Software Visualization in the Large

Thomas Ball
Stephen G. Eick
Bell Laboratories

The invisible nature of software hides system complexity, particularly for large team-oriented projects. The authors have evolved four innovative visual representations of code to help solve this problem.

It is well known that large computer programs are complex and difficult to maintain. Production-sized systems, particularly legacy software, can contain millions of lines of code. Even a seemingly simple, small-team project, such as a spreadsheet, is quite complicated.[1] Understanding, changing, and repairing code in large systems is especially time-consuming and costly.

Knowledge of code decays as the software ages and the original programmers and design team move on to new assignments. The design documents are also usually out of date, leaving the code as the only guide to system behavior. It is tedious to reconstruct complex system behavior by analyzing code.

Perhaps the most difficult software engineering projects involve "programming in the large." These large-team projects, often in maintenance mode, require enhancements involving subtle changes to complex legacy code written over many years. Under these circumstances, programmer productivity is low, changes are more likely to introduce errors, and software projects are often late.

Software visualization can help software engineers cope with this complexity while increasing programmer productivity. Software is intangible, having no physical shape or size. After it is written, code "disappears" into files kept on disks. Software visualization tools use graphical techniques to make software visible by displaying programs, program artifacts, and program behavior. The essential idea is that visual representations can help make understanding software easier. Pictures of the software can help slow knowledge decay by helping project members remember—and new members discover—how the code works.

Three basic properties of software can be visualized:

- *Software structure*. Directed graphs, the foundation of many CASE and program analysis tools, are perhaps the most common method for showing the relationships among software entities. For example, a node can represent a procedure, and an edge can represent a calling relationship between two procedures.[2]
- *Runtime behavior*. Algorithm animation uses graphical representations of data structures and motion to illustrate the higher level behavior of algorithms.[3] Lower level views based on program profiles or traces can reveal bugs and performance anomalies.[4]
- *The code itself*. Pretty printers, which usually indent the code and use special fonts or colors to distinguish keywords and so forth, are a basic, widely used form of visualization.[5]

Previous approaches to software visualization, although useful for small projects, do not scale to the production-sized systems currently being manufactured. The graphical techniques found in programming, program-visualization, and algorithm-animation environments target small systems. Algorithm visualizations are usually hand-crafted and require the designer to understand the code before visualizing it, making this technique infeasible for large systems or tasks involving programmer discovery. The

0018-9162/96/$5.00 © 1996 IEEE

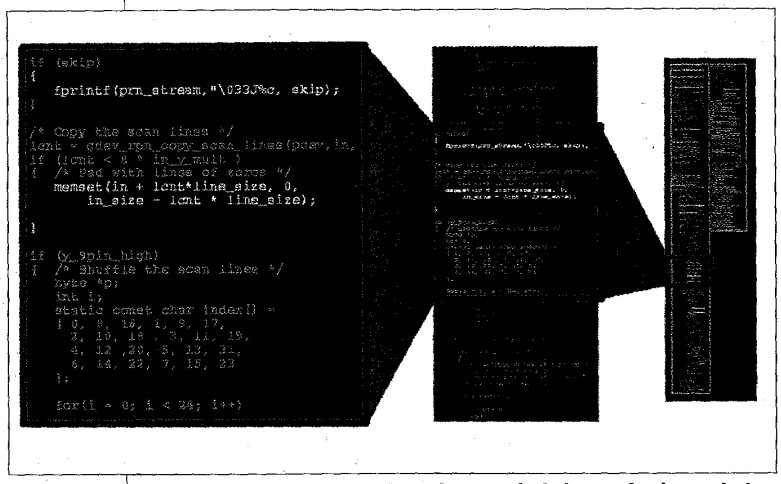


Figure 1. Line representation. Three scaled views of color-coded program text. The left pane shows readable text. The middle pane provides a longer view. The right pane reduces each line of text to a single row of pixels with row length and indentation tracking the original code. The entire file is visible in the line representation, spanning one-and-one-half columns. The color-coded lines show code age; green represents the old code and red the new, with yellow code spanning the age difference.

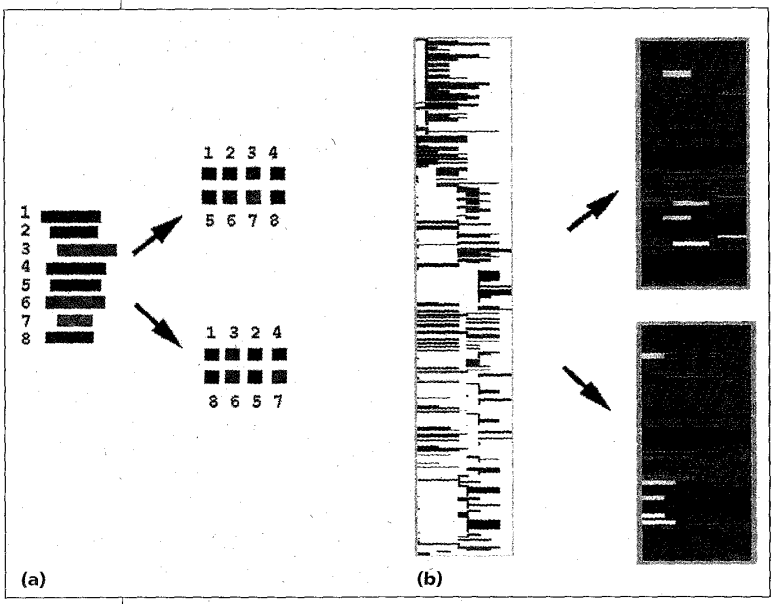


Figure 2. Line and pixel representation comparisons. The left columns in (a) and (b) show the line representation; the right columns depict two possible pixel representations in which each line of code corresponds to one (or a small number of) pixel(s) ordered left to right in rows within a column. In the upper pixel representation, the pixels within a graphical line respect the order of their corresponding lines in the text. In the lower one, the pixels within a line are ordered by their color in the rainbow spectrum, which provides a mapping from statistics (such as code age) to colors.

general strategy for large projects is to decompose the project into modules, usually hierarchically, and display each module individually. In practice, this decomposition is often the most difficult aspect of the visualization. When software is decomposed, the "big picture" is lost, often defeating the purpose of the visualization.

To address these shortcomings, we developed scalable techniques for visualizing program text, text properties, and relationships involving program text. We focus on text because it is the dominant medium for implementing large software systems. Although visual languages have made great strides, particularly in restricted domains,[6] they are not often used for general-purpose, large-scale program development. Virtually all coding of large systems takes place in text, which will likely continue in the foreseeable future. Programs are embodied in text, and programs are modified by changing it.

The novel aspects of our research involve how we represent the code and apply these representations to visualizing production software. Our research was motivated by the examination of several multimillion-line legacy software systems in Bell Laboratories and the issues associated with maintaining and enhancing them. In one example, the code evolved over decades and was written and maintained by thousands of programmers.

We have applied our tools to visualize

- code version history,
- differences between releases,
- static properties of code,
- code profiling and execution hot spots, and
- dynamic program slices.

The remainder of this article illustrates and details our techniques.

VISUAL REPRESENTATIONS AND INTERACTIONS

We first describe four visual code representations and then discuss the interaction techniques for manipulating them.

Line representation

Figure 1 illustrates the line representation approach by providing three color-coded views of text at varying scales. The left pane shows a screen of code from the middle of a file using a *text* representation in readable form. The middle pane shows the same text, this time on a smaller scale, but with the same coloring. Although the smaller code is barely recognizable, the same screen real

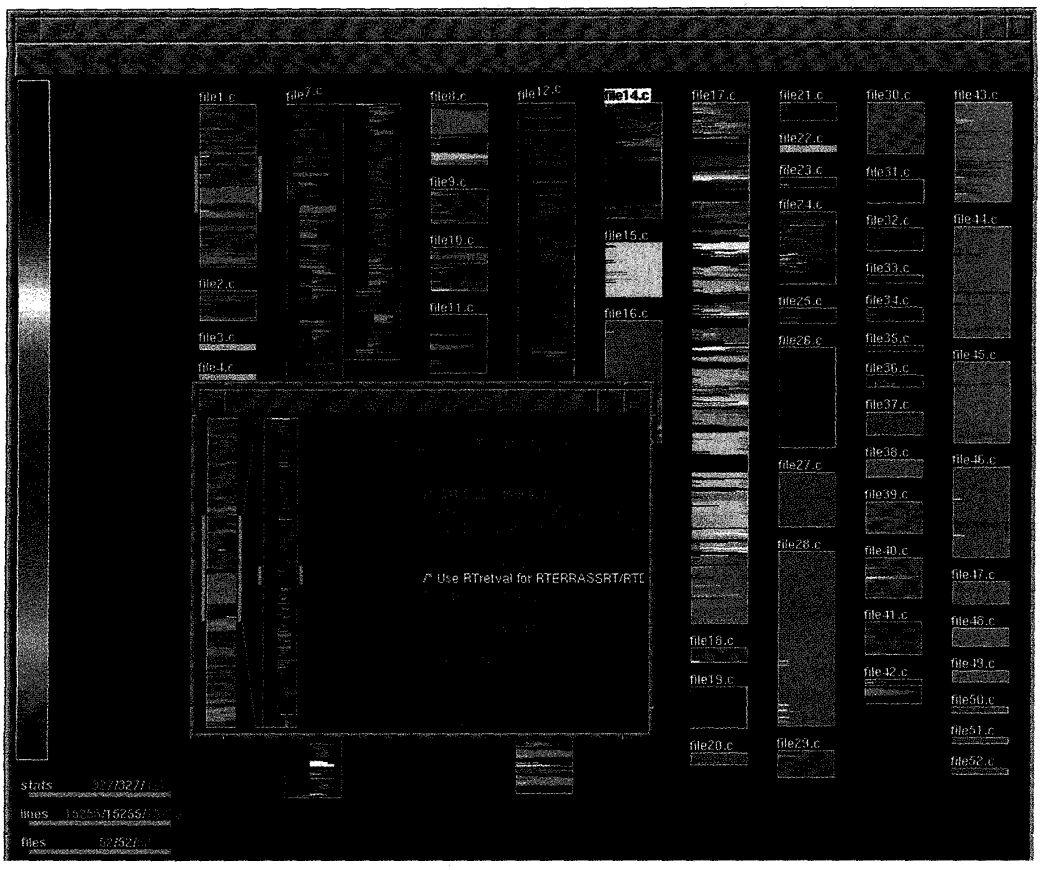


Figure 3. Code age. Fifty-two files comprising 15,255 lines of code are represented. The newest lines are shown in red and the oldest in blue, with a rainbow color scale in between. The browser (smaller window) incorporates text, line, and pixel representations.

estate can show more text. Finally, the right pane shows each line reduced to a single row of pixels, preserving the indentation, length, and coloring of the original code. This reduced *line* representation makes the entire file visible, spanning one-and-one-half columns. The red box in each pane shows the same text at three different scales.

A line's color may also code a categorical or continuous statistic such as each line's age or the number of times it executed in a regression test (see later examples). Each representation in Figure 1 shows code-age distribution in the file through color-coding, an effective technique for layering information. Users can manipulate the color mapping through a highly interactive user interface.

Loops and conditional control structures are usually indented in standard programming practice. By preserving indentation, the line representation makes these structures visible while simultaneously displaying a large volume of code. It is also sometimes useful to turn off the indentation and fill each line to both margins to emphasize the row colors (as shown later in Figure 10).

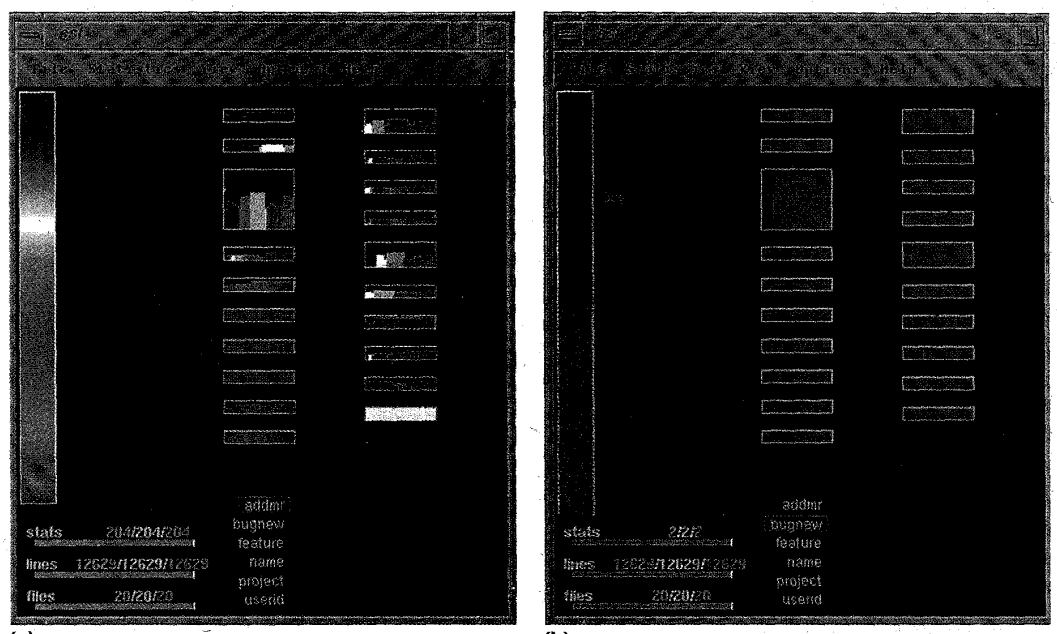
Pixel representation

The *pixel* representation, illustrated in Figure 2, shows each line of code by using a small number of color-coded pixels, thereby achieving a higher information density than the line representation. The pixels are ordered left to right in rows within the columns, each column corresponding to a single file. This representation can show over a million lines of code on a standard high-resolution monitor (a 1,280 × 1,024 monitor contains 1,310,720 pixels).

A pleasing aspect of this representation, as shown in Figure 3, is that the rectangle sizes corresponding to files are tied to file sizes, making it easy to find and compare the sizes of large and small files. It is also easier to relate the vertical pixel positions in a rectangle to the actual line numbers in the pixel representation than it is to relate the row positions in the line representation, especially when the file spans multiple columns.

The pixel representation can function as a scrollbar, as shown in the browser window in the lower left of Figure 3. This browser contains three views of the text:

- the pixel representation on the left,
- the line representation in the middle showing the area of code selected in the pixel representation, and
- the text area on the right showing the text selected in the line representation.



(a) (b)

Figure 4. Summary representation. Each file is represented as a colored rectangle with a small plot inside. For each file, (a) shows the code age (blue represents the old code and red the new), while (b) shows the amount of bug-fixing code relative to new feature development.

One surprising aspect of the pixel representation is the visual effectiveness of color-coding pixels. Even though individual pixels are small, their color is perceivable and often follows a regular pattern. Sorting the pixels by color within each row helps to emphasize the dominant color in an area of code.

File summary representation

In this representation, which presents file-level statistics, each file is represented by a rectangle. There are four possible rectangle heights, corresponding to the four quartiles of file size (as measured by the number of lines). Because file sizes may vary from a few to tens of thousands of lines, grouping the sizes by quartiles ensures that all files are always visible.

Figure 4 shows summary representations of the same files in two different panes, corresponding to two different statistics. In this case, only three of the four file size quartiles are represented in the data set. Figure 4a shows the code age as a miniature time series within each rectangle, while Figure 4b shows the amount of code added for bug fixing and new functionality. Other possibilities for color-coding include Halstead's program volume measure or McCabe's cyclomatic complexity software metrics.

Hierarchical representations

Source code is often stored hierarchically, organized into directories corresponding to subsystems, subdirectories corresponding to modules, and then files within the subdirectories. The source code tree may contain many layers, depending on the system's complexity and size. Furthermore, many other hierarchies are naturally induced by various programming language constructs, such as "namespace" encapsulation, single-inheritance class hierarchies, and syntactic block structure. All this suggests using techniques for visualizing hierarchical data.

A method developed by Johnson and Shneiderman[7] shows hierarchies by using a generalization of the piechart called a *treemap*. Figure 5 illustrates an extension of their technique modified to show source code organized into three subsystems, each containing directories, which in turn contain files. Figure 5a represents the entire software system, and *X*, *Y*, and *Z* represent its three subsystems. Each subsystem's area is based on an additive statistic, such as the number of noncommentary source lines (NCSL). The subsystems are each partitioned vertically to show their internal directories. In Figure 5a, the rectangles labeled 1, 2, 3, 4, and 5 represent the directories in subsystem *X*. Because each vertical rectangle's area is proportional to the directory's NCSL, the sum of the areas over the directories equals the subsystem's area. This technique allows a straightforward visual comparison of directories within a subsystem, since the area of each visual component is always proportional to the statistic for the corresponding software component.

Figure 5b shows how a bar chart encoding an additional statistic (such as the amount of new code development) can be easily accommodated in the hierarchical representation. Figure 5c shows a zoomed view on subsystem *Y* that reveals file-level statistics.

Discussion

The visual encodings show code at five levels of detail: text, line, pixel, summary, and hierarchical representations.

In the first three views, a line of text is represented by itself, a row, and a pixel, respectively. In the summary and hierarchical representations, aggregation takes place when there are more lines of text than pixels. This aggregation occurs within files for the summary representation, and across files for the hierarchical representation.

These representations pack detailed information into a small fraction of the screen. The line and pixel representations show the respective file sizes and positional information about the statistics. Showing the whole system provides the "big-picture" perspective that is often lost. This global overview shows code in context, coordinates other views showing more detail (as in the browser window in Figure 3), and serves as a navigation framework.

The line, pixel, and hierarchical representations can become "busy" when applied to large systems, making it hard to see specific patterns. To address this problem, our code representations and color scales are interactive (through means of direct-manipulation interfaces), letting users modify the representations to answer particular questions and gain insight. Two popular interactive methods we use are *filtering* and *focusing*.

Filtering by elision (that is, turning off selected color ranges) effectively reduces display clutter. Users may deactivate (and reactivate) individual colors, regions, or remap the colors to emphasize differences. For example, to highlight code at different time periods, the user can "brush" the mouse over the color scale in the system shown in Figure 3.

Focusing often involves *conditioning*, an intersection operation in which the user selects a range or subset of a variable and changes the color scale to another variable. The visualization system filters the display to show only those lines in the intersection. For example, in the system shown in Figure 3, a user can show the bug-fixing code written by a particular programmer by selecting the person's name in a menu and switching the color scale to the bug-fixing variable.

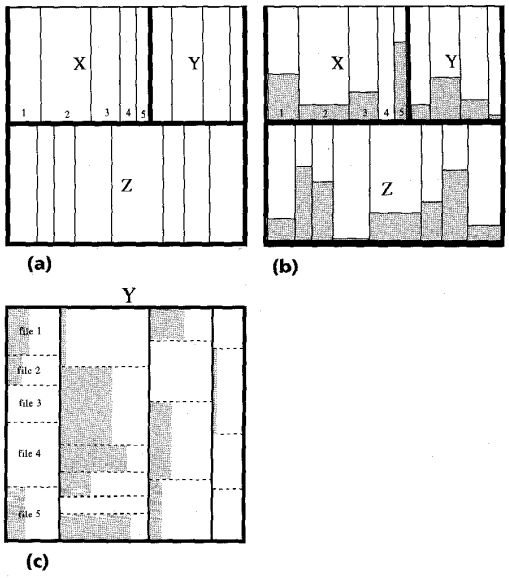


Figure 5. Hierarchical representation. Three views show a statistic for three subsystems (a), a fill statistic for directories (b), and a zoomed view on subsystem *Y* showing file-level statistics (c).

SOFTWARE ENGINEERING

Here we illustrate our software visualization techniques through five case studies. The first three focus on software history and static software characteristics; the last two discuss execution behavior.

Code version history

Version-control systems are widely used for managing code and maintaining a complete history of code changes. These databases contain line-level information, including

- when each line was last modified,
- which programmer wrote particular sections of the code, and
- where bugs and bug fixes are located.

Figures 3, 4, and 6 show views of code age, age and bug fix, and fix-on-fix information, respectively. This information was extracted automatically from a version-control system.

Software tools for exploring version data can increase productivity in three ways. First, new programmers can use the tool for code discovery. For example, color coding by programmer name identifies who last modified the code and might be a source of information about how it works. Second, visualizations can highlight regions in the software that exhibit code decay. Rainbow files that have been changed by many programmers frequently contain errors, suggesting that the code would benefit from reengineering. The third file in Figure 4, a large, frequently changed file containing many bug fixes, is an example.

Finally, in a large project being concurrently developed, it becomes difficult for subsystem owners to track changes. By omitting all but the most recently modified lines, a subsystem owner can identify the current development activity and inspect this code to ensure that it meets coding standards. Figure 3 shows clear patterns of recent modifications. File14.c and file17.c, which are predominantly red and yellow, have been recently changed; several other blue and green files on the far right represent more stable code.

Figure 6 on the next page shows *fix-on-fix* rates using the line representation in a "split-column" mode. Each column is split lengthwise to show two statistic values per line. In this example, only lines representing bug fixes are colored. A fix-on-fix occurs when an error is repaired in the code and subsequently fixed again because the original repair was faulty as well. Fixes themselves indicate rework that directly translates into lost productivity; fixes-on-fixes indicate greater amounts of rework. In Figure 6, each column is split in two, with the left side showing all lines added to fix bugs and the right side showing any subsequent fixes. (No line defines the split; some files do not have bug-fixing code and are subsequently blank.) Only the lines fixing bugs are colored. Half-lines mean that the original bug fix worked (so far), and whole lines indicate a fix-on-fix. The color of the left half-line encodes the age

of the first bug fix, and the color of the right half-line encodes the age of the latest bug fix (if there has been one). If the hues of the respective half-lines are different, the original fix and subsequent fix-on-fix occurred far apart in time, requiring duplicate programmer discovery.

Differences between releases

In the Unix environment, diff is the standard tool for comparing two files. There are versions of diff for performing three-way comparisons of files (diff3), side-by-side comparisons (sdiff), and graphical interfaces for showing diff output.

Although there has been extensive work on differencing algorithms, scant attention has been paid to understanding diff output. It is difficult, for example, to understand more than a few lines of diff output, and impossible to understand the output when diff compares entire directory structures (which can be done with the -r option). The differences between two versions frequently span many lines in many files. Merely identifying and examining the differences is time-consuming.

Figure 7 shows a graphical tool that displays differences between entire directories and between file pairs simultaneously. Four colors code the text: Deleted lines are red; added lines are green; changed lines are yellow, and unchanged text is gray. Figure 7a shows a browser displaying side-by-side differences between two files, so that common text is vertically synchronized. A line representation in the center of the browser globally summarizes the differences. The yellow rectangle acts as a scrollbar.

Figure 7b contains two hierarchical views and a line representation that show differences by directory (top), all files within one level of the directory (middle), and side-by-side comparisons of each file (bottom right). In the top and middle views, a stacked bar chart shows the percentage of each type of text in a directory or file. A selector on the left lets users focus solely on deleted, added, or changed code. The views are linked so that clicking on any file in the hierarchical view points the browser to that file.

Code characteristics and software complexity

Here we show two static properties of code: preprocessor directives and nesting complexity.

PREPROCESSOR DIRECTIVES. Using preprocessor directives such as `#ifdef` and `#endif`, which control conditional compilation, is a common approach to maintaining platform-specific code. These directives break the code into fragments, making it hard to understand. Figure 8 shows 39 of the 119 files in Bell Laboratories' Vz visualization library that contain platform-specific preprocessor directives. Code specific to MS-Windows is red, X-Motif green, common code blue, and all others gray. This visu-

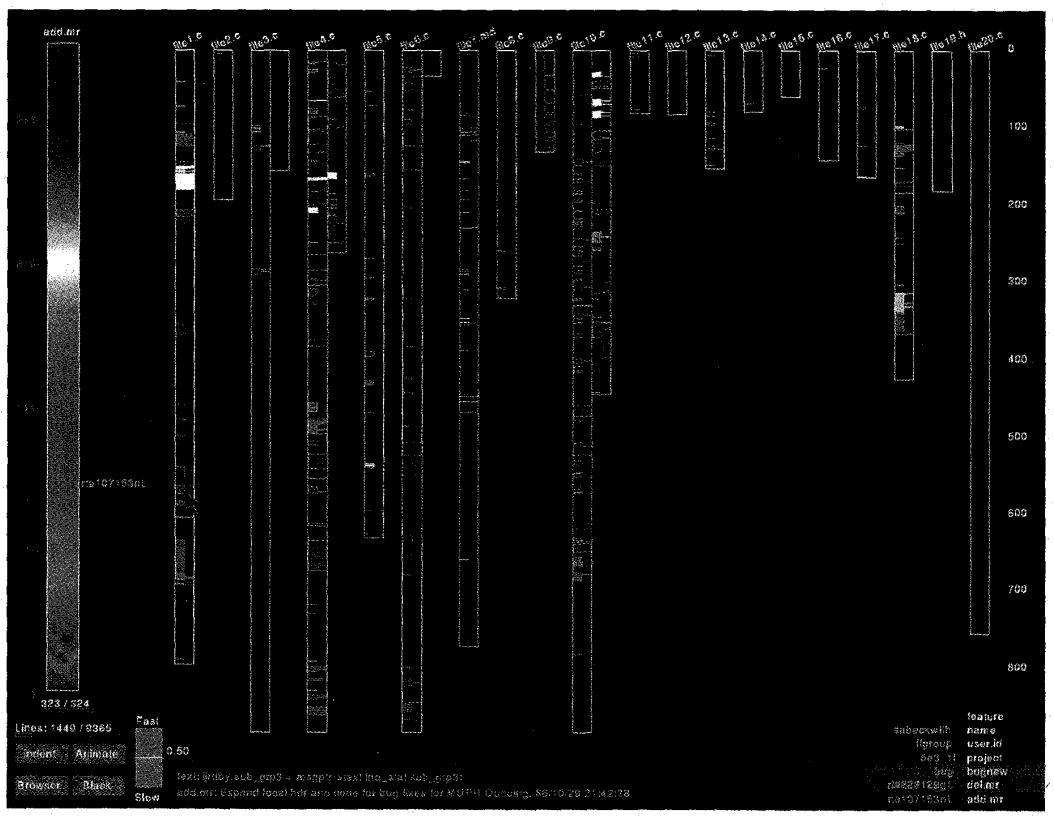
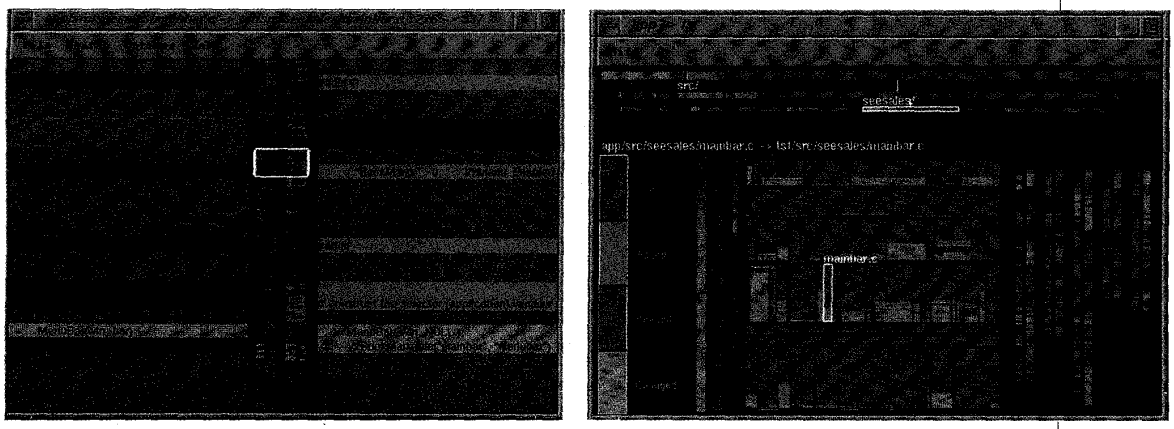


Figure 6. Fix-on-fix rates. Each column is split in two with half-lines on the left indicating initial bug fixes and half-lines on the right showing subsequent fixes to the original changes. Whole lines indicate a fix-on-fix.

alization shows that the amount of code required for each platform is roughly the same. The one exception is a large chunk of MS-Windows-specific code that defines an array mapping the X-Motif color names to red-green-blue (RGB) color values.

CONDITIONAL NESTING COMPLEXITY. The conditional nesting level of a statement is the number of loops and conditionals surrounding it. In general, the greater the nesting level of a statement, the more difficult it is to determine the conditions under which it will execute. Figure 9 on the next page shows 48,913 lines of C code spread across 68 files, using the pixel representation, with color indicating the nesting level. The lines four or more levels deep have been highlighted (see the numbered color bar on screen left). As the figure shows, this code has a high degree of nesting; more than one-fifth of it is nested four levels or deeper. One file contains code 13 levels deep. Such a visualization can be used to target pieces of code that might benefit from restructuring.



(a) (b)

Figure 7. Two views of program differences between versions: between file pairs using synchronized text areas and a line representation for a global overview (a), and at the directory level with several hierarchical views (b). Added lines are green, deleted lines red, and changed lines yellow. Unchanged lines are gray.

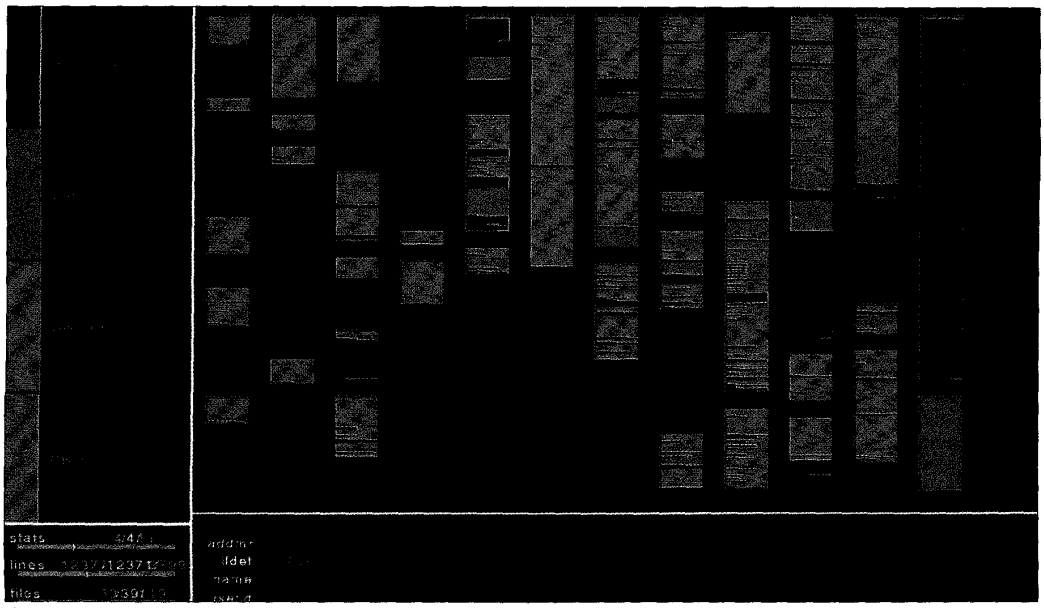


Figure 8. Preprocessor versioning within Vz graphics library comparing X-Motif with Windows-specific code. Windows is red, X-Motif green, and common blue. The latter represents code not contained in ifdefs (no ifdef'ing). Other ifdef'ing, like Irix, Sun, and Borland, is gray. Only the files with Motif- or Windows-specific code are shown (39 out of 119).

Figure 9. Nesting level of 48,913 lines of C source code. Each line is colored to reflect the number of surrounding loops and conditionals. Levels 0-3 have been filtered out. Over 20 percent (10,803 lines) of code is nested within four or more conditionals.

Code profiling and execution hot spots

Optimizing the runtime performance of software is an important problem in systems development. To find the code inefficiencies, programmers often use profiles to determine where the most CPU time is spent and then make changes to reduce it. This process, called code tuning, can result in stunning increases in runtime performance. Most compilers support line-level profiling. However, understanding line-execution counts and relating them to the program's structure is tricky, even with small programs. It is hard to get an overview of the code that helps you understand which lines are being executed, much less gain insight into the specific pattern of code execution.

Figure 10 shows "hot spots" in a source-code brower's code that was run through a user test. The display employs line representation without line indentation to make the color patterns more visible. The color of each line encodes the number of times the line executed. Red denotes high execution frequency and blue denotes the opposite. Gray lines denote the nonexecuted lines—those missed by the test—while blank spaces indicate nonexecutable lines, such as declarations, comments, and static arrays.

Several conclusions can be drawn from Figure 10. This program contains a comment block at the top of each file (the blank regions at the top of each rectangle). The rightmost file, scanner.c, is the largest, spanning five and one-third columns. It was generated by the lex tool. The huge block of nonexecutable code is a static table that implements lex's finite-state machine. For this program, the large number of dark gray lines indicates that code coverage was low. No code in file help.c or invlib.c executed. The "hot spots," or the most frequently executed lines, are concentrated in a few lines within three files: scanner.c, find.c, and crossref.c. These areas are candidates for optimization. The lex parsing code in scanner.c could be hand-optimized. Investigation shows that the hot lines in find.c and crossref.c are key loops, and comments around these loops (not shown) indicate that the programmer is aware that this code is critical.

Dynamic program slices

To understand, modify, or debug code, programmers must determine which parts are relevant to the task at hand. They frequently spend time exploring code only to find it irrelevant to the question that they are trying to answer. An automated technique called dynamic program slicing can determine the code that impacts the computation of a statement or procedure in a particular program execution, relieving the programmer of burdensome discovery work. Although slices are typically smaller than the original program, they may still be quite large and complex, crossing many procedure, file, and module boundaries. Code visualization via reduced textual representations is well suited for displaying and exploring program slices.[8]

Figure 11a shows a dynamic slice from a C-program file. It contains many procedures, each delineated by a rectangle. Within each rectangle appears a line representation of the procedure's code. Executed code is light gray and unexecuted code dark gray; code in the slice appears in color. The point of the slice at the mouse is red. In this example, yellow statements directly affect the slice point, while green statements directly affect the yellow (but not the red), and so on. Figure 11b displays a forward slice. The four red statements shown in the browser window comprise the slice point. These statements correspond to the red lines in the red rectangle in the overview. The overview shows that three other procedures contain code immediately affected by the slice point. This slice contains most of the routespl.c file's procedures, but only half of the splines.c file's procedures.

The slicing interface lets users quickly examine many slices. In brushing mode, a slice is computed and displayed for each statement or procedure that users touch with the mouse. This lets users find patterns. Once a slice is "fixed," users can eliminate and sort procedures and

files based on whether they are in the slice. The slice visualization distinguishes between "open" procedures that show the line representation and "closed" procedures that summarize a statistic (in this case, the percentage of statements in the procedure that are in the slice). Users can interactively elide parts of the display to reduce its complexity.

LIBRARY AND IMPLEMENTATION

The figures in this article were produced by a family of visualization systems, each targeted to a particular application. Underlying all systems is a common software substrate embodied in an object-oriented, cross-platform (MS-Windows, OpenGL, and X-Motif) C++ library. The Vz library, shown in Figure 8, provides a foundation for building highly interactive graphic displays.

With our visualization framework, we can produce applications easily; quick iteration supports the exploration of new ideas. Each view takes between 500 and 1,000 lines of code.

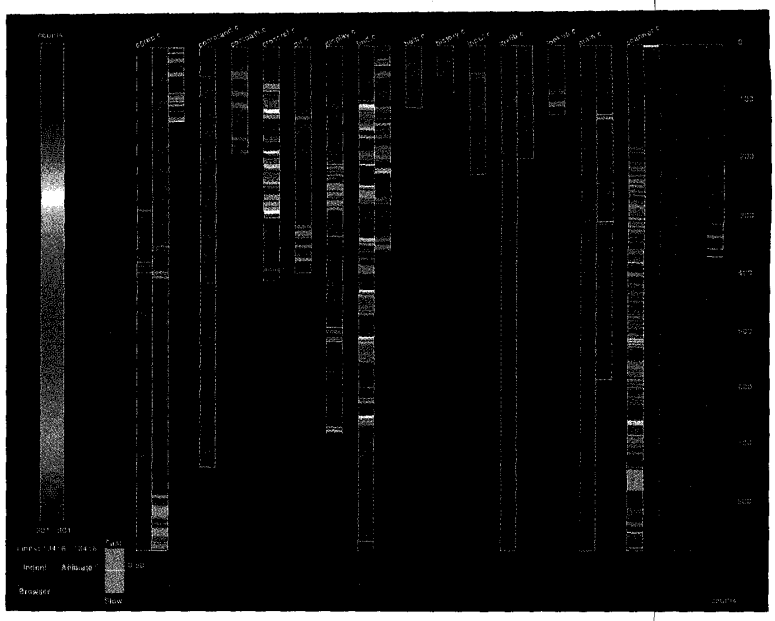
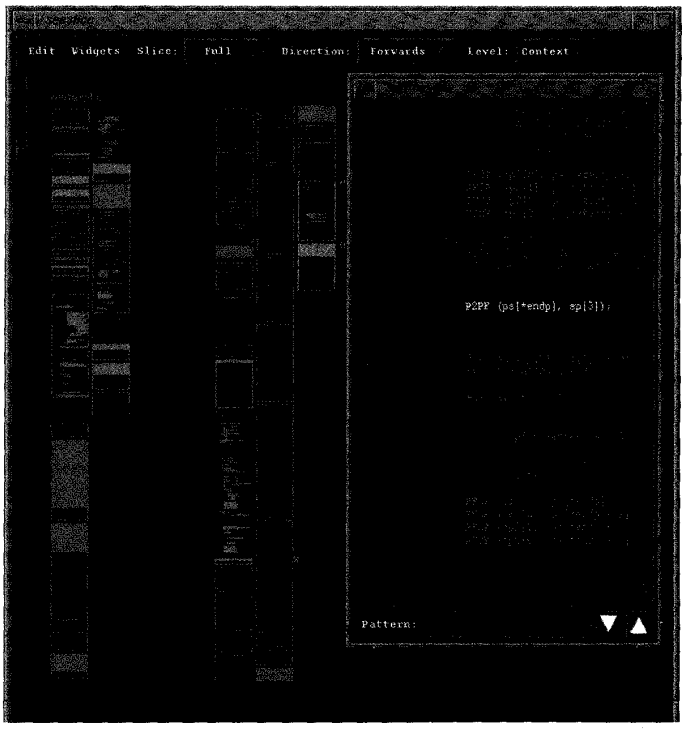


Figure 10. Program-execution hot spots. Red and yellow indicate "hot spots" in program execution based on line-level profile data collected from a test run. Line indentation has been turned off to make color patterns more visible.

(a)



(b)

Figure 11. Two views of dynamic program slices from the SeeSlice system: a slice in a C-program file (a) and a forward slice spanning two files (the slice point is red) (b). Some procedures have been "closed" to hide their line representations.

The Vz C++ library

- hides platform and operating system differences;
- handles display rendering in a portable manner;
- provides a standard "look-and-feel";
- facilitates view linking; and
- includes many utility classes for data management, statistics, and mathematics.

As the foundation for data visualization, the library provides the core and common functions in our system and tools.

DISCUSSION AND RELATED WORK

Now we briefly review some related work and compare and contrast our different techniques for visualizing software. For a good overview and taxonomy of software visualization techniques, see Price, Small, and Baecker.[9]

Algorithm animation

Many people associate software visualization with algorithm animation, that is, using pictures and computer graphics to understand program execution. Brown's dissertation[3] established this technique as fundamental to illustrating complicated algorithms.

Our work takes a complementary approach to algorithm animation, focusing on static or dynamic program properties associated with lines of code rather than illustrating how algorithms operate.

Text views

Eick, Steffen, and Sumner[10] originally introduced the line representation (as in Figure 1) for showing software change history. This line view looks somewhat like that of Baecker and Marcus,[5] who focus on techniques for typesetting C code. The biggest difference is that Baecker and Marcus's views are exact, scaled reductions of pretty-printed code, whereas we focus on a variety of scalable representations.

One of our visualization goals is to use every available monitor pixel to show information. For line representations, the practical upper limit with currently available monitor technology is about 100,000 lines on a single display. This limit and our desire to visualize multimillion line systems motivated the pixel, summary, and hierarchical representations.

Graph drawing

Acyclic graphs are a natural representation for many software artifacts, particularly those involving abstraction. These graphs, as previously discussed, usually consist of node and link diagrams carefully arranged by sophisticated layout algorithms to show the underlying structure of complicated systems. The graphs may describe relationships such as procedure or function calls and class inheritance.[2] The function call graphs can be animated as a visual representation of how a program executes and can be color-coded to show "hot spots."

Perhaps the most difficult aspect of showing software through graphs involves the graph layout problem. The nodes and edges of the graph must be positioned in a pleasing and informative layout that clearly shows the underlying graph's structure. Many techniques have been proposed for laying out arbitrary graphs.[11]

Unfortunately, in practice, drawing informative graphs is exceedingly difficult, particularly for large systems. The function call graph for even a tiny single-person project can contain thousand of links and hundreds of nodes. The resulting graphs, even when drawn carefully, are often too busy and cluttered to interpret.

WE HAVE DEVELOPED A SUITE of scalable representations for code and applied them to several real software-engineering problems by using software visualization tools. Our most successful visualization systems were designed to solve specific problems. The tasks motivating our research have led to many special-purpose views that we generalized and incorporated into the visualization tools.

Our experience has been that the most interesting and engaging views are highly detailed, providing both a global overview and fine-grained detail. We are attempting to use all available screen real estate by having every pixel convey useful information. An interactive user interface lets users quickly filter and focus the display on the areas of interest in the code, with "drill-down" views tightly linked to the global overview.

The systems we presented are used daily within Bell Laboratories' development community, helping software developers work on the 5ESS product, a real-time switching system containing millions of lines of code developed over the past two decades by thousands of software engineers. The initial developer feedback has been very positive. ■

ACKNOWLEDGMENTS

We are grateful for the contributions of David Atkins, Marla Baker, and Graham Wills.

References

1. J.O. Coplien and J. Erickson, "Examining the Software Development Process," *Dr. Dobb's J.*, Vol. 19, No. 11, Oct. 1994, pp. 88-95.
2. *Practical Reusable Unix Software*, B. Krishnamurthy, ed., John Wiley & Sons, New York, 1995.
3. M.H. Brown, "Algorithm Animation," in *ACM Distinguished Dissertations*, MIT Press, New York, 1988.
4. T. Chilimbi et al., "StormWatch: A Tool for Visualizing Memory System Protocols," *SuperComputing 95*, IEEE CS Press, Los Alamitos, Calif., CD-ROM No. 7435, 1995.
5. R.M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs*, Addison-Wesley, Reading, Mass., 1990, p. 235.
6. S.-K. Chang, *Principles of Visual Programming*, Prentice Hall, Englewood Cliffs, N.J., 1990.
7. B. Johnson and B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures, *Proc. IEEE Visualization 91 Conf.*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 284-291.
8. T. Ball and S.G. Eick, "Visualizing Program Slices," in *IEEE/CS Symp. Visual Languages*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 288-295.

9. B.A. Price, I.S. Small, and R.M. Baecker, "A Taxonomy of Software Visualization," *J. Visual Languages and Computing*, No. 3, Vol. 4, 1993.
10. S.G. Eick, J.L. Steffen, and E.E. Sumner Jr., "Seesoft—A Tool for Visualizing Line-Oriented Software Statistics," *IEEE Trans. Software Eng.*, Vol. 18, No. 11, Nov. 1992, pp. 957-968.
11. E.R. Gansner et al., "A Technique for Drawing Directed Graphs," *IEEE Trans. Software Eng.*, Vol. 19, No. 3, Mar. 1993, pp. 214-230.

Thomas Ball *is a member of the technical staff in the Systems and Software Research Center of Bell Laboratories. His research interests include programming languages, software tools, the dynamic and static analysis of programs, techniques for efficiently monitoring program behavior, and software visualization.*

Ball received a BA degree in computer science from Cornell University in 1987 and MS and PhD degrees in the same field from the University of Wisconsin at Madison in 1989 and 1993, respectively. He is a member of ACM.

Stephen G. Eick *is the technical manager of the Data Visualization Research Group at Bell Laboratories. His research focuses on extracting the information latent in large databases using novel interactive visualizations. Eick's research group has developed tools for visualizing geographic and abstract networks, software source code, text corpora, log files, program slices, and relational databases.*

Eick received a BA degree from Kalamazoo College in 1980, an MA degree from the University of Wisconsin at Madison in 1981, and a PhD degree in statistics from the University of Minnesota in 1985. He holds several software patents. Eick is the program cochair of the Information Visualization 96 Symposium, statistical graphics program chair for the 1996 American Statistical Association Conference, and on the program committees for Visualization 96, Visual Languages 96, and the Fourth IEEE Symposium on Program Comprehension. He is a member of IEEE and ACM.

Readers can contact the authors at Bell Laboratories, Rm. 1G-359, 1000 East Warrenville Rd., Naperville, IL 60566; e-mail {tjball,eick}@bell-labs.com.