(12)

Annual Report No. 1
Contract N00014-81-C-0456; NR 049-495
ARPA Order No. 3958

PROGRAM VISUALIZATION

Computer Corporation of America
Four Cambridge Center
Cambridge, MA 02142

22 February 1983

Annual Report for Period 8 May 1981 - 7 May 1982

Approved for public release: distribution unlimited. Reproduction in whole or in part is permitted for any purpose for the United States Government.

OFFICE OF NAVAL RESEARCH
800 N. Quincy Street
Arlington, VA 22217

AD A125291

DTIC ELECTE MAR 4 1983 S D B

DTIC FILE COPY

83 03 03 086

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
programming workstations, graphics, integrated environments, software engineering, computer animation, program instrumentation						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2*a*. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2*b*. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year, or month, year. If more than one date appears on the report, use date of publication.

7*a*. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7*b*. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8*a*. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8*b*, 8*c*, & 8*d*. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9*a*. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9*b*. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through ________________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through ________________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through ________________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Computer Corporation of America Four Cambridge Center Cambridge, MA 02142	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP -

3. REPORT TITLE

PROGRAM VISUALIZATION

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Annual Report for Period 8 May 1981 - 7 May 1982

5. AUTHOR(S) *(Last name, first name, initial)*

Herot, Christopher F.; Brown, Gretchen P.; Carling, Richard T.; Kramlich, D.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
22 February 1983	26	40

8a. CONTRACT OR GRANT NO. N00014-81-C-0456 b. PROJECT AND TASK NO. NR 049-495; ARPA Order No. 3958 c. DOD ELEMENT Office of Naval Research d. DOD SUBELEMENT	9a. ORIGINATOR'S REPORT NUMBER(S) Annual Report No 1 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* -

10. AVAILABILITY/LIMITATION NOTICES

-

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

11. SUPPLEMENTARY NOTES -	12. SPONSORING MILITARY ACTIVITY Office of Naval Research 800 N. Quincy Street Arlington, VA 22217

13. ABSTRACT

This paper reports on the design of a program visualization (PV) environment, intended to provide lifecycle support for software development. The PV environment will capitalize on recent progress in the graphical representation of information and low-cost color graphics, to provide designers and programmers with both static and dynamic (animated) views of systems. The aim is to support maintainers of large (10**6 lines of code), complex software systems. The PV system will, in effect, "open the side of the machine" to permit users to look inside and watch their programs run. In this paper, we survey categories of program illustrations and then present and motivate the design philosophy that we are pursuing for the PV environment.

DD FORM 1 JAN 64 1473

1. MAJOR TECHNICAL RESULTS

In late spring of 1981 we wrote a paper outlining our goals for the Program Visualization (PV) System and the approach we are taking to meet these goals. This paper, "An Integrated Environment for Program Visualization", was delivered at the the IFIP WG 8.1 Working Conference on Automated Tools for Information Systems Design and Development, New Orleans, 26-28 January, 1982. The paper, a copy of which is enclosed with this report, was published in Schneider and Wasserman (eds.), Automated Tools for Information Systems Design, North-Holland Publishing Co., 1982.

The bulk of 1981 was spent designing the Program Visualization (PV) System and identifying certain key classes of images that the system should support. Particular attention was paid to the design of dynamic images to illustrate programs as they run.

This phase of the project culminated in a videotape produced in December 1981. The tape was edited into a more concise version in January 1982, and narration was added. Graphics for the tape were developed using the Paint and Animation subsystems of the Spatial Data Management System. The animated images on the tape were mock-ups in the sense that they were not yet driven by underlying software. The tape shows animated graphic depictions of control flow and data updates. An important feature of the images that were designed for PV is that they present a selection of levels of detail, giving the programmer a choice of either an overview or more detailed views of the system he or she is building.

A copy of the 1982 Program Visualization videotape is enclosed with this report.

Spring 1982, the final portion of the time period covered by this report, was spent doing system design and planning for the implementation effort that began in May of 1982.

2. TECHNOLOGICAL SIGNIFICANCE

We expect on-line, interactive graphics to have a profound impact on software development, analogous to the way that word processors have affected the production of text. With respect to static graphics, the multi-dimensional graphic information structure that we are developing for PV will allow the programmer to move easily between pieces of related information (e.g. requirements and system structure). With respect to dynamic graphics, PV's animated views of program execution can help programmers achieve a deeper and more accurate understanding of the behavior of their programs.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes

Dist	Avail and/or Special
A	

3. PRESENTATIONS AND PUBLICATIONS

December 1981

Christopher Herot, Mark Friedell, and Diane Smith took part in a DARPA conference organized by Craig Fields of the System Sciences Division. Copies of the presentations were compiled in "DARPA Conference on Computer Software Graphics, Key West Florida, Dec. 13-15 1981."

January 1982

Christopher Herot and Gretchen Brown took part in the IFIP WG 8.1 Working Conference on Automated Tools for Information Systems Design and Development, New Orleans, 26-28 January, 1982. The paper presented at this conference, "An Integrated Environment for Program Visualization", appeared in Schneider and Wasserman (eds.), Automated Tools for Information Systems Design, North-Holland Publishing Co., 1982.

February 1982

Christopher Herot gave a presentation on PV at a meeting of the Northeastern ACM Chapter on February 18.

Automated Tools for Information Systems Design
H.J. Schneider and A.I. Wasserman (eds.)
North-Holland Publishing Company
© IFIP, 1982

AN INTEGRATED ENVIRONMENT FOR PROGRAM VISUALIZATION*

Christopher F. Herot, Gretchen P. Brown
Richard T. Carling, Mark Friedell
David Kramlich, Ronald M. Baecker**

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts, USA

This paper reports on the design of a program visualization (PV) environment, intended to provide lifecycle support for software development. The PV environment will capitalize on recent progress in the graphical representation of information and low-cost color graphics, to provide designers and programmers with both static and dynamic (animated) views of systems. The aim is to support maintainers of large (10**6 lines of code), complex software systems. The PV system will, in effect, "open the side of the machine" to permit users to look inside and watch their programs run. In this paper, we survey categories of program illustrations and then present and motivate the design philosophy that we are pursuing for the PV environment.

1. INTRODUCTION

Graphical representations have demonstrated their utility in a wide variety of design and implementation activities as a means of illustrating complex relationships among components of systems. It would be inconceivable to build a ship, airplane, factory, or piece of electronic equipment without the use of diagrams. These illustrations can capture essential features while suppressing extraneous detail, and they can often be understood more readily than ordinary text. While such illustrations find widespread use in computer programming, they are almost always manually generated, making production and revision laborious.

---

* This research was supported by the Defense Advanced Research Project Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-80-C-0683. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Department, the Office of Naval Research, or the U.S. Government.

** R.M. Baecker also with Human Computing Resources Corp., Toronto.

One result of manual graphics generation is that diagrams have a tendency to become obsolete as the software they describe is implemented and changed. A second result is that the full variety and utility of graphical images remains unexploited. Third, the lack of tools for creating animated images restricts the ability to illustrate an essentially dynamic process such as a computer program. This is not to say that automated graphical representations have been totally neglected. There has been some success in automating production of static representations of programs (e.g., [8,16,18]). The idea of using computer-generated images to visualize the dynamic behavior of programs was developed in the earliest days of computer graphics ([10,20,34]), and dynamic visualizations have received continued attention (e.g., [3,4,6,9,15,21,22,40]). Only recently, however, have hardware and software advances been made which would allow automated production of both static and dynamic illustrations to become cost-effective for a broad range of applications. In addition, significant research remains to be done before diverse graphical representations can be successfully integrated within a single environment.

The Computer Corporation of America is in the first year of a three year effort to design and implement a program visualization (PV) system. The verb _visualize_ means "to see or form a mental image of". We want to aid programmers in the formation of clear and correct mental images of the structure and function of programs. Program visualization has great promise for all stages of the software lifecycle. Our research will focus on illustrating the dynamic behavior of programs, which we expect to be of most use in testing, debugging, maintenance, and training. We want the PV system to, in effect, "open the side of the machine" so that the user can form an accurate model of the program.

The PV system will provide an integrated graphics environment, capitalizing on recent progress in the graphical representation of information and low-cost color graphics. The aim is to support builders and maintainers of large (10**6 lines of code), complex software systems. (We are excluding, however, real time systems.) This tool is targeted primarily for use with programs written in Ada [1], the proposed standard DoD language.

The system which is envisioned will provide individuals who must build and maintain a complex software system with access to a variety of graphical representations. These will include static descriptions, such as module hierarchies and requirements specifications, and dynamic illustrations, such as procedure activations and storage allocations. It will be possible to display several different representations of the same portion of a system (or the same representation of several different portions) simultaneously through the use of multiple screens or multiple viewports on one screen. Since we are focusing on the dynamic behavior of programs, we will make extensive use of computer animation.

This paper is intended to initiate a debate concerning the characteristics of a good graphics environment for software production. Section 2 surveys a number of categories of information that merit graphical presentation. The number and variety of these categories suggests that fuller exploitation of graphics can profoundly influence software production, just as text editing facilities have changed the way that papers are written. The challenge that we see is to encompass the volume and diversity of this information within a coherent conceptual framework. Section 3 expounds a design philosophy for PV. We address there three significant problems: integrating the categories of illustrations identified in Section 2, enhancing graphical facilities, and instrumenting the program. Implementation plans for the PV system are discussed in Section 4, with a summary in Section 5.

2. CATEGORIES OF VISUALIZATIONS

It is our claim that although graphics has long been a tool in program development and documentation, the full power of graphics has yet to be acknowledged or exploited. This section lists ten categories of program illustrations that, together, can be of use throughout the software lifecycle. These categories were one result of a six month study conducted to provide a conceptual framework for program visualization. Some categories of illustrations have already been well explored with respect to programs, and the PV environment will draw on this work directly. Other categories have been less thoroughly explored, and suggestions for new directions are included here. The ten categories are:

1. System requirements diagrams
2. Program function diagrams
3. Program structure diagrams
4. Communication protocol diagrams
5. Composed and typeset program text
6. Program comments and commentaries
7. Diagrams of flow of control
8. Diagrams of structured data
9. Diagrams of persistent data
10. Diagrams of the program in the host environment

Many of these categories can apply to either <u>the program</u> or its specific <u>activations</u>. Moreover, illustrations can be either <u>static</u> or <u>dynamic</u>. Static illustrations portray the program at some instant of execution time, or they portray those aspects of a program which are invariant over some interval. Dynamic illustrations portray the progress of an executing program.

We shall now look at each of the ten categories of illustration in more detail.

2.1 System Requirements Diagrams

A computer program always exists as part of some larger system (not necessarily a fully automated one). Therefore, PV tools must assist in the portrayal of the function and structure of that system. The tools should also aid in the specification of the constraints imposed by the system on the program.

One very powerful method of describing system structure is the IDEF or SADT technique, developed by SofTech [29]. An IDEF model is a graphical representation of a system in terms of its subsystems and in terms of the data and control flow that links them together. The method deals with the hierarchic nature of most systems quite naturally, and it provides a methodology for organizing the bookkeeping associated with large complex system descriptions.

A complete requirements specification also contains constraints on the program's design, constraints such as execution speed, program size, user interface style, implementation vehicle, implementation cost, and the like. We are investigating whether there is any significant role for graphics in describing these latter specifications.

2.2 Program Function Diagrams

"What does the program do?" is usually the first question that one asks about a program. For many types of programs, program function can be viewed as a mapping from program input to program output.

We can talk about the relationship of program input to output in two very different ways: a statement of the program's function in general terms or an enumeration of a number of input-output pairs. The former is a more powerful and useful description, but, because it is an abstraction, it poses difficult problems for graphical representation.

Graphical portrayal of sample behaviors is a much more straightforward proposition. Thus one approach to the visualization of program function is to provide a "casebook" through which the user can browse, inducing a model of what the program is supposed to do by seeing what it actually does on a carefully selected set of sample inputs. The choice of these sample inputs can have a significant effect on the utility of this technique. For example, in understanding a factorial function, important values and classes are 0, 1, positive integers, negative integers, reals, and non-numerics.

2.3 Program Structure Diagrams

"How is the program organized?" is often the second question that one asks about a program. Program structure has a well-developed history of graphical notation. Relatively recent examples are the HIPO (Hierarchy plus Input-Process-Output) technique [14,33], which integrates structure diagrams and function diagrams, and the composite design structural notation [23].

2.4 Communication Protocol Diagrams

Once it is known how a program is divided into its component parts, it is useful to know how those parts communicate. This is especially important when the program consists of many processes running on one or more processors. An illustration of the potential paths for data flow between modules can be displayed as part of another diagram. For instance, the program structure diagram can be overlaid with lines showing the data paths between modules. By using this technique dynamically, the actual flow of data can be monitored during execution. For example, the SDD-1 distributed data base system [30] at CCA employs a color graphics terminal to show data transfers between sites on the Arpanet as they occur. This monitoring technique has proven useful both as a demonstration and a debugging aid. Figure 1 is a reproduction of a typical SDD-1 illustration, streamlined for the purposes of this paper. Except where otherwise noted, the figures in this paper are all handdrawn mockups.

2.5 Composed and Typeset Program Text

The central activity in the visualization of programs has always been the reading of program code. While alternative graphical techniques are proposed here, there will still be cases where code must be examined. This task can be made significantly easier than it is at present. Some relevant typographic tools that can be applied to the display of programs are:

1. The use of _typographic hierarchies_ for distinguishing a program's constituent elements that belong to various syntactic or semantic categories. Typographic hierarchies are implemented by the consistent and controlled use of a variety of type fonts, type styles within a font (bold, condensed, italic, etc.), and point sizes.

2. The use of a _rich symbol repertoire_ employing a wider range of symbols and colors than are currently used. (Gutenberg had more than 300 symbols in his type case.)

3. The use of _composition and layout_ to facilitate the structured perusal of a program's constituent substructures. Layout conventions include the use of indentation, horizontal paragraphing, vertical paragraphing, pagination, footnotes, marginal notes, and page headers. The use of computer graphics also permits dynamic techniques such as colored highlighting over selected or active portions of program text.

These techniques can be applied both to produce displays on high resolution terminals and to produce hard copy on a demand basis. Figure 2 shows the application of some of these techniques to a subroutine written in C.

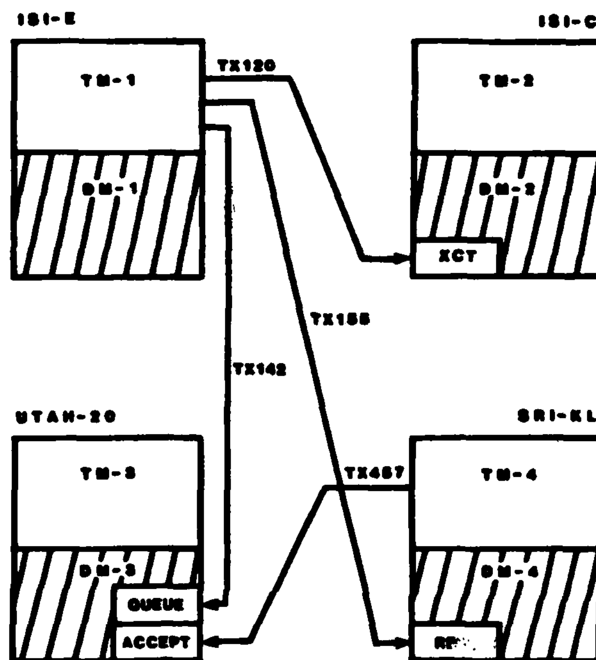


Figure 1. A snapshot of a dynamic illustration of communication in SDD-1, a distributed data base system. Four sites are shown. Transaction modules (TMs) send transactions (arrows) to data modules (DMs).

<table>
<tr><th>sdms_copy (a[illegible]</th><th>copy film from core buffer to local buffer</th></tr>
<tr><td>struct node *afilm;
{
int curmap[], tvfilt[];
int x1, y1, x2, y2;
film = (struct filmdata *) afilm -> id;
if (film = = NULL) return FAIL;
framelink = film -> frame;
frame = (struct framedata *)
framelink -> id;
if (frame = = NULL) return FAIL;</td><td></td></tr>
<tr><td>x1 = film -> xfilm + frame -> dx;
y1 = film -> yfilm + frame -> dy;

if (frame -> len = = 0 | | frame -> ht = = 0)
{
x2 = x1 + film -> frame_len - 1;
y2 = y1 + film -> frame_ht - 1;
}
else
{
x2 = x1 + frame -> len - 1;
y2 = y1 + frame -> ht - 1;
}</td><td>compute coordinates of frame window

(x1,y1)
(x2,y2)</td></tr>
<tr><td>feedout
(curmap, tvfilt, x1, y1, x2, y2);

show_frame (afilm);

return SUCCESS;

}</td><td>call feedout to copy</td></tr>
</table>

Figure 2. Use of layout and typography for code and comments. The subroutine sdms_copy is written in C. Typography is used to emphasize important lines of the subroutine, and comments appear on the right in blocks to indicate their scope.

2.6 Program Comments and Commentaries

Program comments, often known as internal documentation, are analogous to the footnotes and marginal notes of conventional literary expression. Program commentaries, often known as external documentation, are analogous to prefaces, introductions, postscripts, and critical expository analyses. Both comments and commentaries are an important part of conventional programming discipline, yet they fall far short of attaining their ultimate potential. How can they be improved?

The greatest potential for improvement comes from an area for which there is no technological fix, that is, the ability of programmers and documentation specialists to write in English with clarity and consistency. An integrated graphics system can, however, provide significant aid to the programmer in other respects. If a PV system supplies a variety of graphical representations to illustrate different facets of a program, then there will be much less need for pure text comments and commentary. Comments and commentary will tend to be limited to very general remarks on the one hand, and to very particular observations (e.g., noting exceptions and special cases), on the other. Comments and commentary, in their reduced role, can also benefit from the typesetting and composition techniques discussed above for program text. (See Figure 2, for the use of lines and spacing to delineate the scope of comments.) Finally, completeness and consistency of comments and commentary can be supported (although never, of course, assured) by the programming environment.

2.7 Diagrams of Flow of Control

"What happens when the program executes?" is another important question about a program. "In what order do things happen?" is one subquestion, with diagrams of flow of control as a relatively familiar means of conveying the answer. (Another subquestion, apropos of the effect of program execution on underlying data, is discussed in Section 2.8.)

Flow charts, Nassi-Shneiderman Diagrams [8,24], Software Diagram Descriptions [18], GREENPRINTS [5], and others are a beginning, but they portray only the static structure, not the actual flow of control during program execution. Some formalisms already provide a means to indicate flow of control (e.g., the tokens in Petri nets [26]). It is only a short distance to computer animations of flow of control, with tokens, highlighting, and/or color changes used to denote the current state of the execution.

2.8 Diagrams of Structured Data

"What happens when the program executes?" can also be answered in terms of the data base upon which the program is operating. This data base includes the program input at the initiation of execution and the program output at the termination of execution. The data base also includes the variables that are the raison d'etre of a

program, such as the data being sorted by a quicksort, and the variables that are incidental to the program's function, that is, the artifacts of a particular piece of code or programming technique. The major difficulty in representing this information results from the size and complexity of the data bases of most interesting programs. It is for this reason that we have spoken of "diagrams of structured data." It is only through structuring the complexity (either at the design stage or under interactive programmer control) that we are able to comprehend it and master it.

An appropriate illustration of a program's data, updated dynamically during program execution, gives us the feeling of looking into the machine and seeing the program running. Baecker's pilot film on sorting algorithms [3], and the work of Knowlton [20], Hopgood [15], Myers [22], and others, have vividly demonstrated the power of this technique. Figure 3 shows one possible layout for displaying a two dimensional array. The visualization of structured data appears to be one of the most tractable and powerful of the approaches we have presented.

2.9 Diagrams of Persistent Data

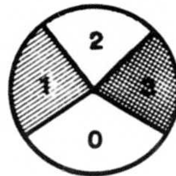
An important category of structured data is that which remains in the computer system after the program has ceased execution, as occurs in a data base management system. Since this data is often several orders of magnitude larger than the memory capacity of the computer, different techniques are required to visualize it. Fortunately, the data base community has developed a rich set of symbols which can serve as a starting point in visualizing persistent data (e.g., [2,32]). In addition, the Spatial Data Management System (SDMS) [11] developed at CCA has demonstrated the feasibility of using graphics to access persistent data. Figure 4 shows layouts taken from an actual SDMS interface to a data base of information about ships.

2.10 Diagrams of the Program in the Host Environment

There is a collection of information about programs that is typically available in various forms from operating systems, but, just as typically, the information is presented in a rather dense and detailed tabular form. This information includes the files in which program parts are stored, their size, age, and ownership. For a program activation, performance and timing information is important. The type and percentage of resources currently in use, priority under which the program is running, and the average response time for interactions are all commonly of interest.

Much can be done to enhance the presentation of this information. Percentages can be displayed as pies, histograms, etc. Color coding can be used to point up similarities among pieces of information or to highlight particular properties of interest. Finally, the typographic hierarchies discussed for program text, comments, and commentaries can play a role here as well.

array AVALS (1-100, 1-4)



Last Entry

Index		Value
98	2	3

Value Distribution

96	3	0	0	1
97	0	0	2	1
98	0	3	0	0
99	1	0	0	2
100	1	3	0	0

Figure 3. Display of the contents of an array called AVALS. Array is set in a window and values can be examined by pressing arrows to scroll up or down.

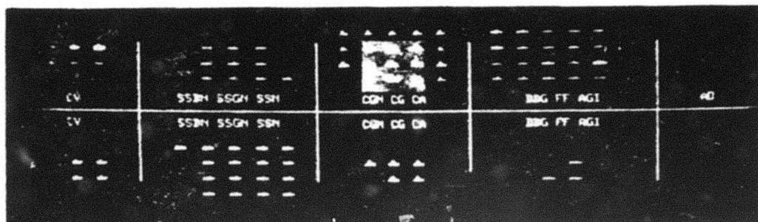


Figure 4. Photos from SDMS showing categorization of ship information by country (horizontal) and by ship type (vertical). Detail photo corresponds to highlighted portion of photo at top. Shape of the ship indicates ship type, and text is identification and command information taken from the data base. Additional information conveyed by color coding is visible as grey scale in these pictures.

3. A PV DESIGN PHILOSOPHY

The previous section enumerated a "shopping list" of ten categories of graphical, and often dynamic, information about a program. It should be clear from this list that graphics already plays a significant role in the software lifecycle; it should also be clear that the possibilities of graphics have only begun to be exploited. Much work has been done in developing techniques for describing programs at various levels, but comparatively little has been done to automate these techniques to give programmers a clear concept of the dynamics of the program. This latter goal is the focus of our work. The PV system will "open the side of the machine" and give the user the feeling of looking in and seeing the program run. The primary impact of such a focus is on the testing, debugging, maintenance, and training phases of a program's lifecycle. We expect good dynamic program illustrations to greatly enhance the programmer's effectiveness in each of these areas.

Before we can "open the side of the machine", there are a number of problems that must be solved. We have chosen three of them to explore in depth. These are:

- Integrating information:
  How can the ten categories of illustrations and text be combined meaningfully?

- Enhancing graphics facilities:
  How can we provide the PV user with very high level graphics facilities?

- Instrumenting programs:
  How can programs be instrumented to permit access to the relevant dynamic information without requiring extensive programmer intervention?

Other research problems exist, but we feel that, given the current state of knowledge, these three are the most immediate. In particular, a whole series of research problems can be classified under the heading of program soundness: how can the correctness of the design and the code be guaranteed? While this is an important research area, it is one that is already receiving a significant amount of attention in a variety of contexts. Program verification is one relatively well developed area, and promising work is also being done in program understanding (e.g., [27,39]) and in the use of program templates to avoid errors (e.g., [37]). The treatment of information about a program as a data base has been coupled with enforcement of standards (e.g., in CADES [25]) and decision support (e.g., in the DREAM System [28]). This treatment also opens the way for application of more general work on the semantic integrity of data bases. While the PV system will include some checking, cross-referencing, and decision support, insuring the soundness of the information is not a current research focus.

In this section, we present and motivate a PV design approach that addresses the three problems of integration of information, enhancement of graphics facilities, and program instrumentation.

3.1 Integrating Information

We expect a large volume of diverse information to be involved in the visualization of a program. This is true first because the PV environment will support the production of large programs and, second, because it will provide a variety of graphical representations. The challenge is to integrate the various techniques identified in Section 2 into a coherent whole.

The organization of program information ("program space") that we are currently exploring is the placement of visualizations in a hierarchy of two-dimensional surfaces. The levels in this hierarchy would match the structure of the program under investigation. As one descended in the spatial hierarchy, one would view successively more detailed representations, for instance viewing increasingly lower-level modules. For each node in the hierarchy, the surface would contain examples of some or all of the illustration categories described in Section 2. (This includes dynamic illustrations, which can be thought of as movies embedded in surfaces, runable under user control.) A mechanism would also be available, most likely on a separate screen, to provide a workspace for the PV user to accumulate views that are currently of interest.

Movement through the program space is illustrated in Figure 5, which shows three displays related to the SDD-1 communication visualization from Figure 1. The first two displays were formed by selecting transaction modules (TMs) and data modules (DMs), as a PV user might do in localizing a bug. The third display is formed by zooming to another level of detail, to access a table of information about transactions running at a particular TM.

While we are emphasizing the hierarchic organization of illustrations because it is the dominant organization, there will actually be a network of links between the illustrations from different categories. The complexity of this subordinate organization poses two types of challenges: the implementation problem of handling multiple connections, and the knowledge representation problem of keeping the information manageable for the user. For the first, a surface must permit an arbitrary number of ports, i.e. links to locations on other surfaces. For the second, clean organization is important, and navigational aids must be provided for the PV user.

Our approach to these problems draws on experience with SDMS [11], the Spatial Data Management System. SDMS was motivated by the needs of a growing community of people who need access to information in a data base management system but who are not trained in the use of such systems. We briefly describe some of the aspects of SDMS that are relevant to PV.

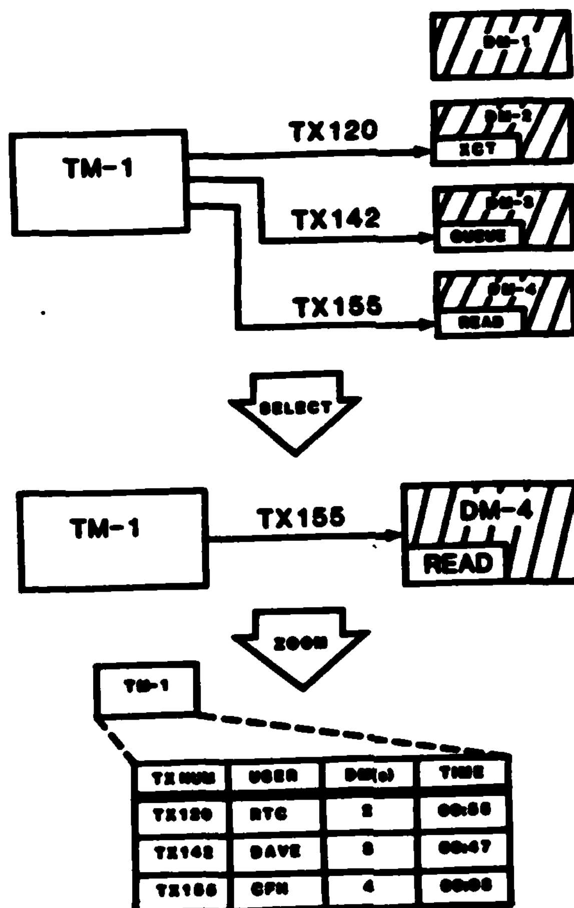


Figure 5. Three successive views of SDD-1. Top, the same snapshot as Figure 1, Section 2.4, but this time only TM-1 and the four DMs are shown. Middle, further selection yields display of a single TM/DM pair. Bottom, zooming in on TM-1 results in a display of the transactions originating from that site.

In SDMS, information is expressed through color graphics, with different categories of structured data having associated icons, i.e., prototypical pictograms. Icons are generated semi-automatically from the data base, according to a predefined set of rules. Information retrieved from the data base is presented in SDMS in a multidimensional framework, so that the user can "zoom" through ports on the data surface to see more detail or to see alternative views of the data. The SDMS workstation includes three screens, a joystick for panning and zooming, touch sensitive screens to permit the user to select icons by pointing at them, and a data tablet for the creation of new pictograms. As a navigational aid, one screen displays a world-view map which shows an entire data surface. A highlighted rectangle on the world-view map indicates the portion of the data that is being presented in detail on another screen. The rectangle moves as the user manipulates the joy stick to move over the data surface. One world-view map was shown in Figure 4, Section 2.9.

The program space that we have described for PV fits well into the multidimensional paradigm of SDMS, and we expect much of the SDMS experience to be applicable to PV. The multilayered PV program space is analogous to the SDMS data surfaces, and the further connections needed between illustration categories can be implemented with multiple ports. The combined use of joy stick, touch sensitive screens, and data tablet can enable the PV user to move through and augment the program space easily. The world-view map is of proven utility; we do, however, have more to say on this count.

It is possible that the complexity of the PV environment will require other navigational aids in addition to the world-view map. We are investigating several different approaches to helping the PV user navigate within the system, and to helping him/her maintain a sense of context. One such approach is the use of a list of miniaturizations of past displays, both to provide context and, when a miniaturization is selected, to provide a means to "pop" the stack back to a past view.

We have, then, a dominant hierarchical organization with a subordinate network organization, realized in a hierarchy of surfaces containing multiple ports. We next take a closer look at the specification of illustrations, both in program space and in the user's workspace.

3.2 Enhancing Graphical Facilities

As a comprehensive environment, it is crucial that the PV system be easy to use or, where it must be more demanding, that the system repay the investment. Graphics makes this imperative all the more challenging. Designers and programmers want access to information about a program without having to spend a great deal of time specifying how the information is to be displayed.

We have identified three ways in which graphics facilities can be enhanced:

1. Incorporation of domain-independent higher level constructs
2. Incorporation of graphical and text constructs specific to programming
3. Automated layout

Work in each of these areas has important implications both for getting information into the PV environment and getting it out. Each area is discussed briefly.

First on our list was the incorporation of domain-independent higher level of constructs. SKETCHPAD [35], PYGMALION [31], the SDMS PAINT program [12], and others, have permitted picture production without programming, by combination of shapes and free drawing. These techniques can be enhanced. The graphical vocabulary need not be restricted to geometric shapes but also can include higher level organizational devices, such as overlays and split screens. Production of animation can move away from frame by frame specification to specification by combining high level dynamic constructs. For example, a growing and then shrinking arrow used to represent communication between modules should be specifiable as a unit. (One approach to this problem is given in [17].) Throughout, defaults for graphical conventions can be more fully exploited.

As powerful as high level domain-independent graphical constructs can be, we believe that the PV environment will also have to provide graphical (and textual) constructs specific to programming. For those times when programmers must dynamically specify displays, we expect the predominant mode of interaction to be the instantiation of templates, not the creation of new pictures from scratch. Programmers will have both default representations for a wide range of data structures and the means to change these defaults to meet special needs. (See, e.g., [22].) Templates for standard programming utilties (e.g., stacks, memory maps) will also be provided. The templates need not be limited to single structures such as stacks; we are exploring the use of composite templates that combine information that is generally useful for particular types of programming. Finally, for those cases in which totally novel pictures are required, the user can turn to a graphics editor that will be included in the PV environment.

Once the PV user selects the components of a display, automated layout can arrange the diagrams on the display area. For layout, we intend to build upon work done as part of the SDMS project. Additional work will be needed to handle the more richly connected, and often heterogeneous, pictures that will be common for PV.

The three facilities listed -- high level domain-independent graphical language, graphical constructs specific to programming, and automated layout -- can permit the PV user to produce visualizations with a minimum of effort, and without being inundated in detail.

3.3 Instrumenting Programs

The central focus in the PV project is on providing an environment that runs programs, not merely one that manipulates static information. Traditional debuggers give us some models and a set of experiences to draw on, but there are ways that they can be improved in terms of bringing the level of interaction with the debugger up to the conceptual level of the user. We outline briefly our philosophy.

First, the user's goal is to see information about the running program. The use of two or more windows on the screen to display a running program is one approach that has been used successfully ([36,38]). For programs designed to communicate with a terminal, one window simulates the terminal screen. Characters typed by the user appear there and output from the program is directed there. Meanwhile, other windows can be used to examine the operation of the system. Sets of windows can also be used to display multiple processes or to display causes and effects.

We intend to experiment with the multiple window approach and other modes of display as well. In setting up the display environment, we are paying particular attention to displaying choices that have been made by the user. This is in hopes of remedying one of the problems with traditional debuggers -- the difficulty of keeping track of the constraints that the user has set up as a filter for viewing the behavior of the program.

Given a display environment, the user's problem is to specify what is to be displayed. The use of high level graphics constructs, including programming construct templates, was discussed in the previous subsection. These templates must also be instantiated, i.e. tied to the code or other representations of the program. For this, debugging statements and code will be conceptually separate entities, so the user will not be thinking in terms of altering code (and later having to remove debugging statements). Pointing (graphical cursors and/or touch sensitive screens) will be used extensively to specify instantiations of the templates. Many slots in the templates can be filled by pointing at variables in the code. The programmer can point to locations in program structure diagrams or typeset listings to specify areas of interest. When these areas are active, the program is slowed down to a visible speed. At other times it runs at normal speed, unobserved but much faster. Similarly, breakpoint locations can be indicated by pointing.

The mechanics of coding the display specification and producing the information desired from the running program have been left almost entirely to the PV system. This is the area about which we have the least to say right now, but about which we expect to have more to say in the coming months. In brief, our philosophy is that whether code is compiled or interpreted should be transparent to the programmer. Further, we hope to stay out of the compiler writing business. We intend to use existing compiler(s), perhaps augmented to save information that might otherwise be thrown away (e.g., a full

symbol table). Unix facilities that allow one process to gain control of another process between statements will be exploited [19]. Finally, we are exploring the tradeoff between accumulating histories of program execution as opposed to facilitating rerunning of programs, particularly as this tradeoff is affected by persistent data.

***

We have outlined three areas of particular concern in the design of a program visualization system: integration of information about the program, design of high level graphical facilities, and instrumentation of the program. Readers interested in more background on the PV framework are referred to [13].

4. IMPLEMENTATION

The current work is focused on the implementation of a tool usable by Ada programmers by 1984, and is proceeding in three phases. The various implementations will explore a variety of techniques on a powerful high-resolution color display environment, with an eye toward identifying a useful subset of techniques which can be implemented on a low-cost terminal costing in 1985 what an ordinary alphanumeric terminal costs today. The three implementation phases are:

1. The design of a visual language for describing programs, together with the processing, translation, and display routines necessary to create a visualization of a program. This language and its concomitant machinery is being developed through interactions with programmers responsible for large-scale working systems and with information-oriented graphic designers. Static and dynamic mock-ups of program illustrations are being created and evaluated.

2. The implementation of a breadboard system which incorporates the results of the first phase and allows integration with the results of other research efforts, for use with a selected language on Unix. It will be evaluated through use by people maintaining and developing software on Unix.

3. The implementation of a production program visualization system for Ada, incorporating improvements identified in the breadboard version.

One early result of the Phase One exploration of illustrations is a five minute videotape of dynamic illustration mock-ups for SDD-1 (from which Figures 1 and 5 were taken). The feedback on this effort has led us to conclude that if the operation of a program is not intuitively obvious, the graphics used must bridge the gap. We are therefore developing (and videotaping) another set of dynamic illustrations with graphical symbols that are more perceptual than

symbolic ([7]) i.e., that portray the semantics more immediately.

5. SUMMARY

This paper, and the project itself, started from the premise that graphical illustrations can make a significant contribution to the process of program visualization. In support of this premise, we first surveyed a variety of existing graphical representations and discussed possibilities for extended uses of graphics, especially animation. We then outlined a design philosophy for an integrated graphical programming environment to support the development of large systems. Our focus was, and is, on the dynamic aspects of programs, to permit the PV user to "open the side of the machine" and watch programs run. Work is underway to provide such an environment, with the goal of making the advantages of graphical representations available without placing an excessive burden on the people responsible for implementing and maintaining the programs. We believe that a comprehensive and integrated approach to program visualization can have great impact on the entire process of software engineering, and on the cost-effective production and maintenance of reliable software.

ACKNOWLEDGEMENTS

The authors wish to thank Jane Barnett, Diane Smith, and Gerald Wilson for helpful comments on drafts of this paper. We are also grateful to Jane Hathaway for the artwork.

6. REFERENCES

[1] Preliminary ADA reference manual, SIGPLAN Notices, 14, 6, Part A, (June 1979).

[2] Bachman, C.W., The evolution of storage structures, Communications of the ACM, 15, 7 (July 1972) 628-636.

[3] Baecker, R.M., Sorting Out Sorting, 16mm color, sound, 25 minutes (Dynamic Graphics Project, Computer Systems Research Group, Univ. of Toronto, 1981).

[4] Baecker, R.M., Two systems which produce animated representations of the execution of computer programs, ACM SIGCSE Bulletin, 7, 1 (Feb. 1975) 158-167.

[5] Belady, L.A., Cavanagh, J.A., and Evangelisti, C.J., GREEN-PRINT: a graphical representation for structured programs, IBM Research Report RC 7763, T.J.Watson Research Center (1979).

[6] Dionne, M.S. and Mackworth, A.K., ANTICS: a system for animating LISP programs, Computer Graphics and Image Processing, 7 (1978) 105-119.

[7] Fitter, M. and Green, T.R.G., When do diagrams make good computer languages?, Int. J. Man-Machine Studies, 11 (1979) 235-261.

[8] Frei, H.P., Weller, D.C., and Williams, R.A., Graphics-based programming-support system, Computer Graphics, 12, 3 (August 1978) 43-49.

[9] Galley, S.W. and Goldberg, R.P., Software debugging: the virtual machine approach, Proceedings: ACM Annual Conference (1974) 395-401.

[10] Haibt, L.M., A program to draw multilever flow charts, Western Joint Computer Conference (1959).

[11] Herot, C.F., Carling, R.T., Friedell, M., Kramlich, D., A prototype spatial data management system, SIGGRAPH '80 Proceedings: ACM/SIGGRAPH Conference (1980) 63-70.

[12] Herot, C.F., Carling, R.T., Friedell, M., Kramlich, D., and Thompson, J., Spatial data management system: semi-annual technical report, Technical Report CCA-79-25, Computer Corporation of America (1979).

[13] Herot, C.F., Carling, R.T., Friedell, M., and Kramlich, D., Design for a program visualization system, Technical Report CCA-81-04, Computer Corporation of America (1981).

[14] HIPO -- A Design Aid and Documentation Technique (IBM Corporation, Data Processing Division, White Plains, New York).

[15] Hopgood, F.R.A., Computer animation used as a tool in teaching computer science, Proceedings of the 1974 IFIP Congress, Applications Volume, (1974) 889-892.

[16] Presentation on AISIM at the Software Tools Fair, Fifth International Conference on Software Engineering (Hughes Aircraft Company, Ground Systems Group, Fullerton, California, 1981).

[17] Kahn, K.M., Creation of computer animation from story descriptions, AI TR-540, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (1979).

[18] Kanda, Y. and Sugimoto, M., Software diagram description: SDD and its application, Proceedings: Computer Software and Applications Conference (1980) 300-305.

[19] Kernighan, B.W. and McIlroy, M.D., Unix programmer's manual (Bell Laboratories, Murray Hill, New Jersey).

[20] Knowlton, K.C., L6: Bell Telephone Laboratories Low-Level Linked List Language, two black and white films, sound (Bell Telephone Laboratories, Murray Hill, New Jersey, 1966).

[21] Model, M.L., Monitoring system behavior in a complex computational environment, CSL-79-1, XEROX Corp., Palo Alto Research Center (1979).

[22] Myers, B.A., Displaying data structures for interactive debugging, CSL-80-7, XEROX Corp., Palo Alto Research Center (1980).

[23] Myers, G.J., Composite/structured design (Van Nostrand Reinhold Company, New York, 1978).

[24] Nassi, I. and Shneiderman, B., Flowchart techniques for structured programming, SIGPLAN Notices of the ACM, 8, 8 (1973) 12-26.

[25] Pearson, D.J., The use and abuse of a software engineering system, Proceedings: 1979 National Computer Conference (1979) 1029-1035.

[26] Peterson, J.L., Petri nets, ACM Computing Surveys, 9 (1977) 223-252.

[27] Rich, C. and Shrobe, H., Initial report on a Lisp programmer's apprentice, IEEE Trans. on Software Eng., 4, 5 (1978).

[28] Riddle, W.E., An assessment of DREAM, in: Huencke, H. (ed.), Software Engineering Environments (North-Holland Publishing Co., 1981).

[29] Ross, D.T., Structured Analysis (SA): A Language for Communicating Ideas, Software Engineering, SE-3, 1 (1977) 34.

[30] Rothnie, J.B., Jr., Bernstein, P.A., Fox, S., Goodman, N., Hammer, M., Landers, T.A., Reeve, C., Shipman, D., and Wong, E., Introduction to a system for distibuted databases (SDD-1), ACM Trans. Database Syst. 5, 1 (1980) 1-17.

[31] Smith, D.C., PYGMALION: a creative programming environment, AIM-260, Stanford Artificial Intelligence Laboratory, Stanford Univ. (1975).

[32] Smith, J.M. and Smith, D.C.P., A data base approach to software specification, in: Riddle and Fairley (eds.), Software Development Tools (Springer Verlag, New York, 176-201).

[33] Stay, J.F., HIPO and integrated program design, IBM Systems Journal, 15, 2 (1978) 143-154.

[34] Stockham, T.G., Some methods of graphical debugging, Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communication (1965) 57-71.

[35] Sutherland, I.E., SKETCHPAD: a man-machine graphical communication system, Proceedings of the Spring Joint Computer Conference (1963) 329-346.

[36] Swinehart, D., COPILOT: A multiple process approach to interactive programming, AIM-230, Stanford Artificial Intelligence Laboratory, Stanford Univ. (1974).

[37] Teitelbaum, R.T., The Cornell program synthesizer: a microcomputer implementation of PL/CS, TR 79-370, Department of Computer Science, Cornell Univ. (1979).

[38] Teitelman, W., A display oriented programmer's assistant, Fifth International Joint Conference on Artificial Intelligence (1977) 905-915.

[39] Waters R.C., A method for analyzing loop programs, IEEE Trans. on Software Eng., SE-5, 3 (1979) 237-247.

[40] Yarwood, E., Toward program illustration, Tech. Report CSRG-84, Computer Systems Research Group, Univ. of Toronto (1977).

END OF FISCAL YEAR REPORT - FY82

Contract Title: Program Visualization

Contract Number: N00014-81-C-0456

ONR Work Unit Number: NR 049-495

Principal Investigator: Christopher F. Herot

ONR Scientific Officer: Robert Grafton

MAJOR TECHNICAL RESULTS

The first half of FY82 was spent designing the Program Visualization (PV) System and identifying certain key classes of images that the system should support. Particular attention was paid to the design of dynamic images to illustrate programs as they run.

This phase of the project culminated in a videotape produced in December 1981. The tape was edited into a more concise version in January 1982, and narration was added. Graphics for the tape were developed using the Paint and Animation subsystems of the Spatial Data Management System. The animated images on the tape were mock-ups in the sense that they were not yet driven by underlying software. The tape shows animated graphic depictions of control flow and data updates. An important feature of the images that were designed for PV is that they present a selection of levels of detail, giving the programmer a choice of either an overview or more detailed views of the system he or she is building.

A copy of the 1982 Program Visualization videotape is enclosed with this report.

The second half of FY82 was spent implementing a base level PV system. During this time period, we produced an initial version of the PV System, with the following components:

1. User Interface

The user sees detailed views of programs, both text

and graphics, through a system of multiple overlapping windows.

Commands to manipulate the information in these windows are displayed as sets of buttons on a separate menu display.

The user inputs information via a combination of data tablet, joy stick, touch sensitive screens, and keyboard.

2. Graphics Editor

The PV Graphics Editor allows the user to construct illustrations interactively by "drawing" on a data tablet. The base level editor allows the user to input shapes and lines (for connectors). A selection of line weights and fonts permits the construction of clear, legible diagrams. Underlying data structures have been implemented to to include information about the structure of the objects drawn. This knowledge will allow upgrade of the existing editor to a knowledge-based graphics editor; for example, when an object is moved, its associated connectors can be moved as well.

3. Text Editor

An existing text editor, EMACS, was extended and integerated into the system.

4. Binder

A set of special-purpose routines was implemented to enable the user to specify the relationships between programs and the graphic depictions of certain standard data types. The system currently handles arrays and single units such as integers. The binding step is necessary so that the PV system can know when the update of a variable in a program should cause update in an associated visualization. In the coming months, we will be extending and generalizing the set of binding routines.

5. Execution Manager

Enhancements have been designed for the Execution Manager, which is responsible for monitoring code for changes that are relevant to program visualizations currently displayed. The new scheme is a variation on tagged memory that tags UNIX pages instead of tagging individual memory locations. The design uses access restrictions, setting pages with variables of interest to read-only. When the system tries to change a variable of interest, the access violation will alert the

system to enable a trace. The variable can then go ahead and be written, and the trace process can identify the location of the variable, so the PV system can access the new value and update the relevant visualizations.

These capabilities were integrated in the base level implementation of September 1982. At this time, the system had developed far enough to permit the creation of an animated visualization of the data structures of a sort subroutine. This visualization, which was adapted from images designed in the first half of the fiscal year, is now fully supported by the PV system.

TECHNOLOGICAL SIGNIFICANCE

We expect on-line, interactive graphics to have a profound impact on software development, analogous to the way that word processors have affected the production of text. With respect to static graphics, the multi-dimensional graphic information structure that we are developing for PV will allow the programmer to move easily between pieces of related information (e.g. requirements and system structure). With respect to dynamic graphics, PV's animated views of program execution can help programmers achieve a deeper and more accurate understanding of the behavior of their programs.

PRESENTATIONS AND PUBLICATIONS

December 1981

Christopher Herot, Mark Friedell, and Diane Smith took part in a DARPA conference organized by Craig Fields of the System Sciences Division. Copies of the presentations were compiled in "DARPA Conference on Computer Software Graphics, Key West Florida, Dec. 13-15 1981."

January 1982

Christopher Herot and Gretchen Brown took part in the IFIP WG 8.1 Working Conference on Automated Tools for Information Systems Design and Development, New Orleans, 26-28 January, 1982. The paper presented at this conference, "An Integrated Environment for Program Visualization", appeared in Schneider and Wasserman (eds.), Automated Tools for Information Systems Design, North-Holland Publishing Co., 1982.

February 1982

Christopher Herot gave a presentation on PV at a meeting of the Northeastern ACM Chapter on February 18.

June 1982

Mark Friedell presented a talk on PV and the VIEW Project at the workshop on Automated Explanation Production. This conference was sponsored by the University of Southern California Information Sciences Institute and was held at the university's Idylwild Campus.

July 1982

Craig Fields and Clint Kelly of DARPA visited CCA on July 6. Christopher Herot and Gretchen Brown gave a short status report on the PV project, and Richard Carling gave a demonstration of the window management software.

OTHER RESEARCH TASKS

Sponsor: NAVELEX
Contract Number: N00039-83-C-0208
Title: Database Interfaces for USS Carl Vinson
Amount of Contract: $691,017

Sponsor: ONR/DARPA
Contract Number: N00014-81-C-0592
Title: Transfer of SDMS to USS Carl Vinson
Amount of Contract: $200,000

PARTICIPANTS

Chrisopher F. Herot
Jane Barnett
Gretchen P. Brown
Richard T. Carling
Mark Friedell
David Kramlich
Steven Zimmerman

Consultants:

Becky Allen
Ronald M. Baecker
Aaron Marcus

STATUS REPORT (N00014-81-C-0456)

8 November 1982 - 7 February 1983

This report summarizes the work completed during the seventh quarter (8 Nov 1982 - 7 Feb 1983) of the Program Visualization project under contract N00014-81-C-0456.

1. IMPLEMENTATION WORK

The first phase of implementation of the Program Visualization (PV) Library is now complete, with basic store and access functions supported. In addition, the system supports automatic library access of appropriate code when the user points to a graphic symbol from the library. Also supported for standard datastructures is automatic library access of graphic depictions when the user points to code he or she has typed in.

We made incremental progress this quarter on implementation of the PV Graphic Editor and further integration/upgrading of the PV Text Editor.

A new system of graphical menus has been installed. Considerable design work went into organizing the large number of commands provided by the system. The new menus were produced using the PV Graphic Editor.

Animated line highlighting is now supported for C code, i.e., lines of code are highlighted as they are executed.

2. DESIGN OF GRAPHIC SYMBOLS

We have been designing a number of graphic symbols for different parts of the system. We now have a first version of a set of four symbols to be used as logos for the different PV navigational aids, as well as about a dozen symbols to mark major categories of operations on the PV menus. Work on graphic symbols to signify different standard datastructures is also underway.

An initial version of the notation used for system architecture diagrams in CCA's Multibase Project is now supported. A "kit" of symbols allows the user to

construct nodes in this notation by copying rather than drawing. Connectors are drawn by the user via the general make-connector command.

3. SITE VISITS

Bob Grafton of ONR visited on December 9. We gave him an extended PV slide presentation and a demonstration of the system.

Clint Kelly of DARPA visited on January 13, and he talked with us for some time about PV. We showed him some slides of the current state of the system, along with a demonstration.

4. PAPERS AND PRESENTATIONS

We were notified of acceptance of the position paper we submitted to the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, which will be held in March of 1983. We also wrote and submitted a paper to the ACM IEEE Design Automation conference to be held in June of 1983.

On January 11, We gave a talk on PV at Intermetrics as part of their lecture series. There was a good-sized audience of 50-100 people, and there seemed to be a lot of interest in our work.

5. VIDEOTAPE COMPLETED

A videotape that demonstrates some key aspects of the current PV implementation was completed in February. The tape, which runs approximately ten minutes, gives an overview of the system and then shows how a programmer could use PV both to compose code and to construct an associated dynamic visualization. The dynamic visualization includes both highlights moving through lines of code and graphic depictions of variables being updated.