Software Visualization in the Large

Thomas A. Ball (`tball@research.bell-labs.com`)
Stephen G. Eick (`eick@research.bell-labs.com`)

November 1996

This paper appeared in IEEE Computer, Vol 29, No. 4, April 1996, pages 33-43.

Software Visualization in the Large

Thomas J. Ball and Stephen G. Eick
Bell Laboratories - Rm 1G-359
1000 East Warrenville Road
Naperville, IL 60566
email: {tball,eick}@research.att.com
phone: (708)-979-4291 and (708)-713-5169
fax: (708)-713-4982

April 17, 1996

Abstract

Software is invisible, disappearing into files on disks. The invisible nature of software contributes to low programmer productivity by hiding system complexity, particularly for large team-oriented projects. Visualization can help software engineers cope with this complexity and thereby increase programmer productivity. We describe four innovative visual representations of software that scale to production-sized systems and illustrate their usage in five software case studies involving: version history, program comparison, static properties of code, performance profiles, and dynamic program slices.

Keywords: software visualization, legacy code, version history, program comparison, profiling, slicing.

1 Introduction

It is well known that large computer programs are complex and difficult to maintain. Production-sized systems, particularly legacy software, may contain millions of lines of code. Even a seemingly simple, small-team software project, such as a spreadsheet, is very complicated [CE94]. Understanding, changing, and repairing code in large systems is time consuming and costly. Perhaps the most difficult software engineering projects involve "programming in the large." These large-team projects are often in maintenance mode and the enhancements involve subtle changes to complex legacy code written over many years. As a result, programmer productivity is low, changes are more likely to introduce errors, and software projects are often late. Knowledge of code decays as the software ages and the original programmers and design team move on to new assignments. The design documents are usually out of date because they have not been maintained, leaving the code as the only guide to system behavior. As a result, it is a very time consuming and tedious task to understand complex system behavior from code.

One way to help software engineers cope with complexity and to increase programmer productivity is through software visualization. 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 through the display of programs, program artifacts, and program behavior. The essential idea is that visual representations can make the process of understanding software easier. Pictures of the software can help to slow the knowledge decay by both helping project members to remember and new members to discover how the code works.

There are three basic properties of software to be visualized:

Software structure. Directed graphs are perhaps the most common method for showing the relationships among software entities. They are the foundation of many CASE and program analysis tools. For example, a node in the graph may represent a procedure and an edge may represent a calling relationship between two procedures [Kri95].

Run-time behavior. Algorithm animation uses graphical representations of data structures and motion to illustrate the higher-level behavior of algorithms [Bro88]. Lower-level views based on program profiles or traces can reveal bugs and performance anomalies [CBEL95].

The code itself. Pretty printers are a basic and widely used form of code visualization [BM90].

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 environments and 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 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. By decomposing the software the "big picture" is lost, often defeating the purpose of the visualization.

To address these shortcomings we have 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. Visual languages have made great strides, particularly in restricted domains [Cha90], but are not often used for general-purpose, large-scale program development. Virtually all coding of large systems takes place in text, and this will likely be the case for the foreseeable future. Programs are embodied in text and program modifications are made through changes to the text.

The novel aspect of our research involves our representations of code and how we have applied them to visualizing production software. Our research has been motivated by the examination of several multi-million line legacy software systems in Bell Laboratories and the issues associated with maintaining and enhancing them. This code has evolved over many years (decades for one of our examples), and has been written and maintained by large teams of programmers (thousands in one example).

We have applied our tools to visualize:

- code version history (subsection 3.1),
- differences between releases (subsection 3.2),
- static properties of code (subsection 3.3),
- code profiling and execution hot spots (subsection 3.4),
- program slices (subsection 3.5).

The remainder of this paper illustrates our techniques and discusses them in detail. Section 2 presents four visual representations of code. Section 3 applies these techniques to five software engineering problems through case studies. Section 4 describes our software infrastructure for building software visualization tools, and Section 5 reviews related work.

2 Visual Representations and Interactions

This section describes four visual representations for code and interaction techniques to manipulate the representations.

2.1 Line Representation

Figure 1 illustrates the line representation by 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 a readable font. The middle pane shows the same text, this time using a smaller font, but with the same coloring. Although the smaller font is "barely" readable, it is possible to show more text using the same screen real estate. Finally, the right pane shows the same text, this time reduced so that each line of text has been shrunk to a single row of pixels, preserving the indentation, length, and coloring. This reduced representation, called the *line* representation, makes the entire file visible, spanning two columns. The red box shows the same text in each of the panes at three different scales.

The color of a line may code a statistic (see the examples in Section 3). Each representation in Figure 1 shows the color and thus the distribution of the statistic in the code. Color-coding is an effective technique for layering information, and color-map manipulation allows for a highly interactive user interface.

A standard programming convention is to indent loops and conditional control structures. By preserving indentation, the line representation makes these structures visible while simultaneously displaying a large volume of code. Although the indentation shows program structure, it is sometimes useful to fill each line to both margins in order to emphasize the row colors (see Figure 10).

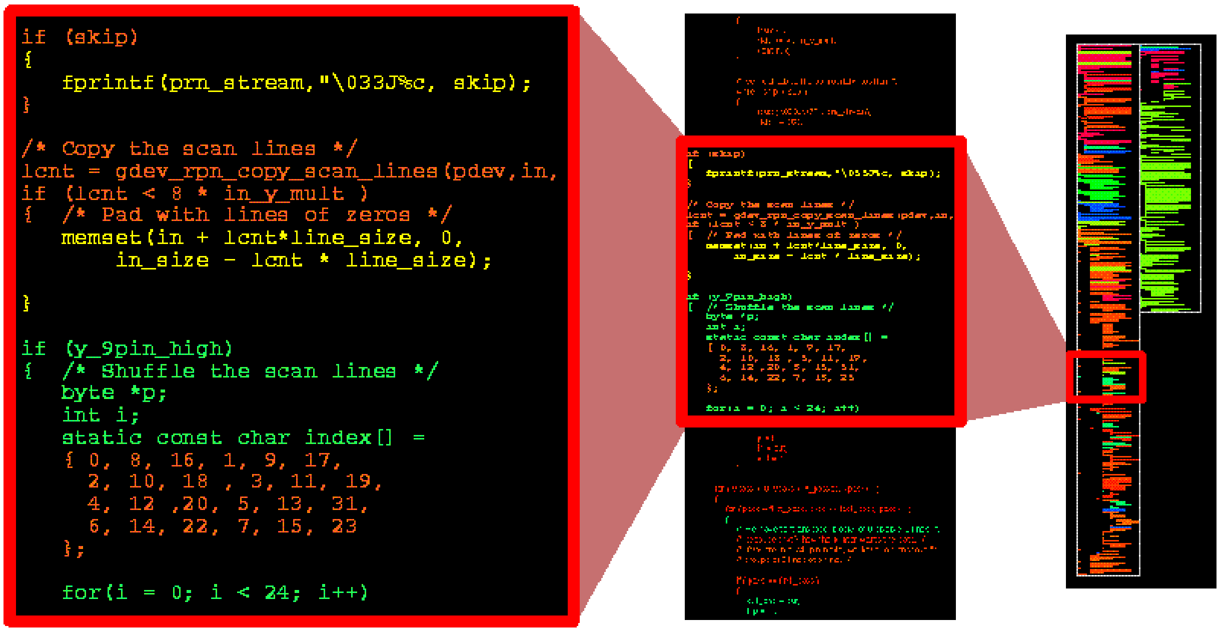


Figure 1: *Line Representation.* Three views using different scaling of color-coded program text. Left pane: *text* representation shows the text at full size. Middle pane: uses a smaller font. Right pane: *line* representation reduces each line of text to a single row of pixels with row length and indentation tracking the original code. The entire file is visible in the line representation, spanning 1.5 columns. The lines are color-coded to show a statistic of interest (in this case, code age, where green represents old code and red represents new code).

2.2 Pixel Representation

The *pixel* representation, illustrated in Figure 2, shows each line of code 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. Using this representation, it is possible to show over a million lines of code using a standard high-resolution monitor.[1]

One pleasing aspect of this representation, as shown in Figure 3, is that the rectangle sizes corresponding to files are tied to the file sizes, making it easy to find large and small files and to compare the relative file sizes. Furthermore, it is easier to relate the vertical pixel positions in a rectangle to the actual line numbers when using the pixel representation than to relate the row positions when using 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; the text area on the right showing the text selected in the line representation.

One surprising aspect of the pixel representation is the visual effectiveness of color-coding the 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.

2.3 File Summary Representation

The *summary* representation 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 number of lines). Because file sizes may vary from a few lines to tens of thousands of lines, grouping the sizes by quartiles ensures that all of the files are always visible.

Figure 4 shows the summary representations of the same files in two different panes, corresponding to two different statistics. In this case, only three of the four size quartiles are represented in the data set. Figure 4(a) shows the code age as miniature time series within each rectangle, while Figure 4(b)

[1] A 1280x1024 monitor contains 1,310,720 pixels.

shows the amount of code added for bug fixing and new functionality. Other possibilities for color encoding include software metrics such as Halstead's program volume measure or McCabe's cyclomatic complexity.

2.4 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 in the hierarchy depending on the complexity and size of the system. Furthermore, there are many other hierarchies naturally induced by various programming language constructs, such as namespace encapsulation, single-inheritance class hierarchies, and syntactic block structure. This suggests showing software by using techniques for visualizing hierarchical data.

A method developed by Johnson and Shneiderman shows hierarchies using a generalization of the pie-chart called a *tree-map* [JS91]. Figure 5 shows an extension of their technique modified to show source code organized into three subsystems, each containing directories, which in turn contain files. Figure 5(a) represents the entire software system, and X, Y, and Z represent its three subsystems. The area of each subsystem 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 5(a), the rectangles labeled *1*, *2*, *3*, *4*, and *5* represent the directories in subsystem X. Each vertical rectangle's area is proportional to the directory's NCSL, and so the sum of the areas over the directories equals the area of the subsystem. This technique allows for a straightforward visual comparison of directories within a subsystem because the area of each visual component is always proportional to the statistic for the corresponding software component.

Figure 5(b) 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, while Figure 5(c) shows a zoomed view on subsystem Y that reveals file level statistics.

2.5 Discussion

The visual encodings show code at five levels of detail: the text, the line, the pixel, and the 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 there may be more lines of code text than pixels and so aggregation takes place. The aggregation is accomplished in two ways: within the files for the summary representation; across the 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 a "big-picture" perspective that is often lost. This global overview shows code in context, coordinates other views showing more detail (see, for example, the browser window in Figure 3) and serves as a navigation framework. Displaying the entire system in a single view also eliminates problems navigating around in the code.

A problem with the line, pixel, and hierarchical representations is that they 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) allowing the user to modify the representations to answer particular questions and gain insight. Two popular methods we use are filtering and focusing.

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

Focusing often involves *conditioning*, an intersection operation where 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 a programmer and switching the color scale to the bug-fixing variable.

3 Software Engineering Examples

This section illustrates our software visualization techniques using five case studies. The first three focus on software history and static software characteristics, and the last two focus on execution behavior.

3.1 Code change 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 version-history.)

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 according to programmer name identifies who last modified the code and who might be a source for information about how it works. Second, the visualizations can highlight regions in the software that exhibit "code decay." "Rainbow" files are multi-colored files that have been changed by many programmers and frequently contain errors, suggesting that the code would benefit from re-engineering. The third file in Figure 4 is an example: It is a large, frequently changed file containing many bug fixes. Finally, in a large project with concurrent development, it is difficult for subsystem owners to track changes. By omiting all but the most recently modified lines, a subsystem owner can identify the current development activity and can inspect this code to ensure that it meets coding standards. In Figure 3 there are clear patterns to the development. Red and yellow files such as `file14.c` and `file17.c` the screen have been recently changed, and several other blue and green files on the far right represent more stable code.

Figure 6 shows *fix-on-fix* rates using the line representation in a "split-column" mode. A fix-on-fix occurs when an error is repaired in the code and then subsequently fixed again because the original repair was faulty as well. Fixes themselves indicate re-work which directly translates into lost productivity, and fixes-on-fixes indicate greater amounts of re-work. 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 bug fixes. Only the lines fixing bugs are colored. Half-lines mean that the original bug-fix worked (so far), and whole-lines mean that there has been 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, then the original fix and subsequent fix-on-fix occurred far apart in time, requiring duplicate programmer discovery.

3.2 Program comparison

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 (see, for example, SGI's *gdiff* command.)

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

Figure 7(b) 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 allows the user to 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.

3.3 Code characteristics and software complexity

This section shows two static properties of code: preprocessor directives and nesting complexity (Figures 8, 9).

Preprocessor directives

A common approach to maintaining platform-specific code is through the use of preprocessor directives such as `#ifdef, #endif` that control conditional compilation. These preprocessor directives break the code into fragments, making it hard to understand. Figure 8 shows 39 of the 119 files in the *Vz* visualization library, developed within Bell Laboratories, that contain platform-specific preprocessor directives. Red represents code specific to MS Windows®, green represents X/Motif, and blue represents common code. This visualization shows that the amount of code required for each platform was roughly the same. The one exception is a large chunk of MS Windows-specific code that defines an array mapping the X color names to RGB color values.

Conditional nesting complexity

The conditional nesting complexity 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 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. As this 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.

3.4 Program profiles and code coverage

Optimizing the run-time performance of software is an important problem in systems development. To find the inefficiencies in their code, programmers often use profiles to determine where the most CPU time is spent and then make changes to reduce this time. This process is called code tuning. There have been many well documented success stories where code tuning resulted in stunning increases in programming efficiency. Most compilers support line-level profiling. However, understanding line execution counts and relating them to the structure of the program is tricky, even for small programs. It is hard to get an overview of the code that helps one understand which lines are being executed, much less to help one gain insights into the specific pattern of code execution.

Figure 10 shows "hot spots" in the code of a source code browser that was run through a user test. The display employs the 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 low execution frequency. Gray lines denote the non-executed lines, the lines that the test missed, while blank spaces indicate non-executable lines, such as declarations, comments, and static arrays.

There are several conclusions that 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 right-most file, `scanner.c`, is the largest file, spanning six columns. It is generated by the `lex` tool. The huge block of non-executable code is a static table that implements `lex`'s finite state machine. For this program, the large number of dark gray lines indicate that code coverage was low. No line in either `help.c` and `invlib.c` was executed. The "hot spots," most frequently executed lines, are concentrated in a few lines within in 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.

3.5 Dynamic program slices

To understand, modify, or debug code, a programmer must determine which parts of the code are relevant to the task at hand. Programmers frequently spend time exploring code only to discover that it is irrelevant to the question that they are trying to answer. Dynamic program slicing is an automated technique that determines the code that impacts the computation of a statement or procedure in a particular execution of a program, relieving the programmer of burdensome discovery work. While 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 [BE94].

Figure 11(a) shows a dynamic slice from one file of a C program. This file contains many procedures, each of which is delineated by a rectangle. Within each procedure's rectangle appears a line representation of the procedure's code. Colors are used to distinguish executed (light gray) and unexecuted code (dark gray), and also to distinguish code in the slice (colored) from other code. The point of the slice at the mouse is colored 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 11(b) 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 there are three other procedures that contain code immediately affected by the slice point. While most of the procedures in file *routespl.c* are in this slice, only half of the procedures in file *splines.c* are in the slice.

The slicing interface allows users to quickly examine many slices. In brushing mode, a slice is computed and displayed for each statement or procedure the user touches with the mouse. This allows the user to find patterns. Once a slice is "fixed," the user can eliminate and sort procedures and files based on whether or not they are in the slice. The slice visualization makes a distinction 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). The capability to interactively elide parts of the display helps user reduce display complexity.

4 Library and Implementation

The figures in this paper were produced by a family of visualization systems, each targeted to a particular application. Underlying all of our systems is a common software substrate embodied in an object-oriented, cross-platform (MS Windows, OpenGL, and X11) C++ library. The Vz library, shown in Figure 8, provides a foundation building for highly-interactive graphic displays. With our visualization framework we can produce applications easily, allowing quick iteration to explore new ideas. With *Vz* each view takes between 500 and 1,000 lines of code.

The *Vz* C++ Library:

- hides platform and operating system differences;
- handles display rendering in a portable manner;
- provides a standard "look and feel";
- facilitates the view linking;
- 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.

5 Discussion and Related Work

This section briefly reviews some related work and compares and contrasts our different techniques for visualizing software. For a nice overview and taxonomy of software visualization techniques see [PSB93].

5.1 Algorithm Animation

Many people associate software visualization with algorithm animation, using pictures and computer graphics to understand the execution of programs. Brown's dissertation [Bro88] established algorithm animation as a fundamental technique for illustrating complicated algorithms.

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

5.2 Text Views

The line representation (Figure 1) was originally introduced for showing software change history in [ESEES92]. This line view looks somewhat like that of Baecker and Marcus [BM90] (p. 235) who focus on techniques for typesetting C code. The biggest difference is that their views are exact scaled reductions of pretty-printed code, whereas our focus is on a variety of scalable representations.

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

5.3 Graph Drawing

Acyclic graphs are a natural representation for many software artifacts, particularly those involving abstraction. These graphs, found in virtually all CASE tools, 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 or class inheritance [Kri95]. The function call graphs may be animated as a visual representation of how a program executes and color coded to show "hot spots," parts of the system using excessive amounts of CPU time that may be candidates for optimization.

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 so as to clearly show the structure of the underlying graph. Many techniques have been proposed for laying out arbitrary graphs [GKNV93].

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

6 Conclusion

Software visualization is important because most software artifacts are naturally invisible. Code disappears into passive files on disks having no physical representation. Obtaining insights into the code is difficult, especially for multi-million line software systems. Much of our research attempts to make software artifacts visible and active.

Our motivation came from analyzing the source code associated with production software systems. We have developed a suite of scalable representations (line, pixel, summary, hierarchy) for code and applied them to several real software engineering problems through software visualization tools.

Our most successful visualization systems were designed to solve specific problems. The tasks that motivated our research have led to many special-purpose views that we have generalized and incorporated into many of our 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. One of our goals is to make maximal use of all available screen real estate by using every available pixel to convey useful information about software. An interactive user interface provides the capability to quickly filter and focus the display on the areas of interest in the code, with "drill-down" views that are tightly linked back to the global overview.

The systems presented in this paper are in daily use within Bell Laboratories' development community. Our motivation to develop the techniques was to help software developers working on Bell Laboratories 5ESS product, a real-time switching system containing millions of lines of code, developed over the last two decades by thousands of software engineers. The initial developer feedback has been very positive.

7 Acknowledgments

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

References

[BE94] Thomas Ball and Stephen G. Eick. Visualizing program slices. In *IEEE/CS Symposium on Visual Languages*, pages 288–295, St. Louis, Missouri, 4 October 1994.

[BM90] Ronald M. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, Massachusetts, 1990.

[Bro88] Marc H. Brown. Algorithm animation. In *ACM Distinguished Dissertations*. MIT Press, New York, 1988.

[CBEL95] Trishul Chilimbi, Thomas Ball, Stephen G. Eick, and James R. Larus. Stormwatch: A tool for visualizing memory system protocols. In *SuperComputing '95 (to appear)*, San Diego, California, December 1995.

[CE94] James O. Coplien and Jon Erickson. Examining the software development process. *Dr. Dobb's Journal*, 19(11):88–95, October 1994.

[Cha90] S.-K. Chang, editor. *Principles of Visual Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[ESEES92] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft^{TM}—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[GKNV93] Emden R. Gansner, Eleftherios E. Koutsofios, Stephen C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization '91 Conference Proceedings*, pages 284–291, San Diego, California, October 1991.

[Kri95] Balachander Krishnamurthy, editor. *Practical Reusable UNIX Software*. John Wiley & Sons, Inc., New York, New York, 1995.

[PSB93] Blane A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3), 1993.

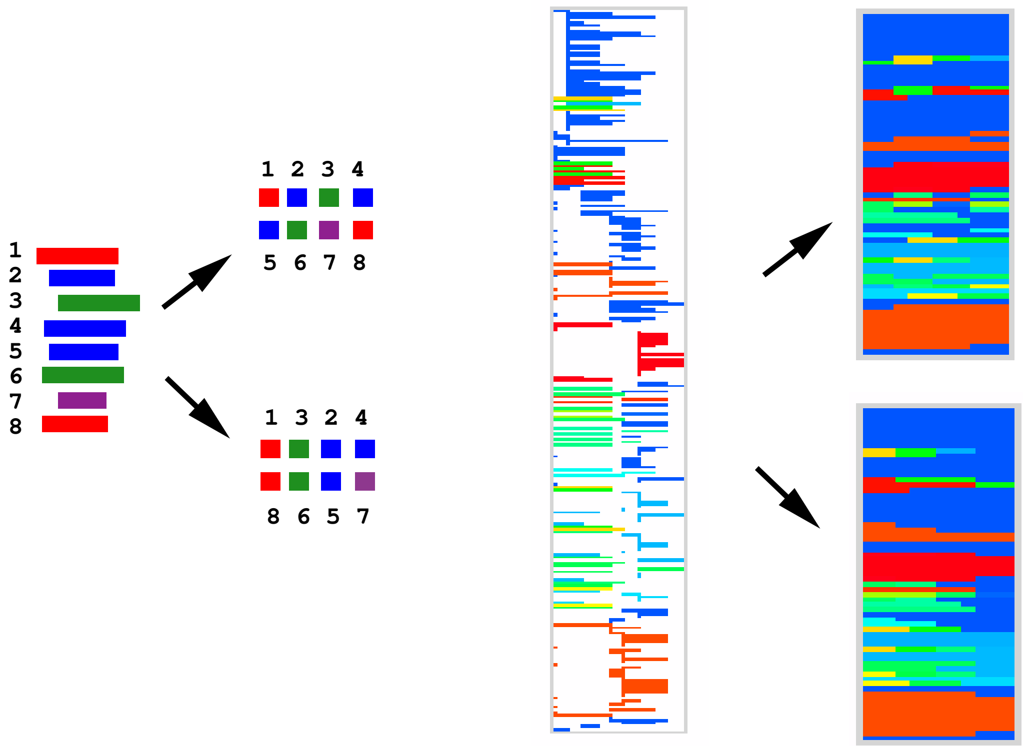


Figure 2: *Pixel representation.* Two examples comparing the line and pixel representations. In each example, the left column shows the *line* representation and the right column shows two possible *pixel* representations. In the *pixel* representation, each line of code corresponds to one (or a small number) of pixels 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 representation the pixels within a line are ordered by their color (in this example, the order of the rainbow color spectrum).

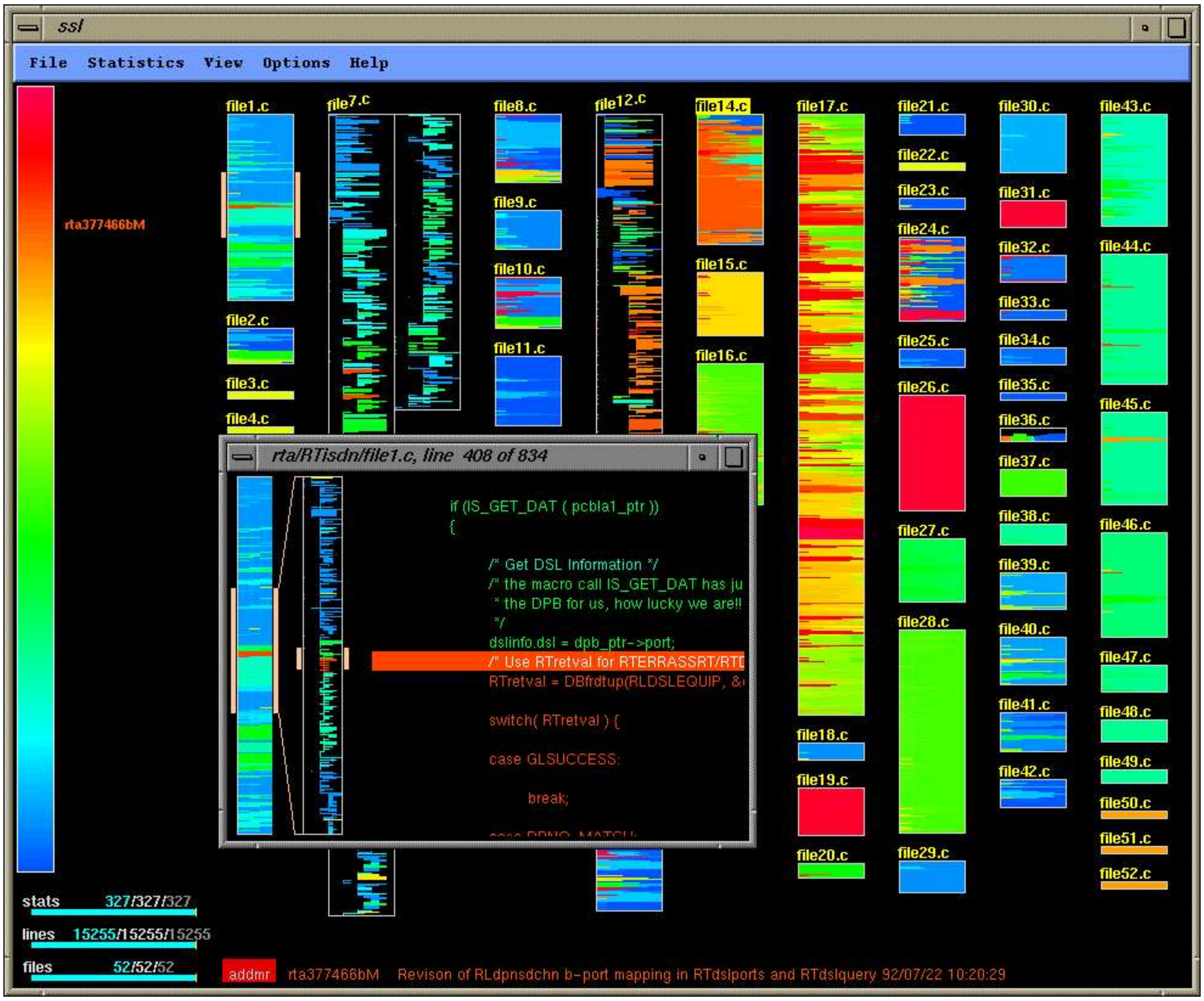


Figure 3: *Code age.* Fifty-two files comprising 15,255 lines of code are represented. The newest lines are shown in red, the oldest in blue, with a rainbow color scale in between. There are three representations of code in this window: text, line, and pixel representations. The browser (smaller window,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

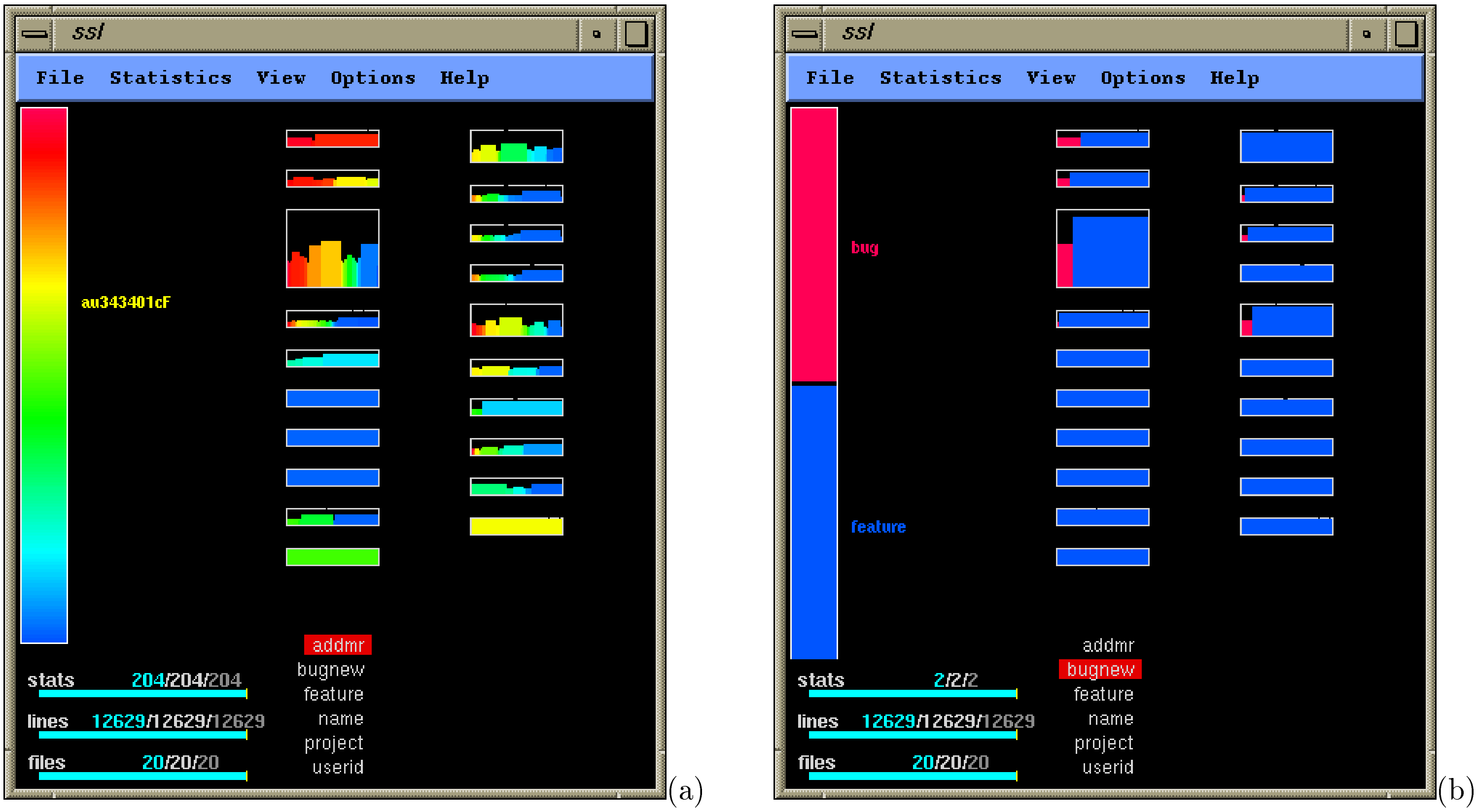
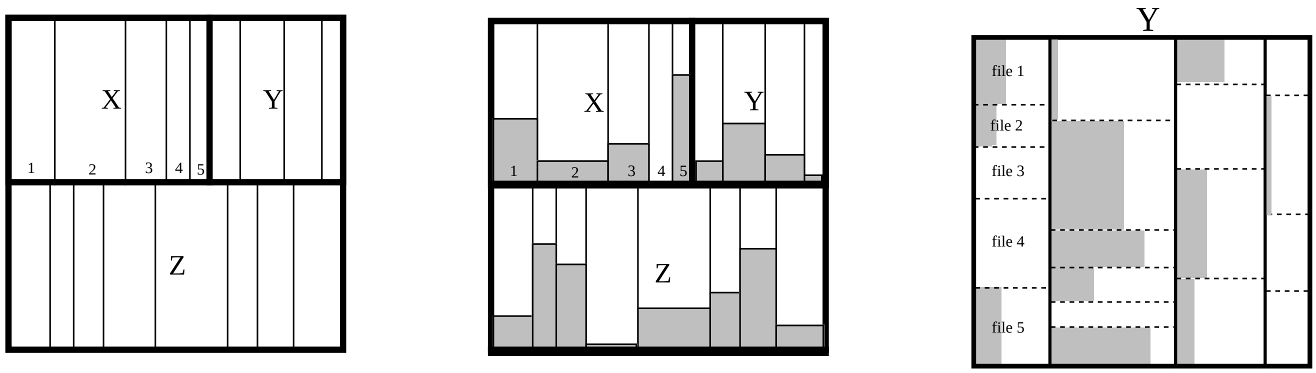


Figure 4: *Summary representation.* Each file is represented as a colored rectangle with a small plot inside. For each file, pane (a) shows the code age (blue represents old code and red represents new code) and while pane (b) shows the amount of bug-fixing code relative to new feature development.



(a) (b) (c)

Figure 5: *Hierarchical representation.* Subsystem and directory statistics (a), a fill statistic for directories (b), and a zoomed view on subsystem Y showing file level statistics (c).

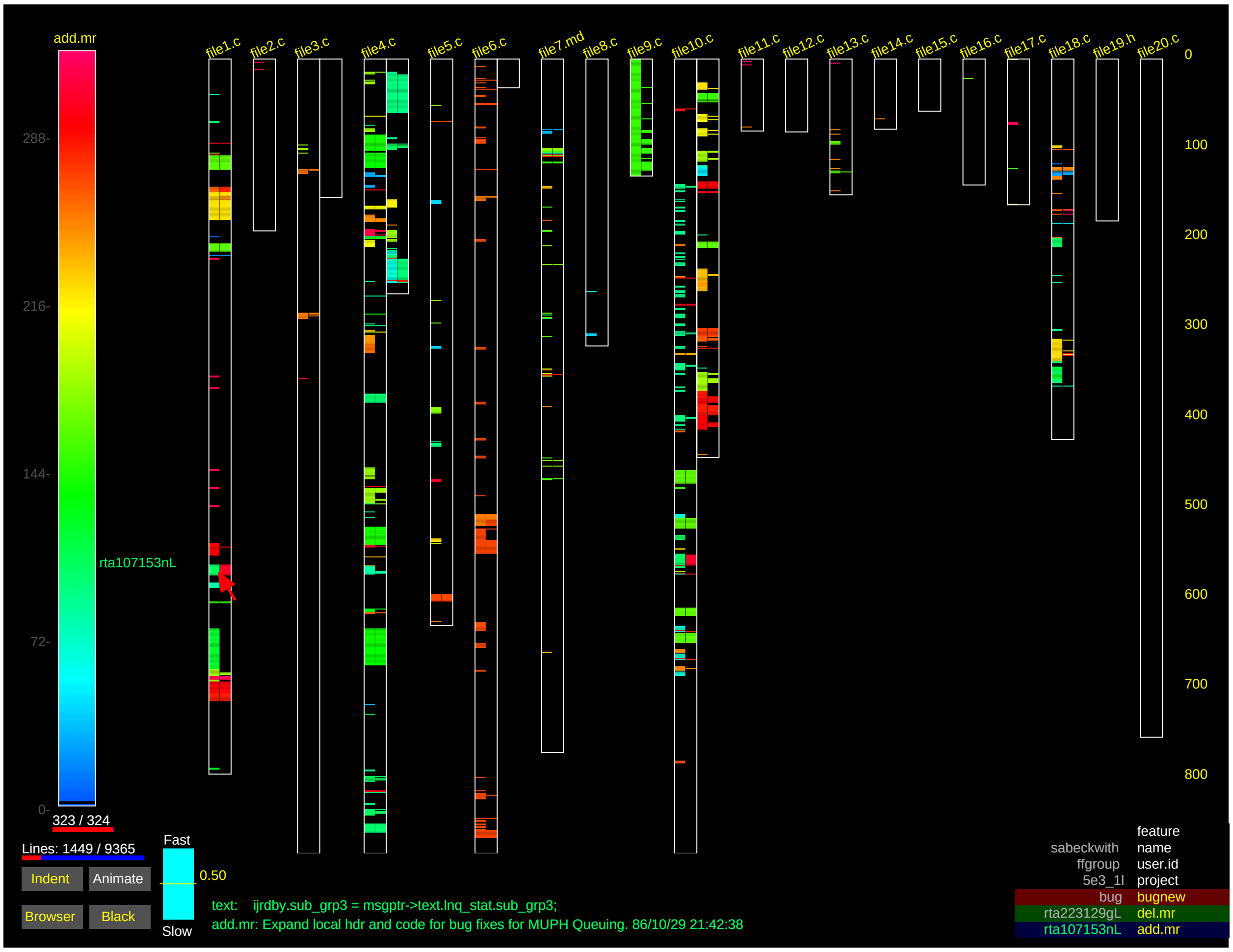
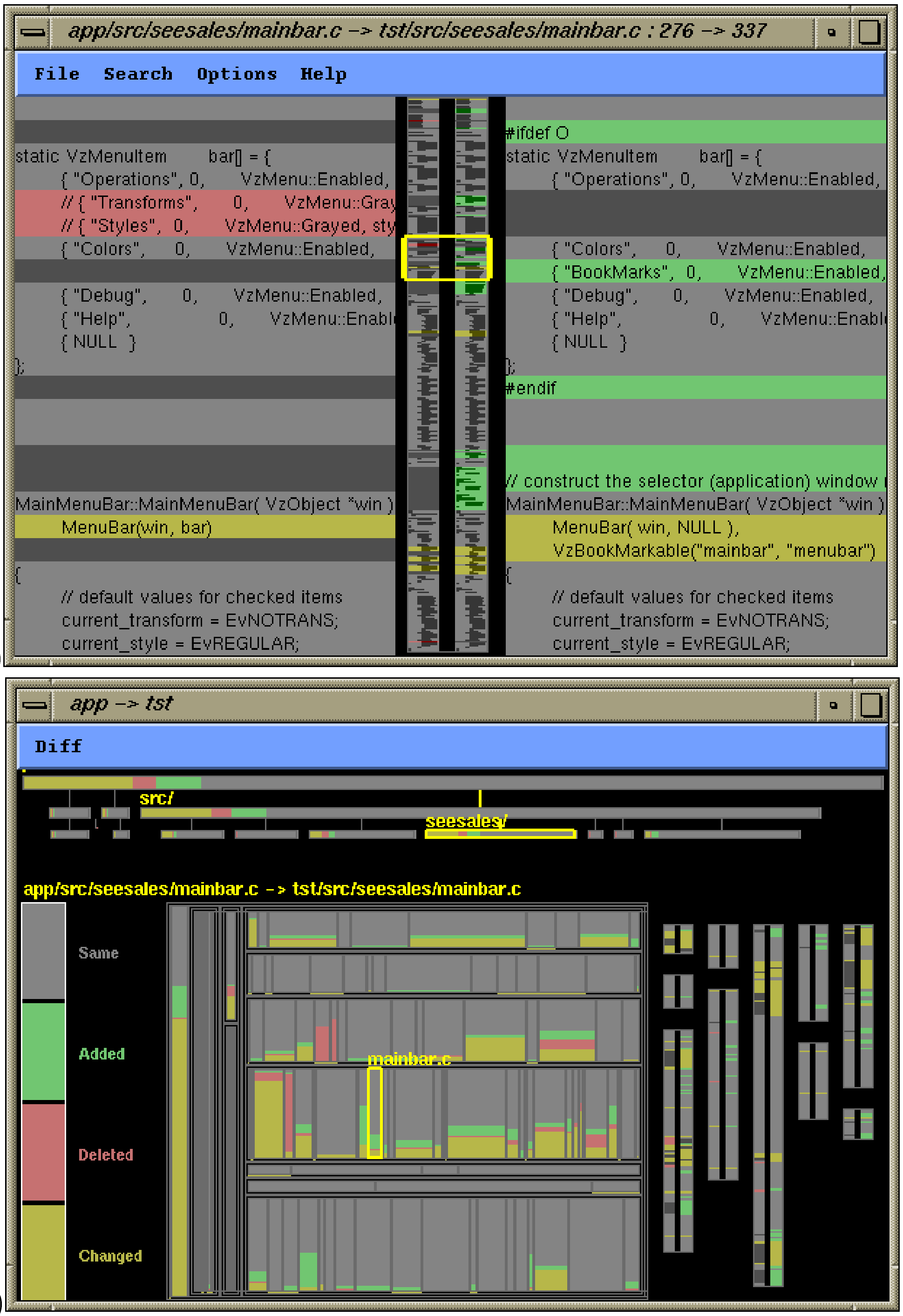


Figure 6: *Fix-on-fix rates.* Each column is split in two with half-lines on the left indicating initial bug fixes and half-lines on the right showing subsequent fixes to the original changes.



(a)

(b)

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

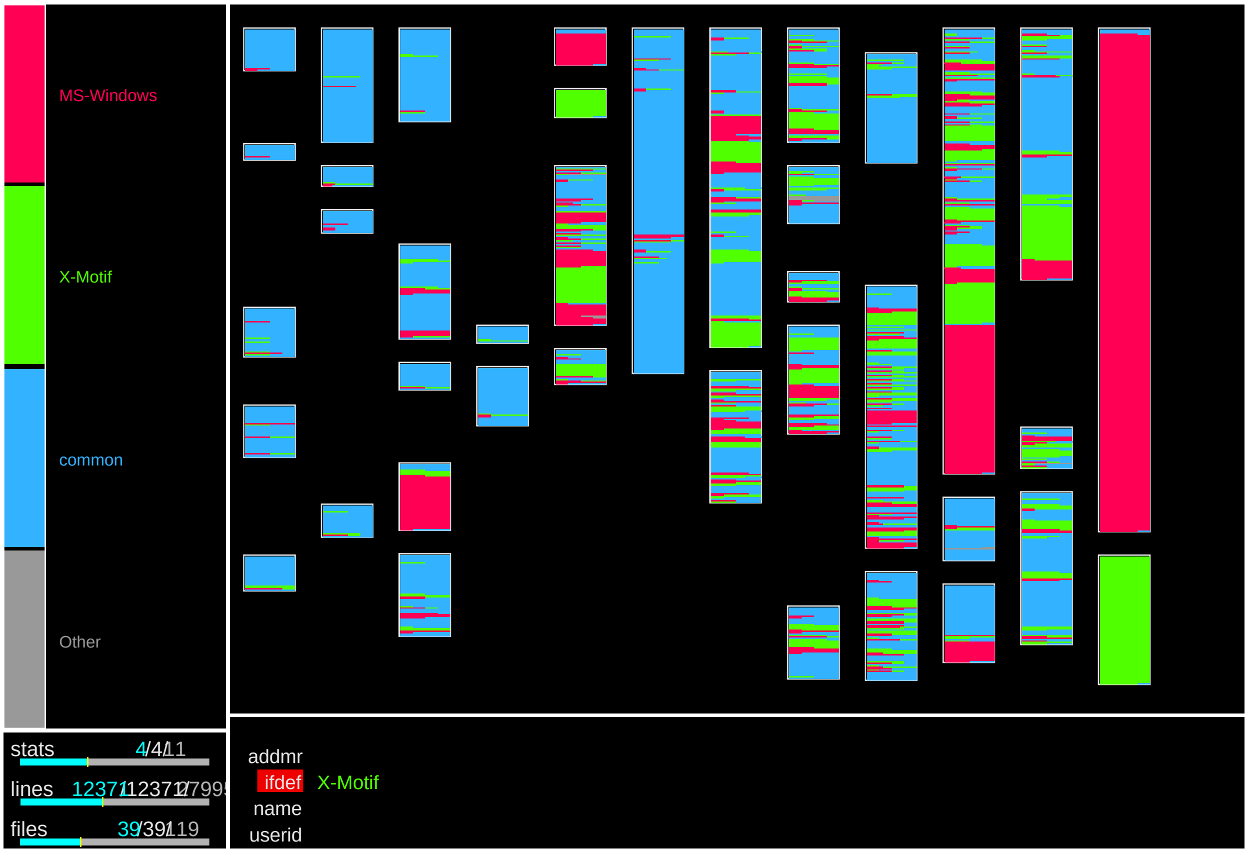


Figure 8: *Preprocessor versioning within a graphics library.* Vz library code showing X/Motif versus Windows-specific code. Red is Windows, green is X/Motif, blue is common (no ifdef'ing). Gray is other ifdef'ing, e.g., irix, Sun, Borland. Only the files with Motif or Windows specific code are shown (39 out of 119 files).

Figure 9: *Nesting level of 48913 lines of C source code.* Each line of source is colored to reflect the number of surrounding loops and conditionals. Levels 0-3 have been filtered out. Over 20 percent (10803 lines) of code is nested within 4 or more conditionals.

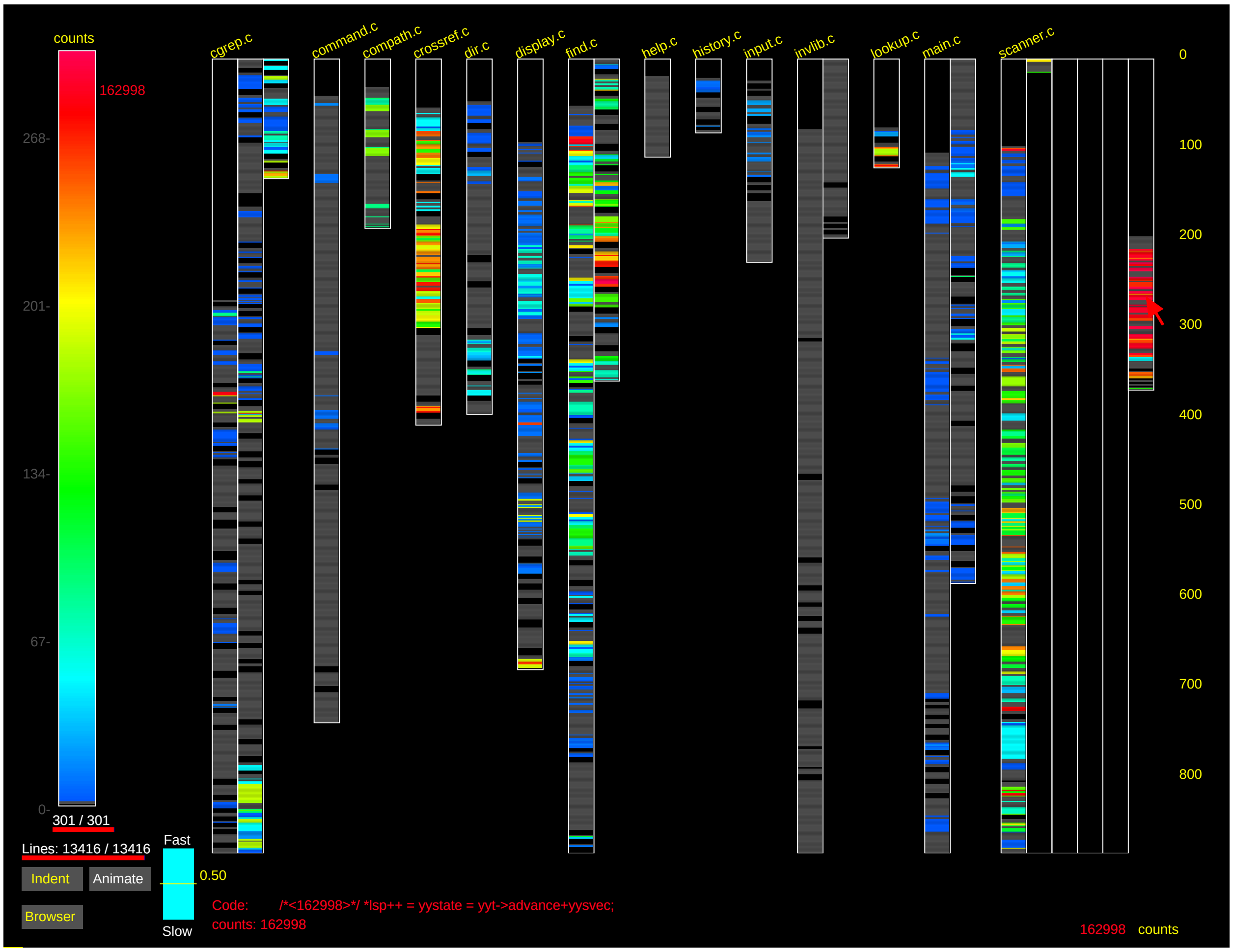


Figure 10: *Program execution hot spots.* The color shows "hot spots" in program execution based on line-level profile data collected from a test run. Line indentation has been turned off to make the color patterns more visible.

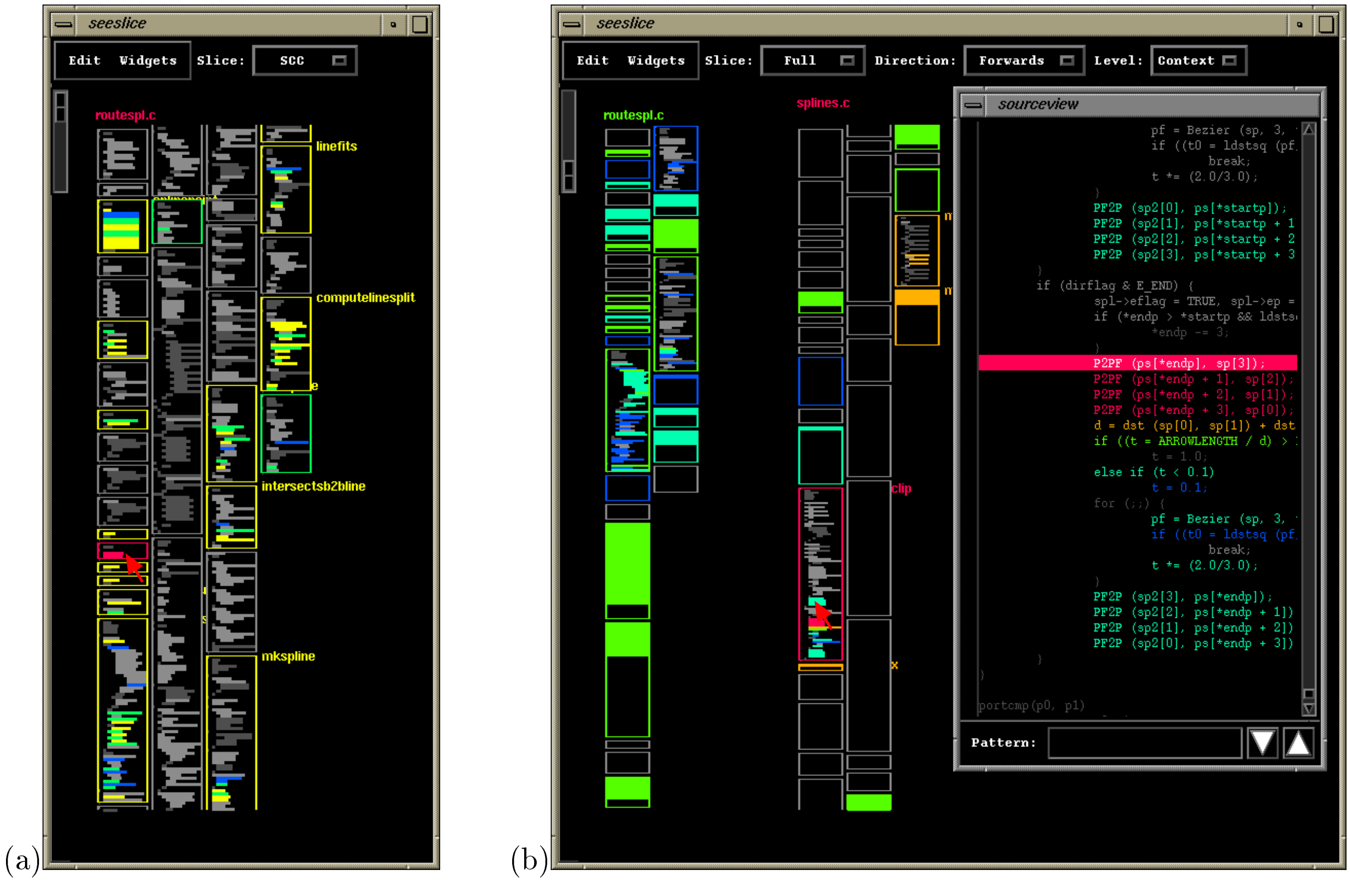


Figure 11: *Program Slices.* Two views from the SeeSlice system. Pane (a) shows a slice in one file. Pane (b) displays a forward slice spanning two files (the slice point is red). Some procedures have been "closed" to hide their line representations.