Innovations Syst Softw Eng (2005) 1: 221–230
DOI 10.1007/s11334-005-0019-8

STATE OF THE ART

Denis Gračanin · Krešimir Matković
Mohamed Eltoweissy

Software visualization

Received: 1 May 2005/ Accepted: 18 June 2005 / Published online: 29 July 2005

Abstract The field of software visualization (SV) investigates approaches and techniques for static and dynamic graphical representations of algorithms, programs (code), and processed data. SV is concerned primarily with the analysis of programs and their development. The goal is to improve our understanding of inherently invisible and intangible software, particularly when dealing with large information spaces that characterize domains like software maintenance, reverse engineering, and collaborative development. The main challenge is to find effective mappings from different software aspects to graphical representations using visual metaphors. This paper provides an overview of the SV research, describes current research directions, and includes an extensive list of recommended readings.

1 Introduction

Software visualization (SV) can be defined as "*a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration*" [60]. SV refers to the visualization of computer programs and algorithms [108] and attempts to give physical shape to shapeless or intangible software that "*disappears into disks*." The goal is to provide better comprehension of software artifacts [5].

The use of SV raises the questions [2]: What can be visualized? How? For what reasons? The effectiveness of SV is also a basic question. Research over the years has envisioned different aspects of source code, the code itself, data flow, and run-time behavior. SV has been applied in various areas like algorithm animation [20,34], software engineering, concurrent program execution [23], static and dynamic visualizations of object-oriented code [81,89], fault diagnostics [1, 95], debugging [4], and requirements analysis [24], to name a few. An extensive compilation of research relating to these fields can be found in [28,58,108].

A number of taxonomies have been developed that identify the properties of SV systems [79,85,86,94]. Attributes defined by Roman and Price [86,94] include:

– Scope and content: What is the aspect of the program being visualized?
– Abstraction: What kind of information is conveyed by the visualization?
– Form and technique: How is the graphical information being conveyed?
– Method: How is the visualization specified?
– Interaction: How can the user interact with the visualization?

Stasko and Patterson [106] identify an additional property, the level of automation provided for developing the SV system.

A task-oriented view of SV [72,75] uses the argument that no single SV tool or technique can address all visualization tasks. It is, therefore, necessary to identify the most appropriate visualization technique for the given SV task based on the SV dimensions [72]:

– Tasks: Why is the visualization needed?
– Audience: Who will use the visualization?

D. Gračanin (✉)
Department of Computer Science,
Virginia Tech, 660 McBryde Hall,
Blacksburg, VA 24061, USA
E-mail: gracanin@vt.edu
Tel.: +1-540-2312060
Fax: +1-540-2316075

K. Matković
VRVis Research Center for Virtual Reality and Visualization, Ltd.,
DonauCity-Strasse 1,
1220 Vienna, Austria
E-mail: Matkovic@VRVis.at
Tel.: +43-1-2050130100
Fax: +43-1-2050130900

M. Eltoweissy
Bradley Department of Electrical and Computer Engineering,
Virginia Tech, 7054 Haycock Road,
Falls Church, VA 22043, USA
E-mail: toweissy@vt.edu
Tel.: +1-703-5388374
Fax: +1-703-5388348

– Target: What is the data source to represent?
– Representation: How should it be represented?
– Medium: Where should the visualization be represented?

2 Evolution of software visualization

SV has progressed from using simple two-dimensional (2D) graphs [8,80,105,107,124] to three-dimensional (3D) representations [75,76,91] and, more recently, virtual environments (VEs).

2.1 2D and 3D visualization

Two-dimensional SV techniques typically involve graph or treelike representations consisting of a large number of nodes and arcs [118]. A complex software system might include thousands of such nodes and arcs. To make conceptualization and comprehension easy for the user, visualizations of such systems present pieces of the graph in different views or different windows so that the user can focus on the level of detail he desires. The software system is therefore represented in multiple windows that present to the observer different characteristics of the system under consideration. Examples of such visualization systems include Seesoft [38], SHriMP [112], GROOVE [59], and FIELD [108].

Two-dimensional visualizations may lead to cluttering a plethora of information on a flat plane. Even though pan/zoom and fisheye views have been explored [112], visualizing software in 2D does introduce a cognitive overload by presenting too much information. Stasko [104] identifies the need for an extra spatial dimension in visualizations and states that "*by adding an extra spatial dimension, we supply visualization designers with one more possibility for describing some aspect of a program or system.*" An example of the advantages of using 3D visualizations is the Cone Tree concept developed at Xerox PARC [104,121]. It has been claimed that the cone tree can display up to 1000 nodes without visual clutter, which is far beyond the capabilities of a 2D visualization. The developed 3D visualization presents structured information such as computer directories and project plans. In line with representing the execution time behavior of object-oriented code [59] in two dimensions, Stasko [104] discusses the development of a system called POLKA-3D to represent the same visualizations as a 3D animation.

Ware et al. [121] developed a system called GraphVisualzer3D to visualize object-oriented code in 3D. They suggest that perception is less error prone if software objects are mapped to visual objects, as there is a natural mapping from the former to the latter. They present the results of experiments that analyzed perception in 2D and 3D and conclude that there is encouraging empirical evidence that error rates in perception are less in 3D visualization. One major advantage of 3D visualization is that it allows a user to perceive the depth of a presented structure. With 3D visualization, users can zoom in or walk around structures or choose another angle (by rotating the design) and hidden structures in a software system may become evident. Three-dimensional visualization might help identify new metaphors, fostering new ideas with respect to design principles [41]. The hierarchy of relations and dependencies in design or source code would also become more readily apparent because of the added depth. It can also help to faster develop a "mental model" in the mind of the user.

Another example of visualizing large nested graphs is the NV3D system [84], which has been tested with graphs containing more than 35,000 nodes and 100,000 relationships. The NV3D system uses techniques like rapid zooming, elision, and 3D interactive visualization to display nested graphs. Both the NV3D system and the POLKA-3D system [104] analyze issues like spatial navigation, layout, semiotics, and common uses of the third dimension to represent characteristics like value, structure position, history of computation, state of computation, and aesthetics to refine the appearance of a 3D visualization.

Three-dimensional visualization has been explored for all areas where 2D visualization is used, including metrics-based visualization of object-oriented programs and visualization to track software errors, isolate problems, and monitor progress of development [18,19,67]. Three-dimensional UML (Unified Modeling Language) representations have also been researched [37].

2.2 Virtual environments

Virtual environments (VEs) open possibilities of "immersion" and "navigation" that may help to better explore software structure. VEs enable the user to interact with a representation of something familiar, namely a world with familiar objects that he/she can interact with. The concept of "worlds" in a VE can be mapped to "entities" or "components" in object-oriented code or a software system. It is possible for all software artifacts from requirements to source code to be represented and linked in a VE to improve comprehension. VEs would enable users to navigate through these links faster and in a more intuitive manner than 2D representations or even 3D structures.

Software systems are large complex systems composed of multiple components. To effectively comprehend these systems, it is necessary to provide varying levels of detail. Any user attempting to understand the system must be able to zoom out and in to each level of detail as necessary. 3D visualizations and VEs allow a user to concentrate on one aspect of the world in detail while providing a distant view of other aspects that are situated farther away. As the user moves close to each entity or visual component, it comes to "life" or presents a higher level of detail. This technique, called elision, is a major property of VEs that abstracts distant objects and details closer objects. The user can move back and forth between objects or structures in this world and rotate them around to view information that might be hidden from normal view.

Examples of SV systems that use VEs for representing object-oriented software systems are ImsoVision [71] and Software World [60]. The former represents C++ code in an immersive VE, while the latter does the same for static Java code. A major characteristic of both systems is the mapping of static properties of object-oriented code to objects in the VE. ImsoVision uses geometrical 3D shapes like platforms, spheres, and horizontal and vertical columns as visual metaphors for the characteristics of C++ code, while Software World uses real-world metaphors like the world, countries, districts, and buildings as visual metaphors for the various parts of Java code. An example of elision can be seen in the ImsoVision system [71], which hides the private attributes of an object under the platform that represents the object. The private attributes are visible only when the user rotates the platform around.

Both visualization systems visualize only static properties of code. They cannot be used to characterize the run-time behavior of an object-oriented system. While it is evident that VEs provide a far richer experience than 2D visualizations for a user attempting to comprehend a software system, it is necessary to further investigate metaphors and representations that allow us to move beyond visualizing static code [2].

2.3 Distributed VEs

A distributed (networked) VE (DVE or net-VE) is a software system in which multiple users interact with each other in real time, even though those users may be physically located around the world [99]. The users have a shared sense of space, a shared sense of presence, a shared sense of time, a way to communicate, and a way to share [99]. DVEs can be used for collaborative SV-based applications dealing with large and distributed software projects including coding, maintenance, and interactive visualization [2].

WYSIWIS (What You See Is What I See) is the basic abstraction that guides such multiuser interfaces, and the design provides a sense of teamwork. WYSIWIS is crucial for collaboration; however, some research has indicated that strict objectivity is too inflexible. It may actually hinder collaboration in some cases since the users are forced to agree on a common representation and can only see the same things instead of being able to tailor their representation of the virtual scene to meet their needs [103, 109, 110].

Indeed, collaboration in the real world often proceeds without the participants having access to the same information. This has led to the development of the Relaxed-WYSIWIS concept. Snowdon introduced the term "subjective views" for the concept of multiple perspectives in VEs [103]. A subjective VE can give the user the ability to control the presentation style to best suit her working needs.

SOLVEN is a model to support subjective views [100]. The core feature of SOLVEN is an access matrix, which defines the representation of individual objects for individual users. The matrix defines an object's view in terms of two independent factors, appearance (differing geometric definition) and modifier (highlighting and deemphasizing abilities).

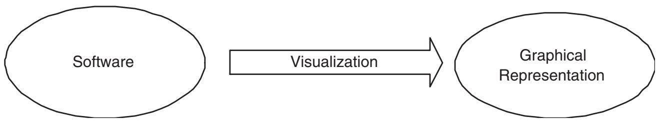

Fig. 1 Mapping software to a graphical representation

VR-VIBE is a multiuser 3D visualization of a collection of documents or document references [102]. The visualization is structured using a 3D spatial framework of keywords, called points of interest, or POIs.

Grimstead et al. [48] describe the use of a distributed, collaborative grid-enabled visualization environment. The resource-aware visualization environment (RAVE) was deployed as Web Services and accessed using a simple PDA.

Efforts like these clear the path for large-scale, multiuser.

3 Metaphors in SV

Building on ideas stated in the previous section, a metaphor can be defined as "*a rhetorical figure whose essence is understanding and experiencing one kind of thing in terms of another*" [63]. Metaphors in the medium of representation affect the expressiveness of the visualization. Metaphors might be abstract geometrical shapes (as in ImsoVision, NV3D, GraphVisualizer3D) or they might be real-world entities (as in Software World). While it is true that a user would be more familiar with a real-world metaphor, the visual complexity of the metaphor should not affect the effectiveness of the visualization. Roman and Cox [93] represent the role of the visual metaphor in a program visualization as shown in Fig. 1 [2].

Evidently, a metaphor is the entity that gives shape to the different faces of intangible software [2]. The next questions in investigating VE visualization techniques are: What are the characteristics of an effective metaphor for a VE? What are the desirable characteristics of a VE for visualizing an object-oriented system? It is necessary to state the desirable properties of a metaphor for SV in 2D, 3D, or VEs. While considering the properties of a metaphor, issues that arise regarding the characteristics of the SV system can also be discussed.

Mackinlay discusses graphical design issues on the basis of two criteria: expressiveness and effectiveness [70]. Expressiveness refers to the medium used to express the graphical representation, and effectiveness is the extent to which the representation is effective for comprehension of the visualized information. These two criteria form the basis on which we propose our design issues for effective visualizations. To be effective and meaningful, any visualization system should consider the following key areas.

(1) Scope of the representation: Visualizing complex, real-time systems can create chaos if the scope of the SV is

not defined. Scope, as identified by Price et al. [86], is isolating the characteristics of the system that the visualization will address. The SV might choose to address static or dynamic features of the software, or it might choose to represent control flow, data flow, dependencies, or all three. For example, visualization of Java source code might address the classes in other packages that a particular class depends on or inheritance hierarchies for a class or interface dependencies for a class, to name a few possibilities.

(2) Medium of representation: The type of information being visualized and the level of detail required in the visualization are just two factors that dictate the type of output medium needed. If the system to be visualized is relatively small and if detail like complexity of the source code, version history, detailed dependency navigation, or linking of the graphical representation to source code is not needed, a simple 2D graph is sufficient. If, however, the system to be visualized should provide detailed information like security vulnerabilities in the code and design or if the representation should present varying levels of information about the system without overwhelming the user, then 3D visualizations might be considered.

(3) Visual metaphor: Metaphors in the medium of representation affect the expressiveness of the visualization. Metaphors might be abstract geometrical shapes (as in ImsoVision, NV3D, GraphVisualizer3D, and other 2D representations) or they might be real-world entities (as in Software World). The visual complexity of the metaphor should not affect the effectiveness of the visualization. However, in the case of DVEs [10], users might feel more comfortable interacting with their colleagues in a real-world immersive VE.

   (a) *Consistency of the metaphor*: The metaphor or the mapping from software artifacts to the representations should be consistent throughout the visualization. Multiple software artifacts cannot be mapped to the same metaphor. Similarly, a software artifact cannot be mapped to multiple metaphors. In a VE, the metaphor should be consistent with the world it is present in.

   (b) *Semantic richness of the metaphor and complexity*: The metaphor chosen should be rich enough to provide mappings for all aspects of the software that need to be visualized. The scope of the representation determines to a certain extent the nature of the metaphor to be chosen. There should be enough objects or equivalent representations in the metaphor for the software entities that need to be visualized. The SV should not divert the user from the information that the SV system is attempting to convey. The VE should provide pertinent representations without giving the user the impression of immersion in endless space. Similar views can be found in [123].

(4) Abstractedness: The user of the visualization system should be able to focus away from certain parts of the representation and focus in detail on other parts of the representation. This is the property of elision (used in NV3D [84]) that permits different users to focus on the level of detail they desire. For example, if a visualization system should aid an evaluator in discovering security vulnerabilities, the evaluator would look for different levels of detail (say, low-level representations that map to source code) as opposed to a user who will be interested only in visualizing if any security problems exist in the system. This ability to zoom in and zoom out is what makes navigation through a 3D system easier than understanding a 2D representation. Roman and Cox [93] identify different levels of abstractedness, namely direct representation, structural representation, synthesized representation, and analytical representation.

(5) Ease of navigation and interaction: Ease of navigation is obviously a major design issue when constructing a visualization. The user should understand what is presented and what level of abstraction in the system he is currently at. It should be easy for the user to move back and forth between different views or different worlds (in the case of VEs). Also, the nature of the medium of representation would affect the level of navigation a user expects to have in the visualization. Three-dimensional visualizations should allow users to rotate the entities around for different angles of view. It should be possible to hide or "close" objects that are not of interest by clicking on them or interacting with them in other ways.

(6) Level of automation: Automation specifies the degree to which the construction of the SV system is automatic. Effective visualizations would need to be fully automated for SV to be more widely used.

4 Software visualization tools and applications

Based on concepts and developments in information visualization [14,111,117], usability [77], and software engineering [16,42,68], new SV frameworks, notations [22], query languages [87], and techniques are proposed [26,29,32]. New SV models enable interactive, online configuration of SV views and mappings (Vizz3D [83]) and better support for software comprehension [12,82,116].

The *rube* framework presents models (multimodels) and their visualizations that are based on user-specified metaphors and aesthetics [53]. The *RiOT* framework can be used to manage testing and provide dynamic visualization of heterogeneously distributed component-based applications [46]. The *Source Viewer 3D (sv3D)* uses a 3D metaphor to represent software system and analysis data that extends the Seesoft pixel metaphor by rendering the visualization in a 3D space [73].

Many SV tools have been developed for specific aspects of software design and development [113,115]. CodeCrawler is an example of a lightweight SV tool that combines metrics information with its visualizations [64,65]. SV tools can be integrated within an integrated development environment (IDE) such as Eclipse [69]. SV tools can also be accessed on the Web [31] and presented as Web services [33].

Object-oriented aspects are often a topic of SV research [51] that includes evolution of class hierarchies [47], versioning [11,96], run-time visualization [101], metrics [57], and component-based software [40]. It also includes C++ [62] and Java [17,45,88] programming languages, as well as UML [54,74].

Other SV areas include formalisms [3], metrics [9,66, 98], slicing [27,92,90], and XML [52,78,114], to name a few. The remainder of this section discusses SV for software evolution, software security, data mining in software systems, algorithms, and software engineering education.

4.1 Software evolution

Software is continually changing and evolving [39]. Today's typical software system is a complex beast spanning millions of lines of code. Manually analyzing the effects and impacts of changes [56] to a software system is a labor-intensive and often error-prone task. Visualizing the evolution of the system may be accomplished, in part, through visualizing the version history of a software system. Visualizing version history typically involves visualizing metrics such as number of lines of code in a particular version of a file included in the system, the percentage of growth and the percentage of change in the system, defect density, and change complexity measures [44]. This section discusses advances in version history visualization.

An important construct in most of the works discussed below is that of a modification request or maintenance request (MR). A software system is assumed to consist of subsystems. Each of these subsystems has a number of modules. The modules include the program elements, which may be a collection of one or more source files, and an MR is the information representing work to be done to each module. Deltas are part of an MR, representing editing changes made to individual files in order to complete an MR. A file can be checked out, edited, and then checked in [6,43]. Note that this terminology works well with version control systems that can record the parent MR for each delta list, along with the number of lines added, deleted, and modified by that change. Alternatively, as in the case of CVS, there is no concept of an MR. Changes made to files are recorded as part of a checkout or update of modules.

The forerunner to most of the attempts at version history visualization can be seen in Seesoft, developed by Eick et al. [38]. Seesoft is a tool for visualization of line-oriented software statistics. Seesoft can visualize up to 50,000 lines of code and provides information about various statistics like the number of files under version control, the age of each line of code in a file, the number of lines of code in each file, the MR that touched a particular line of code in a file, and the number of times the line was executed during testing. Seesoft uses a row–column metaphor. Each column represents a file and the rows in each column represent the number of lines of code in the file. It allows user interaction to decipher interesting patterns in the version history and also provides information about the dates of changes, the reasons for changes, the developer who changed the code, etc.

Another significant effort in version history visualization is presented by Gall et al. [44]. Their work uses color and the third dimension effectively to visualize software release histories. The metrics that they visualize include size of the system in terms of lines of code, age in terms of version numbers, and error-proneness in terms of defect density. Their Software Release History visualization is composed of three entities:

– Time: The visualization is grouped according to the release sequence number (RSN). A snapshot of the system at the time of each release enables the end user to see the evolution of files between releases. Addition, deletion, and modification of files between releases are clearly visible.
– Structure: The system is decomposed into subsystems. Each subsystem is decomposed into modules, and each module comprises the source code files.
– Attributes: These include version number, size, change complexity, and defect density.

The visualization was created using Java and virtual reality modeling language (VRML) to render and navigate the 3D spaces. The end user can navigate through the visualization and use the mouse to extract information about the structure of the entire system for a release or focus on a particular subsystem and extract the values of the modules in the subsystem. The paper concludes with the suggestion that other metrics like lines of code, complexity measures, and defect density can be visualized. It also suggests the automatic detection of change patterns to identify module dependencies. The type of change pattern to be investigated could be input by the user.

Gall et al. [43] discuss another application of version history visualization. They present an approach that uses information in the release history of a system to uncover logical dependencies and change patterns among modules. They have developed a technique that automatically extracts information about the logical dependencies among the modules of a system. These logical dependencies are different from the syntactic dependencies that are evident through source code analysis. The authors propose the idea of change sequence analysis and change report analysis to identify logical dependencies. The change sequence analysis lists the releases in which a module was changed. Different modules can be compared on the basis of such change sequences, and common change patterns can be identified.

Lanza and Ducasse in [35,36] study the evolution of classes in a system using a combination of SV and software metrics. The visualization is 2D, with rows representing the classes in a system and columns denoting the version of the system. The first column would represent version 1 of the system, the second version 2, and so on. The number of methods in the class decides the width of each rectangle representing a class, while the number of instance variables in the class decides the height of the rectangle. The authors suggest that other metrics can also be used effectively to represent a class. This metaphor allows easy visualization of the number of classes in the system, the most recent classes that have been added to the system, and growth and stagnation phases in the evolution of the system. An innovative technique here is the classification of classes based on the kind of changes made to them over the different versions of the system.

Koike [61] presents a 3D visualization framework (VRCS) by means of which a user can interact with a version control system. Versions of the files in a system are represented as cubes arranged along the z-axis, ordered by time. Releases that link versions of various files together are represented as circles. VRCS has been implemented using OpenGL/C and serves as an interface to RCS. Users can check out, edit, and check in files, view differences between two cubes/versions of a file, retrieve all the files that comprise a release, and even build the executable file for a release. The authors also suggest some mechanism that enables the user to select the amount of graphical information presented. VRCS can only be applied to single-user systems.

Finally, CVSscan [120] is an integrated multiview environment that helps users to better understand the status, history, and structure of the source code, as well as, for instance, the roles played by various contributors.

4.2 Software security

One possible application of SV is in the area of software security analysis. For example, visualizing the results of dependency analysis and traceability analysis in a software system can help identify the potential security vulnerabilities if proposed changes to a system are implemented.

Conti and Abdulla [21] discuss the use of SV for security analysis. The authors examine the visual fingerprints left by a wide variety of popular network attack software tools to provide better understanding of the specific methodologies used by attackers as well as the identifiable characteristics of the tools themselves. The techniques used in the paper are entirely passive in nature, making them virtually undetectable by attackers. The paper explores the application of several visualization techniques including parallel coordinate plots and scrolling plots for their usefulness in identifying attack tools, without the typical automated intrusion detection system's signatures and statistical anomalies. These visualizations were tested using a wide range of popular network security tools, and the results showed that in many cases, the specific tool can be identified.

While Conti and Abdullah [21] focused on attack tool fingerprints, Yoo in [122] studied virus fingerprints. Their paper focused on visualizing Windows executable viruses using self-organizing maps (SOMs) without using virus-specific signature information as a prior stage of detecting computer viruses. SOMs are visualized using the unified distance matrix. The paper addresses the fact that each virus has its own character to be distinguished, although it is inserted in the executable file. Yoo observed that the virus features cannot be hidden through the SOM visualization; these features are like a strand of DNA that determines a person's unique genetic code. The authors studied how virus codes effect the whole program projection, without each virus signature, and described how a virus pattern in Windows executable files indicates its family. The paper also shows that variants of a virus can also be covered with the specific virus's mask, which is produced by SOM.

4.3 Data mining in software systems

Visualization is employed in data mining to visually present already discovered patterns and to discover new patterns visually. Success in both tasks depends on the system's ability to present abstract patterns as simple visual patterns [119].

SV is used in Burch et al. [15] for mining software archives. A software archive is comprised of the information stored by a configuration management system and related tools. This information includes versions, changes, bug databases, and electronic mail. The authors claim that the relevance to a project or set of projects of many software engineering rules published in the literature is unclear; either the rules are too general or results of the case studies cannot be transferred, because the constraints of the case studies are not well documented. The authors use visual data mining for extracting rules from software archives for validation of the application of these rules and also for discovering new project-specific rules. The authors developed EPOSee to visualize n-ary association and sequence rules and to study software evolution and relations based on hierarchically ordered items. EPOSee uses pixelmaps and parallel coordinate views and provides visualizations that conform to Ben Shneiderman's visualization mantra: "Overview first, zoom and filter, then details on demand" [97]. As an example, the paper studies the large software archive of the *Mozilla* open source project.

Vityaev and Kovalerchuk [119] propose a technique called inverse visualization (IV) to address the problem of visualizing complex patterns. Their approach does not use data "as is" and does not follow a traditional sequence: discover pattern—visualize pattern. Instead, the sequence proposed [119] is: convert data to visualizable form—discover patterns with predefined visualization. IV is based on specially designed data preprocessing that permits the discovery of abstract patterns that can be presented by simple visual patterns. In the paper, the feasibility of solving inverse visualization tasks is illustrated on functional nonlinear additive dependencies that are transformed into simple and intuitive visual patterns.

4.4 Algorithms and software engineering education

Algorithm and software engineering visualization can help instructors to explain and learners to understand algorithms and software engineering principles and practices [55]. For example, an algorithm can be animated showing relevant parameters and variables, the current state, and a visual representation of the objects being manipulated, as well as an animated formal description of the algorithm. Complex model structures are simplified at a high level of abstraction to highlight only the important aspects. Details can then be shown at lower levels of abstraction by omitting irrelevant details. For better comprehension, the designer scales down data to coarser structures and slows down algorithms that process data. Smooth transitions between different states of moving objects can make it easier to follow the way the algorithm works on graphical representations of data structures.

A recent proposal by Baloian et al. [7] concerns an approach to developing algorithm visualization that seeks to construct context-dependent interfaces allowing the learner to interactively control and explore the actions in the algorithm. The proposed approach replaces standard control using mouse clicks on graphic displays with a concept called concept keyboards (CKs) mirroring the inherent logical structures of the algorithm under investigation. The CK concept separates control elements, data input, and visual output objects by means of an adequate concept keyboard application to be used to configure keyboards, collect startup data, and visualize user actions.

A key on a CK has a special meaning (concept) associated with it instead of just a label. Each key of the CK will be mapped to the execution of an existing method available in the algorithm implementation. In order to choose the interesting events (those that are crucial for understanding the algorithm), the designer has a simple GUI displaying the available actions and allowing them to select the relevant ones. CKs are used to trigger more complex semantic actions on the system in which they have been implemented. The special software supplied allows the user to redefine the function of each key and to regroup keys into fields of differing sizes. The user's attempts at manipulation of algorithms and data structures are reflected by changes in the visualization or another form of output like textual or acoustic information. This provides users, including people with sensory disabilities, with suitable interfaces that may enhance the comprehension of the algorithm being presented.

The GRASP, and its successor jGRASP, were developed in [25,50] with the goal of enhancing software system comprehension efficiency and effectiveness. The developed visualization tools support well-defined cognitive processes employed during a comprehension task, such as top-down, bottom-up, and mixed comprehension models. Grissom et al. [49] measured the effect of varying levels of student engagement with algorithm visualization to learn simple sorting algorithms. Their results showed that learning increases as the level of student engagement increases. The authors concluded that algorithm visualization has a bigger impact on learning when students go beyond merely viewing a visualization and are required to engage in additional activities structured around the visualization.

5 Conclusions

Advances in SV are leading to its pervasive adoption for better comprehension, engineering, and consequently, enhancements in algorithm animation, software evolution, and software metrics. Development of secure software and software engineering education products is also a major benefit.

Interactive visualization can be coupled with other modalities, such as sensing or predictive methods, to provide powerful new capabilities for SV as well as other visualization domains. In addition, the fusion of visualization techniques with other areas such as data mining, grid computing, and Web Services is promoting broad-based advances, particularly in the emerging areas of visual analytics and mobile visualization. Another promising area of SV advancement is collaborative VEs that will lead to better understanding of collaborative software engineering processes.

Indeed, the importance of SV is growing, both in academia and industry [13]. A recent survey of software maintenance, reengineering, and reverse engineering studies [29,30] shows that 40% of researchers consider SV "absolutely necessary for their work" while 42% of researchers consider it "important but not critical." In addition, a significant increase in SV research is apparent in the plethora of recent conferences, workshops, and symposia on SV. For a wide spectrum of new ideas and approaches, the reader is referred to the Dagstuhl seminar "Software Visualization" (2001), the ACM Symposium on Software Visualization (2003, 2005), and the IEEE International Workshop on Visualizing Software for Understanding and Analysis (2002, 2003, 2005).

References

1. Amari H, Okada M (1999) A three-dimensional visualization tool for software fault analysis of a distributed system. In: Proceedings of the IEEE systems, man, and cybernetics conference (SMC'99), 4:194–1999
2. Asokan R (2003) Automatic visualization of the version history of a software system in three dimensions. Master's thesis, Virginia Polytechnic Institute and State University, Falls Church, VA
3. Averbukh VL (1997) Toward formal definition of conception "adequacy in visualization". In: Proceedings of the 1997 IEEE symposium on visual languages, pp 46–47
4. Baecker R, DiGiano C, Marcus A (1997) Software visualization for debugging. Commun ACM 40(4):44–54
5. Ball T, Eick SG (1996) Software visualization in the large. IEEE Comput 29(4):33–43
6. Ball T, Kim JM, Porter AA, Siy HP (1997) If your version control system could talk . . . . In: Proceedings of the ICSE workshop on process modelling and empirical studies of software engineering
7. Baloian N, Breuer H, Luther W (2005) Algorithm visualization using concept keyboards. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 7–16
8. Balzer M, Deussen O (2004) Hierarchy based 3D visualization of large software structures. In: Proceedings of the conference on visualization (VIS'04). IEEE Press, Washington, DC, p 598.4

9. Balzer M, Deussen O, Lewerentz C (2005) Voronoi treemaps for the visualization of software metrics. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 164–172,214
10. Bassil S, Keller RK (2001) Software visualization tools: survey and analysis. In: Proceedings of the 9th international workshop on program comprehension (IWPC 2001), Toronto, pp 7–17
11. Bieman JM, Andrews AA, Yang HJ (2003) Understanding change-proneness in OO software through visualization. In: Proceedings of the 11th IEEE international workshop on program comprehension (IWPC'03), pp 44–53
12. Boisvert C (2004) Supporting program development comprehension by visualising iterative design. In: Proceedings of the 8th international conference on information visualisation (IV'04), pp 717–722
13. Bril RJ, Postma A, Krikhaar RL (2003) Embedding architectural support in industry. In: Proceedings of the international conference on software maintenance (ICSM'03), pp 348–357
14. Brown M, Domingue J, Price B, Stasko J (1994) Software visualization: a CHI '94 workshop. SIGCHI Bull 26(4):32–35
15. Burch M, Diehl S, Weissgerber P (2005) Visual data mining in software archives. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 37–46
16. Burnett M, Cook C, Rothermel G (2004) End-user software engineering. Commun ACM 47(9):53–58
17. Cattaneo G, Faruolo P, Petrillo UF, Italiano GF (2004) JIVE: Java interactive software visualization environment. In: Proceedings of the 2004 IEEE symposium on visual languages and human centric computing (VLHCC'04), pp 41–43
18. Chuah MC, Eick SG (1997) Glyphs for software visualization. In: Proceedings of the 5th iternational workshop on program comprehension (IWPC'97), pp 183–191
19. Chuah MC, Eick SG (1998) Information rich glyphs for software management data. IEEE Comput Graph Appl 18(4):24–29
20. Collins TD (2003) Applying software visualization technology to support the use of evolutionary algorithms. J Vis Lang Comput 14(2):123–150
21. Conti G, Abdullah K (2004) Passive visual fingerprinting of network attack tools. In: Proceedings of the 2004 ACM workshop on visualization and data mining for computer security (VizSEC/DMSEC'04), pp 45–54
22. Costagliola G, Deufemia V, Polese G (2004) A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans Softw Eng Methodol 13(4):431–487
23. Cox P, Gauvin S, Rau-Chaplin A (2005) Adding parallelism to visual data flow programs. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 135–144
24. Cross II JH, Hendrix TD, Barowski LA, Mathias KS (1998) Scalable visualizations to support reverse engineering: a framework for evaluation. In: Proceedings of the 5th working conference on reverse engineering, pp 201–209
25. Cross II JH, Hendrix TD, Mathias KS, Barowski LA (1999) Software visualization and measurement in software engineering education: an experience report. In: Proceedings of the 29th ASEE/IEEE conference on frontiers in education, pp 12b1/5–12b1/10
26. De Pauw W, Reiss SP, Stasko JT (2001) ICSE workshop on software visualization. In: Proceedings of the 23rd international conference on software engineering (ICSE'01), pp 758–759
27. Deng Y, Kothari S, Namara Y (2001) Program slice browser. In: Proceedings of the 9th international workshop on program comprehension (IWPC'01), pp 50–59
28. Diehl S (ed) (2002) Proceedings of the international seminar on software visualization, Dagstuhl Castle, Germany, 20–25 May 2001. Revised papers. Lecture notes in computer science, vol 2269. Springer, Berlin Heidelberg New York
29. Diehl S (2005) Software visualization. In: Proceedings of the 27th international conference on Software engineering (ICSE'05). ACM Press, New York, pp 718–719
30. Diehl S, Kerren A (2002) Reification of program points for visual execution. In: Proceedings of the 1st international workshop on visualizing software for understanding and analysis (VISSOFT'02), pp 100–109
31. Domingue J, Mulholland P (1997) Staging software visualizations on the web. In: Proceedings of the 1997 IEEE symposium on visual languages, pp 364–371
32. Domingue J, Sutinen E (2002) Software visualization – editorial. J Vis Lang Comput 13(3):257–258
33. Dong J, Yang S, Zhang K (2005) VisDP: A web service for visualizing design patterns on demand. In: Proceedings of the international conference on information technology: coding and computing (ITCC'05), 2:385–391
34. Douglas S, Hundhausen C, McKeown D (1995) Toward empirically-based software visualization languages. In: Proceedings of the 11th IEEE international symposium on visual languages, pp 342–349
35. Ducasse S, Lanza M (2005) The class blueprint: visually supporting the understanding of classes. IEEE Trans Softw Eng 31(1):75–90
36. Ducasse S, Lanza M, Bertuli R (2004) High-level polymetric views of condensed run-time information. In: Proceedings of the 8th European conference on software maintenance and reengineering (CSMR'04), pp 309–318
37. Dwyer T (2001) Three dimensional UML using force directed layout. In: Proceedings of the Australian symposium on information visualisation 2001, Sydney, Australia, pp 77–85
38. Eick SG, Steffen JL, Summer EE Jr (1992) Seesoft: a tool for visualizing line oriented software statistics. IEEE Trans Softw Eng 18(11):957–968
39. Eick SG, Graves TL, Karr AF, Mockus A, Schuster P (2002) Visualizing software changes. IEEE Trans Softw Eng 28(4):396–412
40. Favre JM, Cervantes H (2002) Visualization of component-based software. In: Proceedings of the 1st international workshop on visualizing software for understanding and analysis (VISSOFT'02), pp 51–60
41. Feijs L, Jong RD (1998) 3D visualization of software architectures. Commun ACM 41(12):73–78
42. Fiutem R, Merlo E, Antonio G, Tonella P (1996) Understanding the architecture of software systems. In: Proceedings of the 4th workshop on program comprehension (WPC'06), pp 187–196
43. Gall H, Hajek K, Jazayeri M (1998) Detection of logical coupling based on product release history. In: Proceedings of the international conference on software maintenance (ICSM'98). IEEE Press, Washington, DC, p 190
44. Gall H, Jazayeri M, Riva C (1999) Visualizing software release histories: The use of color and third dimension. In: Proceedings of the IEEE international conference on software maintenance (ICSM'99)
45. Gestwicki P, Jayaraman B (2005) Methodology and architecture of JIVE. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 95–104
46. Ghosh S, Bawa N, Craig G, Kalgaonkar K (2001) A test management and software visualization framework for heterogeneous distributed applications. In: Proceedings of the 6th IEEE international symposium on high assurance systems engineering (HASE'01), pp 106–116
47. Girba T, Lanza M, Ducasse S (2005) Characterizing the evolution of class hierarchies. In: Proceedings of the 9th European conference on software maintenance and reengineering (CSMR'05), pp 2–11
48. Grimstead IJ, Avis NJ, Walker DW (2005) Visualization across the pond: How a wireless PDA can collaborate with million-polygon datasets via 9,000 km of cable. In: Proceeding of the 10th international conference on 3D web technology (Web3D 2005), pp 47–56,187
49. Grissom S, McNally MF, Naps T (2003) Algorithm visualization in CS education: comparing levels of student engagement. In: Proceedings of the 2003 ACM symposium on software visualization (SoftVis'03), pp 87–94

50. Hendrix D, Cross II JH, Barowski LA (2004) An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. In: Proceedings of the 35th technical symposium on computer science education (SIGCSE'04), pp 387–391
51. Hill T, Noble J, Potter J (2002) Scalable visualizations of object-oriented systems with ownership trees. J Vis Lang Comput 13(3):319–339
52. Hopfner M, Seipel D, von Gudenberg JW (2003) Comprehending and visualizing software based on XML-representations and call graphs. In: Proceedings of the 11th IEEE international workshop on program comprehension (IWPC'03), pp 290–291
53. Hopkins JF, Fishwick PA (2003) The *rube* framework for software modeling and customized 3-D visualization. J Vis Lang Comput 14:97–117
54. Huang S, Tilley S (2003) Workshop on graphical documentation for programmers: assessing the efficacy of UML diagrams for program understanding. In: Proceedings of the 11th IEEE international workshop on program comprehension, pp 281–282
55. Hundhausen CD, Douglas SA, Stasko JT (2002) A meta-study of algorithm visualization effectiveness. J Vis Lang Comput 13(3):259–290
56. Hutchins M, Gallagher K (1998) Improving visual impact analysis. In: Proceedings of the international conference on software maintenance, pp 294–303
57. Irwin W, Churcher N (2003) Object oriented metrics: precision tools and configurable visualisations. In: Proceedings of the 9th international software metrics symposium, pp 112–123
58. Jeffery CL (1999) Program monitoring and visualization: an exploratory approach. Springer, Berlin Heidelberg, New York
59. Jerding DF, Sasko JT, Ball T (1996) Visualizing message patterns in object-oriented. Technical report GIT-GVU-96-15, Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta
60. Knight C, Munro M (1999) Comprehension with[in] virtual environment visualisations. In: Proceedings of the 7th international workshop on program comprehension, Pittsburgh, PA, pp 4–11
61. Koike H (1993) The role of another spatial dimension in software visualization. ACM Trans Inf Syst 11(3):266–286
62. LaFollette P, Korsh J, Sangwan R (2000) A visual interface for effortless animation of C/C++ programs. J Vis Lang Comput 11(1):27–48
63. Lakoff G, Johnson M (1980) Metaphors we live by. University of Chicago Press, Chicago
64. Lanza M (2003) CodeCrawler: lessons learned in building a software visualization tool. In: Proceedings of the 7th European conference on software maintenance and reengineering (CSMR'03), pp 409–418
65. Lanza M (2004) CodeCrawler: polymetric views in action. In: Proceedings of the 19th international conference on automated software engineering (ASE'04), pp 394–395
66. Lanza M, Ducasse S (2003) Polymetric views: a lightweight visual approach to reverse engineering. IEEE Trans Softw Eng 29(9):782–795
67. Lewerentz C, Simon F (2002) Metrics-based 3D visualization of large object-oriented programs. In: Proceedings of the 1st international workshop on visualizing software for understanding and analysis (VISSOFT'02), Paris, pp 70–77
68. Lieberman H, Fry C (2001) Will software ever work? Commun ACM 44(3):122–124
69. Lintern R, Michaud J, Storey MA, Wu X (2003) Plugging-in visualization: experiences integrating a visualization tool with Eclipse. In: Proceedings of the 2003 ACM symposium on software visualization (SoftVis '03). ACM Press, New York, pp 47–56
70. Mackinlay JD (1986) Automating the design of graphical presentations of relational information. ACM Trans Graph 5(2):110–141
71. Maletic JI, Leigh J, Marcus A, Dunlap G (2001) Visualizing object-oriented software in virtual reality. In: Proceedings of the 9th international workshop on program comprehension (IWPC'01), Toronto, pp 26–35
72. Maletic JI, Marcus A, Collard ML (2002) A task oriented view of software visualization. In: Proceedings of the 1st international workshop on visualizing software for understanding and analysis (VISSOFT'02), pp 32–40
73. Maletic JI, Marcus A, Feng L (2003) Source viewer 3D (sv3D): a framework for software visualization. In: Proceedings of the 25th international conference on software engineering (ICSE'03), pp 812–813
74. Malloy BA, Power JF (2005) Exploiting UML dynamic object modeling for the visualization of C++ programs. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 105–114
75. Marcus A, Feng L, Maletic JI (2003a) 3D representations for software visualization. In: Proceedings of the 2003 ACM symposium on software visualization (SoftVis'03). ACM Press, New York, pp 27–36
76. Marcus A, Feng L, Maletic JI (2003b) Comprehension of software analysis data using 3D visualization. In: Proceedings of the 11th IEEE international workshop on program comprehension (IWPC'03), pp 105–114
77. Marcus A, Comorski D, Sergeyev A (2005) Supporting the evolution of a software visualization tool through usability studies. In: Proceedings of the 13th international workshop on program comprehension (IWPC'05), pp 307–316
78. Marks RM, Wilkie FG (2004) Visualising object-oriented source code complexity using XML. In: Proceedings of the 9th IEEE international conference on engineering complex computer systems navigating complexity in the e-Engineering age, pp 161–170
79. Myers BA (1990) Taxonomies of visual programming and program visualization. J Vis Lang Comput 1(1):97–123
80. Noack A, Lewerentz C (2005) A space of layout styles for hierarchical graph models of software systems. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis '05). ACM Press, New York, pp 155–164
81. Pacione MJ (2004) Software visualisation for object-oriented program comprehension. In: Proceedings of the 26th international conference on software engineering (ICSE'04), pp 63–65
82. Pacione MJ, Roper M, Wood M (2004) A novel software visualisation model to support software comprehension. In: Proceedings of the 11th working conference on reverse engineering (WCRE'04), pp 70–79
83. Panas T, Lincke R, Löwe W (2005) Online-configuration of software visualizations with Vizz3D. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 173–182
84. Parker G, Franck G, Ware C (1998) Visualization of large nested graphs in 3D: Navigation and interaction. J Vis Lang Comput 9(3):299–317
85. Price BA, Small IS, Baecker R (1992) A taxonomy of software visualization. In: Proceedings of the 25th Hawaii international conference on system sciences, 2:597–606
86. Price BA, Baecker RM, Small IS (1993) A principled taxonomy of software visualization. J Vis Lang Comput 4(3):211–266
87. Reiss SP (2002) A visual query language for software visualization. In: Proceedings of the IEEE 2002 symposia on human centric computing languages and environments (HCC'02), pp 80–82
88. Reiss SP (2003) JIVE: Visualizing java in action. In: Proceedings of the 25th international conference on software engineering (ICSE'03), pp 820–821
89. Reiss SP (2005) Tool demonstration: JIVE and JOVE: Java as it happens. In: Proceedings of the 25th international conference on software engineering (ICSE'03), pp 820–821
90. Rilling J, Mudur S (2005) 3D visualization techniques to support slicing-based program comprehension. Comput Graph 29(3):311–329
91. Rilling J, Mudur SP (2002) On the use of metaballs to visually map source code structures and analysis results onto 3D space. In: Proceedings of the 9th working conference on reverse engineering (WCRE'02), pp 299–308

92. Rilling J, Seffah A, Bouthlier C (2002) The CONCEPT project: applying source code analysis to reduce information complexity of static and dynamic visualization techniques. In: Proceedings of the 1st international workshop on visualizing software for understanding and analysis (VISSOFT'02), pp 90–99
93. Roman GC, Cox KC (1992) Program visualization: the art of mapping programs to pictures. In: Proceedings of the 14th international conference on Software engineering. ACM Press, Melbourne, pp 412–420
94. Roman GC, Cox KC (1993) A taxonomy of program visualization systems. IEEE Comput 26(12):11–24
95. Ruthruff J, Creswick E, Burnett M, Cook C, Prabhakararao S, M Fisher I, Main M (2003) End-user software visualizations for fault localization. In: Proceedings of the 2003 ACM symposium on software visualization (SoftVis'03). ACM Press, New York, pp 123–132
96. Seemann J, von Gudenberg JW (1998) Visualization of differences between versions of object-oriented software. In: Proceedings of the 2nd Euromicro conference on software maintenance and reengineering, pp 201–204
97. Shneiderman B (2002) Creativity support tools. Commun ACM 45(10):116–120
98. Simon F, Steinbrückner F, Lewerentz C (2001) Metrics based refactoring. In: Proceedings of the 5th European conference on software maintenance and reengineering, pp 30–38
99. Singhal S, Zyda M (1999) Networked virtual environments: design and implementation. ACM Press SIGGRAPH Series, Addison-Wesley, Reading, MA
100. Smith G, Mariani J (1997) Using subjective views to enhance 3D applications. In: Proceedings of the ACM symposium on virtual reality software and technology. ACM Press, New York, pp 139–146
101. Smith MP, Munro M (2002) Runtime visualisation of object oriented software. In: Proceedings of the 1st international workshop on visualizing software for understanding and analysis (VISSOFT'02), pp 81–89
102. Snowdon D, Jää-Aro KM (1997) A subjective virtual environment for collaborative information visualization. In: Virtual Reality Universe'97, Santa Clara, CA
103. Snowdon D, Greenhalgh C, Benford S (1995) What you see is not what I see: subjectivity in virtual environments. In: Framework for immersive virtual environments (FIVE'95), QMW University of London, UK
104. Stasko JT (1992) Three-dimensional computation visualization. Technical report GIT-GVU-94-33, Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta
105. Stasko JT, Muthukumarasamy J (1996) Visualizing program executions on large data sets. In: Proceedings of the 1996 IEEE symposium on visual languages, pp 166–173
106. Stasko JT, Patterson C (1992) Understanding and characterizing software visualization systems. In: Proceedings of the 1992 IEEE workshop on visual languages, Seattle, pp 3–10
107. Stasko JT, Turner CR (1992) Tidy animations of tree algorithms. In: Proceedings of the 1992 IEEE workshop on visual languages, Seattle, pp 216–218
108. Stasko JT, Domingue JB, Brown MH, Price BA (eds) (1998) Software visualization. MIT Press, Cambridge, MA
109. Stefik M, Bobrow DG, Foster G, Lanning S, Tatar D (1987a) WYSIWIS revised: early experiences with multiuser interfaces. ACM Trans Inf Sys 5(2):147–167
110. Stefik M, Foster G, Bobrow DG, Kahn K, Lanning S, Suchman L (1987b) Beyond the chalkboard: computer support for collaboration and problem solving in meetings. Commun ACM 30(1):32–47
111. Storey MAD, Fracchia FD, Müller HA (1997a) Cognitive design elements to support the construction of a mental model during software visualization. In: Proceedings of the 5th international workshop on program comprehension (IWPC 1997), Dearborn, MI, pp 17–28
112. Storey MAD, Wong K, Fracchia FD, Müller HA (1997b) On integrating visualization techniques for effective software exploration. In: Proceedings of the 1997 IEEE symposium on information visualization, Phoenix, AZ, pp 38–45
113. Storey MAD, Čubranić D, German DM (2005) On the use of visualization to support awareness of human activities in software development: a survey and a framework. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 193–202
114. Telea A, Maccari A, Riva C (2002) An open visualization toolkit for reverse architecting. In: Proceedings of the 10th international workshop on program comprehension (IWPC'02), pp 3–10
115. Tilley S, Huang S (2002) On selecting software visualization tools for program understanding in an industrial context. In: Proceedings of the 10th international workshop on program comprehension (IWPC'02), pp 285–288
116. Tudoreanu ME (2003) Designing effective program visualization tools for reducing user's cognitive effort. In: Proceedings of the 2003 ACM symposium on software visualization (SoftVis'03). ACM Press, New York, pp 105–114
117. Tufte E (1990) Envisioning information. Graphics Press, Cheshire, UK
118. van Ham F (2003) Using multilevel call matrices in large software projects. In: Proceedings of the 2003 IEEE symposium on information visualization (INFOVIS'03), pp 227–232
119. Vityaev E, Kovalerchuk B (2002) Inverse visualization in data mining. In: Proceedings of the 2002 international conference on imaging science, systems, and technology, pp 133–137
120. Voinea L, Telea A, van Wijk JJ (2005) CVSscan: visualization of code evolution. In: Proceedings of the 2005 ACM symposium on software visualization (SoftVis'05). ACM Press, New York, pp 47–56
121. Ware C, Hui D, Franck G (1993) Visualizing object oriented software in three dimensions. In: Proceedings of the 1993 IBM Centre for Advanced Studies conference (CASCON'93), Toronto, pp 612–660
122. Yoo I (2002) Visualizing windows executable viruses using self-organizing maps. In: Proceedings of the 2004 ACM workshop on visualization and data mining for computer security (VizSEC/DMSEC'04), pp 82–89
123. Young P, Munro M (1998) Visualizing software in virtual reality. In: Proceedings of the 6th international workshop on program comprehension (IWPC'98), Ischia, pp 19–26
124. Zernik D (1995) Visualizing programs using graphs. In: Proceedings of the 18th convention of electrical and electronics engineers in Israel, pp 1.3.3/1–1.3.3/4