SOFTWARE VISUALIZATION TOOLS FOR JAVA

by

STEPHEN FILSON SAMPSON

B.Sc., Mount Saint Vincent University, 1995
B.C.S., Acadia University, 1998

Thesis
submitted in partial fulfillment of the requirements for the
Degree of Master of Science (Computer Science)

Acadia University
August 2000

0-612-54538-5

Table of Contents

List of Figures

Software Visualization Tools for Java

Abstract

The field of software visualization exists to facilitate both the human understanding and effective use of computer software. This thesis surveys over twenty modern software visualization systems to acquire information about the current state of software visualization systems. This knowledge is then used in the design and implementation of a new system called Steve's Software Visualizer (SSV).

SSV is a program visualizer. SSV has all the functionality of a debugger, for example, setting break points, a call stack and evaluation of variables. SSV also uses generic software visualization tools: Seeview, Classview, Textview. These tools can be operated interactively by the user, or viewed passively as an animation. By using all three views in combination, certain aspects of the software can be "visualized". This is not possible through traditional text based debugging methods.

1. INTRODUCTION

Modern software visualization began in the early 1980's with the introduction of the bit-mapped display and window interface technology [Price 1993]. Graphical workstations allowed researchers to create systems for visually exploring graphical representations of software. For the first time, dynamic, as opposed to static, representations of software and data structure were available allowing researchers to better understand and see the structure of their programs.

Software is generally created using textual symbols to signify data and operations. These representations, known as source code, are translated into a form the computer can understand. Software visualization tools, such as the one built for this thesis, put the source code in a form that a programmer can better understand by displaying the structure and true dynamic nature of the program.

Currently most source code is created, edited, and displayed from within an integrated development environment (IDE). Modern IDEs have the ability to maintain an enormous amount of source code and text that must be analyzed to determine what the symbols represent. The use of a software visualization tool makes the meaning of the code more apparent and concrete, while making the overall structure of the program easier to grasp.

This thesis describes a software visualization tool for Java, named Steve's Software Visualizer or SSV. The visualizer has the functionality of a debugger as well as having two new software visualization tools. The debugger allows setting of break points and evaluation of variables, while the software visualizer displays program structure using graphical visualizations. Also, SSV is able to animate and step through a running program free of user interaction.

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of software visualization. Chapter 3 reviews various related research and applications. Chapter 4 presents a proposed visualization tool for Java called Steve's Software Visualizer (SSV). Chapter 5 presents an implementation of this proposed tool. Chapter 6 presents proposed future work on the visualization tool. Chapter 7 summarizes the results of user testing of the SSV system. Chapter 8 is the conclusion of the thesis.

2. AN OVERVIEW OF SOFTWARE VISUALIZATION

2.1 DEFINITION OF SOFTWARE VISUALIZATION

Software Visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer-graphics technology to facilitate both the human understanding and effective use of computer software [Price 1998].

2.2 PROGRAM VISUALIZATION

Program visualization is the visualization of actual program code or data structures in either static or dynamic form [Price 1998]. This thesis primarily deals with this branch of Software Visualization. NV3D is an example of a program visualization system [Parker 1998].

2.3 ALGORITHM VISUALIZATION

Algorithm visualization is the visualization of the higher-level abstractions which describe software [Price 1998]. Algorithm visualization deals with showing an abstract representation of an algorithm. Visualizations usually show the data and the effect on that data as the algorithm runs. It is high level in its representation of the algorithm, but at a lower level in that it shows only a specific aspect of a program. Algorithm visualization use is primarily in teaching environments. However several applications exist for experienced programmers. Generally much work is required to create algorithm animations or abstract views because a generic view can not be used for all algorithms.

2.4 ALGORITHM ANIMATION

Algorithm animation is dynamic algorithm visualization [Price 1998]. It is any mechanism which presents the running of an algorithm as a movie where the visual representation of objects of the program smoothly change their location and appearance, according to a script determined by the algorithm [Lahtinen 1998]. The algorithm visualization is created and then put into motion based on time or events within the algorithm. The algorithm animation is often similar to watching a cartoon. BALSA is an example of an algorithm animation system [Brown, 1984].

2.5 PROGRAM AURALIZATION

Program auralization is the use of sound to assist in the formation of mental images of the behavior, structure and function of a program or algorithm. Sound is used instead of painting an abstract picture with the arrangements of pixels and colour on the computer monitor [Francioni 1991]. Different tones, pitches and volumes represent events within a program. Program auralization is traditionally used in combination with visual displays of computer graphics. An example of such a system is the SonicFinder [Gaver, 1989].

2.6 VISUAL PROGRAMMING

Visual programming is a field of software visualization. It is a type of programming, which uses graphical objects to build software [Price 1998]. The visualization is built first, and then the executable code is derived from the abstraction. Either source code is derived from the representation and then compiled, or the graphical representation is compiled directly with no generation of textual source code. An example system is Prograph [Cox 1989].

2.7 BRIEF EVOLUTION OF MODERN SOFTWARE VISUALIZATION

The first major work of modern software visualization research was in 1981 [Baecker, 1981]. The work was a 30-minute, narrated, colour motion picture displaying how nine different sorting algorithms manipulate their data, entitled "Sorting Out Sorting". In 1984, the most well known and important early interactive system, BALSA [Brown, 1984], was announced. This was followed by BALSA-II [Brown, 1988]. BALSA is an algorithm animation system that allows the user to create real-time simulations of programs (as opposed to movies) using high-resolution graphics. BALSA's interactive capabilities inspired the development of many other systems. During this same era, Myers [Myers 1988] carried out pioneering work in automatic data visualization, which integrated debugging capabilities with software visualization tools.

In the early 1990's, more research was done on algorithm animation systems such as TANGO [Sasko, 1990], Zeus [Brown, 1991] and the LENS system [Mukherjea, 1994]. In the late 1990's, research has focused on parallel programs because software visualization lends itself nicely to the complexity of parallel programs. Systems such as VisuaLinda [Kaoike, 1997], or Program Explorer [Lange, 1997] are examples of parallel visualizers. Other recent work includes the development of interactive software visualization environments for teaching [Merlini, 1999], as well as commercial applications for viewing class structure and interactions [NVision, 1999].

2.8 JAVA

Java, is an object-oriented programming language released by Sun Microsystems, Inc. The Java software development kit was originally released with a command line interface. However, many integrated development environments for Java have been developed since. Many IDE support the addition or co-existence of third party tools. This thesis introduces a tool for representing abstract automatic program visualization for Java.

3. SURVEY OF VISUALIZATION WORK

This chapter reviews twenty-two software visualization applications and related research. This previous work is reviewed to help understand the current state of software visualization and to apply that information to the creation of a new system which contributes to the understanding of software.

In the chapter, three concepts are reviewed: the fisheye-view, program auralization, and semantic zooming. All of these can be applied to both algorithm animation or program visualization systems. Six algorithm animation systems are reviewed: BALSA, Tango, Zeus, CAT, ARMVLS and EVA. The Eliot system can be classified as both an algorithm animation system and a data visualization system, whereas Lens is both algorithm animation and program visualization system. Nine program visualization systems are reviewed: See, Polka, NV3D, Program Explorer, Visualinda, ZStep 95, Pie, Field, and SeeSoft. As well as two data visualization systems: AMETHYST and VIPS.

Before the survey begins, three terms must be made clear: user, animator, and programmer. A *user* is a person who uses visualization, or a visualization package, to better understand and visualize a particular piece of software. An *animator* is a person who builds the software visualization for the user. However, the animator role may be further subdivided into scriptwriter, algorithm designer, graphics programmer, etc., which differ from system to system. For that reason, and for the sake of clarity and simplicity, a person who builds visualizations will be henceforth known as an animator. The word *programmer* can be applied to both the user and animator, depending on the context or current role of the individual. Therefore, when the word programmer is used in this thesis, it will be made clear which role is being played.

3.1 SEE

Baecker developed the SEE visual compiler [Baecker 1998] as an aid to communicating information about programs and comprehension of programs by paying attention to the visual schema embodying the program and the visual appearance of programs. Baecker believes the presentation of program source text matters and that effective program presentation portrays program structure, helping the user deal with its complexity.

SEE is a prototype that uses graphic design principles to create a project manual. SEE takes unmodified C source text as input, and produces high quality typeset presentations on a laser printer. This output includes headers, footnotes, metadata, different indices, fonts and annotations. This automated program presentation is intended to produce a significantly better program presentation. The compiler is heavily parameterized to allow the customization of text display to suit individual taste.

A drawback of a system such as SEE is that the project manual that is created is out of date as soon as any changes are made to the project. For large projects that are modified daily, personal experience has shown that an updated manual is rarely available because of the time and expense of creating the manual. Also there is a great waste of paper that occurs because the manual will be out of date again as soon as the source code changes. However, once a project is complete, a comprehensive manual is invaluable.

SSV addresses this downside by reading the current files directly and giving an on-line representation of the code. Therefore, the reporting of information is always current.

3.2 POLKA

Stasko and Kraemer [Stasko 1993] developed a visualization methodology to address requirements for application-specific viewing of parallel programs. The methodology is called POLKA (Parallel program-focused Object-oriented Low Key Animation) and it is an object-oriented basis of visualization and animation that includes high-level graphical object and motion primitives.

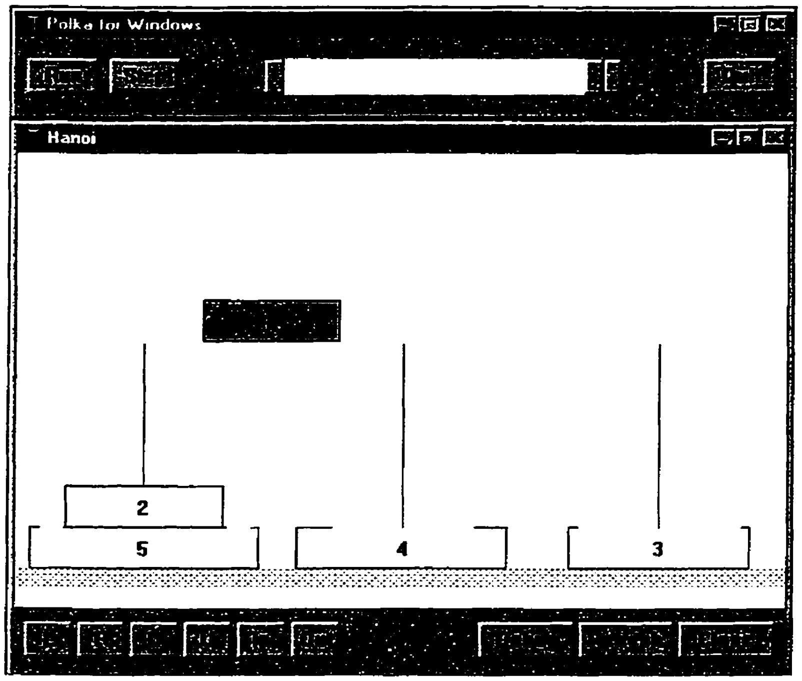


Figure 3.1 POLKA animation of towers of Hanoi

POLKA is a general-purpose animation system that is particularly well suited to building animations of programs, algorithms and computations, especially parallel computations. POLKA supports colour, real-time, 2 dimensional, smooth animations. The focus of the system is on a balance of power and ease-of-use. POLKA provides its own high-level abstractions to make the creation of animations easier and faster than with many other systems. Programmers need not be graphics experts to develop their own animations. POLKA also includes an interactive front-end called SAMBA that can be used to generate animations from any type of program that can generate ASCII text.

3.3 BALSA and BALSA-II

The Brown University Algorithm Simulator and Animator (BALSA) [Brown, 1984] and its descendant BALSA-II [Brown, 1988] were among the first interactive algorithm animation systems. BALSA creates real-time simulations of programs (as opposed to movies) using high-resolution graphics [Brown 1984]. An animator interactively creates a simulation through the BALSA interface. The user simply initiates the desired simulation and interacts with or watches the simulation. "Essentially, BALSA may be thought of as a laboratory for experimentation with dynamic real-time representations of algorithms" [Brown, 1984].

From the user perspective, he or she selects an algorithm from the pull down menu. Users are able to start and stop execution, as well as select different views for the algorithm. Brown states, "A fundamental thesis of an algorithm animation system is that a single view of an algorithm or data structure does not tell a complete story"

[Brown 1988]. Therefore the user is able to select different view, such as point or bar displays.

BALSA is noteworthy for two capabilities. 1) Interpretive runtime system - which allows a user to start, stop, or even run a simulation backwards. 2) Command shell - which allows a user to save, restore or invoke scripts on the current executing algorithm simulation. The greatest drawback of BALSA, which Brown admits, is the overhead required to build a visualization.

To animate an algorithm, the algorithm is annotated with "interesting events" that identify its fundamental operations that are to be displayed. Interesting events are triggers that are placed within the algorithm that lead to changes in the image being displayed. When an algorithm is run under BALSA, an interesting event is fired which instructs the graphics package to change the image.

Brown admits the learning curve for a BALSA programmer is probably a bit steeper than that of other algorithm animation system, however the extra effort seems defensible given Balsa's facilities for manipulating program displays and execution, and for scripting.

SSV strongly addresses the issue of the learning curve. All that is necessary to use the SSV system is the operator of the system must compile his or her programs to generate all debugging information. Then the user simply utilizes the mouse and the keyboard to interact with SSV.

3.4 TANGO

Tango [Stasko 1990] is a popular framework and system for algorithm animation. Tango was designed to provide a clean, powerful, and flexible algorithm animation system with formal models and precise semantics. Tango makes iterative design easier by separating program abstraction from animation design and making animation actions easily and directly accessible. Using Tango, a programmer can create a new animation in a few hours or days rather than many days or weeks.

To produce an animation with Tango, an animator must annotate the program with the necessary algorithm operations. Animation scenes to implement the animation actions must be designed by assembling collections of image, location, path, transition and association operations. Finally a control file must be created to specify the mapping from the algorithm operations to the animation scenes.

TANGO's most outstanding feature is its ability to produce path-transition paradigm animations. That is, animations that use smooth transitions instead of instantaneous swaps. For example, in a sorting algorithm visualization, instead of two elements instantaneously swapping, the objects of the sort physically move along their own path, pixel by pixel, until the objects have traded places.

SSV does not support path-transition. While the paradigm is sound, it is more appropriate at the algorithm level and not at the class or program level which SSV targets.

The largest drawback of Tango is the time it takes to create animations. SSV addresses this by providing generic views instead of requiring the user to create his or her own views.

3.5 LENS

Lens [Mukherjea 1994] is a combination of algorithm animation and program visualization systems. Lens has the capability to build and display animator created simulations and has the ability to automatically display data structures. Lens is implemented on top of UNIX, the X Window System, the XTango animation system, and the debugger dbx. This integration with a system debugger promotes iterative design and exploration. There are capabilities for setting break points, and viewing variable values.

Interacting with the system creates animations, and some coding is needed, but the amount of work required by the animator to achieve a visualization is less than coding an animation from scratch, which is most common with algorithm animation.

The purpose of the Lens system is to bridge the two domains of program visualization and algorithm animation. Lens can provide application-specific animation views for debugging purposes. Programmers are encouraged to design animations, but should not be troubled by learning a graphics toolkit and writing code to use it.

SSV follows the design of the Lens system in that it is implemented on top of a debugger. The ability to set a break point and then proceed with self-exploration is critical in learning a new piece of software.

3.6 ZEUS

Zeus [Brown 1991] is noteworthy for its use of objects, strong typing, parallelism, and graphical development of views. It was one of the first visualization systems to use sound and colour in algorithm animation.

From the user's perspective, invoking the Zeus application opens a control panel on the screen. The control panel provides the user with configuration and interpretive facilities. The configuration facilities allow the user to select which algorithm to run, which view is to be used and the data for the algorithm. The interpretive facilities allow starting, stopping, and single-stepping an algorithm.

To a programmer, Zeus is a framework for associating multiple client-defined views with a set of client-defined events. Zeus is a set of classes written in a in-house dialect of Modula-2.

Zeus does not have any sophisticated graphics, or specially built graphical editors, but it does allow the algorithm animator to graphically demonstrate how an instance of an object used by a view should look.

Brown states in [Brown 1991] that "constructing animations in Zeus appears to be as easy and straightforward as in any other algorithm animation system".

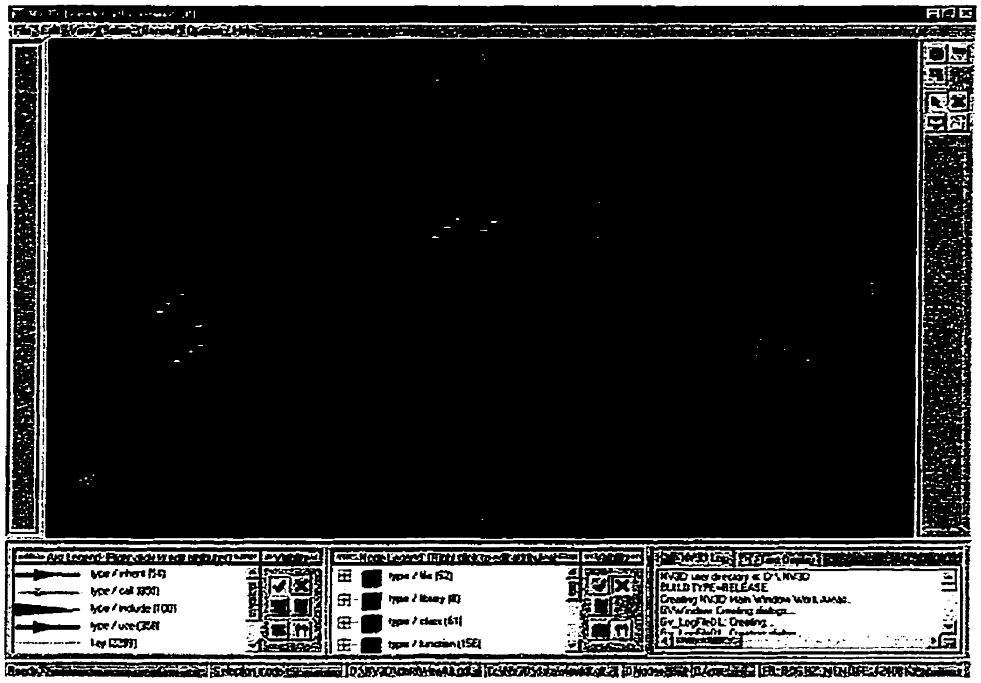
Zeus can generate some utility views automatically based on a set of interesting events that an algorithm generates. From the user's perspective, Zeus is similar to other algorithm animation systems in that an animator is needed to create visualizations, while a user simply runs and interacts with the animator's creation. The animator must associate a set of interesting events with multiple graphical views.

3.7 NV3D

NestedVision3D (NV3D) is a system for visualizing large nested graphs using interactive 3D graphics [Parker 1998]. NV3D is available commercially and in different flavors. The version of interest to this thesis is the object-oriented software-visualizing package, which does program or class visualizations. Nodes in the graphs represent entities, such as methods, modules, or objects, while arcs represent relationships between entities, such as inheritance, or usage.

NV3D uses 3D representation, rapid navigation techniques and nested graphs to help visualize the software. A user is able to see as much or as little information as desired by rotating, zooming, expanding or minimizing nodes. A single class, for example X, can be selected and explored alone (i.e. X is the only class on the screen) or class X can be viewed and explored in relation to all or some of the other classes.

An interesting feature of a different version of NV3D [Parker 1998] is the "snake". Dynamic behavior is shown as a snake, which is animated and travels from one end of an arc to the other.



```
NV3D user directory is D:\NV3D
BUILD TYPE=RELEASE
Creating NV3D Main Window Work Area...
GVWindow: Creating dialogs...
Gv_LogFileDL: Creating...
```

Figure 3.2 NV3D

As mentioned above, NV3D provides generic class and program views. Animators and programmers are not needed and therefore NV3D provides no programming or view editing capabilities. The generic views provide a common interface for all classes and programs that are viewed with NV3D. This common interface promotes familiarity between different programs being visualized and therefore lowers the learning curve when compared to other systems with custom views for each new algorithm, class or program. SSV attempts to capture the essence of NV3D by using generic 3D class views.

3.8 AMETHYST

Amethyst [Myers 1988] stands for A MacGNome (programming) Environment That Helps You See Types. Amethyst was designed for use in an instructional environment to help the students visualize and understand data structures. Therefore Amethyst is designed with students in mind and is easy to use. The representations Amethyst creates are similar to those found in popular data structure textbooks and are created automatically with no animator or programming required. However programming facilities exist to create advanced, custom views.

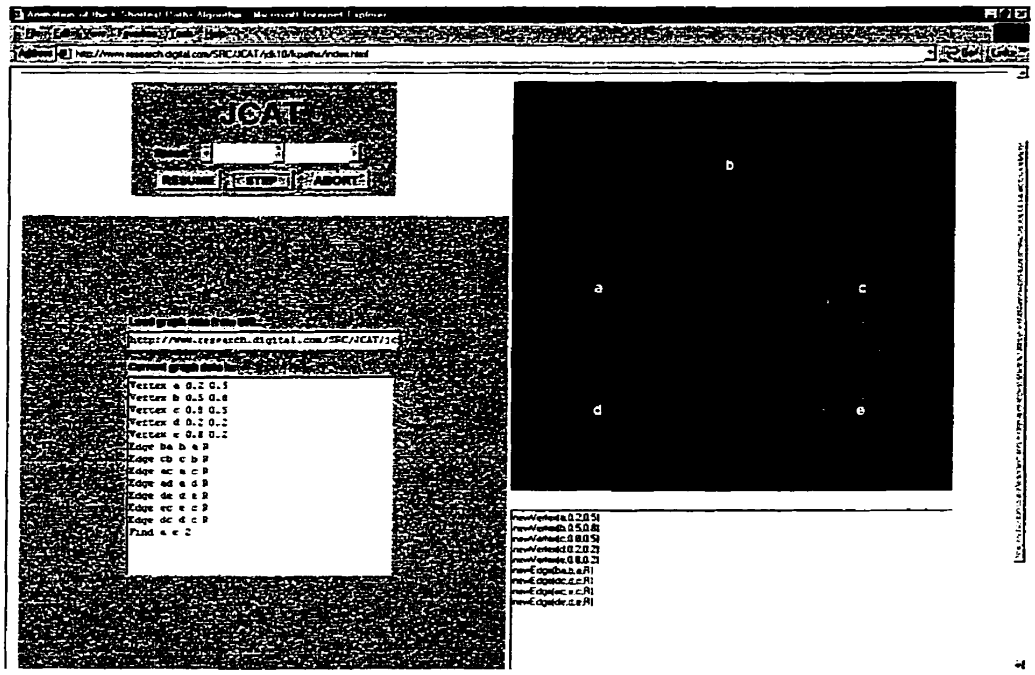
The primary focus of Amethyst is to provide appropriate displays of data structures automatically. These views are updated continuously, so the user never sees an inconsistent view of the data. Users can display a graphical view of the data simply by selecting a variable in the program text and issuing the "Show Value" command from a menu.

The visualizations are integrated into an advanced programming environment that provides a structure editor interface, which automatically inserts the appropriate syntax when the user specifies the type of program structure desired. The integrated system also provides multiple views of the program being edited, such as an outline view, run-time call stack, two different tree-structured decomposition views, and the standard linear program views. A user is also able to set breakpoints.

Amethyst is a model for SSV. Amethyst provides ease of use and automatic data visualization, while SSV is easy to use and provides automatic class information.

3.9 CAT

CAT [Brown 1996], short for Collaborative Active Textbooks, is a web-based algorithm animation system for an electronic classroom. CAT is a collection of web pages that contain text and passive multimedia as well as "active objects" (which are like Java applets). The system can be active, so that a reader can interact with parts of the textbook or the system can be collaborative in that a group of people, such as a teacher and a set of students in an "electronic classroom" setting can share a common interactive experience. The instructor can control the animation for all, or the students can run their own.



```
http://www.research.digital.com/SRC/JCAT/jc
Vertex a 0.2 0.5
Vertex b 0.5 0.8
Vertex c 0.8 0.5
Vertex d 0.2 0.2
Vertex e 0.8 0.2
Edge ba b a 8
Edge cb c b 8
Edge ac a c 8
Edge ad a d 8
Edge de d e 8
Edge ec e c 8
Edge dc d c 8
Find a e 2
```

```
newVertex(a,0.2,0.5)
newVertex(b,0.5,0.8)
newVertex(c,0.8,0.5)
newVertex(d,0.2,0.2)
newVertex(e,0.8,0.2)
newEdge(ba,b,a,8)
newEdge(dc,d,c,8)
newEdge(ec,e,c,8)
newEdge(de,d,e,8)
```

Figure 3.3 CAT

The algorithm and the views are implemented in Obliq, which is an interpreted object-oriented language. CAT runs through a browser which is capable of displaying multiple, simultaneously animated views of an algorithm.

Cat follows the BALSA approach: strategically important points of an algorithm are annotated with procedure calls that generate "interesting events". The interesting events are then passed to each view that responds to the event by drawing appropriate images.

3.10 PROGRAM EXPLORER

Program Explorer [Lange 1997] is a tool to reduce the amount of information presented to a debugging programmer, and is also as an aid to improving a programmer's understanding of the system of interest. Program Explorer reduces the amount of information by merging, pruning or slicing away information. This allows the user to concentrate on only relevant information. Once the data is obtained, Program Explorer has a very advanced visualization display and interface that allows the user to interact with the system.

Program Explorer is very interactive and powerful with many low-level data collection features. It is based on IBM's xlC compiler and uses IBM's Heapview Debugger to monitor objects.

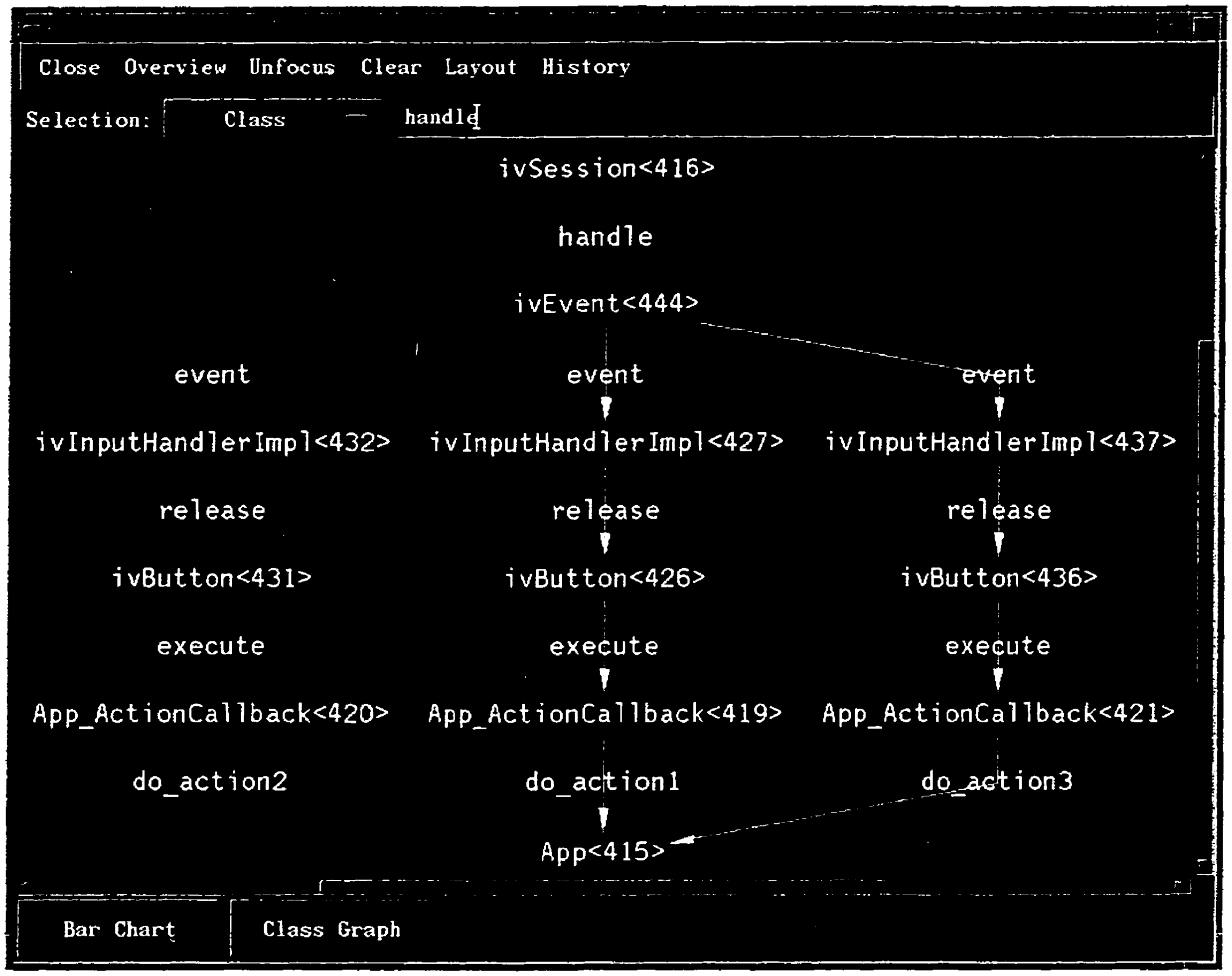


Figure 3.4 Program Explorer

Program Explorer reduces the amount of information presented to a user. SSV was designed to be simple, yet powerful. However, during user testing (see chapter 7) users complained there was too much information presented in the 3D class view. Future work will strive to include reduction facilities such as those found in Program Explorer.

3.11 VISUALINDA

The VisuaLinda system [Kaoike 1997] is an integration of a Linda server and a visualizer of parallel Linda programs. The visualization module is built in the Linda server, therefore programmers do not need to put additional visualization primitives (i.e. indicate "interesting events") in their client programs in order to visualize behavior. This integration helps the programmer debug parallel Linda programs by minimizing the "probe effect," which is one of the main concerns in monitoring parallel programs. Also, VisuaLinda uses automatic three-dimensional views to display the relationship between the Linda server and the client programs, as well as the execution of client programs.

VisualLinda was designed so programmers can find a bug simply by seeing the visualized output. Programmers can observe inter-process communications as well as other information to see when an error occurs. The framework can also display each process's state in addition to an overview of program execution.

SSV does not use probes similar to VisuaLinda. Laboriously adding "interesting events" and then removing them once a bug is found is not efficient. This is especially significant if a similar bug is found and the events have to be re-entered.

3.12 ZSTEP 95

ZStep 95 [Lieberman 1998] is a program-debugging environment designed to help the programmer understand the correspondence between static program code and dynamic program execution. ZStep 95 was designed to support the problem-solving methodology of matching the expectations of a programmer concerning the behavior of code to the actual behavior of the code.

ZStep 95 is notable for its animated view of program execution using the very same display used to edit the source code, one-click access from graphical objects to the code that drew them and as well as one-click access from expressions in the code to their values and graphical output. However, ZStep 95's most interesting feature is its ability to incrementally generate a complete history of program execution and output. With this history, ZStep 95 has the ability to run a program in forward and reverse directions while controlling the speed and level of detail displayed. This reversible control structure allows the user to temporarily ignore the details of a particular expression, however if the need presents itself, ZStep can be backed up to look at the details.

The reversible control structure also effectively handles the common software visualization problem of too much detail. Using ZStep, a user can quickly skim over code until an error or bug occurs. Once found the user can back up slightly and explore in more detail potential trouble spots in the code.

3.13 VIPS

VIPS [Shimomura, 1990] is a visual debugger for list structures. It makes use of the Sun Microsystems dbx debugger to help visualize the execution of programs written in the C language. VIPS implements a multiple window/view mechanism to realize such facilities as: (1) displaying the control flow in both the program text and a module structure chart, (2) displaying the call stack as graphical objects, and (3) displaying data structures as images which represent data semantics.

The VIPS system (version 2), unlike earlier versions of the same system, can acquire data type information necessary for automatic display of data structures. However, the only constructs VIPS can display are list structures because the authors thought that list structures are the most difficult type of data structure to debug. VIPS displays these structures as rectangles containing text and arrows pointing from one rectangle to the next.

Debugging is further aided through the use of multiple windows displaying program information such as a monitor, program-text, list, input-output, editor, variable display and stack display windows. As well, many different views of the list are available such as whole or partial lists, element display and opening multiple views of the same list from different aspects.

Unlike VIPS, SSV does not do any data visualizations. However like VIPS, SSV has multiple views of the source code from different aspects. This gives the user the same information, but with a different presentation.

3.14 PIE

The Parallel Programming and Instrumentation Environment (PIE) [Lehr 1989] is a parallel programming environment designed for developing performance-efficient parallel and sequential computations. PIE provides programmers ways to observe how computations execute by making use of special development and runtime visualization tools. These tools allow for automatic assistance for visually projecting performance data onto programming constructs. For example, a user can indicate, through the use of a graphical representation of the code, where the operations that enforce mutual exclusion occur. The system then automatically observes the execution by using multiple forms of instrumentation to gather statistical information. PIE presents the performance information in a variety of ways, including graphical representations of program constructs showing the progression of each process, histograms of process activity and event timelines.

PIE is designed to be an environment that presents information it retrieves about computations in forms that assist users in making their own qualitative judgements about how their computations behave. The framework helps develop techniques to predict, detect, and avoid performance degradation. PIE supports languages such as C, MPC, C-threads, Ada and Fortran.

It is the hope that SSV, like PIE, will assist users in making their own qualitative judgements about new source code.

3.15 FIELD

FIELD, the Friendly Integrated Environment for Learning Development was created in an attempt to use workstations effectively for UNIX-based programming [Reiss 1997]. FIELD integrates a wide variety of UNIX tools into a common framework. This framework uses ordinary UNIX tools with graphical user interface wrappers around them, as well as new tools to support both programming and program visualization. All these tools are connected by way of a message passing system connected to a database of program information

The UNIX tools in FIELD include configuration management (make), version control (rcs), as well as profiling tools. The visualization tools include a text editor that is augmented with a window that has clickable descriptive icons that give additional information. Other tools include graphical versions of a call graph browser, a class hierarchy browser, a data structure viewer, and a make dependency browser.

FIELD has a wide variety of visualizations for different applications. FIELD is able to show visualizations that represent the static structure of a system, and visualizations to show a system in action. FIELD effectively displays this information using limited screen space to display the large quantities of information inherent to a programming environment.

Unlike FIELD which has multiple, resizable, independent windows, SSV is designed as a single window application, which has multiple views of the program contained within the main window. SSV would benefit from a design similar to FIELD allowing better screen allocation and easier addition of new views.

3.16 LARGE DATA SETS AND SEMANTIC ZOOMING

One of the key open problems in software visualization is that most software visualizations are of smaller, laboratory-created programs [Stasko 1996]. That is, software visualizations do not scale up well, and they poorly portray large systems or program executions on large data sets. One proposed solution is the concept of semantic zooming. In brief, semantic zooming allows the user to zoom in/out or focus on a particular portion of the program or data set. Unlike a standard zoom, the presentation style of the view adjusts at the different zoom levels.

In the context of software visualization, [Stasko 1996] defines semantic zooming as follows:

- All visualizations begin showing a view of the entire data set of the program, usually at an abstract level due to the data size
- At some level, all of the program data should be visible without falling back on the use of scrolling and panning. That is, the presentation of all program data should fit within one window.
- Viewers interact with a view and zoom in on a portion of the program data by interactively selecting a graphical object representing that portion of the data.

- Different zoom levels or perspectives on the program data are shown either in the same window or in separate windows.
- At the lowest, most detailed view level, the visualizations should use recognized algorithm animations or program visualization presentation styles.
- All views update concurrently and always portray the current state of the program execution.

Stasko presents helpful and useful ideas for dealing with large amounts of information. In the design of SSV, some of Stasko's principles of semantic zooming were adopted with slight modification. For example all views update concurrently, zooming is possible, but sometimes the program image does not fit in one window therefore scrolling the image is necessary.

3.17 SEESOFT

The SeeSoft [Eick 1992] visualization tool displays line-oriented source code statistics by reducing each file and line into a compact representation. SeeSoft displays statistics using a rectangle to represent each file and coloured rows within the rectangle to represent the statistics associated with the lines of code. The position of the rows corresponds to the position of the lines within the file and the size of the rectangles to the size of the file. The resulting display looks like a very small representation of a code printout. See Figure 3.5.

Individual statistics are displayed with colour. These colors, chosen by the user, represent information stored in an elaborate database. For example, if the user wants to see information on what lines were added on a certain date, the user could have those line appear in red, while having all lines associated with a bug fix appear in a yellow colour.

Using high-interaction graphics and direct manipulation techniques, the user manipulates the display to discover interesting patterns in the code and statistics. Users of the system are immediately able to recognize the files and lines of code because the display looks like a text listing viewed from a distance.

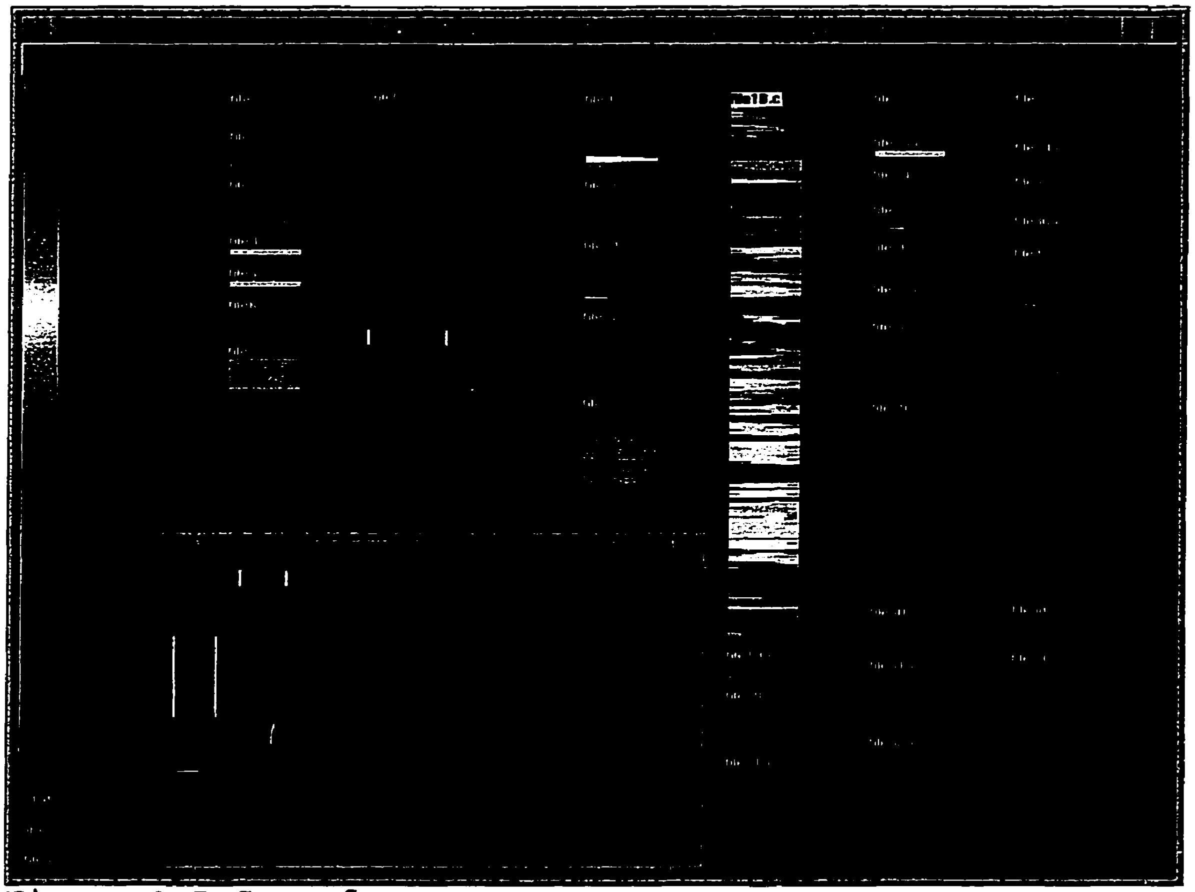


Figure 3.5 Seesoft

Seesoft is an effective tool because the display is informative and clear. Statistics are obvious from the row colours; code windows enable source code to be read, as well as provide an intuitive human interface. SeeSoft is capable of real-time screen updating in response to mouse actions. Moving the mouse over a file representation activates a menu of other statistics associated with the line or file. This technique works well because it allows the user to have both an overview of the statistic and also read the interesting parts of the code.

With SeeSoft's compact representation of data it is possible to comfortably display 35 files containing 50,000 lines of code on a 1280x1024 pixel display. As many as 100,000 lines can be displayed but the representation is tiny.

SSV uses the idea of an iconic file representation, not to display statistics, but to convey information about program flow.

3.18 ARMVLS

ARMVLS [Warendorf 1997], which stands for Atomic Reaction Model Visual Language System, is a visual language algorithm animator. ARMVLS is a system that allows the user to create images to visually demonstrate or to assist in the description of how a computer algorithm works by means of drawing and moving images on screen. The authors claim that the system bridges the fields of visual language programming and algorithm animation however ARMVLS would be better classified as a program by demonstration system.

ARMVLS is a visual programming system to animate algorithms that are themselves programmed in ARM (Atomic Reaction Model). ARMVLS can animate most of the algorithms traditionally done by textual coding. ARMVLS is easy to use and does not require an expert user. Programming illiterates can use the system and quickly create prototypes or useful algorithm animations. There are no special modes or specifically catered for types of animations. All features and constructs of ARM are generic and can apply to all algorithms that it can animate.

What distinguishes ARMVLS from other algorithm animation systems is that visual techniques, instead of textual codes, are employed to specify the animation sequence.

3.19 ELIOT

Eliot [Lahtinen 1998] can be used in algorithm design, visual debugging and learning programming. Eliot animates algorithms written in the C programming language by visualizing data structures as smoothly moving graphical objects. All the movements are connected with the operations of the data structures. The user selects a visual object from a pre-defined library of visual data types. The library includes basic types, like integer, and structured data types, like tree. Each visual data type has a set of visualizations associated with it. The user selects one visualization for each data object he or she wants to animation. Based on these selections, Eliot automatically constructs an animation where the objects as well as their operations are animated. The input C code and selected animation types are then compiled into an executable program.

Eliot was created as a tool for generating animations. The need for Eliot arose because Eliot's authors were spending 100+ hours creating simple animations using the tools that were available to them at the time. The result is a system which reduces the required work time down to just a few minutes.

Eliot makes two contributions to the field of program visualization. These are ease of use and an innovative implementation technique. However, the system requires an animator to create the visualizations, which is a step that SSV is able to skip.

3.20 EVA

EVA [Bykat 1996] was created to reduce the effort required in the production of software visualizations. EVA is an interactive Environment for Visualization and Animation of Programming Concepts. EVA was designed with the goal of providing an authoring environment for visualization and animation of programming concepts. The system integrates authoring, display and control system for the specification and execution of visualizations.

EVA has four distinct players that are involved in animation design, the teacher, the animator, the programmer, and the user. These players use the system in different ways. For example the teacher creates an analogy which means he or she must invent an effective and informative visualization. The animator must then produce the animation, and then the programmer incorporates the animation into a visualization. The user then studies and uses the visualized concepts to understand and generalize the meaning. To assist in these activities EVA provides an object editor and picture description language, as well as a mouse sensitive visualization interface which offers functions such as set/change parameters, redo, what-if, explain and trace.

3.21 FISHEYE-VIEW

The Fisheye-View is a strategy proposed by Furnas [Furnas 1986] that imitates a fish eye in order to display potentially huge structures on one computer monitor, and all associated information. Graphical representation of objects which are currently of interest appear focused and clear, while objects not directly in focus, around the outside are displayed successively smaller and less detailed. It achieves a smooth integration of local detail and global context by repositioning and resizing elements of the graph [Sakar 1994]. See Figure 3.6.

The following analogy was put forth by Furnas [Furnas 1986] as a way to explain the fisheye-view: When drawing a map, humans represent their own "neighborhood" in great detail, yet only major landmarks further away. The neighborhood is said to be in "focus", therefore many building, signs and roads are presented and visible, while the next town over may only be represented as a labeled dot.

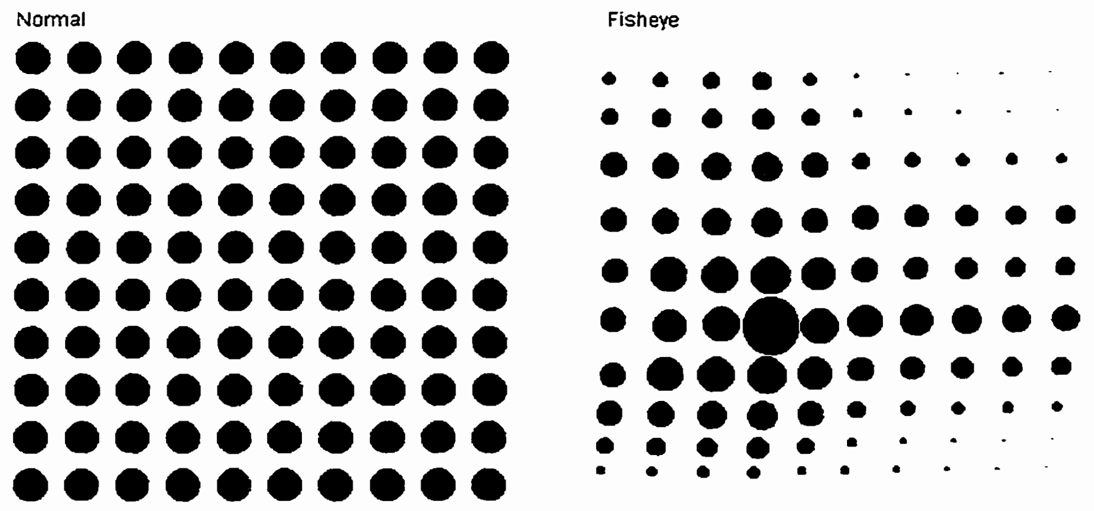


Figure 3.6 Normal Network Fisheye-View Network

The Fisheye-View has gained popularity [Sakar 1994 and Muchaluat 1998] as a way of displaying huge amounts of information. This type of view lends itself nicely to the huge amount of information available on the World Wide Web. See [Noik 1993] for more information on fisheye views of hypertext networks.

The fisheye-view has the potential to change the way large amounts of data are viewed in software visualization as well as in other fields. However, fisheye technology is still emerging and is not universally accepted.

3.22 PROGRAM AURALIZATION

Program auralization is the process of forming mental images of the behavior, structure and function of a program or algorithm using sound. Researchers have identified a number of reasons for using sound [Francioni 1991]:

- Visualization is highly subjective, and what is insightful for one person is meaningless to someone else. Program Auralization provides yet another "view" of a program; a view that might make some things obvious to some people. Furthermore, some types of information might just be difficult to represent graphically.
- Listening can be done passively. That is, one does not have to be paying strict attention listening to the normal behavior of a program in order to notice that some exceptional event has happened. Moreover, listening can be done in parallel with viewing.

- People have remarkable abilities to detect and remember patterns in sound (indeed, most people remember the melody of a song much sooner than he or she learns the words).
- Sound is a powerful medium for delivery of large amounts of data in parallel. This aspect of sound is especially useful for visualizing parallel programs; but even a sequential program can contain an enormous amount of data.
- Sound is inherently temporal, as are computer programs during execution.

All of the above reasons for using sound seem self-evident, yet sound is a rarely used method of visualization. While more software visualization system are beginning to incorporate sound, such as Zeus [Brown 1991], there is promising work in parallel computing [Jackson 1991], sound is not the norm. Visualizing software using sound is a slowly emerging field and as of yet there is not an abundance of research.

Other researchers [Brown 1997] have found that sound is more difficult to use than, say multiple views or colour, smooth animation, or even 3D graphics. Perhaps this is simply because we have less practice (and training) composing music than drawing diagrams. Perhaps we are unaccustomed to using sound as the primary input for problem solving. Or perhaps it is because sound is a more difficult medium to master.

3.23 SURVEY SUMMARY

Table 3.1 is a summary of the systems and research reviewed in this chapter. The abbreviations used in the "System Type" column are as follows: PV stands for "program visualization", DV stands for "data visualization" and AA stands for "algorithm animation".

Table 3.1 - Survey Summary

System Name or Area of Research	**System Type**	**Features**
See	PV	Compiler that creates a high quality typeset project manual from unmodified C source code.
Polka	AA & PV	General-purpose animation system targeted towards viewing of parallel programs.
Balsa and Balsa II	AA	Among the first interactive algorithm animation systems.
Tango	AA	Designed to provide a clean, powerful, and flexible algorithm animation system with formal models and precise semantics.
Lens	AA & PV	Combination algorithm animation and program visualization system. Implemented on top of the dbx debugger.

Zeus	AA	Noteworthy for its use of objects, strong typing, parallelism, and graphical development of views. One of the first systems to use colour and sound.
NV3D	PV	Uses 3D representation, rapid navigation techniques and nested graphs to help visualize the software.
Amethyst	DV	Designed for use in an instructional environment to help the students visualize and understand data structures.
Cat	AA	A web-based algorithm animation system for an electronic classroom.
Program Explorer	PV	A tool to reduce the amount of information presented to a debugging programmer.
VisuaLinda	PV	Designed so programmers can find a bug by seeing the visualized output. Programmers can observe inter-process communications as well as other information to see when an error occurs.
ZStep 95	PV	Program-debugging environment designed to help the programmer understand the correspondence between static program code and dynamic program execution. Can run a program in forward and reverse directions.
Vips	DV	Vips is a visual debugger for list structures.
Pie	PV	A parallel programming environment designed for developing performance-efficient parallel and sequential computations.
Field	PV	FIELD integrates a wide variety of UNIX tools into a common framework. The framework uses ordinary UNIX tools with graphical user interface wrappers around them, as well as new tool to support both programming and visualization.
Semantic Zooming	-	Semantic zooming allows the user to zoom in/out or focus on a particular portion of the program or data set. Unlike a standard zoom, the presentation style of the view adjusts at different zoom

		levels.
Seesoft	PV	Displays line-oriented source code statistics by reducing each file and line into a compact representation.
Armvls	AA	A visual language algorithm animator. ARMVLS allows the user to create images to visually demonstrate or to assist in the description of how a computer algorithm works by means of drawing and moving images on the screen.
Eliot	AA & DV	Can be used in algorithm design, visual debugging and learning programming. Visualizes data structures as smoothly moving graphical objects.
Eva	AA	EVA integrates authoring, display and a control system for the specification and execution of visualizations.
Fisheye-view	-	A strategy that imitates a fish eye in order to display potentially huge a structures on one computer monitor, and all associated information.
Program Auralization	-	The process of forming mental images of the behavior, structures and function of a program or algorithm using sound.

4. A VISUALIZATION TOOL FOR JAVA

4.1 MOTIVATION

Software visualization produces a mental picture. A programmer writes a piece of code, and in the "minds eye" he or she understands what it does and how it is supposed to work. Software visualization helps the programmer get a better understanding what a piece of software is doing by showing a graphic representation of the code.

The following quote serves as motivation and a goal in the design and implementation of a visualization tool for Java.

"A programmer will not use a tool for debugging whose development time outweighs that to simply debug a program with traditional text-based methods." [Mukherjea 1994]

4.2 DESIGN GOALS

Any tool will not be used if it creates more work than it saves. Some of the animation packages such as BALSA [Brown 1988], Tango [Stasko 1990] and Polka [Stasko 1993] require laboriously hand created animations. This may be acceptable for an algorithm animation system but not for program visualization. A visualization tool must be simple to use as well as easily and generically applied if it is to be useful to a developer.

The primary goal of this thesis is to define a tool that needs minimal user intervention to create a software visualization display and which shows both the dynamic and static behavior of the software being developed. As well, the visualization system is interactive to promote iterative design and exploration. Interaction is provided by debugger technology integrated with the visualization system.

4.2.1 EASY TO USE

A major goal of the system is ease of use. SSV provides predefined graphical views of the software. The user is able to simply point and click to see a visual representation of a generic (Java) program. No programming is required. This is in contrast to systems such as BALSA [Brown 1988] or Tango [Stasko 1990], which can be source code level intensive when building animations or using the system. These systems require user programming to create algorithm specific animations and views. However, in defense of these systems, they are designed for algorithm and not program visualization.

All that is required on the part of the user to utilize the system is use of the mouse or keyboard to interact with generic predefined views. Also, the user of the system must compile his or her programs to generate all debugging information. This debugging information embeds itself within the Java class file, enabling the debugger part of the visualizer to read extra information about the source code.

4.2.2 MULTI-THREADED

The tool supports multi-threaded programs insofar as parallel programs run under SSV. However, the multi-threaded nature of the program is not apparent in the visual display. Nonetheless, the implemented program could easily be modified to include graphical views of a concurrent nature.

4.2.3 GENERICALLY APPLICABLE AND SOFTWARE-PROBES

The software-probe approach requires programmers to insert function calls at various points within a program which allows interaction with the software visualization system. Once the bug is found, these procedures must be deleted. However, if another bug is found, programmers must insert them again and also delete them again after the debugging is finished. Most programmers have had similar experiences inserting many output procedures (e.g. "printf") into their programs while debugging and then deleting them afterwards [Koike, 1997]. Also, at the multi-threaded level of debugging, software probes can affect or change the outcome of running program. Therefore, no probes are introduced to existing code in this implementation. It is the job of the visualizing system to create the interesting events and generate a visualization. The system does not change any source code or class files. It only reads them and generically generates visualizations.

4.2.4 NO CUSTOM ANIMATIONS

The visualization system does not support custom user animations. Automatic visualization is paramount in this thesis. In this context, creation of custom animations and custom views distracts the developer from the goal of understanding the software. While a system such as Tango [Stasko 1990] is an algorithm animation system and was built for a different purpose than our implementation, it is the extra work of coding or learning of a graphic library that we wish to avoid in this thesis. Therefore, while these custom views do promote understanding, the time required to build these views is costly and would be better spent elsewhere.

4.2.5 INTEGRATED WITH A SYSTEM DEBUGGER

The visualizer is integrated with a system debugger to promote iterative design and exploration. The user is able to set breakpoints, animate the program, step through the program line by line, start or stop the software, as well as inspect variables.

The idea of integration with a system debugger is borrowed from the Lens system [Mukherjea 1994]. Lens is a combination of algorithm animation and a program visualization system that is built on top of the UNIX debugger dbx. It is the power the debugger gives to Lens that we want in this thesis.

4.2.6 ANIMATION

The use of animation in a software visualizer is extremely important because programs are fundamentally dynamic. The user is able to start the visualizer, sit back and watch an animated Text View, a three dimensional abstract class view, as well as an abstract two dimensional view at the method level of a chosen piece of executing software.

Animation is included in most modern software visualizers such as Zeus [Brown 1991], Lens [Mukherjea 1994], Cat [Brown 1996], as well as SSV. There are benefits to allow the user of the system to be a passive viewer of a running visualization system. Software visualization is highly subjective. What is obvious for one user is not for another. Therefore by allowing the user to sit back and watch a movie-like software visualization, he or she will gain insight into the software because of the different perspective.

4.2.7 TEXT VIEW

The text view of the source code is the traditional text view provided by standard debuggers. The text view has a cursor or indicator at the beginning of the currently executing line and updates itself as the program executes. If the program flow leads to a different file, the text view switches to reflect the change.

4.2.8 CLASS VIEW

The class view is the quintessence of the visualizing system. It is a three-dimensional representation of a loaded class. The generic view displays a class by showing all the methods associated with the class as well as all the methods of its super class(es). The current class is displayed closest to the user with a black color, while subsequent super classes appear further into the screen with a unique color. One exception to the way classes are coloured is the method that is currently executing. This executing method always glows with a yellow color. For example in Figure 4.1 the main method in ShapeDriver is currently executing. Furthermore, methods of different classes that share the same name, for example "String toString()", all appear vertically aligned. See Figure 4.1, Figure 5.4 and Figure 5.5 for examples of the class view.

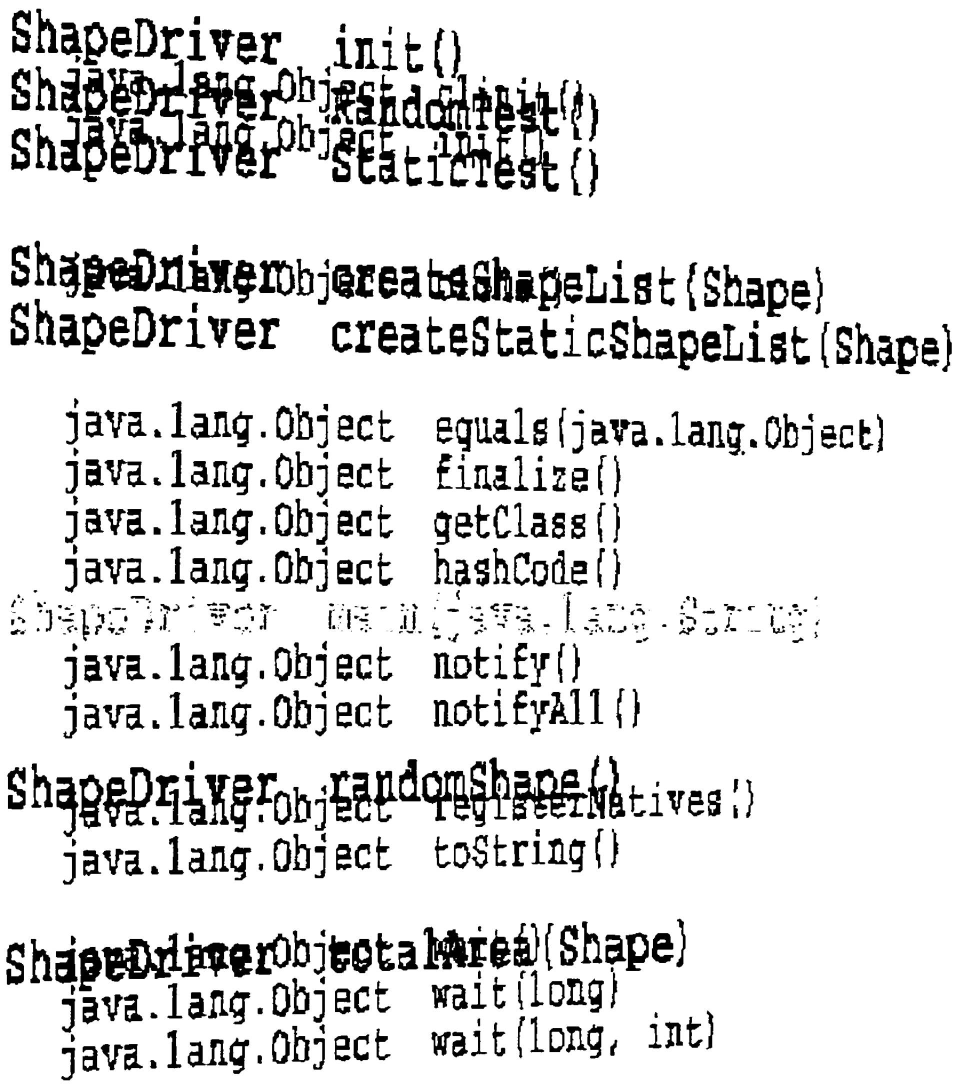

ShapeDriver init()
ShapeDriver randomTest()
ShapeDriver staticTest()
java.lang.Object clinit()
java.lang.Object init()
ShapeDriver createShapeList(Shape)
ShapeDriver createStaticShapeList(Shape)
java.lang.Object equals(java.lang.Object)
java.lang.Object finalize()
java.lang.Object getClass()
java.lang.Object hashCode()
ShapeDriver main(java.lang.String)
java.lang.Object notify()
java.lang.Object notifyAll()
ShapeDriver randomShape()
java.lang.Object registerNatives()
java.lang.Object toString()
ShapeDriver totalArea(Shape)
java.lang.Object wait()
java.lang.Object wait(long)
java.lang.Object wait(long, int)

Figure 4.1 Class view of ShapeDriver. Method main is currently executing.

The idea for the classview is taken from NV3D [NVision 1999]. NV3D uses 3 dimensional boxes and textual labels to represent classes and methods within the class. NV3D is able to display a great deal of information. However, because of the textual label for each method and class, the box is redundant and therefore is a waste of space since the label of the method provides an adequate object of representation. Also, the scenes within NV3D can be zoomed in or out and are highly interactive so our implementation copies this. See Figure 3.2. However what NV3D is noticeably lacking is dynamic interaction. It does not provide run-time information; NV3D provides only static information based on a database of information about the program structure.

As interesting and informative as these views are, it is the addition of animation that makes the visualizer truly powerful. As a program executes, and classes are displayed, the currently executing method "glows".

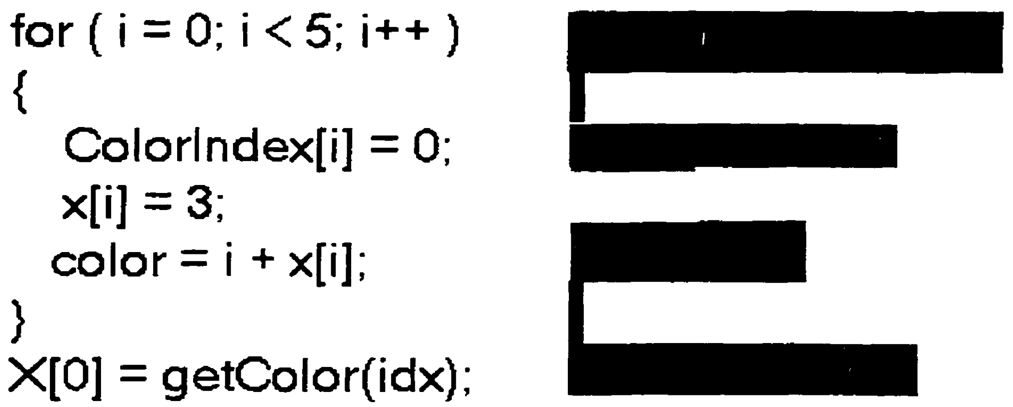
This three dimensional class view gives the user a different look at the code, hopefully allowing for greater insight into any relationships or problems.

4.2.9 SEEVIEW

The Seeview is the other major view of the SSV system. The general idea for the Seeview comes from the Seesoft system [Eick 1992]. See Figure 3.5. Seesoft interactively displays line-oriented statistics by colours as iconic views of a file. It is informative to display an iconic view of the file as the program executes showing the current line of execution because it gives a global view of the file. This

This global view gives the user a sense of program flow. Therefore the view is an abstract line oriented view of a file showing the current line of execution (Figure 4.2). Like the Method view, the currently executing line of code glows to indicate it is the current point of execution.

For each line of text within a source code file, there is a corresponding graphic line. See Figure 4.2 and Figure 5.1. However, figure 4.2 is somewhat misleading because when Seeview is displayed, each line is only one pixel high.



```
for ( i = 0; i < 5; i++ )
{
   ColorIndex[i] = 0;
   x[i] = 3;
  color = i + x[i];
}
X[0] = getColor(idx);
```

Figure 4.2 Example of line oriented Seeview

5. IMPLEMENTATION: FEATURES AND ARCHITECTURE

Steve's Software Visualizer (SSV) is platform independent. The Java compiler and interpreter are from the Java™ 2 SDK Version 1.2.2-001 Standard Edition released by Sun Microsystems. Sun's Java™ Platform Debugger Architecture (JPDA) provides debugging support. Three-dimensional graphics are provided through a package called Magician (an OpenGL implementation) released by Arcane Technologies Ltd.

5.1 THE JAVA DEBUGGER

The Java Debugger, jdb, is a simple command-line debugger for Java classes. The core of the jdb used in the SSV implementation is part of a demonstration package of the JPDA that provides inspection and debugging of a local or remote Java Virtual Machine. The jdb relies on the Java Debugging Interface (JDI), which is a high-level Java API providing information for debuggers and similar systems that require access to the running state of a virtual machine.

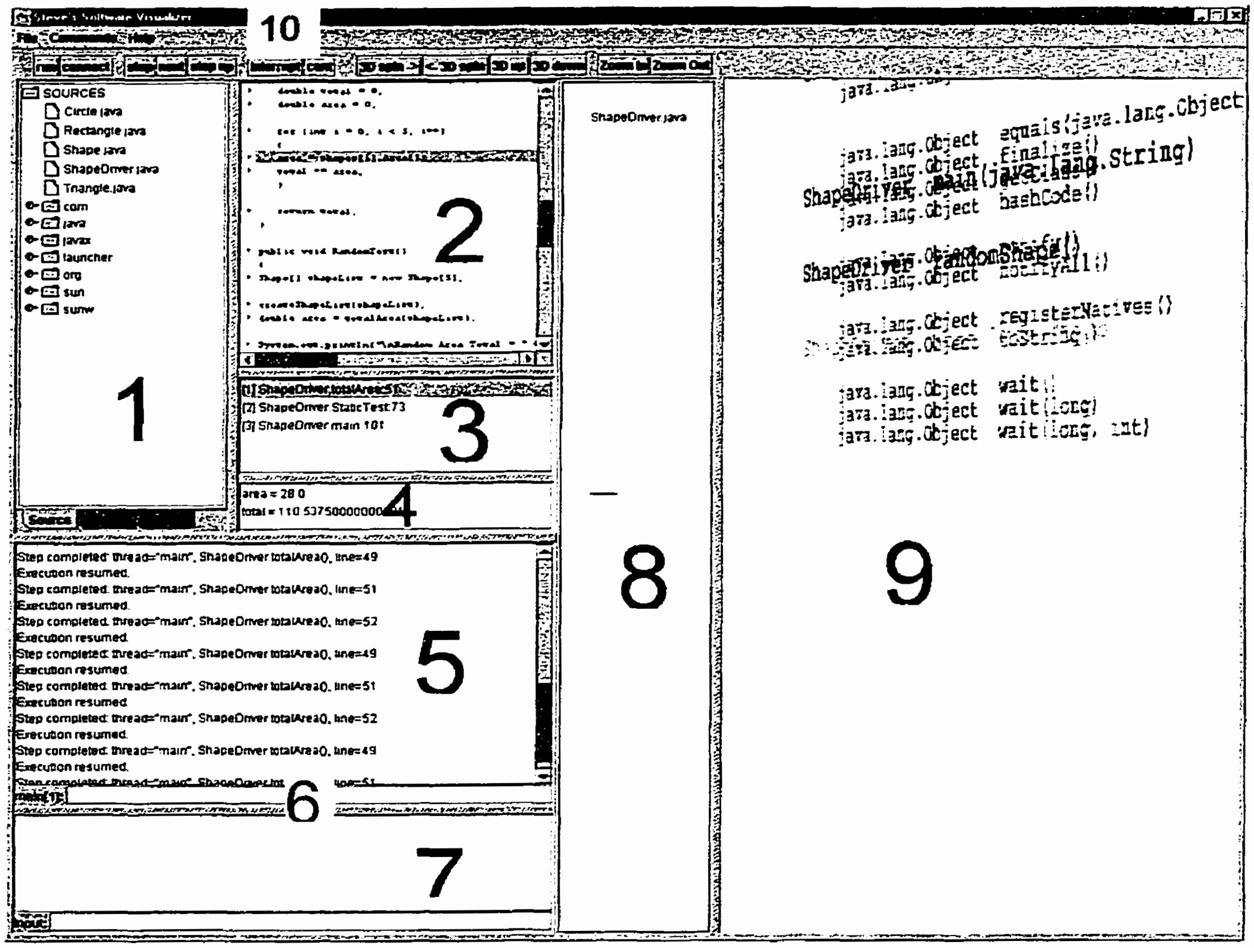
The foundations for this implementation are the JDI package and a debugger that is included. The example debugger includes a working Graphical User Interface.

5.1.1 TERMINOLOGY

- The debuggee is the process or application being debugged.
- The debugger, or visualizer, is a tool used to view and step-wise run the debuggee.

5.2 THE SOFTWARE VISUALIZER - SSV

The software visualizer created for this thesis, named Steve's Software Visualizer or SSV, is a fully automated software visualization tool for Java. See Figure 5.1 for a sample screen shot.


Steve's Software Visualizer
10
File Commands Help
run connect step next step up interrupt cont 3D spin > < 3D spin 3D up 3D down Zoom In Zoom Out
SOURCES
Circle.java
Rectangle.java
Shape.java
ShapeDriver.java
Triangle.java
com
java
javax
launcher
org
sun
sunw
Source
1
2
3
4
5
6
7
8
9
ShapeDriver.java
main[1]
java.lang.Object equals(java.lang.Object)
java.lang.Object finalize()
ShapeDriver main(java.lang.String)
java.lang.Object hashCode()
ShapeDriver randomShape()
java.lang.Object notifyAll()
java.lang.Object registerNatives()
java.lang.Object toString()
java.lang.Object wait()
java.lang.Object wait(long)
java.lang.Object wait(long, int)

```
    double total = 0;
    double area = 0;

    for (int i = 0; i < 5; i++)
    {
    area = shapes[i].Area();
    total += area;
    }

    return total;
  }

public void RandomTest()
  {
Shape[] shapeList = new Shape[5];

createShapeList(shapeList);
double area = totalArea(shapeList);

System.out.println("\nRandom Area Total = " +
```

```
[1] ShapeDriver.totalArea:51
[2] ShapeDriver.StaticTest:73
[3] ShapeDriver.main:101
```

```
area = 28.0
total = 110.537500000000
```

```
Step completed: thread="main", ShapeDriver.totalArea(), line=49
Execution resumed.
Step completed: thread="main", ShapeDriver.totalArea(), line=51
Execution resumed.
Step completed: thread="main", ShapeDriver.totalArea(), line=52
Execution resumed.
Step completed: thread="main", ShapeDriver.totalArea(), line=49
Execution resumed.
Step completed: thread="main", ShapeDriver.totalArea(), line=51
Execution resumed.
Step completed: thread="main", ShapeDriver.totalArea(), line=52
Execution resumed.
Step completed: thread="main", ShapeDriver.totalArea(), line=49
Execution resumed.
Step completed: thread="main", ShapeDriver.tot         line=51
```

Figure 5.1 Implementation of a software visualizer

1. <u>All available source code</u> - a display of all available source code.

2. <u>Source code</u> - currently executing Java source code - highlighted line is currently line of execution.

3. <u>Call stack</u> - shows program control flow.

4. <u>Variable monitor</u> - displays user specified variables and updates them as the program runs.

5. <u>Message window</u> - displays any system information.

6. <u>Command prompt</u> - a command line interface to the system - also available through menu and tool bar.

7. <u>Output window</u> - where System.out.println is displayed.

8. Seeview - a line representation of the Java source file. There is one line for every line of code in the Java source file. The black line indicates the current line of execution.

9. Classview - a 3D Class viewer that displays a class and all of its super-classes.

10. Menu and system buttons - the buttons and menus allow the user to interact with the system.

SSV displays a three-dimensional representation of a loaded class, while highlighting the currently executing method. See the right most windows of Figure 5.1 (labeled "9") and Figure 4.1. SSV displays the source code of the loaded class, provided it is available, in two forms:

1) The actual text (see window labeled "2" in Figure 5.1).
2) An abstract view of the entire file from which the source code originated (see window labeled "8" in Figure 5.1).

Operation of SSV requires the debuggee be generated with all debug information set to be included within the executable class file. This enables the debugging side of the visualizer to access the debug information and provide the unique views. Also, some configuration of SSV is required. For example the user of the system is expected to give the CLASSPATH and location of any additional source code.

5.3 SEEVIEW

SSV has a view known as Seeview which displays the source code of a loaded class, provided it is available, as a compact representation of the entire file from which the source code originated. Seeview simultaneously maps lines of code into thin rows. Each row is coloured light orange, while the currently executing line is coloured black. See Figure 5.2 for four examples of loaded classes displayed by Seeview.

As the visualizer executes, the debuggee program steps through and executes line-by-line. As each line executes, Seeview changes to the currently executing file, as well as updating the black coloured current line indicator.

The user is not involved with or distracted by the syntax of the code, instead the user sees a "high level", syntax free view of the code. This is useful because it gives the user a global view of the file. A large amount of source code is displayed and program flow is easily followed.

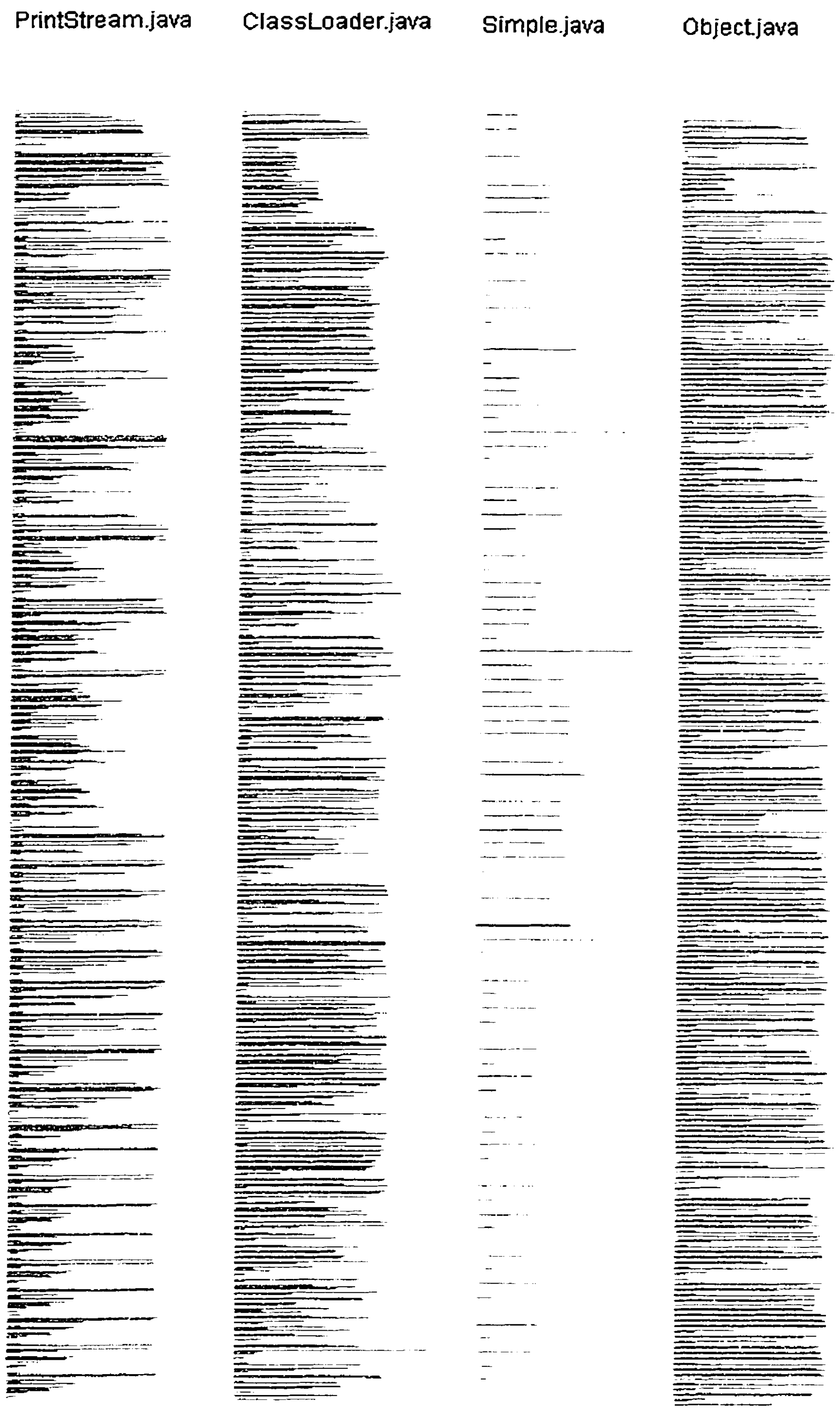
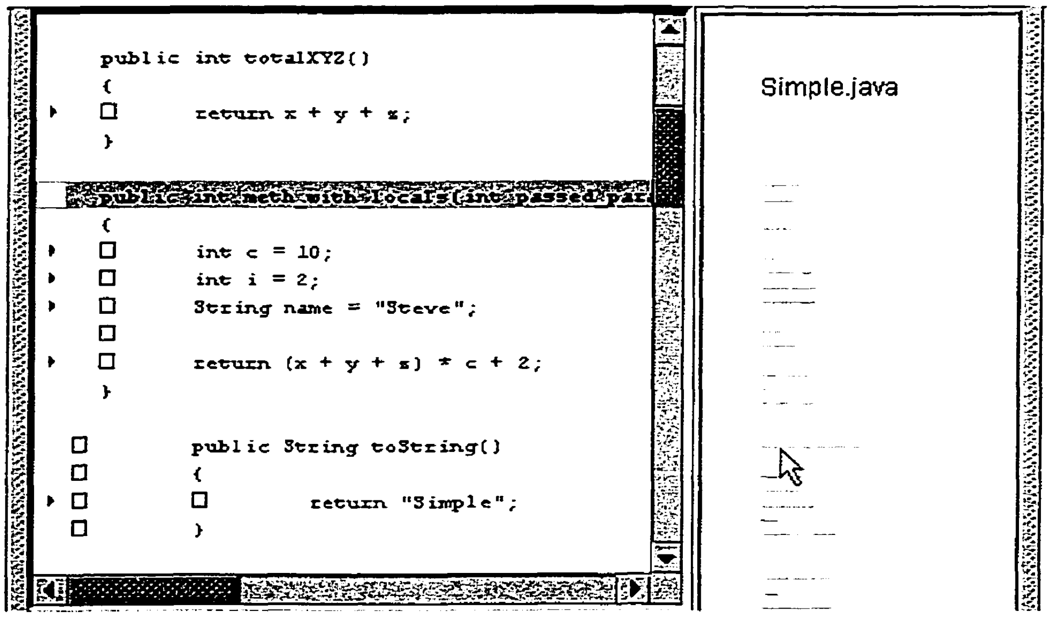


Figure 5.2 Four example files displayed using Seeview

Seeview can also display the actual line of text represented by the thin rows. The user uses the mouse pointer and clicks on a line in the Seeview. Instantly the textview changes showing a highlighted line of text displaying the source code which the thin line represents. See Figure 5.3 for an example.



```
    public int totalXYZ()
    {
▸   □       return x + y + z;
    }

    public int meth_with_locals(int passed_par
    {
▸   □       int c = 10;
▸   □       int i = 2;
▸   □       String name = "Steve";
    □
▸   □       return (x + y + z) * c + 2;
    }

  □         public String toString()
  □         {
▸ □         □       return "Simple";
  □         }
```

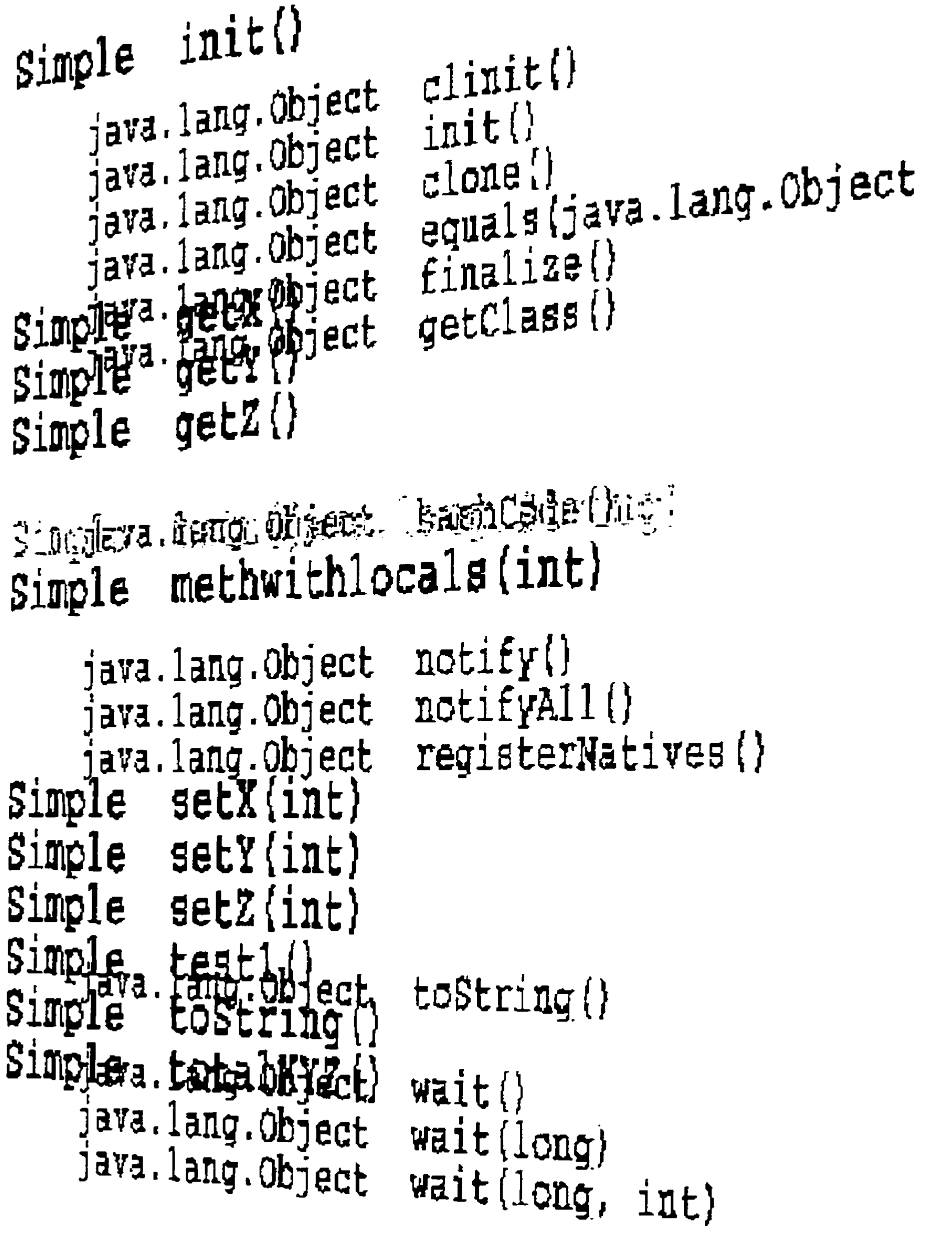
Figure 5.3 Seeview selection of a line

Seeview is particularly useful in helping the user spot patterns within the code. For example, the execution of a loop under Seeview is very noticeable. The repeated pattern of execution of a block of code or the lack of movement of the current line indicator draws the attention of the user.

5.4 CLASSVIEW

SSV has a view known as Classview, which displays the currently executing class and its inherited hierarchy in a 3-dimensional manner. For example, in Figure 5.4, a class called Simple is displayed. Simple is derived from java.lang.object. The methods of simple are displayed in the foreground, while the methods of object are displayed in the background. Figure 5.5 shows an example that has three levels of inheritance. The lowest class is TPrim and is displayed in front. TPrim inherits from DrawPrim, which is displayed in the middle, and finally DrawPrim inherits from java.lang.object, which is displayed furthest from the viewer.

As the visualizer executes, the debuggee program steps through and executes line-by-line. As each line executes, Classview changes to reflect the currently executing class. If the current class changes, for example, the code steps into a string output routine, the Classview display changes to reflect the string class structure. When the string routines complete and execution returns to a different class, the Classview display adjusts automatically and displays the current class.



```
Simple  init()
    java.lang.Object  clinit()
    java.lang.Object  init()
    java.lang.Object  clone()
    java.lang.Object  equals(java.lang.Object
    java.lang.Object  finalize()
Simple  getX()
    java.lang.Object  getClass()
Simple  getY()
Simple  getZ()

[illegible]java.lang.Object  hashCode()
Simple  methwithlocals(int)

    java.lang.Object  notify()
    java.lang.Object  notifyAll()
    java.lang.Object  registerNatives()
Simple  setX(int)
Simple  setY(int)
Simple  setZ(int)
Simple  test1()
    java.lang.Object  toString()
Simple  toString()
Simple  totalXYZ()
    java.lang.Object  wait()
    java.lang.Object  wait(long)
    java.lang.Object  wait(long, int)
```

Figure 5.4 Classview of class Simple

As SSV executes the debuggee program and Classview displays the structure of the currently executing class, Classview also highlights the currently executing method within that class. Highlighting is done setting the current method to a blinking yellow colour and ensuring that the method is always visible. This draws the user's attention to the method. See Figures 5.4 and 5.5.

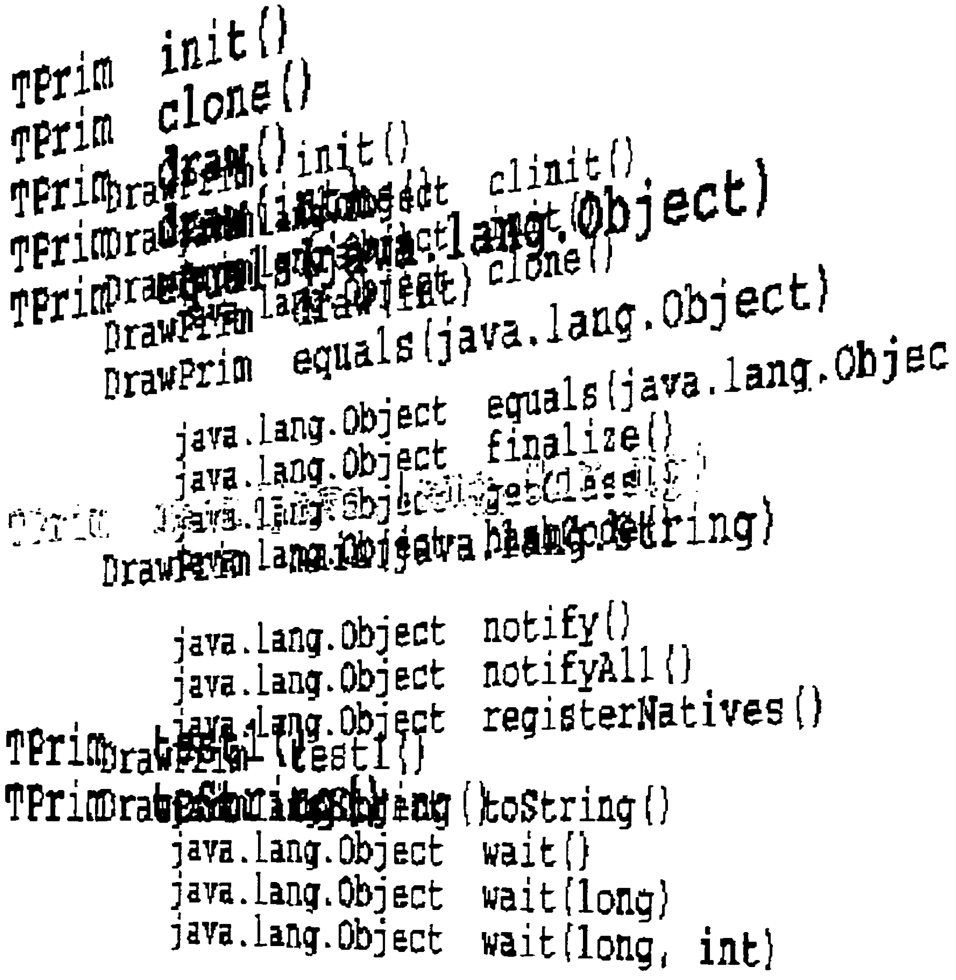
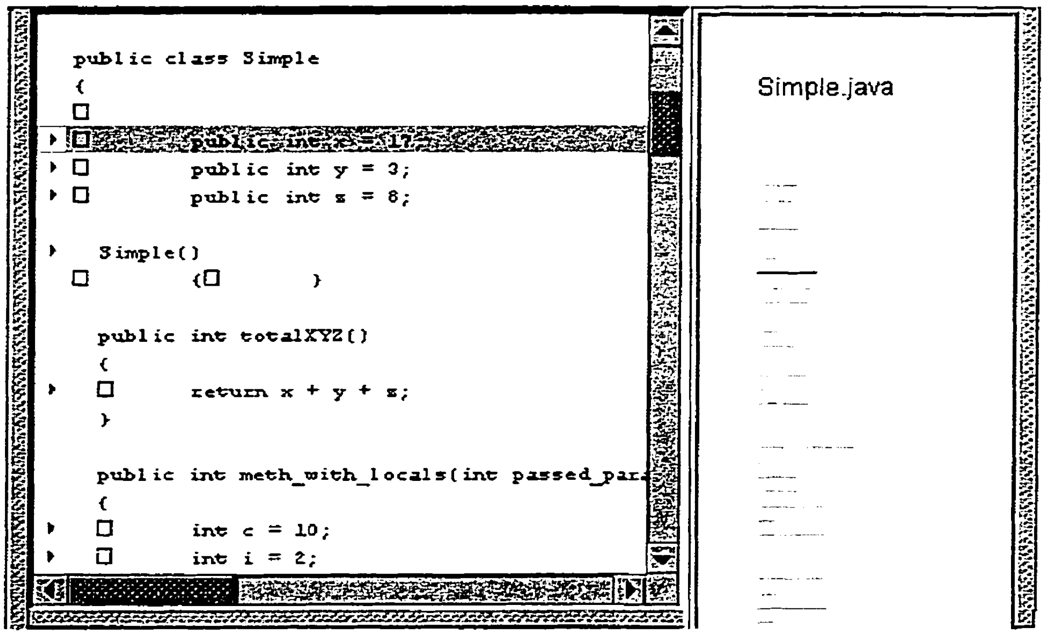

TPrim init()
TPrim clone()
TPrim draw()
DrawPrim init()
clinit()
java.lang.Object
DrawPrim equals(java.lang.Object)
equals(java.lang.Objec
java.lang.Object equals(java.lang.Object)
java.lang.Object finalize()
java.lang.Object getClass()
java.lang.Object hashCode()
java.lang.Object notify()
java.lang.Object notifyAll()
java.lang.Object registerNatives()
TPrim test1()
DrawPrim test1()
TPrim toString()
java.lang.Object toString()
java.lang.Object wait()
java.lang.Object wait(long)
java.lang.Object wait(long, int)

Figure 5.5 Classview of a class with 3 levels of hierarchy

5.5 TEXTVIEW

SSV has a view known as Textview, which displays the actual source code of a loaded class if it is available. The Textview has many uses including highlighting the current line of execution (which changes as the program runs), indicating breakpoints or displaying different files and specific lines. See Figure 5.6.



```
public class Simple
{

        public int x = 17;
        public int y = 3;
        public int z = 8;

  Simple()
        {            }

  public int totalXYZ()
  {
        return x + y + z;
  }

  public int meth_with_locals(int passed_par
  {
        int c = 10;
        int i = 2;
```

Figure 5.6 Textview of an executing piece of code

5.6 ANIMATION

SSV is useful if it helps the user understand the software. To aid in this recognition process, the ability to animate SSV is included; where animation is defined as the automatic and repeated stepping through of source code while the visualization displays are simultaneously updated. The period of time a visualization can be animated for is determined by the user. The user can choose any time period, for example, a thirty second interval.

5.7 ARCHITECTURE

SSV has all the functionality of a debugger, for example, setting break points, a call stack and evaluation of variables. SSV also has the software visualization tools added for this thesis: Seeview, Classview, Textview and animation. Each is interesting and useful by itself however when used in combination SSV becomes a powerful visualization tool. By using the views in combination, certain aspects of the software can be "visualized" that would not be possible through traditional text based debugging methods.

SSV is part software visualization tool and part debugger because the software visualization tools are built on top of a Java debugger. See figure 5.7 for an illustration of the architecture.

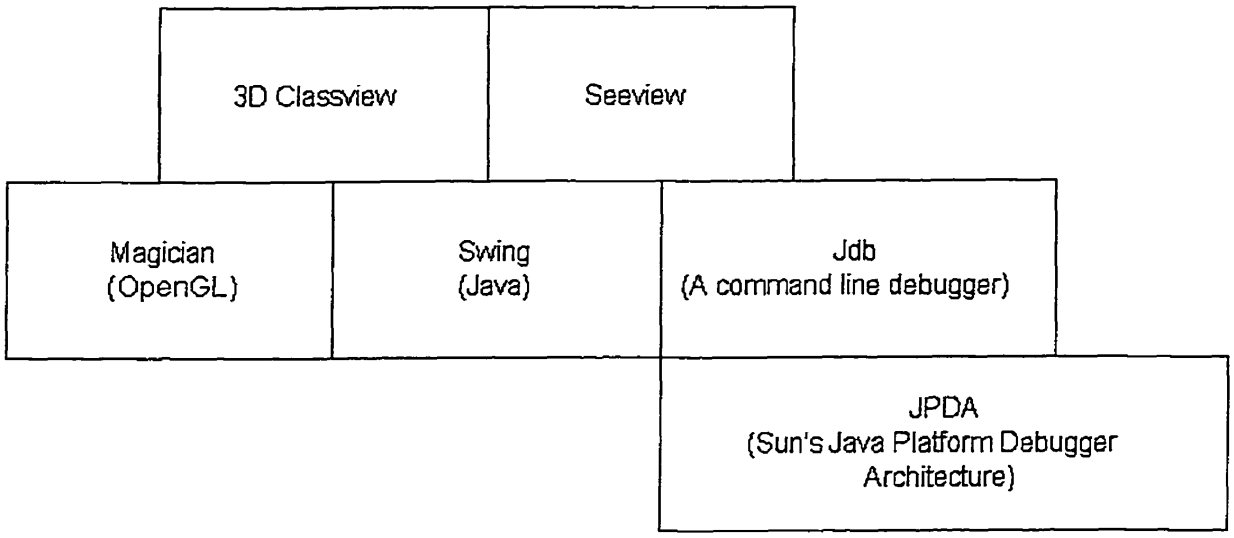

3D Classview
Seeview
Magician (OpenGL)
Swing (Java)
Jdb (A command line debugger)
JPDA (Sun's Java Platform Debugger Architecture)

Figure 5.7 Basic architecture of SSV

6. USER TESTING

A pilot study involving six Acadia University computer science students was conducted to test the benefits of the SSV tool. Subjects were given two tasks to perform with SSV. The first task was relatively simple. The subjects were asked to determine the hierarchy (both superclass and descendants) of a class called "Shape". There are three possible avenues to find the answer: source code explorations using the text view, a command line function, or program stepping. The second task was more challenging. Subjects were asked to trace through the execution of a function showing the value of two local variables during each iteration of a loop and the final return value. The function is called "totalArea" which is passed a list of five Shape objects which are a combination of Shape derived classes called Circle, Triangle and Rectangle. The two local variables are "total" and "area" which accumulate the area of the Shapes and report the current Shape's area respectively. The subjects were then given a survey to complete. See Appendix A for details of the tasks and for the questionnaire. Appendix B contains the individual responses.

6.1 PURPOSE OF THE STUDY

The purpose of the study was to test if SSV gives users insight to Java source code and the execution of the compiled byte code. That is, does SSV deliver on the ideals of its design? For example, does the global view of the file (Seeview) help the user follow program flow? Or does the three-dimensional class view help the user understand a new class? Also, the survey was used to gather general feedback and comments about the system.

6.2 RESULTS

As a whole the feedback from the questionnaire was positive. In addition there were some valuable and valid criticisms and suggestions. Overall the system was found to be useful (Appendix B, 6c), and respondents generally like using the system (Appendix B, 6a).

The survey has four points of evaluation: 1) learning to use the system, 2) the screen, 3) using the system, and 4) an overall assessment. On average respondents were indifferent about learning to use the system (Appendix B, 3). However, individual responses concerning learning to use the system were either difficult or easy. Responses concerning the screen layout, color, and arrangement of information were positive (Appendix B, 4). However several users complained that the source code window font was too small, or suggested that the window be larger. Responses concerning using the system (Appendix B, 5), such as the number of steps per tasks, or remembering how to use the system are positive.

The overall assessment had mixed scores (Appendix B, 6). Overall users liked using SSV, but gave it low marks for ease of use. However, the question "Overall I found SSV useful" received the highest positive average response of all survey questions. Despite this, the overall helpfulness of the line representation of the Java file was rated low, but not as low as the 3D representation. Yet the individual responses were either very high or very low.

Comments concerning the system were mixed. Some respondents liked the line representation but others did not. Some respondents found the system easy to use, while one subject wrote, "it wasn't the most intuitive". However, with one exception, most found that the three-dimensional class hierarchy (Classview) contains too much information. One subject responded, "the new 3D hierarchy is a good idea, it just needs a bit of work". While another wrote, "the 3D class representation is cool, but…it is too busy". The comments included, "the 3D class view is bewildering and intimidating" and "the 3D class hierarchy was frustrating".

General comments about the SSV system include improvements to moving, resizing, or toggling of certain windows. One respondent suggested that keyboard short cuts would make the system easier to use. Also, several respondents suggested the system should have source code search capabilities. Users wanted to be able to search a single file, all files, or locate a certain class.

6.3 SURVEY CONCLUSIONS

The pilot study was the first public showing of SSV. Based on user testing, more work is needed on screen layouts and the three-dimensional class representation before SSV is ready for general use.

Blackwell [Blackwell, 1996a] presents an assessment of visualization tools that focuses on the way computer scientists think that visual programming assists the thought processes of the programmer, with a list of twelve categories of possible benefit.

One of Blackwell's categories is abstraction, where, despite agreement that abstraction is an important issue, the author remarks that some computer scientists believe that pictures are good at showing abstraction, while others say that abstract data is challenging. This difference of opinion is shown by our own user testing with clear separation of abstract usefulness (Appendix B, 6d and 6e).

Cognitive resources is another of Blackwell's categories. The author writes, "computer scientists claim that the human mind is optimized for vision, making shapes easier to process than words." However, Petre [Petre, 1993] presents evidence that learning to read graphics and "seeing' an information display is an acquired skill. SSV is targeted towards those individual who like abstract representations of their code and after a little training and practice, SSV could be a useful visualization tool.

7. FUTURE WORK

Animated visual displays let the user assimilate information rapidly and help identify trends and anomalies. However academia seems to be further ahead than the real world in software visualization. In [Price, 1993], the author suggests that "software engineers (and their employers) have not seen demonstrable gains from using this technology. It is clear that if SV systems are to make a contribution to software engineering then solid results proving their benefits will be necessary." Hopefully in the future more companies will find results that lead to integration of more software visualization tools into their systems.

7.1 PARALLEL DEBUGGING

Future work on SSV should include true support for parallel processes. In the current form SSV visualizes a parallel program but the concurrency is not immediately apparent. Visualization of all threads and the state of those threads would be interesting and informative, as well as any message passing that occurs.

7.2 PROFILING

Future work on SSV should include profiling such as that seen in [De Pauw 1998]. Histograms displaying method time lengths, method usage or histograms of instances would be interesting and useful debugging information, as well as giving insight into program execution.

7.3 IMPROVED CLASS VIEW

Future work on SSV should involve the class view. The NV3D system can display multiple classes on the screen at the same time. This multiple display allows the user to explore and investigate with less screen switching and therefore a smoother, more pleasant experience. Therefore adding this ability to SSV would give the user easier access to information. Multiple classes on the same screen lends itself nicely to the proposed future addition of parallel program support.

Based on user testing (Chapter 6), some respondents felt too much information is currently displayed by the 3D class view. A possible solution to this problem could be allowing the user to select the level of detail displayed.

The user could be given the choice of displaying only the current method, or only the current method with any possible superclass methods the current method is derived from, or limit the display to a certain number of superclasses or methods as specified by the user.

7.4 SEARCHING

Search capability for SSV was suggested during user testing. Allowing searching of source code gives the user the opportunity to easily satisfy a need for information. Searching provides information that the user desires, yet only when he or she requests it, therefore the user is not overwhelmed with too much information. Consequently searching would fit nicely into the SSV system.

8. CONCLUSION

The building of software has become complex. Software visualization tools have the potential to make the meaning of the code more apparent and concrete, while making the overall structure of the program easier to grasp.

This thesis has created a visualization tool for Java. This tool helps show the structure of the Java code with minimal work required by the user. Views are automatic and effortless allowing the user to concentrate on the problem and not on creating debugging information. User testing shows that users like using the system and, overall, find SSV useful.

The visualization tool has been built on top of a debugger to promote iterative design and exploration. The tool is intended to be an add-on to a user's existing arsenal of programming tools.

APPENDIX A

Task Questions

1. List the class hierarchy for class Shape and its superclass and its descendants.

2. Trace the execution of totalArea. Show the values of the local variables "total" and "area" during each iteration of the for loop. Show the return value of totalArea.

Questionnaire

Identification

Identification Number ________________

Gender
Female ______ Male ________

Age _______

Background

University Degree of Study ______________

Current University Year of Study
1 _____ 2 _____ 3 _____ 4 _____ 5 _____

University Level Computer Half Courses (completed or taking)
None _______
1 or 2 ________
3 or more _______

Used on-line debugger software
Yes _____
No _____

3. Learning to use the system

Learning to use the system was
Difficult 1 2 3 4 5 6 7 8 9 Easy

Getting started was
Difficult 1 2 3 4 5 6 7 8 9 Easy

Exploration of features by trial and error was
Discouraging 1 2 3 4 5 6 7 8 9 Encouraging

4. The Screen

Were the characters on the computer screen
Hard to read 1 2 3 4 5 6 7 8 9 Easy to read

Was the use of colour helpful
Not at all 1 2 3 4 5 6 7 8 9 Very much

Were the screen layouts helpful
Not at all 1 2 3 4 5 6 7 8 9 Very much

Was the amount of information displayed
Inadequate 1 2 3 4 5 6 7 8 9 Adequate

Was the arrangement of information
Illogical 1 2 3 4 5 6 7 8 9 Logical

5. Using the System

Remembering how to use the system was
Difficult 1 2 3 4 5 6 7 8 9 Easy

Could you do the task in a straight forward manner
Never 1 2 3 4 5 6 7 8 9 Always

Number of steps per task was
Too many 1 2 3 4 5 6 7 8 9 Just right

The steps required to complete a task follow a logical sequence
Not at all 1 2 3 4 5 6 7 8 9 Very much

6. Overall Assessment

Overall I liked using Steve's Software Visualizer
Not at all 1 2 3 4 5 6 7 8 9 A lot

Overall I found it easy to use
Hard to use 1 2 3 4 5 6 7 8 9 Easy to use

Overall I found it useful
Not at all 1 2 3 4 5 6 7 8 9 A lot

Was the line representation of the Java file helpful
Not at all 1 2 3 4 5 6 7 8 9 Very much

Was the 3D class representation helpful
Not at all 1 2 3 4 5 6 7 8 9 Very much

Comments

How easy was this system to use?

Would you prefer a standard debugger? Yes__________ No__________
Why?

Which features did you like?

Which features did you not like?

What features do you wish the system had?

Other comments?

APPENDIX B

Points of Evaluation	**Individual Subject Score***						
Subject Number:	**1**	**2**	**3**	**4**	**5**	**6**	**Avg**
3. LEARNING TO USE THE SYSTEM							
3a. Learning to use the system was *Difficult → Easy*	2	2	1	3	3	1	**2**
3b. Getting started was *Difficult → Easy*	2	1	1	3	3	1	**1.8**
3c. Exploration of features by trial and error was *Discouraging → Encouraging*	3	1	1	3	3	1	**2**
4. THE SCREEN							
4a. Were the characters on the computer screen *Hard to read → Easy to read*	2	3	1	2	2	1	**1.8**
4b. Was the use of colour helpful *Not at all → Very much*	3	3	1	3	3	3	**2.6**
4c. Were the screen layouts helpful *Not at all → Very much*	2	2	1	3	2	2	**2.2**
4d. Was the amount of information displayed *Inadequate → Adequate*	3	3	1	3	2	2	**2.5**
4e. Was the arrangement of information *Illogical → Logical*	3	2	2	3	3	3	**2.7**
5. USING THE SYSTEM							
5a. Remembering how to use the system was *Difficult → Easy*	3	2	3	3	2	2	**2.5**
5b. Could you do the task in a straight forward manner *Never → Always*	3	2	1	3	2	2	**2.2**
5c. Number of steps per task was *Too many → Just right*	3	2	2	3	3	1	**2.3**
5d. The steps required to complete a task follow a logical sequence *Not at all → Very much*	3	3	1	1	3	3	**2.3**
6. OVERALL ASSESSMENT							
6a. Overall I liked using Steve's Software Visualizer *Not at all → A lot*	3	2	2	3	3	1	**2.3**
6b. Overall I found it easy to use *Hard to use → Easy to use*	3	2	1	2	2	1	**1.8**
6c. Overall I found it useful *Not at all → A lot*	3	3	2	3	3	3	**2.8**
6d. Was the line representation of the Java file helpful *Not at all → Very much*	2	1	1	3	3	1	**1.8**
6e. Was the 3D class representation helpful *Not at all → Very much*	2	1	1	2	3	1	**1.6**

** Data has been normalized from a subject score between 1 and 9 to a score between 1 and 3, where subject scores of 1, 2 and 3 have been folded into a normalized score of 1, scores of 4, 5 and 6 into 2, and scores of 7, 8 and into 3.*

BIBLIOGRAPHY

[Baecker 1981] Baecker R., "Sorting Out Sorting," (film), Dynamic Graphics Project, University of Toronto, Toronto, 1981.

[Ball 1996] Ball T. and Eick S., "Software visualization in the large," *Computer*, Vol.29, No.4, 1996, pp. 33-43.

[Blackwell 1996a] Blackwell A.F., "Metacognitive theories of visual programming: What do we think we are doing?", IEEE Symposium on Visual Languages, 3-6 September 1996, pp. 240-246.

[Brown 1984] Brown M.H. and Sedgewick R., "A System for Algorithm Animation," *Computer Graphics*, July 1984, pp. 177-186.

[Brown 1988] Brown M.H., "Exploring algorithms using Balsa-II," *Computer*, Vol.21, No.5, 1988, pp. 14-36.

[Brown 1991] Brown M.H., "Zeus: a system for algorithm animation and multi-view editing," *Proceedings of the IEEE Workshop on Visual Languages*. Kobe, Japan, October 1991, pp. 4-9.

[Brown 1996] Brown M. and Najork M., "Collaborative Active Textbooks: a Web-Based Algorithm Animation System for an Electronic Classroom", *Proceedings of the 1996 IEEE International Symposium on Visual Languages*, Boulder, CO, September 1996, pp. 266-275.

[Bykat 1996] Bykat A., "Visualizing program concepts using EVA", Energy Conversion Engineering Conference, 1996. IECEC 96., Proceedings of the 31st Intersociety, Vol. 1, 1996, pp. 271-276.

[Cox 1989] Cox P., Giles F., and Pietrzykowski T., "Prograph: a step towards liberating programming from textual conditioning", *IEEE Workshop on Visual Languages*, 1989, pp. 150-156.

[De Pauw 1998]De Pauw W., Kimelman D., and Vlissides J., "Visualizing Object-Oriented Software Execution," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 329-346.

[Eick 1992] Eick, S., Steffen, J., and Sumner, E. (Jr), "Seesoft—A Tool For Visualizing Line Oriented Software Statistics", *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, November 1992, pp. 957-968.

[Francioni 1991] Francioni J., "Debugging parallel programs using sound," *SIGPLAN Notices*, Vol. 26, No. 12, December 1991, pp. 68-75.

[Furnas 1986] Furnas G., "Generalized Fisheye Views," *Proceeding of CHI'86*, Boston, April 1986, pp. 16-23.

[Gaver 1989] Gaver W., "The Sonic Finder" An Interface That uses Auditory Icons," *Human-Computer Interaction*, Vol. 4, No. 1, Spring 1989, pp. 67-94.

[Kaoike 1997] Kaoike H., Takada T. and Masui, T., "VisuaLinda: A Framework for Visualizing Parallel Linda Programs," *IEEE*, 1997, pp. 174 - 178.

[Lahtinen 1998] Lahtinen S., Sutinen E., and Tarhio J., "Automated Animation of Algorithms with Eliot," *Journal of Visual Languages and Computing*, Vol. 9, No. 3, 1998, pp. 337-349.

[Lange 1997] Lange D. and Nakamura Y., "Object-Oriented Program tracing and Visualization," *Computer*, May 1997, pp. 63-70.

[Lehr 1989] Lehr T., Segall Z., Vrsalovic D.F., Caplan E., Chung A.L., and Fineman C.E., "Visualizing performance debugging," *Computer*, Vol. 22, No. 10, 1989, pp. 38-51.

[Lieberman 1998] Lieberman H. and Fry C., "ZStep 95: A Reversible, Animated Source Code Stepper," *Software Visualization - Programming as a Multimedia Experience*, MIT Press, 1998, pp. 277-292.

[NVision 1999] NVision Software Systems Inc. "NV3D for Visual Studio - User's Guide (beta)," 1999.

[Merlini 1999] Merlini D., "A System for Algorithms' Animation," *IEEE*, 1999, pp. 1033-1034.

[Muchaluat 1998] Muchaluat D., Rodrigues R. and Soares L., "WWW Fisheye-View Graphical Browser," *IEEE*, 1998, pp. 80 - 89.

[Mukherjea 1994] Mukherjea S. and Stasko J., "Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger," *ACM Transactions on Computer-Human Interaction*, Vol. 1, No. 3, September 1994, pp. 215-244.

[Myers 1988] Myers B.A., Chandhok R. and Sareen A., "Automatic data visualization for novice Pascal programmers," *Proceedings of the IEEE Workshop on visual Languages*, Pittsburgh, Pennsylvania, 1988, pp.192-198.

[Parker 1998] Parker G., Franck G., and Ware C., "Visualization of Large Nested Graphics in 3D: Navigation and Interaction," *Journal of Visual Languages and Computing*, Vol. 9, No. 3, 1998, pp. 299-317.

[Petre, 1993] Petre M. and Green T.R.G., "Learning to read graphics: some evidence that 'seeing' an information display is an acquired skill", Journal of Visual Languages and Computing, Vol. 4, No. 1, 1993, pp. 55-70.

[Price 1993] Price B., Baecker R. and Small I., "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, Vol. 4, No. 3, 1993, pp. 211-266.

[Sarkar 1994] Sarkar M. and Brown, M., "Graphical Fisheye Views," *Communications of the ACM*, Vol. 37, No 12, Dec 1994, pp. 73-84.

[Shimomura 1990] Shimomura T. and Isoda S., "VIPS: a visual debugger for list structures," *Computer Software and Applications Conference, 1990*. COMPSAC 90, Proceedings, Fourteenth Annual International, 1990, pp. 530-537.

[Stasko 1989] Stasko J.T., "TANGO: a framework and system for algorithm animation," *IEEE Computer*, Vol. 23, No. 9, 1989, 27-39.

[Stasko 1990] Stasko J.T., "Simplifying Algorithm Animation with TANGO," IEEE, 1990, pp. 1-6.

[Stasko 1993] Stasko J.T. and Kraemer E., "A Methodology for Building Application-Specific Visualizations of Parallel Programs," *Journal of Parallel and Distributed Computing*, Vol. 18, 1993, pp. 258-264.

[Warendorf 1997] Warendorf K., Wen Jing Hsu, and Poh Yeen Seah, "ARMVLS-atomic reaction model visual language system-a new way of animating algorithms," *Proceedings of 1997 International Conference on Information, Communications and Signal Processing*, vol.2, 1997, pp. 939-943.

FURTHER READING NOT DIRECTLY REFERENCED

[Averbukh 1997] Averbukh V., "Toward Formal Definition of Conception "Adequacy in Visualization"," *IEEE Proceedings of VLL'97, September 23-26, 1997 in Capri, Italy,* 1997, pp. 46-47.

[Baecker 1998a] Baecker R., "Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 369-382.

[Baecker 1998b] Baecker R., and Price B., "The Early History of Software Visualization," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 29-34.

[Baecker 1998c] Baecker R., and Marcus A., "Printing and Publishing C Programs," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 45-62.

[Bazik 1998] Bazik J., Tamassia R., Reiss S. P. and van Dam A., "Software Visualization in Teaching at Brown University," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 383-398.

[Berner 1998] Berner S., Joos S., and Glinz M., "A Visualization Concept for hierarchical Object Models," *IEEE*, 1998, pp. 225-228.

[Blackwell 1996b] Blackwell A.F. and Green T.R.G, "Does metaphor increase visual languages usability?", IEEE Symposium on Visual Languages, 13-16 September 1999, pp. 246-253.

[Brown 1998a] Brown M. H., "A Taxonomy of Algorithm Animation Displays," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 35-42.

[Brown 1998b] Brown M. H., and Hershberger J., "Fundamental Techniques for Algorithm Animation Displays," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 81-102.

[Brown 1998c] Brown M. H., and Hershberger J., "Program Auralization," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 137-144.

[Brown 1998d] Brown M. H., and Najork M. A., "Algorithm Animation Using Interactive 3D Graphics," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 119-136.

[Brown 1998e] Brown M. H., and Sedgewick R., "Interesting Events," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 155-172.

[Burkwald 1998] Burkwald S., Eick S., Rivard K., and Pyrce J., "Visualizing Year 2000 Program Changes," *IEEE*, 1998, pp. 13-18.

[Bykat 1996] Bykat A., "Visualizing program concepts using EVA," *Proceedings of the 31st Intersociety Energy Conversion Engineering Conference*, Vol. 1, 1996, pp. 271-276.

[Chuah 1997] Chuah M., and Eick S., "Glyphs for Software Visualization," *IEEE*, 1997, pp. 183-191.

[Domingue 1998] Domingue J., "Visualizing Knowledge Based Systems," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 223-236.

[Eick 1998] Eick S. G., "Maintenance of Large Systems" *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 315-328.

[Eisenstadt 1998] Eisenstadt M., and Brayshaw M., "The Truth about Prolog Execution," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 207-222.

[Franck 1994] Franck G., and Ware C., "Representing Nodes and Arcs in 3D Networks", *IEEE Conference on Visual Languages Conference Proceedings*, 1994, pp. 189-190.

[Gaver 1991] Gaver W. W., Smith R. B., and O'Shea T., "Effective sounds in complex systems: the ARKOLA simulation," *Human Factors In Computing Systems Conference Proceedings On Reaching Through Technology*, 1991, pp. 85-90.

[Gloor 1998a] Gloor P. A., "Animated Algorithms," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 409-416.

[Gloor 1998b] Gloor P. A., "User Interface Issues for Algorithm Animation," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 145-152.

[Green 1996] Green T.R.G. and Blackwell A.F., "Thinking about visual programs", IEE Colloquium on Thinking with Diagrams (Digest No: 1996/010), 1996, pp. 5/1-5/4.

[Heath 1998] Heath M. T., Malony A. D., and Rover D. T., "Visualization for Parallel Performance Evaluation and Optimization," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 347-366.

[Jackson 1991] Jackson J.A., and Francioni J.M., "Aural Signatures of Parallel Programs," *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992*, Vol. 2, 1992, pp. 218-229.

[Javasoft] Javasoft Inc, Home Page, http://java.sun.com

[Jayaraman 1996] Jayaraman B., and Baltus C., "Visualizing Program Execution," *IEEE*, 1996, pp. 30-37.

[Jeffrey 1998] Jeffrey C. L., "Visualizing Graph Models of Software," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 63-72.

[Kimelman 1998] Kimelman D., Rosenburg B.,and Roth T., "Visualization of Dynamics in Real World Software Systems" *Software Visualization – Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 293-314.

[Kraemer 1993] Kraemer E., "The Visualization of Parallel Systems: An Overview," Journal of Parallel and Distributed Computing, Vol. 18, 1993, pp. 105-117.

[Kraemer 1998] Kraemer E., "Visualizing Concurrent Programs" *Software Visualization – Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 237-256.

[Mulholland 1998a] Mulholland P., "A Principled Approach to the Evaluation of SV: A Case Study in Prolog," 1998, pp. 439-452.

[Mulholland 1998b] Mulholland P. and Eisenstadt M., "Using Software to Teach Computer Programming: Past, Present and Future," *Software Visualization – Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 399-408.

[Noik 1993] Noik E.G., "Exploring large hyperdocuments: fisheye views of nested networks", Conference on Hypertext and Hypermedia, Proceedings of the fifth ACM conference on Hypertext, Seattle, WA, USA, November 14-18, 1993.

[North 1998] North S., "Visualizing Graph Models of Software," *Software Visualization – Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 63-72.

[Osawa 1996] Osawa N., Hisano K., and Yuba T., "A Visual Performance Debugging System for Parallel Programs," *IEEE Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, 1996, pp. 300-308.

[Petre 1998] Petre M., Blackwell A., and Green T., "Cognitive Questions in Software Visualization," 1998, pp. 453-480.

[Preece 1994] Preece J., Rogers Y., Sharp H., Benyon D., Holland S. and Carey T., *Human-Computer Interaction*, Addison-Wesley Publishing Company, 1994.

[Price 1998] Price B., Baecker R., and Small I., "An Introduction to Software Visualization," *Software Visualization - Programming as a Multimedia Experience* Cambridge, Massachusetts, MIT Press, 1998, pp. 3-28.

[Reiss 1997] Reiss S.P., "Cacti: a front end for program visualization," *Proceedings of the IEEE Symposium on Information Visualization*, 1997, pp. 46-49, 120.

[Reiss 1998] Reiss S. P., "Visualization for Software Engineering -- Programming Environments," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 259-276.

[Roman 1993] Roman G. C., and Cox K., "A Taxonomy of Program Visualization Systems," *Computer*, Vol. 26, No. 12, December 1993, pp.11-24

[Roman 1998] Roman G., "Declarative Visualization," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 29-34.

[Seemann 1998] Seemann J., and Gudenberg J., "Visualization of Differences between Versions of Object-Oriented Software," *IEEE*, 1998, pp. 201-204.

[Stasko 1996a] Stasko J., "Smooth Continuous Animation for Portraying Algorithms and Processes," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 103-118.

[Stasko 1996b] Stasko J., and Muthukumarasamy J., "Visualizing Program Executions on Large Data Sets," Visual Languages, 1996. Proceedings., IEEE Symposium on , 1996, pp. 166-173.

[Stasko 1998] Stasko J., "Building Software Visualizations through Direct Manipulation and Demonstration," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 187-204.

[Stasko 1998] Stasko J., and Lawrence A., "Empirically Assessing Algorithm Animations as Learning Aids," *Software Visualization - Programming as a Multimedia Experience*, Cambridge, Massachusetts, MIT Press, 1998, pp. 419-438.

[Storey 1997] Storey M. A. D., Wong K., and Muller H.A., "Rigi: A Visualization Environment for Reverse Engineering," *Proceedings of the 1997 International Conference on Software Engineering*, 1997, pp. 606 -607.

[Ware 1994a] Ware C., and Franck G., "Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions", *ACM Transactions on Graphics*.

[Ware 1994b] Ware C., and Franck G., "Viewing a Graph in a Virtual Reality Display is Three Times as Good as a 2D Diagram", *IEEE Conference on Visual Languages Conference Proceedings*, October 1994, pp. 189-190.