An Investigation of 3D Visual Metaphors for Software

Andrew Sutherland
Software Research Lab
Department of Computer Science
University of Saskatchewan
andrew.sutherland@usask.ca

April 6, 2005

Abstract

Software visualization is an emerging field that is becoming of greater interest due to the increasing size and complexity of software. Three-dimensional visualization has the possibility of improving understanding and comprehension of software by providing a visual metaphor that more closely resembles the way human beings interact with real-world objects. An investigation into what kinds of visual metaphors support cognitive understanding of software is performed. Novel techniques for representing software in three dimensions are offered as supplements to existing implementations and ideas for future prototypes.

1 Introduction

When presented with a difficult problem involving relatively large amounts of data, it is human nature to externalize the problem by writing down facts and figures. Creating visual aids takes the burden of remembering large amounts of information and frees up mental power, allowing us to identify patterns and work out solutions to the problem more easily. Card et al. describe a visualization as serving two separate purposes: the first being to communicate the idea (which requires having an idea in the first place), and the second being able to create or discover new ideas; to use the special properties of visual perception to resolve logical problems [4].

Designing software is a complex process and the software itself is a complex artifact. Like any difficult or complex problem, understanding and working with a piece of software can be enhanced by using visual aids. Software visualization is an area of information visualization that focuses on visualizing the varying types of information associated with software design, development, and maintenance. The use of visualization helps developers and non-developers alike gain a better understanding of how software is built, executed, debugged, and changed over time.

In recent years, the capabilities of graphic processors have increased to impressive levels. Naturally, the visualization community has an interest in utilizing this new technology to create more interesting and useful visualization applications. However, it is not immediately clear how to harness this new technology in order to create not only an aesthetically pleasing visualization, but an application that enhances some aspect of software engineering with the use of visual aids. This paper defines an organization of the various techniques currently used to visualize software. A number of problems are identified with current visualizations, and three-dimensional visualization is offered as a viable solution.

2 Related Work

Given that the necessary technology is becoming more available, it seems logical to extend visualizations to take advantage of the third dimension. There is no doubt that three-dimensional visual representations are more exciting and engaging than flat representations, but it is an open question under what conditions 3D visualizations are better than their 2D counterparts [4]. 3D poses much greater implementation challenges as it introduces additional parameters to the visualization such as lighting, texture, and additional degrees of freedom for movement. There is also a need for more processing power and specialized hardware when using 3D although this is becoming less of a disadvantage as more powerful technology is quickly becoming more available. Thus, the question becomes how much of an advantage is gained by involving the third dimension and is it worth the tradeoff of increased complexity?

There have been a number of studies that give encouraging results in favour of the use of a third dimension in visualization. Tavanti and Lind [13] performed an experiment with spatial memory using 2D and 3D displays, which was later extended and repeated by Andy Cockburn [5]. The experiment involved subjects recalling the locations of letters of the alphabet that were hidden behind squares placed on a flat surface (for 2D) and on a landscape (for 3D). Their results indicated that using a 3D display better supported the ability of the subject to recall the placement of the letters and they claim it supports the notion that user interface performance can be improved by incorporating perspective effects.

Irani et al. [9] investigated recent research on human perception and made use of structured object recognition theory to define a syntax for mapping informational objects such as software components and relationships to three-dimensional graphical objects. Structure-based recognition theory states that when a structure is viewed, the human perception system breaks the structure down into primitive shapes such as cones, cubes, lines, etc. Irani claims that if informational structures can be mapped into structured objects built from these primitive shapes, then the structure of the information will be automatically extracted as a part of normal human perception. The syntax they derived involves a number of simple 3D shapes along with a ruleset that defines how the shapes can be arranged to create various structures. The best way to represent a particular aspect of software (for example, a dependency between classes) was determined by performing experiments where the subject would choose the best representation from a variety of compositions. Figure 1 shows an example of a selection of compositions that could be used to represent class dependency. Overall, the most popular choice in this case was Composition C.

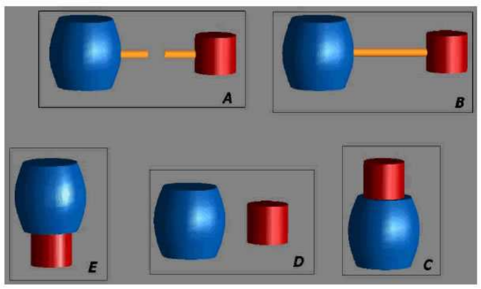

Figure 1: Perceptual Syntax for Class Dependency

Similar experiments were performed for representations of relationship strength, multiplicity, aggregation, and generalization. After determining what representations were most accurately perceived, a UML diagram was converted to a 3D representation using the derived syntax. Both the regular UML diagrams and their 3D derivations were presented to a classroom setting of students with no experience in UML, and questions were asked to see what representations best portrayed the relationships between components. The results obtained indicated that there were 5 times as many errors when using the UML diagrams as with the 3D diagrams. Irani claims these results favour the use of structured 3D representations for visualizing the structure of software.

While the preceding studies present encouraging results for the use of three dimensions in visualization, there still remains much investigation to be done on how best to exploit the additional dimension to encode information. Work on this area has been limited, especially in the application of 3D graphics for visualizing software.

3 Creating an Effective Visualization

To investigate how 3D visualization can be used to improve understanding of software, it is necessary to explore the various visualization techniques already in use and determine the types of metaphors that represent software in a manner that encourage better understanding. A classification of visualizations for software was determined. Along with determining what representations are worthwhile, it is important to determine some of drawbacks or problems that visualizations tend to suffer from.

3.1 Visual Metaphors

Card et al. state that before understanding the intuition behind visualization, it is useful to gain an appreciation for the important role of the external world in thought and reasoning. To gain this appreciation, the mapping between real world entities and visual representations is defined as a *visual metaphor*. Averbukh states that a visual metaphor is a mapping that provides correspondence between notions and objects of the modeled application domain and a system of similarities and analogies [1]. Bosch et al. take a more specific approach when defining a visual metaphor. They claim that metaphors create the visual representations for sets of data (e.g. a table) by using *primitives*, which in turn create the visual representations for the individual pieces of data (e.g. data tuples) [3]. For the purposes of this paper, a visual metaphor is defined as the transformation of non-physical artifacts and metrics to viewable entities with colour, shape, and other visual characteristics. Lanza's work on software evolution matrices is a good example of how software metrics can be mapped to visual attributes [10]. Metrics such as number of methods, number of instance variables, or lines of code are used to determine height and width of a rectangle representing a system component. The same component is mapped to a rectangle for multiple versions of the system, each rectangle having a different height and width, depending on the value of the metrics. When the rectangles are placed in sequential order, an idea is obtained of how the metrics for that component change over time.

Some visual metaphors work better than others. This is because the human mind is attuned to interpret certain visuals in certain ways. Finding visual metaphors that are intuitive to human understanding is a difficult task.

3.2 Visual Metaphors for Software

There are a number of different metaphors for visualizing software. A categorization of visual metaphors was determined in order to better understand what types of visualization techniques work better for visualizing certain aspects of software development. The classification resulted in 3 categories. Firstly, there are visual metaphors that map individual lines of code and their attributes to visual primitives. Visual metaphors based on this idea can be categorized as being *Code-level Metaphors*. Many visualization fall into the category *Structural metaphors*, which map the logical structure of a software system to visual primitives. Lastly, visualizations representing some temporal component of software (such as software evolution or run-time execution) can be categorized as *Temporal Metaphors*.

3.2.1 Code-level Metaphors

Visualizations in this category operate on individual lines of code. The purpose of these types of visualizations is to glean information such as how files are organized in a system, the age of particular pieces of the system, and the developers responsible for writing certain pieces of code. SeeSoft [6] is the classic example of a software visualization of code.

Lines of code are mapped to individual lines that are coloured based on a determined attribute (relation to a system, type of statement, author, date written, etc.). Figure 2 shows the result of this mapping. The idea of using colour to represent various types of a similar element is a central theme in many software visualizations.

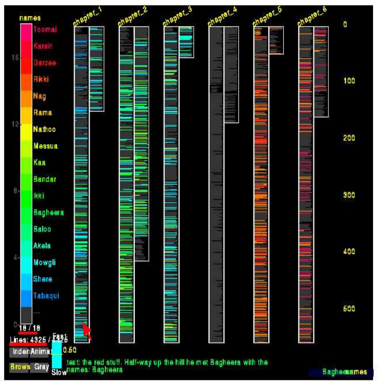

Figure 2: SeeSoft software visualization

Griswold et al [8] use a visualization called Aspect Browser that is based on the same metaphor that SeeSoft uses but have added in an additional *Map Metaphor*. Aspect Browser supports the Map Metaphor with features such as indexing, a cursor indicating where you are looking in the actual source, zooming, magnification, and folding to reduce the size taken up by non-interesting portions of the visualization. These features preserve screen-space and allow for greater scalability.

3.2.2 Structural Metaphors

An alternative to viewing low-level code is to extract and visualize the logical structure of the software. Reverse-engineering techniques are used to perform the extraction of a fact base containing information about the components of the system and the relationships between the components. Visualizing such a structure can give insights into the design of a system and aid in decision making when performing maintenance changes.

These types of visualizations are particularly effective at representing object-oriented systems, as the concept of entities or objects is easy to map to a visual representation.

SHriMP (Simple Hierarchical Multi-Perspective) Views [12] is a visualization environment that was designed to enhance how users explore complex information spaces. It was one of the first visualizations to use the concept of *nested interchangeable views*, which allow the user to view information at different levels of abstraction. If the user is interested in a particular subsystem being displayed, the node representing that subsystem can be selected and the perspective will narrow down onto the subsystem, revealing additional details that are of interest to the user (such as the classes comprising that subsystem) and hiding extraneous details (such as the other subsystems).

3.2.3 Temporal Metaphors

Visualizing the structure of a piece of software may provide understanding of the system at that particular moment, but sometimes it may be desirable to understand how software changes over a longer period of time. Software evolution is defined as the intrinsic need for continuing maintenance and development of software used to address an application or solve a problem in the real world domain [11]. That is, software must change in order to retain it's usefulness in the real world.

Wu et al. [14] have adapted sound spectrographs to visualize the evolution of software. A sound spectrograph is used to visually represent frequency content of sound and its variation over time. Software spectrographs are used to visually represent how the various components of the system change over versions. Colour is used to indicate the degree of change. Figure 3 displays an example of a software evolution spectrograph for the OpenSSH system.

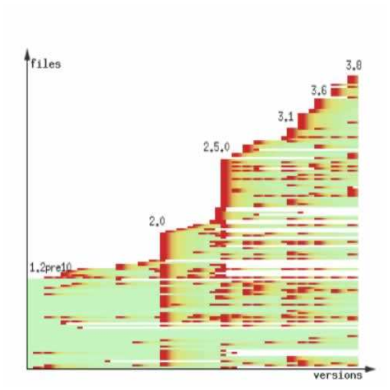

Figure 3: Evolution Spectrograph - Fan In of Changed Dependencies for OpenSSH

3.2.4 Visualization Pitfalls

Visualization metaphors in all 3 categories tend to suffer from one or more of the following problems.

- *Occlusion* - Refers to when objects overlap or obstruct one another. This problem is compounded if there are no features allowing the user to alter their viewpoint or change the orientation of the visualization.

- *Lack of screen-space* - Graph-based visualizations and visualizations of large amounts of data often suffer from being unable to view a substantial part of the system in one screen, or objects become so densely packed that it is impossible to find specific elements.

- *Lack of intuitive interaction* - Static visualizations may be suitable for specific tasks, but it is more desirable to have an application that can support a variety of tasks through the use of interaction techniques.

- *Poor performance and/or lack of scalability* - It is common for applications to support visualization of a dataset of a specific size, but when the size of the dataset is larger than the target size, the application fails to maintain an interactive framerate due to the increased processing power required.

- *Lack of integration with software development tools* - Although not strictly a requirement, a visualization application is much more likely to be used when integrated with existing tools. For example if a software visualization for Java programs was implemented as an Eclipse plug-in, it would be much more accessible to software developers than if it was deployed as a stand-alone application.

These problems are additional constraints on the metaphor used for representing software visually. They must be taken into account in order to achieve an effective visualization. The use of three-dimensions may alleviate some of these problems but may also exacerbate others.

4 Three-Dimensional Visualization Techniques

There are a number of techniques that support three-dimensional visual metaphors for software. These techniques work towards eliminating some of the problems with current metaphors described in the preceding section.

4.1 Transparency

As mentioned earlier, occlusion is a common problem found in visualizations, especially graph-based implementations. The use of a third dimension may even exacerbate the problem, by introducing perspective effects [4]. More distant objects grow smaller and may be more easily occluded by objects at the forefront. Appropriate use of transparency can effectively eliminate the problems of occlusion. By making an object translucent, the occluding object still has representation in the visualization and the occluded objects are also made visible. Transparency can also be used to preview the interior structure of objects in a hierarchical scheme. For example, by mousing over an object representing a class, the object could temporarily become less opaque, revealing the interior methods and fields of the class. The user could then decide if they want to explore this class based on their preview of the structure. This technique may aid in freeing up screen-space and aid in providing intuitive interaction techniques.

4.2 Immersion

Balzer et al. [2] suggest that a particularly intuitive visualization metaphor is the *landscape metaphor* for software visualization. The landscape metaphor is based on navigating through a virtual landscape that is based on the hierarchical structure of a software system. While it may not be useful to take the metaphor to the extremes of representing software components as landscape artifacts, the navigation techniques used in similar applications (such as video games), may be inducive to providing a familiar means for interacting with hierarchical data. For example, when a user selects a node representing a package in a Java program, instead of expanding the node to show the classes contained in the package, the user actually enters the node to view the interior structure. This better represents the relationship between package and class, and may also allow for the use of navigation mechanisms that are more similar to how a user would normally move through a 3D environment.

4.3 Colour and Lighting

Many visualizations map some attribute of the visualized software to colour. Colour is often used to represent recent or sudden change, degree of faults or bugs, or relevancy to some task the user is performing. Wu's software spectrographs [14] are a good example of where colour is used to indicate punctuated change in the evolution of a software system. When working with three-dimensional visualization, it may be beneficial to take advantage of the lighting mechanisms that are used for applying shading to an object or lighting up particular areas of a scene. For example, if the user indicates they wish to view what classes would be affected by making a particular change to a method, the potentially affected classes could be coloured or lit dynamically depending on the user's choice. The brightness of the lighting could be mapped to how closely that class is coupled to that particular method - the more that particular change would affect the class, the brighter the effect should be.

4.4 Animation

It is easy to become disoriented if a visualization of a graph or a diagram instantly changes arrangements. Change is more easily observed if the steps in between that sudden change are shown. Animation can aid in the transition from one visualization state to another. Software evolution visualization can be aided by using key-frame interpolation. Key-frame interpolation is often used in video games to animate characters using still poses or key-frames [7]. The character is animated by picking two key-frames and blending together each corresponding pair of vertex positions using weights. The weight of the blend is based on the passing of time between the two frames. The result is a smooth animation without the need to manually provide each intermediate frame. Visualizing software evolution often involves multiple versions of a software system. By using visualizations of these versions as key-frames, an animation of the evolution of the system could be viewed by interpolating the individual visualizations that correspond to versions.

5 Prototype

To test the proposed techniques, an experimental prototype was constructed. The prototype uses the source transformation language TXL to extract a fact base from a Java program. The fact base is then used as data in a graph-based visualization in OpenGL. Java packages, classes, methods, and fields are mapped to spheres. The colour of the sphere is determined by the type of entity (package, class, etc.), although in future prototypes, colour will likely be used to represent other software metrics. The prototype utilizes transparency in a fashion similar to what was described earlier. If a certain component is moused over, the component becomes transparent, showing the other components that would have otherwise been obstructed. Interaction techniques such as zooming and rotation were included to give

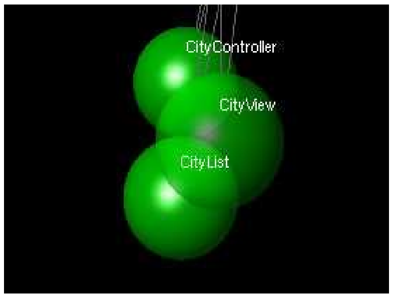

Figure 4: Transparency as a means for solving occlusion problems

the user the ability to arrange the visualization with the desired orientation. Clicking or picking an object (a Java class for example) will zoom the user into that class so that they can view the interior methods and fields of the class that was selected. Hierarchically organizing the data substantially frees up screen-space and makes the visualization simpler and easier to understand.

Relationships between classes and methods (method calls, inheritance, etc.) are mapped to lines between the related components. A clustering algorithm was used to arrange the components in a manner that reflected their logical coupling. Classes that were more closely coupled were positioned within closer proximity to one another than classes that were not as closely coupled. This feature was implemented with the idea that it would support the natural human tendency to group related objects. Animation was used to demonstrate how the structure of the program changed over several versions of the same program. When used in conjunction with the clustering algorithm, an idea of how coupling changed between components was observed and followed throughout versions. The prototype was shown to a class of software engineering graduate students at points throughout its development to measure the usefulness and effectiveness of visualizing software in three dimensions. The reactions of the observers was promising. Students expressed their interest in visualizing their own programs with the application. This provides further encouragement to explore

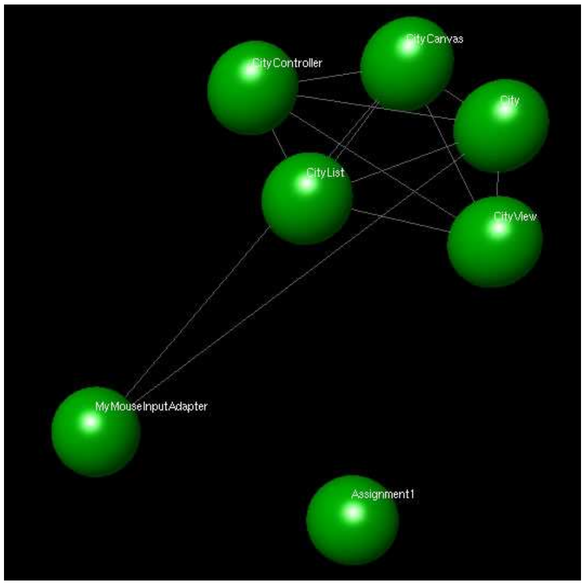

Figure 5: Positioning related objects in clusters

the use of three-dimensional visual metaphors for software.

6 Conclusion and Future Work

An introduction to software visualization was presented in the context of creating intuitive 3D visual metaphors to aid in software understanding. A number of problems were identified with current visual metaphors for software, and a number of techniques supporting the use of three-dimensional visualization were offered as possible alternatives. Possible avenues for further research include a more formal study of what arrangements of three-dimensional structures allow for improved understanding of software and creating additional prototypes to explore novel techniques for representing software visually. It is believed that there is a large potential for significantly aiding in development and maintenance of software through the use of well-designed visualization tools.

References

[1] Vladimir L. Averbukh. Toward the conceptions of visualization language and visualization metaphor. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 390. IEEE Computer Society, 2001.

[2] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *VisSym*, pages 261–266, 2004.

[3] Robert Bosch, Chris Stolte, Diane Tang, John Gerth, Mendel Rosenblum, and Pat Hanrahan. Rivet: a flexible environment for computer systems visualization. *SIGGRAPH Comput. Graph.*, 34(1):68–73, 2000.

[4] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Using vision to think*. Morgan Kaufmann Publishers Inc., 1999.

[5] Andy Cockburn. Revisiting 2d vs 3d implications on spatial memory. In *CRPIT '28: Proceedings of the fifth conference on Australasian user interface*, pages 25–31. Australian Computer Society, Inc., 2004.

[6] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[7] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[8] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274. IEEE Computer Society, 2001.

[9] Pourang Irani, Maureen Tingley, and Colin Ware. Using perceptual syntax to enhance semantic content in diagrams. *IEEE Comput. Graph. Appl.*, 21(5):76–85, 2001.

[10] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 37–42. ACM Press, 2001.

[11] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 70–74, New York, NY, USA, 2001. ACM Press.

[12] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. Shrimp views: an interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521. ACM Press, 2002.

[13] Monica Tavanti and Mats Lind. 2d vs 3d, implications on spatial memory. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, page 139. IEEE Computer Society, 2001.

[14] Jingwei Wu, Richard C. Holt, and Ahmed E. Hassan. Exploring software evolution using spectrographs. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 80–89. IEEE Computer Society, 2004.