Program Visualization in Programming Environment

A thesis
submitted in partial fulfilment
of the requirements for the degree
of
Master of Science in Computer Science
at
Massey University
By
Shurong Dong

Department of Computer Science
Massey University
New Zealand

1991

Abstract

This thesis describes the investigation of techniques for the interactive display of program source code, with particular emphasis on large and complex programs. Folding and fisheye display techniques have been investigated, coupled with "hot-spot" or embedded menu interaction techniques typical of hypertext systems. Automatically created links through identifiers, procedures, etc, are employed to enhance the user's ability to comprehend the program structure and to methodically move about within it. Some suggestions are made concerning high level data processing and more advanced programming environments. A prototype system has been developed on a UNIX workstation to provide an environment for the C language.

Acknowledgements

I would like to thank Professor Mark D. Apperley who proposed and supervised the project. His support and encouragement has enabled this thesis to be completed in the time possible.

Thanks are also due to Colin Engle, John Holley and other staff of the School of Information Sciences who support computers in the laboratory.

Thanks are also given to those lecturers and students who have helped me with my English writing and helped me through the process of programming.

Table of Contents:

References

Appendix 1 : Quick Use of HI

Appendix 2 : A Simple Program for Testing

Appendix 3 : Abstract Program

Appendix 4 : Compressing One Procedure

Appendix 5 : Compressing All Procedures

Appendix 6 : Multiple Text Subwindow & Structure Diagram

Appendix 7 : Comment

Appendix 8 : A Complete Map of A Complex Program

Appendix 9 : Map of A Simple Program Showing A Variable

Appendix 10 : Map Viewing with A Small Window

Chapter 1

Introduction

A program is subject to modification during its life cycle. In order to modify it, the program must be easy to understand by programmers. This is particularly important during the maintenance phase of a software system life cycle.

Software maintenance is a time-consuming task, and occupies most of the time in a software life cycle. Many factors influence the time taken for maintenance. These include the readability of the program, the complexity of algorithms, the available tools, and various other factors. For a very large and complex piece of software, an efficient and powerful environment that supports the interaction of the user with the source code would be of great benefit.

Until now, the main technique used to read and understand a program has been a mental simulation of computer runs, along with the reading of specifications. However, running a program does not give any sense of how a program is organized, nor how to modify and improve it. This is because programs are written for execution on a computer, not for reading. The comments associated with source code are peculiar to the programmer who created the source code, and may be difficult for others to follow. There is a demand for a simple and standard representation of source code.

In the remainder of this chapter, the general concepts of an interactive programming environment and the two aspects of Program Visualization and Visual Programming are introduced. The emphasis here is on Program Visualization, which is the main concern of this thesis. Chapter 2 discusses some factors of program comprehension, and concentrates mainly on the understanding of large programs by humans. Chapter 3 introduces some current display techniques for large and complex documents, and describes how they can be used in an interface. Chapter 4 analyses the objects in a programming environment with which the user needs to interact. This includes both the data that the program uses and the tools which support the environment. An advanced programming environment is proposed. Finally, Chapter 5 describes the

implementation of a prototype of a simple interactive programming environment with a graphical representation of the program.

1.1 Development of The Programming Environment

1.1.1 Introduction to Programming Environments

The earliest programming environments were concerned only with the communication between source code and the computer. The user told the computer how to do the tasks, not simply what to do. As techniques in computing developed, the languages used, from assemblers to high level languages, became more understandable by humans. Facilities included operating systems, debugging tools, text editors and so on, and became more and more powerful and efficient. These assisted in the interaction between the programmer and the code.

Command and batch oriented systems are a poor match for the human cognitive system, which is very visually oriented and computes incrementally. So people began to realise the need to improve the user interface and its performance, and developed data oriented systems. A data oriented system is one in which users interact directly with system data, instead of using a complex set of commands.

The "Smalltalk" system developed at the Xerox Palo Alto Research Centre [Goldberg, 1983] is an interpreter for an Object-Oriented Language based on message passing. "Smalltalk" is extremely good at application prototyping. It is not only a programming language, but also a programming environment. It successfully uses a window-based graphic display system to reduce complexity. There are many other environments created on the basis of "Smalltalk", which use a high resolution display with more powerful interaction provided by the use of graphics or multiple views of a program.

1.1.2 Proposed Development

The goal is to develop a unified programming environment that stimulates program conception at a high level of abstraction, promotes programming by refinement, frees the user from frustrating syntactic details, allows source code

to be displayed in multiple views while editing programs, and provides extensive diagnostic facilities during program execution. This can be achieved by providing programmers with more powerful tools to cope with the increased complexity of modern systems. Such tools will enhance the interaction between programmer and code.

To achieve this goal, the source code should be considered as a structured database which is one of the objects in a vast space - that is the programming environment implemented on a screen. In the past, what a user could see was the source code only. Now, as a whole, source code is just one of the objects which needs to be analysed and manipulated visually. Other objects in the space are tools that users can use to act on the source code database. These tools include a compiler, a debugger, a run-time tool, those which support the environment such as icons and windows for text and graphics, and also control objects, such as a mouse, cursor, menus and buttons.

The environment not only has to provide for all queries the user may want to make of the source code database but also to call related tools to give the user the proper feedback.

1.2 Program Visualization

Glinert has drawn a destination between visual programming and program visualization [Glinert, 1990]. Visual programming refers to the use of graphics to define or to help define programs. The purpose of it is to make programming easier. PV is the use of graphics to make programs, and their execution, understandable.

Some high level languages are similar to natural language in order to be more understandable. The higher level the language is, the more complex the grammar will be. Even though programming language can be made the same as English grammar, this is still impractical because people do not all speak in the same way. So a formal programming language is still needed in writing. On the other hand, to understand a program, users should be able to read it in the way which they think they can understand it the best. This requires different kinds of views of a program. Tools for translating from the formal programming

language to different representations will be a great help for users to understand the program.

Program visualization can be used in teaching algorithms, complex data structures, program debugging, algorithm design and so on. It is used to support the basic and particular needs of users. The primary needs of a programmer during development and maintenance are getting information, finding the relationships between data (i.e source code or tools), and viewing objects in different ways [Rajlich, 1990].

1.3 Considerations for Program Visualization

1.3.1 Type of Information

One problem with program visualization is how to represent different types of information. The information can be as small as one character or one symbol, and it also can be as large as a graphics with many times screen size or a document with thousands lines. Information type can be anything which is able to be shown on the screen. The designer have to choose the most useful and important ones in order to make the interface clear and efficient.

For example, in a map, how should a procedure be represented? Are variables local or global? Which information is static, and which is dynamic? How should feedback on correct actions be provided to the user? All the information has to be presented in a suitable way and it should be consistent. For example, the static information, which should always be shown in a particular place such as the main frame, includes the text window, abstraction window, message/control window, and an additional window(see Figure 1.3.1). In order to distinguish the main frame from other frames, the main frame can be defined using a double outline, while others can be single outlines. Different types, formats, shapes and positions can mean different objects.

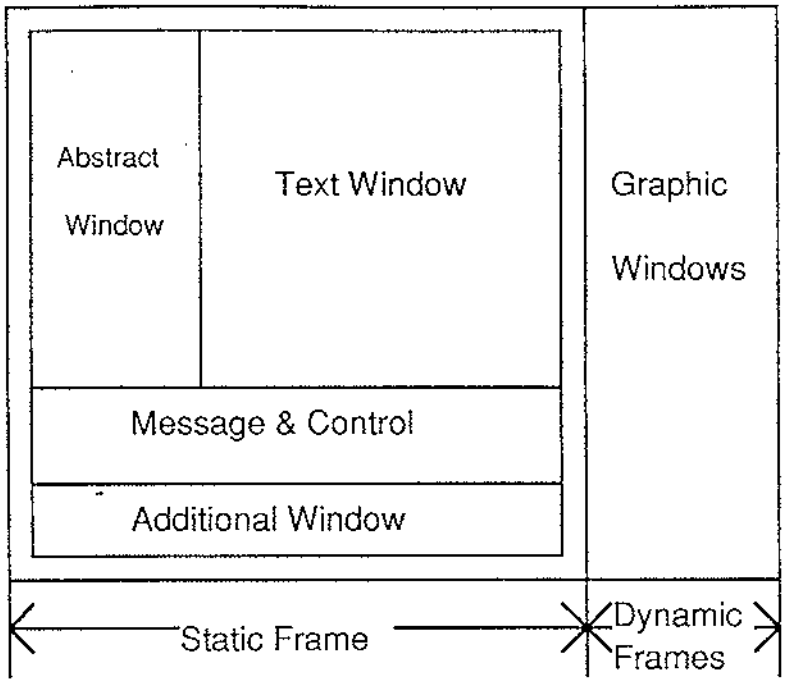


Figure 1.3.1 Different Information in Different Windows

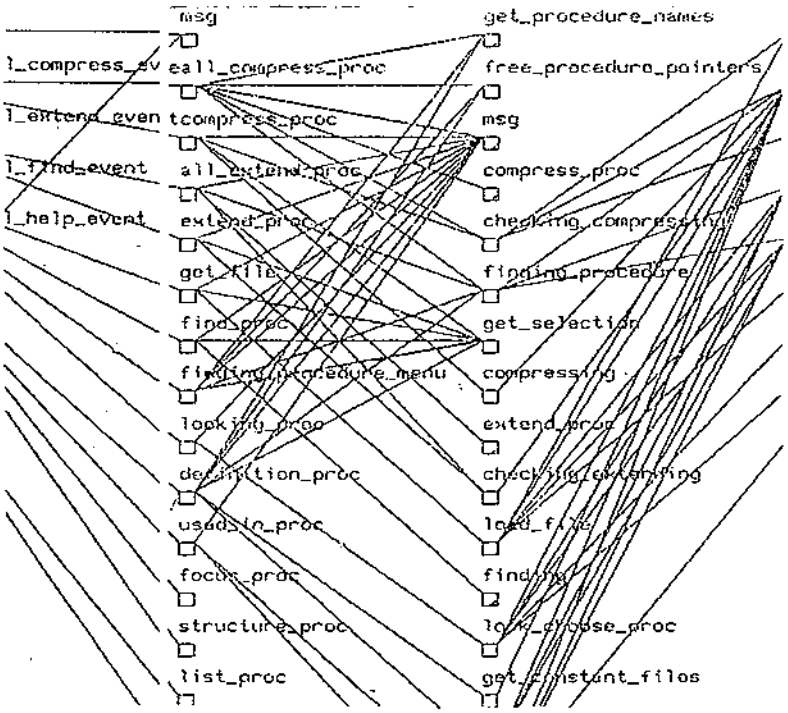


Figure 1.3.2 The Complexity of A Map
(Part of Map in Appendix 8)

1.3.2 Map Size and Complexity

A map is a diagram which shows the relationship between procedures in a program, and is used by programmers as an aid in understanding the program. For a small program, programmers probably view a map in one or two views. When the program is very large, i.e. the number of procedures is about 100 or more, the map to represent the structure will be very large and complex. In these cases a clear, comprehensible, and well designed map is needed. Figure 1.3.2 shows an example which is part of the map used in the prototype introduced in Chapter 5. Although the map has been simplified, it still appears complex even to a moderately sized program.

1.3.3 Speed

The generation of a map should be fast enough to be used in practical situations. During programming, interactive formatting and syntax checking can take some time. If the time delay is too long, users will be dissatisfied no matter how good the interface looks. In a graphical environment, many of these operations are slow. Algorithms to accelerate the creation of a map are currently being studied by others.

1.3.4 Limited Screen Size

Screen size is the bottle neck between human and computers. How can all of the information be displayed? What information is more important, and should the user make use of tools? Such problems have been considered for a long time by many people and still exist. With the spread of applications of computers, much more information needs to pass through the size-limited screen. Although the screen size can be enhanced or several computer screens can be used at a time for one task, the cost is very high and some others problems emerge, such as the inconvenience of multiple screen views, and difficulty in concentrating on one particular point on a very large screen, which may be beyond the direct control of humans.

On a size-limited screen a well designed interface is essential. But designing a good user interface is not easy. Nothing can satisfy all the people for all tasks, and not all techniques can be used on all computers. Designers should know the computer and have a good understanding of user requirements.

Chapter 2

Program Comprehension

It can be difficult for a programmer to understand large programs. The following discussion looks at this problem, and examines how programmers relate to programs. It also looks at techniques for assisting programmers.

2.1 Programs in the Large

2.1.1 Creating and Understanding Large Programs

Most programs are created using text editors. A general text editor is usually sufficient for a small program. For a large program, the editor may have a special display format for a particular language and it may provide error checking while editing. However users find this is not enough for a very large program. Analyzing the structure of a large program is one of the most frustrating parts of software maintenance; it takes a lot of time.

In creating a program, the time for a single cycle is generally long from editing to compiling and sometimes execution. There are always some errors in compiling or at run-time. Some methods used to limit error occurrences in editing such as highlighting reserved words, symbol matching and so on, help programmers to work more quickly and correctly. There are also some techniques which can be used in compiling. For example, one large program can be divided into small subprograms which are compiled separately. In this way, errors can be found and corrected immediately and easily.

To understand a very large program is not an easy job, but this has to be done in the maintenance phase of the life cycle of a software product. A program maintainer must have an understanding of the whole program before reading it in more detail.

Program abstraction systems and program structure analysis can be a great help in the discovery process during software maintenance. Unfortunately there

is no efficient tool for programmers to comprehend the program in this way. Maintainers can understand the program only by reading the specifications at the first stage, but they still do not know how the program is structured before reading it closely.

2.1.2 Program Analysis

Large programs, may comprise many subprograms linked together. These will often be produced by more than one programmer. Each subprogram is independent and may be written in a different style, so the links between subprograms and links between procedures in one program are very important in creating a large program.

Inside a large program, some key parts exist. Such as global control variables, main flow function calls and basic statements of a function. These represent the core of a function or program, and also they show the relationship with other modules or functions. Experienced programmers always try to find these focus points when they read a program. Although the other parts of the program are also necessary, focusing on the key points enables programmers to get the main idea of each part immediately. Some current interactive techniques, which will be introduced later, enable key parts to look more obvious than other parts.

Different languages have different formats. All programming languages are not represented in the same way, but there are some standard tools which can represent a particular language in a common way. People will understand the structure of the program by reading the representation just as they understand the details by reading the source code. This structure is the abstraction of large amount of basic data. It takes less space and users can find where they are quickly.

Other information about a program can be considered, such as where it is going to be used and what the result of the application is. These information are the first step to understand the program.

2.2 Programmers

The first real study of human factors in programming appeared in 1980 [Shneiderman, 1980]. Gradually, this work grew from concerning student programmers to experienced programmers, and from small programs to large programs. By understanding how and why programmers do a task, it is easier to make prescriptions that can aid programmers in their task.

Programmers can be the designers or maintainers of a program. There are demands for designers to have not only a simple and complete language, but also a powerful programming environment which enables them to work creatively with fewer mistakes. To help the maintainer to comprehend a program, it is necessary to be in a good position to recommend changes in documentation standards that would enable the maintainer to more effectively glean from the documentation the necessary information. This implies a more understandable representation derived from the program. This representation can be in any form provided it is helpful to understanding the program.

Both designers and maintainers think about a program in two ways. One is bottom-up, the other is top-down. Bottom-up design is generally used in creating small programs. The programmer studies and interprets small groups of statements and stores the interpretations in his or her memory. The top-down method is used with larger programs so that the programmers can organize the structure beforehand and then create individual parts.

It is easy to create a program when the programmer has the basic idea of what the structure will be. Otherwise the program will become longer and longer, and the programmer will forget how it is structured. They have to reread the source code line by line to find out what it means and where it can be changed. Having the program structure represented visually is not only helpful in creating programs, but also useful for maintenance. Without the idea of the whole structure of the program, sometimes it is even harder to read a program than it was to create it in the first place, and it takes a longer time to understand the work of others by just reading the source code itself.

The study of programmers is still in its first steps. It is not only necessary, but also important. Working on computers is an art created by human beings, not just the carrying out of tasks that programmers should and should not do.

2.3 Program Comprehension and Manipulation through an interactive interface

The human computer interface is the window through which humans and computers communicate to each other. A user can get much benefit from a well designed user interface. It enables the user to be in control of the system, encourages the user to try anything with few errors, and users can learn very fast.

A program is understood by reading it in a certain environment. The environment contains a user model, command language, feedback and display. Figure 2.3.1 shows a common used environment for editing a text file. The user model may be a form of text editor which contains the source code, and some tools. It is the framework which provides guidelines for applying techniques. The command language is the set of operations that can be carried out on the objects in the model. These operations may involve dialogues, commands, menu selections or direct manipulation. The feedback is much more important in an interactive environment than in other environments. It is not only the feedback of the commands but also the visualization of the structure and content of the program source code. More attention must be paid to the display for large and complex systems, particularly to the layout and the information structure.

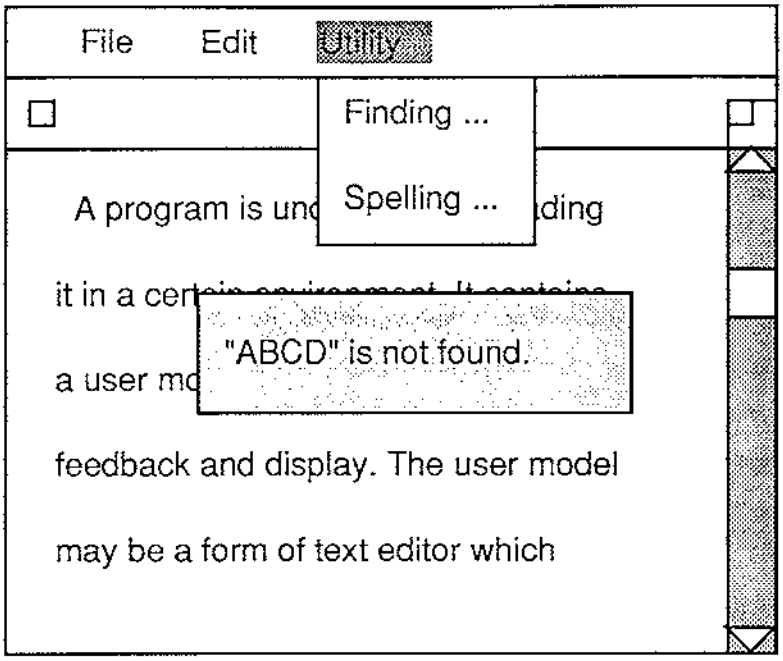


Figure2.3.1 A Common Environment

A few models have been proposed for program comprehension. Shneiderman and Mayer [Shneiderman, 1979] created a largely bottom-up model. According to this model, the programmer reads and interprets a program using the bottom-up method. New information will be understood on the basis of what the programmer thought earlier. Eventually by this incremental process, the whole program, or as much of it as possible, is comprehended. A similar model proposed by Basili and Mills [Basili, 1982] rejects the idea that a program is understood on a line-by-line basis. However, both methods are basically inductive and are driven by the data of the program text.

Brooks [Brooks 1987] proposed a model which can be considered as top-down process. The program is understood first by reading the specification, understanding the functions and guessing the structures. According to this understanding, the program then can be read in more detail.

No matter which model is used, the emphasis is always on the understanding of the source code. Consider the source code as an object composed of other objects in different levels. The simplest object is a character. A word is composed of characters. A semantic line is composed of words and symbols. Lines are grouped together as procedures and then programs. A line is the smallest unit for grammar checking. The most immediate feedback in editing the source code is the line grammar checking. It greatly reduces the occurrence

of errors. Placing a line with correct grammar in the right position using indenting techniques makes the program structure more clear and understandable.

Different operations are used on the different kind of objects identified above. Dialogue boxes are slow and are only used in some situations. Menu selection is simple, flexible and there can be no command grammar mistakes. But this method is also slow and it can be difficult to find the correct submenu. For experienced users, command-based interaction may be more efficient. The feedback required for an operation includes the correctness of an action, the start, ongoing and the end of the action, and also the effects of an action. The information for correcting errors, and the visual display of changes, are required in an interactive environment.

A model is the way to read a program, it is supported by the available display techniques and tools. The display techniques and tools enable programmers to read a program in multiple ways. By many experiments, graphics is more understandable by humans, while the display is central to interactive computer graphics manipulations.

Chapter 3

Current Display Techniques for Large Documents

Techniques for the effective display of information have received attention since the earliest use of the CRT. The earliest computer graphics displays came out in 1950. In the last 10 years, various kinds of display techniques have appeared. With the development of computer and electronic techniques and the reduction in cost of graphics devices, display techniques have been widely used to increase the quality and productivity of computer applications.

3.1 Techniques

Most techniques currently used for display are based on two basic concepts. One is data abstraction, the other is structuring. Abstraction covers information hiding and high level presentation, while structuring deals with the relationship between objects. Data abstraction makes the structure clearer and it reduces the complexity of the display. From the following techniques derived from these two methods it is possible to identify more special features that contribute to the interface display. Most of these can be implemented on graphical terminals.

3.1.1 Scrolling

Scrolling is a common viewing technique which enables users to view the content in a window continuously. Generally, it is realised by pressing the cursor up-down-left-right or page up-down buttons on the keyboard or by scrollbars visually (see Figure 3.1.1). The position of the rectangle in the scrollbar shows the position of the content in the window.

```
pos_x1 = 200;
pos_y1 = 660;
draw_box(pw, pos_x1, pos_y1, 10);
line(pw, pos_x+10, pos_y, pos_x1, pos_y1);
pw_text(pw, pos_x1, pos_y1-10, PIX_SRC, 0, "Copy");
draw_box(pw1, pos_x1/20, pos_y1/20, 2);
line(pw1, pos_x/20+1, pos_y/20+1, pos_x1/20,pos_y1/20);
}
```

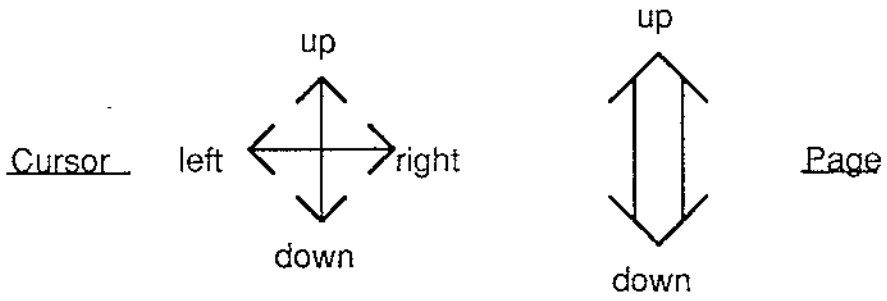


Figure 3.1.1 Scrolling with scroll bars

3.1.2 The "Fisheye" Display

In daily life, people are always interested in some particular things, and ignore others. For example, consider the representation of a map, the world map or street map for a local area. Some people are interested in the names of countries, some are interested in geographical layout and some want to see the special people of each area or get the information of products in each part of the world. All of those aspects of interest can be the key points which a "fisheye" view focuses on.

The "fisheye" view was proposed by George W. Furnas in 1986 [Furnas, 1986]. He suggested that it might be useful for the computer display of large information structures like programs, data bases, online text, etc. In viewing a large system through a size limited screen or window, it is always difficult for users to find the right place quickly by scrolling or other scanning methods because all the information is of the same importance, only a part of it can be seen and users have no idea where they are.

The "fisheye" technique, which is widely and naturally used in life, can also be used in the interface between human and computers. The application has

been used on hierarchical file systems and menu access systems, hierarchically organized text, graphic display and structured programming languages. With different algorithms, the view can also be used in different disciplines by seeing in different ways. It focuses on one particular part like keywords or what the user thinks is important. It is especially used in the representation of large and complex information spaces.

According to the "fisheye" view theory, the view is created from two aspects; one is the importance of an item, the other is the distance between the focus points. The data which is more important to the user or nearer to the focus point will be shown in more detail. There is a simple example which is shown below:

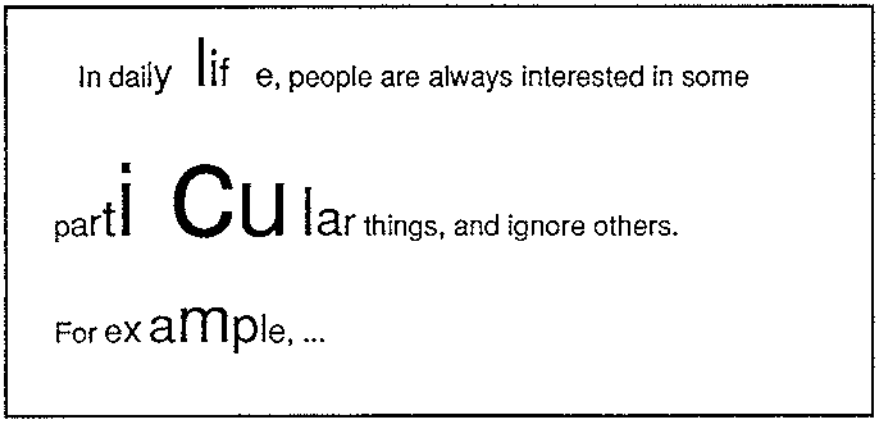


Figure 3.1.2 "fisheye" Display

In this example, "c" is the symbol a user wants to find out and understand its place or other information about it in detail. Those characters which are near "c" are larger than those far from it. The distance affects not only in horizontal but also in vertical.

Although users can get some important information immediately, it is impossible to catch all that they need. The larger the range is, the less detail the display can provide. Also in one view, users cannot see the detail in two separate parts which are far apart, for example, in comparing the geography of two countries, which are on just opposite sides of the Earth.

3.1.3 The "Bifocal" Display

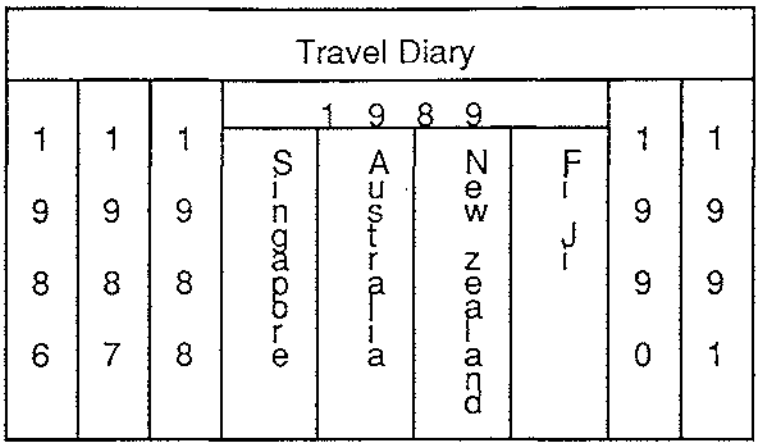
Instead of viewing objects continuously from detail to less detail, "bifocal" display [Apperley, 1982] focals on two parts: one is the detailed part, the other is the less detailed part. For example, consider reading a record which keeps the path of countries a person travels around the world. Instead of reading all the details in each part, readers may want to read one part in detail and scan other parts. By using "bifocal" display, readers can read any travelling record in one country and find out from where the traveller goes to the country and where he is going to.

"The key concept of the "bifocal" display is to focus on the area (of the data space) of interest, rather than to window on it, so combining an overview of the entire data-space with a detailed view of a small portion of the space. The implementation of the "bifocal" display can be achieved by providing three separate viewports, each with an individually programmed origin which can be moved continuously through display memory. [Apperley, 1982]"

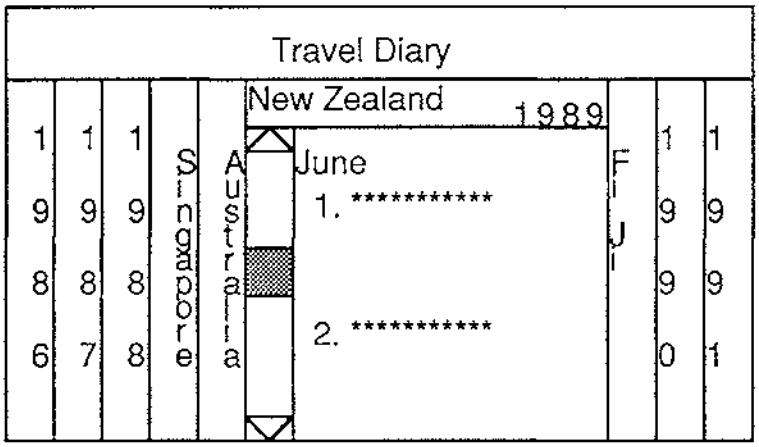
There is an example which shows how a "bifocal" display is different from the conventional display (see Figure 3.1.3).

Travel Diary					
1	1	1	1	1	1
9	9	9	9	9	9
8	8	8	8	9	9
6	7	8	9	0	1

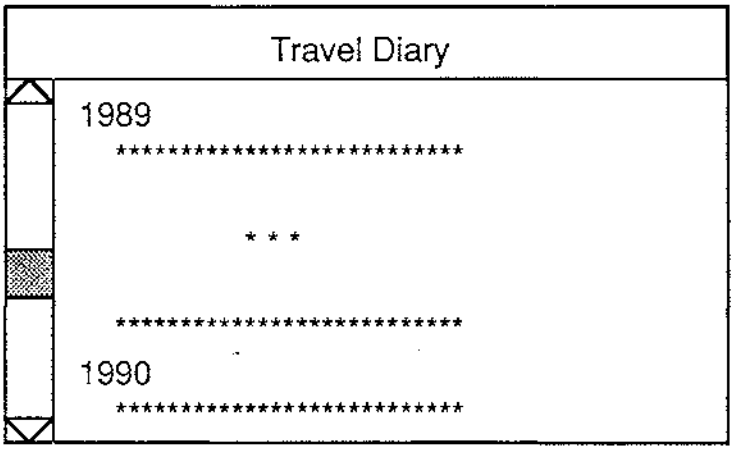
(a) A """Bifocal""" Viewing System - Level 0



(b) A "Bifocal" Viewing System - Level 1



(c) A "Bifocal" Viewing System - Level 2



(d) Conventional Viewing Window

Figure 3.1.3 ""Bifocal"" Display

Open the record of year 1989 from the top level abstraction of a travel diary (see Figure 3.1.3(a)). Figure 3.1.3(b) shows the countries in record in the middle viewport. Other years go to two ends. By choosing one to see the detail, other countries go to the ends as well (see Figure 3.1.3(c)). Those years and countries which are not in the central viewport can be easily dragged to the middle when they are required. Figure 3.1.3 (d) shows the conventional viewing window in which the text is in one level. Users can only see the screen size contents.

Like an amplifier, the "bifocal" display amplifies the middle part of an object, while the leaves the two sides as before. No matter where the amplified part is, it always remains in context. It is easy to drag other information to the central viewport. For the object in the viewport, there is detailed information about it, and there are relatively high data abstractions of those details. The information at either end, which is far from the current viewport, can be moved quickly to the central viewport by moving the objects at high level.

3.1.4 "Hypertext"

"Fisheye" and "bifocal" displays are for viewing objects continuously in a sequence. Whilst it is hard to get logical related parts immediatly for a large object.

Large complex programs are different from small simple programs in many ways. They do not just have different amounts of data, but also different kinds of links, display formats and storage formats. They need to consider the screen size, as well as speed and convenience of retrieving data. In the past, the data to be physically linked was put together. To find a particular place, users had to remember line numbers or page numbers, which can be very easy to forget. It is impossible to remember several logic relations of one object or one kind of relation of all associated objects. Generally, users use scrolling to find a place, which is very slow for a large program. Logic links enable users to get the relative information immediately. The links between data are shown in a high level data abstraction which can get rid of very detailed information. It frees users from trying to remember the places of all sorts of information or spending too much time on finding the places which obviously interrupts the user's thinking. Users can concentrate in their tasks.

Figure 3.1.4 is an example which shows the links used in "Hypertext". There are two kinds of windows; one is active in which the contents are generally the indexes, the other is additional. Positioning the mouse pointer on a number in the indexing information for a section and clicking has the effect of overwriting the text in the other window with the chosen section.

```
SECTION : 2
pos_x1 = 200;
pos_y1 = 660;
draw_box(pw, pos_x1, pos_y1, 10);
line(pw, pos_x+10, pos_y, pos_x1, pos_y1);
pw_text(pw, pos_x1, pos_y1-10, PIX_SRC, 0, "Copy");
draw_box(pw1, pos_x1/20, pos_y1/20, 2);
line(pw1, pos_x/20+1, pos_y/20+1, pos_x1/20,pos_y1/20);

USES SECTIONS    USED IN SECTIONS    SEE ALSO
    3  4                  1
```

```
SECTION : 4

 Pixwin *pw;
{
 Pixfont *bold;
 pw_vector(pw, x, y, x+size, y, PIX_SRC, 1);
 pw_vector(pw, x+size, y, x+size, y+size, PIX_SRC, 1);
 pw_vector(pw, x+size, y+size, x, y+size, PIX_SRC, 1);
 pw_vector(pw, x, y+size, x, y, PIX_SRC, 1);

USES SECTIONS    USED IN SECTIONS    SEE ALSO
                          2
```

Figure 3.1.4 The ""Hypertext"" Browser

USES_SECTIONS denotes all the other sections used by this section. Such as the procedure in section 2 in the upper window calls draw_box procedure which is in section 4 in the bottom window. On the other hand, USED_IN_SECTIONS indicates the sections which call the current procedure. The draw_box procedure in section 4 is used in section 2. SEE_ALSO is another link in horizontal. The procedures which have the same kind of functions can be

listed under SEE_ALSO. For example, draw_box function, draw_circle function and so on.

Although "Hypertext" makes it easy to get a related section, it is still difficult to know how all the sections are connected and how to get other sections quickly. According to the experiment of comparing "Hypertext" with the techniques [Monk 1988], it has a significant improvement of performance by using "Hypertext" with a structure map. The user not only can see where to go, but also where he or she is going.

3.1.5 "Folding"

The data necessary to describe the structure and behaviour of a large system is far too complex to be displayed entirely. Data that is necessary for the user is scrolled off the screen and lost. This makes a complex system harder to understand. There is the need to filter out necessary information and only this useful information is shown on the screen.

For a large document which needs to be read in one view, such as a procedure in a program, there is a demand for a large screen. However, the screen can not be unlimited. To process a large document with a limited screen, it is not possible to view all the detailed information simultaneously. It is necessary to cancel some information which is not important at the moment, and make the important information more obvious. ""Folding"" is a technique for hiding some portions of a piece of text, leaving only those sections which are important. ""Folding"" is different from other techniques in that it does not show the unimportant parts at all. ""Folding"" makes the display clear and simple.

""Folding"" also helps users to understand the whole text quickly. For example, each book has a table of contents. Readers always get first feeling about the book from the introduction and the table of contents. The same applies to analysing a program. Figure 3.1.5 shows the procedure names of a program (see Appendix 2).

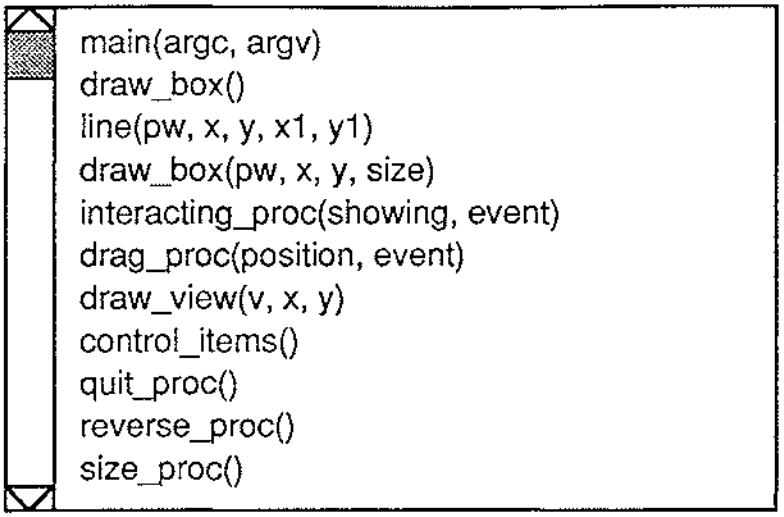


Figure 3.1.5 ""Folding"" Display

The headings of procedures are all taken from the program itself. Showing headings and "folding" detailed contents are not something new. Users find it easy to get use to this. It greatly reduces scrolling time, offers a large range of view and enables the direct manipulation to objects in different levels.

3.1.6 Browsing with Different Techniques

Many display techniques are available, but not all techniques are suitable to all interfaces. Different techniques have advantages or disadvantages with different tasks.

For example, to compare some feature of two countries which are very far away on the map, it is impossible to use scrolling because the map is too big to put in one view. There are also similar problems in using "fisheye" view technique unless comparing between all the countries. The possible solutions are using "folding" or "Hypertext" techniques. By "folding", two countries can be put together physically while with "Hypertext", there can be other windows to meet the needs of particular uses. Also the "folding" technique is not good for reading information which has an effect on other parts around it.

"The different browsers constrain which sections can be simultaneously visible in different ways. The scrolling browser constrains a user to viewing sequentially adjacent sections. A user of the "Hypertext" browser can only view sections which are adjacent in the "Hypertext" space i.e., each of the two sections displayed must have a reference to the other in its 'uses', 'used in' or

'see also' indexing information. A user of the "folding" browser is unconstrained as to what sections are open at the same time. There are also different constraints upon the way a user can move about the program. It might be thought that the "Hypertext" browser will require many more operations to reach some required state than the "scrolling" or "folding" browsers. This is not true because the 'uses'/'used by' hierarchy is very shallow and in addition there are the 'see also' links." [Monk 1988]

Several of these display techniques have been used in the programming environment which is described in the next chapter.

3.2 Interface Design Principles

Many techniques and tools are available for interface design. There are no definite rules because there is no general purpose interface, perfect for all environments. However, some methods are helpful for displaying information in the most efficient way. The ability to express the same kind of data in different ways offers users more flexibility. In addition to those display techniques below, there are some measures which should be available in the system; such as "undo", on-line specifications and so on.

3.2.1 Layout of the Interface

Generally screen size is limited. So consideration must be given to where and how information should be placed and how much can be placed on the screen before it becomes cluttered. Both information and control objects have to be shown on the screen, even further limiting the space available. Conflict is generated by the different kinds of information that might be displayed, and the space allocated to them. This requires careful consideration in designing the layout. In order to show more information, each display may have to be abstracted or only a small part may be seen in each view. One area of the screen may be dedicated to a particular type of display, which can be used for different information scrolling, etc. Some information need not take up static space all of the time, and need not be considered in the basic layout.

Multiple display windows are used on high resolution display screens. These enable users to see many objects or one object in different views at the

same time. No matter whether the information is divided logically or physically, the multiple display window technique can represent it clearly. Windows can be folded, closed or open, and each window can be called quickly by changing the current control.

To make good use of screen space, it should be possible for users to control the layout themselves. The position of an item of information, the size of a window, the different views of one object and the type of representation are variables. They can be adjusted or changed according to the user's needs and habits. Window techniques are currently widely used. They are considered to provide flexibility to users.

3.2.2 Data Abstraction

What an object looks like influences the interface heavily. An icon can represent a system file, a user file or a tool. As abstractions of those data, icons express some meaning of the data though this may not be very accurate, and there are no links shown among data which would be easy for users to see the relationships between them.

Different geometric shapes, colours, line shapes, line widths, light and so on can be used to distinguish objects. It is better to use several distinctions rather than one. For example, if only colour is used to distinguish many different kinds of objects, it is difficult for users to remember the meaning that each one represents.

3.2.3 Type of Information

What the interface looks like depends on what kind of interface is suited to a particular class of user. To experienced users, not all kinds of information need to be static. The basic layout can only consider those that are the most important. The more experienced the user is, the less the static parts in the basic layout are on the screen and the more information can be held in the environment. Static menus are clear and easy, and provide guiding information which is helpful for inexperienced users, but not always necessary for others.

Only dynamic and continuous feedback can immediately meet the requirements of users whilst performing tasks although dynamic information

need not be considered in the basic layout. The appearance point, when an object appears, should be designed to be consistent. On the other hand, these features should be able to be changed, such as repositioning, resizing, giving an appropriate label, and so on.

3.2.4 The User Interface in Applications

The user can not expect to do everything through one interface. Most of the interface is designed for a particular environment in order to make it more powerful in this particular area. This is also easier to implement. No matter what kind of interface it is and what techniques it uses, independence from the practical task is necessary. The division of an interactive system into several layers was suggested by M-R Koivunen [Koivunen, 1988]. With the idea of layers in an interface design, applications become simpler, because no interaction modes or states are needed. Applications programmers can concentrate on the essential algorithms of the system.

Chapter 4

Programming Environments

The problem in creating a programming environment is how to combine multiple techniques and devices to meet the needs of the particular task. An effective interactive environment is one in which a user carries out his or her work with minimal conscious attention to how to use the tools. It should be one easily controlled by a user to do his or her task with maximum effectiveness.

4.1 The Analysis of A Programming Environment

For a structured program, it is easy to show the structure as a graphical view. In addition, to read a program in totally different ways, such as maps or structure charts, the program source code itself should be readable and understandable. No matter what kind of data types or what level of objects, no matter whether it is a variable or an application program, they all have to be meaningful and consistent.

In an Object-Oriented Programming Environment, editors can be designed to highlight the objects in programs (procedures, variable names, or keywords), instead of merely dealing with strings of characters. When writing the source code, a syntax-directed editor can guide the programmer to create syntactically correct programs by offering a choice of permissible structures as templates.

4.1.1 Variables

Meaningful and distinctive mnemonic variable names are helpful in the understanding of a large program. Also, names can be hierarchical, denoting the range used and the function. For example, an external global variable name can be composed from program_name, usage and range, as *program-name_usage_range* . For a local variable, the procedure name can be part of the name.

All variable declarations (object parameters) can either be shown in a symbol table or separately. This is the basic information about a variable. The relations between variables are important because variables' values are often

used for control in a program. An understanding of variables would be a great help in understanding the program. Up to now, not enough attention has been paid to the role of variables by users, except the declarations. A diagram which shows the control by variables would be useful.

In an interactive programming environment, the types or values of variables or constants can be defined or changed at any place during programming; perhaps even during the execution, if this technique is possible in the future. Variables having temporary values at run-time will have more control over the execution. This can be implemented by combining a compiler and run-time controller. Two dynamic arrays are created to contain variables and their values. One is for global variables, the other is for local variables[James, 1987]. This information can be viewed through a window which contains any variables users need to know (see Appendix 9).

4.1.2 Phrases

Phrases are dynamically defined as objects. They can be the component of some statements. For example:

if (*condition*)
    then (*statement*)
    else (*statement*)

Each part i.e. "if (condition)", "then (statement)" or "else (statement)" is called a phrase. Each phrase is regarded as one object. Some editors allow users to input any characters during editing, and point out mistakes in compiling. With a syntax-directed editor, each phrase should conform to the permitted grammar of the particular language. For example, the matched "(" and ")", "else" can not follow "if" but "then" and so on. The strategies of indentation and highlighting the reserved words of these phrases are useful ways to read the source code as shown above. Because the reserved words are highlighted, it is easier to see the feedback of error messages.

A structure diagram can be suitable for showing the structure of phrases in a procedure, as shown in figure 4.1.2.

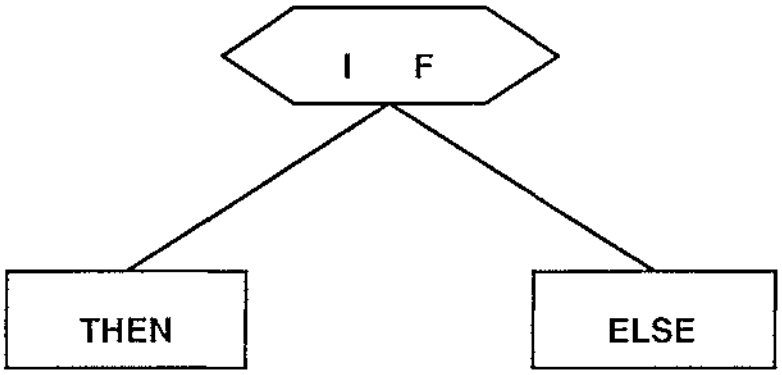


Figure 4.1.2 Structure Diagram for Phrases

The structure in one procedure is different from the whole program; the statements are organized logically. Structure diagrams can clearly describe the logic. Figure 4.1.2 shows one kind of physical implementation. "What makes a diagram look good?". Several factors were proposed by Ding and Mateti [Ding, 1990]. For a large system, complexity is one of the most important factors. For a large program, phrases not only make the diagram simple and clear, but also keep the semantic meanings. A phrase can be selected by double clicking the key word of it (the middle button may be used since double clicking with the left button is generally used for selecting a word if more than one button is available). As in the last example, double clicking "if" with the middle button will cause "if (*condition*)" to be selected. Triple clicking may also select the whole structure which includes the "if" phrase and "then" and "else" phrases. The functions can be acted on a phrase, both in the source code and in the structure diagram.

With a diagram, users can not only easily understand the contents of the procedure but also find out the errors of structure or unmatched symbols easily and clearly. Combining this information with the error message, the user can get the proper feedback which shows the position of the errors.

4.1.3 Procedures

A program is composed of procedures. Program views are visual representations of abstract syntax trees [Moen 1990]. At the top level, each

procedure is a node when the program is shown graphically. Links show the relations between those nodes [Frank, 1987]. Multiple windows enable each procedure to be edited separately. Users will feel safe to act on one procedure without influencing others.

The data in the procedure, such as procedure_name, can be stored in a tree in order to be retrieved and manipulated more quickly. The structure map of the program derived from the tree gives users a structured overview of the program. It helps group members or others to easily understand the program. A good procedure name will have the same usage. It can be defined as *program-name_usage* which is consistent with variable names.

A procedure can be focused on by creating a multiple text window which displays only this procedure. A procedure content can be compressed or extended. Users should be able to get the information about the procedure, such as its main function/usage and where it is used in the program. The information can be shown in text or graphical form.

4.1.4 Comments

Comments are used to help programmers understand parts of the program; generally, a line, a procedure, or a subprogram. Comments can be written along with the program or in a separate file and can be represented in different ways inside a program [Ronald, 1986]. They also can be embedded in the diagram or under a menu. Different users prefer different ways. For example, figure 4.1.2 shows the comments for a line and a procedure. The problem is obvious. The top message separates this procedure from the procedure above, and the comments in the middle separate the lines of the program when the source code line is long. Figure 4.1.3 shows the comments stored and shown separately. Users can store as much information as they like without influencing the source code. The comments in figure 4.1.2 can be associated with a menu or a function key, and also they can be shown in graphics.

```
/* This procedure will locate a position as the top point of a fixed sized
view area by clicking on the screen with mouse */

void
drag_proc(position, event)
    Canvas position;
    Event *event;
{
 int pos_x, pos_y;
 pos_x = event_x(event);              /* Get x cordinate of the event */
 pos_y = event_y(event);              /* Get y cordinate of the event */
 if ((event_id(event) == MS_LEFT) && (event_is_down(event))) {
                                      /* Click with left button */
   if ((view_x != pos_x) || (view_y != pos_y)) {
                                      /* Draw view box if view point has
                                         changed */
     draw_view(-1, view_x, view_y);
     view_x = pos_x;
     view_y = pos_y;
     draw_view(1,pos_x, pos_y);
   }
  }
}
```

Figure 4.1.2 Comments in a Program
(Part of the Program in Appendix 2)

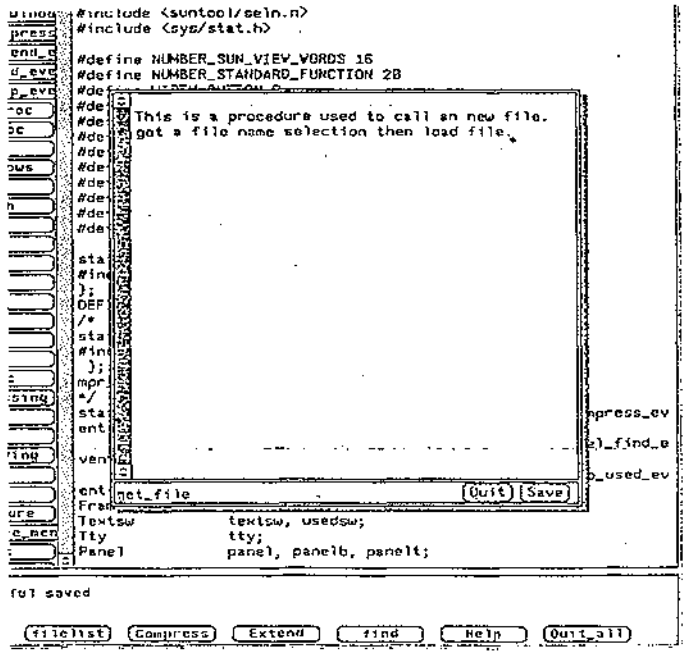


Figure 4.1.3 Comments in Separate File

Also, comments can be a simple specification or a detailed document. This varies according to different levels of information. Parallel to the program, comments also should consider hierarchy, information hiding and data abstraction.

One aspect is usually ignored by programmers, that is the updating the comments. There either should be an indication that changes have been made or they can be updated automatically.

4.1.5 Programs

Generally, a program is a file, and can be processed by the operating system. It is always an independent task which can be called separately. For one task, different programs with different styles enable the more flexible integration of tools. One program with different parameters is another way of file organizing. Which method should be used depends on the application. A structured program can be shown in a tree format or a network diagram.

A processed program can be stored in another file. For example, a compressed program, which contains only procedure headings and parameters, is a brief view of the program. This dynamic information can be used for a quick search of procedure names. Searching is much slower when reading a large program than when reading a small and sorted one. Appendix 3 shows the compressed file of the program used in Appendix 2.

4.1.6 Display Strategies and Comprehension of Different Objects

For each dynamic object there is an inner code. It can be named by combining time in sequence, function code and information level. Showing or hiding is one of the parameters of the object. The parameters of an object define what you can do to the object. For example, a procedure of a program can have some parameters - compressing/extended, multiple text showing, structure chart on/off and the position in the main source code text window.

Procedure names play an important role in understanding the source code. If a procedure is thought of as an object, a good procedure name represents the function of the object. The benefits are more obvious in a compressed text

window and a map composed by procedure names. Reading it is just like reading a brief specification written in natural language.

More and more display strategies are emerging which are convenient to users. Some of them have special requirements for the hardware and software. For example, the more complicated graphics demand a high resolution screen. More complex control needs a good interface. The tools which support the environment are fundamental for program reading.

4.2 High Level Data Store and Processing

4.2.1 Integrated Tools

In a programming environment, the program will be edited, compiled, debugged and executed. The environment is used by many people for different tasks and in different ways. It should support various kinds of tools and views in a consistent manner. When new tools or applications are integrated in the environment, it should have the flexibility to accept them.

Many current systems still use tools separately. For example, the UNIX system is a powerful environment which comprises many tools while all the tools operate on separate files. For example, text editor, compiler and execution tools are used for high level languages, object codes and binary files respectively. The separated debugger and text editor cause an inconvenience in programming.

Generally, programs are made not for programmers but for the compiler. The machine and human understand programs in totally different ways. Some low level language translation can be done by machines automatically, while some high level translations still have to be understood by programmers or maintainers. For example, it is necessary for programmers to know how a program runs; even how a compiler works. It will be very helpful to show this execution in graphic way. Because of the differences between machines and humans, it is not an easy thing to integrate graphic compilation and execution into the environment.

4.2.2 Database

Databases, such as symbol tables, are created while entering data into the environment, and are used to store dynamic data. They enable one to retrieve and process information speedily and systematically. How a database is created heavily influences the speed and memory used. A good database structure should have less information duplication and have a high speed in retrieving and refining data. Different entries enable the database to be used for more purposes.

4.2.3 Files

Some tools can only be used on files such as LEX and YACC (see UNIX Series References). The current editing program has to be saved before using those tools. Of course, some functions used for part of the program are useful and they are fast enough when the process is very simple. These are performed in the memory without file processing which needs to communicate with the operating system. Some processes need to analyse each word, even characters, of the whole program and different patterns result in different actions. In this situation, using LEX to find patterns will be much faster than using macro functions in the memory. For a very large program with very complex processes, the time spent on saving a program a few times is not relatively significant.

4.2.4 Comments

Nowadays designers are paying more and more attention to the use of comments. These are not only used to explain the program, but also used as commands to call other tools at run time. The specifications tend to be executable.

Sometimes the source code has been modified, while the comments related to the code are not changed. So, experienced maintainers do not rely on the comments. They interpret the source code themselves. It is necessary to find a technique that is a reading assistant tool derived from source code. Structure charts or maps provide one approach. They describe the relationships within the source code. They give users a general idea of how the program is organized, but they don't say what the program does.

Another possible solution is suggested by Knuth [Knuth, 1984] who proposed a tool, called WEB, which permits the programmer to extract the source code from an exhaustive description of the design reasoning. This comes close to programming in natural language.

4.3 Current Development of Supporting Tools

4.3.1 Hardware

Choosing the method by which a user selects facilities or features using a particular input device depends on similar factors. The main alternatives are menu driven systems and linear command language specification. Both have their advantages and disadvantages depending on the application. More important, both can be made more effective by the addition of certain facilities such as "undo" or on-line help, and by careful attention to detail. Coloured displays and diagrams have a lot to recommend them. Multiple display windows can also be useful.

Not only the presentation, but also the performance, have to be considered and improved. With improvements of computer technology, interactive programming environments [Barstow, 1984] came out to meet the increasing of complexity and the desire for more productivity. The input devices include a keyboard, all kinds of pointing devices (eg. a mouse with buttons, a joystick, tracker ball, a light pen, a control panel or touch screen), a tablet, graphic input from TV cameras or image softwares, voice recognitions and so on. But the output of computers is much more complex. Although the output devices are generally the display screen and a printer, many things have to be considered in an interactive environment. Input and output devices should be chosen to fit the particular application they are to be used in. This means that the designer must consider the environment in which the equipment is to be used, the kind of users it will have and the kind of task they will use it for.

4.3.2 Real-Time Control

A line interpreter is an application of the context sensitive technique. It gives the fastest feedback of grammar correctness, and it can greatly reduce error

occurrences. When a program is running, the path of it is presented in the map. It shows where the program goes, what one procedure has done, and the changes to the data. In debugging step by step, the high level changes will be shown in the map clearly. Some particular data (eg. global variables) can be changed at run time in order to get the correct result by trying different data without going back to the compiler again. In this situation the program may need to be able to be interpreted. When an error occurs, the illegitimate part is highlighted or an arrow points to the right place.

At a stop point, a verification view, which shows the status of predicates and other necessary information, enables the user to prove correctness while programming. Reverse execution may give more sense and convenience in real time control. The program keeps tracking until an error occurs in the execution, the results of each step will show the cause.

4.3.3 Information Processing

To process a large amount of information, an environment needs to consider proper response speed, data abstraction, structured information display as those mentioned before. Also the environments are getting powerful, reliable and flexible. For a programming environment, the source code has to be readable and comprehensible in different ways.

Hidden information can be shown when the part where ellipses are used instead of detail, is clicked. The ellipsis feature of comment templates provides an incentive to use comments during program development, rather than afterwards. Because it rewards the skilful, precise use of comments, this feature promotes good programming style and method.

At debug time, a symbol table may be created which contains all dynamic values of variables and constants, or even other facilities can be called and represented.

4.3.4 Self Defined and Extended Integrated Environment

There can be many views of a program, general text editor, graphic structured flow chart, syntax trees, symbol table, data type table, control flow chart, execution diagram and dynamic data structure diagram of program

execution and so on. In order to manage the dynamics of multiple access paths and related information, links between different views have to be known to users.

Graphic packages, such as MacDraw and MacPaint are generally used for drawing pictures. They may be used on a document and integrated in a programming environment. The graphic terminals make it possible to act on source code with some functions of MacDraw or MacPaint. But it will be different from when they are used in a picture. For example, partial selection will not select anything in a circle or a rectangle but the objects in it. The objects here are variables, phrases, procedures and programs, not individual character. Within a selection, cut, copy, and font changing can be performed in the same way as processing a picture, except they only operate on objects. This technique could be provided in the future. A program is an object that some macro functions can work on. For example, "map", "fold" and "chart" can be used on program xxx.c. These three functions can be called in any programming environment. Each one gives a different view of the program.

map xxx.c	---	graphic display 1
fold xxx.c	---	generates headings
chart xxx.c	---	graphic display 2

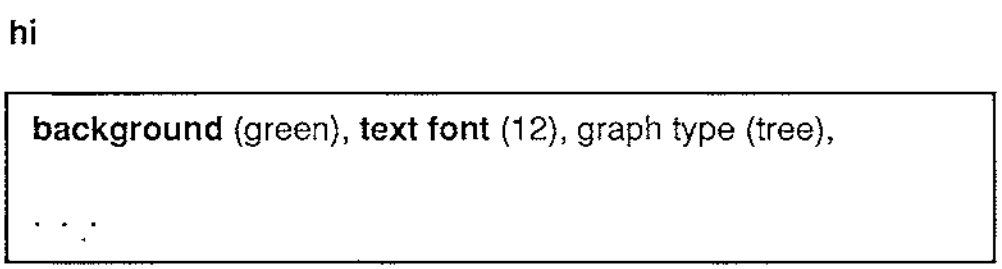
In this case, some application programs made in other environments can be used as tools in this environment if the programs are reusable and compatible. By combining objects and functions in a self defined environment, the work becomes choosing the parameters [Reiss, 1990]. They should be available when text editing, compiling and at run time.

By using comments or high level languages, users should be encouraged to tailor the most suitable specification style to each class of problem being solved, rather than being forced to use a pre-specified language for every kind of application. Thus, the environment should be flexible, either by tailoring or extending a particular tool, or by creating specialized tools to cope with users' particular tasks.

In the past, source code could only be read line by line in text form. Now it can be interpreted using different approaches. All the characters in the environment can be defined by default and changed in a dialogue box or menu.

Users first want to know what they can do before understanding how to do it. Generally, there is a text manual to specify the format and usage of each function, but many users do not have the patience to read it. They need to know the next step within the environment itself and be able to use the environment without referring to something else. So in the environment anything which is not going to be used should be locked automatically and may be shown in dim. This limits the occurrence of errors and gives users an idea that of what they can do next. Below is an example which shows how this technique is used in the environment.

Parameters of the environment can be changed at any moment, at the start point or in the middle of processing. Suppose a user calls an environment from the system menu. All the parameters which can or cannot be changed are shown in a box on the screen. For example, when an environment called "HI" is being created, the box containing the parameters comes out where graph type is unchangeable at this stage. Others may be accessed by scrolling in a cycle and clicking on them.



The user can make changes at any place. Any action outside the box will accept the parameters (they are default if there are no changes) and go into the environment. This box is not only used for changing parameters but also used for showing the features of the environment, that is what a user can do in it. It also frees the user from command line input for parameter setting.

4.3.5 Other Tools

As techniques develop, more and more tools will be used for an interactive interface. It would be helpful to have various tools available on the workstation if the techniques can be supported in the future. For example, voice could be used as an alternative feedback of error occurrence or special position in the environment. Touch screens are not only visual but also physical and will be more attractive with 3D interactive graphics. Also handwriting recognition is more familiar to users when they are doing text modification. There is a trend to more gesture based dialogues which humans are used to.

A stack is one way to store execution history. Users could look at the history from start or in reverse. It could be shown on the map and structure chart.

4.4 Advanced Programming Environments

4.4.1 Problems in Current Programming Environments

In comprehending a program, graphics is easily used for representing the structure, as in Program Visualization. As programs get larger, the picture becomes bigger and bigger, and it looks much more complex than the picture of a small program. Sometimes it is harder to read than the source code itself.

The layout design is very important for an interface, because users may achieve their goals in an inefficient manner if the layout is not well organized. The interface is the channel for users to reach their goals. Yet there is no unique standard for a good interface for all people and all tasks.

There are lots of tools available. But what can be efficiently used in a programming environment? Many tools are implemented just for one particular application. This results in difficulties in combining tools in another environment.

4.4.2 Improvements

There is always a goal before a programmer producing programs. This goal is going to be achieved by running some programs. According to the top-down program comprehension model, a program is understood by reading the specification, understanding the function and guessing the structures. The same methods can be used in programming. The goal of the task can be implemented by steps or by calling some independent functions. Each step or function can be a subprogram or a procedure.

Users should always be aware of all the functions which can be used in the environment. For example, if a user wants to know something about a procedure in the program which is not in the view, what can he or she do? If the user wants to make some changes to this procedure, which way is more convenient and flexible to the user? A document window or a subtext window may be called according to what the user wants. For a very large program, generally users are experienced programmers and maintainers. They are concerned more about convenience than the meaning of each simple function.

The graphics should be able to be zoomed in or out with simple or complex representation. Generally, the graphics can only be zoomed in/out by mapping or it only can be abstracted as a simple representation. By combining two techniques together, less detail can be seen when it is zoomed out and more detail can be seen when it is zoomed in. For the map of a program in one view (Figure 4.4.2), for example, it can only have the nodes and links (Figure 4.4.2(a)). When the graphics is clear enough through zooming in, the nodes can be different which represent different kind of procedures (eg. recursive procedures, first time call procedures and so on). Then the procedure names appear (Figure 4.4.2(b)). By further zooming in, the brief specification of the procedure can be embedded in the nodes, even the contents(Figure 4.4.2(c)).

The environment should look simple and clear but be powerful and convenient. It should be able to handle all possible errors and limit them to the smallest amount. It should offer the opportunities to be extended and modified. It also should be in control at any time and offer the most help to users. In this environment, the user need not know anything about the computer and the system. The whole programming operation can be done inside the environment.

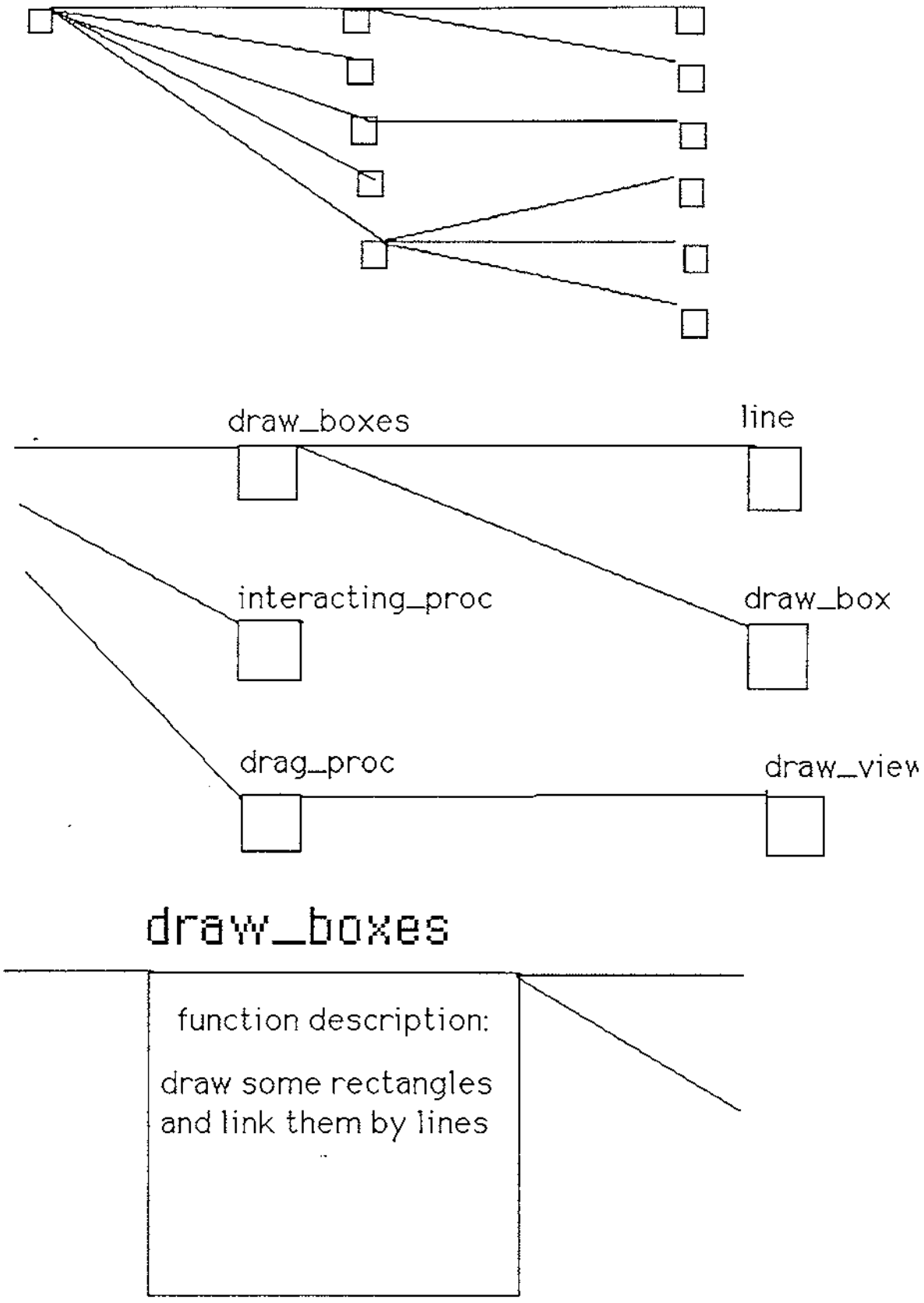


Figure 4.4.2 The zooming of graphics (a)(b)(c)

In a more advanced programming environment, all kinds of input should be available. Command input by menu selection is the most common way in current environments. But menus cannot cover all kinds of users' demands especially when a environment can offer many tools and users want more freedom. Even if all sorts of commands are in the menu, it is still a problem to find a particular one. And to experienced users, menu selection seems too slow to meet the demands. To express users' demands completely, command input is still necessary, as experienced users require it.

In an interactive graphics system, pictures like icons are used instead of text commands or menus. They can be very attractive and more meaningful. A menu is always associated with submenus and can not be deleted whenever it is created, while an icon can be a separated object or function. Many processes can act on it in the same as they do with other objects.

It would be flexible if conversions between textual view and graphics exist at the same time. The system should support the changes from one kind of editing to another easily. The relative modifications should be done automatically, otherwise users will be confused by changing the viewing mode frequently.

If the line interpreter is available during text editing, the changes to the map and structure diagram or other graphics can be done automatically at the same time. This visual feature of dynamic change would have more advantages than static change. The feedback is immediate, and the influence is limited and easy to tell differences from the original.

Keeping history is a big problem in any system. Generally, "undo" includes the last action or the whole actions since last saving. In the static editing of a program visualization environment, "undo" can also include going back to the state since last update. For a dynamic environment, there is no middle stage. Of course, users can go back character by character. Consider a program, which is different from a document, each line has to be completed, and the graphics is influenced by the explanation of program lines. With a line interpreter, one line can be treated as an unit. All the continuous changes in one line can be grouped as one step of the history. By continue choosing the "undo" menu, users can see the graphics as well as text changes quickly and clearly.

Chapter 5

A Practical Prototype

Using the techniques that have just been described, a prototype programming environment "HI" has been developed. Section 1 of this chapter introduces some features used in the environment such as frames, windows, and menus. Section 2 provides the specification of the interface design. Section 3 shows the analysis of C programs. Section 4 describes the implementation of tools used for program comprehension and section 5 provides some more considerations of the prototype.

HI is a programming environment, designed for large C program work by one or more people, which provides contextual, general and detailed information, and integrates some tools. It is a taste of object program visualization.

HI has been developed on a Sun workstation which has a high resolution graphical display screen with a mouse of three buttons to interact with it. The steps to create the environment begin with interpreting the source code, finding a dynamic storage to store data, creating windows to show the information in multiple kinds of view, then interacting with the environment to get messages and drawing new graphics from a procedure node tree derived from the source code or from the source code directly.

Sun workstations use the Unix system and support several language editing formats like C, Fortran 77 and Pascal. They have Vi and EMACS as editors. Other utilities include *dbxtool, make, source code control system* and lots of application programs and so on. These tools have greatly satisfied users' needs separately, and they do offer a friendly environment to encourage users to learn what benefits they can get from using them or when, where, and how they can use them.

The program is written in the C programming language which was developed in 1973 by Dennis Ritchie [Kernighan, 1978]. He designed and implemented C on the UNIX operating system on the DEC PDP-11. C includes commands to handle strings, files, and floating point arithmetic. Unlike Pascal in

which subroutines are embedded in other routines, C's procedures are written in the same level. But they can call or be called by others. Recursive calls and loops are allowed.

HI is created in SunView (Sun Visual/Integrated Environment for Workstations) [UNIX, 1988] which is a system to support interactive, graphics-based applications. In SunView, the background is the root window, in which multiple windows can be shown at the same time. Each window has a menu associated with it. The features of SunView used in HI will be briefly described below.

5.1 SunView Features

5.1.1 Frames

A frame is the background used to put subwindows together as a unit. It can contain subwindows and other frames as its subframes. It can be viewed as a tree of windows in which the leaf nodes are subwindows; other leaf nodes are subframes (Figure 5.1.1). Subframes are typically used to implement pop-up windows (Figure 5.1.2).

Figure 5.1.1 illustrates the structure of "iconedit" as a tree of windows. Figure 5.1.2 illustrates "iconedit" with its pop-up displayed.

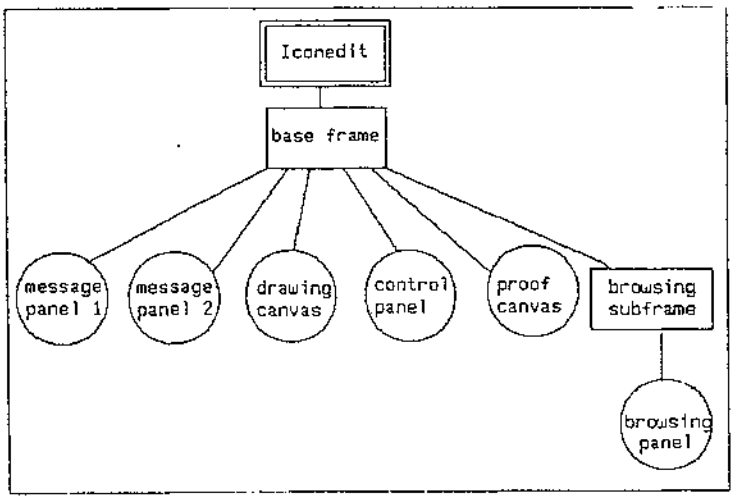


Figure 5.1.1 Structure of Iconedit

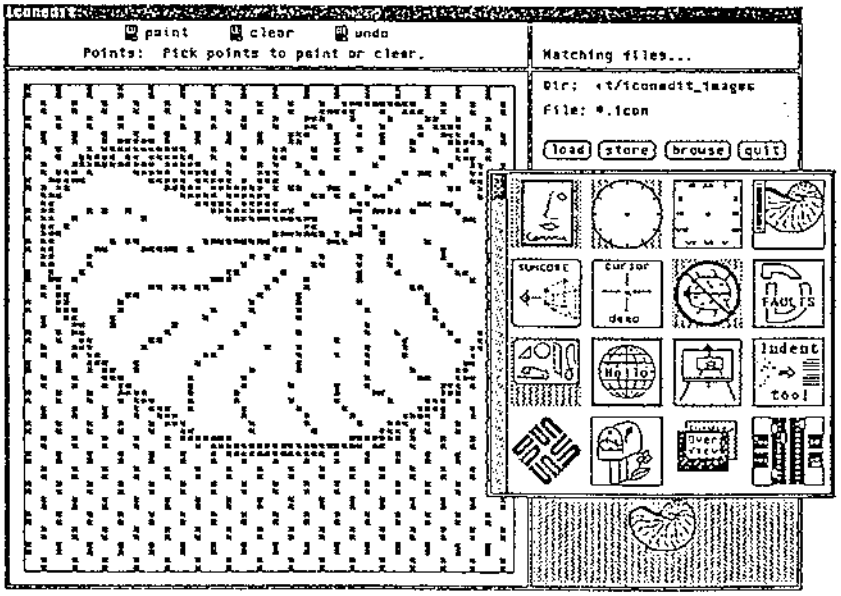


Figure 5.1.2 A Subframe

5.1.2 Windows

1) Text subwindow

There is a main text subwindow and a multiple text subwindow. The content of a file containing the source code is called from the Unix system. The content can be modified, compiled, executed and saved back to the file. In HI, the program compiling and execution act only on the source code in the main text subwindow.

In one text subwindow, one static file is called into memory. Multiple files can be opened and stored into different windows. Between text windows, the contents can be copied.

2) Panel

There are six kinds of items in panels -- message items, button items, choice items, toggle items, text items and slider items. These items are used by the user to interact with the environment.

3) Canvas

A canvas is used to draw graphics. Users can draw points, lines and text. The picture can be very large and viewed by scrollbars. A canvas can be repainted and resized. It enables users to handle their own events and set the colormap segment.

4) Tty subwindow

Some standard routines in the SunView can communicate with UNIX system directly. They include calling a file into the text window, saving the contents of text window to a file and so on. In addition to those limited numbers of routines, a tty window enables users to do anything as they do under the UNIX system.

The tty subwindow emulates a standard Sun terminal, the principal difference being that row and column dimensions of a tty subwindow can vary.

5.1.3 Menu

The SunView menu is the walking menu which contains items. The items can be text or icons. Menus are associated with frames, subwindows, buttons and text editors.

Text input is used when a special function is not available in the menu. The menu associated with it makes the text editing more convenient. For example, when a user wants to find a word or any string which does not appear on this screen, he or she can use the text input without changing the view of the main text window to find the word or string.

5.1.4 Other Facilities

There are many other facilities in the SunView in addition to those mentioned above. For example, input handling, alert, cursor control, icon, scrollbar, selection, the notifier and the event process. They offer great flexibility to users.

5.2 Layout

Some programming environments support editing by dividing the program source code into two parts. One part consists of the headings of the program. This part shows the structure and the abstraction. The other part is the program contents. Generally, the window with headings is above the contents window. On a size limited screen, users can only see about 15 lines of the source code. The user has to change or browse quite frequently to see other parts. If some other functions have to be shown, the real editing part of the window will be less.

To solve this problem, either a large sized screen is needed, or the way of showing headings must be changed. Some environments put headings in several levels. This makes it easy to see the structure and for each subroutine, it is clear to see the path called, and possible to interact with any parts in each level. But problems still exist. Experienced programmers always give a heading a meaningful name which is relatively long. It is difficult to see complete names if different levels of headings are put horizontally. Even if it can be done like this, the contents window will only take the left half of the window for source code, while the right half is empty or contains just a few comments.

In order to make good use of the screen space in HI, it is divided into three parts. The main frame always exists in the environment. The other two parts will be shown when they are needed. Figure 5.2.1 shows the top level layout.

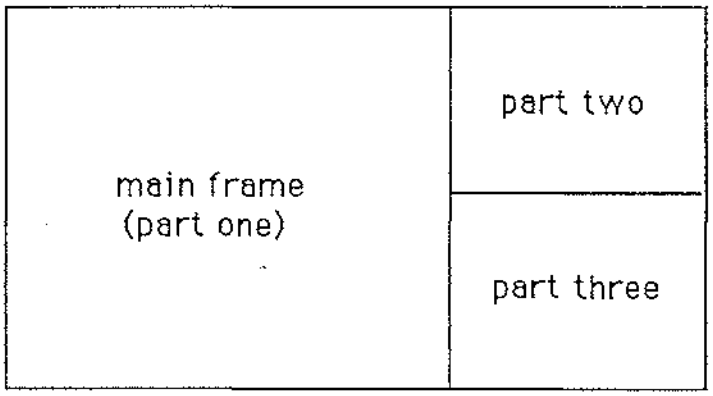


Figure 5.2.1 Top Level Layout of the Display

In the main frame (Figure 5.2.2), the headings are listed in text sequence in a separate window at the left of the text window. It gives users a sense that the contents in the text window are abstracted in the left window and by interacting with headings, users can see the relevant part of the source code in the text window immediately. The text window is still the normal size which is used for presenting the source code as with other single window screens.

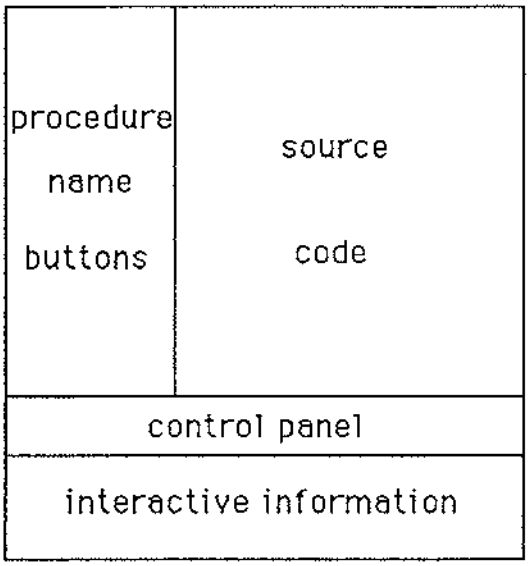


Figure 5.2.2 Main Frame

The display of the whole screen is consistent. Anything that is an additional view of the program is shown on the right of the screen. Anything which communicates with the file system can be done in the TTY window at the bottom of the screen. Anything that acts on procedures can be called in the panel on the left of the screen which shows all the procedure names. All other functions, messages and controls are in the control panel, which is just below the main source code editing text subwindow. The static main source code window gives users a stable feeling. From the static control panel and static procedure names panel, a user can reach anywhere in the environment.

5.3 Objects of the Source Code

5.3.1 Variables/Constants

The variables in C can be local, global and external. They are the essential objects. It is very helpful if the programmer or maintainer can find the information about the variable or constants immediately. The information includes declaration, usage, links and initial value. In HI, the declaration is used not only for local/global variables, but also for other objects like standard functions of SunView, reserved words of C and procedure names defined by the user.

5.3.2 Phrases

A group of statements, and those beginning with reserved words at the top level, are defined as phrases. For example, each box below contains one phrase:

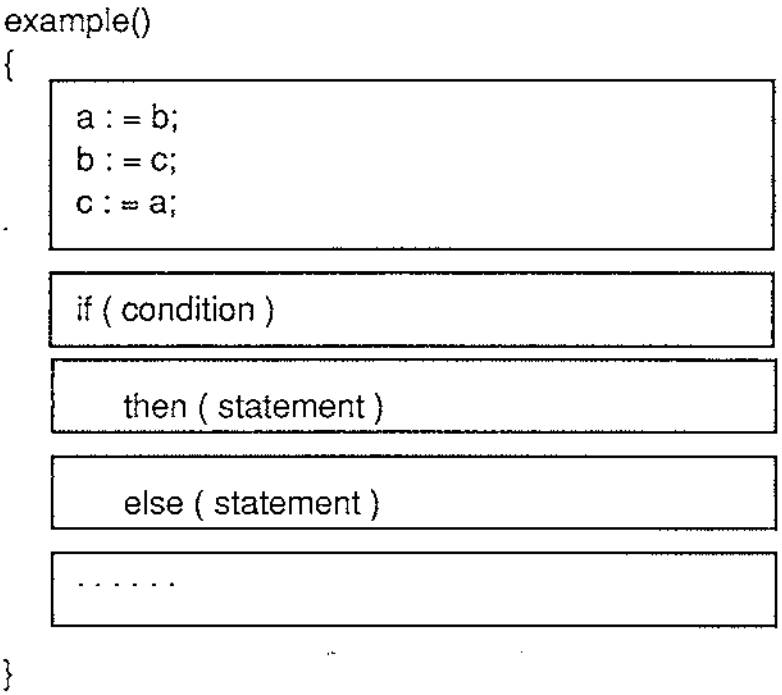


Figure 5.3.1 Phrases in the Source Code

The condition content of "if", the "then" statement and the "else" statement are all called phrases. In the prototype, they are the top level in the procedure.

5.3.3 Procedures

The program in C has one main() procedure. The execution starts and ends in it. There are two ways to view the procedure names. One is in program text sequence which is listed as buttons in a panel in HI; the other is in procedure call structure which is represented as the map in HI. Both are derived from the source code and they have different interactions with users.

On a Sun workstation there are three buttons on a mouse. When a procedure is chosen as the object, a different action can be associated with each button on the mouse. The left shows a menu which contains those procedure names that call this procedure. The middle shows a menu which includes all procedure names that are called by this procedure. The right one contains all the commands to a procedure in a menu, like positioning in the text window, compressing, extending, showing it in multiple subtext window, showing the structure chart, showing the document associated with this procedure and other functions such as deleting, cutting and so on. Some of these functions have been implemented in the prototype.

5.3.4 Files

C is a compiled language. In HI, it is simply interpreted for some special usage.. In order to display particular information to meet the user's need and to support tools in the environment, the source code is interpreted and the high level data abstract is stored dynamically in a more convenient form for reuse.

LEX is used to find patterns of procedures through the whole program. It includes normal C procedure calls and special calls in SunView. For example, notify procedures and event procedures. In the process of LEX, procedure names and some parameters associated with each procedure are recorded in the form of an Avl tree which is used to draw the map of the program and show the related parameters in the map. It is easy to enhance this by adding more patterns or more processes.

There can only be one program being edited at one time in the environment. Changing to another file can be effected by selecting the menu which is embedded in a button in the control panel. The file name can be given by selecting a name in the included files or selecting a file in the tty window or

giving a file name in the text edit of control panel. The current file has to be saved if there are any changes in the text window.

5.4 Macro Functions

5.4.1 Compress / Extend Procedures

The content of each procedure can be hidden with the procedure name and parameters left. Ellipses ". . ." are used to represent the hidden content. There is a sign "c" at the left of each compressed procedure name in the procedure name panel which shows the compress state. When it is extended, the sign will be erased. A message will be given when the action is finished or to tell the user what to do (see Appendix 4).

Procedures can be compressed or extended separately (see Appendix 4) or together (see Appendix 5). Before the current program is changed to another, it has to be saved if there is any change with the program. All the procedures need to be extended before saving, otherwise the contents will be lost. The headings (e.g. program with all the procedures compressed) without parameters are stored in a separated file.

5.4.2 Multiple Subtext Window

There is another window to view text (see Appendix 6), but it contains only one procedure at a time which is chosen from the procedure name panel. With this multiple subtext window, users can read or modify another procedure without changing the view in the main program. It frees the user from context switching.

Another benefit is that the multiple window can be used as temporary backup of a procedure content(see Figure 5.4.2). When a user makes some changes in the procedure in the main text window, such as from MS_LEFT to MS_RIGHT, then compiles the program and runs it, if it's not what the user wants, he or she can change back to the original procedure contents by saving from multiple subtext window (additional window) to the main text window. Normally, programmers make changes in many places within a section, and it

can be difficult to remember them all. The additional window offers users the convenience of comparing and backup.

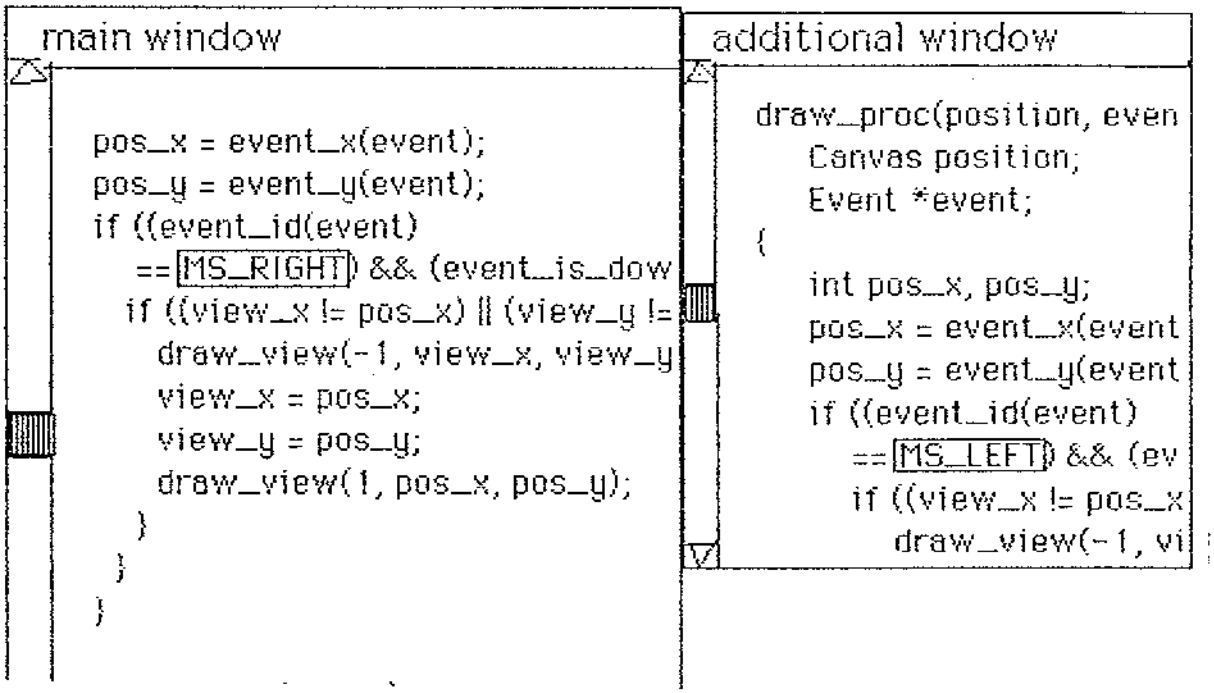


Figure 5.4.2 Multiple Text Subwindow for Backup

In HI, there is only one multiple text subwindow, which is easy to implement and easy to manage, though it is not powerful. It could be improved by creating more than one multiple subtext window, then users could read several procedures at the same time if the screen was big enough.

5.4.3 Structure Diagram

This is the top level structure diagram (see Appendix 6). It enables programmers to see the content of a procedure in graphical way. The reason for drawing only one level is because the purpose of the structure diagram is not to read each statement, but to get the main idea of the procedure. If each box represents one statement, it may be more complex than a well formatted program itself. All the information potentially exists, and can be called when needed. This makes the interface more clear. For example, pressing on the 'if' box, you can see the condition; pressing on 'then' or 'else' you can see the contents. The diagram embedded comments show when the mouse is pressed on any box and

disappear when the mouse button is released. Figure 5.4.3.1 shows the contents of "then" box when it is clicked.

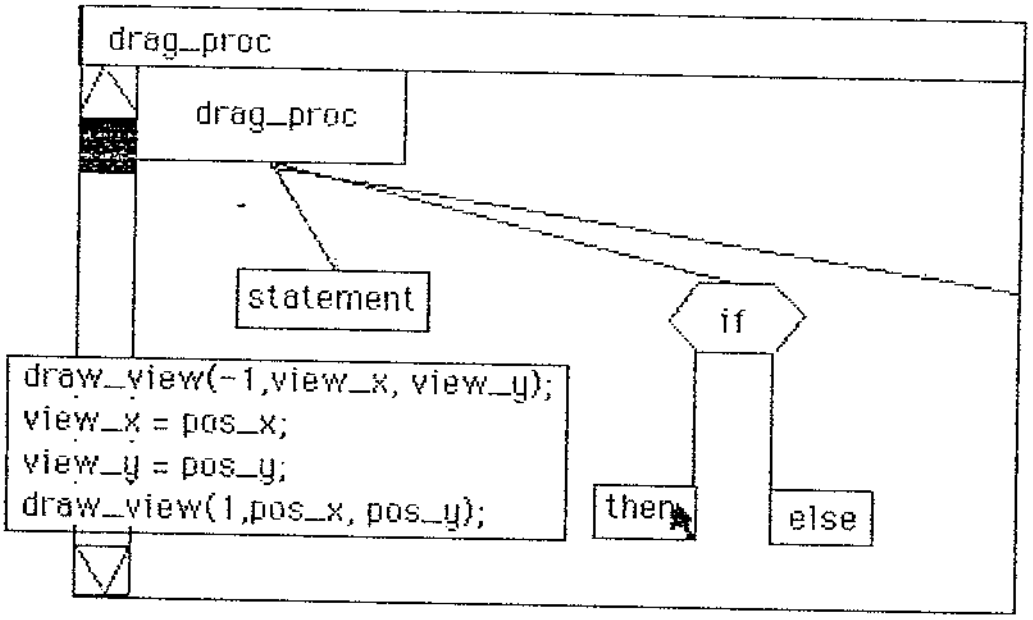


Figure 5.4.3.1 The Contents Showing in Structure Diagram

Some structure diagrams have been implemented which include the visual representation of locating and linking nodes according to the sorted key automatically. Up to now, they have all been very simple exercises, only used for a few nodes. When the diagram becomes larger, speed becomes very slow, which makes practical use impossible.

To avoid this, some statements are put in a group which is embedded in one box in the diagram. The contents of the group will be shown when the box is chosen (see Figure 5.4.3.2). All the information potentially exists and can be called when needed. It makes the interface more clear. For example, pressing on the 'if' box, you can see the condition; pressing on 'then' or 'else' you can see the contents. The diagram embedded comments show when the mouse is pressed on any box and disappear when the mouse button is released.

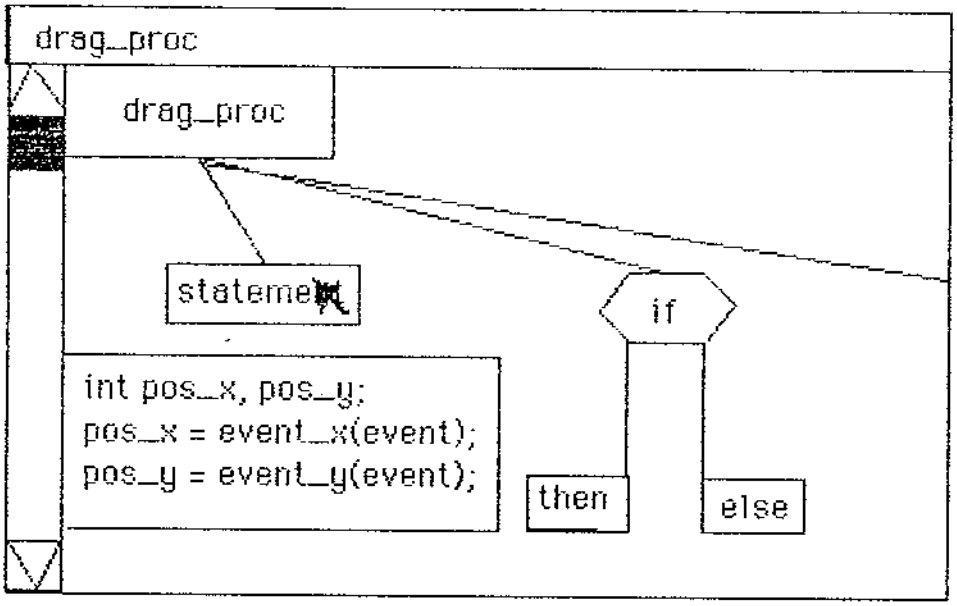


Figure 5.4.3.2 The Contents in Statement Box

5.4.4 Document

The document is used as a tool for programmers to make comments about procedures, such as a key idea, the main function, an important change, and so on. In fact anything the user wants to remember or which might be helpful in the future (see Appendix 7).

For a small program, it's easy to view the program by browsing and jumping to a particular place through the page up and down facilities or locating by line number if the position can be remembered. In a very large program, many routines or functions may have been made long ago. Keywords may be used in many places and line numbers are always changing. It is hard to find a particular function without a higher level browser. The document can help the user to find the information required quickly. Programmers need not know where the document will be stored as it is hidden. Whilst in the past, comments were written in the program, these made the program too long to read. All the documents are stored in one file which is created when program is called for the first time.

The document provides one kind of comment in HI. There is the common comment i.e. /* *comment* */ which is embedded in the source code at any point. Another kind of comment is the declaration of variables, reserved words and so on.

5.4.5 Map

The map shows the whole structure of the program (see Appendix 8). Each box denotes one procedure. It is not just another view of source code, but its importance is that it allows the user to see the relationships between data. The user can find out the links between procedures, and how they call or are called. Also the user can find where a variable or a pattern is used. The map of Appendix 9 shows variable "pw" is used in procedure "draw_boxes()", "line()" and "size_proc()".If the program is not well structured, this can be seen from the map. For example, in figure 5.4.1 most of the procedures are only called once.

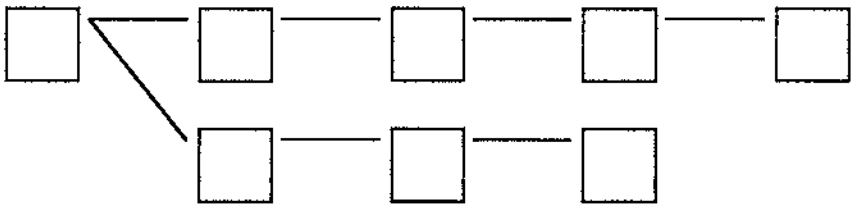


Figure 5.4.5.1 A Tree Structure

Although a tree is more understandable, a network diagram can represent the procedure calls more clearly without redundancy, even though processing the links between procedures is a complex algorithm.

In the prototype, a tree notation is used. The map is drawn from the left of the window to the right. For those procedures which have appeared on the left it will stop processing, and there is a sign to denote the repeat appearance(see Figure 5.4.5 2). This notation avoids loops in procedure calls. For recursive calls, there is a line connected with itself as shown in figure 5.4.5.3.

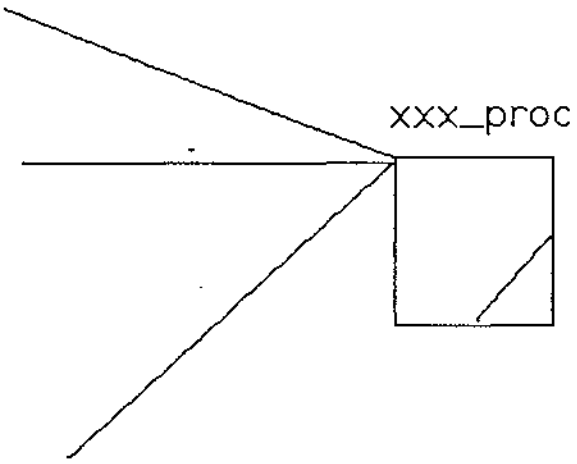


Figure 5.4.5.2 Processed Procedure

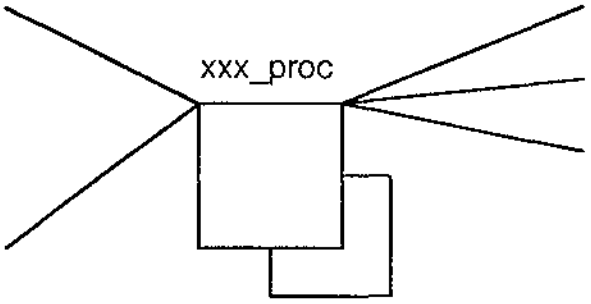


Figure 5.4.5.3 Recursive Call

The map can be very large. If so, users may have difficulty in finding where they are. A small window containing the whole diagram is used to indicate the position. The rectangle in the small window shows the part of the map appearing in the large window (see Appendix 10). As the user changes the program, the original map is still shown on the screen until it is updated.

5.4.6 Interacting with the File System

The TTY subwindow is used to communicate with the Unix environment. In this window, users can load a new file, or compile the current file and run it. Users can also call the debugging tool, though the debug window and text edit window are not the same.

5.5 Possible Improvements

5.5.1 Speed

All the data processed in the environment is in memory. This enables fast processing. From the users' point of view, producing the map takes the longest time. A small program with tens of procedures will take a few seconds from collecting information to drawing the map, whereas a large one with hundreds of procedures will take tens of seconds to create the tree and store the information. However, it takes only few seconds to draw a map no matter how complex it is. Also some time is spent on creating the window, except when the window has already been created.

To accelerate processing, the tree is changed from a binary tree to an Avl tree. An Avl tree is a balanced tree. It is much faster than binary tree for searching. The improvement is obvious in a large program.

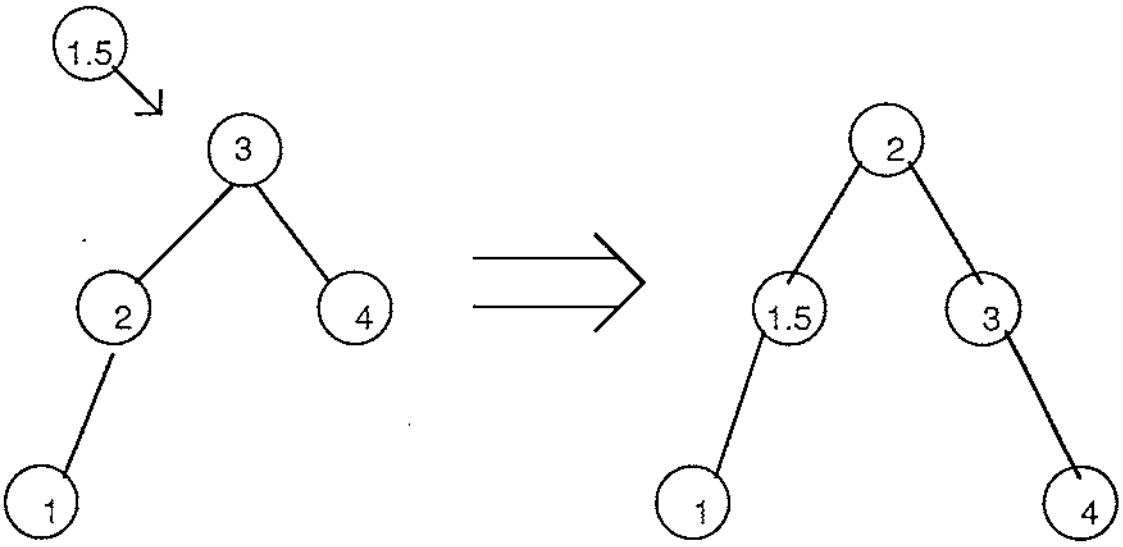


Figure 5.5.1 AVL Tree

An Avl tree is still not fast enough. A Btree would be much better, but the algorithm is complex.

Each time the source code changes, the whole map has to be updated and the tree has to be recreated even though most parts of the tree remain the same.

The solution is to change the subtree where procedures are modified. The modification doesn't include the contents of each procedure but the procedure names. For example, name changing, adding new procedures and deleting procedures. Changing the contents may influence the links between procedures. So, to partially change a tree in a short time needs a more complex algorithm.

When an entire large program needs to be processed, the using of LEX is more efficient than processing by program with SunView's functions. Because it needs to save the source code in the text window to a file before doing lexical analysis, it is not useful for a small program or for simple retrieval.

5.5.2 Information Feedback

1) action --> message --> adjustment of action

Feedback messages (except for virtual actions) are always at the same place in the static window if there should be a message from the environment to the user. The message indicates what the user has just done if it was correct or what he or she should do if it is wrong.

The changing of a procedure's contents will influence its structure chart, while the changing of a procedure's heading (create an new one, delete an old one or modify the heading) will affect the map and its links.

2) program --> data abstraction --> well structured

The structure chart and map are derived from the source code. From the structure of the procedure or the map of the program, it is perhaps possible that users may be able to judge if the code is well designed. How this will be done is not known at present.

5.5.3 Some Specifications of the Prototype

If the procedure is not complete, for example, there is no matched "{" and "}", the procedure heading reading will stop there and give a message to denote the incompleteness. The top level phrase can be shown in the structure chart, but there is no function acting on them.

In SunView, grammar sensitivity is not available and font or style cannot be partly changed in the text window. Only the content in text and subtext

window can be changed while the canvas window used for drawing structure chart or map can only be modified through the changing of the contents in main text window. Also, they will not be redrawn until the user chooses an update function. If the graphics need to be sensitive to the text, the text window has to support the grammar sensitive text editor. In SunView, only one place is highlighted. There is no direct application for simultaneous multiple selections.

HI, a taste of an integrated interactive graphic programming environment for C, is not just a graphic environment to represent the C program source code, but is also efficient and contains most of tools for programming in a simple and clear HI display screen.

The debug tool is separate from the text editor at this stage. To integrate them in the future, part of the compile and debug tools have to be modified. These tools can be integrated into one window. During debugging, the value of a variable or expression can be displayed dynamically or in a static window. Users can see the changes of the values by running the program step by step. Users may even be able to change the value in the symbol table created in compiling, then users need not recompile the source code again and the program can continuously run with the current values.

Chapter 6

Conclusion

This thesis has discussed the environment required for very large program development. It has also discussed some of the current display techniques used for human computer interaction. It suggests what and how these techniques can be used in a programming environment. According to the requirements of programmers and large programs, a practical prototype of the programming environment HI has been created by applying some of these display techniques.

Program Visualization has become more and more important for programming and maintenance along with the further use of computers in different applications. The programs are getting more and more complicated. This causes difficulties in every stage in software life cycle. Program Visualization will help programmers to work efficiently in programming and maintenance.

Graphics, which is easy to understand by humans, can be better to represent the ideas of users. Graphics used in Program Visualization will help programmers to understand the program, especially when the data type is too complex to understand in a linear manner. Graphics is not only used for experienced programmers with large and complex programs, but also can be used to teach programming at the first stage. By reading the graphical display of a program, it is easy to introduce programming methodology which helps users to learn and develop programs more quickly.

Now, a programming environment means not only the text editor of the source code, but also the tools supporting the programming. HI attempts to integrate some tools in the environment. There are several aspects which need to be enhanced in the HI environment. The debugger and editor can be in one view, the graphics can be updated along with the changing of the source code, the method of storage may be improved, the environment can be enhanced for different languages, the graphics can have more different ways to display, and current display techniques can be better used for a better user interface design. HI shows the way to apply current techniques for a good programming environment.

Obviously, a good interactive integrated graphical environment is very helpful in program design and maintenance. However, representing source code in graphical way, interacting with source code and integrating tools have their own problems. For example, the complexity, the speed, and the problems of understanding the different graphics, the problem of integrating different tools in one view and the efficiency of interaction for a particular task. These problems can be solved with careful considerations and better applications of advanced techniques.

If the prototype is further developed in ways such as those that have been suggested, it will become a more useful tool for programmers. However, better understanding of the programmers themselves will better enable more effective tools to be developed to help programmers to read programs.

References

[Agrawal, 1990] Hiralal Agrawal, Joseph R. Horgan: "Dynamic Program Slicing" ,*Proceedings of the ACM Sigplan'90,* pp 246-255.

[Alexander, 1987] Heather Alexander: *Formally-Based Tools and Techniques for Human-Computer Dialogues*, Ellis Horwood Limited, England.

[Apperley, 1982] M. D. Apperley, R.Spence and I.Tsavaras: "A Bifocal Display Technique for Data Presentation" *Eurographics'82* , pp 27-43.

[Baecker, 1986] Ronald Baecker, Aaron Marcus: "Design Principles for the Enhanced Presentation of Computer Program Source Text", *CHI'86 Proceedings,* April 1986, pp 51-58.

[Barstow, 1984] Edited by David R. Barstow, Howard E. Shrobe and Erik Sandewall: *Interactive Programming Environments,* McGraw-Hill Book Company.

[Basili, 1982] V. R. Basili and H. D. Mills: "Understanding and Documenting Programs", *IEEE Transaction on Software Engineering,* pp 270-283.

[Bigelow, 1987] James Bigelow and Victor Riley: "Manipulating Source Code in Dynamic Design", *Hypertext'87 Papers,* pp 397-408.

[Briot, 1989] Jean-Pierre Briot and Bierre Cointe: "Programming with Explicit Metaclasses in Smalltalk-80", *OOPSLA'89 Processings,* Oct. 1989, pp 419-431.

[Chen, 1990] Yih-Farn Chen, Michael Y. Nishimoto and C. V. Ramamoorthy: "The C Information Abstraction System", *IEEE Transactions on Software Engineering* Vol. 16. No. 3.

[Conradi, 1986] Edited by Reidar Conradi, Tor M. Didriksen and DagH. Wanvik: *Advanced Programming Environments,* Proceedings of an International Workshop, Trondheim, Norway.

[Cordy, 1990] James R. Cordy, Nicholas L. Eliot and Michael G. Robertson: "Turing Tool: A User Interface to Aid in the Software Maintenance Task", *IEEE Transaction on Software Engineering*, Vol. 16, No. 3, pp 294-301.

[Ding, 1990] Chen Ding and Prabhaker Mateti: "A Framework for the Automated Drawing of Data Structure Diagrams", *IEEE Transaction on Software Engineering*, vol. 16, No. 5.

[Frank, 1987] Frank G. Halasz: "Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems", *Hypertext'87 Papers*, pp 345-364.

[Furnas, 1986] George W. Furnas: "Generalized Fisheye Views", *CHI'86 Proceedings*, pp 16-23.

[Glinert, 1990] Ephraim P. Glinert, Marc H. Brown and Brad A. Myer: "Introduction to Visual Programming Environments", *ACM Conference on Human Factors in Computing Systems*.

[Goldberg, 1983] Adele Goldberg and David Robson: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Company.

[Halasz, 1987] Frank G. Halasz: "Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems",*Hypertext'87 Papers*, pp 345-364.

[James, 1987] James Bigelow and Victor Riley: "Manipulating Source Code in Dynamic Design", *Hypertext'87 Papers*, pp 397-408.

[Kaehler, 1986] Ted Kaehler, Dave Patterson: *A Taste of Smalltalk*, W.W.Norton & Company, New York, London.

[Kernighan, 1978] Brian W. Kernighan, Dennis M. Ritchie: *The C Programming Language*, Bell Laboratories, Murray Hill, New Jersey.

[Knuth, 1984] Donald E. Knuth: "Literate Programming",*The Computer Journal*, Vol. 27, No. 2, pp 97-111

[Koivunen, 1988] M-R. Koivunen and M. Mantyla: "HutWindows: An Improved Architecture for A User Interface Management System", *IEEE Computer Graphics & Applications,* Jan 1988, pp 43-52.

[Marshall, 1987] Chris Marshall, Bruce Christie, Margaret M Gardiner: "Trends in the Technology and Techniques of Human-Computer Interaction", *Applying Cognitive Psychology to User-Interface Design*, pp 286-312.

[McDonald, 1989] John Alan McDonald: "Object-Oriented Programmingfor Linear Algebra",*OOPSLA'89 Processings,* pp 175-183.

[Moen, 1989] Sven Moen: "Drawing Dynamic Trees", IEEE Software, July 1990, pp 21-28.

[Monk, 1989] Andrew Monk: "The Personal Browser: A Tool for Directed Navigation in Hypertext Systems", *Interacting with Computers,* Vol. 1, No. 2, pp 190-197.

[Monk, 1988] Andrew F. Monk, Paul Walsh and Alan J. Dix: "A Comparison of Hypertext, Scrolling and Folding as Mechanisms for Program Browsing", *People and Computers IV,* pp 421-435.

[Olsen Jr., 1983] Dan R. Olsen Jr., Elizabeth P. Dempsey: "Syngraph : A Graphical User Interface Generator", *Computer Graphics,* Vol 17, Num 3, July 1983.

[Palmer, 1987] T R Palmer: "GRAPE Programming Environment", *Applications,* Vol 29, No 4, may 1987.

[Rajlich, 1990] Vaclav Rajlich: "VIFOR: A Tool for Software Maintenance", Software Practice and Experience, Vol. 20(1), pp 67-77.

[Raskin, 1987] Jef Raskin: "The Hype in Hypertext A Critique", *Hypertext'87 Paper*, pp 325-330.

[Reiss, 1984] Steven P. Reiss: "Graphical Program Development with PECAN Program Development System", *Technical Report No. CS-84-04,* Department of Computer Science, Brown University.

[Reiss, 1990] Steven P. Reiss: "Interacting with the FIELD environment", Software Practice and Experience, Vol. 20(s1), June 1990.

[Ronald, 1986] Ronald Baecker: "Design Principles for the Enhanced Presentation of Computer Program Source Text", CHI'86 Proceedings, pp 44-58.

[Seabrook, 1989] Richard H.C. Seabrook and Ben Shneiderman: "The User Interface in Hypertext, Multiwindow Program Browser", *Interacting with Computers,* Vol. 1, No. 3, pp 299-337.

[Shneiderman, 1979] Ben Shneiderman and R. Mayer: "Synthetic/Semantic Interactions in Programme Behaviour: A Model and Experimental Results", International Journal of Computing and Information Science, June 1979, pp 219-238.

[Shneiderman, 1980] Ben Shneiderman: *Software Psychology*, Little Brown and Company, Boston Toronto.

[Shneiderman, 1985] Ben Shneiderman, Philip Shafer, Roland Simon, Linda Weldon: "Display Strategies for Program Browsing", *IEEE Conference on Software Maintenance.*

[Sobelman, 1985] Gerald E. Sobelman, David E. Krekelberg: "Advanced C : Techniques and Applications", *Que Corporation,* Indianapolis, Indiana.

[Sommerville, 1989] I. Sommerville and R. Thomson: "An Approach to the Support of Software Evolution" *The Computer Journal,* Vol. 32, No. 5, Oct 1989, pp 386-398.

[Teitelbaum, 1981] Tim Teitelbaum and Thomas Reps: "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communication of the ACM,* Vol. 24, No. 9, pp 563-573.

[Thimbleby, 1986] H. Thimbleby: "Experiences of 'Literate Programming' using Cweb", *The Computer Journal,* Vol. 29, No. 3, pp 201-211.

[Walker, 1990] John Q. Walker II: "A Node-positioning Algorithm for General Trees", *Software-Practice and Experience*, Vol. 20(7), pp 685-705.

[Watson, 1989] Des Watson: *High-Level Languages and Their Compilers*, School of Cognitive and Computing Sciences, University of Sussex.

[Hypertext, 1987] *Hypertext '87*, The University of North Carolina, Chapel Hill, North Carolina.

[UNIX, 1988] *UNIX Series References, SunView Programmer's Guide 1988.*

Appendix 1

Quick Use of HI

1. Enter:

In the UNIX environment, enter into the HI environment by type in "HI" command with the program name as the parameter. The main frame is created with main text window at the up right, procedure name panel at the up left, control panel below and the help window at the bottom.

format : > hi *filename* <return>

2. Familiar with each static window :

up left panel: procedure name panel

On the up left panel, it is a statement and a button "read and quit first". By clicking this button, users can see the procedure names in the program text sequence. Each procedure name is in one button. The menus are associated with the three mouse buttons.

click with left mouse button:

locate at this procedure in the main text window

click with middle mouse button:

compress the procedure

extend the procedure

click with right mouse button:

show multiple subtext window for this procedure

show structure diagram of this procedure

call document window of this procedure

up right window : main text window

A program is called into this window. Users can do general editing and use the new functions in HI to the program. Those new functions include finding strings or procedure names, watching the declaration of a string, compressing and extending the contents of procedures, locating the display on the screen at a particular procedure, finding those procedures which use the same string and so on.

middle panel : message/control/edit panel

There are three lines in this panel. The first line is the message which gives the feedback of actions. The last line contains some buttons which either are functions which effects the program or used as control in the environment. Panel edit is used for doing some processes to a string which is not available on the screen. It appears at the second line and the processes are implemented with the associated menus. The maximum length of the string is 40 characters.

bottom window : help/list/tty window

At the bottom, it shows the help information at first. In HI, the help information is not complete. It only shows the usage of the up left panel. It should be improved to offer different kinds of help for different area.

Select a string as highlighted, and then choose the "used in" submenu within the "help" button in the control panel, it shows the procedure names which have the same string as highlighted currently on the screen. It returns to the help window by clicking the "cancel" button in the list window.

Click the "filename" button in the control panel, tty window will show below. It is the interface between the HI environment and the UNIX system. Users can do what they want to do in UNIX without quit from HI. For users' convenience, the current program in the main text window can be compiled and executed by choosing the button in tty window.

Note: In list or tty window, cancel should be chosen before choosing another one.

3. Call dynamic windows :

multiple subtext window

Choose "text" submenu in procedure name panel with the right mouse button. A new frame is created on the up right screen with the multiple subtext window in it. In this window, users can do general editing, call structure diagram and save the modified procedure contents to the main text window.

structure diagram window

Choose "structure chart" submenu in procedure name panel with the right mouse button. A new frame is created on the up right screen if there is no frame for the multiple

subtext window. Otherwise this frame will at the down right of the screen. In this frame, two canvas windows are created for drawing the structure diagram of the procedure chosen. One is the detail view with scrollbars. With it, users can read the contents of each box by clicking it and browse the whole structure by using the scrollbars. The other small one is the entire view of the structure diagram. With it, users can see the position of the detailed view.

map of the program

Choose "map" submenu embedded in "help" button in the control panel with the right mouse button. A frame is created on the up right screen if there is no frame for the multiple subtext window. Otherwise this frame will at the down right of the screen. It is the same frame as the structure diagram. The window for map and the window for structure diagram can swap viewing and can be updated.

Interacting with the map, users can see the "used in" procedures of a highlighted string in map by clicking the ? button, those boxes which represent the procedures will be filled in with black.

Click the box with left mouse button, the main text window will positioning the view point at the procedure chosen in map.

For each procedure, the link between it and others is represented by lines which can be shown or hidden. These two states changed by clicking the box with middle mouse button.

4 Quit :

Quit from the dynamic frames

Quit from the dynamic frames by clicking "quit" button in them. With only one integrated frame, it is useful for a multi control system to see more in the background on the screen.

Quit from HI

Click the "quit all" button in the control panel. It exits from the whole environment which includes the main frame and all dynamic frames and returns to the UNIX system.

Appendix 2

A Simple Program for Testing

```
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/panel.h>
#include <stdio.h>

Frame frame;
Panel control_panel;
Canvas canvas, view;
int view_x = 0, view_y = 0;

void interacting_proc(), drag_proc();

main(argc, argv)
    int argc;
    char **argv;
{
 int x, y, view_win_point, control_x;

 printf("Drawing a map...");
 frame = window_create(0, FRAME, FRAME_LABEL, argv[0],
                       WIN_HEIGHT, 505,
                       WIN_WIDTH, 400,
                       0);

 canvas = window_create(frame, CANVAS,
                        CANVAS_AUTO_SHRINK, FALSE,
                        WIN_X, 0,
                        WIN_Y, 0,
                        WIN_HEIGHT, 400,
                        CANVAS_WIDTH, 2000,
                        CANVAS_HEIGHT, 2000,
                        WIN_VERTICAL_SCROLLBAR, scrollbar_create(0),
                        WIN_HORIZONTAL_SCROLLBAR, scrollbar_create(0),
                        WIN_EVENT_PROC, interacting_proc,
                        0);

 view_win_point = (int) window_get(canvas, WIN_HEIGHT) +2;
 view = window_create(frame, CANVAS,
                      WIN_X, 0,
                      WIN_Y, view_win_point,
                      WIN_WIDTH, 100,
                      WIN_HEIGHT, 100,
                      CANVAS_WIDTH, 100,
                      CANVAS_HEIGHT, 100,
                      CANVAS_FIXED_IMAGE, FALSE,
                      WIN_EVENT_PROC, drag_proc,
                      0);

 control_x = (int) window_get(view, WIN_WIDTH) + 2;
 control_panel = window_create(frame, PANEL,
                               WIN_X, control_x,
                               WIN_WIDTH, 300,
                               0);
```

```
  draw_boxes();
  draw_view(1,view_x, view_y);
  control_items();
  window_fit(frame);
  window_main_loop(frame);
  exit(0);
}

draw_boxes()
{
  Pixwin *pw, *pw1;
  int pos_x, pos_y, pos_x1, pos_y1;

  pw = canvas_pixwin(canvas);
  pw1 = canvas_pixwin(view);
  pw_text(pw, 10, 10, PIX_SRC, 0, "Map");

  pos_x = 0;
  pos_y = 500;
  draw_box(pw,pos_x, pos_y, 10);
  pw_text(pw, pos_x, pos_y-10, PIX_SRC, 0, "Main");
  draw_box(pw1, pos_x/20, pos_y/20, 2);

  pos_x1 = 200;
  pos_y1 = 330;
  draw_box(pw,pos_x1, pos_y1, 10);
  line(pw,pos_x+10,pos_y,pos_x1,pos_y1);
  pw_text(pw, pos_x1, pos_y1-10, PIX_SRC, 0, "Getlineline");
  draw_box(pw1, pos_x1/20, pos_y1/20, 2);
  line(pw1, pos_x/20+1, pos_y/20+1, pos_x1/20, pos_y1/20);

  pos_x1 = 200;
  pos_y1 = 660;
  draw_box(pw,pos_x1, pos_y1, 10);
  line(pw,pos_x+10,pos_y,pos_x1,pos_y1);
  pw_text(pw, pos_x1, pos_y1-10, PIX_SRC, 0, "Copy");
  draw_box(pw1, pos_x1/20, pos_y1/20, 2);
  line(pw1, pos_x/20+1, pos_y/20+1, pos_x1/20, pos_y1/20);
}

line(pw,x, y, x1, y1)
    int x, y, x1, y1;
    Pixwin *pw;
{
  pw_vector(pw, x, y, x1, y1, PIX_SRC, 1);
}

draw_box(pw,x,y,size)
    int x, y, size;
    Pixwin *pw;
{
  Pixfont *bold;
  pw_vector(pw, x, y, x+size, y, PIX_SRC, 1);
  pw_vector(pw, x+size, y, x+size, y+size, PIX_SRC, 1);
  pw_vector(pw, x+size, y+size, x, y+size, PIX_SRC, 1);
  pw_vector(pw, x, y+size, x, y, PIX_SRC, 1);
}
```

```
void
interacting_proc(showing, event)
    Canvas showing;
    Event *event;
{

}

void
drag_proc(position, event)
    Canvas position;
    Event *event;
{
 int pos_x, pos_y;
 pos_x = event_x(event);
 pos_y = event_y(event);
 if ((event_id(event) == MS_LEFT) && (event_is_down(event))) {
  if ((view_x != pos_x) || (view_y != pos_y)) {
   draw_view(-1, view_x, view_y);
   view_x = pos_x;
   view_y = pos_y;
   draw_view(1,pos_x, pos_y);
  }
 }
}

draw_view(v,x,y)
    int v,x,y;
{
 Pixwin *pw;
 int i,j;

 pw = canvas_pixwin(view);
 for (i = x; i < x + 21; i++) {
  j = pw_get(pw, i, y);
  pw_put(pw, i, y, v+j);
  j = pw_get(pw, i, y+20);
  pw_put(pw, i, y+20, v+j);
 }

 for (i = y+1; i < y + 20; i++) {
  j = pw_get(pw, x, i);
  pw_put(pw, x, i, v+j);
  j = pw_get(pw, x+20, i);
  pw_put(pw, x+20, i, v+j);
 }
}

control_items()
{
 Panel_item item1,item2, item3;
 void quit_proc(), reverse_proc(), size_proc();

 item1 = panel_create_item(control_panel, PANEL_BUTTON,
                           PANEL_LABEL_IMAGE,
panel_button_image(control_panel, "Quit" , 0,0),
                           PANEL_NOTIFY_PROC, quit_proc,
```

```
                        PANEL_ITEM_X, 20,
                        PANEL_ITEM_Y, 50,
                        0);
  item2 = panel_create_item(control_panel, PANEL_BUTTON,
                    PANEL_LABEL_IMAGE, panel_button_image(control_panel,
                                                          "Reverse", 0,0),
                      PANEL_NOTIFY_PROC, reverse_proc,
                      PANEL_ITEM_X, 120,
                      PANEL_ITEM_Y, 50,
                      0);
  item3 = panel_create_item(control_panel, PANEL_BUTTON,
                     PANEL_LABEL_IMAGE, panel_button_image(control_panel,
"Size",                                                   0,0),
                      PANEL_NOTIFY_PROC, size_proc,
                      PANEL_ITEM_X, 220,
                      PANEL_ITEM_Y, 50,
                      0);
}

void
quit_proc()
{
  window_destroy(frame);
}

void
reverse_proc()
{
  Pixrect *pr, *tmp;
  int yin, yout;

  if (!(pr = pr_open("/dev/fb")) ||
     !(tmp =
         mem_create(pr->pr_size.x, pr->pr_size.y, pr->pr_depth)))
    exit(1);

  for (yin = 0, yout = pr->pr_size.y - 1; yout >= 0; yin++, yout--)
    pr_rop(tmp, 0, yout, pr->pr_size.x, 1, PIX_SRC, pr, 0, yin);

  pr_rop(pr, 0, 0, pr->pr_size.x, pr->pr_size.y, PIX_SRC, tmp, 0,0);
  exit(0);
}


void
size_proc()
{
  Pixrect *screen, *pr_open();
  int height, width, depth;

  screen = pr_open("/dev/fb");
  width = screen->pr_size.x;
  height = screen->pr_size.y;
  depth = screen->pr_depth;
  (void)printf("width = %d, height = %d, bits/pixel = %d\n",
            width, height, depth);
  (void)pr_close(screen);
}
```

Appendix 3

Abstract Program

main(argc, argv)

draw_box()

line(pw,x, y, x1, y1)

draw_box(pw,x,y,size)

interacting_proc(showing, event)

drag_proc(position, event)

draw_view(v,x,y)

control_items()

quit_proc()

reverse_proc()

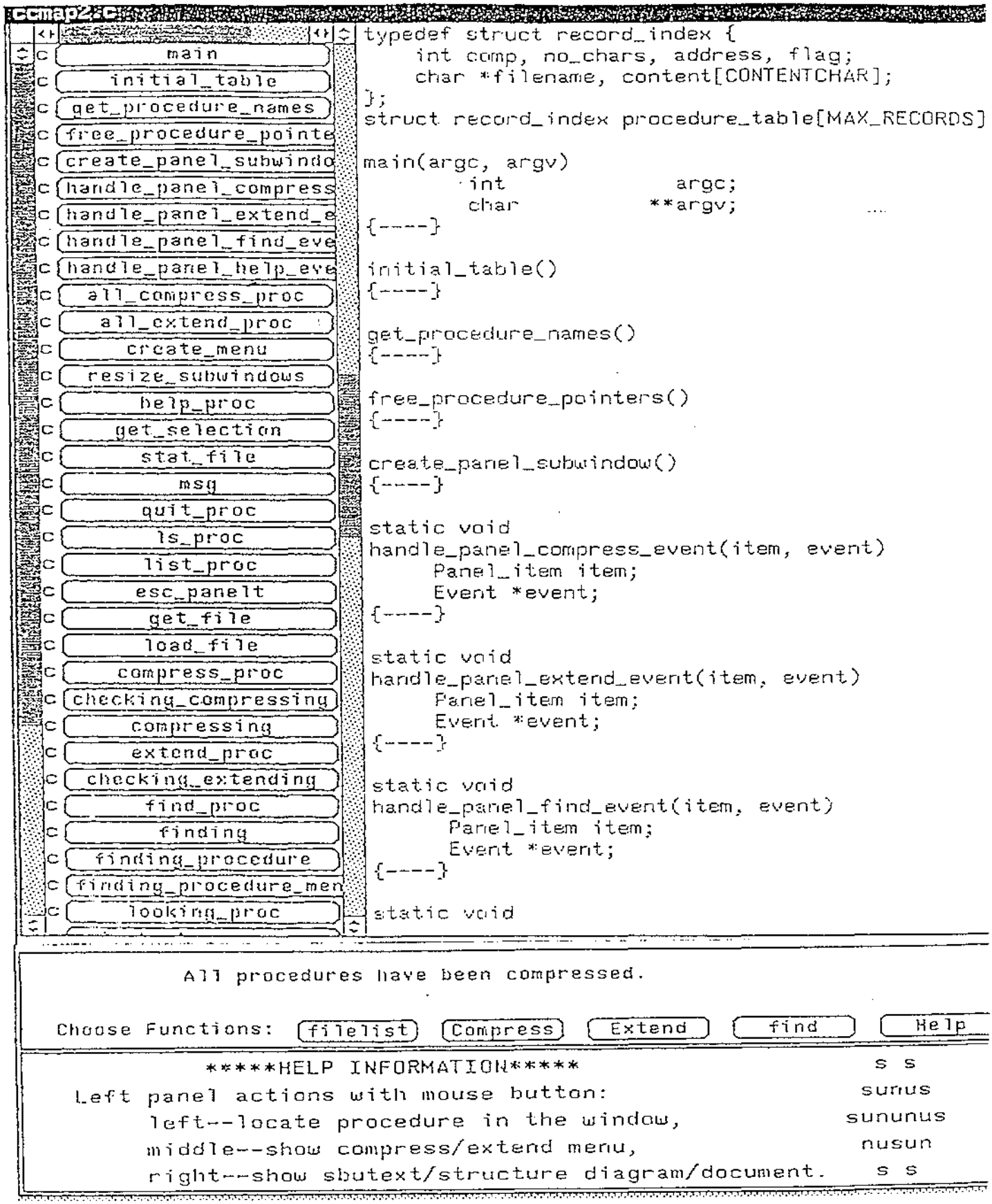
size_proc()

Appendix 4

Compressing One Procedure

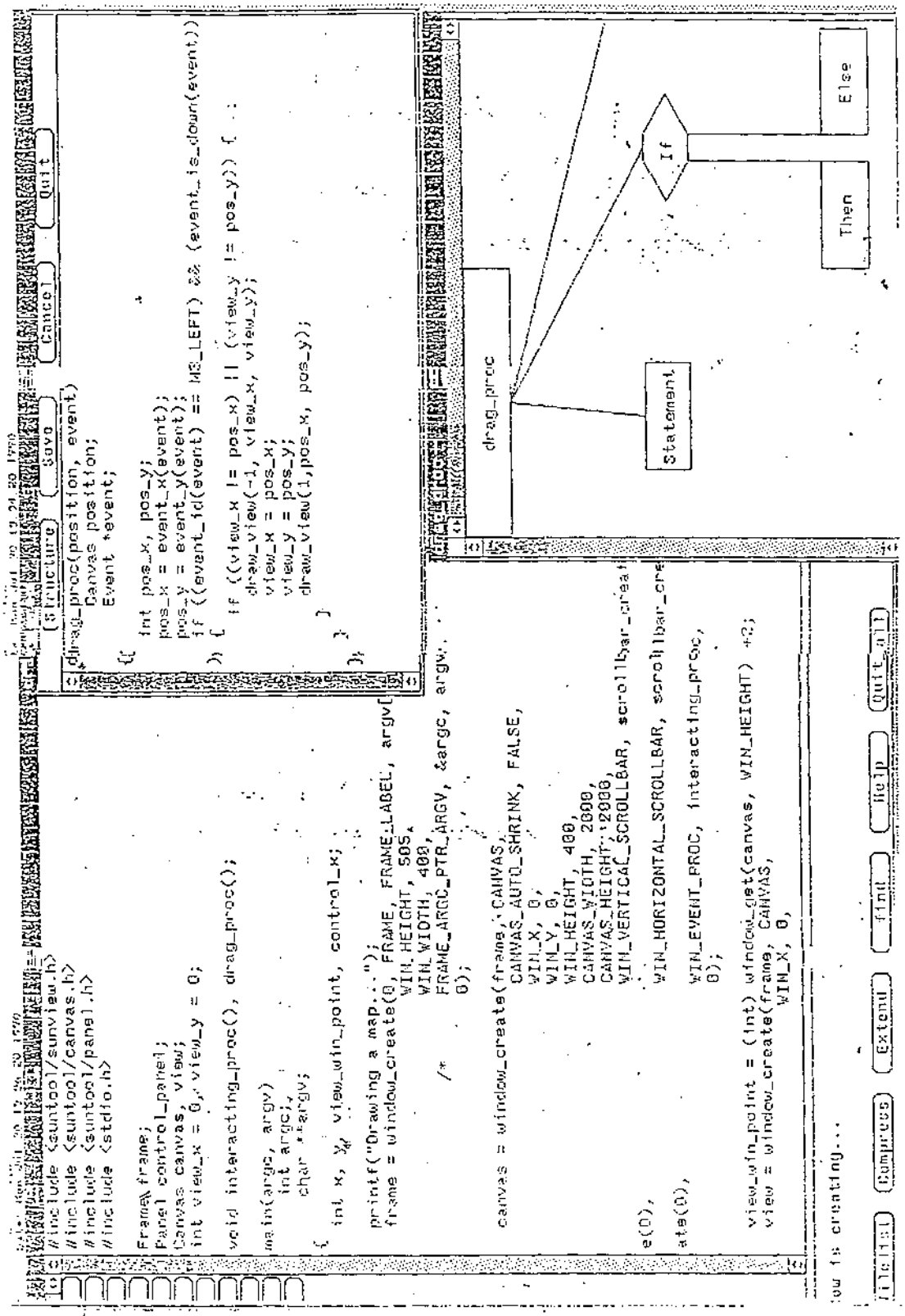
Appendix 5

Compressing All Procedures



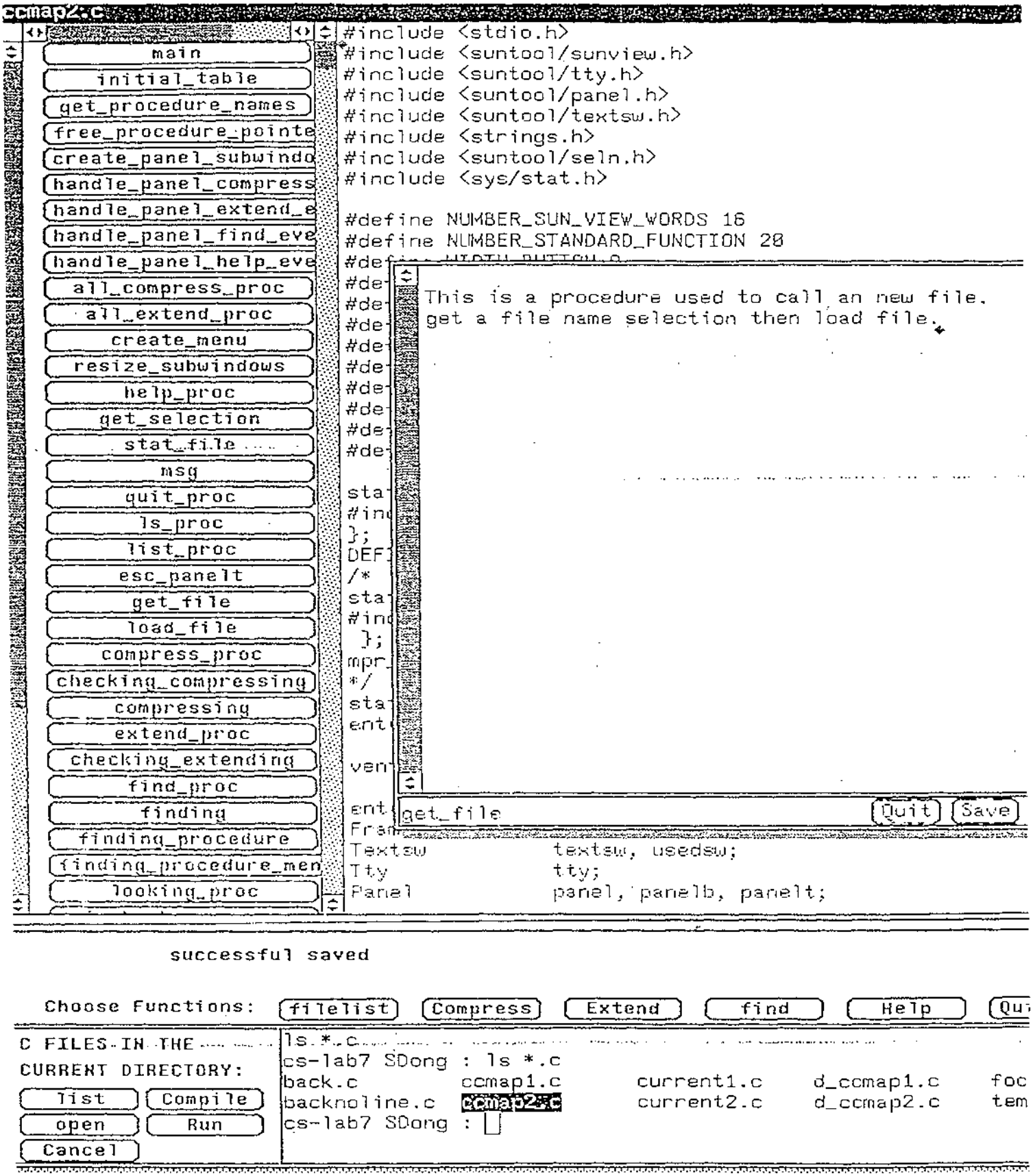
Appendix 6

Multiple Text Subwindow & Structure Diagram



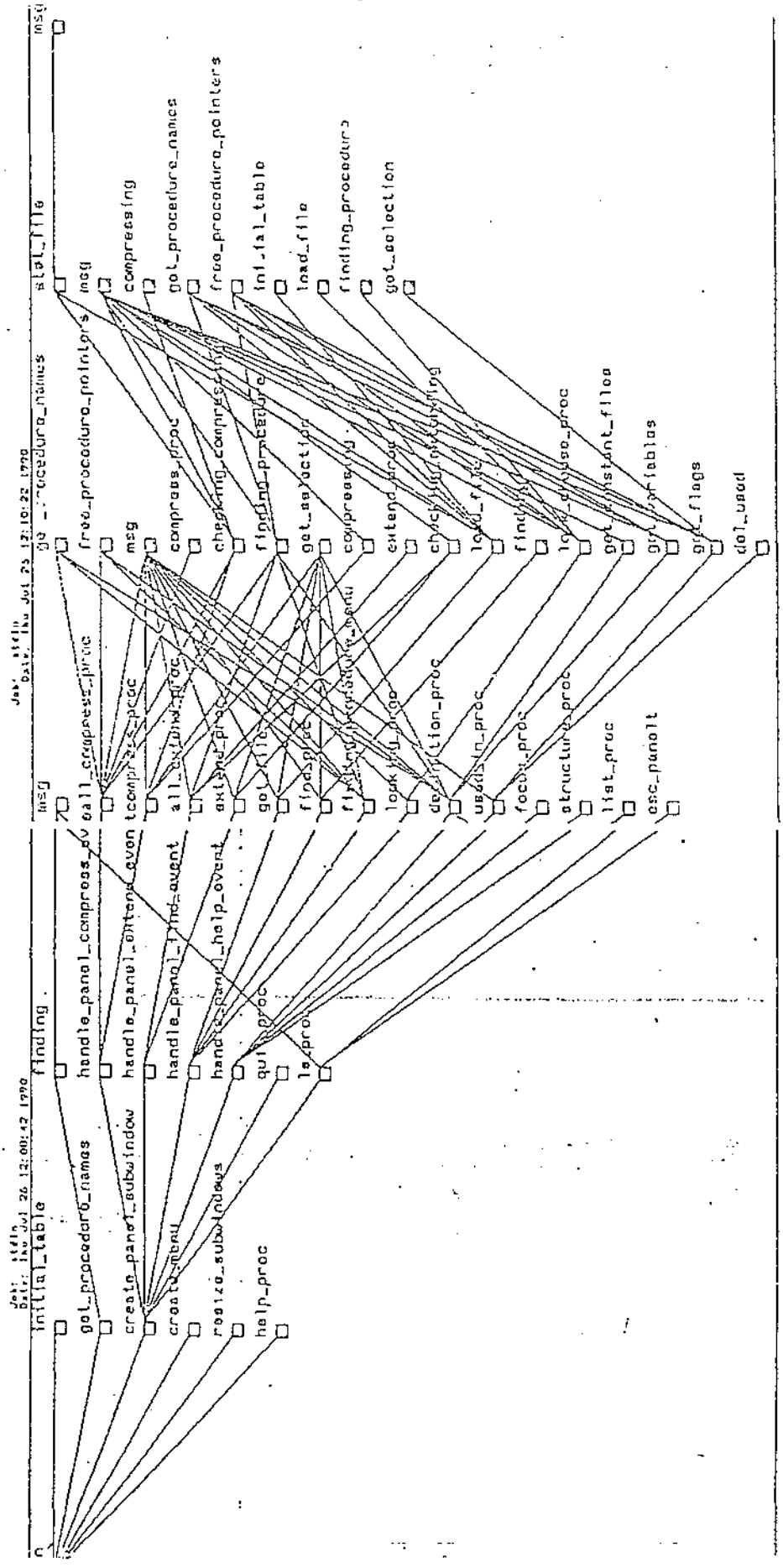
Appendix 7

Comment



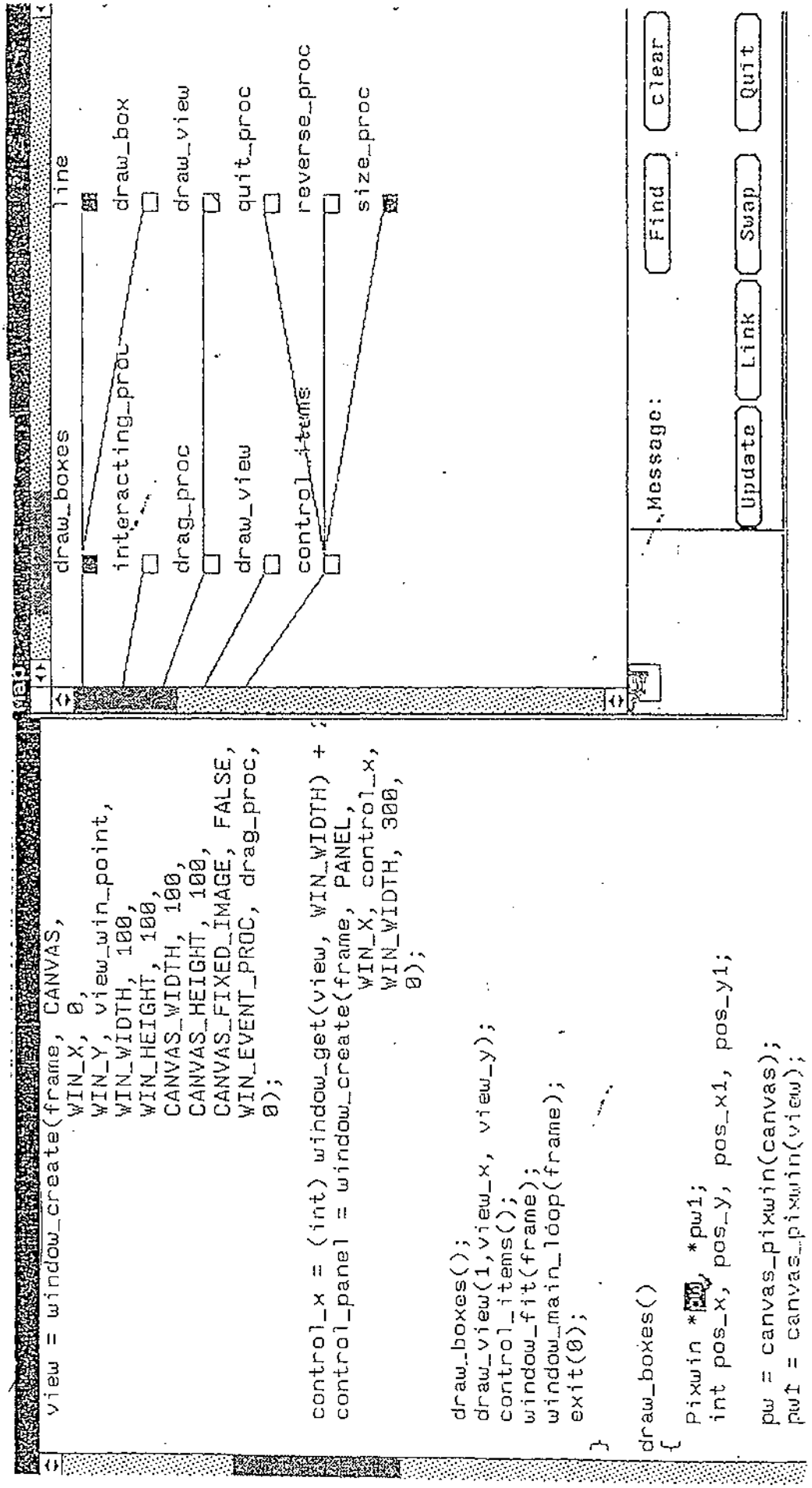
Appendix 8

A Complete Map of A Complex Program



Appendix 9

Map of A Simple Program Showing A Variable



Appendix 10

Map Viewing with A Small Window