Evaluating Window Placement for Hypertext-based Source Code Exploration Tools

Jeffrey Blank
University of Maryland
Department of Computer Science
blank@cs.umd.edu

ABSTRACT

Developers incur significant interaction effort while browsing hypertext-based source code windows on large displays. A window placement strategy designed to address this problem, as well as assist program comprehension, is presented. The strategy uses program structure information to position hypertext-based source code windows. For example, when a function implementation is summoned from a function call site, it is placed to the right of the window of focus. Thus, a user following flow of execution shifts their focus rightward to browse deeper into the program call graph. A user study was conducted to compare this strategy with an available space-filling placement system. The study measured window positioning effort and completion time during source code browsing tasks. The study revealed variations in users' window placement behavior, but no significant differences between the window placement strategies.

INTRODUCTION

Although the display of program abstractions has been the primary focus of software visualization research over the last decade, recent developments motivate the investigation of improvements to exploring source code itself. First, studies have shown that software maintenance tasks occupy the greatest percentage of programmer time and that comprehension is its largest component – and the source code may be the only accurate source of information [6]. Second, low-cost 2-megapixel displays are available now and, as prices drop, popular adoption can be safely anticipated for 4-megapixel and higher displays [10]. Recent workshop activity also demonstrates interest in using textual views of source code to support comprehension [3].

When multiple source code windows are open – which easily happens on a large display [11] – their arrangement becomes a noticeable component of the hypertext browsing activity. The programmer expends time and effort to arrange the source code windows to suit the exploration or analysis task at hand, and this arrangement may affect the speed of program comprehension. We suggest that programmers might benefit from careful layout of the large amount of source code that can be shown on-screen.

To test this idea, a new window layout strategy was implemented. The strategy places hypertext source code windows based on the underlying program control flow structure. When a link to a function implementation is requested, the new source code window appears to the right of the current window. Shifting focus rightward across the screen should create a sense of being deeper in the program's call graph. When a link to a references (call-site) list is requested, it opens a new window below the current window. Requesting a call-site from this list results in placement of a window to the left, allowing a user to shift focus leftward to move up the call graph. Data type declarations are placed above the current window. Declarations and references thus appear in the same column as the window whose comprehension they are supporting. To evaluate the placement strategy, a study was conducted in which participants performed source code browsing tasks. The first hypothesis of the experiment is that a window placement system based on the underlying program structure would decrease the amount of effort spent on window manipulation to complete browsing tasks. The second hypothesis of the experiment is that this window placement system would decrease the time required for a developer to make certain realizations while browsing code. Participants' time spent for the tasks was recorded, as well as all window position manipulations. The study did not reveal a significant effect for completion times or window manipulation effort.

After a review of previous work, window placement strategies for hypertexts on large displays are described. Next, a user study designed to evaluate these placement systems is described, followed by its results and a discussion.

PREVIOUS WORK

The most common approach to program exploration at the source code level is hypertext. Hypertext browsing support can be found in popular integrated development environments such as Eclipse and Visual Studio, as well as in more specialized tools such as HyperSoft [9], Source Navigator, CodeSurfer [1], and the Linux Cross Reference (LXR). In these tools, source code elements effectively become links, such as from a function's call to its implementation, or an abstract data type's usage and its

declaration. However, on systems with adequate screen size to display many windows, we are aware of no techniques that attempt to place a new hypertext document in a location that would attempt to minimize manual layout and maximize comprehension. Many of these browsing systems rely on the system window manager. Typical strategies for placement include cascading the new window above the current one, placing it in the nearest available open space, or placing it at an arbitrary absolute position. The resulting layout of source windows on large displays is distracting, tedious and time-consuming, and the resulting arrangements foster confusion.

The SHriMP [17] multi-perspective approach to program comprehension and navigation is related, but its implementation does not provide multiple, simultaneous source file views. Its intent is to provide many different perspectives at different levels of program abstractions, not to effectively expose large amounts of code. The NavTracks project [16] supports the browsing experience by recommending related files to developers, based on browsing history. Our approach depends instead on program structure information and additional display space to show related files near each other. The SeeSoft approach [2] is also related in that it employs large displays to aid program comprehension based on the display of code. However, its goal is to convey high-level information about the code by representing it as colored lines, not to allow users to browse the code.

The Elastic Windows placement system attempted to lay out windows using knowledge of the underlying structure of the items on display [7] and the effect was evaluated [13]. The system demonstrated improved interaction time and was even translated to display hypertext. The window placement strategy proposed here draws from this approach. Our window placement algorithm relies on the program structure to decide the optimal placement of each window. As in the case of Elastic Windows, we hope this will improve task completion time.

PLACEMENT STRATEGIES

Placement strategies for large displays address how to position new windows, possibly taking into account how their contents relate to already-open windows. The problem of window placement and its effects only becomes interesting on large displays, and may become more interesting as displays become larger. Because there are a limited number of positions for new windows on small displays, the initial placement may not matter much. The user is condemned to jarring context switches when changing focus from one overlapping window to another anyway. A 2560x1600 display, however, can easily show 6-8 source code windows at once with minimal overlap.

Many programs (or their underlying window manager) simply cascade new windows atop the window of origin. This behavior may be a vestige of the belief that the user will be working on a screen that can barely show more than one window at a time. In order to see both the origin and destination window, the user needs to immediately move the window, possibly into a completely free space where the window system could have placed it. This approach does, however, respect locality [14]; the new window is always placed near (atop, in fact) the prior area of focus. Such placement strategies are common in popular development environments and web browsers.

Space-filling window placement systems place new windows into available free space on the screen. Such a system is implemented in the Gnome desktop environment, and is used by applications that do not explicitly specify new window positions. When that system cannot find adequate free space on the display, it positions the new window at the top left of the display. Subsequent new windows are cascaded from that arbitrary position, as shown in Figure 1. The system first chooses free space over locality, but has an advantage over simple cascading in that, at least while the screen is relatively unfilled, the user is able to see both the origin and destination hypertexts.

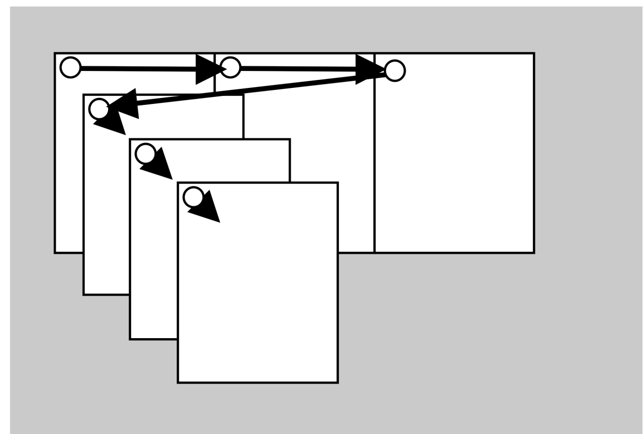

Figure 1 - Space-filling strategy, reverting to cascading when space is not available.

New Strategy.

We implemented a window placement strategy dubbed locality+splay specifically for the task of browsing source code. The strategy chooses position based on the type of link being followed, attempting to layout the source code on-screen in a manner roughly representative of the program call graph. When a user is following execution forward by opening function implementations, windows open rightward across the screen as new source files are encountered. Shifting focus rightward across windows thus creates a sense of being deeper in the program call graph. To track execution backward, as might happen when determining where a variable of interest was allocated, the user can change shift focus leftward across the windows. Windows containing declarations and reference listings are presented above and below the current window as they provide information that supports comprehension of its code.

To implement this strategy, the Source Navigator program version 5.2b2 was modified. Source Navigator implements 3 primitive operations for hypertext source code browsing: finding a function implementation, finding a type/structure declaration, and finding a list of references (calls) to a function.

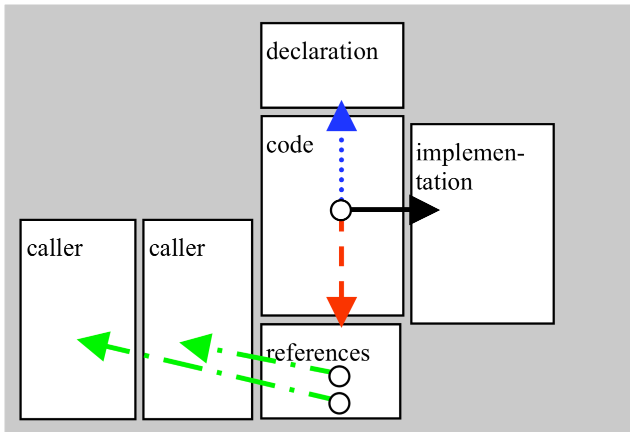

Figure 2 – Locality+splay strategy. From the center code window, a new window can be opened to the right, above, or below, depending on the type of link requested. Requests for call-sites from a reference listing open to the left.

The different placement actions are illustrated in Figure 2. When opening a hyperlink to request a function implementation (shown with solid black arrow), the new window will appear directly to the right of the current window. This allows the user to see the implementation along with the code in which it was used. Additional requests for implementations from the same source window will splay outward to the right (not shown).

Information that supports the current source window, such as a declaration or reference listing, appears in the same column on-screen. When requesting a datatype or function declaration (shown with dotted blue arrow), the new source file (typically a header) will appear higher on the screen than the current window. When choosing to request a list of references to a function (shown with dashed red arrow), the reference listing window will appear below the current window. Clicking on successive items in the reference window will splay them to the left of the reference window, shown with alternating-dash green arrows. The ability to splay the windows should allow a user to easily compare source code from many call sites side-by-side, which would be useful in identifying patterns in code. Alternatively, if browsing a single code path, a user can explore higher in the call graph by moving leftward across the screen by summoning call sites.

This layout attempts to honor locality, by placing new windows near their origin hyperlink when possible. It tries to avoid overlap by not typically placing the new window atop its hyperlink origin or in any set arbitrary position. Of course, screen space is not infinite; when more windows are summoned, the system will place windows against the edges of the screen; if a window already exists in that position then overlap via cascading remains the fallback strategy. Overlap also occurs if, for example, function implementations are requested from two mostly-overlapping windows. Both new implementation windows would be overlapped, in much the same way as their windows of origin. Another example of overlap occurs because a function can have multiple callers. This occurs when a user opens an implementation window, requests the other callers for that implementation in a reference listing, and then opens one. Assuming that the just-opened caller was in a different file from the original, the two windows would then overlap in approximately the same column on-screen.

USER STUDY

A user study was conducted to test the hypotheses that the window placement strategy could affect window manipulation effort and browsing task completion time. The locality+splay strategy was compared with the space-filling strategy. Despite its widespread use, the simple cascading strategy was not considered because it would not be competitive in terms of window manipulation effort; nearly every window would require adjustment if the user wanted to make use of the large display area at all. In the study, window manipulation effort was measured as the mouse dragging distance traversed while making window position adjustments.

Tasks.

The source code to be browsed was an older version of GNU Wget, a command-line program designed to accept FTP and HTTP URL's as arguments and then retrieve them. Wget was chosen because all users could be expected to either know its purpose or be trivially informed of it. Wget's code is well-commented and well-organized, and does not use function pointers or any particularly opaque sequences of pointer arithmetic or dereferencing. Although some participants were certainly capable of parsing obfuscated code, the point of the experiment was to observe their browsing of code in a limited amount of time.

In order to test participants' browsing of program source code, a questionnaire was created. The questionnaire consists of 18 multiple-choice questions a programmer would browse source code to answer. A multiple-choice format was chosen to remove the need for users to spend time entering answers. Because browsing is generally considered an exploratory strategy with no fixed endpoint, the questionnaire had to be carefully designed to prompt natural exploration of the code, posing questions that the programmers might pose to themselves. Some of the easier questions involved little more than applying reading comprehension. For example (with answers provided for the benefit of the reader):

- *Expand main.c to open the function main(). What function does main() call in order to retrieve a*

URL entered from the command line? (A: retrieve_url)

Other questions built off the results of these and were designed to prompt deeper browsing of the source code; the following question involves the participant going 3 function calls deep:

- *Assume the user has entered an HTTP URL on the command line. Starting from retrieve_url(), follow the path of execution until a network socket connection is made. In what function does this happen? (A: make_connection)*

An exploration question such as the previous one would be followed by questions involving the data passed down the particular control path:

- *In gethttp's call to make_connection(), the second argument is the hostname. What is the name of this argument in the call, and what is the name inside the called function? (A: u->host/hostname)*
- *The string you identified in the call to make_connection, u->host/hostname, is an element of a urlinfo struct passed in from previous callers. What function in the path you explored took care of allocating memory for this struct? (A: newurl, called from retrieve_url)*

Because it often requires familiarity with the source code, other questions involved identifying where aspect-oriented refactoring could be applied. This was accomplished by asking the users to identify whether patterns existed in the code such that they could be considered join points [8] for advice:

- *After every call to make_connection, a switch statement with other code is run. What does this code do, and is it identical after every call to make_connection? (A: Error processing and logging; it's NOT identical.)*

Design.
For the experiment, half the participants used the normal Source Navigator browser which makes use of the underlying window manager's space-filling placement system while the other half used the locality+splay version. A between-subjects design was chosen due to concerns about participants' familiarity with the source code dominating any improved performance during the second half of the session. Although this concern could be mitigated by using a different target source program for the second half of the questionnaire, time limits made this impractical. Another possibility would be to alternate window placement systems during the questionnaire, but it was felt this would prove too unpredictable and potentially annoying or jarring to the users.

Protocol.
A pre-questionnaire was used to verify that the participants had adequate knowledge of the target language (C) to perform the browsing tasks. When asked about their programming language experience, all participants indicated at least 1 year of experience with C. When asked about memory management in C, all were able to describe the malloc and free functions. The pre-questionnaire also verified that none of the participants had previously seen the Wget source code.

Users were seated at the test system and invited to make ergonomic changes. Each participant was given a verbal description of the Wget program, as well as a printed summary of its operation, and given time to read it carefully. Each participant was then directed to use the 3 primary hypertext operations available for C programs in the Source Navigator browsing tool (Find Implementation, Find Declaration, References). Next, users completed a 6-question warm-up questionnaire (presented by an on-screen program) targeting the vsftpd program to ensure their familiarity with Source Navigator, as well as the directed-browsing style questions.

Next, each participant began the questionnaire. Once a question or task was complete, it could not be revisited. Participants were urged to advance to the next question as soon as they felt confident in their answer. Although time was recorded, no timer was present on the screen; all users completed all questions. Only one user incorrectly answered one question.

A survey to gather feedback was taken immediately following the questionnaire. The total time for the session was approximately 1 hour.

Participants.
Sixteen participants were selected from a pool of professional programmers at the Department of Defense. The programmers regularly perform software maintenance and analysis tasks in which browsing source code plays a major part. All had experience with hypertext-based source code exploration tools.

Apparatus.
The same display, input devices, and computer were used for all participants. The display was an Apple 30" Cinema Display at its native resolution of 2560x1600. The input devices were a standard Dell USB keyboard and optical scrolling mouse, and the computer was a Dell XPS Gen 4 running a default installation of the Fedora Core 4 Linux distribution with the Gnome desktop environment.

RESULTS

The results for task completion time and window manipulation effort are presented, as well as further investigation into the results.

Task Completion Time.
The two groups' average times to complete the questionnaire were calculated and a t-test was performed to check for statistical significance. The means differed in favor of the locality+splay placement as shown in Figure 3,

but the results were far from significant: (t(14)=0.81, p=0.43). Cohen's *d* for the two groups is 0.61, which is considered a medium effect size. This suggests it may be possible to reach significance with more participants.

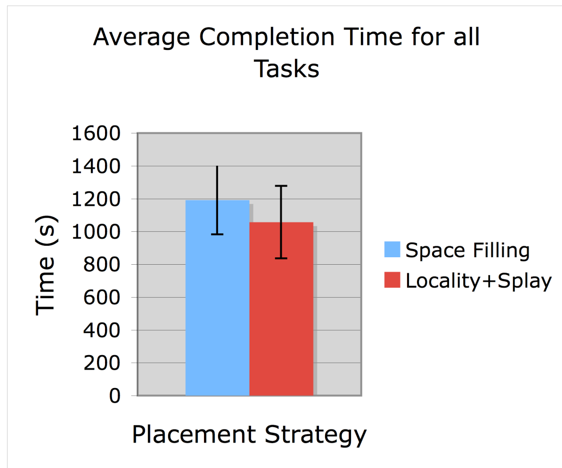

Figure 3 – Average task times and confidence intervals (as error bars) for the two groups.

Window Manipulation Results.
The total mouse dragging distance involved in window placement operations was captured for all tasks. This distance will be used as a measure of the amount of mouse effort required. As shown in Figure 4, the means differed to favor the space-filling strategy for total distance traversed, but again the difference was far from significant (t(14)=-0.32, p=0.75). Cohen's *d* for the two groups is 0.24, a smaller effect size than for the task completion time.

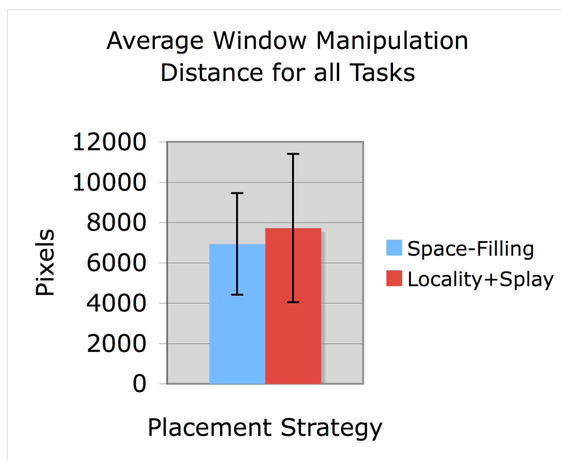

Figure 4 – Average window manipulation distance and confidence intervals (as error bars) for the two groups.

It is worth noting that one participant in the locality+context group had a manipulation distance of 17,684 pixels, about 2 standard deviations from the mean of 7725 pixels. If this user were eliminated, the window manipulation mean would change to favor the locality+splay strategy, (6303 vs. 6938 pixels). While this difference is now in the same direction as the time analysis, the difference is still not significant (t(13)=0.29, p=0.77).

In conclusion, the data collected do not support the original hypotheses regarding manipulation effort and completion time. To investigate why, we visualized the window manipulations.

Visualizing Window Manipulations.
Logs of the window manipulations were translated into window trace diagrams to show users' behavior. These diagrams, along with a post-session questionnaire and a playback system capable of displaying the window situation over time, provided information about users' interaction with the window placement strategy. In a window trace diagram, the background represents the display surface. An open circle represents the center location of where a window was opened. Its subsequent path (if any) is represented by a line, and its position when closed (either by the user or automatically at the end of the session) is represented by a circle with an 'X.'

The window traces in Figure 5 were created from participants using the space-filling window placement strategy. The top user represents a "successful" user (in terms of interaction with the window placement strategy). Successful is defined here as characterized by little window movement, fewer than 3 movements for the entire session. Surprisingly, the number of windows in the vicinity of the top-left of the screen of the successful space-filling strategy user did not motivate the user to move them. Screenshots and a playback system showed that many of these windows were open at the same time, cascaded and heavily overlapped. The bottom participant in Figure 5, as well as all other diagrams for the space-filling participants, is heavily characterized by movement away from the arbitrary top-left position. We will call this position a "hotspot" in the window trace diagram. A "hotspot" is defined here as a 100x100 pixel region where more than 3 windows are placed and then subsequently moved. Identifying such patterns can demonstrate how a window placement strategy could be improved.

The window traces in Figure 6 were created from users of the locality+splay system. The top trace in Figure 6 represents a successful user of the locality+splay system. In this trace, the entire screen is effectively used during the browsing activity. The bottom trace in Figure 6 represents users for whom the window placement system was less successful. In this case, the window movement is characterized by a hotspot in the middle-right of the screen. Another notable behavior in this trace is the movement of other windows toward the center of the screen, which suggests a tendency to use only the center of the large display.

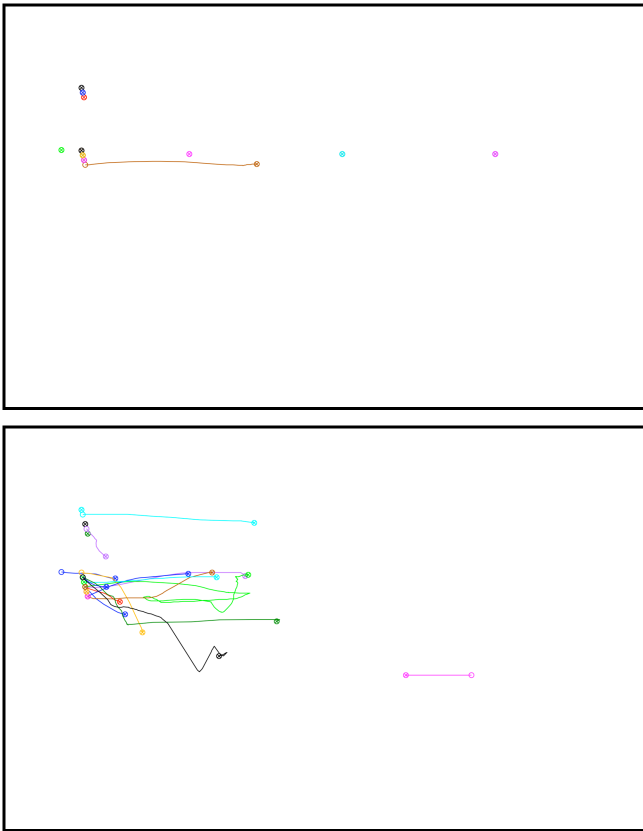

Figure 5 – Window trace diagrams for two participants using space-filling placement strategy.

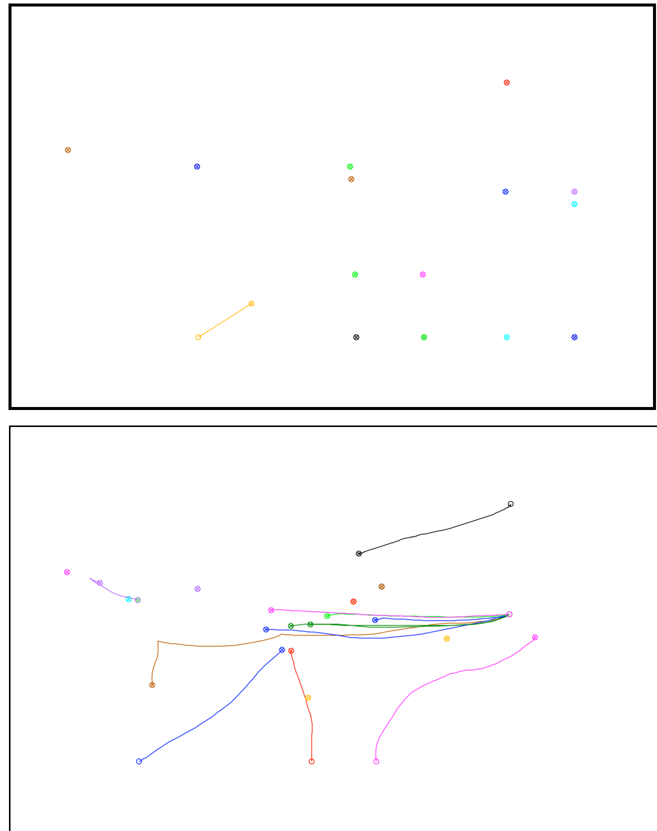

Figure 6 – Window trace diagrams for two participants using locality+splay placement strategy.

Totals for these characterizations are shown in Figure 7. The “Other” category indicates that the window manager was not successful for the user, but it is not yet possible to characterize the behavior.

Feedback. Fourteen of the 16 participants felt that the questions were similar to ones they might ask themselves while browsing code, which argues for the validity of the directed-browsing technique. All participants except for 2 felt that using the large display made answering the questions easier. Participants were also asked about how conveniently they felt windows were placed on the large display. Half the participants who used the space-filling strategy described the arbitrary placement behavior at the top-left of the screen, and said that it was inconvenient. Of 16 participants, 3 mentioned specifically that they would *always* prefer that the window of origin and the new window never overlap, a criticism of the cascading strategy that is still employed in some cases.

DISCUSSION

The results here suggest that the window placement system has no effect on code browsing performance. However, the strategies compared here may very well perform better than most of those used today. The common cascading strategy was not used in the study, as it would not have been competitive in terms of window manipulation; in order to make use of the large display, every window would need to

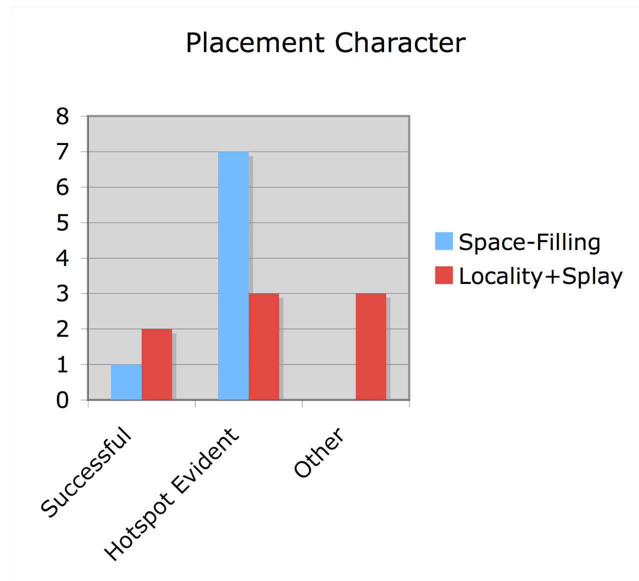

Figure 7 – Character of placement strategy evident in window trace diagrams.

be moved from atop its original position into free space. Whether such a system would have had an effect on task completion time remains unknown. The space-filling strategy from the Gnome environment considers all windows on-screen when determining whether it can place a window in free space, not simply the current program's windows. If an unused documentation, IM, or e-mail window were left open during the experiment, the space-filling strategy would have reverted to placement at the top-left sooner. Given the window manipulation associated with this hotspot, it seems likely that such a change would have affected its results negatively.

Some users' tendency to use only a portion of the display during the sessions appeared responsible for mouse movement. These users chose to move the windows to their area of focus, instead of moving their head to focus on a different part of the screen. In order to accommodate this behavior, it may be worth exploring the idea of shifting all other source browsing windows so that a new window can be placed inside or near the user's area of focus. Any windows shifted outside the focus area may still provide a useful sense of context. Another potential solution to this problem is to simply not place windows outside the area of focus.

The problem of overlap still exists in the locality+splay placement strategy. One overlap situation occurs at the edge of the screen. This can be seen when a user requests a function implementation from a window that is already on the right edge of the screen. One way to address this problem is to shift all other windows in order to make space for the new window. This could be done in combination with reducing the size of the most distant window to avoid pushing any windows off-screen. Another situation in which overlap occurs is in dealing with multiple callers to the same implementation. A potential solution to this problem is to place the new window in a column to the left of the implementation, but where it would overlap least.

FUTURE WORK

The data collected could reveal more information about participants' motivations for window placement. The extent to which participants tended to arrange windows in a way that reflected underlying program structure, simply sought available screen space, or respected locality remains open for analysis. Simpler reasons, such as the placement of a new window directly atop its hyperlink of origin, may also be responsible. A more extensive study might also be able to further characterize browsing styles. The hotspot evident with the locality+splay system was unexpected and any future iteration of such a placement strategy should address it.

The variation that exists in browsing speed between users suggests that a within-subjects design may be preferable for any future experiments. Major problems with this approach would be users' gaining familiarity with the code and the browsing environment during the experiment, which would favor the second system tested. In order to mitigate these dangers, isomorphic browsing tasks could be identified on two different pieces of software, and the participants could be counterbalanced. Alternatively, more users could be recruited for a between-subjects design.

Other basic questions remain unanswered about how the user interface serves the software developer who needs to browse code. A comparison could be run between browsing in tabbed source code windows (in Eclipse and Visual Studio) and multiple-window systems. Although other works have quantified the benefits of large displays over small displays for office tasks [4], no study has compared the performance of software developers browsing code on small and large displays.

CONCLUSION

A window placement strategy was implemented to assist programmers browsing source code on large displays. In a study, the users of the system demonstrated a reduced mean time to complete browsing tasks, but not in a statistically significant way. Window arrangement effort was slightly increased on average for the new placement system, but also not in a significant way. Further analysis revealed differences in users' window arrangement tendencies that, for a study of this size, created variations that overpowered any window manipulation effects of the placement system. More study is needed to determine how a window placement strategy might support the source code browsing activity.

ACKNOWLEDGEMENTS

The author wishes to thank the project participants for generously donating their time and mental effort. The author also wishes to thank François Guimbretière for his guidance.

REFERENCES

1. Anderson, P. and Zarins, M. The CodeSurfer Software Understanding Platform. In *Proc. of IEEE IWPC'05*.
2. Ball, T. and Eick, S.G. Software Visualization in the Large. IEEE Computer, Vol 29, No. 4, April 1996.
3. Cox, A. and Collard, M. Working Session: Textual Views of Source Code to Support Comprehenions. In Proc. of *IEEE IWPC'05*.
4. Czerwinski, M., Smith, G. et al. Toward Characterizing the Productivity Benefits of Very Large Displays. INTERACT '03.
5. Eick, S.G., Steffen, J.L., and Sumner, E.E. SeeSoft–A Tool for Visualizing Line Oriented Software Statistics. In *IEEE Trans. on Software Engineering*, Vol. 18, N. 11, Nov. 1992.
6. Hassan, A. and Holt, R. Using Development History Sticky Notes to Understand Software Architecture. In *Proc. of IEEE IWPC'04*.

7. Kandogan, E. and Schneiderman, B. Elastic Windows: A Hierarchical Multi-Window World-Wide Web Browser. In *Proc. of ACM UIST'97*. (also in CHI'98)
8. Kiczales, G. et al. Aspect-Oriented Programming. In *Proc. of European Conference on Object-Oriented Programming*, 1997.
9. Koskinen, J. HyperSoft System: Tool Demonstration and Use Example. In *Proc. of IEEE IWPC'05*.
10. Raskino, M. Bigger and Better Displays Will Boost Productivity at Last. Gartner Research. April 1, 2005.
11. Robertson, G. et al. Scalable Fabric: Flexible Task Management. In ACM *Proc. of the working conference on Advanced visual interfaces '04*.
12. Robillard, M., Coelho, W., and Murphy, G.C. How effective developers investigate source code: an exploratory study. In *IEEE Trans. on Software Engineering*, Vol. 30, pp. 889-903, 2004.
13. Schneiderman, B. and Kandogan, E. Elastic Windows: Evaluation of Multi-Window Operations. CHI'97.
14. Shneiderman, B. Designing the User Interface. Addison Wesley Publishers, 1997.
15. Sim, S.E., Clarke, C.L.A., Holt, R.C., and Cox, A.M. Browsing and Searching Software Architectures. In *Proc. IEEE ICSM'99*.
16. Singer, J., Elves, R., and Storey, M.-A. NavTracks: Supporting Navigation in Software. In *Proc. of IEEE IWPC'05*.
17. Wu, J. and Storey, M.-A. A Multi-Perspective Software Visualization Environment. In *Proc. of CASCON'2000*, November 2000.