# 3D Representations for Software Visualization

Andrian Marcus
Kent State University
Department of Computer Science
Kent, Ohio USA 44240
amarcus@cs.kent.edu

Louis Feng
Kent State University
Department of Computer Science
Kent, Ohio USA 44240
lfeng@cs.kent.edu

Jonathan I. Maletic
Kent State University
Department of Computer Science
Kent, Ohio USA 44240
jmaletic@cs.kent.edu

## Abstract

The paper presents a new 3D representation for visualizing large software systems. The origins of this representation can be directly traced to the SeeSoft metaphor. This work extends these visualization mechanisms by utilizing the third dimension, texture, abstraction mechanism, and by supporting new manipulation techniques and user interfaces. By utilizing a 3D representation we can better represent higher dimensional data than previous 2D views. An overview of our prototype tool and its basic functionality is given. Applications of this method to particular software engineering tasks are also discussed.

**CR Categories:** D.2.2 [Software Engineering] Design Tools and Techniques, D.2.7 [Software Engineering] Distribution, Maintenance, and Enhancement, H.5.2 [Information Interfaces and Presentation] User Interfaces

**Keywords:** Software visualization, 3D visualization, File maps, SeeSoft

## 1 Introduction

Software visualization addresses a wide variety of problems that range from algorithm animation and visual programming to visualizing software design issues of large-scale systems. Our particular focus, with this research, is the visualization of large scale software to assist in comprehension and analysis tasks associated with maintenance and reengineering. This work brings together research from software analysis, information visualization, human-computer interaction, and cognitive psychology.

Research in software visualization has flourished in the past decade. A large number of tools, techniques, and methods were proposed to address various problems. Unfortunately, the success of many of these results is still to be proven and qualitative evaluation of software visualization system is often extremely difficult.

One of the most successful and well-known application, SeeSoft [Ball and Eick 1996, Eick et al. 1992], was proposed by Eick et al. in the early 90's. Several attributes of the SeeSoft metaphor warrant its success and usefulness. One of the most important of these attributes is the natural and direct mapping from the visual metaphor to the source code and back. This in turn leads to a natural navigation between the representations. This makes the visual representation easy to understand; yielding high levels of trust on behalf of the user. Color and pixel maps are used to show relationships between elements of a software system (rather than graph-based representations). This allows the representation of large amounts of source code, the non-trivial relationships, and data on a standard 2D visualization medium (e.g., monitor or screen). Many other software visualization tools use graph-based representations that suffer from scalability, layout, and mapping problems.

In this paper, we present the sv3D (source viewer 3D) framework, which implements a 3D metaphor for software visualization. Our 3D metaphor is based on the SeeSoft representation however it brings a number of extensions to the original concept. The underlying motivation of this work is in exploring new mediums and representations to address particular software engineering tasks [Maletic et al. 2002].

The next section presents related work in the field that motivates our approach. Section 3 describes which aspects of software visualization are addressed by our work. The proposed 3D representations are implemented within the sv3D (source viewer 3D) framework. The architecture, implementation and main features of sv3D are presented in the following sections. Examples and applications of sv3D are shown with respect to related work. The paper concludes with listing the aspects of sv3D that are under development and require further research.

## 2 Related SeeSoft Work

SeeSoft-like representations are used by a number of existing tools: Tarantula [Jones et al. 2001], The Aspect Browser [Griswold et al. 2001], The Aspect mining Tool [Hannemann and Kiczales 2001], Bee/Hive [Reiss 2001], G$^{SEE}$ [Favre 2001], Advizor [Eick 2000], etc.

Despite its success, SeeSoft and most of its versions have noted limitations. Namely, the use of 2D pixel bars limits the number of attributes that can be visualized as well as the type of relationships that can be shown and hierarchical relationships are difficult to represent. Additionally, one of the major strengths of the metaphor (i.e., direct linking to the source code) also yields one of its weaknesses that is, little support for multiple abstraction levels and limited usage of the 2D space.

A number of improvements of the original SeeSoft representation were made by researchers. In particular, Tarantula [Jones et al 2001] uses brightness to represent and extra attribute. However, as noted by its authors brightness is confusing and very poorly perceived by the users. Bee/Hive [Reiss 2001] introduces the file maps, which make use of texture and the third dimension in the visualization. The file maps form only one view supported by Bee/Hive. By supporting multiple views of the data and multiple data sources, Bee/Hive overcomes many of the limitations of the SeeSoft view. However, the supported user interactions are somewhat limited for the 3D renderings, thus suffering from some of the problems inherent to 3D visualizations (e.g., occlusion).

sv3D builds on the success of SeeSoft and Bee/Hive, while trying to address some of the inherent limitations of the medium and representation. In particular, sv3D supports object-level manipulations that differs from Bee/Hive and SeeSoft, which support only manipulation of the entire space.

## 3 Software Visualization

We view software visualization systems in light of their applications toward supporting large-scale software development and maintenance. In order to accomplish this we define five dimensions of interest with regard to software visualization [Maletic et al 2002]. These dimensions reflect the why, who, what, where, and how of the software visualization. The dimensions are as follows:

- Tasks – why is the visualization needed?
- Audience – who will use the visualization?
- Target – what is the data source to represent?
- Representation – how to represent it?
- Medium – where to represent the visualization?

A detailed view over these dimensions can be found in [Maletic et al 2002]. The focus of the work presented here is along the *representation* dimension of software visualization and we will further elaborate on this issue.

### 3.1 Representation

Depending on the goals and target of the software visualization system, the type of users, and available medium, a form of representation needs to be defined to best convey the target information to the user. In addition to the related work, presented earlier, we look to the research in information visualization and cognitive sciences [MacKinlay 1986, Tufte 1983, Ware 2000] to make the best choices in designing the representation for software visualization. This research centers on methods to best map raw data into a visual structure and view.

MacKinlay [MacKinlay 1986] defined two criteria to evaluate the mapping of data to a visual metaphor: expressiveness and effectiveness. These criteria were used in 2D mappings, but can also be applied for 3D mappings.

*Expressiveness* refers to the capability of the metaphor to visually represent all the information we desire to visualize. For instance, if the number of visual parameters available in the metaphor for displaying information is fewer than the number of data values we wish to visualize, the metaphor will not be able to meet the expressiveness criterion.

The relationship between data values and visual parameters has to be a univocal relationship; otherwise, if more than one data value is mapped onto the same visual parameter than it will be impossible to distinguish one value's influence from the other. On the other hand, there can always be visual parameters that are not used to map information, as long as there is no need for them to be utilized.

The second criterion, *effectiveness*, relates to the efficacy of the metaphor as a means of representing the information. Along the effectiveness dimension we can further distinguish several criteria: effectiveness regarding the information passing as visually perceived, regarding aesthetic concerns, regarding optimization (e.g., number of polygons needed to render the view).

In the case of quantitative data (e.g., software metrics, LOC, trace data), not only the number of visual parameters has to be sufficient to map all the data, but also, they must be able to map the right data. There are visual parameters that are not able to map a specific category of data; for instance, shape is not useful for mapping quantitative data, while the size of a metaphor is adequate.

Effectiveness implies the categorization of the visual parameters according to its capabilities of encoding the different types of information. Moreover, this also implies categorizing the information according to its importance so that information that is more important can be encoded more efficiently when options must be taken. This categorization of the importance of the information has two expressions: one is an assigned importance of the information in the context of a software system; the other is a preference of the user. Nonetheless, the user may choose to override this and define his own importance of the data, according to his priorities is usually the first step to understand a phenomenon or system. Although these characteristics of data apply mostly to data visualization, they must be taken into consideration when visualizing a software system.

In order to satisfy these criteria for the mapping, one must have a solid data characterization. The metaphors should be designed such that they maximize the amount of data that can be represented with an accent on the user's information seeking goals. In a similar manner as Bee/Hive, sv3D is designed to be a visualization front-end, independent from the source of the data. Thus sv3D can be used as a general data visualization tool to some degree. However, it is intended to be used for software visualization; the data mapping and choice of metaphors are determined by this aspect.

The power of a visualization (language/representation) is derived from its semantic richness, simplicity, and level of abstraction. The aim is to develop a language with few metaphors and constructs, but with the ability to represent a variety of elements with no ambiguity or loss of meaning. An important aspect to be considered in defining a visual representation is the nature of its users. One may design a language for use by software developers with solid knowledge of programming, program designs, and system architecture. The metaphors in the language should be simple, having a familiar form and straightforward mapping to the target.

With all these considerations in mind, the representation can take several forms (e.g., source code, tables, diagrams, charts, visual metaphors – icons, figures, images, virtual worlds, etc.) and have various attributes (e.g., interactive, static, dynamic, on-line or off-line views, multiple views, drill-down capabilities, multiple abstraction levels, etc.). Once again, these elements and attributes need to be defined and designed with several goals in mind, to support the needs of the user.

### 3.2 Support for user needs

Shneiderman [Shneiderman 1996], presents seven high level user needs that an information visualization application should support. For evaluation purposes, we must refine these into lower-level tasks as done by Wiss, Carr, and Jonsson [Wiss et al. 1998]. The needs are presented below and act as a guideline for developing navigational needs of the user in sv3D:

**Overview**: Gain an overview of the entire collection of data that is represented. This is often a difficult problem in the case of visualizing the structural information of large systems.

**Zoom**: Zoom in on items of interest. When zooming, it is important that global context can be retained. This subsumes methods to drill down to lower levels of abstraction.

**Filter**: Filter out uninteresting items. Filtering by removing parts of the visualization will necessarily disturb the global context. Therefore, it is important to see whether the design supports some kind of abstraction of the removed parts.

**Details-on-demand**: Select an item or group and get details when needed. Getting details on a selected item is usually implemented by the embedding application.

**Relate**: View relationships among items. For a hierarchical data structure, it is necessary that the visualization show parent-child relationships. This is one of the most important features of many software visualization systems. Software systems rely on many inter-related components, working together to solve problems.

**History**: Keep a history of actions to support undo, replay, and progressive refinement. A visitation path should be supported.

**Extract**: Allow extraction of sub-collections and of query parameters. This is related only to the application and the underlying data set. How the data is visualized does not affect this.

We will describe later in the paper how sv3D supports each of these requirements.

## 4 2D versus 3D Representations

No visualization method addresses all the needs of the users. One successful approach to address more of the user's needs is to offer multiple views of the data as done by [Knight and Munro 2001, Reiss 2001, Storey et al. 2001]. Using one view of the data limits the number of attributes and the available exploration space. The solution we propose to overcome this problem is the efficient use of a 3D space for visualization.

Visualization in the 2D space has been actively explored. Many techniques for generating diagrams, graphs, and mapping information to the 2D representation have also been studied extensively. Although the question of what benefits 3D representation offer over 2D still remains to be answered, some experiments have given optimistic results. These results further motivate our work presented here.

The work of Hubona, Shirah and Fout [Hubona et al. 1997] suggest that users' understanding of a 3D structure improves when they can manipulate the structure. Ware and Franck [Ware and Franck 1994] indicate that displaying data in three dimensions instead of two can make it easier for users to understand the data. In addition, the error rate in identifying routes in 3D graphs is much smaller than 2D [Ware et al. 1993]. The CyberNet system [Dos Santos et al. 2000] shows that mapping large amount of (dynamic) information to 3D representation is beneficial, regardless of the type of metaphors (real or virtual) used. Also, 3D representations have been shown to better support spatial memory tasks than 2D [Tavanti and Lind 2001]. In addition, the use of 3D representations of software in new mediums, such as virtual reality environments, are starting to be explored [Knight and Munro 1999, Maletic et al. 2001].

The debate in the information and software visualization fields on the 2D vs. 3D battle is still heated. We support the results that show the advantages of 3D representations. In our view the design of these representations and the underlying mapping to the data is what makes a 3D visualization successful or not. The following section describes the design details and rationales behind sv3D.

## 5 The sv3D Framework

sv3D is a software visualization framework that builds on the SeeSoft metaphor. It brings a number of major enhancements over SeeSoft-type representations:

- It creates 3D renderings of the raw data.
- Various artifacts of the software system and their attributes can be mapped to the 3D metaphors, at different abstraction levels.
- It implements improved user interactions.
- It is independent of the analysis tool. It accepts a simple and flexible input in XML format. The output of numerous analysis tools can be easily translated to sv3D input format.
- Its design and implementation are extensible.

### 5.1 Mapping Raw Data to Visualization

We intentionally separated visualization from data collection. sv3D is designed to work with a variety of analysis tools as an independent visualization front-end. Therefore the input format to sv3D is kept as generic as possible.

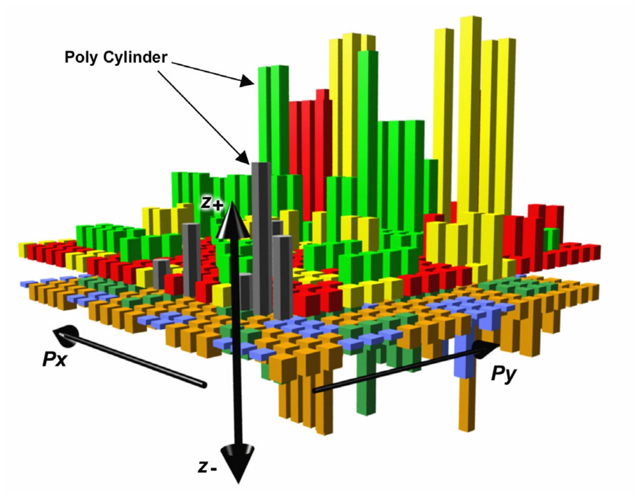

**Figure 1. Elements of the visualization. In the current version sv3D uses containers, poly cylinders, height, depth, color and position.**

We define a sv3D application $P$ as a quadruple $P = \{V, D, S, M\}$:

- $V$ defines the visual metaphors to be used.
- $D$ represents the data resulted from software analysis stored as a set of files $D = \{d_1, d_2, ..., d_n\}$, corresponding to a set of source code files $S = \{s_1, s_2, ..., s_n\}$.
- $M = \{m_1, m_2, ..., m_k\}$ defines the mapping between data and visualization as a set of relations $m_i \in D \times S \times V$.

Each source code file $s_i \in S$ is composed of lines of text $s_i = \{t_i^1, t_i^2, ..., t_i^p\}$. For each source code file $s_i$ there is an

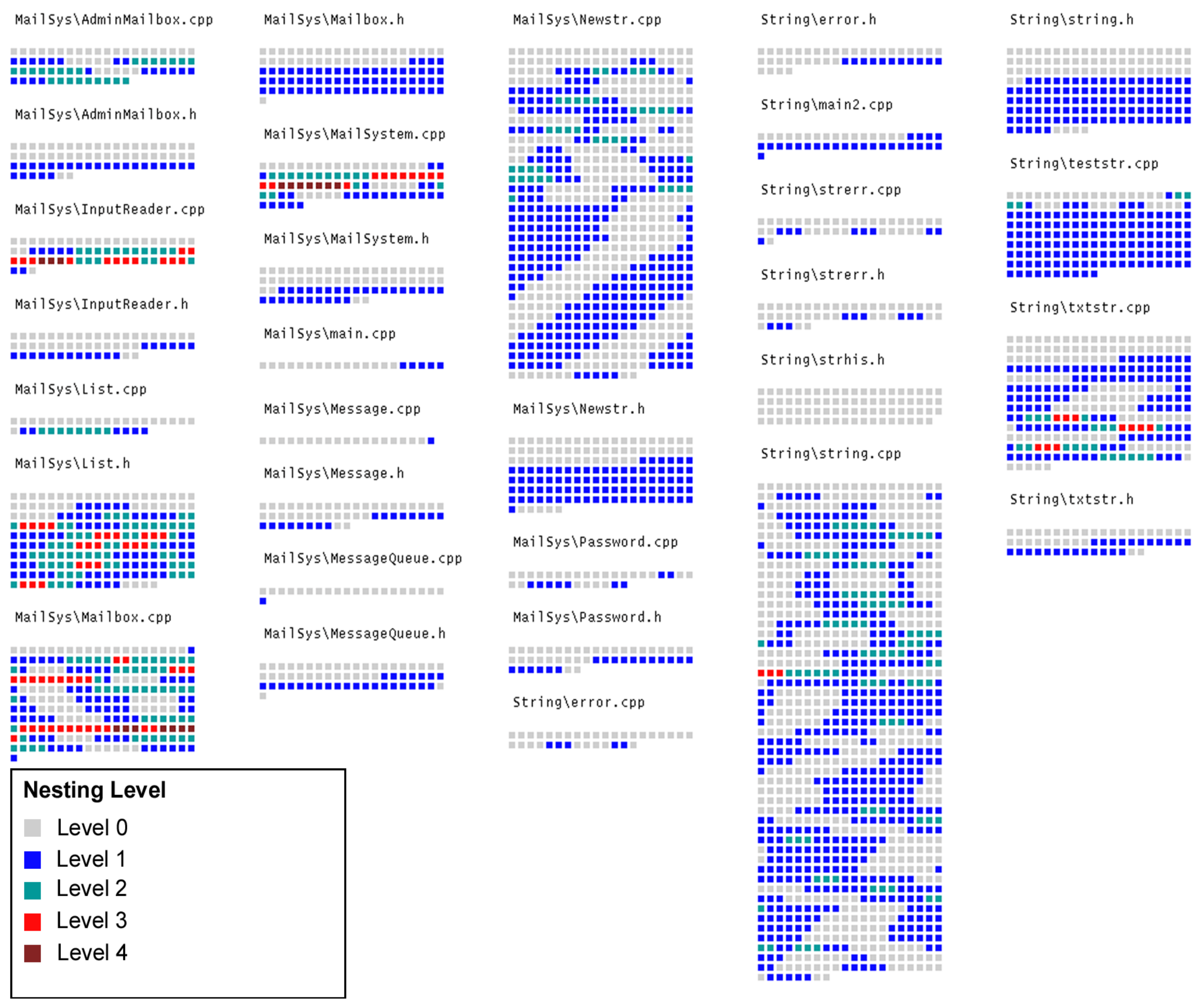

**Figure 2. A 2D overview of a system containing 30 C++ source code files (approx. 4000 LOC). Each file is mapped to a container and the name of the file is shown on top of the container. Color is used to show nesting level of the line of source code.**

associated analysis data file $d_i \in D$. Each $d_i$ is an XML file with elements $e_{ik} \in d_i$ corresponding to a line of text $t_i^k \in s_i$. Each element $e_{ik}$ has a set of attributes that contain the analysis data $e_{ik} = \{a_{ik}^1, a_{ik}^2, \ldots, a_{ik}^q\}$. In the current version of sv3D each attribute is linked to an element of the visualization $v_j \in V$, by a mapping $m_i \in M$. The number of elements in the visualization is fixed, but the number of the attributes in the data is not. If there are more attributes than visual elements, the user will decide which ones will be represented, or the system chooses a subset automatically. The same is true if the number of visual elements exceeds the number of data attributes.

The current version of sv3D supports mapping to the following elements of the visualization, defined in $V$:

- Poly cylinder - $p$
- Poly cylinder container - $o$
- Poly cylinder position in the container on its $o_x$ axis - $p_x$
- Poly cylinder position in the container on its $o_y$ axis - $p_y$
- Poly cylinder height - $z_+$
- Poly cylinder depth - $z_-$
- Poly cylinder color on $o_{z_+}$ axis - $c_+$
- Poly cylinder color on $o_{z_-}$ axis - $c_-$
- Poly cylinder shape - $\sigma$

Every element $v_j \in V$ is a nine-tuple: $v_j = \{p, o, p_x, p_y, z_+, z_-, c_+, c_-, \sigma\}$.

Figure 1 shows a close-up on a container highlighting the elements that support representation of analysis data. In this view each poly cylinder represents a line of text from the source code associated with the container. The visual components of the container represent values from the associated data file. The diameter of a poly cylinder is adjustable and is defined in the mapping.

Future versions of sv3D will also support container position in the space, relationships between containers, and texture of the poly cylinders. This will allow representation of hierarchical data and other relationships between software elements.

Expressiveness and effectiveness were the guiding principles in defining the visual elements and the default mappings. In addition, we must balance two opposing issues with regard to the user namely, the simultaneous display of as much information as possible and the dangers of information overload.

sv3D provides the user with a set of default mappings. By default, in the current version, sv3D maps a container $o_i$ to a source code file $s_i$. Each poly cylinder $p_j \in o_i$ is mapped to a line of source code $t_i^j \in s_i$. The coordinates $p_x$ and $p_y$ of a poly cylinder within in the container are determined by the position in the source code file, with a fixed width of the container. Finally, the first 4 attributes in every element of $d_i$ are mapped to cylinder colors ( $c_+$ and $c_-$ ), height ( $z_+$ ), and depth ( $z_-$ ) respectively.

The user can define, save, or load mappings, as well as other parameters such as the diameter of the cylinder. In addition to the mappings the user can define and save views that highlight different elements of the visualization. These views preserve a current state of the visualization (i.e., the source data, the mapping, and the current manipulations and visual parameters).

The default mapping is not ideally suited for all user needs. When defining custom mappings, the user need to consider what types of data can be mapped to each visual element. Some elements are better suited for quantitative data, some for categorical data. In different views, some of the elements cannot convey the information as well as in others. Poly cylinder height, depth, and color are best suited for quantitative data representation. Shape and texture are suited for categorical data representation. Only a very few shapes and textures should be used (2-3 types each). In addition, these attributes of the visualization are less effective at increased zoom levels and loose their effectiveness during overviews. Reducing the diameter of the cylinder to one pixel will of course remove this information from the visualization. Position within containers and links between containers are best suited for representation of relations.

Once the data is rendered based on the current mapping, the user can manipulate any part of the visualization, or change parts of the mapping. In the design of sv3D, particular attention was given to user interactions and manipulations. These aspects make the difference between an effective 3D visualization and an ineffective one.

## 5.2 Support for user interaction

sv3D provides support for the user tasks discussed previously. We now describe this support for each type of user task.

**Overview**: This is one of the strongest features of sv3D. The underlying 2D visualization construct used in designing the poly cylinder containers is the pixel bar chart [Keim et al. 2002], which generalizes the concept used by SeeSoft. Thus sv3D can show large amounts of source code in one view just as the SeeSoft metaphor is able to show. The simplicity of the metaphor is a feature that permits the user to zoom out and see the entire system in a single view. In addition, the visualization is rendered in a 3D space.

Navigation in each direction is supported, as well as panning, thus the user can get a view of the system form any angle and can rearrange individual elements such that the overview is most effective. Figure 2 shows a 2D overview of a small system with 30 C++ source code files and approximately 4000 lines of code. Each file is mapped to one container. Each poly cylinder represents a line of code. In this simple example color is used to represent the nesting level of a statement. On top of each container the name of the associated file is visible. When manipulating a container in the 3D space, the name of the file always faces the camera. Display of strings associated with containers (e.g., file names) can be enabled or disabled by the user.

**Zoom**: sv3D supports zooming and panning at variable speeds. This is especially important because the visualization space can possibly be quite large. Each container in the visualization can be manipulated individually (rotate, scale, translate). The user can also zoom in and out on the entire representation. Figure 4 represents the same system represented by figure 2, rendered in the 3D space. Here, some of the files were brought into a closer view and manipulated for a better camera angle.

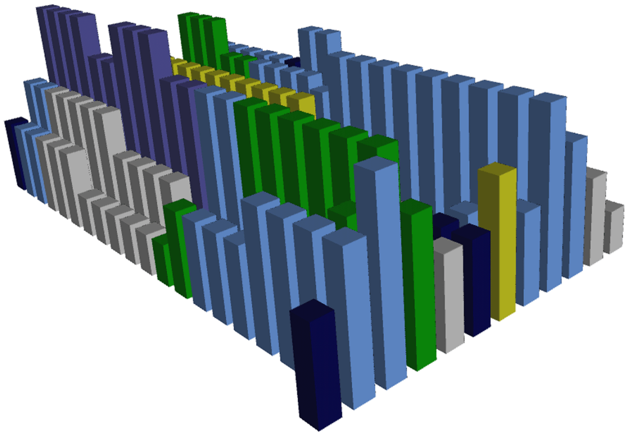

**Figure 3a. The container is associated with one C++ source code file (mailbox.cpp). Each poly cylinder represents a line of text. Color is mapped to control structure type. Height is mapped to nesting level. See figures 2 and 6 for details. See also color plate 2a.**

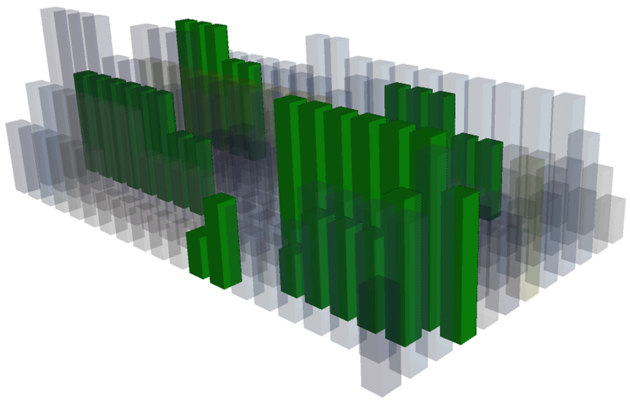

**Figure 3b. Eliminating occlusion with transparency control. Same file as in figure 3a (mailbox.cpp) is shown from the same angle. The green color is opaque and the other colors are at 85% transparency. See also color plate 2b.**

**Filter**: At this point sv3D directly supports a number of filtering methods. Un-interesting units can be filtered through their attributes or by direct manipulation. Transparency is used to deal with both occlusion and filtering. The user can chose various degrees of transparency on each class of cylinder, based on their attributes (color, shape, or texture). With semi-transparency the global context is preserved and heuristic information is retained. Elevation [Chuah et al. 1999] can also be used to filter out uninteresting units by lifting them into separate levels. Figure 3a shows a container representing a file (mailbox.cpp) from the system shown in figure 2. Figures 3a and 3b show how transparency is used to solve the occlusion problem. One can see a number of green cylinders in figure 3b, which are not visible in figure 3a. Figure 3c shows how elevation is used to separate colors, shades of blue are separated from yellow, green, and grey on different levels.

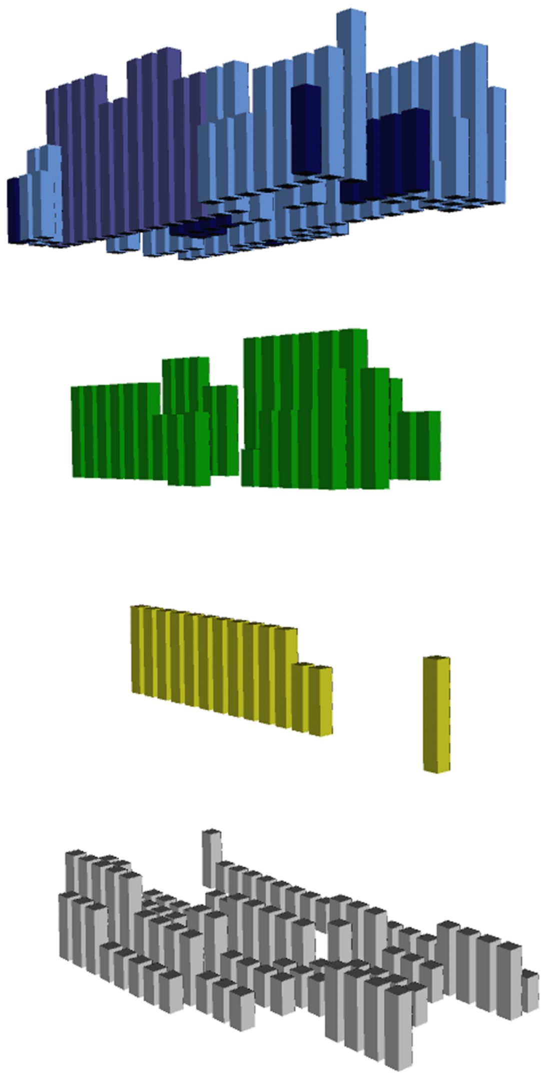

**Figure 3c. Eliminating occlusion with elevation control. Same file as in figure 3a (mailbox.cpp) is shown from a different angle. Shades of blue (on top) are separated from green, yellow, and grey respectively (on the lower levels). See also color plate 2c.**

Unwanted container can be shrunken or moved outside the current view. Display of names of elements and values can be turned on and off.

**Details-on-demand**: Current metaphors implemented in sv3D emphasis simplicity for a number of reasons. Complex visual representations do not necessarily convey information well. It is also important to be able to support user interaction, therefore performance is important. Two types of 3D manipulators (i.e., track ball and handle box) are available to the user to interact with the visualization. Also, a number of 2D interactions are supported. An information panel displays the data values on selected items. Figure 4 shows several containers selected and scaled. The name of the files linked to the containers is also shown. Two of the containers have active manipulators (e.g., handle box on the left container, and track ball on the right container).

**Relate**: The relationships between items are shown through the elements of the visualization that do not directly support representation of quantitative data (such as shape, texture, and position). The other elements (such as color and height) could also be used to show relationships. Although pixel bar charts and its variations do not directly support representation of hierarchical relationships, we are investigating a variant representation based on set-based visualizations of overlapping classification hierarchies [Graham et al. 2000]. In addition, the 3D space allows arranging the containers in any place. We are investigating ways to use links between the 3D containers and arrange them in a graph layout.

**History**: The user can take snapshots of the current view. The current view is described by a scene graph, which is composed by the attributes of the camera and all 3D objects. These snapshots of the scene graph can be saved and reviewed. A sequence of such snapshots can be played, thus representing a path within the visualization. More than that, we intend to build into sv3D change tracking based on individual users.

**Extract**: The development of sv3D at this stage is focused on representation and user interaction. Extraction and querying features will be added in the future.

Our view of the representation of a software visualization system subsumes much of the taxonomical categories proposed by Price [Price et al. 1993] and Roman [Roman and Cox 1993].

## 6 sv3d Architecture and Implementation

The user needs were the driving factors in the design and implementation of sv3D. We tried to achieve a high level of extensibility, flexibility, and performance. In order to achieve these goals sv3D is designed as an extensible framework using Qt [Trolltech 2002] for the user interface and Open Inventor [Wernecke 1994] for the rendering components. The SoQt Toolkit [Coin3D 2002] allows sv3D to use Qt and Open Inventor together to generate applications. Figure 5 shows a high level view of sv3D's architecture.

Qt is a well known cross platform GUI framework. The Linux KDE was built using the Qt GUI framework. Qt offers great portability and generates common user interfaces. Since sv3D is intended to be used in concert with other analysis tools on various platforms, Qt was a natural choice for the GUI implementation.

OpenGL has long been the standard cross platform API for high quality, high performance interactive 3D visualizations. However, a higher level toolkit suitable for developing large visualization applications is beneficial.

Open Inventor is an open source high level C++ object oriented toolkit originally developed at SGI. The toolkit is system-independent and runs on major platforms, such as Microsoft Windows, Linux, and UNIX.

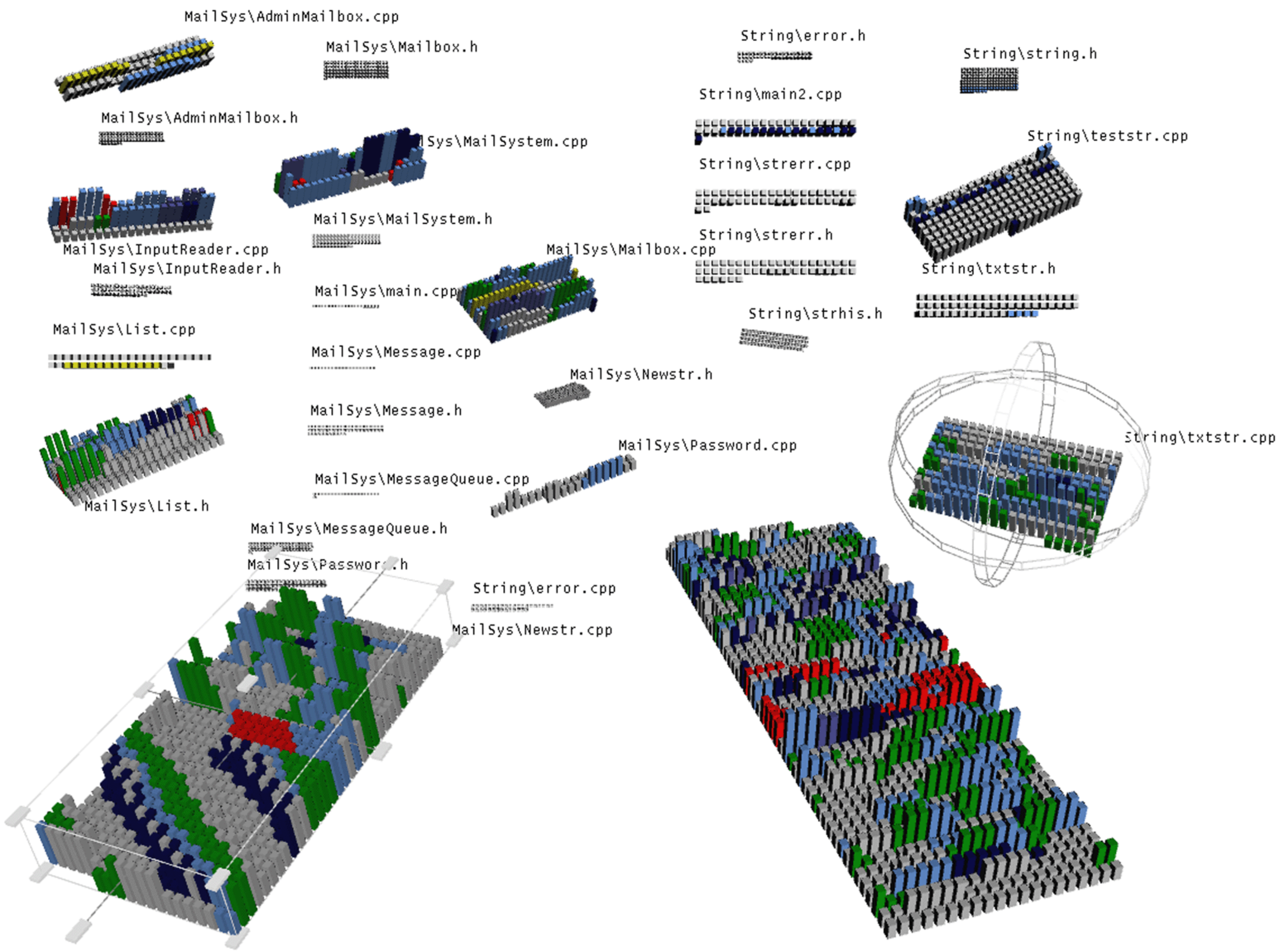

**Figure 4. Overview in the 3D space of the mailing system. Color represents control structure (figure 2) and height represents nesting level. Two files have active manipulators (handle box for scaling in the left and track ball for rotating in the right). See also color plate 1.**

The input data for a sv3D application is in XML format. Sv3D utilizes the SAX XML parser in Qt to process data files. The SAX parser is an event driven, memory efficient interface, no data object tree is needed. We partially addressed one of the burning issues in software visualization – scalability. In addition, all the implementation is in C++, which offers considerably higher efficiency in 3D rendering than Java3D.

The data processing and mapping component is currently implemented in two steps. The processing step converts the value of each entity attribute to an internal representation, normally as integers. The internal representation of the visualization is represented as a scene graph allowing the management of complex visualizations. A scene graph consists of 3D objects, called nodes, arranged in a tree structure. Complex objects are composed of collections of other simpler objects. The visualization is rendered by traversing the tree. Scene graph objects are constructed by creating a new instance of the desired class and are accessed and manipulated using the methods of the class. Nodes can be added or removed from the scene graph dynamically allowing run time user interaction. Open Inventor provides a number of customizable manipulators to handle user interactions. sv3D uses a standard Open Inventor file format to load and store the 3D scene database and exchange with other applications.

In addition, sv3D is designed such that the user can extend its functionality easily. The core components of sv3D are designed

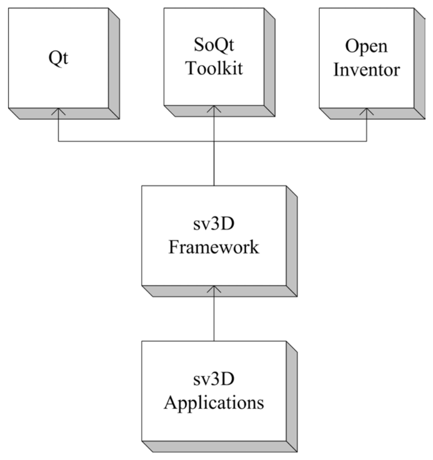

**Figure 5. sv3D architecture**

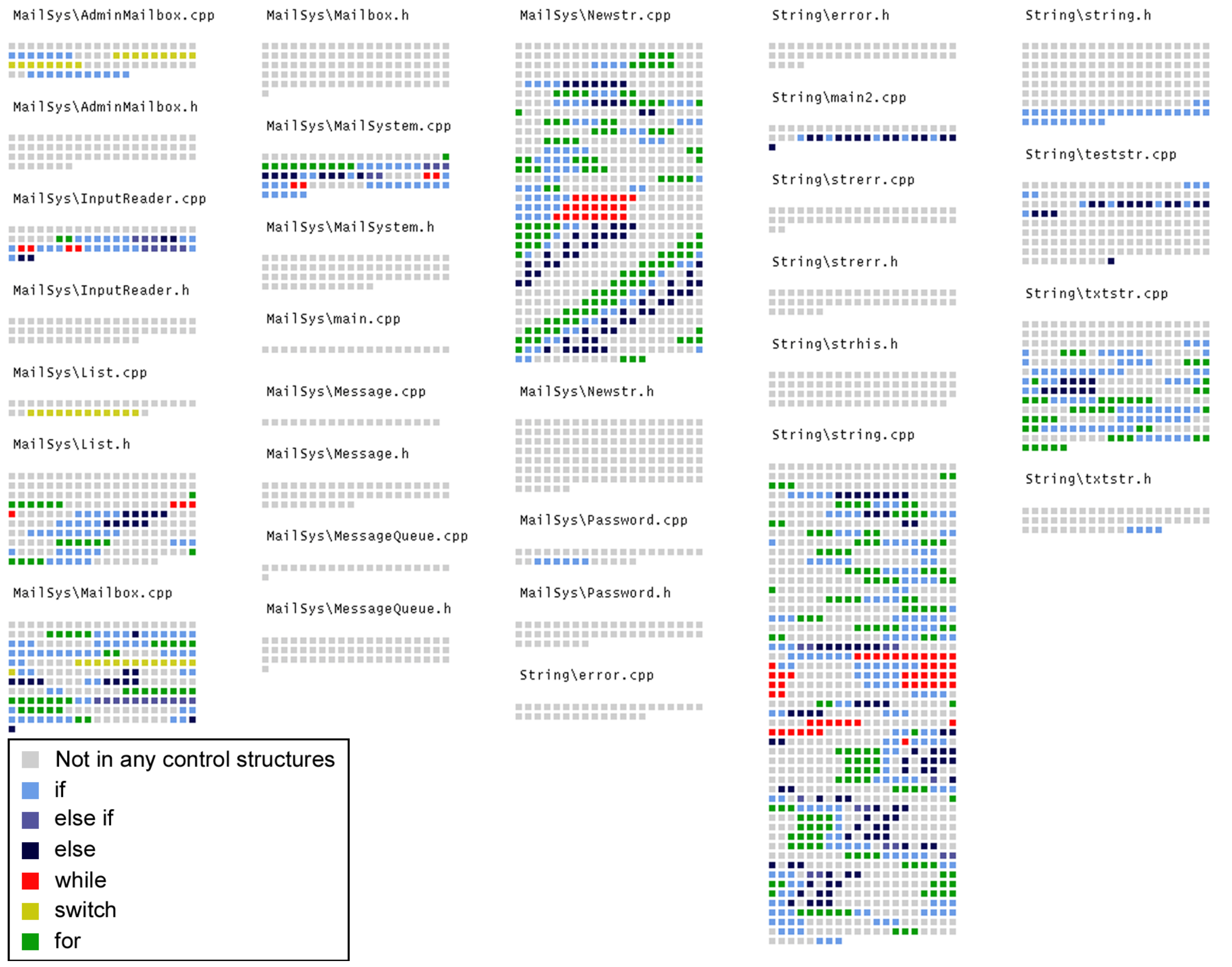

**Figure 6. A 2D overview of a system containing 30 C++ source code files (approx. 4000 LOC). Each file is mapped to a container and the name of the file is shown on top of the container. Color is used to show control structures.**

as an application framework. A number of hot spots are provided that allow the user to customize the framework and generate applications that best suit its needs. The GUI can be extended, new methods for mapping and new visual elements can be defined. In addition the user can extend the framework to define collaboration with other applications.

## 7 Applications of sv3D

SeeSoft-like tools have a variety of uses in assisting the user solving software engineering tasks. Obviously, sv3D can be used for all these tasks such as: fault localization [Jones et al 2001], visualization of execution traces [Reiss 2001], source code browsing [Griswold et al 2001, Hannemann and Kiczales 2001], impact analysis, evolution, slicing [Ball and Eick 1996], etc.

In addition, by allowing visualization of additional information (via 3D), sv3D can be used for solving other more complex tasks. For example, in the case of Tarantula [Jones et al 2001], using height (sv3D) instead of brightness will improve the visualization and make the user tasks easier.

To show how the use of 3D brings advantages over the standard SeeSoft view, we present an example based on the one described in [Ball and Eick 1996]. We chose to represent the implementation of a simple voice-mail system. The software system consists of 30 C++ source code files and has approximately 4000 lines of code. Figure 2 depicts a 2D overview of this system. This view is similar to a SeeSoft pixel representation. Each file is represented by a container; the file name is indicated on top of the container, and the heights of the cylinders are zero. Color is used to show the nesting level of a statement (quantitative data). In this example, the deepest nesting level is 4.

The same system is shown in figure 6 (please see above). In this view the color is used to represent the control structure to which the statement belongs. The following control structures are represented: if, else if, else, while, switch, and for. Also the statements that are not inside any control structure are colored. If a structure is contained within another (e.g., there is an *if* statement within a for loop) the color of the poly cylinder will show the included structure. One problem with this representation is that the user cannot differentiate between nested *for* statements and contiguous *for* statements. To do that, the user needs both views (i.e., from figure 2 and figure 6) available and switch between them. This is one of the situations when combining the two views, by using height to map to one of the attribute is highly beneficial. Figure 4 shows a view of the same mailing system with color mapped to control structure (just as in the view from figure 2) and nesting level mapped to the height of the cylinders. In order for the user to perceive the height of the cylinders, containers need to be brought closer to the camera, rotated, and scaled. In figure 4 several of the containers were moved and rotated. Two of them have active manipulators for stretching (the handle box) and for rotating (the track ball).

As discussed before, occlusion can hamper the user's efforts. Figure 3b and 3c show how transparency and elevation can be used in this example to counter the occlusion. Both techniques are used on one of the containers from the mailing system example.

In addition to using height, we could also use depth, shape, or texture, as indicated in figure 1. For example, the depth of the cylinder can be used to show method limits and texture to show the difference between declarations and implementations.

## 8 Conclusions and Future Work

The paper presents sv3D, a framework for software visualization. The framework uses 3D metaphors to represent source code and related attributes. It is based on the SeeSoft [Eick et al 1992] pixel representation and the 3D File Maps [Reiss 2001]. It brings a number of extensions to these concepts, especially in regard with the manipulation of the 3D structures. Through a more flexible mapping and use of 3D, the representation is able to show more information than previous SeeSoft-type software visualization tools. Using transparency, elevation and special 3D manipulators, sv3D overcomes many of the shortcomings of 3D visualizations such as occlusion. In addition, by using Open Inventor and Qt as support for the implementation we ensure portability and efficiency, which is critical for the success of 3D renderings. The presented examples, while simple, show how using 3D allows the representation of multiple attributes in one view.

Several aspects and extensions of sv3D need to be addressed. We plan to allow definition of mappings that will represent the software system at higher abstraction levels. For example, a container can be mapped to a function, a class, a hierarchy of classes, or a package, rather than just one source code file. The position of the cylinders within a container currently map to the position of the associated lines of text in the source code. In the future versions, position of the cylinder within a container will represent some other type of information. For example, if the container represents a class, the declaration part could be shown in a different part of the container than the implementation part.

In its current version, sv3D only represent poly cylinders with 4 edges and uniform fill. Variable number of edges will be supported and also different textures. We need to define these visual attributes very carefully to ensure their usefulness. As mentioned previously, containers in the 3D space may be connected by edges to form a 3D graph. This will allow representation of hierarchical data and also diagrammatic visualizations such as UML class diagrams.

Several aspects are important to make sure that sv3D fully exploits the advantages of the 3D space. First, a stereoscopic version (sv3Ds) is being implemented. This will be used with passive stereo displays and allow the user to experience depth of the image through stereopsis. In addition, the current version of sv3D is already designed to be used on dual monitors. One monitor can be used exclusively for the rendering, while the other for displaying the user controls and textual information.

One of the major problems of software visualization tools is scalability. By using the 3D space, sv3D deals with the real estate problem. However, efficiency is the limiting factor for 3D renderings, in general. In the current version, sv3D performs exceptionally well in representing up to 40-50 KLOC. For larger software systems the performance of the rendering and user interaction is reduced. We are working on making the rendering more efficient. We expect that the next version will work very fast representing systems in the 100 KLOC range.

Finally, we need to conduct controlled user studies to better assess the advantages and disadvantages of the sv3D, as well as the efficiency issues.

## 9 Acknowledgments

This work was supported in part by grants from the Office of Naval Research N00014-00-1-0769 and the National Science Foundation CCR-02-04175.

## 10 References

BALL, T. and EICK, S. 1996. Software Visualization in the Large. *Computer*, vol. 29, no. 4, April, pp. 33-43.

CHUAH, M. C., ROTH, S. F., MATTIS, J., and KOLOJEJCHICK, J. 1999. SDM: Selective Dynamic Manipulation of Visualizations, in *Readings in Information Visualization Using Vision to Think*. S. K. Card, J. D. MacKinlay and B. Shneiderman, Eds., San Francisco, CA Morgan Kaufmann, pp. 263-275.

COIN3D. 2002. Coin3D. Webpage, Date Accessed: 12/2002, http://www.coin3d.org.

DOS SANTOS, C. R., GROS, P., ABEL, P., LOISEL, D., TRICHAUD, N., and PARIS, J. P. 2000. Mapping Information onto 3D Virtual Worlds. in *Proceedings of International Conference on Information Visualization (IV '00)*, London, England, July 19-21.

EICK, S., STEFFEN, J. L., and SUMMER, E. E. 1992. Seesoft - A Tool For Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, vol. 18, no. 11, November, pp. 957-968.

EICK, S. G. 2000. Visual Discovery and Analysis. *IEEE Transaction on Visualization and Computer Graphics*, vol. 6, no. 1, January/March, pp. 44-58.

FAVRE, J.-M. 2001. A Flexible Approach to Visualize Large Software Products. in *Proceedings of ICSE'01 Workshop on Software Visualization*, Toronto, Ontario, May 12-13.

GRAHAM, M., KENNEDY, J. B., and HAND, C. 2000. A Comparison of Set-Based and Graph-Based Visualisations of Overlapping Classification Hierarchies. in *Proceedings of AVI 2000*, Palermo, Italy, May 23-26.

GRISWOLD, W. G., YUAN, J. J., and KATO, Y. 2001. Exploiting the Map Metaphor in a Tool for Software Evolution. in *Proceedings of 23rd IEEE International Conference on Software Engineering (ICSE'01)*, Toronto, Ontario, May 12-19, pp. 265-274.

HANNEMANN, J. and KICZALES, G. 2001. Overcoming the Prevalent Decomposition in Legacy Code. in *Proceedings of ICSE 2001 Advanced Separation of Concerns Workshop*, Toronto, Canada, May 15.

HUBONA, G. S., SHIRAH, G. W., and FOUT, D. G. 1997. 3D Object Recognition with Motion. in *Proceedings of CHI'97*, pp. 345-346.

JONES, J. A., HARROLD, M. J., and STASKO, J. T. 2001. Visualization for Fault Localization. in *Proceedings of ICSE 2001 Workshop on Software Visualization*, Toronto, Ontario, Canada, pp. 71-75.

KEIM, D. A., HAO, M. C., DAYAL, U., and HSU, M. 2002. Pixel bar charts: a visualization technique for very large multi-attribute data sets. *Information Visualization*, vol. 1, no. 1, March, pp. 20-34.

KNIGHT, C. and MUNRO, M. 1999. Comprehension with[in] Virtual Environment Visualisations. in *Proceedings of Seventh IEEE International Workshop on Program Comprehension (IWPC'99)*, Pittsburgh, PA, 5-7 May, pp. 4-11.

KNIGHT, C. and MUNRO, M. 2001. Mediating Diverse Visualisations for Comprehension. in *Proceedings of Ninth International Workshop on Program Comprehension (IWPC'01)*, Toronto, Canada, pp. 18-25.

MACKINLAY, J. D. 1986. Automating the design of graphical presentation of relational information. *ACM Transaction on Graphics*, vol. 5, no. 2, April, pp. 110-141.

MALETIC, J. I., LEIGH, J., MARCUS, A., and DUNLAP, G. 2001. Visualizing Object Oriented Software in Virtual Reality. in *Proceedings of International Workshop on Program Comprehension (IWPC01)*, Toronto, Canada, May 21-13, pp. 26-35.

MALETIC, J. I., MARCUS, A., and COLLARD, M. L. 2002. A Task Oriented View of Software Visualization. in *Proceedings of IEEE Workshop of Visualizing Software for Understanding and Analysis (VISSOFT 2002)*, Paris, France, June 26, pp. 32-40.

PRICE, B. A., BAECKER, R. M., and SMALL, I. S. 1993. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, vol. 4, no. 2, pp. 211-266.

REISS, S. P. 2001. Bee/Hive: A Software Visualization Back End. in *Proceedings of ICSE 2001 Workshop on Software Visualization*, Toronto, Ontario, Canada, pp. 44-48.

ROMAN, G.-C. and COX, K. C. 1993. A Taxonomy of Program Visualization Systems. *IEEE Computer*, vol. 26, no. 12, December, pp. 11-24.

SHNEIDERMAN, B. 1996. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. in *Proceedings of IEEE Visual Languages*, pp. 336-343.

STOREY, M.-A. D., BEST, C., and MICHAUD, J. 2001. SHriMP Views: An Interactive Environment for Exploring Java Programs. in *Proceedings of Ninth International Workshop on Program Comprehension (IWPC'01)*, Toronto, Ontario, Canada, May 12-13, pp. 111-112.

TAVANTI, M. and LIND, M. 2001. 2D vs 3D, Implications on Spatial Memory. in *Proceedings of IEEE Symposium on Information Visualization (INFOVIS'01)*, San Diego, CA, October 22-23, pp. 139-148.

TROLLTECH. 2002. Trolltech - Qt - Overview. Webpage, http://www.trolltech.com/products/qt/.

TUFTE, E. R. 1983. *The Visual Display of Quantitative Information*, Graphic Press.

WARE, C. 2000. *Information Visualization. Perception for Design*, Morgan Kaufmann Publishers.

WARE, C. and FRANCK, G. 1994. Viewing a Graph in a Virtual Reality Display is Three Times as Good as a 2D Diagram. in *Proceedings of IEEE Visual Languages*, pp. 182-183.

WARE, C., HUI, D., and FRANCK, G. 1993. Visualizing Object Oriented Software in Three Dimensions. in *Proceedings of CASCON'93*, Toronto, Ontario, Canada, October, pp. 612-620.

WERNECKE, J. 1994. *The Inventor Mentor*. 2nd ed., Addison-Wesley Publishing Company.

WISS, U., CARR, D. A., and JONSSON, H. 1998. Evaluating Three-Dimensional Information Visualization Designs A Case Study of Three Designs. in *Proceedings of International Conference on Information Visualisation (IV'98)*, London, England, July 29-31.