Software Visualization in a Multiscale Environment

by

Fred Basas

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Master of Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

© Fred Basas, MCMXCIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

May 12, 1994

Certified by

Henry Lieberman
Research Scientist, MIT Media Lab
Thesis Supervisor

Ben Bederson
Member Technical Staff, Bell Communications Research
Thesis Supervisor

Accepted by

Fredric R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

JUL 18 1994
Software Visualization in a Multiscale Environment

by

Fred Basas

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1994, in partial fulfillment of the
requirements for the degrees of
Master of Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

Software systems are becoming increasingly complex. While good software engineering practices help deal with the understanding of these systems, often they do not keep up with the rising complexity of these systems. Furthermore, software in the middle of development often does not conform to good software engineering practices, which further adds to the complexity. The use of techniques that graphically represent software in order to aid in its understanding is called software visualization.

This thesis describes the use of different software visualization techniques in a novel interface: the multiscale environment. A multiscale environment is an environment where information can be entered at any location and at any scale. We will look at three ideas in visualizing large software systems: source code cross-referencing, frame-based visualization, and animation of Lisp code. With each of these ideas, we seek to exploit the advantages that multiscale environments provide over more traditional interfaces.

In this thesis, we will give some background on more traditional and effective software visualization techniques. Background will also be given on multiscale environments, and their motivation. Then we will explore in detail the three main ideas mentioned above, and will describe their implementation in a particular multiscale environment. Finally, we will describe any insights that have been gained on software visualization while working on this project.

Thesis Supervisor: Henry Lieberman
Title: Research Scientist, MIT Media Lab

Thesis Supervisor: Ben Bederson
Title: Member Technical Staff, Bell Communications Research
Acknowledgments

This thesis would not be possible without the help of many people. It would be impossible to name them all, but I wish to thank those who have been especially helpful.

First and foremost, thanks to Ben Bederson and Henry Lieberman, who were both kind enough to advise this thesis. I thank Ben for his patience in helping me learn Pa3d. Equally important were his honest and critical opinions of the ideas presented in this thesis. I thank Henry for his persistence in encouraging me to look for new and creative ideas. His willingness to advise an off-campus thesis was critical to this project, and for that I am very grateful.

Many thanks should go to the Computer Graphics and Interactive Media Research Group at Bellcore, especially to Jim Hollan, who allowed me to pursue a topic of my choice, and to Larry Stead, whose creation of the Ar3t system allowed me to do things I never thought possible. I also would like to thank George Furnas, Will Hill, Todd Outten, Mark Rosenstein, Louis Weitzman, and Kent Wittenberg for supporting this research and providing some needed feedback. These guys taught me that research can be fun, and for that I am forever grateful.

In my five years at MIT, I’ve met many hundreds of people. These people were my classmates, roommates, officemates, teammates, and soulmates. They taught me that the real genius of MIT lies not in the institution, but in the people inside the institution. Many thanks to those whose paths crossed mine.

Finally, thanks to my family. My parents and two sisters were the constants in my five years at MIT, and have held me steady through the toughest times of my life. Nothing I’ve accomplished here would have happened without their help.
Contents

1 Introduction ......................................................... 9

2 Using the Environment .............................................. 11
   2.1 Source Code Cross-Referencing .............................. 11
   2.2 Frame Based Visualization .................................... 13
   2.3 Lisp Code Animation ........................................... 17

3 Software Visualization ............................................. 23
   3.1 Program Visualization ......................................... 23
   3.2 Visual Programming ............................................ 24
   3.3 Programming by Example ....................................... 24
   3.4 Algorithm Visualization ....................................... 25
   3.5 Code Visualization ............................................. 26
   3.6 General Comments ............................................... 27

4 Multiscale Environments ........................................... 30
   4.1 SDMS ................................................................ 31
   4.2 Pad .................................................................. 32
   4.3 Pa3d .................................................................. 33
      4.3.1 Background and Motivation ............................... 33
      4.3.2 Implementation .............................................. 34
      4.3.3 The Pa3d World .............................................. 34
      4.3.4 Features ....................................................... 35
5 Exploring Source Code in Pa3d

5.1 Why View Source Code in Pa3d? ........................................... 39

5.2 Cross Referencing of Source Code ........................................ 41
  5.2.1 Cross Referencing Through Navigation ............................... 41
  5.2.2 Cross Referencing Through Graphics ............................... 42

6 A Frame System for Visualizing Code .................................... 44

6.1 Frame System Description .................................................. 44

6.2 Motivation for Using Frames .............................................. 45

6.3 Frame Descriptions ......................................................... 46

6.4 Parsing Lisp Code to Generate Frames ................................ 49

7 Lisp Program Animation ...................................................... 54

7.1 Using the Frame System to Animate Code .............................. 54

7.2 Dealing with Lisp Animation ............................................. 58

8 Conclusions ............................................................................. 61

8.1 Results ................................................................................. 61

8.2 Unresolved Issues .............................................................. 62

8.3 The Future ............................................................................ 63
List of Figures

2-1 Viewing source code in Pa3d ........................................ 12
2-2 Level of detail changes with scale ................................. 13
2-3 Navigating from a function call to a function definition .......... 14
2-4 Highlighting function calls ........................................ 15
2-5 Graphical links from definition to calls ............................ 15
2-6 Frame Based Visualization ......................................... 16
2-7 Viewing frames within frames ....................................... 17
2-8 Function call frame corresponding to (sum-of-squares 3 4) .... 18
2-9 Evaluating the arguments .......................................... 19
2-10 The function definition frame .................................... 20
2-11 Evaluating the local variables ................................... 20
2-12 Applying the substitution model again .......................... 21
2-13 The final picture ............................................... 21

3-1 Rehearsal World .................................................. 25
3-2 Sorting out Sorting ............................................ 26
3-3 BALSA ...................................................... 27
3-4 SEE ...................................................... 28
3-5 SeeSoft .................................................... 29

4-1 SDMS Media Room ............................................. 32

6-1 Frame structure ................................................ 51
6-2 Frame structure (cont.) ....................................... 52
List of Tables

6.1 Frame description summary ........................................ 49
6.2 Frame slot description summary ................................. 50
Chapter 1

Introduction

Programming is a skill that is not easily acquired, and worse yet, requires skills that many people do not have [18]. With the proliferation of computers, many people own or use a computer, but do not know how to program. Such a gap makes packaged software popular. Users simply plug in the software and go. However, as people become more proficient in computers, the need for a more customized system grows. While it is true that many software packages offer customizable configurations, this is not enough to satisfy many users. In fact, the level of customization needed starts to demand some type of programming. The popularity of spreadsheets, with its built in programming capabilities, illustrates this point. However, the advantages of programs like spreadsheets can not be realized without narrowing the gap between user and programmer. It becomes necessary to explore ideas that would narrow this gap. Often, these ideas deal with graphics and interesting computer-human interfaces. These ideas fall into the realm of software visualization.

Modern computer graphics capabilities has made it possible for us to explore more and more ideas in software visualization. No longer do we need to rely on simple visualizations such as pretty printing source code. Through computer graphics, we can visualize thousands of lines of source code, thus gaining a more complete perspective of the entire software system. The power of modern graphics workstations also makes real-time animation of code possible, thus exposing some fundamental aspects of an algorithm that might not be seen by just viewing source code text.
There are many potential applications of this type of graphics visualization. One is the application towards large software systems, ones containing many thousands, or even millions of lines of code. As the size of the system increases, the complexity rises more than proportionally, even when employing sound software engineering practices. Software visualization becomes a desirable aid in managing large projects. Another potential application of software visualization lies in computer science education. The use of computer animation has shown to have positive results in teaching algorithms to students [6].

Another use for graphics is the exploration of novel interfaces. In this thesis we explore a novel interface called the multiscale environment. A multiscale environment is an environment in which information can be placed at any scale and location. Multiscale environments have effectively infinite extent and resolution, and allows users to organize information spatially.

In this thesis we will describe software visualization ideas as applied to the multiscale environment. This exploration is intended to explore issues not covered by current software visualization systems. These issues are:

- Visualization of large software systems
- The ability to visualize arbitrary source code in its original form

The unique characteristics of the multiscale environment provide us with the ability to explore these issues in some detail.

This thesis is organized into eight chapters. The next chapter will demonstrate the ideas that have been developed in this research. You will see how these ideas are used, and how they can be useful. Chapter 3 will give background on software visualization, which will include examples of software visualization systems. Chapter 4 will give background on multiscale environments. Particular attention will be paid to Pa3d, the environment in which these visualizations were developed. Chapters 5-7 will describe in more detail the ideas presented in Chapter 2. In Chapter 8, we

---

1 The SeeSoft system by Eick is a recent example (and one of the first) that visualizes systems with large amounts of source code.
will give some tentative results, and some issues involved in further research of this project.
Chapter 2

Using the Environment

Imagine that you are a new software engineer asked to write some code for a software system in the midst of development. The picture is not pretty: you see thousands of lines of code you've never seen before in your life. You have no idea which functions are important, and which ones are not. Where is the source for those library functions? What about the code for the class definitions, or the file that contains the important global definitions?

In this chapter we introduce a new environment that is designed to aid in these problems. Examples will be shown on how this environment can be used, and how it can be useful. Three visualization ideas will be shown in this chapter: source code cross-referencing, frame based visualization, and Lisp code animation. These ideas are introduced here, and are described in more detail in Chapters 5-7. Also, this chapter will introduce, through example, a multiscale environment called Pa3d. We will explore Pa3d and other multiscale environments in Chapter 4.

2.1 Source Code Cross-Referencing

Figure 2-1 shows source code in the Pa3d environment.\footnote{The diagrams in this thesis are schematic diagrams of Pa3d and not actual screen shots. This is due to some difficulty we had in capturing quality screen shots. For actual screen shots, see Bederson's paper on Pad++.} Through the use of the multiscale environment we can view the code at different scales. Source code is
arranged in columns, each column corresponding to a source code file. Also notice that at different scales, different levels of detail may be seen. If many lines of code are seen at once, then only the lines without the detail of text may be shown. As we zoom in closer to the actual code, the actual text may be seen. Figure 2-2 illustrates this.

Suppose we are looking at a function in this environment, and we would like to see the definition of this function. By placing the cursor on the source code line of the function call, and pressing CTRL-s, we can navigate from the function call to the function definition. Navigation occurs by zooming out until both the function call and the function definition appear on the screen, and then simultaneously panning and zooming until the definition appears centered on the screen. An example of this appears in Figure 2-3. This is an example of navigation based cross-referencing.

Now let’s consider the opposite problem. Suppose you are looking at the definition of a particular function and would like to see all the calls to that function. We can accomplish this through one of two ways. First, shown in Figure 2-4, we can change the color and increase the scale of the source code lines corresponding to the function...
calls. We do this by placing the cursor over the `defun` line and pressing CTRL-z. Thus, one can easily pick them out from the rest of the source code. In the case of Figure 2-4, we highlight lines that contain the `abs` function. In Figure 2-5, we show another way, which is to draw lines from the definition to each of the function calls. We do this in a similar way to the previous cross-referencing, except we press META-z. This particular way of cross referencing has one advantage over the previous way, in that one can pick up the location of the function calls no matter what level of detail is shown for the source code (this is also seen in Figure 2-5; the highlights would not show up in this level of detail). These two visualizations are an example of graphically based cross-referencing.

### 2.2 Frame Based Visualization

While source code cross-referencing may help for certain tasks, it does suffer from some of the same problems as viewing source code outside of a multiscale environment. Viewing thousands of lines of code in this manner does not gain the viewer much more
Figure 2-3: Navigating from a function call to a function definition
(defun splice (p q)
  (let ((v (car (last p)))
         (w (first q)))
    (and (far-out v)
         (far-out w)
         (>= (abs (- v w)) path-out)
    ; Two far-apart far-out points
    ; outside the perimeter
    (let* ((pdiff (phase (/ v w)))
           (npoints (floor (abs (delta (/ pdiff (+ npoints 1))
                    (incr (cis delta)))))

R:1, C:1, O:1

Figure 2-4: Highlighting function calls

Figure 2-5: Graphical links from definition to calls
information, than viewing thousands of lines of code in an text editor. We would like to gain more information from a more global view of code. We do this by introducing a more graphical way at looking at software systems, called a frame based visualization.

In Figure 2-6, we take a look at this type of visualization. Shown in the environment are nested graphical boxes. The white boxes represent source code files, and the colored boxes within them represent Lisp expressions. Using the multiscale abilities of this environment, we can nest these boxes called frames at an arbitrary depth, and can view them by zooming in to a particular frame, as shown in Figure 2-7.

Each frame represents a Lisp expression, and frames take on different colors for different types of expressions. The structure of the frame also differs for different expressions. There are eight defined frames in this system, which are described in more detail in Chapter 6.

This type of representation for code lends itself to a more global view of the software system. Interesting views can be seen for different software systems. For example, one file may contain mostly class definition frames, or mostly function definition frames. Another file may have a mix of these. From this, one can get a view
of the organization of the software system that is not easily seen from simply viewing source code.

2.3 Lisp Code Animation

In this section we introduce the final visualization idea. While static representations of code can be very useful, we desire a more dynamic view of code. After all, we use the verbs "run" and "execute" quite often when talking about code. We introduce a dynamic system based on the previous frame visualization that shows the execution of Lisp code through animation.\footnote{The reader should note that the format for the function definition frames has changed slightly, and in fact, the format of the function definition frame in the animation system is slightly different than that of the static frame system.}
Consider the following Lisp expressions:

(defun sum-of-squares (a b)
  (let ((a-squared (* a a))
        (b-squared (* b b)))
   (+ a-squared b-squared)))
(sum-of-squares 3 4)

These expressions, while simple, will serve as a good example of the animation system presented here.

We evaluate the first expression, which produces a function definition frame, similar to the ones seen in the frame based visualization. This particular frame will not be shown until we actually evaluate the body of the definition. Evaluating the second expression produces a function call frame. This frame is shown in Figure 2-8.

We can then start to evaluate this frame, which produces an animation. The arguments of the expression are evaluated first. The animation for this evaluation is shown in Figure 2-9. We then change the function call frame into the function
Figure 2-9: Evaluating the arguments

definition frame that was created when we evaluated the first expression, shown in Figure 2-10.

Then we evaluate the body of this frame. First, we evaluate the bindings in the let statement. Using a graphical version of the substitution model, we show the frames in the local variables section of the definition frame change into binding frames, which show the value of a-squared and b-squared. This is shown in Figure 2-11.

Once we have evaluated the local variables, we now execute the body within the let statement. Once again, we apply the substitution model by changing the frames containing the variables a-squared and b-squared into their proper values (9 and 16 respectively). This is shown in Figure 2-12. Once that happens, we evaluate the body, which is another function call frame corresponding to the addition of a-squared and b-squared. Evaluating this expression then creates a new simple frame, which corresponds to the value of the evaluated expression. The final picture is shown in Figure 2-13.

We have just seen three ideas that combine software visualization and the multiscale environment. All three ideas are potentially powerful ones in aiding the under-
Figure 2-10: The function definition frame

Figure 2-11: Evaluating the local variables
Figure 2-12: Applying the substitution model again

Figure 2-13: The final picture
standing of large and complex software systems. In later chapters we will describe these ideas in more detail, including some of the implementation issues involved. The next two chapters will give background on the field of software visualization, and the evolution of the multiscale environment. Particular attention will be paid to the environment called Pa3d, which is the environment used in this thesis.
Chapter 3

Software Visualization

Software visualization is the use of visual techniques and modern human-computer interaction technology to facilitate better human understanding of computer software. Software visualization is an idea that has been explored for as long as software has been written. An early example of software visualization was flowcharts. More modern examples, which take advantage of the abilities of a graphics workstation, will be described in more detail in this section.

Software visualization may be approached in many different ways. We will describe five approaches: program visualization, visual programming, programming by example, algorithm visualization, and code visualization. These five methods cover many software visualization systems, and provide a solid basis for understanding software visualization. This is by no means a definitive taxonomy of software visualization. For more detailed treatments of this subject, see Myers [18] and Price [20].

3.1 Program Visualization

Program visualization can be thought of as the use of visual techniques to enhance the general human understanding of many aspects of a computer program. While other categories of software visualization tend to focus on one aspect of the computer program, program visualization tends to provide general tools for understanding many aspects of the program.
Incense [17] is an example of a program visualization system. Created by Brad Myers, while at Xerox PARC, it created visual debugging tools for programmers of the SmallStar [12] system. The goal of the Incense project was to make debugging easier by presenting data structures to programmers in the way that they would draw them by hand on paper. Another, more current, program visualization system is the UWPI compiler [13] developed at the University of Washington. UWPI analyzed Pascal source code and produced pictures of the data structures that resided within the code.

3.2 Visual Programming

Visual programming is the use of visual techniques to specify a program. Instead of using text to specify a set of instructions to a computer, the visual programmer uses pictures or icons; generally something more visual than simple text. Flowcharting is an early and primitive example of visual programming. A later example was the AMBIT/G system [7]. This system supported symbolic manipulation programming using pictures. Programs and data were represented as directed graphs, and programming operated by pattern matching. Algorithms could be described graphically as local transformations on graphs. Shu’s book on Visual Programming [23] is definitive for these type of systems.

3.3 Programming by Example

Programming by example\(^1\) involves presenting data to the computer, and using this data as an example of what the computer is supposed to process. In some ways, this is similar to visual programming, the main difference being that visual programming works directly on the specification, while programming by example works on the input data.

Tinker [15], created by Henry Lieberman at the MIT Media Lab was one of the

\(^1\)The more common term used today is Programming by Demonstration
first, and is one of the more definitive Programming by Example systems. Tinker is a Lisp-based system that was designed to aid user interface design by creating code out of examples of what the programmer wanted. Rehearsal World [11] by Gould is another programming by example system, which was designed to help teachers develop lesson plans through carefully chosen examples.

3.4 Algorithm Visualization

Algorithm visualization (also called algorithm animation) uses visual techniques to enhance the high-level understanding of a piece of code. The idea is to provide understanding about the algorithm, but not necessarily the implementation.

The classic example of algorithm animation is the Sorting out Sorting video created by Baecker [2]. This video demonstrated different sorting algorithms in a unique way. Instead of looking at code for each type of sorting algorithm, you would see the effect of each sorting algorithm on simple graphical objects, such as rectangles with varying heights and colors. The rectangles would start out colored blue or green, and turned red when they were moved into their final (sorted) position. The video would also show different sorting algorithms running in parallel, so the speed between the sorts could be compared. Another system of algorithm animation is BALSA [6] developed at Brown University. Many of the pictures that appear in Sedgwick's algorithms
Figure 3-2: Sorting out Sorting

book [22] were developed through BALSA.

3.5 Code Visualization

Code visualization deals with visually presenting actual code. Note that this differs from algorithm visualization in that code visualization deals with a particular implementation of a program rather than its high level description (which may have many different implementations).

SEE [3] is a good example of code visualization. It uses text formatting to produce a "code book". Various aspects of code, such as procedures, variables, and flow control were visually organized through the use of text formatting, to enable the programmer to zero in on key points of interest. SeeSoft [9], developed by Stephen Eick is a recent
look at software as a type of information system. He uses revision history and views many thousands of lines of source code at once. The result is an impressionistic view of the current state of the code, where the programmer can pick out sections that have been recently changed, or have remained relatively stable.

### 3.6 General Comments

Software visualization systems have been in existence for more than twenty years. Many have been extremely useful in education, such as BALSA and Tango. Visualizations similar to Sorting out Sorting have been very useful in distinguishing between different sorting algorithms. Many of these systems require some "pre-processing" in order to produce a useful visualization or animation. This is not necessarily a bad thing, as this can produce highly specialized and useful visualizations. However, we would like to see a visualization system that can work with arbitrary code in any format. Such a visualization would be much more useful in the development environment, where we might not want to view the execution of a particular algorithm per se; just a particular piece of code. SEE and SeeSoft are examples of this type of vi-
sualization, which works on any piece of code\textsuperscript{2}. These two systems are the exception, rather than the norm, and we would like to extend this type of visualization. The three ideas explored in this thesis fall into this realm of software visualization.

\footnote{\textsuperscript{2}Although SEE is actually language specific (towards C), and SeeSoft works on systems annotated by SCCS or RCS}
Figure 3-5: SeeSoft
Chapter 4

Multiscale Environments

A multiscale environment is an environment where information can be entered at any location and at any scale. One way to see this is to imagine an surface similar to a bulletin board that is infinitely long and infinitely high. In addition to an infinite extent, we can enter information on a scale from meters to microns. To accomplish this, we allow ourselves to adjust our view of scale, much like a microscope, except the zoom-in factor is not limited by the machinery of the microscope. In other words, we can consider a multiscale environment to be one of infinite extent and resolution.

To understand the implications of infinite extent and resolution, consider the following. Imagine that you are given a surface to write upon, like a bulletin board or a whiteboard. Now imagine that you are blessed with extraordinary eyesight, and can read writing the size of an atom. Now add extraordinary hand-eye coordination to your eyesight, and you can write information on the microscopic level. With these abilities, you would be able to enter and read an enormous amount of information in a small amount of space.

Of course, nobody has such good eyesight or hand-eye coordination, and normally we can not store information this way. However, the computer provides a partial solution. We can use the computer to be our "eyes" in this situation, much in the same way a magnifying glass enhances our vision. If we were given an infinitely large surface to enter information upon, we could use this "magnifying glass" to aid our reading and writing. This is what a multiscale environment provides. The systems
described here provide an infinitely "zoomable" environment to work in, thus allowing us to work with large amounts of information in a reasonable amount of space.

Early work in multiscale environments started with the SDMS [8] system. Recent work includes the Pad [19] system developed at NYU, and the Pa3d and Pad++ [4] systems developed at Bellcore.

4.1 SDMS

William Donelson first explored multiscale environments with the Spatial Data Management System (SDMS) system, which he developed in the Architecture Machine Group at MIT. SDMS arose out of the Architecture Machine Group's work in creating interactive systems that relied on a user's prior knowledge. Donelson used SDMS to exploit users' notion of spatiality. He noted that many people liked to describe location of objects in relative terms, as in "I put my book next to the phone", or "My mouse is next to my monitor", and so forth. He noted that a desktop was a good metaphor for this type of organization. SDMS was one of the first systems to explore this idea.

The user environment for SDMS was called the Media Room. It was an 18' wide by 11' deep, 11 1/2' high room. One wall of the room contained a large six foot by eight foot rear projection television screen. An eight channel sound system was used to produce directional special effects, like Doppler shifts. Eight feet from the display screen sat an instrumented control chair. On each arm of the chair was a joystick and a 3” square touch sensitive tablet called a "Joypad". Users could also use a 10” square Summagraphics tablet for annotation.

The large screen would display a portion of what Donelson called a data surface. SDMS used a planar geometry to place and organize its data. The data surface was much larger than the large screen could display at one time. A smaller display was used in conjunction with the large display to show a world view, as well as a "you are here" rectangle to mark your current location. Users could navigate through SDMS in two ways. One was to use the joysticks on each arm on the control chair. The
other way was to use the touch sensitive screen for the world view.

The SDMS system was significant in that it was one of the first to explore this idea of spatiality, and the notion that information organization should exploit strengths of human perception. In addition to the spatial organization, a multiscale element was added, which became more of a focus for the next two environments, which were developed fifteen years later.

4.2 Pad

Pad is the result of some recent work done by Ken Perlin and David Fox at New York University. Pad was developed as an alternative approach to computer interfaces, and, like SDMS, was designed to take advantage of people's spatial organization abilities. One can think of Pad as a two-dimensional infinite plane that is shared by users, much in the same way a network file system is shared by users. Objects can be placed in Pad at a well-defined region on the surface, with a specific location and scale.

Pad contains portals that aid navigation. In Pad, a portal is a magnifying glass that can peer into and roam over different parts of the Pad surface. A portal can view highly magnified objects or a broad panorama of objects. An important distinction must be made between the notion of a portal in Pad and a window in a more traditional interface. A window represents a link to a specific object, such as UNIX shell,
or an Emacs editor. A portal represents a view onto a shared desktop. You may or may not be looking at a specific object, and links to objects can be established or broken on the fly.

An important idea in Pad and in other multiscale environments is the concept of semantic zooming. Since we can see an objects at different magnifications in an environment like Pad, it becomes useful see different types of information at different magnifications. For example, when viewing a text document in Pad, at a low magnification, we might want to see only the title of the document. As we zoom in closer, and view the document at a higher magnification, more details may become apparent, such as the chapter headings. Or, we may see a short outline or abstract of the document. If we choose to zoom in even closer, we might start to see entire sections of the text itself. In short, an environment such as Pad can support the idea of different representations for an object at different levels of magnification. This is a powerful idea, and allows us to browse large amounts of information effectively.

4.3 Pa3d

Building upon Perlin’s work on Pad, Ben Bederson at Bellcore has developed Pa3d. Like Pad, Pa3d can be thought of as an infinite two-dimensional plane where objects can occupy a well defined region. However, there are differences between Pad and Pa3d, which have arisen out of Bederson’s motivation for developing Pa3d.

4.3.1 Background and Motivation

While Perlin was more interested in exploring the idea of a shared workspace with Pad, Pa3d was motivated by the desire to visualize large information spaces. Pa3d was built into a 3D visualization prototyping environment called Ar3t, which was used by the Computer Graphics and Interactive Media Research Group at Bellcore. Pa3d exists as a three-dimensional object in the Ar3t world. On the surface of this object exists a surface similar to Perlin’s Pad surface, with a few differences.
• Pa3d is a 2D surface that exists in a 3D world. Therefore 3D objects may be placed on the surface.

• Objects in Pa3d are defined in a high-resolution three-dimensional world space, and are only rasterized upon rendering.

• Pa3d exploits the special graphics hardware found on the SGI machines. Pad may be run on many machines, and does not use any special hardware.

• Pa3d objects are stored hierarchically, which makes grouping of objects easy, and provides a kind of spatial indexing which increases efficiency.

• Pa3d has a continuous zoom, while Pad zooms by factors of two.

4.3.2 Implementation

Pa3d exists as an object within an environment called Ar3t [5], also developed at Bellcore. Ar3t is a Lisp (and CLOS) based environment. It runs on Silicon Graphics computers and is based on their GL graphics library. Ar3t provides a high-level interface to GL, and contains a set of CLOS mixins that support a variety of 3D interface objects. Ar3t also provides a set of Motif-like widgets, as well as the ability to incorporate these widgets into the Ar3t world itself.

4.3.3 The Pa3d World

Pa3d is, for all practical purposes, infinite in extent and resolution\(^1\). The user effectively has a finite window looking onto an infinite space. This means that we can arbitrarily scale and translate information in Pa3d.

Pa3d uses a two-handed interface for navigation. In the dominant hand is a mouse or pen, which is used to point to objects. A 6-degree of freedom input device called a spaceball is used to pick up and move objects, as well as navigate within Pa3d itself. Selecting an object attaches the spaceball to that object. The object can then

\(^1\)Technically, the resolution of Pa3d is limited by the floating point capability of the SGI, which is much more than what is typically needed.
be translated or rotated directly. If we attach the spaceball to a Pa3d, we enter a
special navigation mode that attaches the spaceball to the information on the Pa3d.
Pushing the spaceball up, down, right, or left translates the information accordingly.
Pushing the spaceball forwards or backwards rescales the information. In addition
to attaching the spaceball to the Pa3d itself, we can also attach it to specific objects
within the Pa3d, translating and scaling them as we wish.

The mouse, pen, and keyboard are used for adding information to Pa3d. We can
draw lines of different colors and width with the mouse or pen. We can also place a
cursor on the Pa3d with the mouse, and enter text at the cursor with the keyboard.
The text entered onto Pa3d is editable, just as if we had entered it into an Emacs
buffer. In fact, many of the Emacs keystrokes have been incorporated into Pa3d. For
example, Control-p and Control-n move the cursor up and down, respectively, while
Control-b and Control-f move the cursor back and forth. Text is not the only type
of information that we can put into Pa3d. Pressing the Control key, and the left
button of the mouse simultaneously brings up a menu that allows the user to add
different objects to Pa3d, such as bit-mapped images, entire text files, and pre-defined
graphical objects. This menu also allows the user to start a Pa3d application, such
as the directory browser.

Pa3d objects are defined hierarchically. A basic Pa3d object is composed of Ar3t
CLOS mixins that provide generic methods of handling, for example, device input
through the mouse or spaceball, movement within the Ar3t world, or hierarchical
composition of objects. A Pa3d object is composed of a surface, which defines posi-
tion and scale, and a collection of graphics, images, or text. An object is also defined
through a set of methods that describe, among other things, the rendering of ob-
jects and the handling of input events. Many default methods can be used through
inheritence of Ar3t mixins.

4.3.4 Features

Pa3d includes many features that demonstrate its power as an interface. In this
section we will describe some of these features.
Levels of Detail and Refinement

In computer graphics, there is often the problem of deciding at which level of detail we should draw an object. The problem is particularly relevant in Pa3d due to the various scales at which we can draw an object. There is a tradeoff between the amount of information we want to show, and the speed of rendering the object. When we increase the detail of an object, we increase the rendering time of the object. In interactive systems, this issue is very important. In Pa3d, this problem is dealt with by defining various levels of detail for different scale ranges. An example of this would be the drawing of an image in Pa3d. If we start by displaying the image at a small scale, we do not get much value added by fully rendering the image. So we can increase rendering speed by displaying only part of the image (say, for example, by rendering every other pixel, or every third pixel). As we increase the scale (i.e. zoom in to the image), we would like to render more of the image. As we pass through different ranges of scale, more of the image is rendered, until we are finally close enough to appreciate a fully rendered image.

This issue is not limited to scale transformations. If we translate an image on Pa3d, we become less interested in the details of the image, as we are unable to see the details. Thus, we would like to use a lower level of detail for an image while we are moving it, and revert to a higher level of detail while the image is not being moved. Pa3d uses a concept called temporal refinement to accomplish this. When an image is moved, then image is drawn in a low level of detail. After the image has stopped moving, then the imaged is "refined" over time. Successive levels of detail are used to render the image. Given enough time, the image will be rendered at a level of detail appropriate for its scale, at least until it is moved again.

Semantic Zooming and Fisheye Views

Like Perlin's Pad system, Pa3d supports semantic zooming. An example of software visualization that uses the idea of semantic zooming is an interesting view of source code text. Instead of using traditional scaling, one can attach different scaling properties to different lines of code. For example, we can keep interesting lines a constant
size while other lines scale normally. For lisp code, interesting lines can be chosen according to keywords within the line, such as `defun`, `defclass`, `defmethod`, etc. As we zoom out, these interesting lines maintain their size and detail, while other lines shrink and lose their detail. This is one way we can get a quick overview of the functions in a file.

A more general way of taking advantage of different scale properties for different lines of code would be the *fisheye* view. Developed first by Furnas [10], the fisheye view is a method of simultaneously seeing detail and broad structure. We assign a degree of interest to each line of code, and we assign scaling properties to each line based on that degree. We could, for example, assign degree of interest as a function of the line's distance from the cursor's current position. Lines around the cursor would appear in greater detail, and the detail would fall off further away from the cursor. We may also look at the parenthesis structure of Lisp code, which defines a hierarchical structure, and assign degree of interest as a function of a line's place within that structure. These two degree of interest functions can be combined to get yet another view of code. A well designed function used for a fisheye view in Pa3d can provide some interesting visualizations.

**Directory Browser**

Pa3d is designed to work best with hierarchical data. A natural example of a hierarchical structure is a computer's file system. A directory browser was developed for Pa3d, in which files and directories are graphically depicted. Files and directories are represented by square outlines. When zoomed close enough to a file outline, the contents of the file itself may be seen within the outline. If the file is a bitmap file, the bitmap image itself may be seen; similarly, if the file is a text file, the text itself may be seen. Text that appears within the file outline may subsequently be edited (using Pa3d's editing capabilities), or be shown in the fisheye mode described above. Directories have the same square outline as files, except that more outlines (files or subdirectories) appear within the directory outline. The directory browser is a typical, but useful example of Pa3d visualization. As we zoom out, we can get a general
impression of the directory structure. Zooming in provides a more detailed (and less
global) view of the directory and its contents. The browser provides a simple, and
useful representation of the objects we are exploring. We will use this representation
later when we explore a frame system for software visualization.

**Read Wear and Edit Wear**

Read Wear and Edit Wear are concepts that were developed independent of Pa3d,
but is well suited to Pa3d. Basically, we consider the idea that physical objects "wear
down" as they are used. For example, a paperback's binding becomes more pliable
the more times we open and close it. Also, the pages of the book may become "dog-
eared" as we turn them. Generally an object takes on physical characteristics through
use over time. Read Wear and Edit Wear applies this concept to digital objects. As
parts of a digital object are read or edited, they acquire analogous characteristics
to our book. This can be graphically represented by simply drawing different parts
of the object in a different color, or by adding a graphical annotation to the object
itself. Through this idea, we can see which parts of a distributed paper are being read
the most. Or, if the paper is being written jointly, which parts are being rewritten
the most. We can exploit Pa3d's ability to easily scale text to aid this visualization.
If we zoom away from a document that has been "read worn", we can easily see
which major portions have been read and which have not. Zooming towards areas of
interest, we can see the text itself, and a more precise distinction between levels of
wear.

We can apply the idea of Edit Wear to software visualization through the use
of software revision control systems such as RCS [24] or SCCS [21]. Through these
systems, we can assign an age (the time since the last edit) to each source code line in
the system. Again, Pa3d’s control over rendering text allows us to explore different
ways of depicting the editing history of the software. One can also imagine other
similar visualizations for source code, such as bug reports, or profiling histories. For
more detail in work done in this area, see Hill et al [14].
Chapter 5

Exploring Source Code in Pa3d

In this chapter we will describe the first of our three main ideas in software visualization in the Pa3d environment. The first idea is that of using actual source code text in Pa3d. We will explore ways in which the Pa3d environment allows us to manipulate the text to give a better presentation of code. Manipulating the actual source code text is a very practical method of software visualization. Baecker's SEE system is a very good example of what can be done with simple text formatting. Here, we take advantage of Pa3d's special scaling abilities with text.

5.1 Why View Source Code in Pa3d?

We have already seen one example of manipulating text to highlight features within the text. The fisheye view described in chapter 2 allows us to maintain a level of "importance" associated with a line of code. The code can be scaled arbitrarily to show this importance. This allows us some very powerful formatting within Pa3d.

Another advantage of Pa3d is the ability to easily move between views which show different amounts of lines of code. In other words, we can vary the "code density", i.e. the amount of code shown in the screen. In one moment we can be viewing thousands of lines of code, perhaps an entire software system, and in the next moment we can view a single procedure within that system. There is, of course, a trade-off between code density and the level of detail at which we can see the actual code. As the code
density grows higher, the level of detail lessens. In a high density view of code, a line of code may be drawn as a simple line, but in a low density view of code, we may actually see the actual text of the code itself. In between, we can take advantage of Pa3d’s ability to quickly switch between different representations to help us make this tradeoff.

The best example of this use of code density is the Read Wear/Edit Wear example shown in chapter 2. In one view onto Pa3d, we see a "macro view" of the code; a high density view. In that view we can get a general impression of the wear properties of the file itself. However, as a means of tradeoff, detail about the lines itself are suppressed. In another view, we see a "micro view" of the code; a low density view. Now we look at a specific procedure within the file. Again, there is a tradeoff. Now given a lower density view than before, we can add more detail to the display, with bar graphs added to the side. This gives us a more precise picture of the wear on this specific part of the code.

The idea of trading off code density with level of detail is not an idea that comes directly out of Pa3d. Eick’s SeeSoft system also makes use of this idea, using one window for his high density view, and a second window for looking at the text itself (his low density view). Pa3d, with its smooth zooming abilities, makes this idea attractive, since it is now very easy to switch between representations. In addition, we can see the intermediate views; say, with a medium density view with a medium level of detail. Eick’s system only shows the extreme ends of the spectrum.

In viewing source code text, we would like to take advantage of Donelson’s original ideas in his SDMS system. We would like to exploit users’ ability of spatiality. We would like to see if it is useful for programmers to know "where" a specific procedure is, in a spatial sense. We will couple this idea with the idea of linking procedure calls to their definitions. In the next section, we will see how these two ideas are used together to enhance understanding of a large amount of code.
5.2 Cross Referencing of Source Code

In this section we explore the idea of code cross referencing, meaning, the linking of the text of a procedure, to the text of the procedure calls within a large software system. We will do this in two ways: navigation-based and graphically-based. Each of these ways takes advantage of a feature in Pa3d.

Linking procedure definitions to their procedure calls is almost always a useful exercise for those trying to understand an unfamiliar software system in the midst of development. We would like a way for the new programmer to move quickly between the call and the definition. Often, especially in large systems, simply finding the definition can be difficult. Thus, the new programmer wastes much time searching rather than assimilating. Emacs’ TAGS system provides some help with this problem. Simply by pressing META-. the programmer can have Emacs search the system for the definition of the function that the cursor is currently pointing at. This helps reduce dramatically the search problem, but it is not very useful in helping the programmer actually understand the structure of the system. The programmer still has no real sense of the location of important function definitions. In Pa3d, we add some spatial information about the system. The idea of cross-referencing gives us a more tangible sense of “where” that important function definition may reside.

How might cross-referencing be useful? Imagine that you are working on a project many thousands of lines long. You see a function that you believe that you can optimize with a good amount work. You wonder if performing that optimization is worth it. You can use graphical cross-referencing to easily see how many times that function is called. This saves you the trouble of searching the entire system by hand.

5.2.1 Cross Referencing Through Navigation

We would first like to make use of Pa3d’s navigation abilities. An environment such as Pa3d tries to make use of one’s ability to spatially organize information. The idea is that knowing where an object is, perhaps in relation to another object, enables us to easily organize these objects. With source code, we would like to use this idea to
help us relate function calls to their definitions or vice versa.

Pa3d has the ability to "remember" different locations, which consist of a place on the surface and a particular scale. Once we remember a location, Pa3d then has the ability to navigate back to that location. Navigation within Pa3d is a computer-controlled animation from one location to another. Pa3d zooms out to a scale at which both the initial and target location are located on the screen, then simultaneously zooms in and translates to the target location. Pa3d navigation allows us not only to easily move between important spots in this information space, but it also allows us to see the spatial relation between the initial and target locations.

We use navigation to relate function calls and function definitions. First we "remember" the locations of the different function definitions. For each definition, we create a table entry comprised of the name of the function, and its location on Pa3d. For the entire system of code, we create a table very similar to the Emacs TAGS table. By placing the cursor on a particular function call, and pressing Ctrl-S, Pa3d searches the table for the corresponding function definition. If one exists, the Pa3d location for this definition is returned, and a navigation to the definition location is performed.

5.2.2 Cross Referencing Through Graphics

Navigation is only one way to provide a link between function calls and definitions. Navigation is a nice feature in that it automatically moves us from the call to the definition, and gives us a sense of the call's position related to the definition. However, at times we might want something different in terms of cross-referencing. For example, we might want to see all the function calls from a given function definition, which is the "inverse" of the problem described above. Instead of moving from a function call to a function definition, we want to start at the function definition and see the corresponding function calls. This type of mapping is a one-to-many mapping; a function definition may have more than one corresponding function call, as opposed to the one-to-one mapping described above (where a function call usually has only one corresponding function definition). This type of mapping is not really suitable
for navigation, so we resort to other, more graphical means.

One simple way to link function definitions to function calls is to highlight calls for a given definition. The ability for Pa3d to use a specific scale and/or color for a single line of code makes this easy. This feature can be activated by placing the cursor on the function header of the definition we would like to cross-reference, and pressing Ctrl-Z. The lines of code that contain calls to that function are then enlarged in scale and colored green. Zooming out enables one to see many, if not all, of the highlighted function calls. One can then get a sense of "location" for the calls; are they spread among different files, or are they concentrated in one file? Has someone added a call recently? Again, such an impressionistic view may be very useful to a programmer exploring a large software system for the first time. We can also add a variant to this theme by actually "linking" the definition to the calls, by drawing a red line from the definition to each of the calls.
Chapter 6

A Frame System for Visualizing Code

While looking at source code text can be illuminating, we would like other ways at looking at code, in particular, a more graphical way of looking at code. In this chapter we describe a such a way, which we call a frame system. It’s immediate roots are the directory browser described in Chapter 4, although there are similarities between this system and Abelson’s Boxer system [1]. This system is meant to enhance code written for any language, as such, no particularly language dependent concepts are needed for this system. For the purposes of this thesis, we have chosen Lisp code as sample code to visualize. The main reason for this is Lisp’s simple syntax. The syntax allows one to easily break down a Lisp program into its constituent procedures, and the procedures into its own parts. This accounts for Lisp’s popularity in software visualization research.

6.1 Frame System Description

The basic unit in this frame system is a frame. A frame, in the graphical sense, is a Pa3d object that appears as a square outline, similar to the "boxes" used in the directory browser. Semantically, a frame corresponds to a Lisp s-expression. In Lisp programs, s-expressions may be contained within s-expressions. Similarly, in the
frame system, frames may be included within frames, recursively. Frames come in several types, called: procedure, function call, class, if, conditional, simple, and data. The organization of frames represents a partition of Lisp s-expressions. Any Lisp s-expression can be mapped to one type of frame. Each type of frame has a different appearance on Pa3d. While each appears as square outline with text inside, each type has a different outline color, and each type is divided up into regions within the square, which differ among types.

Frames, like any other object in Pa3d, can be viewed at various scales and various levels of detail. As before, when we were viewing source code text, we have a tradeoff between "frame density" and level of detail shown. However, in the case of the frame system, the tradeoff is not so straightforward. Since frames can appear within frames, objects seen at one time can have different scales. The level of detail depends on the scale, so we may see some frames with a high level of detail, and some frames at a low level of detail, appear on the screen simultaneously. In general, as we zoom in for closer detail, the number of frames that appear on the screen decreases, since there are fewer amount of s-expressions to examine as we zoom in. For example, at a lower level of magnification, we may be looking at a file of Lisp code, and at a higher level of magnification, we may be looking at a specific procedure within that file.

### 6.2 Motivation for Using Frames

Having seen a basic description of the frame system, one might wonder why we use them at all. The answer is twofold: first, we would like to take advantage of the ability of multiscale environments to represent hierarchical data, and second, we desire a more graphical way at looking at code over source code text.

We have already seen a good example of hierarchical data as a Pa3d visualization: the directory browser introduced in Chapter 3. Each file or subdirectory is represented by a graphical box. Boxes can appear in boxes, corresponding to files appearing within subdirectories. Lisp code, as well as file systems, has a hierarchical structure. A single software system may be composed of different source code files. A file may be made
up of many function, variable, class, or method definitions. A definition may be made of more definitions, function calls, or standard Lisp forms. Each level of the hierarchy is represented by a frame at a specific scale, and we enclose frames within frames to show the relationship between different levels. Viewing code; indeed, viewing the whole software system as a hierarchy is very natural. Many programmers break down problems into some sort of hierarchical structure as a way of controlling complexity. Multiscale environments are well suited to visualizing hierarchy, so the frame system is a good example of the suitability of software visualization in such environments.

Introducing graphics into our visualization of software systems is also necessary. While looking at "enhanced" text as described in previous chapters is a good way to gain better understanding code, it still has some fundamental limitations. Graphics gives us a more flexible representation, and allows us to emphasize or de-emphasize certain aspects of code. Especially important is the ability for graphics to suppress detail when we need to, something that is very difficult to do when actually looking at the source code itself. A non-trivial issue that arises is the question of which detail should be suppressed, and which should not. This issue will be dealt with later when we discuss the parsing of Lisp code to generate frames.

6.3 Frame Descriptions

In this section we will describe the frames that reside within the frame system. There are eight types of frames: function definition, function call, data, simple, if-then, conditional, class definition, and method definition.

Function Definition Frames

A function definition frame is created for defun expressions, which define functions in Lisp. Function definition frames have four slots: name, parameters, local variables, and body. All sections except the name slot (which just contains the name of the function) contain additional frames. The parameters slot contains data frames corresponding to the parameters of the function. Data frames are also kept in the
local variables slot, to denote local variables of the function. In this frame system, local variables are created when a \texttt{let}, \texttt{let*}, \texttt{setf}, or \texttt{setq} expression is encountered. This handles the majority, but not all local variables in Lisp. The body slot contains frames that comprise the body of the function, in this case all expressions not covered by the above slots. Some special parsing was done to handle the body of \texttt{let} and \texttt{let*} and include those the body of those expressions in this slot.

**Function Call Frames**

A function call frame is created whenever a Lisp expression does not conviently fit into one of the seven other frames. Unfortunately, this leads to the creation of function call frames that are not function calls (see section 6.4 for more discussion of this issue). This type of frame has two slots: \texttt{name} and \texttt{arguments}. The name slot just contains the name of the function being called. The arguments slots contains the arguments to the function, which can any type of frame. Function call frames are often found within the other seven frames. For example, in the body in the function definition, or as part of the consequent of a conditional.

**Data Frames**

A data frame contains three slots: \texttt{name}, \texttt{type}, and \texttt{value}. The name slot contains the variable name and the type slot contains the variable type. These two slots contain text. The value slot contains frames which corresponds to the value of the variable. Most of the time this slot contains a simple frame which denotes the value.\footnote{This brings up the issue of representing values in this frame system. For the sake of simplicity, they are represented by simple frames, with the printed representation appearing as the text in these frames. This is one area of the frame system which could use more work.} Data frames are used to denote parameters of the function, and the local variables within a function.
Simple Frames

Simple frames are indeed simple in structure. They simply appear as a frame with text inside. We use simple frames to represent values (in which case the printed representation of the value appears in the box), or self-evaluating expressions.

If-Then Frames

This frame is created when we encounter an if expression. If-then frames contain three slots: clause, true-consequent, and false-consequent. The clause slot contains the Lisp text of the clause of the if-then statement. The true-consequent slot contains a frame that represents an expression to be executed when the clause evaluates to true. Similarly the false-consequent slot contains a frame that represents an expression to be executed when the clause evaluates to false. Usually a function call frame appears in the consequent slot.

Conditional Frames

A conditional frame is created when we encounter a cond expression. A conditional frame contains two types of slots: clause and consequent. The number of each type of slot depends on the number of clauses appearing in the expression itself. Each clause slot contains the text of the Lisp code corresponding to each clause in the expression. The consequent slot contains a sequence of frames that represent actions to take when the corresponding clause is true.

Class Definition Frames

This frame is created when we encounter a defclass expression. It contains three slots: name, inherited classes, and slots. The name slot contains text of the name of the class. The inherited classes and slots slots contain simple frames corresponding to each inherited class and class slot in appearing in the definition.²

²Simple frames are used here instead of text, so that one can see the amount of inherited classes or slots without zooming in very closely.
Method Definition Frames

This frame, which is created when we encounter a defmethod expression, is very similar to the function definition frame. The slots are similar, although we add an associated class slot. This slot contains the class that uses this method (shown in text).³

6.4 Parsing Lisp Code to Generate Frames

While the actual parsing of Lisp code itself is relatively simple (compared, say, to C code), a non-trivial issue regarding parsing arises. Namely, how should one parse

³Because of the multiple inheritance properties of CLOS, we simply show the parameters of the method to show what class is associated with this method.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Contents</th>
<th>Found In</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>name associated with expression</td>
<td>text</td>
<td>Func. Def.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Func. Call.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Class Def.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Method</td>
</tr>
<tr>
<td>Parameters</td>
<td>parameters for function</td>
<td>frames</td>
<td>Func. Def.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Method</td>
</tr>
<tr>
<td>Local variables</td>
<td>local variables in function</td>
<td>frames</td>
<td>Func. Def.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Method</td>
</tr>
<tr>
<td>Body</td>
<td>body of function</td>
<td>frames</td>
<td>Func. Def.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Method</td>
</tr>
<tr>
<td>Arguments</td>
<td>arguments of function call</td>
<td>frames</td>
<td>Func. Call.</td>
</tr>
<tr>
<td>Type</td>
<td>data type</td>
<td>text</td>
<td>Data</td>
</tr>
<tr>
<td>Value</td>
<td>value of expression</td>
<td>text</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Simple</td>
</tr>
<tr>
<td>Clause</td>
<td>predicate in a conditional</td>
<td>text</td>
<td>If-Then</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Conditional</td>
</tr>
<tr>
<td>True-Consq</td>
<td>result of true in if-then</td>
<td>frames</td>
<td>If-Then</td>
</tr>
<tr>
<td>False-Consq</td>
<td>result of false in if-then</td>
<td>frames</td>
<td>If-Then</td>
</tr>
<tr>
<td>Consequent</td>
<td>result of clause in conditional</td>
<td>frames</td>
<td>Conditional</td>
</tr>
<tr>
<td>Inherited Classes</td>
<td>inherited classes in a class definition</td>
<td>frames</td>
<td>Class Def.</td>
</tr>
<tr>
<td>Slots</td>
<td>slots in class definition</td>
<td>frames</td>
<td>Class Def.</td>
</tr>
<tr>
<td>Associated Class</td>
<td>class associated with method</td>
<td>text</td>
<td>Method</td>
</tr>
</tbody>
</table>

Table 6.2: Frame slot description summary
Lisp code to produce frames? More precisely, how should we partition the range of frames? Admittedly, the parsing presented here is far from perfect, most notably with regard to function call frames, which correspond to expressions not fitting into the other seven frames. Obviously, this covers a lot of expressions, many of which do not fall neatly into the idea of a simple function call. One solution to this is to expand the range of frames in our system, i.e. partition the space of expressions even further. However, this solution is not necessarily the best. One of the advantages of the current frame is system is the ability to easily discern the types of expressions that are in a particular file. This gets harder when we add more different types of frames.

A more serious issue in parsing is the internal organization of these frames. One could easily imagine a different way at looking at function definitions (one that did not have an explicit slot for local variables, for example). In this case we have tried to make the frames as language independent as possible, by choosing slots that would be common to all languages. All function definitions have parameters, local variables, and a body, whether the definition is written in C, Pascal, or Lisp. Similarly, function calls have arguments, which correspond to parameters in function

---

4defvar and in-package expressions are a good example of this.

---

<table>
<thead>
<tr>
<th>Class Definition Frame</th>
<th>Method Definition Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Class Name&gt;</td>
<td>&lt;Method Name&gt;</td>
</tr>
<tr>
<td>&lt;Inherited Classes&gt;</td>
<td>&lt;Parameters&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;Associated Classes&gt;</td>
</tr>
<tr>
<td>&lt;Slots&gt;</td>
<td>&lt;Local Variables&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;Body&gt;</td>
</tr>
</tbody>
</table>

Figure 6-2: Frame structure (cont.)
definitions. Thus, we have tried to take fundamental elements of programming and use those elements to guide our organization of frames. The trade-off is that a much more useful visualization may be gained by adding language-specific elements to it. The use of the substitution model in the animation visualization (shown in the next chapter) is a bit more specific to Lisp than any other language (although it is a useful way of showing variable bindings in any language), and is a crucial aspect in that visualization. Certainly, further explorations into the space of possibilities is desired.
Chapter 7

Lisp Program Animation

Up to this point we have looked at static representations of code. We have looked at source code itself and enhanced it through cross-referencing, and looked at a graphical representation of code through the frame system. For large software systems, the static view of code is useful for discerning the overall structure of the software system. Static views are useful in describing the framework of the system, which goes a long way towards an understanding of the system.

However, static views are often not good enough. They can not tell how a program executes. At some point in our understanding of code, we look like to see code in action. Instead of just seeing the structure of the system, we would like to see its operation. As a result, we examine more dynamic views of code; views that show us how the system runs. In this chapter, we will examine the animation of Lisp code. We will use animation in the context of the frame system we looked at in the previous chapter, and will describe a system that can provide us with a graphical "movie" given a Lisp function call and definition. The idea of graphical animation of Lisp code is not a new one, see for example Lieberman's paper on re-rooting [16].

In this instance, the ability to manipulate the scale of frames is important. We use scale to focus attention on important frames, i.e. frames that contain the expression currently being evaluated. Scale also helps us to de-emphasize frames, keeping focus on the frame being evaluated. This idea of focusing attention properly is important
when dealing with Lisp animation\textsuperscript{1}, and the multiscale environment allows us much
control in this area.

\section*{7.1 Using the Frame System to Animate Code}

This animation system described in this chapter is an extension of the frame system
described in the previous chapter. Frames are used for animation for a few reasons.
First, the mapping of Lisp expressions into frames is not only appropriate for a static
representation of code, but also for a dynamic representation. Frame animation
corresponds to the evaluation of a Lisp expression. A second and more practical reason
is that the frame system provides a good substrate in which to build new visualizations.
Particularly important are the graphics and parsing routines "inherited" from the
frame system, which are heavily used in the animation system.

The animation system works in the following way. A Lisp expression is read in and
translated into a frame (using the same engine that produced frames in the previous
chapter). The frame is then passed into a \textit{frame evaluator}, which reads in a frame,
evaluates the expression that corresponds to that frame, and returns another frame
that corresponds to the value of the evaluated expression. A "side effect" of this
evaluation is the actual animation of the frames in the environment. As frames are
evaluated, some corresponding animation occurs.

\textsuperscript{1}Lieberman's re-rooting paper deals with this issue in more detail.
As an example of the system at work, let us consider the following code sequence (which we had looked at in chapter 2):

```lisp
(defun sum-of-squares (a b)
  (let ((a-squared (* a a))
       (b-squared (* b b)))
    (+ a-squared b-squared)))
(sum-of-squares 3 4)
```

The animation system evaluates the expressions in order. Evaluating the first `defun` expression produces a function definition frame. At this time, the frame itself is not shown. Evaluating the next expression produces a function call frame, shown in Figure 7-2. The frame that is produced is a function call frame. The name of the frame is given at the top, which is `sum-of-squares`, and the two arguments of the function call 3 and 4, are shown as simple frames inside the function call frame.

Now that we have parsed the expression into the frame, we evaluate the expression. First, we evaluate the arguments, which means evaluating the simple frames containing 3 and 4. We perform special animation for evaluating arguments. First
we move all arguments to the right side of the function call frame. Then, we animate the argument frames one by one. They are moved to the center of the function call frame, then moved to the top of the frame. During their movement to the center of the frame, the frame changes to indicate its evaluation. In this case, the frame simply changes color, since 3 and 4 are self evaluating expressions. Any arbitrary frame can be used as an argument, and the argument frame changes into a simple frame to indicate evaluation. The contents of the simple frame is the value of evaluated argument.

After we have evaluated the arguments, we perform more color changes to change the frame into the function definition shown in Figure 7-3. In addition to the color changes, we add more frames that appear within the function definition frame. In the top line, we see the name of the frame, (defun sum-of-squares indicating the function definition for the function sum-of-squares. The body of the definition contains two local variables a-squared and b-squared, which are created by the let statement, and a function call frame to denote the body of the definition, which corresponds to the expression (+ a-squared b-squared).
We now continue the evaluation by evaluating the local variables. We do this by evaluating the expressions associated with the local variables: (* a a) and (* b b). We employ a graphical substitution model by moving the parameter/argument frames (which have started in the upper right corner of the function definition frame), and moving them over the simple frames a and b. This animation is intended to show the bindings that need to be retrieved in order to evaluate the expression. We then evaluate the expressions with respect to those bindings, and create a simple frame to represent the value of the evaluated expression. Once the local variables have been evaluated, we then evaluate the body of the function definition by evaluating the frames that appear in the body. Once again, we use the graphical substitution model to represent the bindings we need to evaluate a particular expression. In this case, the local variables a-squared and b-squared are used, and their frames are moved over the simple frames a-squared and b-squared. Evaluating the function call frame using these bindings produces a result, which we show as a simple frame that appears in the bottom right corner of the function definition frame. This is final result of evaluating (sum-of-squares 3 4). In Figure 7-4, we show what the frame
7.2 Dealing with Lisp Animation

With the frame animation system, we have sought to create a general engine to show a graphical evaluation of code. This differs from most so-called algorithm animation systems in a crucial way. Algorithm animation systems require some type of "setting up". For example, you might need to place function calls to animation system within the algorithm itself in order to produce the needed animation. Or, in a more extreme case, you might have to express the algorithm itself more in terms of the animation system. These algorithm animation systems, while useful, do not seek to animate arbitrary pieces of code. This Lisp animation system seeks to fill that void. The goal is to be able to take any series of expressions, and produce an animation/visualization for those expressions.

Naturally, this is a fairly ambitious undertaking, and one that is not fully implemented here. In designing such an engine, there are a multitude of issues that
one must consider. For example, we handle data and bindings through a graphical substitution model. We do this because the substitution model is easily understood, and has a good graphics/animation analogy. We can represent bindings in a different way, say, in a graphical environment model. Another issue is how we represent procedures. Here we represent them by breaking a procedure into its component parts, such as name, parameters, local variables, body, and value returned. Such a representation works well for many languages, not just Lisp. We can choose different representations, depending on for instance, how the language being animated treats scoping of variables. Since we inherit the properties of frames, we inherit their issues also, namely, the best representation for a particular frame, or the best partition of expressions into frames. Yet another issue is the level of interactivity needed. At this point, the animation system produces a "movie", which you can not easily control. Certainly, the system would move forward if more interactivity were present.

In some sense, we would like to create a graphical language, which gives us an idea not only of the result of evaluation, but the process of evaluation. Often, this process is hidden from us, and often it is an error in our thinking of the process which causes bugs in a program. Thus lies the true potential of this kind of animation system, where a full implementation can reveal the act of computing, in addition to results. In essence, we have added a graphical component to the notion of tracing through a program. The graphical component becomes useful in helping us organize our trace; it is much easier to view a frame than to try to parse a large Lisp expression on the fly.
Chapter 8

Conclusions

In a thesis such as this one, there are often many lessons to be learned about the topic itself. This thesis is no exception. There are many insights that have been gained about software visualization itself, and the use of multiscale environments for large information spaces. Such insights are needed to guide future work into this area. In this chapter we will share some of these insights, and try to comment on the work we have presented in the previous chapters, particularly the ones on cross-referencing, the frame system, and Lisp animation.

8.1 Results

One of the more difficult aspects of computer-human interaction research is a formal testing of a system by potential users. Such testing may take quite a while, and is beyond the scope of this thesis. A few people have commented that while the Lisp animation system may be useful for learning Lisp, there are many issues that need to be resolved (some of which we talk about below). A higher level of interactivity was desired by all who saw the animation system. Many people found the cross-referencing system interesting, and I found it useful myself, especially when trying to understand some aspect of Pa3d. Many people also commented on the "depth"

Indeed, the idea of a multiscale environment has not yet been user-tested, although it is now becoming a priority for the developers of Pad++.
problem when viewing the frame system (i.e. one loses context when viewing deeply
nested frames). The frame system also helped me, although to a lesser extent than
the cross-referencing system, in trying to understand some of the code for Pa3d.

8.2 Unresolved Issues

There are quite a few interesting issues that remain unexplored in this realm. One
issue that has arisen over and over again through the development of this thesis is
the problem of parsing Lisp code. Syntactically, Lisp code is very easy to parse, which
enables one to think (and worry) about the semantics of the parsing. The question
arises: what is the ideal structure for the frames presented here? In this thesis we show
a structure that would work for many procedural languages, not just Lisp. One could
imagine, however, a parsing scheme that is much more directed towards Lisp (one that
views code more in terms of lambda expressions, for example). Would the structure
be language dependent? Which aspects of a procedure should be emphasized, and
which should be suppressed?

Another issue that arises is one of parsing depth, seen mainly in the animation of
Lisp code. Many real-life Lisp programs will nest s-expressions very deeply, thus pro-
ducing many nested frames. Deeply nested frames can cause a serious problem, which
is the loss of context. For example, in the static frame system, as we move deeper
and deeper into nested frames, we begin to lose our sense of location. It becomes
difficult to remember the procedure we are currently looking at if we are looking at
the 14th nested frame within that procedure. There should be some preservation of
context when viewing deeply nested frames. Deep nesting also causes problems in
the dynamic frame system, where we might not care to view the execution of some
nested frames. The solution in this case might be to make the system more interac-
tive, and allow the user to control the level of nesting. The loss of context problem is
more difficult, although Furnas' fisheye ideas described in Chapter 4 might provide a
solution.
8.3 The Future

The problem of software complexity is not one that will get better immediately. Programs are requiring more and more complexity, and some type of visual help is needed to manage this complexity. The popularity of such packages like "Visual C++" or "Visual Basic" reflect programmers desires for visual aids. The ideas reflected in this thesis propose an alternative direction in which to take software visualization. The idea of using a novel environment such as the multiscale environment should not be ignored, and can provide new and interesting views of code. We also would like to provide a visualization that can look at arbitrary pieces of code. Code visualization systems such as SEE, SeeSoft, and Lieberman's re-rooting system have the advantage of looking at a piece of code without any special preparation of the code. The ideas in this thesis were presented to explore these possibilities, and to point out new directions for software visualization. It is our hope that others also find them interesting, and give them the thorough exploration that they deserve.
Bibliography


