down triangles. Class instances are represented by ovals, and message passing is shown by animating the drawing of arrows between instances. Global functions are represented by rectangles. The behaviour of an object oriented program is visualized by graphically animating message passing between instances that occurs during execution.

The article then introduces a framework for visualizing object oriented software. The framework contains two basic components, namely *Event Generation* and *Event Visualization*. The event generation component is responsible for generating program events such as method calls and object creations. The event visualization component manages events as they are generated, and creates graphical, animated views of program execution based on the collected events and using the above mentioned way of visualizing object oriented programs.

3.5 Three-Dimensional Computation Visualization

The article by Stasko [9] identifies the needs for 3D graphics in computation visualization, and introduces a toolkit that supports development of 3D computation visualizations.

In a 3D computation visualization, each dimension can be used to encode certain information. For example, one dimension can be used to encode position, another dimension can be used to encode value, and the third dimension can be used to encode some other information such as history.

There are computation visualizations whose information display requires only two spatial dimensions, such as position and value, but a third dimension is added for presentation purposes. This is because people can process information from three dimensional displays more quickly and as accurately as from 2D displays [8]. Some other types of computations minimally require two spatial dimensions. However, the third dimension is used to encode some other attribute of the computation. For example, in a bar chart sorting view, one dimension encodes the position, another dimension encodes the value, and the third dimension can be used to show a history of the pairs of elements exchanged at each step of the program. Finally, there are computations which involve inherent 3D entities such as a visualization of a cube parallel architecture.

The article then introduces a toolkit called Polka-3D which supports 3D computation visualization development. Polka-3D is an object oriented animation methodology. In the model imposed by Polka-3D, an entire animation consists of different views. Each view is made up of 3D graphical objects and their motions.

An entire computation visualization is encapsulated in an Animator class. The Animator base class defines methods that receive and handle events from the system being visualized. Animation designers subclass Animator, adding individual views as data members and providing a mapping to specify which views are called in response to program events.

There are 3D classes, such as line, sphere, etc, supplied by the toolkit for animation developers to draw 3D graphical objects which make up views. The class Actions can also be used to manipulate the motions of 3D objects.

3.6 Using Animation to Design, Document and Trace Object-Oriented Systems

The article by Shilling and Stasko [6] introduces a visual design tool which helps object oriented software designers develop and capture object oriented designs.

The tool allows software developers to visually specify both static structure of a program and the run-time dynamics and protocols through menus and direct manipulation operations. A programmer uses pull-down menus to select program design and specification commands such as 'Class create', 'Instance create', 'Function invoke', etc, during development.

Design sessions are stored in script files and replayed later to animate program execution.

In displaying a program execution, the tool does not concurrently present the relationship between all program entities since this can quickly lead to an information overload. Instead, the tool utilizes the concept of a *current focus* to drive the display. A program entity in the display becomes the current focus when the entity's representation is selected. When describing a protocol, the object last receiving a message becomes the focus. Once an entity becomes the focus, all other objects update their view to reflect their relationship to the focus entity. If an entity has no relationship to the focus entity, its view is a simple black outline of its shape surrounding its name. The tool also uses color to represent the different class hierarchies within a program.

Chapter 4

Design

4.1 Requirements

A large software application may contain hundreds of modules. One of the very important tasks a software engineer has to do is to understand the uses relationship between these modules. Our task is to build a tool to help the engineer perform this task quickly and efficiently.

The overall goal of the 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 new to a large software system understand the system quickly by looking at the system visually.

System Visualizer will display a module as a sphere and a uses relationship between two modules as a line between two spheres. There are two problems to solve:

- The tool needs to be able to determine the relationships between all modules of a system.
- How should we graphically display these modules and their relationships as spheres and lines in three dimensional space?

The first problem is fairly easy to solve. In a typical software system developed using the C programming language, there is a main module which is a .c file which contains the main() function. This main module normally has an associated .h header file.

As a matter of fact, each module of a system has an associated header file which is included in the .c file of the module. When a module uses another module of the system, it includes the header file of that module in its header file. Therefore, starting from the main module, we can infer the uses relationships between each module of the system.

The second problem is more complicated. However, we can see the natural tree-like structure of the relationships between modules of a system. For example, if module A uses modules B and C. Module B in turn uses modules D, E and F. Then 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. Therefore we can display modules and their relationships as a tree where each module is represented as a sphere and the relationship between two modules is an edge between the spheres. In three dimensional space, the children of a module M are placed on the surface of a cone with its vertex at M, instead of on a plane. We use the term *cone tree* to refer to this kind of tree in three dimensions.

In a system where the uses relationships between the main module and some of the leaf modules, i.e. modules which do not use any other module of the system, is very deep, the cone tree will contain many levels. As a result, it may not be possible to draw the whole cone tree without confusing the picture, given the limited space of a computer terminal. Therefore, the tool by default will display the cone tree starting from the root down to a maximum of 3 levels. To make the tool more flexible, the user can select to display the cone to have any levels of 1, 2 or 3.

Since the tool displays a cone tree of at most 3 levels, part of the system may not be seen. Therefore, the tool has to provide the user with the capability to select any non-leaf modules and display the cone tree starting from the selected module.

The cone tree normally will have non-leaf nodes and leaf nodes. To help the user distinguish these two different types of nodes visually, we use different colours for leaf nodes and non-leaf nodes.

It often happens that two modules A and B use the same module C in a system.

This is because the system structure may actually be a directed graph rather than a tree. In this case, if C is a non-leaf node then the subtree with root at C may appear twice in the cone tree. In order to avoid this, we only draw the subtree rooted at C once and represent C in both places as red spheres. Moreover, if A is at level 1, and B is at level 0, we do not want to draw the subtree C starting from A, since this will make the cone deeper unnecessarily. Instead, we will display the subtree C starting at B.

If the cone tree has a lot of nodes, the picture could become very confusing even though we display the cone at at most 3 levels. This is because the spheres will normally overlap each other. In three dimensional space, we can look at the cone tree from different positions, resulting in the cone having different shapes. The spheres may overlap each other when we view the cone in a straight-on position. But if we move our eyes a little bit up, a little bit to the right or a little bit to the left, then the spheres may not overlap at all, resulting in a clear picture. Utilizing this important characteristic of three dimensional space, the tool allows the user a powerful capability to move their eyes around, i.e. a little bit up, a little bit down, a little bit to the right or a little bit to the left, resulting in the cone tree being rotated, until the cone can be seen best from a good position. In order to help the user speed up the process of locating the best position for viewing the model, the tool allows the user to select one of the 6 following views:

- Looking straight-on.
- Looking from right.
- Looking from left.
- Looking from straight up.
- Looking from right up.
- Looking from left up.

With this, the user can quickly move to a good position by selecting the best view from these, and from here the user can move the eyes around a lot faster to locate the best position for viewing the model. By default, the tool uses the first view to display the model.

In three dimensional space, if an object is closer to our eyes then the object appears larger. System Visualizer also makes good use of this characteristic to display the spheres closer to our eyes larger, and display the spheres farther to our eyes smaller. This increases the clarity of the 3D picture and helps the user interpret the image.

4.2 User Interface

System Visualizer is started up by running the tool executable with the name of the root module as the first parameter and the directory where all the header files of the system under study are located as the second parameter.

The tool then parses the header files, infers module relationships and displays the cone tree representing modules and their relationships of the system in a separate window. Let us call this window the drawing window. There is a menu bar on top of the drawing window.

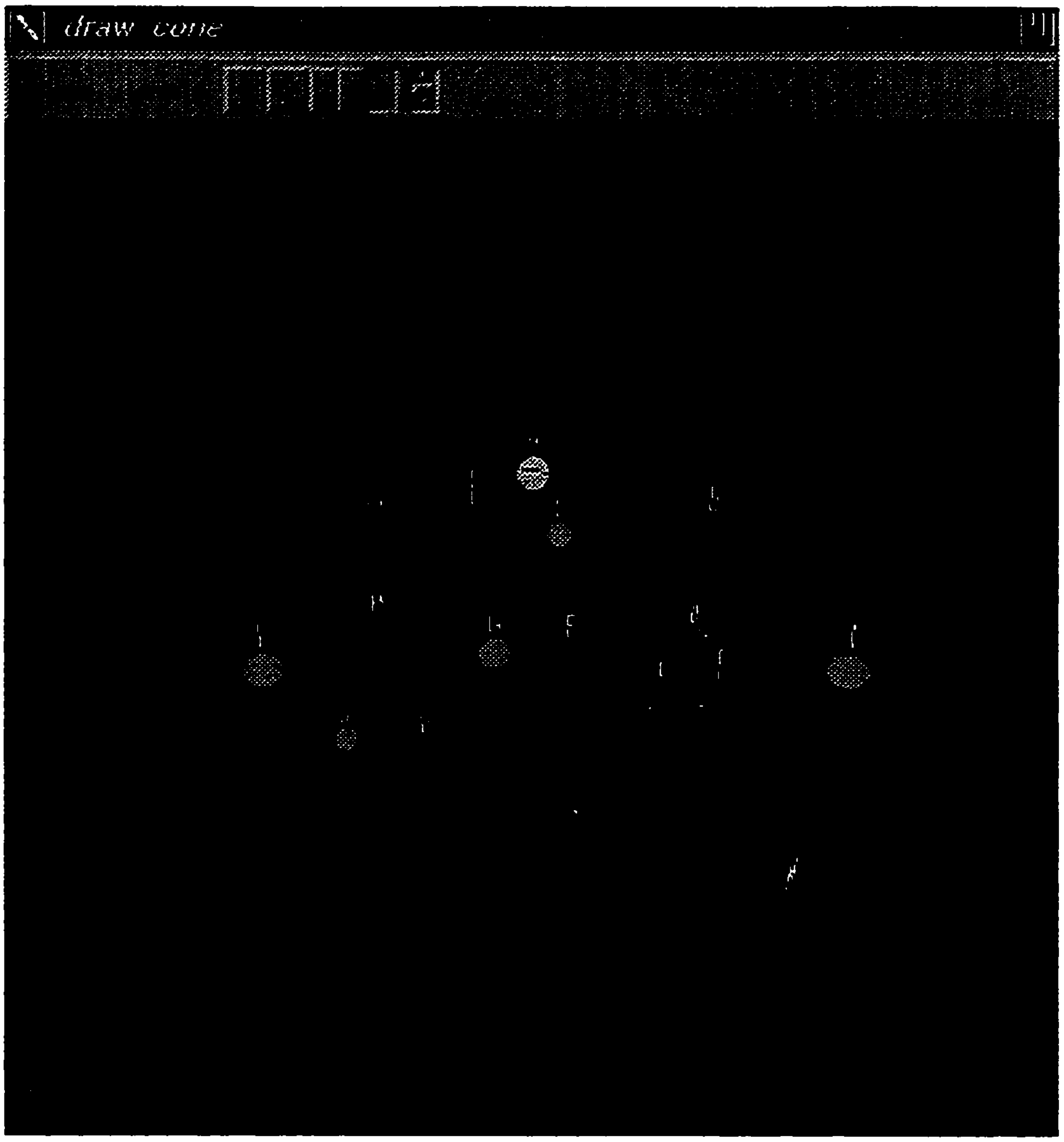


Figure 5: An illustration of the user interface

Figure 5 is an image displayed by the tool. Please refer to Figure 5 in the following description of the user interface portion of the tool for clarification.

The first button on the menu bar is a scrolled-down menu which is labelled 'Views'. Selecting this button results in a scrolled-down menu which has 6 items for 6 different views. When a view is selected, the tool clears the existing picture and redisplays the model using the selected view.

The second button on the menu bar is another scrolled-down menu which is labelled 'Levels'. This scrolled-down menu has 3 items. One item is labelled '1 level' and if selected will cause the tool to re-display the cone tree of one level. The other two are labelled '2 level' and '3 level' and are used to display the cone tree of 2 levels and 3 levels respectively.

The next four buttons on the menu bar are right, left, up and down arrow buttons. Clicking on the right arrow button means that we want the picture to be shifted a little bit to the right. In order to accomplish this, our eyes have to move a little bit to the left. Clicking on the left arrow button causes the reverse action to be performed. Clicking on the up arrow button means that we want the picture to be shifted a little bit up, which means that our eyes need to move a little bit down. The down arrow button performs the reverse action of the up arrow button

The last item on the menu bar is a scrolled list. Each item in the scrolled list is the name of a non-leaf module. If an item is selected, System Visualizer redisplays the cone tree which has the module in the item as the root.

When System Visualizer is started up, it creates the drawing window of a reasonable size. However, the user can move the mouse cursor to the top edge of the window to select a scrolled-down menu, in which there is an option to resize the drawing window to full screen. This also increases the size of the cone tree and improves the clarity of the visualization.

To quit System Visualizer, we can move the mouse cursor to the top edge of the drawing window to select the scrolled-down menu, and then select the 'Done' item. This causes the drawing window to disappear and the tool to terminate.

4.3 Design

There are two major components in the design of System Visualizer: a parser which reads the header files of the software system under study then infers modules relationships and a display module which uses the information obtained by the parser to construct modules and their relationships into graphical form.

The parser when extracting module relationships has to store this information into memory, as a collection of data structures, and then pass the information to the display module. The display module will then read the information from the data structures to build the required graphics. So these data structures are also an important part in the design of the tool.

4.3.1 Data Structures

Since a module may use a list of other modules, the data structure must contain the module name and a pointer to a list of other modules. Consequently, the data structure shown in Figure 6 is suitable:

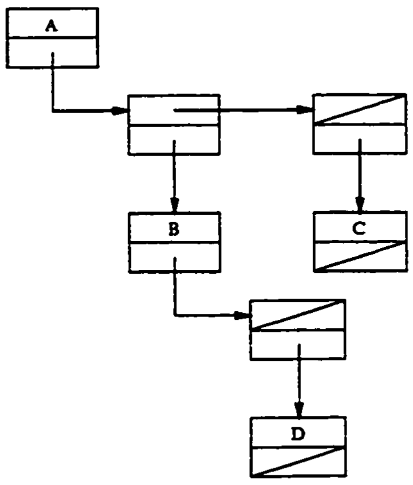


Figure 6: Data structure representing a collection of modules

This data structure has some disadvantages for what we want to do. First, as we can see that the data structure represents a tree structure, and could be very deep depending on the depth of the tree. Also, the data structure imposes the shape of the tree, but the display module does not necessarily want to draw the cone tree in this shape. Moreover, it is a more difficult task and requires more time to traverse the data structure to construct the list of non-leaf nodes.

Figure 7 shows the data structure that we have used in the design since it removes the disadvantages imposed by the previous data structure

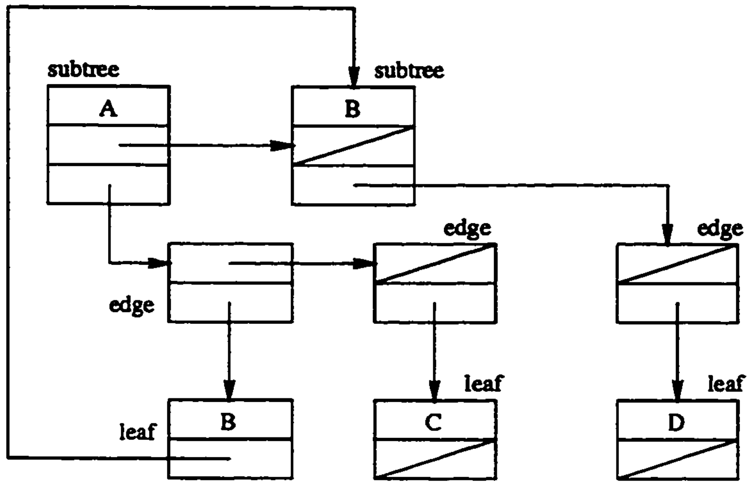


Figure 7: Improved data structure

The data structure is a flat linked list where the top two nodes A and B represent non-leaf nodes which have a subtree underneath. It is needless to say that it is very easy and efficient to traverse this data structure.

4.3.2 The Parser

Having described the design of the data structure passed between the parser and the display module, let us now consider the design of the first component of the tool, namely the parser. In the design described below, please refer to Figure 7 which illustrates the selected data structures for clarity.

Given the name of the main module A, the parser will have to read the header file of A to extract modules which A uses. The parser then has to construct a subtree where module A itself is the root and the modules A uses are the leaves using the described data structure. Basically, the parser will first allocate a subtree node which has the main module as the node name. Then the parser creates a linked list of edges in which each edge points to a leaf node which represents a module used by the root. The parser then creates the linked list of subtrees which contains only one element which is the subtree the parser just created.

For each module used by the root which is a non-leaf node, the parser will construct a subtree for the module by reading the module's header file to extract modules used by this node. This task is exactly the same as just described above. After having constructed the subtree, the parser then needs to append the subtree to the end of the subtree list.

Now we start to see the recursive nature of the parser. Starting from the root, for each non-leaf nodes the parser repeats the following tasks:

1. If the non-leaf node is already in the linked list of subtrees then nothing needs to be done. Otherwise perform the tasks below.
2. Construct a subtree which has the non-leaf node as the root and the modules it uses as the leaves by reading the header file of the non-leaf node to extract those modules.
3. Append the subtree just created to the linked list of subtrees.
4. If a module in a leaf node of the subtree is the root of another subtree in the linked list of subtrees, then make this leaf node points to the other subtree in the linked list of subtrees. Please refer to the leaf node B of subtree A in Figure 7 which illustrates the data structures for an example. In this case leaf node B has a subtree B in the linked list of subtrees.
5. For each module used by the root of this subtree which is a non-leaf node, repeat the above steps.

4.3.3 The Display Module

The display module was the most difficult part of System Visualizer to design. First of all, we need to be able to draw a subtree with a root sphere and a number of leaf spheres.

Consider the XYZ coordinate in 3D space which has the X, Y and Z axis and the origin at (0,0,0). First, we can place the root sphere at the origin, i.e. the root sphere will have its center at the origin. Then we can try to place the leaf spheres around and below the root sphere. This could be a difficult task, especially when the number of leaf spheres varies.

Consider a simple example, where a subtree, or subcone, has module A as root and modules B and C as leaves. We can draw this simple cone as illustrated in Figure 8.

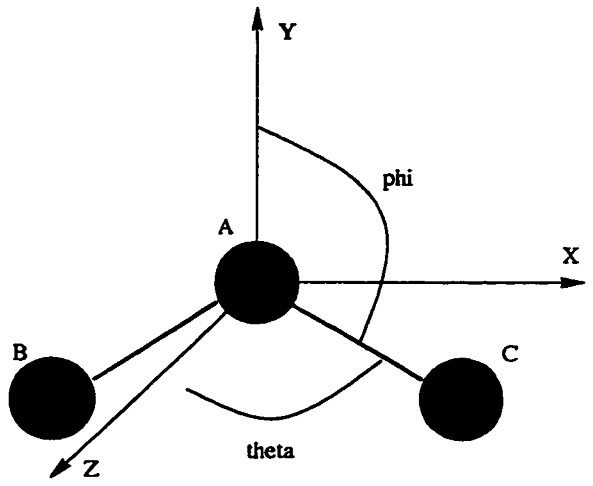


Figure 8: Visualization of a simple tree

The location of the center of sphere C in the XYZ coordinate is computed as:

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \cos\phi \\ z &= r \cos\theta \; sin\phi \end{aligned}$$

where r is the distance between the center of sphere C and the origin. As we can see from the figure, ϕ is a fixed angle which is greater than 90 degrees whereas θ varies for spheres B and C. In the example, θ for sphere C is 90 degrees and for sphere B is 270 degrees. In order to draw spheres B and C at their correct location, we can first draw them at the origin, then compute r, θ and ϕ for B and C, and then move the sphere from the origin to their new XYZ coordinate using the above formulas.

From the analysis of this simple example, we now can draw a subcone with a root and any number of leaves, N. The only thing we have to worry is how to compute θ for each leaf. θ for a leaf i can be computed using the following formula:

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

where i is from 0 to $N-1$. If constant angle is 0 then θ for the first sphere is going to be 0 which means that the sphere is located right on the YZ plane.

Let us now see how we can draw lines between spheres. One way to do this is that we would draw a line connecting the centers of the two spheres, and the line only starts from the surface of one sphere and ends on the surface of the other sphere. But this approach requires a lot of calculations, since we need to compute the two endpoints of the line. The problem becomes worse when we need to do this for a cone with several levels.

Instead, we will use the following approach which is very similar to the one we use for drawing the spheres. Let the distance between two spheres be L. This distance is a constant for any two spheres. First we draw a line along the Y axis starting at the origin which has the length of L. We then move the line up a distance R where R is the radius of the sphere. Now, we rotate the line down ϕ angle. Finally, we rotate the line to the right θ angle. ϕ and θ are the units used to locate the leaf sphere as described above. Figure 9 illustrates this method.

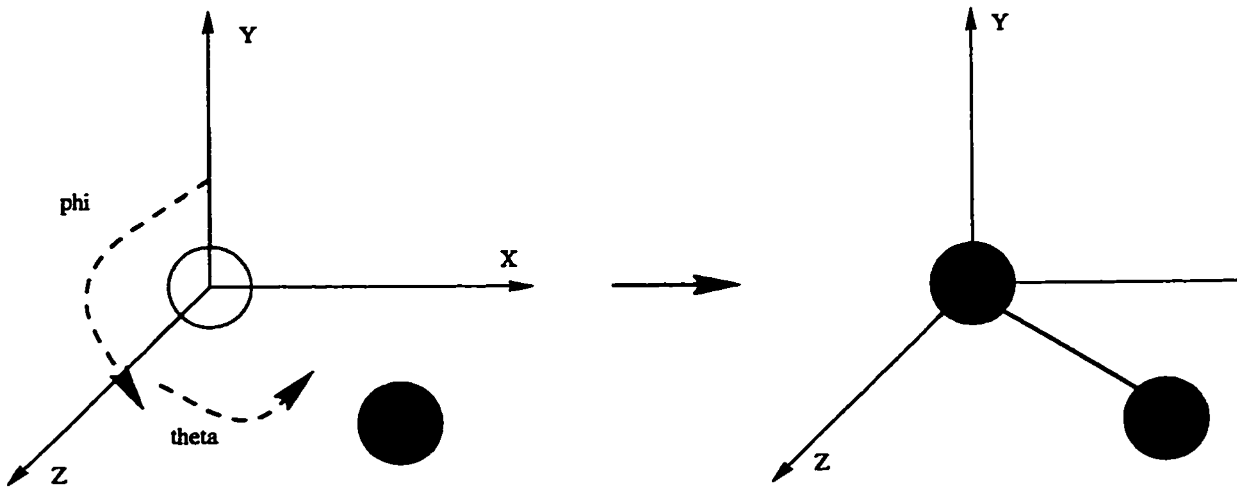


Figure 9: Computing the line between two spheres

Now, we are ready to construct the design for the Display Module. The modul relationships information is stored in the form of our data structures and passed int the Display Module. The Module first needs to extract the subcone information fron the data structure, i.e. what are the modules used by the main module. With thi information, it can compute the locations of each sphere in the subcone using th above described method. Therefore it can draw each sphere and the lines connectin them to construct the subcone.

Now, for each module X represented by a leaf sphere of the subcone which use some other modules, the Display Module has to construct a subcone for it using th above method. But then this subcone has the root located at the origin of the XY coordinate which is not the location the subcone is supposed to locate. However, al we have to do is to move the whole subcone to its proper position, i.e. its root spher will be located at sphere X.

The task is going to be repeated until there is no more non-leaf nodes or until th maximum number of levels the cone supposed to be drawn is reached. The design c the Display Module can be summarized in the following steps:

1. Start from the main module as the current node.
2. Initialize the current level to 0.

3. If the current level is greater than the maximum number of levels then stop. Otherwise do the following.
4. Extract the subcone information from the data structure, starting from the current node as the root.
5. Compute the locations of all spheres for the subcone which is rooted at the current node using the above extracted information.
6. Using the computed locations, construct the subcone to be drawn with root located at the origin.
7. Move the subcone to its proper location.
8. Increment the current level by 1.
9. For each non-leaf nodes of the leaves of this subcone, repeat the above steps, starting back from step 2.

As we have specified in the requirements that, by default the maximum number of levels is 3. However, the user can select to have the cone tree drawn at any levels between 1 and 3. If this is the case, all we have to do is to reset this maximum number of levels, and execute the above steps to draw the cone tree with the desired number of levels.

From the Data Structure, we can extract very easily and efficiently the list of non-leaf modules and place it in the scrolled list of non-leaf modules as specified in the User Interface section. Now, if the user select a non-leaf module from this list, the Display Module needs to be able to redraw the cone tree which is rooted at this module. Looking back at the algorithm we just developed, we see that all we have to do is to replace the first step as:

1. Start from the selected module as the current node.

By default the selected module is the main module.

Now, we need to paint different colors for spheres representing different types of modules. The problem is to determine the type of a module. For leaf modules, it is very easy to determine. Looking back at our data structures, if the pointer of the leaf data structure is nil, i.e. it does not point to any subtree, then this is a leaf node, and will be painted blue. It is more difficult to determine whether a non-leaf node appears only one time or more in the cone to be drawn. If we try to count the number of times a module appears in the cone while drawing, then it may be too late. For example, as we draw module A the first time, the count will be 1, therefore it will be painted yellow. However, the module may be drawn again and only subsequent drawings of the module will paint it red since only subsequent drawings will have the count greater than 0. To overcome this problem, we need to traverse the cone first to compute the count, and then we can start drawing using the count to determine what color should be used.

As specified by the requirements, we do not want to draw the same subtree more than once in the cone. Moreover, if a subtree is supposed to appear twice in the cone, one time with root at level 1, for example, another time with root at level 2, then we will draw the subtree with root at the smaller level. Because of the recursive nature of our algorithm, if care is not taken, we may traverse to the subtree with root at level 2 first, and as a result, we may draw the subtree with root at level 2 instead. Therefore, before we start to draw, we need to compute the smallest level a subtree could be rooted in the cone. Then, during drawing, if we encounter a subtree at a level which is greater than its smallest level, then we do not want to draw it yet. We will draw the subtree only at the smallest level it could be rooted. There is another potential problem. A subtree may appear more than once in the cone tree at its smallest level. To overcome this problem, once a subtree has been drawn, we will mark it as drawn. So the next time during drawing, if we encounter the subtree at its smallest level again, we know that it has been drawn.

By default, the cone tree is drawn as if our eyes look at it in the direction of point X (0, 0, 1) towards the origin in the XYZ coordinate, where the XY plane is the plane of the screen and the Z axis comes towards the viewer. Since looking at the cone in this direction may cause the spheres to overlap each other, making the cone difficult to view, we need to be able to draw it as if we were looking at it in different directions. As a matter of fact, the spheres may overlap when it is looked at in one direction, and may not overlap when it is looked at in another direction. The direction which cause the spheres to overlap or not to overlap is more or less random. This is why we need to be able to look at the spheres from different directions. Most 3D library packages provide the capability to draw a model as the eyes look at it from different directions. As the user interface of System Visualizer provides 4 different arrow buttons to do this, if the user presses on the right arrow button, then we decrement the X coordinate of point X by a small constant, say 0.1, and then redraw the cone using the new direction. For left arrow button, we do the reverse, i.e. increment X coordinate of point X by 0.1. For up arrow button, we subtract the Y coordinate of point X by 0.1, and we do the reverse for down arrow button.

Figure 10 shows the directions which can be used to draw the cone tree for the 6 different views as specified in the requirements:

Views	*XYZ coordinate of point X*
straight	(0, 0, 1)
right	(1, 0, 1)
left	(-1, 0, 1)
straight up	(0, 1, 1)
right up	(1, 1, 1)
left up	(-1, 1, 1)

Figure 10: Six views

So far, we have only talked about looking at the cone tree from different directions. If the eyes looking at the cone are positioned far away from it, then the sizes of all spheres will appear to be the same. However, if the eyes are positioned close to the model, then spheres closer to the eyes will be larger and spheres farther from the eyes will be smaller. Most 3D graphics libraries allow us to do this.

4.4 Design Decisions

System Visualizer has been developed to analyze software applications written using the C programming language to infer module relationships of the application and transforms this information into three dimensional graphics form. The tool was also written in the C language and is supposed to run under the Unix operating system which supports X windows. The tool assumes that an X server is running on the terminal in which the tool is executed.

These choices were made because many software applications have been developed in C to run on the Unix operating system. Also, C and Unix are standard language and platform used in many Universities. Moreover, X windows are installed in most Unix machines.

System Visualizer uses the OSF Motif toolkit for its User Interface portion since Motif is also available on most Unix machine and is compatible with X windows.

The Display Module of System Visualizer utilizes the X library PEXlib to draw the three dimensional graphics portion. PEXlib is a 3D extension of the Xlib which is a very powerful 3D library. It can provide all the graphic facilities the tool requires. Also, we can incorporate Motif into programs which use PEXlib.

Since the 3D extension of X servers run mainly on color terminals, System Visualizer also assumes to run on color terminals which support PEX servers.

Chapter 5

Implementation

5.1 Steps in implementation

As we can see from the Design chapter, there are two major components to implement System Visualizer, namely the Parser and the Display Module. But, first let's discuss the detailed implementation of the Data Structures which are a major component required to integrate the parser with the drawing code.

5.1.1 Data Structures

There are three major data structures for System Visualizer: subtree, edge and leaf. The subtree data structure represents the root node of a subtree. Each subtree node is an element of the linked list of subtrees which is a linked list of non-leaf modules, i.e. modules which use other modules. The first field a subtree needs to have is the name of the root module of the subtree. It also contains a pointer *NextSubtree* to the next subtree in the subtree linked list, and a pointer *Uses* to a linked list of edges. These two pointers will be discussed shortly.

Also as specified in the Design, a non-leaf module may appear more than once in the cone tree to be drawn, and as such the sphere will be colored red and we need to store the count of the number of times the non-leaf module is to appear in the cone. Let's name this counter the *AppearCount*. If we create a separate data structure to store *AppearCount* then we would need a pointer between this new data structure and the subtree data structure. Therefore the best place is to include *AppearCount* as another field in the subtree data structure.

If *AppearCount* of a subtree is greater than 1, i.e. the subtree may appear more than once in the cone, then we only want to draw it once. Moreover, we want to draw the instance of the subtree which has root at the smallest level. Therefore we need to compute and store this information somewhere. Let's call this *SmallestRootedLevel*. Also, storing *SmallestRootedLevel* alone is not sufficient since a subtree may be rooted at its *SmallestRootedLevel* more than once, and therefore we may draw the subtree a few times at this level. Therefore we also need to mark a subtree as having been drawn once it is drawn the first time, so that subsequent occurrences of the subtree at this level will not be drawn again. We will call this information *Drawn*. Both *SmallestRootedLevel* and *Drawn* are placed inside the subtree data structure for the same reason as discussed above.

Another field is needed to store in the subtree data structure. This field is named *Traversed*, and is used to compute *AppearCount* of the subtree. We explain its function on page 61.

To summarize, a subtree contains the following fields: *NodeName*, *NextSubtree*, *Uses*, *AppearCount*, *SmallestRootedLevel*, *Traversed*, and *Drawn*. Note that the fields *AppearCount* and *SmallestRootedLevel* need to be computed before drawing so that the Display Module can use their values to determine what to draw.

The linked list of edges represents the uses relationships between modules of a subtree. Each element in the linked list points to a leaf node which represents a module used by the root node of the subtree. The edge data structure contains a pointer *NextEdge* which points to the next element of the linked list, and a pointer *Leaf* pointing to a leaf node of the subtree.

The leaf data structure needs to contain the *NodeName* which is the name of the module used by the root of the subtree. It also contains a pointer to the subtree node where the module itself is the root of this subtree. In case the module is a leaf module, i.e. it does not use any other module, then this pointer will have the value

Nil. Let's name this pointer *RootedAt*.

5.1.2 The Parser

The Parser takes the name of the main module of the application software, reads its header file and then recursively reads header files of modules which the main module uses to extract module relationships. The parser then stores the extracted module relationships information using the data structures described above. The information will then be passed to the Display Module for drawing.

The main function of the parser is *BuildModuleRelationshipInfo* which takes a module name A, which is the name of the main module of the application software when originally called from the main program, and performs the following tasks:

1. Calls the function *IsInSubtreeList* to check if the subtree rooted at module A has been built. If it has, then *BuildModuleRelationshipInfo* just returns.
2. Construct the subtree rooted at module A by invoking function *BuildSubtree*.
3. Append the subtree of A just created to the linked list of subtrees by calling function *AppendToSubtreeList*, if A is a non-leaf module.
4. For each edge E in the linked list of edges pointed to by *Uses* of the subtree of A, do the following in this order:
   (a) invoke the function itself, which is *BuildModuleRelationshipInfo*, passing in module name B stored in the leaf node pointed to by *Leaf* of edge E to extract module relationships and build the information for the sub system rooted at module B.
   (b) set the pointer *RootedAt* of the leaf node B of the subtree of A to point to the subtree rooted at module B.

The implementation of function *IsInSubtreeList* is straightforward. Starting from the first element, the function traverses the linked list of subtrees to find a match if one exists, and returns the result.

The function *BuildSubtree* is a little bit more complicated. The function takes the name of a module, and opens the header file H of the module for reading. It then reads every line in the header file, parses the line looking for a header file which is included in header file H. If the module is a non-leaf module, the function allocates a subtree node for it, and for every header file included in H, it allocates an edge for it to be part of the linked list of edges and the corresponding *Leaf* node. If the module is a leaf module then nothing is created. However, the function still has to read and parse the header file H to determine whether the module uses any other modules.

The function *AppendToSubtreeList* builds the linked list of subtrees, i.e. the linked list of non-leaf modules, by appending the subtree being passed in to the function. At first, if the linked list is empty then the function creates it with the first element to be the subtree passed in.

5.1.3 The Display Module

The Display Module takes in module relationships information inferred and built by the parser, uses this information to construct and draw the corresponding cone tree.

The decision to use PEX to display structure was made at an early stage in the design of System Visualizer. PEX is a 3D extension of the X window system. The implementation of the display part is tightly coupled with the 3D library package being used, which is PEXlib in our case. Therefore in describing the implementation, we need to discuss important characteristics of PEX which are used by System Visualizer.

In PEXlib, there are two modes of drawing. The first mode is the immediate-mode. In this mode, we specify the picture to be drawn and then draw the picture. After being drawn, since the information of how to draw the picture is not stored anywhere, the information is lost. Therefore every time we want to redraw the picture, we need to respecify it, and this will increase real time execution. This mode is clearly not suitable for our application. System Visualizer needs to keep changing the way the picture looks depending on the user's needs and needs to redraw the picture many times.

Another mode of drawing in PEX is to use PEX Structures. The application can specify how it wants to draw the picture, and store this information in PEX structure. Later on, the application can ask PEX to draw the picture using the information specified in the PEX structure. The application then can change some characteristics of the picture by just changing some information stored in the structure without having to respecify the whole picture again. The Display Module uses this mode to perform its drawing task.

The main task of the Display Module is therefore to create a PEX structure and build the information of how to draw the cone tree using the module relationships information, and store the information into the PEX structure. *BuildConeStructure* is the main function of the Display Module. It uses the linked list of subtrees, and performs the following tasks:

1. Invoke the PEX function PEXCreateStructure to create a PEX structure.

2. Invoke PEXSetViewIndex to tell PEX to use the view which has been defined earlier in the main() function of System Visualizer during initialization code. This will be discussed in greater detail later.

3. Call function *ComputeSmallestRootedLevel* to compute *SmallestRootedLevel* for all subtrees to be displayed in the cone.

4. Call function *ComputeAppearCount* to compute *AppearCount* for all subtrees to be displayed in the cone.

   Let's recall that we need to compute values for *SmallestRootedLevel* and *AppearCount* prior to drawing, so that the drawing process can use these values to decide how to draw. The functions to compute these values will be discussed shortly after.

5. Call *BuildSubconeStructure* to recursively build all subcones of the cone tree to be drawn and store this information into the PEX structure, starting at the first subcone pointed to by *CurrentConeRoot*, and starting at *CurrentLevel* to be 0.

CurrentConeRoot is a pointer to a subtree in the subtree linked list. This subtree has the root which is also the root of the entire cone tree to be drawn. Originally, when System Visualizer is started up, *CurrentConeRoot* points to the first subtree in the linked list, which is rooted at the main module of the software application.

We also need to specify that we want to move the first subcone to its *CorrectSubconeLocation* as (0,0,0) in the XYZ coordinate.

Let us have a few words to discuss the use of *CorrectSubconeLocation*. Every subcone we draw has the root sphere centered at the origin. This is to make our drawing algorithm simple. However, only the root sphere of the entire cone tree to be visualized is centered at the origin. Therefore, each time *BuildSubconeStructure* is invoked to draw a subcone, we need to tell the function to move the subcone to *CorrectSubconeLocation*. Since the first time *BuildSubconeStructure* is invoked, the function draws the subcone having the root sphere which is also the root of the entire cone tree, *CorrectSubconeLocation* is specified to be the origin.

BuildSubconeStructure is a recursive function and does most of the work for the Display Module. It takes in mainly *CurrentSubconeRoot*, a pointer to a subtree in the subtree linked list. Each time the function executes, it draws a subcone whose root is the root of the subtree pointed to by *CurrentSubconeRoot*. *CurrentSubconeRoot* is initially set to *CurrentConeRoot* when *BuildSubconeStructure* is called from *BuildConeStructure*. This means that the first subcone to be drawn has the root sphere which is also the root of the entire cone tree. The function *BuildSubconeStructure* performs the following tasks:

1. If *SmallestRootedLevel* of subtree A pointed to by *CurrentSubconeRoot* is smaller than *CurrentLevel* then return. This means that during drawing, we encounter a subtree at a level which is greater than its smallest rooted level, therefore we do not want to draw the subtree.

2. If *Drawn* of subtree A is true then return. This means that we have encountered a subtree at its smallest rooted level more than once, and the subtree has been drawn previously.

3. Set *Drawn* of subtree A to be true, since we are going to draw it.
4. Compute θ for each sphere representing a module used by module A which is the root of subtree A. As we noted from the design that r and ϕ are fixed and need not be computed.
5. Draw sphere representing module A at the origin. The color of the sphere is determined by *AppearCount* of subtree A, i.e. if *AppearCount* is greater than 1 then it is red, otherwise it is yellow.
6. Move sphere A to *CorrectSubconeLocation*.
7. For each module X used by module A, do the following:
   (a) Draw a sphere for X at the origin. The color of the sphere depends on whether or not X is a leaf module. If it is a leaf module, it is colored blue. If it is not a leaf, its color is determined by *AppearCount* of its own subtree.
   (b) Using θ, ϕ and r of X, compute its location L in the XYZ coordinate.
   (c) Move the sphere to the new location L + *CorrectSubconeLocation*.
   (d) Draw a line connecting sphere of A and sphere of X using the method described in Section 4.3.3, using ϕ and θ of X. This causes the line to rotate properly.
   (e) Move the line to the new location L + *CorrectSubconeLocation*.
8. If *CurrentLevel* is smaller than *MaxLevelToDraw* then do the following:
   (a) Increase *CurrentLevel* by 1.
   (b) For each module X used by A, if it represents a non-leaf module, i.e. *RootedAt* pointer of *Leaf* X points to a subtree node in the subtree linked list, then set *CurrentSubconeRoot* to *RootedAt*. This is to prepare so that the next time *BuildSubconeStructure* is called, it will draw the subcone rooted at X.
   (c) Using θ, ϕ and r of X, compute its location in the XYZ coordinate and add this to *CorrectSubconeLocation*. This new value of *CorrectSubcone-Location* will be used by *BuildSubconeStructure* to move subcone X to its proper position.

(d) Recursively invoke *BuildSubconeStructure* to draw subcone X.

Note that *MaxLevelToDraw*, as the name suggests, is the maximum number of levels the cone tree is to be drawn. By default it is set to 3.

A few things should be mentioned in the algorithm of *BuildSubconeStructure*. In the function, we construct information of how to draw the cone tree in the PEX structure created by *BuildConeStructure*, and later the structure will be used to draw. In the algorithm, we need to rotate the line connecting the root sphere and its leaf sphere. In PEXlib, we can call function PEXrotate consecutively two times, the first one to rotate it through an angle ϕ downwards, the second one to rotate it through an angle θ rightwards. To move the spheres and lines to their proper position, we can call PEX function PEXSetLocalTransform.

There is one important characteristic in 3D visualization which we need to discuss here. In three dimensional space, when parts of two objects overlap, the part of the object which stays behind is obscured. Consider the following figure:

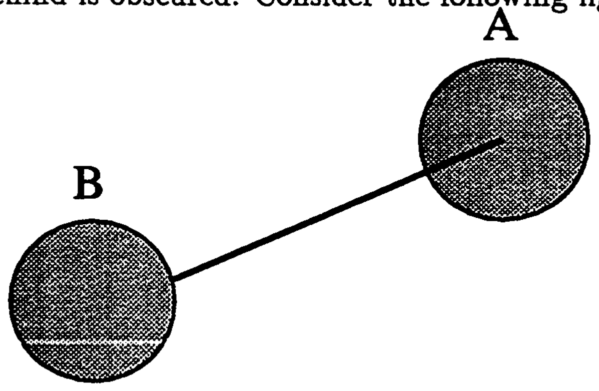


Figure 11: Sphere B in front of sphere A

In Figure 11, sphere B seems to appear in front of sphere A. This is because the line connecting the 2 sphere is obscured by sphere B, meaning that the line is behind sphere B. Also, since the line is not obscured by sphere A, the line seems to appear before sphere A.

In three dimensional space, for the same picture, if the hidden parts are different, then the resulting pictures will appear very different. Let us look at the picture as in in Figure 11, however the hidden portions are different as in the following figure:

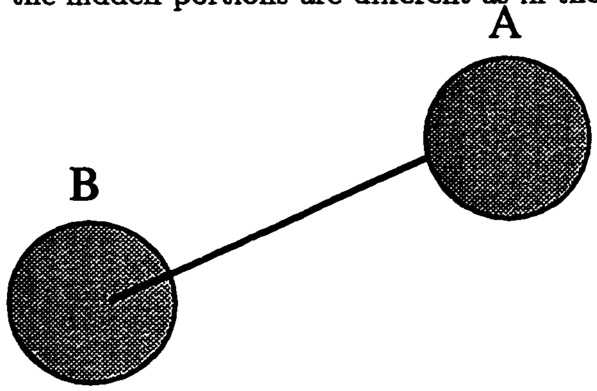


Figure 12: Sphere B is behind sphere A

This time, sphere B seems to appear behind sphere A. This is because the line is obscured by sphere A which results in the line being behind sphere A. And since the line is not obscured by sphere B, the line appears in front of sphere B.

This three dimensional characteristic is called Hidden Line And Hidden Surface Removal, or HLHSR for short, in PEX. As we can see, HLHSR is very important. When a 3D picture is drawn using a wrong HLHSR, this will mislead the user to interpret the picture differently.

HLHSR can be accomplished by drawing the object which stays behind before drawing the object which is in front. The result of this is that the in-front object, since it is drawn after, will obscure the object which is behind. Therefore, the order in which objects are drawn is very important for HLHSR.

When a PEX server supports HLHSR, the order in which an application asks PEX to draw is not important. For example, if object A is in front of object B, even if an application asks PEX to draw A before asking PEX to draw B, PEX will still take care of HLHSR properly since it knows which object is before which object by their locations in the XYZ coordinate. This relieves the application of having to worry about the order in which it must ask PEX server to draw objects in order to have a proper HLHSR 3D image.

Unfortunately, not all PEX servers support HLHSR mode. In particular, the PEX server where System Visualizer was developed does not support HLHSR. Therefore, the tool must take care of the order it asks PEX server to draw objects. From the analysis of a simple subcone, we know that leaf spheres Xs whose θ is between 90 and 270 degrees lie after leaf spheres Ys whose θ is less than 90 or greater than 270 degrees. Therefore leaf spheres Xs need to be drawn before leaf spheres Ys. Also, the lines connecting leaf spheres Xs and the root sphere lie before Xs. Therefore these lines must be drawn after Xs. However, the lines connecting leaf spheres Ys and the root sphere are behind Ys, so they must be drawn before drawing leaf spheres Ys. The root sphere can be the last thing to draw since we look at the cone from top down. The code which actually implements the function *BuildSubconeStructure* rearranges steps 5, 6, and 7 of the algorithm for *BuildSubconeStructure* taking into account the above analysis.

Also, since we look at the cone tree from top down, subcones which lie below will appear after subcones which stay above. Therefore, lower subcones need to be drawn before higher subcones. As we can see from the algorithm of *BuildSubconeStructure*, step 8 prepares and recursively invokes the function itself to draw subcones below the subcone the function is drawing. So the actual code of *BuildSubconeStructure* moves step 8 before step 4 so that the function can draw lower subcones before drawing the higher ones.

The function *ComputeSmallestRootedLevel* takes a pointer to subtree A and traverses subtree A, and recursively traverses all subtrees under subtree A which would appear in the cone tree to be drawn, to compute *SmallestRootedLevel* for each subtree. Originally, when the function is called from *BuildConeStructure*, A is the subtree which is the root of the entire cone tree to be displayed. The function is implemented as follows, starting at *CurrentLevel* of 0, and stopping at *MaxLevelToDraw*:

1. If *SmallestRootedLevel* of A is greater *CurrentLevel* then set it to be *CurrentLevel*. Originally, when subtree A is allocated, *SmallestRootedLevel* is given a large value, e.g. 999.

2. If *CurrentLevel* is less than *MaxLevelToDraw* then for each non-leaf module X used by A, do the following:

(a) Increment *CurrentLevel* by 1.

(b) Recursively invoke *ComputeSmallestRootedLevel*, passing in the pointer to subtree X.

It is fairly easy to compute *SmallestRootedLevel* for a subtree. As the function traverses, if it encounters a subtree with *SmallestRootedLevel* greater than the level it is at, then it resets *SmallestRootedLevel* to the current level. It is very difficult to compute *AppearCount* for a subtree. Let's consider the following example:

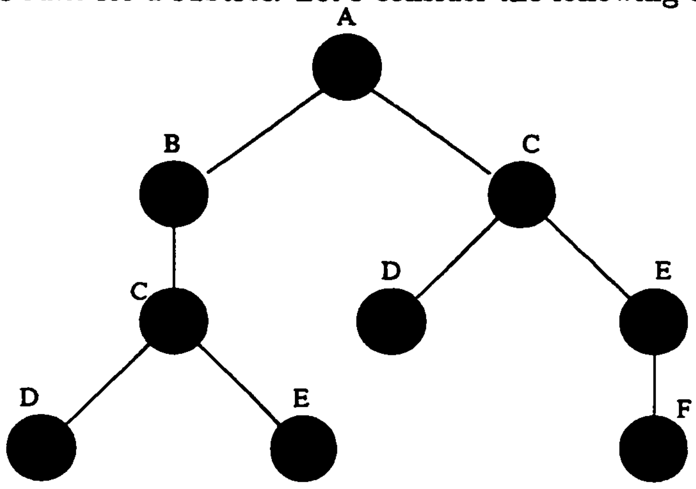


Figure 13: An example of a tree structure with duplicated subtrees

Because of the recursive nature of the algorithm, we will encounter subtree C at level 2 first. Therefore, *AppearCount* of C will be 1, and as we go down to subtree E, *AppearCount* of E will be 1. Now, as we traverse along, we will hit subtree C again at level 1. This time, *AppearCount* of C is incremented to become 2, and as we go down to subtree E, *AppearCount* of E will be 2.

However, the Display Module only draws the subtree C rooted at the smaller level which is at level 1. The cone tree which is drawn by the Display Module will look like this:

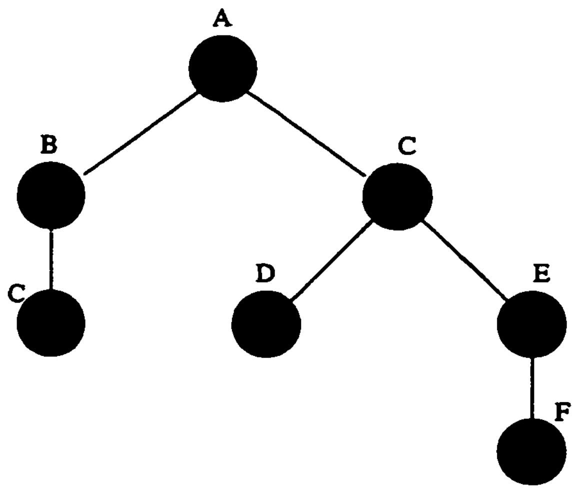


Figure 14: The visualization of the tree structure with duplicated subtrees

Since, the *AppearCount* of E is 2, E will be colored red, which is not correct because E appears only once in the cone tree.

What we should have done is that when we reach subtree C rooted at level 2 the first time, we should still increment *AppearCount* for C, but we should not have traversed down subtree C, since it is not rooted at its *SmallestRootedLevel* yet. Now, when we reach subtree C at level 1 the second time, we will increment *AppearCount* for C, and this time since subtree C is at its *SmallestRootedLevel*, we will traverse down subtree C. When subtree E is reached, *AppearCount* of E is incremented the first time, so it will have a value of 1. Therefore E will not be colored red. One thing which can be noted is that *SmallestRootedLevel* for each subtree must be computed before *AppearCount* can be computed, since the algorithm to compute *AppearCount* uses *SmallestRootedLevel* to decide whether it should traverse down a subtree.

There is another potential problem. A subtree may be rooted at its *SmallestRootedLevel* twice as in Figure 15:

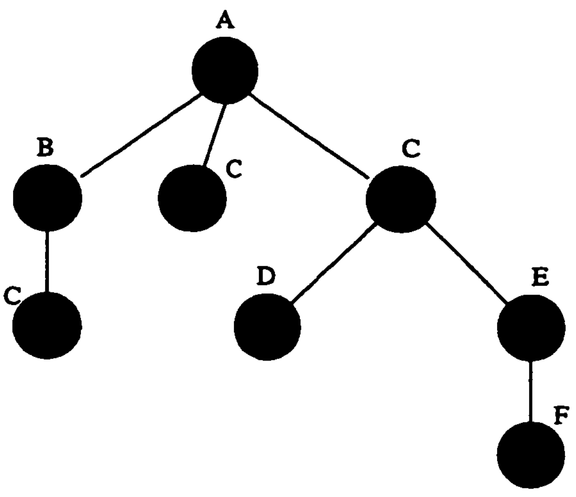


Figure 15: A subtree is rooted at its smallest level twice

If we have traversed subtree C at its *SmallestRootedLevel* once, when we reach subtree C at this level the second time, we should not traverse it again. Therefore we need to set a flag once we have traversed a subtree the first time. This is why *Traversed* was introduced in the data structures we discussed earlier. *Traversed* is initially set to false when a subtree is created.

The function *ComputeAppearCount* takes a pointer to subtree A and traverses subtree A, and recursively traverses all subtrees under subtree A which would appear in the cone tree to be drawn, to compute *AppearCount* for each subtree. Originally, when the function is called from *BuildConeStructure*, A is the subtree which is the root of the entire cone tree to be displayed. The function is implemented as follows, starting at *CurrentLevel* of 0, and stopping at *MaxLevelToDraw*:

1. Increment *AppearCount* of A by 1.
2. If *CurrentLevel* is equal to *MaxLevelToDraw* then stop.
3. If *SmallestRootedLevel* of A is equal to *CurrentLevel*, which means that we are reaching subtree A at its *SmallestRootedLevel*, and *Traversed* of A is false, do the following:
    (a) Set *Traversed* of A to true.
    (b) Increment *CurrentLevel* by 1.

(c) For each non-leaf module X used by A, recursively invoke *ComputeAppearCount*, passing in the pointer to subtree X.

5.1.4 Viewing Transformations

System Visualizer needs to allow the user to look at the cone tree, or the model, in different directions, i.e. draw the model using different *views*. PEXlib provides utilities to an application so that the application can define the *view* it wants to use to draw the picture.

It is easy to explain loosely the factors that go into taking a picture with a camera: stand back far enough to get the whole scene, point the camera at the thing you want in the middle of the picture, hold the camera either vertically or horizontally. However, it is not easy to reduce these factors to mathematically precise values. Precise values are what we need in order to describe the view to PEXlib. In the following paragraphs, we are describing what are the relevant factors needed by the tool to define in describing the view it desires to PEXlib.

In the XYZ coordinate, we need to select a point. Wherever we place the camera, we will aim it directly at this point, which becomes, so to speak, the direction of our gaze. The point is called the *view reference point* in PEXlib. The view reference point is by default located at the origin.

Next, we need to define how we shall tilt, or rotate, the camera around the axis provided by the direction of our gaze. Depending on this rotation, the resulting view will appear 'normal', sideways, upside down, or at some strange angle. What we are defining, in effect, is which direction is up. System Visualizer uses the Y axis to be the view up vector.

The direction of gaze is called the *view plane normal* in PEXlib. It is the view plane normal the tool needs to change in order to allow the user to look at the cone tree in different directions. By default, the view plane normal is defined to be the (0, 0, 1) vector, which is the direction along the Z axis.

Now, we need to define how we want 3D objects to be projected to 2D images. First we need to define the plane onto which the 3D model is projected. This plane is called the *view plane* in PEXlib, and is defined by our tool to be the XY plane.

In PEXlib, there are two kinds of projections: parallel and perspective projections. In the case of a parallel projection, the projecting lines are parallel to the direction of gaze. As a result of this, all 3D objects of the same size will be projected to have the same size onto the view plane. In a perspective projection, the point where we place the camera is important. In PEXlib, this point is called the *projection reference point*. In the case of a perspective projection, the projection reference point is the point from which all 3D objects are projected onto the view plane. If the projection reference point is close enough to the 3D model, then for 3D objects of the same size, the ones which appear closer to the camera will be projected to be larger, and the ones which appear farther to the camera is projected to be smaller. System Visualizer uses the perspective projection and places the projection reference point at (0, 0, 1) to satisfy its requirements.

PEX has several different kinds of table, one of which is the view table. The *view table* has several entries, each entry contains definitions of a view. A view entry is accessed by indexing into the view table, using the proper view index of the entry. View indexes are numbered, starting at 0, going up 1, 2 ... System Visualizer defines the view for its cone tree model, and stores the view into a fixed entry in the view table. The function *BuildConeStructure* calls PEXSetViewIndex to specify that when drawing the picture using the PEX structure, it should use the view whose definition is stored in this view entry.

5.1.5 Integration of the Parser and the Display Module

Now, we will put everything together to construct the implementation of System Visualizer. All this is done in the main() function of our tool.

The first thing we need to do is to setup PEX to create a drawing window among other things. This is a standard step in all PEXlib programs, and we will not discuss it further here.

Now, we initialize *MaxLevelToDraw* to its default value, which is 3. Recall that *MaxLevelToDraw* is used by the function *BuildSubconeStructure* to decide when to stop drawing. Then we invoke the parser by calling *BuildModuleRelationshipInfo* to infer and build module relationships of the application software.

We then utilize Motif to create a menu bar on top of the drawing window. Now we create a pull down menu to sit on the menu bar, which contains six buttons labelled 'Front view', 'Right view', 'Left view', 'Up-Front view', 'Up-Right view', 'Up-Left view', and specify *SetView* to be the callback function for these buttons. *SetView* will be discussed later. As a matter of fact, the discussion of all callback functions which are mentioned here will be deferred a little later.

Again, using Motif we create a pull down menu on the menu bar which contains 3 buttons labelled '1 level', '2 level', '3 level', and assign function *SetLevel* to be the callback function of these buttons.

Now, on the menu bar, we create four arrow buttons: right, left, up and down, and specify *RightArrow*, *LeftArrow*, *UpArrow* and *DownArrow* to be callback functions for these buttons respectively.

Next, we utilize Motif again to create a scrolled list which contains the names of non-leaf modules, and specify *NonLeafModuleListCallback* to be the callback function when a module name in the scrolled list is selected.

We then register an event handler *ExposeHandler* for the expose event. This is mainly needed to draw the cone tree the first time when the drawing window of the tool appears on the screen.

Then we invoke the main function *BuildConeStructure* of the Display Module so that it can construct the cone tree to be drawn and store the information in a PEX structure for drawing.

Finally we define the default view to be used for drawing, with the view plane normal to be (0, 0, 1) and store the view in a fixed entry of the PEX view table, as discussed previously. Recall that step 2 of the algorithm for the function *BuildConeStructure* asks PEX to use this view when drawing the model. We then ready to go to the main loop, and wait for user's input.

Let us now discuss the implementation of each of the callback functions mentioned above.

SetView sets the view plane normal to the appropriate value as specified in the Design chapter depending on the button selected. It then redefines the view using the new value of the view plane normal and redraw the cone tree using the updated view, i.e. redraw the picture using the PEX structure storing the information on how to draw the cone tree, and the structure is supposed to use the view stored in the view entry which has just been updated.

SetLevel sets *MaxLevelToDraw* to the appropriate value depending on the button selected. Now, since the cone structure will change, the function destroys the old PEX structure, and invokes *BuildConeStructure* again to contruct the new cone structure and store the information into a new PEX structure. *SetLevel* then redraws the cone using the new PEX structure.

The callback function *RightArrow* subtracts the X coordinate of the view normal plane by a small number, say 0.1. This means that we are shifting the direction of gaze a little to the left, causing the cone tree to be shifted a little bit to the right. The function then redefines the view using this new value of the view plane normal, and redraws the cone using the updated view. *LeftArrow*, *UpArrow* and *DownArrow* do basically the same thing, The difference is that *LeftArrow* increments X coordinate of the view normal plane by 0.1, causing the cone to be shifted a little bit to the left, *UpArrow* decrements the Y coordinate of the view plane normal by 0.1, causing the cone to be shifted a little bit up, and *DownArrow* increments the Y coordinate of the view plane normal by 0.1, causing the cone to be shifted a little bit down.

Figure 16 shows the changes to XYZ coordinate of the view plane normal with respect to the arrow buttons.

Arrow Buttons	*X*	*Y*	*Z*
right	−0.1	unchanged	unchanged
left	+0.1	unchanged	unchanged
up	unchanged	−0.1	unchanged
down	unchanged	+0.1	unchanged

Figure 16: Arrow buttons

NonLeafModuleListCallback traverses the subtree linked list looking for a match of the module name selected, and sets *CurrentConeRoot* to point to the matched subtree. Recall that *CurrentConeRoot* points to the subtree which is rooted at the root of the entire cone tree to be drawn. Now, since the structure of the cone tree will change, the function destroys the old PEX structure which contains information about the old structure, and then invokes *BuildConeStructure* to rebuild the new cone, and redraws the new cone tree.

ExposeHandler is called every time the drawing window is exposed. This means that the first time when the drawing window is created and exposed on the screen, *ExposeHandler* is called. The function is also called when the drawing window was behind some other windows on the screen, but then selected to appear on top, or to be exposed. *ExposeHandler* basically just redraws the picture using the information stored in the PEX structure for the cone.

5.2 Problems

A few problems were encountered during the initial implementation of System Visualizer. One of the problems was that a subtree appeared twice if it was rooted at the same *SmallestRootedLevel* two times. This is why the field *Drawn* is introduced in the subtree data structure. While drawing the cone, if System Visualizer encounters a subtree at its *SmallestRootedLevel*, it first has to check *Drawn* to see whether the subtree has been drawn. If it has, then the tool will not draw it again. Otherwise, the tool will set *Drawn* to true for the subtree, then start drawing it.

Let us take a look back at Figure 14 on page 60 which shows the visualization drawn by System Visualizer of the tree structure as shown in Figure 13 on page 59. Another problem which was encountered was that node E in Figure 14 was colored red, which means that E appears more than once in the cone. But this is not true, since E was drawn only once by the tool as shown by Figure 14. This, as discussed earlier, is because subtree C was duplicated as shown in the tree structure of Figure 13, and the problem is solved by computing properly the value of *AppearCount* for each subtree.

An initial version of the tools drew all spheres to be of the same size, regardless the locations of the spheres. This is incorrect in three dimensional visualization, and made it harder to interpret the cone tree since the user has to try to analyze the image to determine which sphere is closer and which sphere is farther. The problem is solved by utilizing PEXlib, using the perspective projection and placing the projection reference point close to the model.

Another problem which was encountered was the HLHSR problem, because HLHSR was not handled properly as we discussed earlier. Spheres which should have been behind appeared in front. The problem was then fixed by rearranging the algorithm of *BuildSubconeStructure* properly.

Also, the tool initially provided only six views of the model, namely straight on, right, left, straight up, right up, and left up. It is clearly insufficient to look at the model using only these six views. It then became clear that the tool needed to provide the capability for the user to shift the model around, a little bit to the right, a little bit to the left, a little bit up or a little bit down, until the model can be seen to its best advantage. The Arrow Buttons were introduced to allow the tool to provide this very powerful capability, using the view utilities of PEXlib, as discussed earlier.

5.3 Discussion of implementation

The implementation of the data structures was fairly easy. It was just a straight mapping from the design. However, let us note that it was not easy to derive the improved data structures from the original data structures. What was needed was flat data structures which do not impose any depth and tree structure. Another difficulty was the introduction of the fields which are used by the Display Module to draw the model, e.g. *AppearCount*, *SmallestRootedLevel*, etc. Even though these fields are not part of the module relationships, they are needed to properly visualize the information.

The parser was also not difficult to implement. All that we needed was to build the recursive function *BuildModuleRelationshipInfo* to analyze module relationships of the application software and store the information for later use by the Display Module.

It was more difficult to visualize the module relationships on the screen. In order to draw just a simple one level subcone, some analysis needed to be performed to discover formulas to compute locations of each sphere and how to draw lines connecting these spheres. The implementation of this required moving spheres around after originally having drawn them at the origin, and rotating the lines to their proper positions. All this has to be done prior to actually drawing the subcone on the screen, and the information has to be stored somewhere, since we can not actually draw a sphere on the screen originally at the origin, and then move it to another location later. PEXlib provides PEX structure to store information on how to draw an image without actually drawing the image yet. It also provides powerful utilities to move or rotate objects around.

Now, to draw a cone tree of several levels, the most difficult thing was to implement properly a recursive function, namely function *BuildSubconeStructure*, to recursively draw each subcone at a time from the origin, and then move the subcone to its proper position.

Also, the viewing capabilities of PEXlib had to be studied in depth in order to implement the tool to provide the capabilities of shifting the model around, drawing objects to have different sizes depending on whether the object is closer or farther to our eyes, etc. But, once PEX Views was understood, the implementation of these was fairly straight forward.

It is also worth mentioning that PEX can draw a complicated image rapidly, especially if the information of how to draw the image is already stored in a PEX structure. When the user uses the arrow buttons to shift the cone tree around, all System Visualizer does is to update the view definition which is used by the PEX structure. It does not recompute the PEX structure. Therefore it is very fast to shift the cone tree around using System Visualizer. Even if the tool has to recompute the PEX structure, e.g. to redraw the cone tree to have a different number of levels, etc, experiments show that PEX is also very fast.

Chapter 6

Discussion

6.1 Accomplishment of Design Goals

The goal of System Visualizer is to help a software engineer understand module relationships within a software system quickly and efficiently. Specifically, the tool will display module interconnections in three dimensional space as a cone tree. In order to achieve this, the tool needs to be able to:

- infer the relationships of all modules of the system.
- display module relationships as a cone tree in three dimensional space, where a sphere represents a module and a line connecting the two spheres represents the uses relationship between the two modules.

The three dimensional display of module interconnections will help the user extract module relationships of the system quickly and efficiently.

6.1.1 Module interconnections

System Visualizer extracts module relationships in a software system which was developed using the C programming language. In order to do this, the tool assumes that each module has a .c file and an associated .h file. Moreover, if module A uses module B, then the .h file of A includes the .h file of B. With these assumptions, the tool recursively reads all the header files of the system to infer module interconnections.

These assumptions are very reasonable. Each module A developed in the C programming language is associated with a .c file. Normally, module A has data structures, and these data structures are defined in the associated .h file of module A. Moreover, A also defines its function prototypes, i.e. function declarations, in its .h file. This is especially true if A was developed in ANSI C. When module B uses module A, this means that B uses some data structures defined in A, and/or B calls functions defined and implemented in A. If B uses data structures of A, then B must include the header file of A. If B calls some functions of A, normally B also includes the header file of A in its header file, especially if the system was developed in ANSI C.

Therefore, for well written software system developed in C, the tool will be able to infer module relationships of the system very accurately.

There are, however, existing software systems developed in C which violate the assumptions System Visualizer has made. For instance, module A of certain systems calls some functions of module B, and yet module A does not include the header file of B. In some systems, a module does not even have a header file. System Visualizer does not work very well with these ill-written software systems. These systems would need to be modified a little bit in order to use System Visualizer, i.e. each module needs to have an associated .h file, and if module A uses module B, then the header file of A needs to include the header file of B.

6.1.2 Three dimensional display

System Visualizer displays module interconnections of a software system in three dimensional space as a cone tree. Because of the limited space of the screen, System Visualizer will display a cone tree up to three levels only. If the 'uses' relationships of all modules in a system is less than or equal to three level deep, then the tool will display the entire module interconnections of the system. If the depth of the 'uses' relationships of the system is greater than three, then the tool will still display the cone tree only up to three levels. However, the user can select non-leaf modules at level three, and the tool will re-display the cone tree, this time with the root of the cone as the selected non-leaf module. Therefore, the tool can provide the user with the capability to quickly extract all module interconnections of a system.

As an example to illustrate how a user can extract all module interconnections of a system, consider Figure 17:

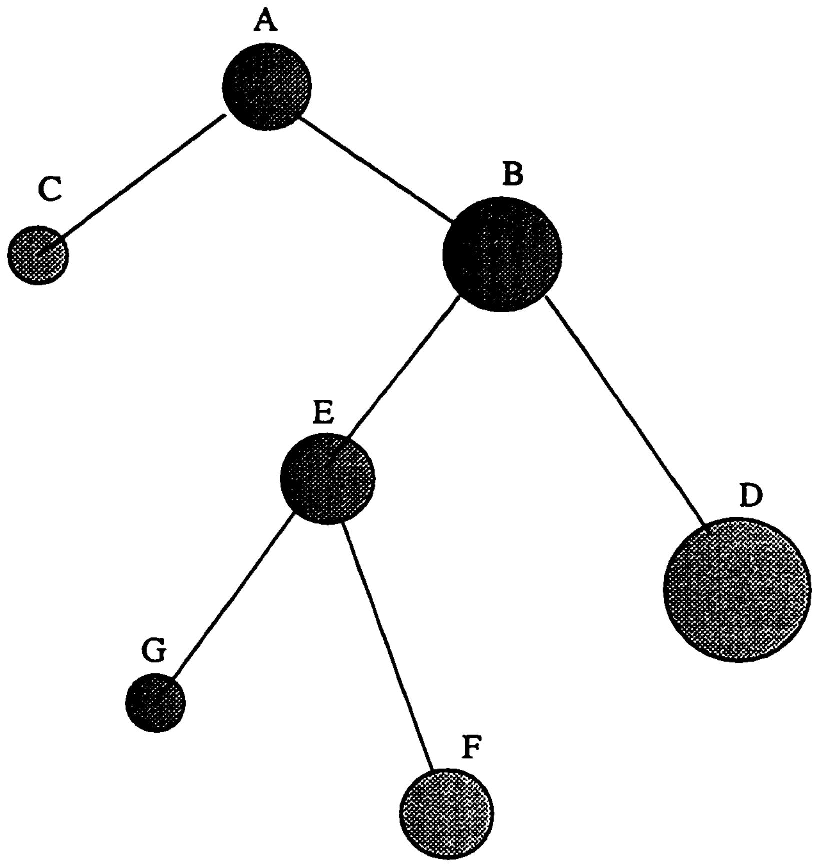


Figure 17: Image of more than three-level deep system

In Figure 17, the system has A as the main module. Non-leaf modules are A, B, E, and G and they are colored differently from leaf modules. Since G is colored differently from leaf modules, the user knows that it is a non-leaf module and he can select module G at level three to extract the rest of module interconnections of the system, as shown in Figure 18:

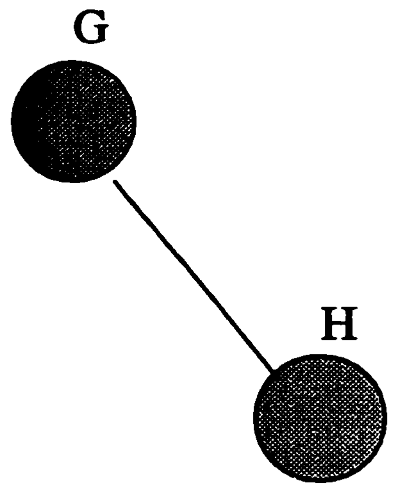


Figure 18: Module interconnections display starting at module G

Because of the limitation of the available space of the screen, the tool cannot provide the user with a cone tree which represents all modules of a software system if the uses relationships among the modules of the system is deeper than three levels. However, the tool provides the user with the capability to extract all module relationships within a system, as we have demonstrated with the simple example above.

Normally, a software engineer will need to change or enhance a small part of the system only. In this case, the engineer would want to study the module relationships starting at a specific module M, and down a few levels only. Then the engineer would just need to select module M, and the tool will display exactly what the engineer would like to study.

6.2 Contribution of System Visualizer to Software Engineering

A software engineer new to a software system needs to understand the system in order to maintain and enhance the system. One of the very important tasks for the engineer to understand the system is to understand the module relationships of the system.

It would be a tedious task for the engineer to try to understand module interconnections without the help of a software engineering tool. The engineer would have to manually read the header file of the main module M to find out the modules used by M. The engineer would then probably need to plot down this information on a piece of paper in some form. Then the engineer needs to repeat the task for every module which is used by module M.

Traditional tools do help a software engineer to accomplish the task in some form. However, these tools display the information in two dimensional space. There are limitations associated with two dimensional display, such as space, i.e. there is not enough room for many nodes on the screen.

System Visualizer provides three dimensional display of module interconnections. There are advantages associated with three dimensional displays over the two dimensional approach. One such advantage is space. With the same window space, three dimensional images can display a lot more nodes than the two dimensional images. Moreover, since in three dimensional space, we can display objects closer to the viewer's eyes larger, and objects farther from the viewer smaller, a 3D image can be understood better.

There is also a very powerful capability associated with three dimensional space. It is the capability to rotate a 3D image. As a result, a 3D image may look confusing from a certain direction. But if the image is rotated, it may look very clear from another direction.

As an example to illustrate the usefulness of this capability, consider the image as shown in Figure 19:

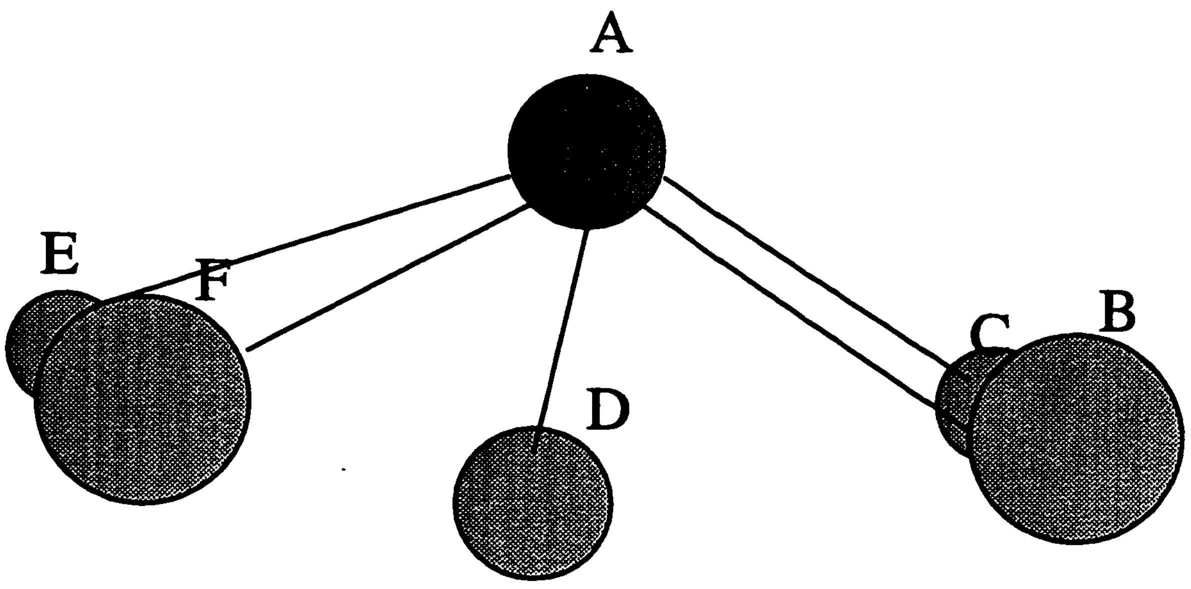


Figure 19: A cluttered 3D image

In Figure 19, it is hard to extract modules which are used by module A. Nodes B and C overlap each other. Nodes E and F also overlap each other. However, if we rotate the image a little bit, the result is as shown in Figure 20:

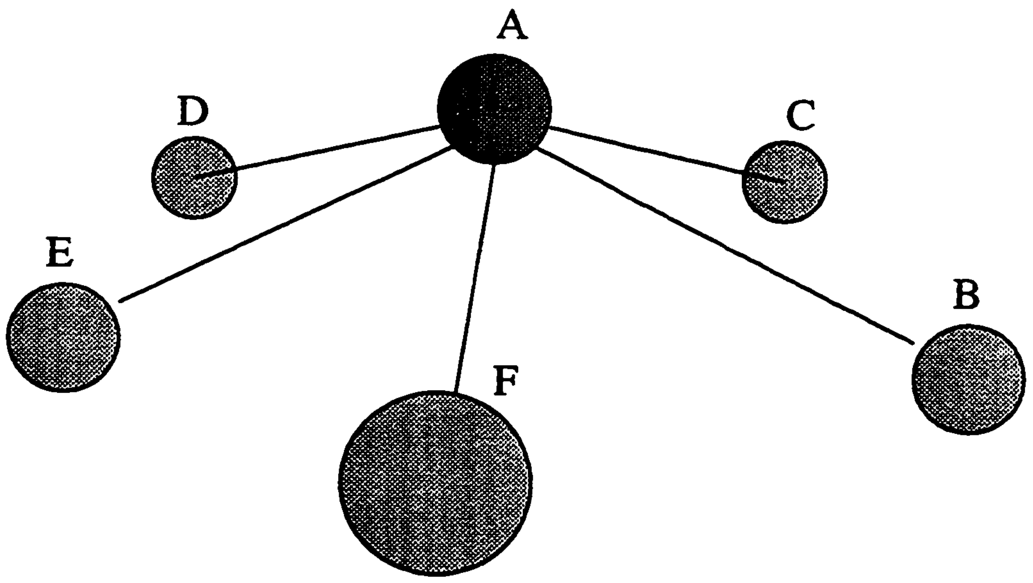


Figure 20: The cluttered 3D image which has been rotated

Figure 20 is a rotated image of Figure 19. In Figure 20, the image is very clear. The user can easily see that module A uses modules B, C, D, E, and F.

The capability of being able to rotate an image in three dimensional space can help the user see a part of the system very clearly.

6.3 Capacity of System Visualizer

Let us first try to see that for a one-level cone tree, how many leaf spheres the cone tree can have before the screen gets too crowded to be useful.

Consider a one-level cone tree as shown in Figure 21:

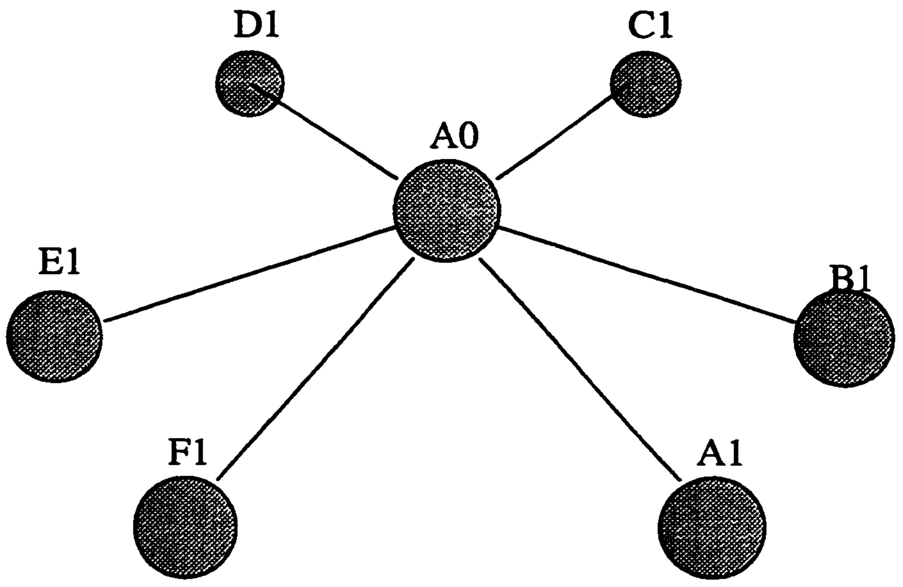


Figure 21: A one-level cone tree

In Figure 21, the image has been rotated so that module A0 is in the center. All modules which are used by A0 are placed around module A0. As we can see from Figure 21, A0 can have many more nodes around it and the image will still look very clear. From this example, it is reasonable to say that when the cone tree of one-level is to be displayed, the tool can display a very large number of leaf spheres and the image will still look clear. This means that even for a module which uses many other modules, the tool will be able to display the information very clearly. This capability is a lot more powerful than any 2D images generated by traditional tools.

For a cone tree of two or three levels, if there is a very large number of modules to be displayed at each level, then probably we will not be able to see the whole cone tree very clearly. However, since in three dimensional space an image can always be rotated, the user will be able to rotate the cone tree so that the part of the system which the user is interested in can be shown very clearly.

Let us consider an image displayed by the tool as shown in Figure 22:

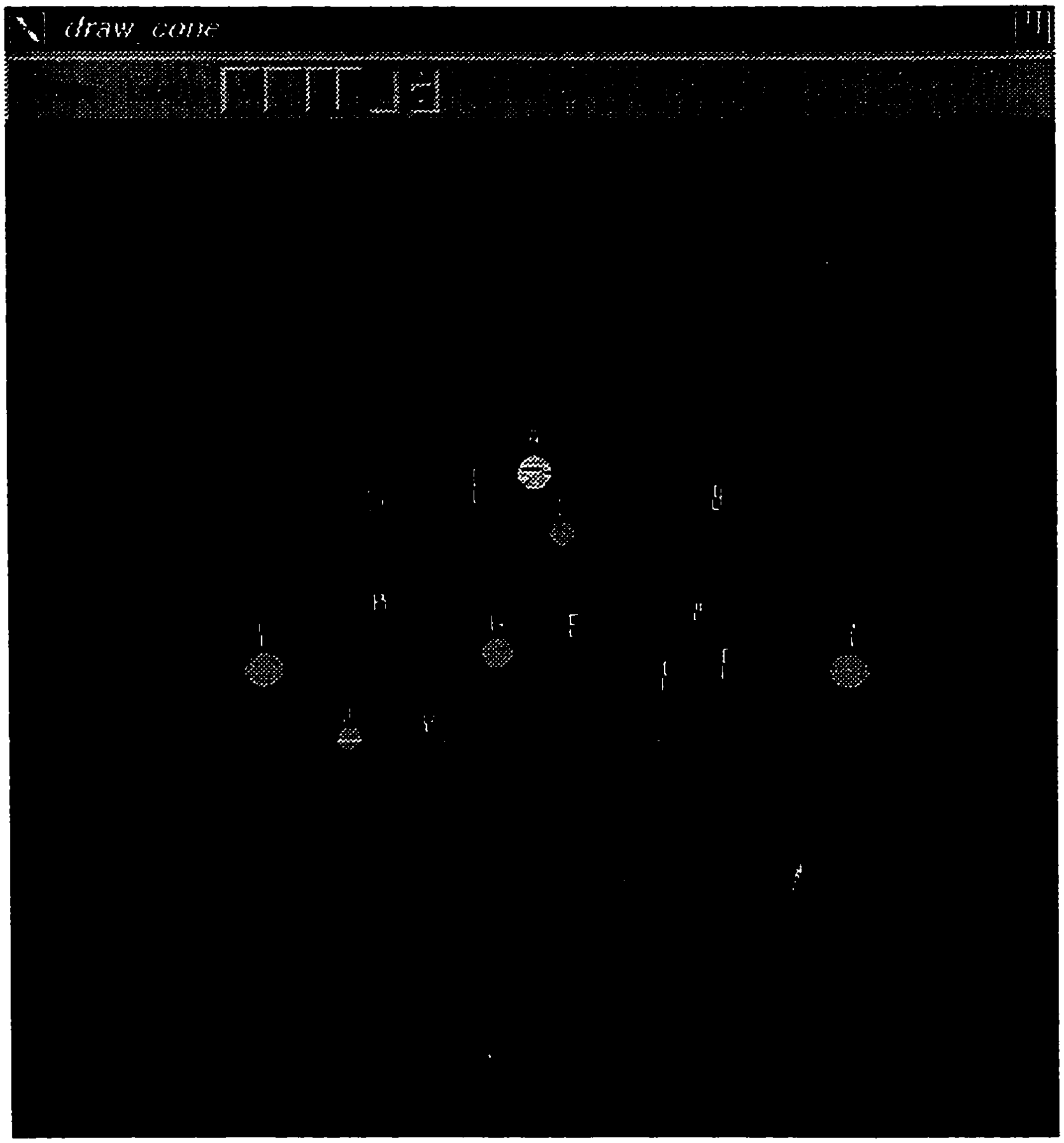


Figure 22: A three-level cone tree

Figure 22 is an image of a cone tree with three levels. The cone tree has a number of modules, and it has been rotated so that the whole image looks very clear. As we can see from the image, there is still room for a lot more nodes without cluttering the image.

6.4 Cyclic dependencies

Cyclic dependencies between two modules A and B, in its simplest form, occurs when module A directly uses module B and module B directly uses module A. This means that A either calls a function defined and implemenmted in B or A uses some data structures defined in B. At the same time, B also either calls a function defined and implemenmted in A or B uses some data structures defined in A.

In a more general form, there is a cyclic dependency between modules A and B if A uses B1, B1, uses B2, ..., Bn uses B and B uses A1, A1 uses A2, ..., An uses A.

Consider a simple example. Module A uses modules B and C. Module B uses modules C and D. Module C uses modules B and E. Here we have a cyclic dependency between modules B and C since B uses C and C uses B. How can we visualize the module relationships of this simple system?

One way of visualizing this system is as illustrated in Figure 23

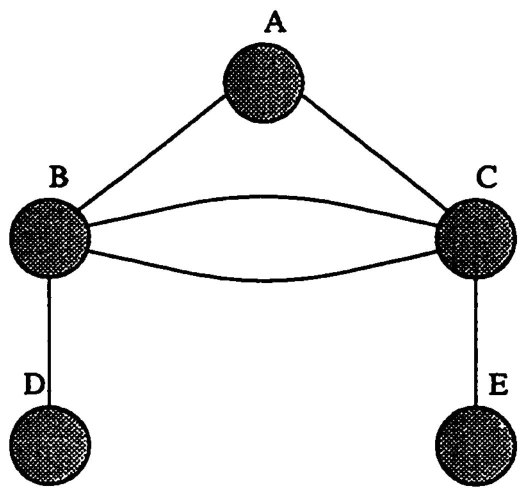


Figure 23: An example of visualizing cyclic dependencies

There are two problems with the way to solve cyclic dependencies as shown in Figure 23. The first problem is that it violates the tree structure of the system. With respect to the subtree rooted at B, B is the root and C is a leaf. However, with respect to the subtree rooted at C, C is the root and B is a leaf. The second problem is that the visualization can easily become cluttered. Consider Figure 24:

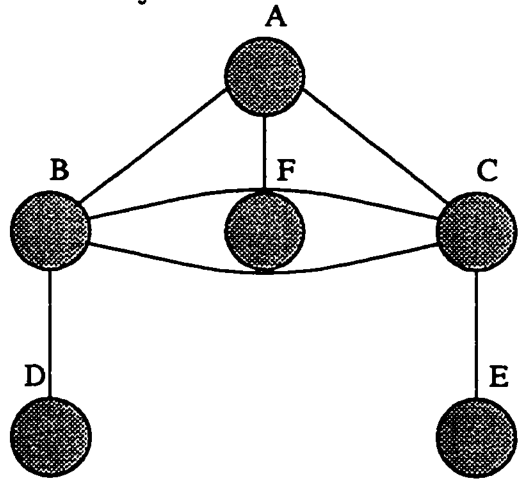


Figure 24: A cluttered visualization of cyclic dependencies

As we can see from Figure 24, with the introduction of node F in between B and C, the image becomes cluttered already. And this is a very simple image. Imagine how confusing an image is going to be when the system becomes more complex with more cyclic dependencies.

System Visualizer solves the problem using the approach as illustrated by Figure 25:

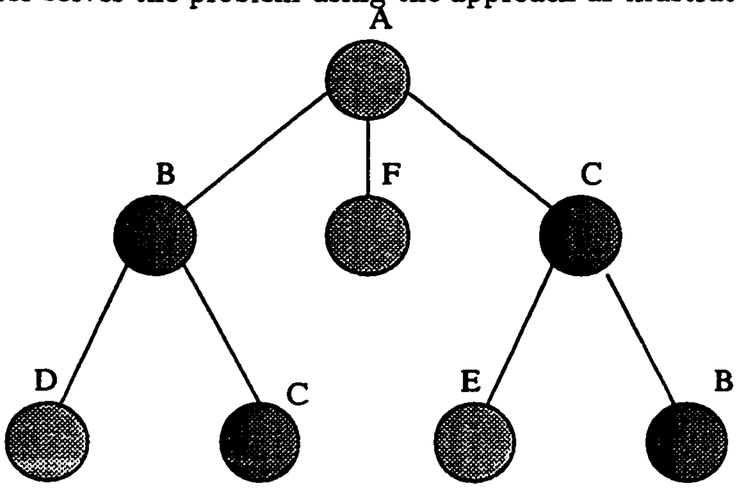


Figure 25: An uncluttered visualization of cyclic dependencies

In Figure 25, since module B uses C and D, we have a subtree rooted at B which has two leaf nodes C and D. Also since module C uses B and E, subtree rooted at C has its own leaf nodes B and E. However, since nodes B and C appear twice, they are painted with different colors.

System Visualizer represents different types of modules using spheres of different colors. Modules which do not use any other module of the application software system are represented as blue spheres. These are leaf modules. Modules which use some other modules of the system and only appear once in the cone tree are represented as yellow spheres. These are non-leaf modules. Non-leaf modules which appear more than once in the cone tree are represented as red spheres. When there is a cyclic dependency between two modules, these modules are represented as red spheres since they are non-leaf modules which appear more than once in the cone tree.

6.5 User Interface Facilities

Let us take a look back at Figure 22, which is an image generated by System Visualizer. On the left of the bar on the top of the window, there is a 'Views' button. Clicking on this button results in a scrolled down menu which has the following entries: straight, right, left, straight up, right up, left up. Please refer back to section

4.3.3 for the use of each of these entry. If the user select one of these entries in the scrolled down menu, the tool will re-draw the image as if the image is being looked at from the corresponding direction. Using the 'Views' scrolled down menu, the user can quickly select the direction from which the image looks best among the other 5 directions. Now, if the image still looks not completely clear, the user can use the arrow buttons, which will be discussed shortly, to rotate the image until it becomes completely clear.

The 'Levels' button next to the 'Views' button is another scrolled down menu. If the 'Levels' button is selected, a scrolled down menu will appear. There are 3 entries in the scrolled down menu: 1-level, 2-level, 3-level. When the user select one of these entry, say 2-level, the tool will re-display the image which is a cone tree of 2 levels. By default, the tool displays the cone tree of 3 levels. But if the 3-level cone tree has too many nodes, and therefore the image may look confusing, then the user can use the 'Levels' button to re-display the cone tree with less levels so that the image looks cleaner. As discussed in section 6.3, a cone tree of 1 level can have many leaf nodes, and the image will still look very clear.

The next four buttons on the menu bar are arrow buttons. These buttons are used to rotate the image around, e.g. a little bit up, a little bit down, a little bit to the right, a little bit to the left, as discussed in sections 4.3.3 and 5.1.5, until the image appears clear. These arrow buttons are by far the most useful user interface facility provided by System Visualizer. With the use of the arrow buttons, the user can rotate a very complicated 3D image until the part of the system which the user is interested in appears very clear. These arrow buttons show a clear advantage of 3D visualization over the traditional 2D visualization.

The list at the most right corner of the menu bar is a scrolled list where each entry is the name of a non-leaf modules, i.e. a module which uses some other modules of the software system being visualized by the tool. System Visualizer by default displays a cone tree of 3 levels where the root node of the cone tree represents the main module of the system. However, the module relationships of the system may be deeper than 3 levels. In this case, there will be non-leaf modules at level 3. The user can recognize whether a node is a leaf or non-leaf module by its color. A sphere which represents a leaf module is colored blue. Therefore any sphere which is either colored yellow or red represents a non-leaf module. The user then can scroll up or down the scrolled list to find a corresponding entry, and select the entry. Upon an entry is selected, the tool will re-display the cone tree which has root to be the non-leaf module selected. With the use of the scrolled list, the user can find out module relationships of all modules of a system, even when the system has module relationships which are very deep.

Chapter 7

Future Research and Conclusions

7.1 Possible Future Enhancements

There are a number of possible enhancements which can be made to System Visualizer:

1. Currently System Visualizer displays a cone tree of at most three levels. This is because of the limited space of a window. Therefore, for an application software which has the depth of the 'uses' relationships between modules greater than three, the tool will only display a partial 'uses' relationships information. This does not provide the user with an overview of the system structure.

   One possible enhancement is to modify the tool to display the whole system structure, even if the cone tree may contain many levels. The thesis already provides an algorithm to display a cone tree of many levels. The only thing which needs to be changed is to set *MaxLevelToDraw* appropriately. However, the important thing which has to be looked at is to scale the cone tree properly so that the whole system structure can fit into the window space.

   Since the overview of the system structure can be very crowded and confusing, the size of the spheres can be very small, the tool needs to be able to allow the user to display the cone tree of a desired number of levels, e.g. 3 levels, with a selected node as the root of the cone tree. This will allow the user to look at a part of the system very clearly. This is the capability the tool currently provides.

2. Another possible enhacement is to allow the user to exclude certain nodes from the cone tree to be displayed. As an example to illustrate this, consider a simple cone tree as shown in Figure 26:

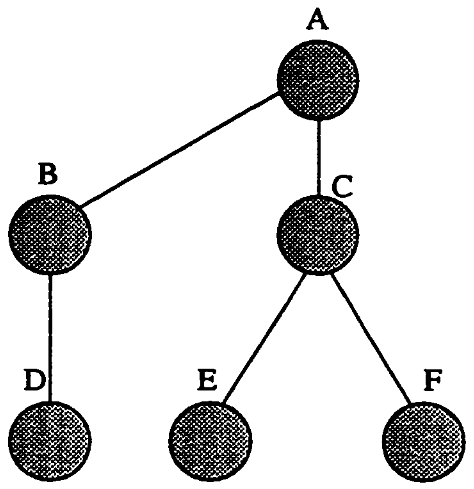


Figure 26: A whole cone tree with no nodes excluded

Suppose that the user is not interested in the subcone rooted at B. Therefore displaying the cone with the subcone rooted at B is really useless to the user. With the current implementation of the tool, the whole cone tree will be displayed. Now, the enhancement will allow the user to select node B, and to exclude the subcone rooted at B. The result will be the cone tree as shown in Figure 27:

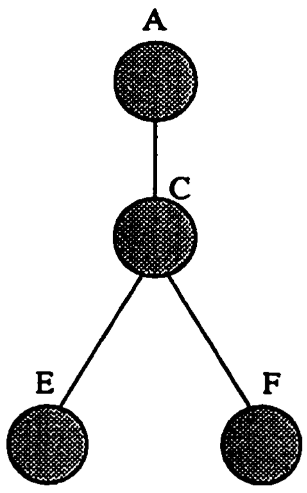


Figure 27: A whole cone tree with nodes B and D excluded

Sometimes, even a part of the system may contain many nodes. As a result, the cone tree representing this part of the system may be very crowded. In three dimensional space, the cone tree can be rotated so that at least a portion of the cone tree can be seen very clearly. However, the whole cone tree may not be seen very clearly. It happens often that the user is only interested in looking at a number of nodes of the cone tree. Other nodes of the cone may not be of interests to the user. Therefore the ability to display a cone tree, excluding uninteresting nodes, together with the ability to rotate a 3D image may result in a very clear cone tree which contains only interested nodes to the user.

3. Currently, the tool extracts module relationships by reading header files. The tool has made the assumption that each module has a '.c' file and a corresponding '.h' file, and that when a module A uses another module B, the header file of A includes the header file of B. As discussed in Chapter 6, there are existing software applications which violate the assumption the tool has made.

   Therefore, the tool can be enhanced to extract module relationships of a software application without making the above assumption. In order to do this, for example we want to find all modules which are used by module A, we need to perform two tasks:

   (a) For every function which is called in module A, we have to find the module B which defines and implements the function. B is then a module which is used by A.

   (b) For every data structure which is used in A, we need to find the module C which defines the data structure. C is again a module which is used by A.

   Clearly, it would be necessary to develop efficient scanning algorithms to make such an approach feasible.

4. System Visualizer can also be enhanced to extract function calling relationships. Given function A, the tool would then find all functions which are invoked by A. The tool can do this by just reading the implementation of A and extracing all functions invoked by A. Now, for every function B called by A, the tool will need to find the module which contains the implementation of B. The tool then can read the implementation of B to extract all functions called by B. The tool will need to do this recursively to build the function calling relationships which has the root as function A.

   The Display Module of System Visualizer is completely reused to display the cone tree which represents function relationships. The only thing we need to take care of is to build the calling relationships, and to pass this information using the data structures the Display Module expects.

   There is one potential problem we will need to look at though. A module does not use itself, but a function can call itself. Currently, System Visualizer does not take care of this. In order to allow the tool to display function calling relationships, we have to enhanced the tool to take care of recursive functions. The change for this is however very minimal. If a function A calls itself, instead of displaying two spheres representing A connected by a line, we may just display one sphere representing A. However, we need to draw sphere A using another color.

5. System Visualizer can also be enhanced to extract class hierarchy of a given class A in a software system developed using the C++ programming language. For a given class A, we can search for all subclasses of A by looking in each

module of the system for classes which inherit from A. Now, for each subclass B of A, we can recursively find all subclasses of B. Using this method, we can build the class hierarchy of A.

Again, the Display Module of the tool does not need to be changed in order to display the cone tree representing a class hierarchy. The class hierarchy information just needs to be stored using the data structures expected by the Display Module, and the Display Module will be able to display the corresponding cone tree representing the class hierarchy.

6. System Visualizer can be integrated with other software tools such as a text editor. A text editor is used to edit source files of a software application. One possible way for a text editor to work with System Visualizer is to enhance the text editor to have an additional button on the menu bar on top of the text editor window. When the user edits a module A of a software application, if the user clicks on the button, then the text editor can start up System Visualizer to build the module relationships with A as the root, and to display the cone tree representing the corresponding module relationships.

   A text editor will also work very well if System Visualizer is enhanced to be able to extract function calling relationships or class hierarchy, and to display the corresponding cone tree. When the user edits a source file of an application developed using the C or C++ programming language, if the user selects a function or a class, the text editor could then start up System Visualizer to build the information and display the corresponding cone tree.

7.2 Conclusions

As discussed in Chapter 2, visualization is extremely helpful in countless areas, including software engineering. However, research on visualization in helping the understanding of existing software systems or in helping to speed up the development of a software system which is free of errors is still in its very early stage. Moreover, the traditional research on software visualization concentrated mainly on two dimensional visualization which has its limitations.

The thesis serves as an initial research in the field of three dimensional software visualization. Understanding the module structure of an existing software application is a typical problem in software engineering. The thesis has tried to solve this problem by using three dimensional display of the information.

The thesis has succeeded to solve the problem in the following points:

1. An algorithm to extract module structure of a software application developed in C or C++ is provided. The algorithm was developed based on the assumption that each module has a '.c' file (or .C file in the case of C++) and an associated '.h' file. Moreover, if a module A uses another module B, then A.h will include B.h. The assumption is very reasonable for well-written application.

   Although the algorithm was developed to extract module relationships information of an application only, it can easily be enhanced to extract other types of information such as function calling relationships or class hierarchy.

2. The thesis has provided an algorithm to display the system structure in three dimensional space as a cone tree. Even though System Visualizer only displays a cone tree of maximum three levels, the algorithm can be used to display a cone tree of any number of levels.

3. A generic interface to the Display Module of System Visualizer was developed through the introduction of the data structures expected by the Display Module. With this generic interface, the tool can easily be expanded to display other types of information without having to change the Display Module at all.

4. Using PEXlib, a 3D extension of the X window system, in its implementation, the thesis has shown some outstanding characteristics of three dimensional display.

   One of these outstanding characteristics is the ability to rotate a 3D image. As a result, a very crowded cone tree with many nodes can be rotated until the whole cone or the interested portion of the cone becomes clear.

Another useful characteristic associated with three dimensional display is that objects which are closer to our eyes appear larger and objects which are farther from the eyes appear smaller. Also, objects in three dimensional space may overlap. If an object A is behind object B, and A and B overlap, then the portion of A which overlaps with B is hidden in three dimensional space. The result of these is a much more comprehensive cone tree.

The thesis has also shown that for the same window space, many more objects can be displayed in three dimensional space as opposed to in two dimensional space.

There are however a few limitations associated with System Visualizer:

1. The tool fails to extract module relationships information for software applications which violate the assumptions the tool has made, i.e. each module has a '.c' file and an associated '.h' file, and if module A uses module B then A.h includes B.h. Even though this is a good assumption for well written application, many existing software systems violate this assumption.

2. The tool fails to provide an Overview of the whole system structure. However, as mentioned above, the algorithm to display a cone tree can be used to provide an Overview of the whole system structure. The only problem which needs to be looked at is how to scale the cone tree accordingly.

In summary, despite of some limitations associated with System Visualizer, the thesis has provided an initial research in the field of three dimensional software visualization.

Bibliography

[1] Jacqueline M. Antis, Stephen G. Eick, and John D. Pyrce. Visualizing the structure of large relational databases. *IEEE Software*, 13(1):72–79, January 1996.

[2] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–42, April 1996.

[3] Richard Mark Friedhoff and William Benzon. *The Second Computer Revolution: Visualization.* W.H. Freeman, 1989.

[4] Andrew J. Hanson, Tamara Munzner, and George Francis. Interactive methods for visualizable geometry. *IEEE Computer*, 27(7):73–83, July 1994.

[5] Dean F. Jerding and John T. Stasko. Using visualization to foster object-oriented program understanding. *Georgia Institue of Technology, Technical Report GIT-GVU-94-33*, July 1994.

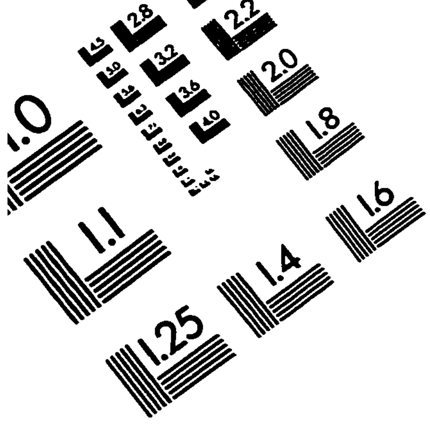
[6] John J. Shilling and John T. Stasko. Using animation to design, document and trace object-oriented systems. *Georgia Institue of Technology, Technical Report GIT-GVU-92-12*, June 1992.

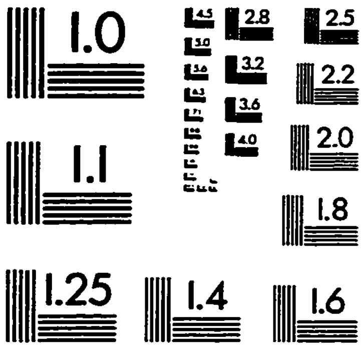
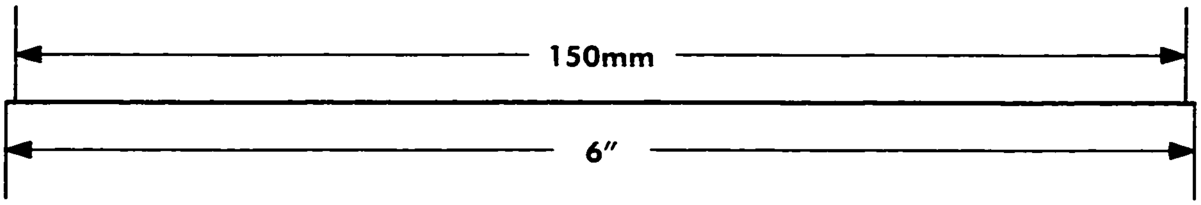
[7] Nan C. Shu. *Visual Programming.* Van Norstrand Reinhold, 1988.

[8] Ian Spence. Visual psychophysics of simple graphical elements. *Journal of Experimental Psychology: Human Perception and Performance*, 16(4):683–692, 1990.

[9] John T. Stasko. Three-dimensional computation visualization. *Georgia Institue of Technology, Technical Report GIT-GVU-92-20*, September 1992.

[10] Hans van Vliet. *Software Engineering: Principles and Practices.* John Wiley and Sons, 1993.





150mm
6″