Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger

SOUGATA MUKHERJEA and JOHN T. STASKO
Georgia Institute of Technology

Much of the recent research in software visualization has been polarized toward two opposite domains. In one domain that we call data structure and program visualization, low-level canonical views of program structures are generated automatically. These types of views, which do not require programmer input or intervention, can be useful for testing and debugging software. Often, however, their generic, low-level views are not expressive enough to convey adequately how a program functions. In the second domain called algorithm animation, designers handcraft abstract, application-specific views that are useful for program understanding and teaching. Unfortunately, since algorithm animation development typically requires time-consuming design with a graphics package, it will not be used for debugging, where timeliness is a necessity. However, we speculate that the application-specific nature of algorithm animation views could be a valuable debugging aid for software developers as well, if only the views could be easy and rapid to create. We have developed a system called Lens that occupies a unique niche between the two domains discussed above and explores the capabilities that such a system may offer. Lens allows programmers to build rapidly (in minutes) algorithm animation-style program views without requiring any sophisticated graphics knowledge and without using textual coding. Lens also is integrated with a system debugger to promote iterative design and exploration.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Tools and Techniques—user interfaces; D.2.5 [Software Engineering]: Testing and Debugging—debugging aids; H.5.2 [Information Interfaces and Presentation]: User Interfaces; I.3.8 [Computer Graphics]: Applications; I.6.8 [Simulation and Modeling]: Types of Simulation—animation

General Terms: Algorithms, Human Factors, Verification

Additional Key Words and Phrases: Algorithm animation, debugging, programming environments, software visualization, user interfaces

This article is an expanded version of the report published in the Proceedings of the 15th International Conference on Software Engineering, May 1993.

This work was supported in part by the National Science Foundation under contract CCR-9109399.

Authors' address: Graphics, Visualization and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1994 ACM 1073-0516/94/0900-0215 S03.50

1. INTRODUCTION

People invariably have a difficult time understanding abstract concepts or processes. One way to improve understanding is to provide specific examples, possibly using pictures to make an abstract concept more concrete. The expression “Seeing is believing” relates that what we can visualize, we can grasp and understand.

This notion can be applied to software understanding and the use of graphics and visualization to depict computer algorithms and programs. Software visualization provides concrete representations to previously inanimate, abstract entities that have always been, and most likely always will be, relatively difficult to understand. The use of graphics for illustrating software was originally called program visualization [Baecker 1986; Myers 1990; Roman and Cox 1993], but more recently the term software visualization [Price et al. 1993; Stasko and Patterson 1992] has been favored. The term “software visualization” is better because it is more general, encompassing visualizations of data structures, code, algorithms, programs, and processes. Nevertheless, software visualization research has primarily focused on two different subtopics: program visualization (including data structure display) and algorithm animation.

Program visualization systems such as Incense [Myers 1983], GDBX [Baskerville 1985], and VIPS [Shimomura and Isoda 1991] illustrate particular data structures within a program, showing both the values and interconnections of particular data elements. These systems generate automatically a view of a data structure when the user issues a command to do so. Other systems such as Pecan [Reiss 1985] provide additional program state views of control flow, the run-time stack, and so on. Program visualization systems' main application has been for debugging and software development. Commercial systems such as Borland C for PCs, ObjectCenter for Centerline, Ensemble Viewer from Cadre, Lucid Energize, and CaseVision for Silicon Graphics workstations even provide rudimentary data structure display capabilities within a software development environment.

Algorithm animation, the second main subarea of software visualization, provides views of the fundamental operations of computer programs, concentrating more on abstractions of behavior than on a particular program's implementation [Brown 1988; Cox and Roman 1992]. The views presented are much more application specific than the generic views of data structure display systems. The movie Sorting Out Sorting^1 motivated many of the subsequent algorithm animation software systems that have been developed, including Balsa [Brown and Sedgewick 1985], Animus [Duisberg 1986], Movie [Bentley and Kernighan 1991], Tango [Stasko 1990], and Zeus [Brown 1991]. In all these systems, first a developer designs a visual presentation for an algorithm, and then implements the visualization using a support graphics platform. To date, the primary use of algorithm animations has been for teaching and instruction.

^1By Ronald M. Baecker and David Sherman, shown at SIGGRAPH '81, Dallas, Tex.

Because algorithm animation views are complex user-conceptualized depictions and may span an infinite number of different algorithms and depictions, it appears impossible to create them automatically from a “black-box” program illustrator [Brown 1988]. Rather, an animation designer crafts a particular view or views that are specifically tailored for a program. Consequently, designing algorithm animations is time intensive and usually restricted to already working programs. That is, a designer utilizes a fully functional working program, designs a set of animation routines for its visualization, and maps the program to the corresponding routines. The resulting views can then be used as instructional aids for illustrating the program’s methodologies and behaviors. Unfortunately, the time and effort required to develop animations is considerable enough to limit their use to pedagogy and preclude their use in software development and debugging. A programmer will not use a tool for debugging whose development time outweighs that to simply debug a program with traditional text-based methods.

This fact is unfortunate, however, because algorithm animation could offer key benefits to program debugging and testing. The use of pictures to illustrate how programs work has always played an important role in software engineering. Programmers, in designing code, often implicitly construct a mental model of their program, how it should function, how it utilizes data, and so on. It is much easier for people to think in terms of these higher-level abstractions (the big picture) than to try and comprehend how all of the individual operations and data work in conjunction—particularly so for larger systems. Data structure display systems, which have already been utilized in debuggers, can only offer views of raw data; the notion of an overall picture does not exist there. Algorithm animations, conversely, offer views of a program’s application domain and semantics. In essence, an algorithm animation is a visual manifestation of a programmer’s mental model of his or her program. This manifestation presents the program in the way that the programmer conceptualizes it. Consequently, these types of views can be critical for understanding what a program is doing and determining why it is not performing in the desired manner.

The use of animation is extremely important also because programs are fundamentally dynamic. Illustrating program data and states is useful for understanding, but illustrating how the program changes from state to state and evolves over time is even more helpful. Consider developing a computational geometry program, a quicksort, a particle chamber simulation, or a graph colorability algorithm. Would it not be extremely advantageous to have a dynamic visualization of the program to watch during program testing in order to see how it is working and to help identify erroneous program actions?

Particular systems have taken steps toward this merge of data structure display and algorithm animation. The data structure display system Incense [Myers 1983] allows developers to design their own abstract views of data structures, such as showing a clock face to represent integer values hours and minutes. This design, however, requires writing the low-level graphics code to implement the view.
The algorithm animation system Movie [Bentley and Kernighan 1991] focuses on rapid development of relatively straightforward algorithm visualizations. The system provides a few simple commands which can be used to quickly develop a visualization of a program in order to understand it better. Programmers still must learn the system's commands, however, and the system does not support smooth, continuous animation effects.

The Gestural system [Duisberg 1987] supports purely graphical animation development on Smalltalk programs; but its only images are black rectangles, and its only actions were movements following mouse-traced paths.

The Dance system [Stasko 1991] allows developers to build algorithm animations graphically, then it generates the corresponding Tango [Stasko 1990] animation code to carry out the specified actions. Unfortunately, programmers still must learn the underlying animation paradigm of Tango to develop views of algorithms or programs. Also, animation designs cannot be incrementally tested. The code must be generated, compiled, and run, but the design cannot be read back into the system for modifications.

The University of Washington Program Illustrator [Henry et al, 1990] requires truly no designer input to generate an algorithm animation. It analyzes program source code and produces its own abstract depiction. The system was only developed to analyze sorting and graph algorithms, however, using one particular style of view for each. Similarly, VCC [Baeza-Yates et al. 1992] synthesizes automatic views, but they are restricted to a few particular presentations or appearances.

Essentially, our work seeks to bridge the two domains of program visualization and algorithm animation as shown in Figure 1. We want a system that can provide application-specific animation views for debugging purposes. Unlike the UWPI system, we still want programmers to design their own animations, but we do not want to require the programmers to learn a graphics toolkit and write code using it, however. We also want our tool to work in conjunction with a debugger so that animation can be incrementally developed without going through the edit-compile-run cycle.

This work addresses the tail-end of the software development pipeline, when a designer has already written portions of, or perhaps all of, a target program or system. It is best suited for high-level debugging, program testing, and refinement, not low-level debugging typically focusing on correcting inadvertent lexical or syntactic misuses. Debugging has been characterized as the acquisition of clues to help programmers generate hypotheses on why a program is not working [Gould 1975; Katz and Anderson 1987]. Viewing program execution through dynamic visualizations is an interesting potential way to generate the clues which might be critical for correcting programs.

This article describes a methodology that allows rapid development of program animations, and a system called Lens that manifests the methodology. Lens supports application-specific semantic program views as seen in many algorithm animation systems, but it does not require graphics programming. Lens is integrated with a system debugger to support iterative testing and refinement. In the remainder of this article, we describe the
Fig. 1. This work bridges the different areas of program visualization and algorithm animation, seeking to gain the benefits of both.

conceptual model on which Lens is based; we illustrate how program animations are built with Lens; we describe some of the implementation challenges the system presents; and we discuss why many algorithm animation capabilities are so difficult to integrate into a tool such as Lens.

2. CONCEPTUAL FOUNDATION

For a tool to be used as a debugging aid by programmers, ideally it should not have a steep learning curve and should be fast and easy to use. This objective influenced strongly the goals of our research on this project. Rather than forcing developers to write animation description code, we sought to provide a palette of commonly used operations for direct invocation by programmers. Rather than forcing graphical objects to be specified by long lists of numerical parameters, we sought to provide a graphical editor for defining the appearance of program variables through drawing and direct manipulation. Most importantly, we wanted to keep the lists of these animation and graphical editing operations to a minimum of often-used directives, thus reducing the cognitive load on the programmer. Rather than building a comprehensive environment which could support any animation but which also included many rarely used directives, we sought to build a compact kernel of commands that could easily be learned and mastered.

In essence, this kernel of operations would be the conceptual foundation on which our system design was based. Choosing the operations for this kernel would be crucial, however, so it could not be based on speculation or guesses. Therefore, in order to develop this foundation, we studied 42 algorithm animations built with the XTango system [Stasko 1992]. The animations' topics included sorting, searching, graph, tree, string, and graphics algorithms, as well as animations of matrix multiplication, fft, hashing, producer-consumer problems, etc. These animations were built by over 25 different people from 4 different institutions, so they were not biased to a particular person's design methodology. Understanding what action commonly occurred in these animations would help us establish the basis for the visual debugging system being developed.

The first step in this analysis was to determine which types of graphical objects were commonly used in the animations, and also how the appearance of an object depended on the program it represented. Although XTango supports a wide variety of different graphical objects, only lines (15 times),

circles (11 times), rectangles (13 times), and text (17 times) commonly appeared. Other objects such as polygons or ellipses appeared 2 times at best.

When one of these graphical objects is created in an algorithm animation, its appearance depends usually on the state of the underlying program and the values of variables within the program. For instance, the position, size (e.g., length, width, height, radius) and label (for text) all may depend on program values.

For lines, we found that the start and end position of the line were its two attributes that often varied. The positions were either predetermined (no program dependence), dependent on the values of variables in the program, or relative to other existing animation objects. For instance, when a program had variables \( x \) and \( y \) whose values designated the position to use as an endpoint of a line, we called that “program variable dependence.” If a line was positioned so that its one endpoint was attached to the lower right corner of a rectangle in an animation, we called that “relative to another existing animation object.” If a line’s end position was always constant relative to the start position, then we designated it as “predetermined.”

For rectangles, the position and size (width and/or height) attributes varied across animations, with both commonly being either predetermined, program variable dependent, or relative to other graphical objects. Circles had similar attributes, where the circle’s radius was considered its size.

Finally, text objects varied along position and text string attributes. Text position was commonly either predetermined, specified by a program variable, or dependent on the location of some other graphical object. (Text is often used to label other objects.) Text strings were either predetermined or dependent on program variables.

In addition to individual graphical objects, many of the algorithm animations manipulated rows or columns of objects. These structures were commonly used to represent arrays in the underlying program. Frequently, it is useful in algorithm animations to create such composite structures via one common call. Often, the geometric layout or appearance of individual items depends on some other element or the group as a whole. In specifying a row of objects, designers would identify a bounding box inside of which the individual objects were placed. The number of objects was either predetermined or dependent on the value of a variable or expression. Often, one dimension of the objects, such as rectangles’ heights for a row, varied according to the values of the variables in the array the row represented. Other attributes that varied were the structure’s orientation (horizontal or vertical) and the spacing between individual objects.

Table I lists a summary of our findings about graphical object presentations along with their program-dependent attributes. Note that independent research examining animations with the Zeus and GraphVBT systems [DeTreville 1993] identified similar requirements: the need for vertices (rectangles and ellipses) with text, edges (lines), vertex highlighting, and polygons.

XTango animations also include the capability to designate particular positions in a display window. These positions (called “Locations” in XTango)
Table I. Summary of Graphical Objects Commonly Used in XTango Algorithm Animations and How Their Attributes are Specified. "Predetermined" means that the designer provided a numeric constant value.

<table>
<thead>
<tr>
<th>Object</th>
<th>Attribute</th>
<th>Specification(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>Start position</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative to another object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td></td>
<td>End position</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative to another object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Position</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative to another object</td>
</tr>
<tr>
<td></td>
<td>Width, height</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative to another object</td>
</tr>
<tr>
<td>Circle</td>
<td>Position</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative to another object</td>
</tr>
<tr>
<td></td>
<td>Radius</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative to another object</td>
</tr>
<tr>
<td>Text</td>
<td>Position</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative to another object</td>
</tr>
<tr>
<td></td>
<td>String</td>
<td>Predetermined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program variable dependent</td>
</tr>
<tr>
<td>Object array</td>
<td>Position</td>
<td>Predetermined (bounding box)</td>
</tr>
<tr>
<td>Number</td>
<td>Predetermined</td>
<td>Program variable/expression dependent</td>
</tr>
<tr>
<td>Size</td>
<td>Predetermined</td>
<td>Program variable/expression dependent</td>
</tr>
</tbody>
</table>

serve often as the destination points of object movements. We found that this feature was very commonly used, so we included it in the constituent set of capabilities for the new system. Additionally, many programs functioned within some sort of coordinate system, either integer or real coordinate based. Consequently, we felt the need to allow programmers to specify either type of coordinate system as the backdrop for an animation.

The second major step in identifying core algorithm animation features was to determine common actions or changes that objects underwent after they were created. In all the sample XTango animations examined, only six actions occurred more than a few times, and most of the animations only used some subset of the six actions. The actions were:

—Move an object to a particular absolute position, by a relative amount, or to a position relative to another object.
—Change the color of an object.
—Change the fill style (outline or filled) of an object.
—Make an object flash.
—Make two objects exchange positions (a special combination of movement actions).
—Delete an object or make it invisible.

The inclusion of the "exchange" operation was probably an artifact due to the sorting algorithms included in our sample of the 42 algorithm animations. Nevertheless, we felt that the operation would be useful in other types of programs, so we decided to keep it in the core group of operations.

Note that by omitting the lesser-used graphical objects, attributes, and animation actions, we were restricting the set of subsequent visualizations that could be generated. Our approach here, however, was to begin with an absolute minimal set of constituents in order to make them as easy to learn as possible. Subsequent trials and usage could then further identify desired constituents and capabilities.

After completing this research on existing animations, we organized the sets of graphical objects and common animation actions into a kernel of capabilities to be used as the basis for the graphical debugging system we would develop. Our intention was to allow designers to instantiate objects graphically through direct manipulation and the use of a graphical editor, and to specify animation actions through a set of menu-based commands where each choice corresponded to one of the core actions we identified.

3. INTERACTING WITH THE LENS SYSTEM

Once the set of core operations had been established, we began development of a visual debugging systems that would embody these operations. The system is implemented on top of UNIX, the X Window System, the XTango algorithm animation system, and the debugger dbx. Its target language is C. All the capabilities and operations described in this article have been implemented and are functional.

In this section, to provide a better "feel" for how the system works, we describe how programmers interact with Lens. To begin a visual debugging session, a programmer issues the command,

```
% lens foo
```

where `foo` is the name of an executable program that has been compiled with the appropriate debugging flags. The entire Lens display window appears as shown in Figure 2. After startup, the user selects an initial source file to view via a selection within the "File" menu (it may be any source file such as `foo1.c` or `foo2.c` used to build the binary `foo`), then Lens loads and displays this source and awaits the entry of animation commands. (Other source files can be subsequently loaded as well.) In the Lens window, the left area presents program source code; the upper right area is the graphical editor for designing objects' appearances; and the lower right area is for issuing debugger commands.

Toward Visual Debugging

At this point the programmer can issue normal debugger commands or designate the special animation commands at particular points in the source code. To do this, the programmer chooses commands from the "Animation" menu above the source code. It has nine options that correspond to the kernel of algorithm animation constituents found in our study of algorithm animations:

(1) Define coordinate system
(2) Create graphical object
(3) Create location marker
(4) Move
(5) Fill
(6) Color
(7) Flash
(8) Exchange
(9) Delete.

Fig. 2. Lens display presented to a programmer. The left section shows source with animation annotations, and the right section contains the graphical editor and debugger command window. In the upper left the File menu contains commands for loading and saving source and annotations; the Animation menu contains the nine animation operations; the Template menu contains commands for building data structure views such as lists and trees; the Debug menu supports running and stopping program execution; the Edit menu supports clearing all animation annotations.
When either the “Create graphics” or “Create location” command is chosen, the programmer is prompted to enter the variable name of the object or location marker being depicted. The variable name is subsequently used to identify the object for the action commands. This is a key advantage of Lens: a programmer works in the context of familiar program entities, not graphics primitives. After a name has been entered, Lens asks the programmer to use the graphical editor to design (draw) the object or location’s appearance and/or position. The design is structured according to the object specifications that were discovered in our earlier research on existing animations. For example, when a line is created: its position can be specified using the mouse; it can be specified relative to another object (by name or by picking a graphical object); or it can be specified to relate to the value of a program variable. All these choices are made via a dialog box entry or by graphical direct manipulation. Once the design of the object's appearance is complete, Lens asks the user to designate, using the mouse, the source code line at which object or location creation should occur. Lens indicates the presence of animation commands by showing an “a” beside the code line. The presence of multiple commands per line is designated by “A.”

When a programmer chooses one of the six action commands, Lens asks for the name of the variable(s), and consequently graphical object(s), to which the action should apply. The programmer simply types in a variable name such as x or a[i].² Lens notes the choice and saves the association in an internal database. Finally, the programmer selects the source code line on which to place the command. Programmers also can examine existing animation annotations simply by clicking on a source code line containing them. Lens pops up a dialog box or boxes summarizing the animation command(s) that has been designated there. Every animation annotation also can be deleted, thus supporting trial and experimentation. Figure 3 shows both a question dialog used for specifying a “Move” action, and the summary dialog shown when the same Move annotation is queried later.

When the programmer wishes to execute the program and see its animation, he or she chooses the Run command from the “Debug” menu or simply types “run” in the debugger command shell. Lens then pops up an animation window and displays the animation that corresponds to the programmer's design and this particular execution. Lens uses the routines from the XTango system to generate the animations it presents.

If the animation is not sufficient or not what the programmer wanted, the programmer can go back, add or delete animation commands, and rerun the animation, all without leaving the Lens environment. Full dbx capabilities are supported within the lower right command window also, so the traditional power of the debugger can also be utilized.

To add extra debugging support, Lens provides “templates” for common data structure visualizations. A template is a predefined graphical data

²The system currently supports single variables, pointer variables, or an array variable with an expression as its index, but not arbitrary C code.

structure depiction, much like those shown in Incense [Myers 1983] or VIPS [Isoda et al. 1987; Shimomura and Isoda 1991]. Lens provides templates for scalars, arrays, linked lists, and binary trees. To instantiate a template, take the linked list as an example. The programmer merely specifies the pointer into the list, the name of the field in the structure used as the "next" pointer, and some "value" field of the nodes to present graphically. Through its communication with dbx, Lens can traverse the list and render an image of the list's state at that moment. A sample list depiction is shown in Figure 4.

Additionally, Lens supports saving animation annotations defined during a session, so that program debugging can be resumed at a later time using the same animation context.
4. BUILDING SAMPLE ANIMATIONS

Using Lens, it is straightforward to build animated presentations of programs in minutes. In this section we provide some examples of how the system can be used in this manner. Unfortunately, the static nature of a paper makes it difficult to portray adequately the dynamics of Lens and its ease of use. Here, we will attempt to convey as best we can through textual descriptions and figures how one interacts with the system.

The first example is the definition of an animation for a first-fit binpacking program. How to build this animation using the textual Tango animation library and toolkit is discussed in detail in Stasko [1990, pp. 35–38]. It requires roughly 75 lines of code. Below we generate a similar animation, a frame of which is shown in Figure 5, using 7 animation actions in Lens.

The source code for the binpacking program, along with the positions for animation annotations, is shown in Figure 6. Note that the last three annotations are all on the same line of code.

The animation was created via the steps below. Each one corresponds to a different animation annotation, and they are given in the order they appear in the program source. Figures 7–13 show all the dialog box entries that were needed for each animation command. Not shown are operations in the graphical editor.
#include <stdio.h>

main()
{
  int n, b, i, wtnum;
  double wt, bin[100];

  printf("How many bins?\n");
  scanf("%d", &n);
  for (i=0; i<n; ++i)
    bin[i] = 0.0;

  wtnum = 0;
  printf("Enter the weights (range 0.0->1.0, with 0.0 to quit)\n");
  for (;;)
  {
    scanf("%lf", &wt);
    if (wt == 0.0) break;
    b = 0;
    while (bin[b] + wt > 1.0)
      b++;
    bin[b] += wt;
    wtnum++;
  }
}

Fig. 6. Program source and animation annotation positions for the binpacking program.

— At program startup, a rectangle called box (corresponds to no program variable) that represents the container is created. The user draws out the rectangle in the graphical editor as it should appear in the animation window. Its position, width, and height are all predetermined and independent of any program variables. The dialog boxes used in this command are shown in Figure 7.

— A row of location objects called bin is created to lie at the bottom of the box. The user draws a line in the graphics editor to hold them. These locations are used to represent the positions for individual bins within the container. That is, they designate where the weights should fit into the surrounding container. The number of elements in this array is dependent on the variable n in the program, which corresponds to the number of bins, as shown in Figure 8.

— When a new weight is entered, a rectangle corresponding to it (the variable wt) is created. The lower left corner of the rectangle will always appear at the same position, but specifying the height and width of the rectangle is trickier. We want the height of the rectangle to correspond to the value of the variable wt, which ranges from 0.0 to 1.0. To do this, we specify these minimum and maximum values the variable can take on, and we draw the rectangle in the graphics editor at its maximum value which is simply the same height as the box in the first annotation (minimum height defaults to
being a flat rectangle). We want the width of the rectangle to be equal to the width of the individual bins within the container box because we want the weights to fit snugly. Hence, in the Lens dialog box for specifying the width, we enter the value $\text{box.width}/n$. Here, $\$ \text{ indicates that the trailing identifier name refers to a graphic object already created, and not to a program variable. So we are specifying that the width of the weight should be the same as the width of the box, divided by the value of the program variable $n$ which is the number of bins. The dialog boxes used in this command are shown in Figure 9.

When an attempt to fit a weight into a particular bin is made, we want to represent this activity by moving the weight smoothly over to the corresponding bin. Hence, we specify a Move animation action: the graphical object corresponding to wt is moved, and its bottom left position ends up at the location corresponding to bin [b]. See Figure 10 for dialog entries.

When a weight has successfully been placed into a bin, we need to update the bin entry position to the top left of the weight, so that the next weight (if any) is placed there on top of this current weight. To move the bin position to the top of the weight, another Move action is specified. The
location object bin [b] is moved to the upper left corner of wt. This specification is shown in Figure 11.

—To signify the weight is in place, we change it from outline to filled as indicated in Figure 12.

—To signify further the weight is in place and to differentiate the different weights which are all solid red, we draw a black rectangle outline around the weight. The three dialog boxes used in this command are shown in Figure 13.

These seven animation actions are specified at the four lines with an “a” and the one line with an “A” in the binpacking source of Figure 6. Once the annotations are in place, the program can be run, and the corresponding animation will be generated.

The next example is a program that does a depth-first search on a graph. Suppose we wish to visualize this algorithm in the traditional vertex-edge manner with circles for vertices and lines as edges. Also, the vertices should initially be red; then they should change to green as they are visited, and a numeric label should be added indicating the depth-first ordering. Figure 14 shows the source code for the program and the placement of four animation commands for building this animation view.

To position the vertices of the graph at the appropriate locations, we added a Create Graphics command to the program. In Figure 14, it is the first animation annotation in the file. We simply specify that a red circle should be drawn to represent variable vertex[i], and that the x and y position of the circle is dependent on the variables loc[i][0] and loc[i][1]. Note that the program has no array vertex, however. In examining the existing algorithm animations, we noted that designers created graphical objects often which
had no direct correspondence in the program. Consequently, we included this capability in Lens. Programmers can create graphic objects with no name, or they can make up a name if that name will need to be used in subsequent action operations.

Edges are drawn in by the second annotation, another Create Graphics command. Here, a line is drawn with its two endpoints determined by the positions of the graphical objects representing the vertices at the ends of the edge.

Recall that the vertices are red when they are initially created. In the animation, we would like them to turn green once they are visited in the depth-first search. Therefore, in the visit subroutine we added a Change Color animation action. We also would like to number the nodes according to the order they are visited. Therefore, we added a second animation command in this subroutine, Create Graphics command, was used to create a text object. In specifying the graphical object, we designated that the contents of the displayed text string should be the value of the variable now, and the string should be placed in the northwest corner of the corresponding vertex.

Figure 15 illustrates the view of this animation specification in Lens, and it shows the resulting animation window after a particular execution. Note that we created a fairly sophisticated animation of the program, one that matches
#include <stdio.h>

void visit();
int n,now,val[50],adj[50][50];
double loc[50][2];

main()
{
    int i,j;

    printf("How many vertices?\n");
    scanf("%d", &n);

    printf("Enter the locations\n");
    for (i=0; i<n; ++i)
    {
        scanf("%lf %lf\", &loc[i][0], &loc[i][1]);
        printf("Vertex %d: Locations %lf %lf\n", i, loc[i][0], loc[i][1]);
    }

    printf("Enter the adjacency matrix\n");
    for (i=0; i<n; ++i)
        for (j=0; j<n; ++j)
        {
            scanf("%d", &adj[i][j]);
            if ((i > j) && (adj[i][j]))
                printf("Edge between vertex %d and vertex %d\n", i, j);
        }

    now = 0;
    for (j=0; j<n; ++j)
        val[j] = 0;

    for (j=0; j<n; ++j)
        if (val[j] == 0)
            visit(j);
}

void visit(k)
{
    int t;

    now += 1;
    val[k] = now;
    for (t=0; t<n; ++t)
        if ((adj[k][t] != 0) && (val[t] == 0))
            visit(t);
}

Fig. 14. Source code along with animation annotation positions for the depth-first search program.
Fig. 15. View of Lens for depth-first search animation.
the programmer’s mental model of the computation, and it was all done with four simple Lens animation annotations.

Finally, let us illustrate how Lens can be used to track down bugs in a program. The problematic program is a particle chamber simulation program shown in Figure 16.

When we ran this program and generated some output from it, we noticed that it was not behaving correctly. Consequently, we loaded it into Lens and defined an animation for it that matched our mental model of the program. A frame from the resulting animation is shown in Figure 17.

This animation used three simple commands. The first established a coordinate system ranging from 0 to 100 in both x and y. The second creates a circle corresponding to each particle, where the circle’s initial position is specified by the variables x[i] and y[i]. The third command merely moves each particle to a new position designated by the updated values of x[i] and y[i].

On the first run of the program with the animation, we noticed that all the particles started in the center, but then they immediately jumped to the bottom of the view. That is, it appeared that their y position was always set to the y value at the bottom. This assignment is made in the last of the four if boundary condition checks in the program, and it always appeared to happen. Upon closer examination, we realized that the condition

\[
\text{if } (y[i] - \text{SIZE} > \text{by})
\]

is always true, because the operator should be "<" to check if a particle has gone below the bottom.

After correcting the problem, recompiling, reloading, and rerunning the animation,\(^3\) we noticed that all the particles moved around the chamber in a reasonable fashion at first, but when they struck the right wall, they seemed to stick there and just move up and down. This caused us to check the conditions for detecting particle movement past the right edge. There, we found that the calculation of a new velocity

\[
vx[i] = \text{random( ) \% 8} + 1
\]

did not change the sign, so that the particles would not move back to the left as they would from

\[
vx[i] = -(\text{random( ) \% 8}) - 1.
\]

Upon making this fix, the animation worked correctly and gave us confidence that the program was working as intended.

Note that it would have been possible to find these bugs simply by inspection, by using a traditional textual debugger, or by adding “print” statements to the program. Using Lens, however, we were able to create very rapidly a visual depiction for the program that illuminated its behavior in a clear manner that made its functionality obvious to us. We were able to receive feedback about the program’s operations in a way that matched our conceptualization of the program itself.

\(^3\)We speculate that Lens might be even more helpful in an integrated programming environment which helps automate these steps, or even in an interpreted environment.
#include <stdio.h>

#define BALLS 50
#define SIZE 3

main()
{
    int x[BALLS], y[BALLS], vx[BALLS], vy[BALLS];
    int lx, by, rx, ty;
    int i, clock=0, numballs, bounced;

    lx = by = 0;
    rx = ty = 100;
    printf("Number of balls (up to 50)\n");
    scanf("%d", &numballs);
    for (i = 0; i < numballs; ++i) {
        x[i] = (lx+rx)/2;
        y[i] = (ty+by)/2;
        vx[i] = rand()%20-10;
        if (vx[i] == 0) vx[i] = 1;
        vy[i] = rand()%20-10;
        if (vy[i] == 0) vy[i] = 1;
    }

    for (;;) {
        clock++;
        for (i = 0; i < numballs; ++i) {
            x[i] += vx[i];
            y[i] += vy[i];
            bounced = 0;
            if (x[i]-SIZE < lx) {
                x[i] = SIZE;
                vx[i] = random() % 8 + 1;
                bounced = 1;
            }
            if (x[i]+SIZE > rx) {
                x[i] = rx-SIZE;
                vx[i] = random() % 8 + 1;
                bounced = 1;
            }
            if (y[i]+SIZE > ty) {
                y[i] = ty-SIZE;
                vy[i] = -(random() % 8) - 1;
                bounced = 1;
            }
            if (y[i]-SIZE > by) {
                y[i] = SIZE;
                vy[i] = random() % 8 + 1;
                bounced = 1;
            }
            if (bounced)
                printf("Particle %d bounces at (%d,%d)\n", i, x[i], y[i]);
        }
    }
}

5. SYSTEM IMPLEMENTATION

The development of the Lens system presented a number of challenges. A few of the most interesting are highlighted in this section.

5.1 Integration with a Debugger

To acquire information about the program that is being run, Lens establishes a connection with the dbx debugger. The interface with dbx is similar to the approach used by the program xdbx, a widely available X Window System facility and graphical interface to dbx. Lens communicates with dbx through a pseudoterminal (pty) that is a pair of master and slave devices. The pty is opened for both reading and writing. After a child process is created via fork, the child process closes the master side of the pty, redirects stdin, stdout, and stderr of dbx to the pty, unbuffers output data from dbx, and performs and exec dbx. The parent process closes the slave side of the pty, sets the dbx file pointer to nonblocking mode, opens the file pointer for read/write access to dbx, sets line buffered mode, and then monitors the output from and passes input to dbx. The overall structure of the Lens system is shown in Figure 18.

When the user instructs Lens to create an animation action at a particular line, the parent process sends a stop at command to dbx. Later, when the program is executing and dbx stops at that line, the parent executes the
animation action that was specified at the line. The parent also may acquire
other information from dbx; for example, if the value of a variable is required,
a print command is passed to dbx. If the type of a variable is required, a
what is command is passed to dbx. If dbx passes an output, the parent
processes it and takes the appropriate action. For example, if both the
program that is being debugged has an error and dbx sends an error
message, the parent will display the error message for the user and halt the
execution of the program. If dbx sends an output which the parent does not
recognize, the parent assumes that the output is from the program itself and
outputs it for the user to see. Lens must recognize common output strings
from dbx, many of which arise from error conditions, so that it can recover
gracefully from any number of possible scenarios that arise.

Typically, the program being debugged will require some input from the
user via standard input. In Lens, dbx (and consequently the underlying
program) receives all its input from the parent process which was established
at system startup, and is shown in Figure 18. Since we cannot possibly know
when input is required, the parent must set its input to nonblocking mode
and constantly poll the external input buffer. If there is ever any user input,
Lens simply passes it along to dbx.

Note that all communication, both input and output, with dbx must be
carried on through textual strings. This forces Lens to do extensive parsing in
the background to facilitate this communication. Interacting with a debugger
that has a programmatic, command-based interface would greatly simplify
this task.

5.2 Specifying the Animation Actions
In order to make the specification of animation actions as easy as possible for
the programmer, Lens must perform some subtle internal manipulations. For
example, it would be convenient for the system to be able to work in terms of
variables and graphical object pointers, but that would not be best for the
programmer. Rather, when a programmer types in the target object for a
command such as Color, the programmer simply enters at the dialog prompt
a text string such as b1 for the variable name. That is, Lens interacts with
the programmer at a string level, using names as identifiers and references. Lens resolves variable name entries within an internal database of graphical objects and locations that already have been created. Lens also must alert the programmer to syntactic errors made at this level.

If an object's attribute, such as the radius of a circle, is dependent on a program variable, the system asks the user to specify its maximum and minimum value in order for Lens to scale the object appropriately during the actual animation. Additionally, when the user chooses a variable, the system checks the type of that variable and rejects the animation action if the variable is not of the appropriate type. For example if a position is dependent on a variable, the variable must be an integer, double, short, long, or float.

Lens also must be flexible in its interpretation of animation actions. If a programmer chooses a Move command, he or she can specify the name of another program entity which should be the destination of the movement action. This entity can be either a location or a graphical object, however. So, Lens must determine which type of object it is. If it is a location, Lens simply moves the object to that position. But if it is another graphical object, Lens interprets the action so that the original object is moved to the current position of the target object.

The Exchange animation requires the user to specify the variables for the two graphical objects (they cannot be locations) that are to be exchanged. The Flash, Change Fill, Change Color, and Delete animation actions simply require the user to specify the one variable for the graphical object on which the action is to take place. The Change Color animation also requires the user to select a color from a palette of colors that are displayed. All these animation actions may be applied to a graphical object that is a scalar variable or is an array element. For example, the user may specify that elements a[i] and a[i + 1] should be exchanged.

5.3 Executing the Animations

After the programmer has built all the animations and wants to run the program, Lens must dispatch a run command to dbx. Before doing that, however, Lens traverses the list of animations and sends a stop at command to dbx for each animation at the appropriate line of source code. One unique feature of dbx is its assignment of line numbers to the logical file that are different from the actual line numbers of the source text file. This is due to blank lines and statements stretching over more than one line. Therefore, if we pass a stop at n command to dbx, the line number that dbx actually stops at may be different from n in the original source file. Fortunately, dbx returns the line number where it will stop given a particular stop at request. Lens uses this value and stores it for use in subsequent actions.

When dbx stops at a particular break point and sends a message to Lens, Lens scans its list of animation commands to find the particular command(s) that reside on the line that caused the break point. If none is found, Lens assumes the message is from a true dbx breakpoint. If Lens does find an animation command(s) for that line, it executes the animation action(s) specified there. To do this, Lens may need to send other messages to dbx, as
well. For example, if the command is to create a rectangle that will represent a variable \( i \), and the value of \( i \) should control the width of the rectangle, then Lens must acquire the variable's current value. If the variable being examined is \( k(m + 10) \), Lens must parse the variable name string and determine that the value \( m + 10 \) must be acquired first.

As you may have noticed above, Lens must carry out significant error checking to prevent harmful situations from occurring. For example, when a program executes, it may initially encounter an animation annotation that states to move the object corresponding to variable \( j \). But if the annotation that creates the appearance of the object corresponding to \( j \) has not yet been reached (certainly possible in a buggy program) then Lens must catch this situation and alert the programmer. Similar situations occur from any number of spelling, syntactic, and conceptual errors made by programmers in defining their program's animation.

6. CONTINUING CHALLENGES

Clearly, a system like Lens would not be needed if we were able to generate meaningful, useful algorithm animations automatically through some black-box approach. The Lens system finesses this problem by providing easy methods of specifying animations rather than attempting to do it automatically. However, we still have found limits to the sophistication of the animations that can be created using the direct-manipulation and dialogue-filling techniques provided in Lens. This is because some aspects of algorithm animation are inherently difficult and simply are better facilitated by having a graphical toolkit or library available for programming support, and sometimes a desired feature is simply outside the current capabilities of Lens. (Recall that we intentionally kept the kernel of features provided by Lens to a minimum.) In this section we describe some of the challenges that we have encountered using Lens.

6.1 Conditional and Repeated Executions of Animations

In some cases a programmer may want to specify that an animation action should occur only if a certain condition is true in the program being examined. In other instances, an animation action may need to be repeated a certain number of times, perhaps as denoted by a program variable. If the corresponding condition or loop exists in the program being animated, the user can easily specify that the animation be generated within the scope of that construct. However, in some cases there may not be a corresponding condition or loop in the program. For traditional textual algorithm animation this is not a problem, because the condition or loop can be shifted and programmed into the animation routines. But this is not possible under Lens, which intentionally does not support programming at the animation level. For example, reconsider the depth-first search program of Figure 14. This program uses an adjacency matrix representation to store the graph. The animated presentation of the program draws in edges between vertices only if the corresponding adjacency matrix entry is a "1." This is possible because the program has the pertinent if conditional. But if no if statement
existed in the program to check this condition, we would not be able to generate the edges because Lens cannot evaluate the condition on its own. In cases like this, additional programming support simply is required and, hence, is beyond the scope of Lens as currently defined.

6.2 Specifying the Layout of the Graphical Elements

In many sophisticated program animations, a complicated layout of graphical elements is necessary to make the presentation match the programmer’s mental model of the program. To facilitate such layouts Lens allows the user to specify simple formulae as values for image attributes. For example, the height of a rectangle presentation for the variable $x$ can be equal to the width of the presentation for the variable $y$ divided by the value of the program variable $n$. In Lens this is specified in a dialog box entry as $(y.width) / n$. (Recall that the $\$\$ tells Lens to use an attribute of the presentation of a variable, rather than the variable’s value in the program.) This is simply a very limited form of textual programming, and it clearly makes the use of Lens more complicated. Moreover, for more involved animations, there may be even more complex relationships involving 2 or more graphical elements. For these, programming is extremely advantageous.

Actually producing the graphical elements themselves also may be exceptionally difficult. For example, to draw an aesthetically pleasing tree using only simple direct manipulation and dialogue fill-in is quite challenging. Lens tries to reduce this particular problem by providing templates for common data structure presentations. However, it is not possible to provide all the graphics that the programmer may need. For instance, a computational geometry program may require a different kind of tree structure. For producing these types of graphics, the capability of programmatic control is again advantageous.

6.3 Depicting the Visual Representation of Program Variables

In a program animation, graphical elements are used to depict program variables, and different algorithmic actions are depicted as manipulations of the graphical elements. For programmingless creation of the animations, we should be able to specify the mapping from the program variables to their visual representations easily. However, this may not be possible for all animations. For example, consider an implementation of the *Towers of Hanoi* program as shown below:

```c
int peg[3];
main( )
{
    int n;
    printf("How many disks?\n");
    scanf("%d", & n);
    peg[0] = n;
    hanoi(n,0,2);
}
```

void hanoi(n, from, to)
    int n, from, to;
{
    int spare;
    if (n == 1)
        move(from, to);
    else
    { spare = (to != 0 && from != 0) ? 0:
        (to != 1 && from != 1) ? 1 : 2;
        hanoi(n – 1, from, spare);
        move(from, to);
        hanoi(n – 1, spare, to);
    }
}
void move(from, to)
    int from, to;
{
    peg[from] – –;
    peg[to] + +;
}

A “traditional” visualization of this program, one that would match most people’s mental model of the problem, would show three pegs with graduated size disks being interchanged among them. In the program the pegs are represented as an integer array of 3 elements (peg[3]), and the number of disks on the pegs are represented by the values stored in the 3 array elements. As these values are altered by the move procedure, disks should be interchanged among the pegs in the animation. However, specifying this mapping from the array peg to its visual representation through direct manipulation and declarative specifications is exceptionally difficult and currently beyond the scope of Lens. Such difficulties are a big impediment to the easy creation of animations. If there is no easy way of mapping programming entities to graphical entities via direct manipulation, programming efforts may be required to create the animations.

6.4 Specifying the Location of Animations in the Program

In Lens, to specify where an animation should execute during the running of the program, the user selects with the mouse the line at which the animation should execute. Lens calls dbx to set a break point at that line. Note, however, that in dbx, breakpoints are encountered before the command for the line of code that is executed. That is, the animation will be performed before the debugger executes the program statement on that line.

Consider a modified version of the depth-first search program in Figure 14, in which the user is entering the locations of the graph nodes:

\[
\text{for } (i = 0; i < n; i++)
    \text{scanf("%lf %lf", &loc[i][0], &loc[i][1]);}
\]

During the animation, the graph nodes should be drawn at the locations specified as these locations are read in. However, the break point cannot be specified at the scanf statement since the pertinent location has not been read in yet. It cannot be specified after the for loop since inclusion inside the loop is required to draw all the graph vertices. In general, Lens is not able to deal with cases where the user requires to execute animation actions after a statement in a one-line loop body. The usual solution is to add a following "do-nothing" statement such as a printf to the loop body, and then put the animation annotation on that line.

This particular problem is symptomatic of a more general problem: dependence on the system debugger used. In fact, we found that different versions of dbx on our local system operated differently. One particular "bug" within Lens had us stumped because it worked properly for one of us, and improperly for the other, even though we were using the exact same code. After a day or two, we discovered that different versions of dbx were being invoked (we had slightly different shell command paths), and they operated in subtly but importantly different fashions.

Note that this is a different type of problem than the earlier ones in this section. This problem is more particular to the host environment of Lens. Nonetheless, it is an important problem that restricts what we would like to provide in such a system.

7. CONCLUSION AND FUTURE PLANS

This article has described the development and functionality of the Lens system. Lens integrates algorithm animation-style capabilities into a traditional source-level debugger. The primary focus of the system has been to provide these capabilities without requiring a programmer to learn a graphics paradigm and without requiring the programmer to write more code. Although Lens works with the dbx debugger on C programs, there is no reason why other debuggers and programming languages could not be supported as well.

In addition to the potential applications for debugging, we have found Lens to be a good tool for rapidly prototyping traditional algorithm animations. Since no textual coding is necessary, animation development is easier and faster than with other current algorithm animation systems. Nevertheless, as described in Section 6, this ease of development is not without a cost. Lens cannot currently support the same breadth of animation operations as systems based on graphics coding. It works best on smaller programs that are classical computer science algorithms, those often seen as the examples in algorithm animation systems. Extending this window of coverage while further simplifying animation design is our primary future research goal. Two other future directions include:

—Improving the somewhat "clunky" user interface for specifying object appearances and binding these representations to program entities. Currently, much of this interaction occurs through dialog boxes. A better design would utilize more of a direct-manipulation approach. Two possibili-
ties for improvement are: choosing program variables by selecting them with the mouse from the program text and specifying graphical “connection links” between graphical attributes (height, size, position) and program variables.

—Performing empirical tests to examine the usability of Lens’ interface, and most importantly, to understand better how Lens can be utilized in program debugging. Currently, the system is a research prototype that explores the capabilities that might be offered in this kind of environment. Assessing the true utility of such a system and, more accurately, identifying the requisite features it should support remain.

In addition to the directions above, a system such as Lens would require further pure development work to make it more robust. Debugging systems are extremely complex software entities that must be as bug free and optimized as possible. We have made Lens available via anonymous ftp, and we hope that feedback from its use and modification, along with future empirical studies helps us to better understand the possible role of visualization and animation in program development and debugging.

ACKNOWLEDGMENTS
Scott Hudson read an earlier draft of this article and helped improve its content.

REFERENCES


*Lens may be acquired from the machine ftp.cc.gatech.edu as the file pub/people/stasko/lens.tar.z. The system runs on UNIX workstations and requires the X Window System and the motif widget set. Full source with installation instructions is provided in the distribution, and documentation is included also.


Received January 1994; revised May 1994; accepted Month 1994
Editor: Brady Myers