An Experimental Evaluation of Continuous Semantic Zooming in Program Visualization

Kenneth L. Summers*
Center for High Performance Computing
The University of New Mexico

Timothy E. Goldsmith†
Department of Psychology
The University of New Mexico

Steve Kubica‡
Khoral, Inc.

Thomas P. Caudell§
Department of Electrical and Computer Engineering
The University of New Mexico

Abstract

This paper presents the results of an experiment aimed at investigating how different methods of viewing visual programs affect users' understanding. The first two methods used traditional flat and semantic zooming models of program representation; the third is a new representation that uses semantic zooming combined with blending and proximity. The results of several search tasks performed by approximately 80 participants showed that the new method resulted in both faster and more accurate searches than the other methods.

CR Categories: H.1.2 [User/Machine Systems]: User/Machine Systems— [H.5.2]: User Interfaces—Graphical user interfaces (GUI), Theory and methods D.2.6 [Programming Environments]: Graphical environments— [D.1.7]: Visual Programming

Keywords: Program visualization, Human subjects testing, Visual program languages

1 Introduction

Programs are complex and often difficult to understand. This difficulty hinders the creation of complex programs, as well as the understanding of existing programs, as defined by speed and accuracy on a visual search test. Although professional programmers might be able to examine raw code and decipher its essence, inexperienced programmers such as scientists and engineers, who only program to better understand their area of interest, have a need for tools to clarify the meaning of code.

In this study, we examined how three different methods of visualizing programs affected users' understanding of the programs. As a first step in exploring new visual representations for programs, this study started with an already available abstraction of standard computer code: visual programs. Traditionally visual programs have been represented as large, flat layouts of program elements, which consist of functional icons and program flow connections. See Figure 1. Because each component in the visual program requires space for its visual representation, real estate is at a premium. Thus large and complex visual programs can take up a large amount of space. Traversing the components and understanding the relationships among program elements is difficult for large visual programs.

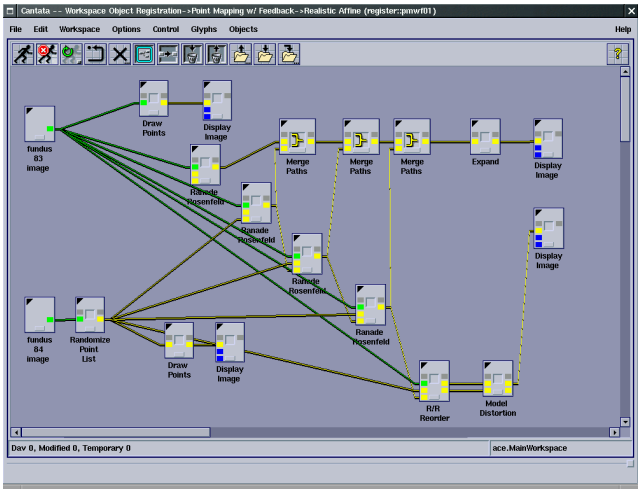

Figure 1: A crowded, difficult to understand visual program workspace. Screen real estate is very valuable in a visual program, prompting the use of methods to compress programs into less space.

To alleviate this problem, some visual programming environments have implemented procedural programming, a multiresolution technique. Similar to their use in textual programming languages, procedures allow program elements to be grouped by function, increasing readability and encapsulating complex sections of a program. Figure 2 shows the same visual program as Figure 1 with a procedure and its contents. This method, defined by Bederson and Hollan as *semantic zooming* [Bederson et al. 1996; Muthukumarasamy and Stasko 1995], improves functional grouping and saves real estate. By encapsulating related program elements this method can reasonably be expected to improve overall program understanding.

The principle behind semantic zooming (SZ) is that, as the viewpoint zooms on an area the details not only become more distinct, as would be expected by virtue of being "closer," but the representation changes as well. For example the procedure shown in the upper left window of Figure 2, has been opened, or zoomed in the lower right picture. Although this example was arranged carefully so as not to obscure any of the top level program by the contents of

*e-mail: summers@hpc.unm.edu
†e-mail: gold@unm.edu
‡e-mail: kubica@khoral.com
§e-mail:tpc@eece.unm.edu

IEEE Symposium on Information Visualization 2003,
October 19-21, 2003, Seattle, Washington, USA
0-7803-8154-8/03/$17.00 ©2003 IEEE

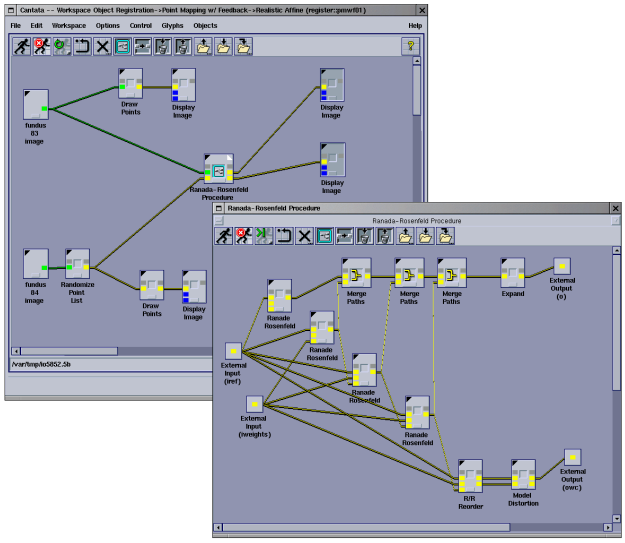

Figure 2: Top: The same program as shown in Figure 1 with some of the program elements encapsulated into a procedure. Bottom: The contents of the procedure.

the procedure, it is evident that more encapsulated elements in the procedure, or more open procedures would obscure representation.

In textual programs, procedures often completely obscure the calling code, by virtue of being large and usually in different areas of the program or in another file altogether. Once again, while splitting the editor window may allow viewing of multiple levels simultaneously, this is severely limited by available screen area.

The principal drawback, therefore, with this method is that, when viewing the top level program, the contents of the procedure are hidden. Likewise, when viewing the contents of the procedure, the top level structure is hidden.

Many methods have been applied to solve this problem. Examples include *SHriMP* [Storey et al. 1997] and *Continuous Zoom* [Dill et al. 1994]. SHriMP is a system integrating pan+zoom and fisheye-view visualization approaches to explore nested graphs. Continuous Zoom incorporates a fisheye-view method with multiple focus points and a smooth transitions.

This paper introduces the method called *continuous semantic zooming* (CSZ) [Summers 2002], which differs from these approaches in that it is a technique for viewing hierarchical data which could, but is not required (and as presented here, does not) use distortion techniques. CSZ could be used as an ancillary method to other systems, or can be integrated into new, stand-alone systems.

The remainder of this paper describes the method, and reports the results of a study with human subjects comparing CSZ to both traditional semantic zooming representations and flat representations.

2 Continuous Semantic Zooming

The main drawback of traditional multiresolution techniques for grouping, such as procedures, both in visual programs and text programs, is that the different level representations are distinct and separate. Text programs, such as those written in C, often use procedures and subroutines to group and encapsulate functionally related code. In these programs the procedure definition and the procedure call are in different places in the file or in separate files altogether. Visual programs also encapsulate procedures in ways that obscure other levels of detail, such as having to open another window to display procedure details, often hiding the original program structure. In this traditional form of semantic zooming the transition between views of the program is typically abrupt and often disorienting to the user [Sindre et al. 1993].

In contrast, continuous semantic zooming combines semantic zooming and blending or morphing to allow the user to view the details of a procedure while still viewing the surrounding higher level constructs. This combination also allows a smooth transition between view levels, helping the user maintain the mental model of the relationships between the different representations. CSZ uses viewpoint proximity to trigger a viewing mechanism, such as transparency [Rekimoto and Green 1993], lenses [Robertson and Mackinlay 1993; Rao and Card 1994; Fishkin and Stone 1995], or environment distortion [Leung and Apperley 1994; Carpendale et al. 1997], which exposes the details encapsulated in a procedure while allowing the higher level constructs to remain visible in the viewing area.

Figure 3 shows an example of the progression of continuous semantic zooming using transparency. As the user comes closer to the procedure object, the contents of the procedure appear. As the user moves closer still, the procedure object fades even further, while the lower level contents become even more visible.

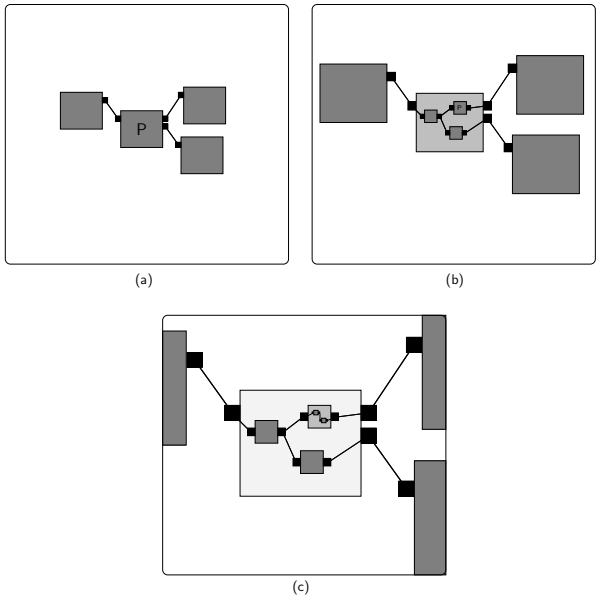

Figure 3: Three successive views illustrating the concept of continuous semantic zooming: (a) shows a top level view; (b) shows a top level view, when CSZ has been triggered and details of the procedure are visible; (c) shows a detailed view of the procedure, with other, higher level program elements still visible.

Contrast this example with an illustration of the same program in Figure 4. To assist the overall comprehension of programs, a method is needed for simultaneous viewing both a procedure definition and the context in which it is called. CSZ allows a gradual transition between resolutions of the procedure, thus minimizing user disorientation and confusion. Additionally CSZ allows the user to view both the procedure and the context in which it is called with minimal obscuration. The primary mechanism used by CSZ to trigger this transition is proximity. The closer one comes to an encapsulated object the more detailed it becomes. Since closer also implies larger (due to perspective) the details of the proximate object are visible, while the distance to other objects remains approximately the same and hence so does their relative detail.

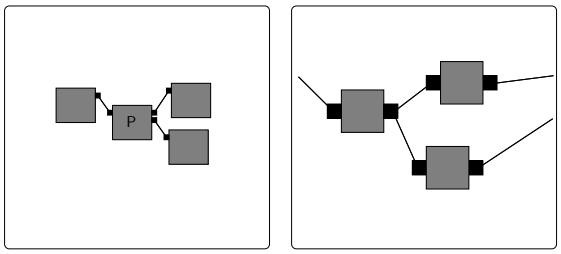

Figure 4: Two successive views illustrating the concept of semantic zooming: (a) shows a top level view; (b) shows the detailed view of the procedure. Notice that, unlike CSZ, SZ does not permit intermediate views. Likewise, the second, encapsulated procedure's contents remain hidden.

This multilevel view allows the user to see some details while viewing higher level structure, and to see some of the higher level structure while viewing procedure details.

3 Evaluating CSZ

We compared CSZ with two other methods of program representation: flat and semantic zooming (SZ).

The flat program representation contained no procedures. It represented the same program as the other two but with all procedures exploded into their constituent program elements. The logical grouping of the program was maintained, i.e., sections of related code, that might have been in a procedure, were grouped together in the flat program. See Figure 5.

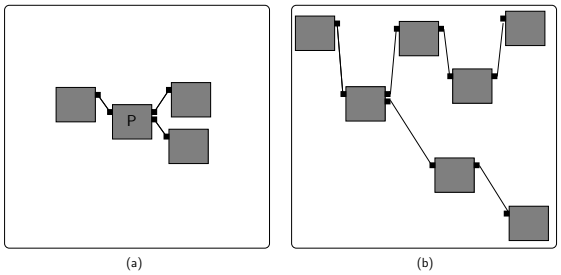

Figure 5: Two views, using the hierarchical representation presented in Figures 3 and 4, showing, in (a), the hierarchical view with procedures and, in (b), the corresponding flat view where the procedures have all been "expanded" in place.

Three empirical studies were run, two of which were pilot studies. In each study, subjects were asked to find program elements that were responsible for creating specific output results. Time to find the element and accuracy were the primary measures of performance. We assumed that these performance measures would be related to program understandability. We argue that, in studies of this type, improved performance is most directly related to speed and accuracy.

3.1 SGPL

For these experiments a simple graphics programming language (SGPL), inspired by the Logo [Papert 1980] graphics language, was developed. See Figure 6. In this language each program element is

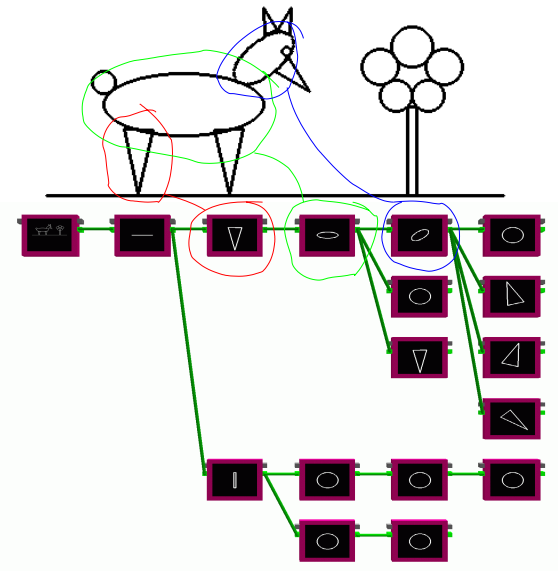

Figure 6: Picture of an animal and the program that produced it.

capable of drawing one polygon. The available polygons are lines, triangles, rectangles, and ovals. The orientation, size, and aspect ratio of each polygon is adjustable. The polygon's relationships to one another are determined by the connections between program elements. The polygons produced by any two connected program elements must, by definition, touch each other. So the leg of the animal in Figure 6 touches the ground, the body the leg, and the head the body. Note that this property is not commutative. The animal's other leg touches the ground, but the program element that draws it is not connected to the program element that draws the ground, only to the body.

The program designer has control of the position, orientation, size, aspect ratio, and type of the polygon. The representation on the face of each program element, however, only displays the type, orientation, and aspect ratio of the polygon, as seen in Figure 6.

4 Pilot Studies

Two pilot studies were run to calibrate the methods. The studies compared traditional SZ and a flat representation of an SGPL program. We hypothesized that hierarchical representation of data, especially programs, would lead to increased understanding of those data, as represented by decreased time and increased accuracy in finding specific data. The pilot studies were, therefore, designed to compare time to target and accuracy of traditional SZ and flat representations of an SGPL program. The pilot studies were used to calibrate the methods for future studies. An informal sample of 16 subjects was drawn from friends and colleagues.

A static, 2D representation of an SGPL program was presented to subjects using Microsoft PowerPoint. In the flat version all program elements were at the same level, spread across several pages. Different pages could be accessed by clicking on arrow icons in the corners of each page, thereby simulating scrolling the screen. The SZ representation allowed users to click on procedures, replacing the page with a new page containing the procedure program elements. A "back" button allowed the user to back out of procedures.

One polygon would be highlighted on a paper copy of the picture "produced" by the program on-screen, and the subject asked to find the program element that drew the highlighted polygon. The sub-

jects' response time from when the picture was revealed to calling out a program element number was measured. The test consisted of ten iterations of this procedure, each time asking the subject to locate a different highlighted polygon.

One observation made during the first pilot study was that subjects would often fall back on pure pattern matching by simply looking for a polygon that matched the target polygon and ignore the structure of the program. While this is a useful method to find objects, it is seldom useful for examining program code because text representation is quite uniform. The program structure is usually the main clue as to what a particular code fragment does.

Building on these experiences, the second pilot study increased overall complexity and avoided easily recognizable features. The second pilot study was run with the same parameters and a revised, more complex picture. A more complex picture, consequently, was produced by a more complex program. The new program/picture combination relied more on similar objects to better simulate the uniform nature of a program (many similar code fragments with slightly differing functionality) and no "unique" elements as were present in the first pilot picture.

Observation of the second pilot study showed an increased reliance on program structure for element identification, and increased search time that helped minimize the effect of administrator, instrumentation, and subject delays. This pilot study also showed that subjects were capable of dealing with significantly more complex programs without becoming confused. It was determined, therefore, that the main study should consist of an even more complex program, with more self-similar features to reduce even further the reliance on visual pattern matching.

The main result of these pilot studies was to show that the experimental methods were sensitive enough to detect significant differences in time and accuracy, and that time and accuracy were valid measures of performance. They both showed that SZ was significantly faster and more accurate than flat. These pilots also pointed out the areas where proper design of the target picture would result in more accurate and representative results.

5 Main Study

The study extended the pilot studies by adding a third dimension, and a third representation: CSZ. While the two pilot studies used static, 2D representations of a program, constructed in Microsoft PowerPoint, the study used a fully functional program displayed in a 3D *virtual environment* (VE). This allowed a natural introduction of the concept of proximity.

The program was an adaptation of the visual programming language Khoros Cantata, by Khoral, Inc. Cantata was modified to export representation and control of the visual program workspace to the VE. The VE could load, start, stop, clear, etc., the workspace, and all images and program element placements were transmitted back to the VE. This allowed the VE to act as an alternate display, alongside Cantata's internally generated X window display.

The SGPL program elements were made with customized Cantata program elements designed to show orientation and aspect ratio of the polygon on the face of the program element. Additional "screen" program elements were created that showed the aggregate contents of either a procedure or the entire program, depending on placement. In this way the finished picture, as well as sub-pictures consisting of the polygons drawn by a procedure were also transmitted to the VE. Figure 6 shows an example of this 3D interface.

The VE used was Flatland [Caudell 2000; Caudell and Summers 2003], a development of the University of New Mexico Department of Electrical and Computer Engineering and the UNM Center for High Performance Computing Visualization Laboratory. Flatland is a general purpose, highly configurable virtual environment which allows run-time loading of user generated objects into a centrally managed scene graph. The Cantata interface is one such object.

The study used a between-subjects design, with each subject tested under only one representation. A total of 60 subjects were randomly assigned to three methods, 20 per method.

The experiment was divided into 14 tasks, similar to the pilot tasks. The subject was shown the output of the program (a completed picture) with one highlighted polygon and asked to find the program element that drew that polygon. Figure 7 shows the picture used for the study. This picture contained 164 polygons, and the corresponding program consisted of approximately 200 elements. Each procedure element consisted of between two and eleven subelements, and the maximum procedure depth was three.

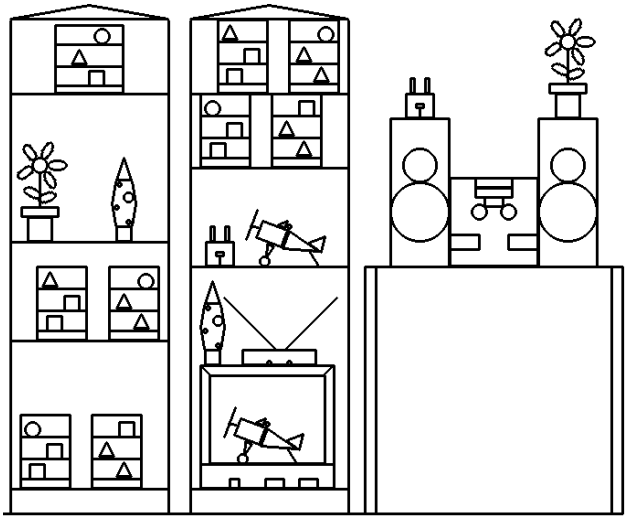

Figure 7: Picture used for the study. Note the large amount of self-symmetry and the multiple identical objects.

Two new features were added to the study picture design.

- Multiple identical objects were included to help eliminate simple pattern matching. Subjects were forced to follow the program structure to pick the correct element.
- Even more complicated picture designs were created, with an emphasis on elements that were self similar. This required that the subject examine the program structure in detail to differentiate between two similar, but not identical, structural elements.

Because the entire program was never completely visible, a method for moving the viewpoint around the program was required. In the 2D case used in the pilot studies this was accomplished by "paging" between different sections of the program using buttons. In the 3D environment of the main study the viewpoint change was continuous, with the viewpoint position controllable from the keyboard.

The flat method was restricted to movement in a plane parallel to the the program layout. Thus the viewpoint was always held a fixed distance from the program elements. With the SZ and CSZ methods movement was allowed perpendicular to the plane of the program. All movement was controlled using the keyboard arrow keys, with the shift key as a modifier. To move in the plane parallel to the

program, the four arrow keys caused movement of the viewpoint in the appropriate direction. For movement perpendicular to the plane of the program the shift key was used to modify the up and down arrow keys into in and out functions, respectively. In all cases the speed of movement was subject controlled by repeated use of the appropriate key. A "stop" key (in this case, the `Home` key) was available to recover from runaway movement.

To "open" a procedure the subject would move closer to the plane of the program. Using the SZ method this was an abrupt transition to view the contents of a procedure, with all further movement restricted to the interior of the procedure (until the user backed out of the procedure again). This restriction emulated the all-or-nothing nature of standard semantic zooming.

Using the CSZ method the transition was continuous, with a procedure gradually fading to transparency the closer the viewpoint came. No restrictions on in-plane movement were used, allowing the user to view the procedure and its neighbors from any position. This animation of continuous motion, with blending effects, meant that more time was required to move into and out of procedures when using CSZ than when using SZ, with the anticipated benefit that the relationships between the different level views would be better understood by the subject. This time difference lies at the crux of the experimental objectives: would the CSZ method increase subject performance sufficiently to offset the extra time required to transition between program levels.

Figure 10 shows a screen capture from the study using the CSZ method. Notice that the central program element is nearly transparent. This allows the user to see the procedure's component parts. At the same time some of the surrounding procedures are slightly transparent, and their component parts can be indistinctly seen.

In all three methods the mouse was used to point at the desired program element and select it, once it was on the screen. Upon selection the time and the selected program element were automatically recorded, and the next task loaded.

5.1 The Experiment

Sixty five subjects participated in the study. Subjects were recruited from second semester programming classes, thereby fulfilling the inexperienced user requirement.

To speed the data collection process, four machines were used. Subjects were randomly assigned to the three conditions, and were assigned to machines in order of appearance.

The subjects were given five practice tasks, detailed written instructions, five more practice tasks, then performed the 14 tasks of the study.

6 Results

Of the 65 participating subjects, the results from three subjects were dropped from the study, because of incomplete runs (two) and because of outliers (one). Two more subjects were randomly selected and dropped to equalize the number of subjects used for each method.

Figure 8 shows the mean response times and the number of correct answers for each method. The times for CSZ and SZ were essentially the same, with the flat method taking a little over 40% longer. The flat and CSZ methods, however, were more accurate than SZ.

Figure 9 shows a scatter plot of time vs. number correct. Each point shows the mean time and number correct for each subject. The method is represented by point shape and color. As can be seen, the CSZ points tend to group in the low time, high accuracy corner of the graph, with one outlier. The SZ points are well grouped in the low time, low to middle accuracy section of the graph. The flat

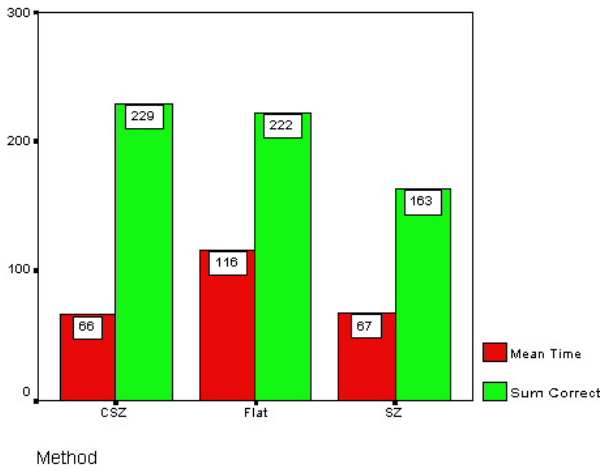

Figure 8: Mean time to completion and number correct for each method. The vertical axis represents time or number correct for each set of bars, appropriately.

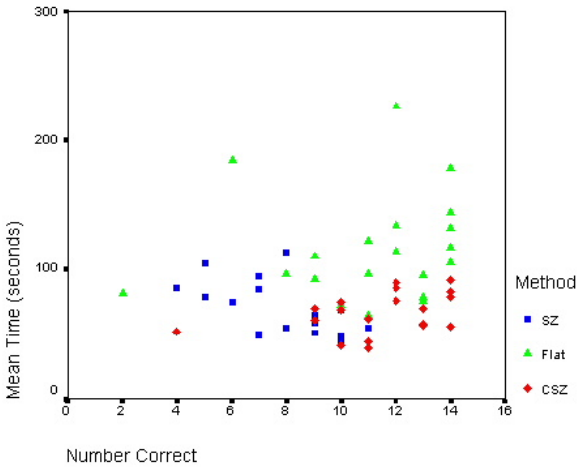

Figure 9: Scatter plot of mean time to completion and number correct for each subject. The method used by each subject is indicated by bullet type and color.

points tend to group at the high accuracy end, with large variations in time and several outliers.

Figure 11 abstracts the data one step further by depicting the mean time and accuracy of all subjects shown in Figure 9. This figure clearly illustrates that the CSZ method has the speed of the SZ method and the accuracy of the flat method.

A one-way analysis of variance (ANOVA) was conducted to evaluate differences in time to respond among the three conditions. The ANOVA was significant, $F(2,57) = 20.43, p < .001$. Follow-up tests were conducted to evaluate pairwise differences among the means. A Tukey's adjustment for post hoc comparisons showed that CSZ was significantly faster than flat ($p < .05$) and hierarchical was significantly faster than flat ($p < .05$), but no differences between CSZ and hierarchical.

A one-way ANOVA performed on accuracy showed a significant effect, $F(2,57) = 9.88, p < .001$. Follow-up tests using Tukey's

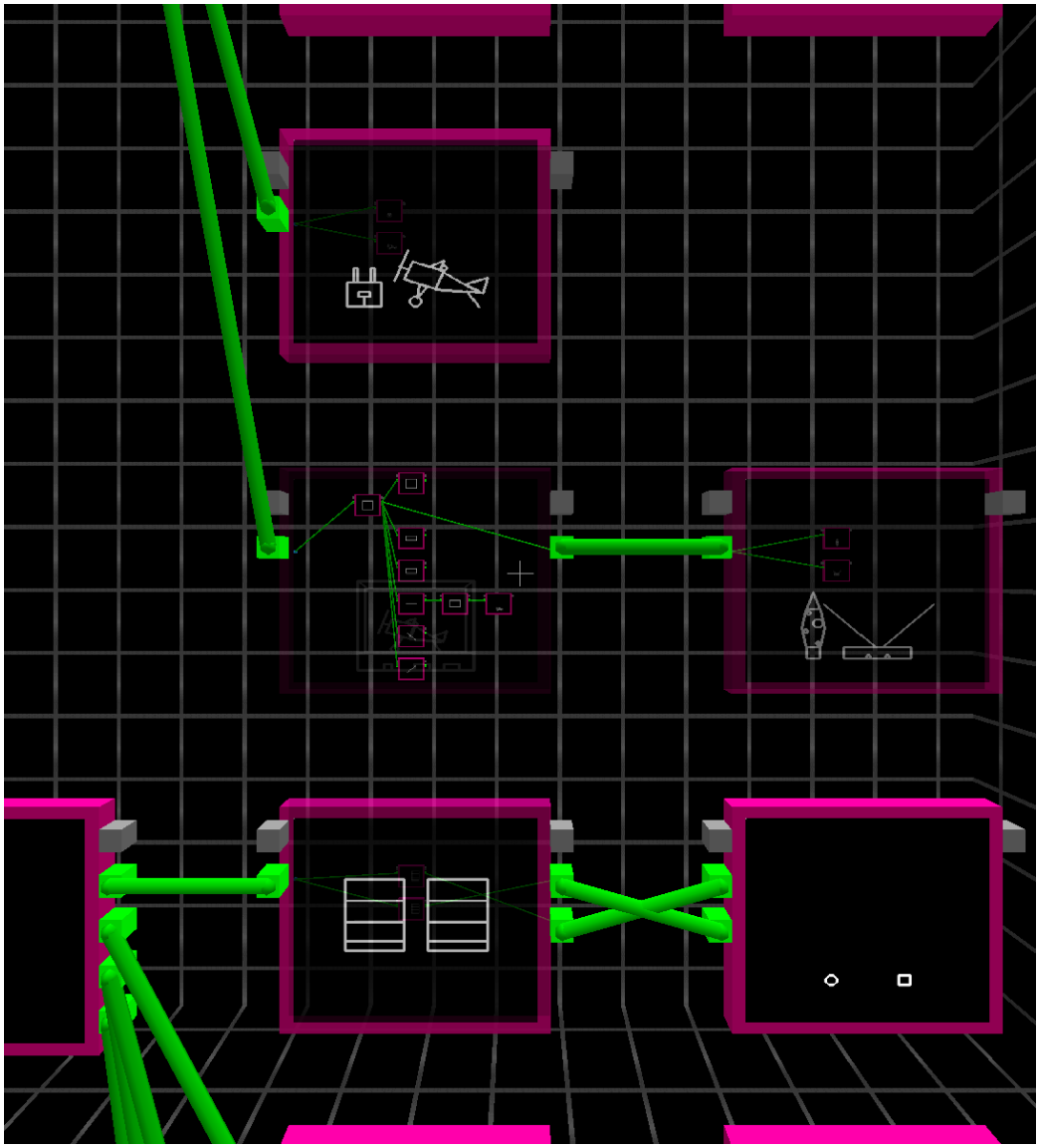

Figure 10: Screen capture from a run of the study. The central program element is almost completely transparent and it's component parts can be seen, while surrounding program elements are just beginning to fade.

adjustment showed that CSZ was significantly more accurate than hierarchical ($p < .05$) and flat was significantly more accurate than hierarchical ($p < .05$); no difference was found between CSZ and flat.

In order to consolidate time and accuracy onto a single measurement, Z-scores [Hays 1988] of both time and inaccuracy (to make both measures more desirable as the value decreases), were averaged together. Figure 12 shows a plot of these mean Z-scores. Note that the more negative the score, the higher the value, since the goal is a short time to completion and a small inaccuracy.

7 Discussion

This section will discuss both the experiment and the results of the experiment.

7.1 Methods

The task environment constructed appeared to be appropriate for comparing subjects' performance across the three methods. The complexity, repeated objects, and self similarity forced subjects to use the structure of the program to determine the correct answer, rather than just scrolling through the workspace, pattern matching the desired polygon. This more closely matches the behavior of a programmer looking at text code, or a complex visual program, where pattern matching is of limited usefulness.

7.2 Results

The results clearly show that the CSZ method is more accurate than than the SZ method, while being faster than the flat method, which would seem to be an excellent indication that the subject's understanding of the program was better using the CSZ method. This is

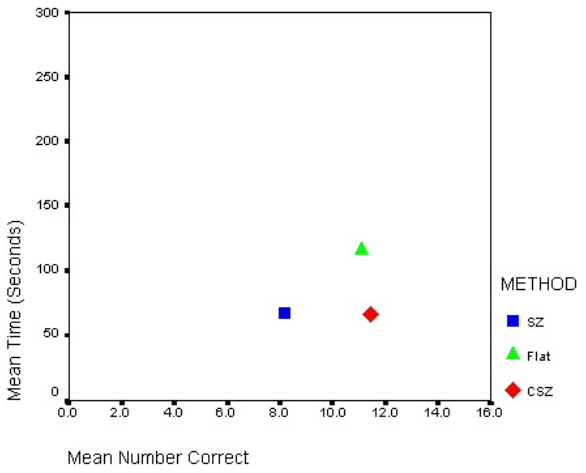

Figure 11: Scatter plot of mean time to completion and mean number correct for each method. This shows a summary of information presented in Figure 9

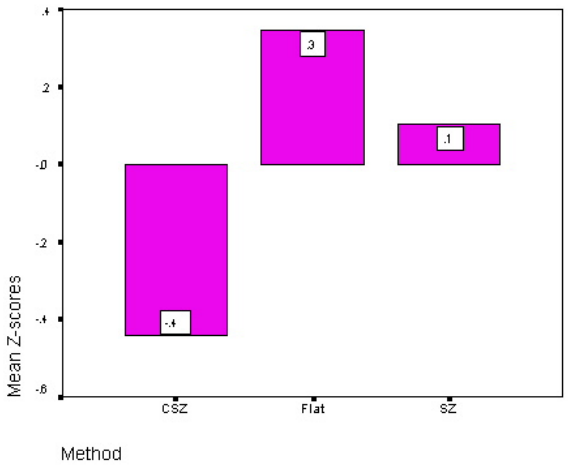

Figure 12: Mean of time and inaccuracy Z-scores. Note that the goal is a short time to completion and a small inaccuracy, therefore the more negative the score, the higher the value.

clearly shown by the plots in Figures 8 and 11, and supported by the statistical analysis of means shown in Subsection 6.

Subsequent analysis of the path-to-target data collected during the study shows the reason for the time and accuracy differences.

Subjects using the Flat method were extremely slow in the first few tasks, then their speed picked up as they progressed to the later tasks. This pronounced learning curve dramatically increased the time for subjects using the Flat method. They were still, however, slower than the other two methods, even on the later tasks.

Subjects using the SZ method showed comparable accuracy results to those using the CSZ method for simple drill-down tasks. The accuracy difference between them appeared when information was required from multiple sources in the program to select the correct target. The subjects using the SZ method rarely even looked at these required data, while those using CSZ evidently used it in their assessments.

The Z-scores shown in Figure 12 are weighted evenly between accuracy and time. Since comprehension is the overall goal, accuracy would seem to be more important than speed, and a different weighting might deliver a more meaningful result. Given enough weight on accuracy, the Flat method will even be shown as preferable to the SZ method, in this case. The question becomes how to distribute the weighting.

Although the present study used a simple visual programming paradigm, we believe the results will generalize to more traditional imperative programs. The hierarchical structure designed into the study is commonly used in C programming, for example, and the structure is easily extended to cover objects for object oriented paradigms.

7.3 Future Work

Several directions are available for future work. First, it would be interesting to further test the key difference between SZ and CSZ to ascertain why the subjects using SZ did not make use of all the required information. These investigations can take the form of making the remote required information more obvious while using the SZ method, or making it less obvious when using the CSZ method. Such test would undoubtedly suggest other directions as well.

The CSZ method also needs to be investigated with dynamic, rather than static programs, to see how it might help increase understanding of executing programs. This could be easily done with the current test setup, as the capability is already there and was merely hidden from the subjects of this study.

Another direction is to attempt similar tests using representations of textual programs. Since a great deal of programming is done using textually based programming languages, defining how the CSZ technique can be used and its efficacy with such programs is essential.

Finally, we would like to expand the CSZ technique to work with more real-world type problems. Visualizations of parallel programs [Summers et al. 2000] should benefit greatly from this method, and integration with those visualization systems is imminent.

This work should provide a base from which to extend experimental procedures with the goal of creating a test bed for performing empirical research on methods of representation. Now that CSZ has been shown to be a viable method for viewing hierarchical data further experiments can be conducted with more specialized program environments.

8 Conclusion

Understanding programs is difficult, especially for inexperienced users, and particularly if they have not written the code. A traditional approach to facilitating understanding has been to encapsulate functionality inside procedures and subroutines. This approach draws boundaries between functional elements and avoids distractions from unrelated pieces of code.

One drawback, however, with partitioning programs in this way is that those same boundaries make it difficult to see the overall context of the procedure while examining it's details. Likewise, the boundaries tend to lock out the details when viewing the overall structure. The transition between overview and detail level is also quite abrupt.

Continuous semantic zooming allows a continuous change between these levels which helps the user retain contextual relationships, and allows a limited amount of crossover.

We have presented an empirical study framework, as well as the first set of studies for exploring problems of this type. These studies supported the thesis that CSZ uses the attributes discussed above to provide an opportunity for greater understanding of programs.

The studies were designed to compare three methods of viewing the same basic program. The experimental results suggest that CSZ is indeed an improved method for looking at these types of programs, and these results should be extensible to more generalized programming languages. At the same time, the experimental procedures themselves have been shown, through experience, observation, and internal consistency, to be useful for comparisons of these types of paradigms.

9 Acknowledgements

James Lin designed and conducted the second pilot study. Mark Young of Khoral, Inc., designed and implemented a large portion of the Cantata–Flatland interface. The University of New Mexico Center for High Performance Computing contributed valuable resources toward this research.

References

BEDERSON, B. B., HOLLAN, J. D., PERLIN, K., MEYER, J., BACON, D., AND FURNAS, G. 1996. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. *Journal of visual languages and computing 7*, 1, 3. ISSN: 1045-926X.

CARPENDALE, M. S. T., COWPERTHWAITE, D. J., AND FRACCHIA, F. D. 1997. Extending distortion viewing from 2D to 3D. *IEEE Computer Graphics and Applications* (July/August), 42–51.

CAUDELL, T. P., AND SUMMERS, K. L., 2003. Homunculus project home page. http://www.hpc.unm.edu/homunculus.

CAUDELL, T. P., 2000. A guide to flatland. Unpublished Technical Document.

DILL, J., BARTRAM, L., HO, A., AND HENIGMAN, F. 1994. A continuously variable zoom for navigating large hierarchical networks. In *IEEE International Conference on Systems, Man, and Cybernetics*, IEEE, 386–390 vol. 1.

FISHKIN, K., AND STONE, M. C. 1995. Enhanced dynamic queries via movable filters. In *Proceedings of CHI'95, ACM Conference on Human Factors in Computing Systems*, ACM, 415–420.

HAYS, W. L. 1988. *Statistics for psychologists*, 4th ed. Holt, Rienhart and Winston.

LEUNG, Y. K., AND APPERLEY, M. D. 1994. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction 1*, 2 (June), 126–160.

MUTHUKUMARASAMY, J., AND STASKO, J. T. 1995. Visualizing program executions on large data sets using semantic zooming. Tech. Rep. GIT-GVU-95-02, Georgia Institute of Technology, January.

PAPERT, S. 1980. *Mindstorms: children, computers, and powerful ideas*. New York: Basic Books. ISBN 0465046274.

RAO, R., AND CARD, S. K. 1994. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Proceedings of CHI'94, ACM Conference on Human Factors in Computing Systems*, ACM, 318–322 and 481–482.

REKIMOTO, J., AND GREEN, M. 1993. The information cube: Using transparency in 3d information visualization. In *Proceedings of the Third Annual Workshop Information Technologies and Systmes (WITS '93)*.

ROBERTSON, G. G., AND MACKINLAY, J. D. 1993. The document lens. In *Proceedings of UIST'93, ACM Symposium on User Interface Software and Technology*, ACM, 101–108.

SINDRE, G., GULLA, B., AND JOKSTAD, H. G. 1993. Onion graphs: Aesthetics and layout. In *Proceeding, IEEE/CS Symposium on Visual Languages (VL '93)*, IEEE, 287–291.

STOREY, M.-A. D., WONG, K., FRACCHIA, F. D., AND MULLER, H. A. 1997. On integrating visualization techniques for effective software exploration. In *Proceedings of IEEE Symposium on Information Visualization*, IEEE, 38–45.

SUMMERS, K. L., GREENFIELD, J., AND SMITH, B. T. 2000. A survey of parallel program performance evaluation techniques using visualization and virtual reality. In *Proceedings, IEEE Aerospace conference*, IEEE.

SUMMERS, K. L. 2002. *Visualization of Programs Using Proximity to Trigger Continuous Semantic Zooming: An Experimental Study*. PhD thesis, The University of New Mexico.