Graph visualization and applications in software visualization

Vizualizácia grafov a ich aplikácie pri vizualizácii softvéru

Peter Kapec

Key words:

graph visualization, software visualization, hypergraph, magic lenses vizualizácia grafov, vizualizácii softvéru, hypergraf, magické šošovky

Abstract:

Graph visualization is an active and fast developing research are. Graph visualizations have found applications in many fields, especially in fields in which the amount of structured information is huge and difficult to understand without visualization. In this paper we discuss the applications of graph visualization to visualize software at various levels of abstraction. We present our software visualization approach based on hypergraph representations and related interaction methods.

Introduction

Graph visualization has found many applications in various research and application domains. Graphs are often used to represent information and relations between different types of data. Although many new approaches have been published in recent years, open problem still exist. Currently research focuses on graph-based knowledge representations, social network analysis and other Internet related areas. A common aspect of current research is the problematic of visualization of very huge graphs and how to present them in an interactive and comprehensible way. Although graph visualization algorithms are often general, in specific applications the graph layout algorithms have to be modified to empathize specific graph parts or even vastly modified to be at least partially readable and comprehensible. Among various data types that can be represented trough graphs, graphs can be utilized to represent software at various levels of abstraction, ranging from source code to software development process. In this paper we discuss problems of graph visualization and focus on applications in software visualization. We build on previous work in which generalized graphs are used to represent software structures and present our visualization method that is based on hypergraphs. Hypergraphs are generalized graphs that allow to connect more than 2 nodes – commonly used graphs, in which edges connect exactly two node, are just a special case of hypergraphs. Hypergraphs, and other generalized graphs, are more suitable for information and knowledge representation, because the modeled real-world relations often relate more than two objects.

In the following section we discuss open problems of graph visualization. In Section 3 we present the software visualization field, which is followed by our approach for hypergraph-based representation of software. We illustrate our approach with the visualization of an existing

software system. To lower visual clutter we propose a magic lens that in combination with query mechanism filters and highlight parts of the visualized hypergraph, but preserves contextual information. In the last section we present our current experiments with collaborative graph exploration environment.

Graph visualization

The main problem of graph visualization is the graph's size and density. We can categorize graphs according their number of nodes $|N|$ and number of edges $|E|$ into following groups: **sparse** $|E|<|N|$, **normal** $|N|<|E|<3|N|$, **dense** $|E|>3|N|$. However, more important than density is the size of the graph to visualize. Due to limited display space on current monitors it is difficult to comprehensibly display graphs contain more than 100-thousands nodes and edges, and it is even more complicated for graphs that contain millions of nodes or more nodes than pixels available on monitor. The graph visualization to be usable, it is often needed that information stored in nodes are displayed as well e.g. in form of short labels, thus even more contributing to visual clutter making the graph visualization less comprehensible. For cases where also node content is directly visible in graph visualization, the graph layout algorithm must take into account the spatial dimensions needed to display node's content. For graph visualization several basic aesthetic principles and rules have been identified that produce pleasant and more comprehensible visualizations and are related to (Bennett, C., Ryall, J., Spalteholz, L., Gooch, A., 2007): *positioning of nodes* (balanced node placement – symmetry, not overlapping nodes, related nodes create clusters, nodes are not to close to edges etc.), *edge placement* (minimize edge crossing and bending, equal edge lengths, maximize angles between edges etc.) and the *whole graph layout* (maximize graph global and local symmetry, minimize layout area, adjust layout area to display area etc.). These principles can be combined, but some combinations are contra-productive.

Graph layout algorithms can be categorized into two categories: deterministic and non-deterministic. Deterministic layout algorithms use exact equations to place nodes. Typical examples of this approach are layered/hierarchical views (Herman et al, 2000), Reingold-Tilford's views, cone trees (Robertson, G.G., Mackinlay, J.D., Card, S.K., 1991) and radial views (Herman et al, 2000), tree-maps etc. Non-deterministic graph layout algorithms use a physical model in which nodes are positioned by applying forces and their final position is reached when the whole system reaches minimal energy state. These force-based methods consist of a physical model and a simulation algorithm. The model defines graph nodes as physical objects that react on forces. The simulation algorithm then iteratively applies forces and reassigns node positions until an equilibrium state is reached. Force-based layout methods have many advantages: easily implementable, very parametrical, modifiable by adding new forces, effective for small graphs, produce symmetrical layouts, animation of layout preserves mental map, easily expendable into 3D. Force-based layout methods have, similarly to other methods, also disadvantages: slow for large graphs and final layout is not predictable. The first force-based layout method was developed by Eades (Eades, P., 1984) in which adjacent nodes are attracted by spring forces defined by edges and nodes are repelled when no edge connects them. Various modifications of this approach have been developed (Kaufmann, M., Wagner, D., 2001), each scaling better for larger graphs. Recently implementations of force-based layout algorithms on GPUs allow to layout very large graphs in very short time (Frishman, Y., Tal, A., 2007).

The visualization of common graphs often uses simple graphical elements for nodes and lines and/or arcs for edges. However generalized graphs, e.g. hypergraphs or hierarchical graphs, need more complicated visualizations. In our work we have focused on hypergraphs. A hypergraph is pair $H=(V,\varepsilon)$ where $V=\{v_1,\ldots,v_n\}$ is a finite set and it's members are called nodes, and ε is a family $(E_i)_{i\in I}$ of subsets of V. The members of ε are called hyperedges.

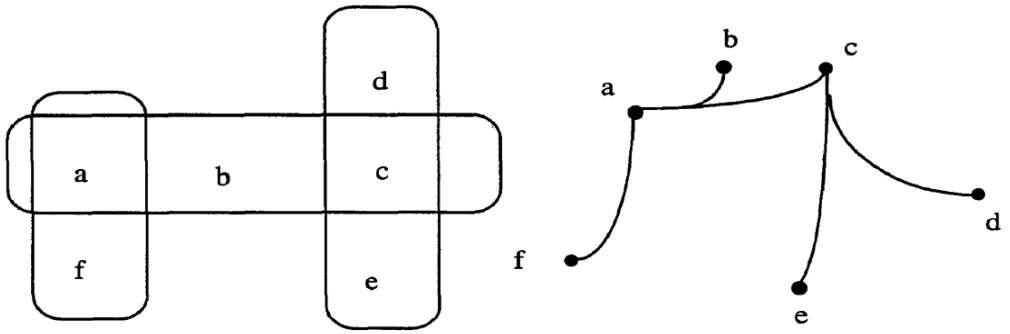


Fig. 1 *Hypergraph drawing: (left) set-like (right) graph-like approaches.*

Figure 1 illustrates two main approaches for hypergraph visualization. The hypergraph $H=(V,\varepsilon)$ contains five nodes $V=\{a,b,c,d\}$ and three hyperedges $\varepsilon=\{\{a,b,c,\},\{a,f\},\{c,d,e\}\}$. Visualizations displaying hyperdeges as lines or arcs are more common than the set-like visualizations. Similarly for common graphs, we can also define a hypergraph incidence matrix. Let $H=(V,\varepsilon)$ be a hypergraph with $m=|\varepsilon|$ edges and $n=|V|$ nodes. The edge-node incidence matrix of H is $M_H \in M_{m\times n}(\{0,1\})$ and defined as: $m_{i,j}=\begin{cases}1 if\ v_j \in E_i\\ 0\ else\end{cases}$. Using this hypergraph incidence matrix we can construct a bipartite incidence graph $B_H=(N_V \cup N,E)$ defined as: $E=\{\{m_i,n_j\}:m_i \in N_\varepsilon,\ n_j \in N_v, m_{i,j}=1\}$ $N_\varepsilon=\{m_i:E_i \in \varepsilon\}$ $N_V=\{n_i:v_j \in V\}$. Using this transformation of a hypergraph into a bipartite incidence graph we can utilize for hypergraph visualization all current well-known layout algorithms for common graphs.

Software visualization

Software is not directly tangible – a well known fact that is very often mentioned to students of software engineering. Also software developers in praxis have to deal with the intangibility of software when developing new products or studying internals and functionality of existing software systems. Software visualization aims at providing means for better access to software by providing visualizations that may provide better insight into software structure and internal software behavior. Software visualization focuses on three main software aspects: software structure, behavior of executing processes, and the evolution of software development (Diehl, S., 2007). Visualization of software structure provides useful insight at various levels of software abstraction: from decomposition to modules and components, trough hierarchy of classes towards the structure at the level of programming language constructs. Visualization of the behavior of executing processes allows to reveal behavior not easily derivable from static source code as program runtime may be significantly more complex depending on the computation and data that are processed. As software evolves during development and also after deployment, it is necessary to store information about software changes. Visualization of software evolution can contribute to the understanding how often individual parts of the software changed, who contributed to development and can be useful for future projects by showing how a previous similar project was managed. Although many interesting software visualization approaches have been published, very few of them find their applications in praxis. For example the UML modeling language is often used for structural modeling of software. Although UML class diagrams can be generated from source code, very few existing tools actually provide useful visualizations and especially interaction functionality that can be useful for program analysis. In an experimental visualization tool the authors (Šperka, M., Kapec, P., 2010) implemented a cone-like tree visualization of a class diagram as show in Figure 2.

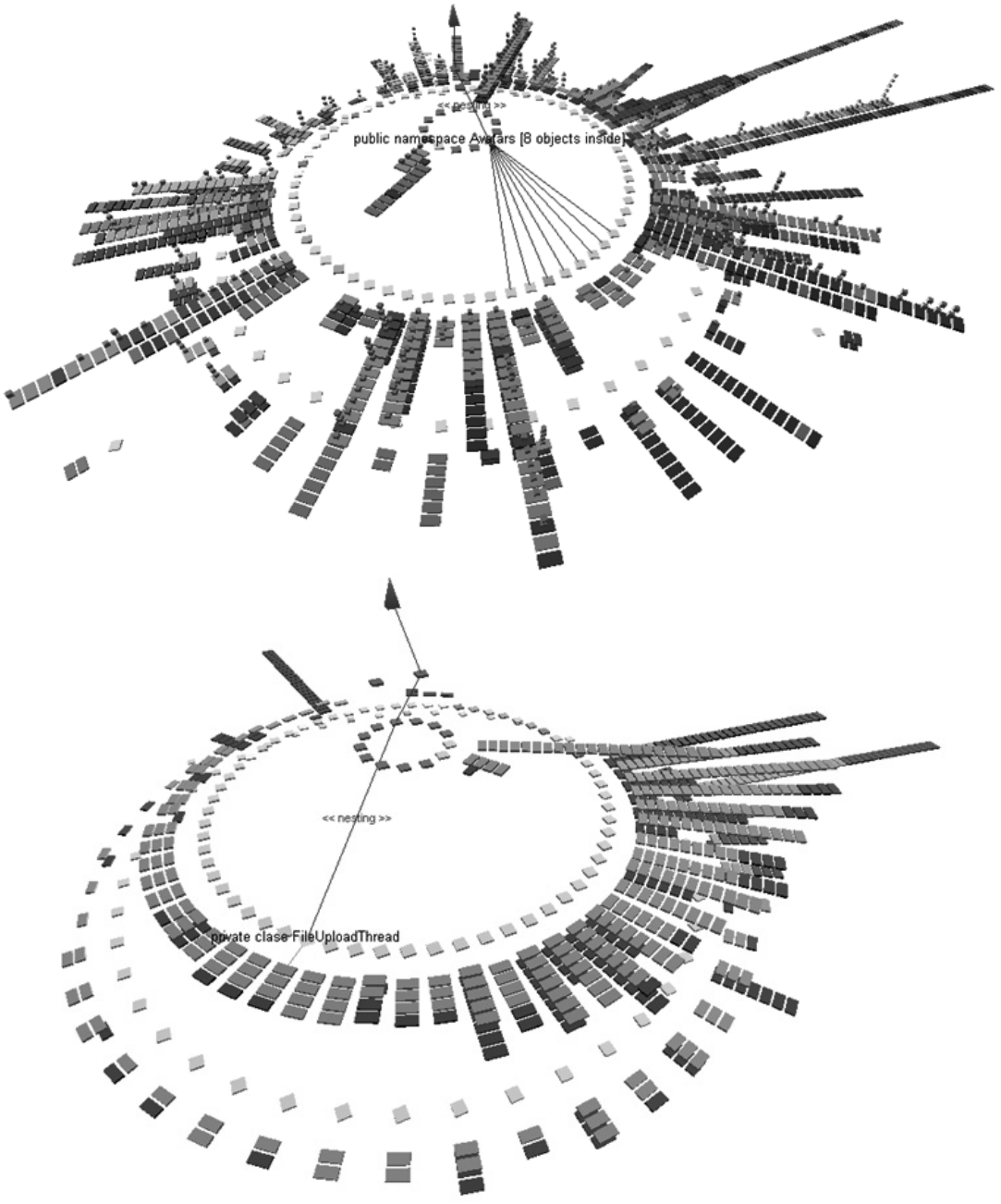


Fig. 2 *Visualization of an existing system (left) Sorted view of classes and their methods (right)*

The left figure shows a class diagram with name-spaces, classes and their methods and attributes. The right figure shows the same class diagram, but sorted and filtered, thus effectively showing which classes are the most complex (based on the number of methods). The user can freely explore the visualization by navigating a virtual camera and use sorting and filtering to focus on class diagram elements of interest. In another interesting project (Šperka, M., Kapec, P. Ruttkay-Nedecký, I., 2010), authors have developed a program runtime visualization system that provides interesting 3D views how programs implemented in the Java programming language work.

Visualizations of program runtime can be more useful than common program debugging in standard debbugers, because they may also show more contextual information.

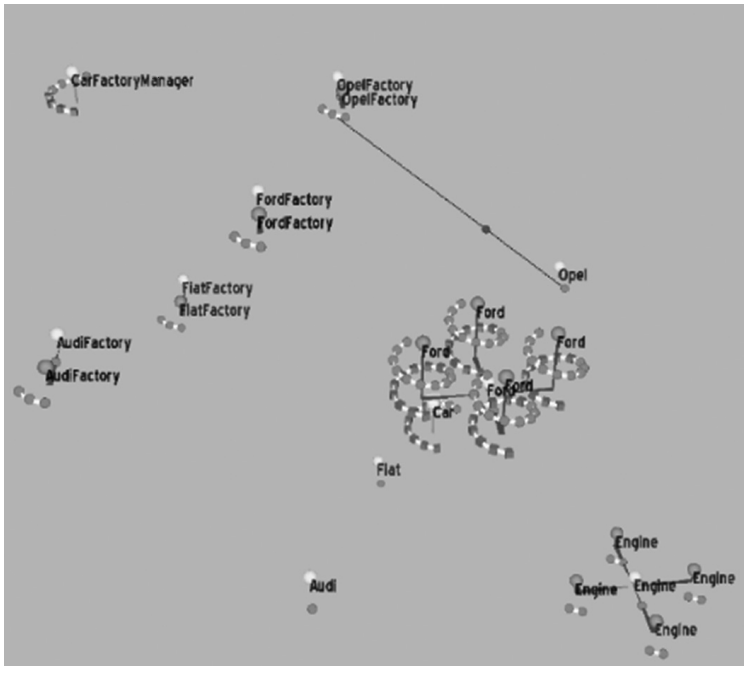


Fig. 3 *Factory method program elements.*

The Figure 3 shows a snapshot of program runtime visualization. The four groups represent (from left to right): a FactoryManager class; a cluster of four factories; products cluster; and 4-classes that are shared between all products. The line between a factory and a product shows message passing between object instances. This visualization system allows to play the program execution forth and back, to jump to a specific time and to examine the state of individual object instances. These features are a major innovation for program debugging and are very helpful for developers.

Graph-based representation of software artifacts

The term software is not only source code, but it also covers all software artifacts that can be found in the software development process and contribute to the final software product. Among various software representations the representation trough graph structures are very useful, because they allow us to look at software as a repository of knowledge that can be queried. Graph node's can be used to represent individual software artifacts from source code level like functions, classes, files, documentations, their revisions etc. to subjects found in software development like developers, tasks, deadlines, bugs etc. Graph edges can be used to represent relations between these software artifacts, e.g. call relations between functions, inheritance relations, or which developer implemented which functionality, which bug he fixed or on which tasks is he working. Although common graphs can be used, several papers propose the use of generalized graphs. Hypergraph representations, as proposed in (Kapec, P., 2010), are more suitable, because they borrow ideas from knowledge representation field. To illustrate the possibilities of hypergraph representation let us consider a function from OpenGL's API*: void glGetShaderSource(GLuint shader, GLsizei bufSize, GLsizei*length, GLchar *source)* that takes four parameters, but the length, source are used as return parameters, and of course the parameters are ordered.

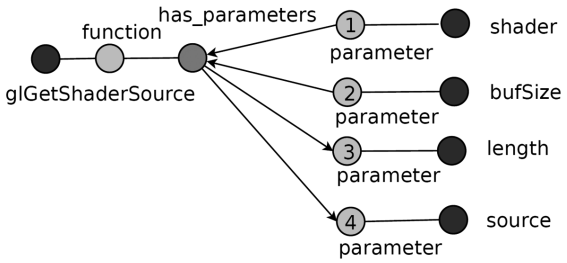
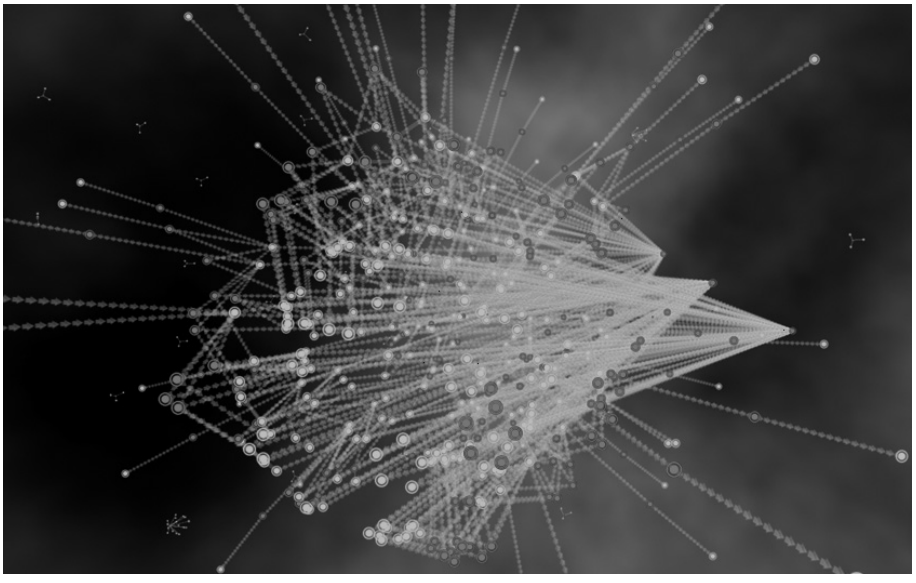


Fig. 4 *A hypergraph representation of a function.*

The hypergraph representation of this function is shown in Figure 4: the relation *has_parameter* is represented as a hyperedge (shown as a red node) and connects the function *glGetShaderSource* with it's parameters *shader, bufSize*, *length* and *source* (shown as a blue nodes). The proposed hypergraph representation also adds hyperedge orientation a ordering (shown as green nodes). Using this hypergraph representation it is also possible to define a query language also based on hypergraphs as proposed in (Kapec, P., 2010). Such queries allow to filter the hypergraph repository of software artifacts.

Visualization of an existing software system

The above discussed hypergraph-based representation was used in a prototype software visualization system and used for the visualization of an existing open-source software system (Kapec, P., 2010). The authors were able to extract more than 1200 nodes and more than 450 hyperedges from the software system by searching only for eight node types and seven different hyperedge types. Searching for other node/hyperedge types and in larger software systems would certainly lead to dramatic increase in extracted artifacts and relation, thus making the visualization more uncomprehending. The Figure 5 shows the visualization nearly of the whole extracted hypergraph (left) and a sub-graph obtained by a query. The query searched the extracted hypergraph for modules and main functions found in these modules. As can be seen, the whole hypergraph visualization is difficult to comprehend; on the other hand the filtered hypergraph clearly shows nine clusters containing modules. From these clusters we can also easily identify the more complex modules (they contain more functions and/or functionality) and distinguish them from simpler modules.



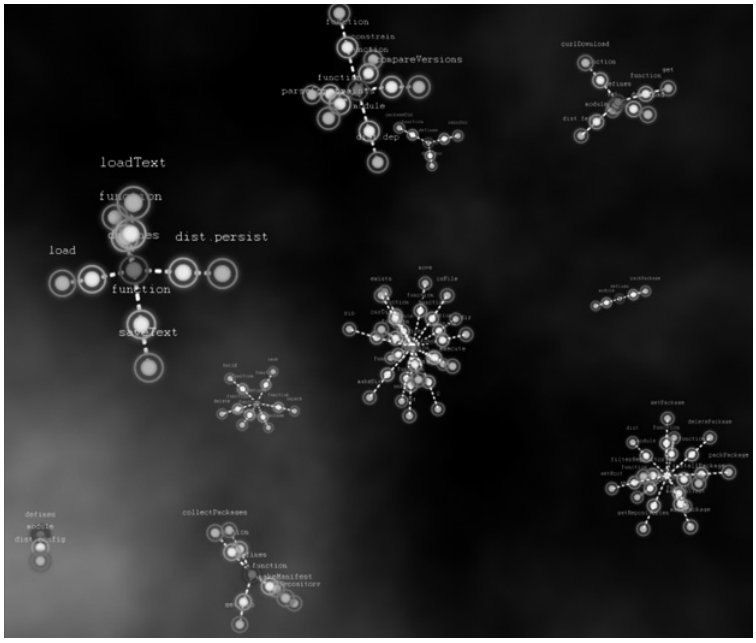


Fig. 5 *Hypergraph visualization of an existing open-source system (left) Results of a query (right).*

The extracted clusters containing modules where obtained by the following simple query: *defines(module:*; function:*)*, and the query can be read as "Find all modules that define functions and extract all these modules and functions". Actually the query also extracts the *defines* hyperedges, thus actually extracting a sub-hypergraph. As presented in (Kapec, P., 2010) the queries can be more complex by joining such simple queries using an *and* operator. The textual queries actually define a hypergraph pattern that is used by a query mechanism to search and match in the queried hypergraph.

Lowering visual clutter using magic lenses

Although the query mechanism described in previous section can filter the large hypergraph, the resulting matched sub-hypergraph may still be very large and difficult to comprehend. Also the query removes the unwanted hypergraph parts, thus the observer looses the global view and contextual information. For these reasons we have developed, based on similar approaches (Bier, E. A., et al, 1994), a magic lens technique that incorporates the discussed query mechanism. A magic lens is displayed in the screen center and is equipped with a user specified hypergraph query. Figure 6 shows a hypergraph visualization from two different viewpoints – both contain a magic lens that filters the hypergraph and shows and highlights only hypergraph parts matched by the query.

The magic lens is semi-transparent, thus slightly showing also the unmatched hypergraph parts. Also the magic lens does not cover the whole display space, thus the user can see the rest of the hypergraph not affected by the magic lens. As can be seen from Figure 6, the original hypergraph is not colored, but the magic lens shows some nodes in color. We have enhanced the query language so that it is possible to set various visual attributes to matched nodes and/or hyperedges.

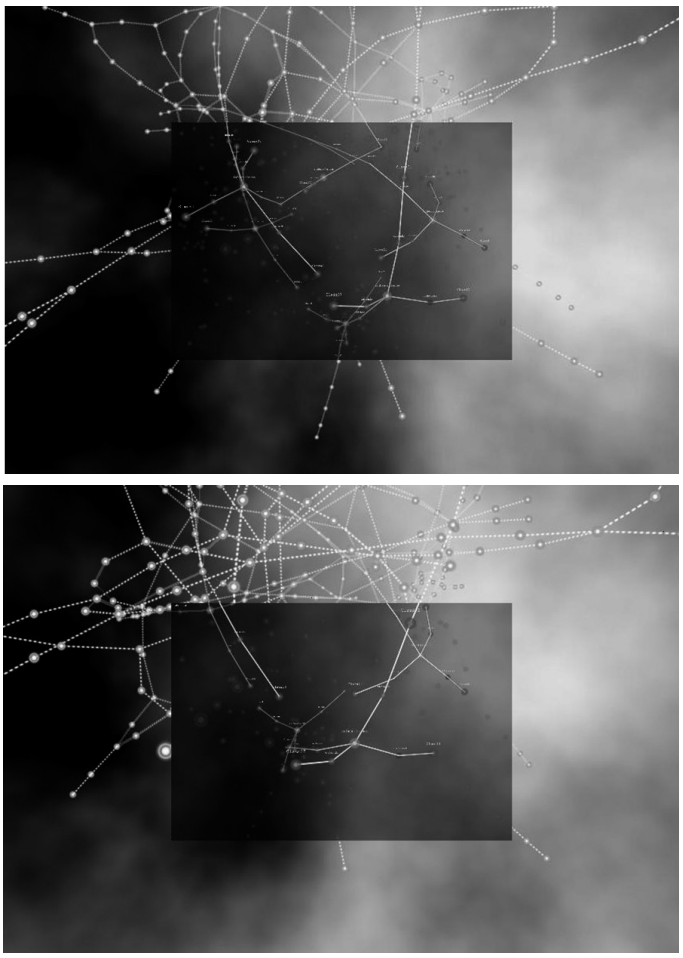


Fig. 6. *A view on a hypergraph from different angles using magic lenses.*

For example to find all subclasses inherited from parent A and show the label of the found subclasses, assign color, size and transparency to them, we can write the following query: *inheritance(*{L = yes; C = [255; 255;0]; S = 3; T = 0.5} : subclass, A : parent)*. The visual attributes modification part of the query is in curly brackets has the form of *key = value* and can be applied either on matched nodes or hyperedges. This way the user can customize the visualization and highlight hypergraph parts of interest.

Towards a collaborative graph exploration environment

We are currently experimenting with collaboration during graph exploration. We have implemented a collaboration extension to the hypergraph visualization system that allows multiple users to explore and navigate the visualized hypergraph over a network. The system is based on the client-server network architecture, in which the server handles the graph layout and sends information about node positions to clients. All clients see the visualized hypergraph from their own perspective and the users can individually explore the hypergraph. Figure 7 shows two clients exploring the same hypergraph from different viewpoints. To increase cooperation, we included into the visualization also graphical avatars in the form of oriented cones that represent users in the virtual scene, thus allowing to observe the position of other users and their behavior. From the implementation view, collaborative viewing of graphs is complicated by the layout algorithm, because the layout algorithm is often very computation

intensive and it is necessary to synchronize the layout algorithm with the rendering loop and with sending layout updates over network to clients. The client-server architecture may be the simplest; however moving to peer-to-peer architecture may open several interesting aspects. In peer-to-peer architecture we could distribute the layout algorithm to all peers and thus utilize their computation power.

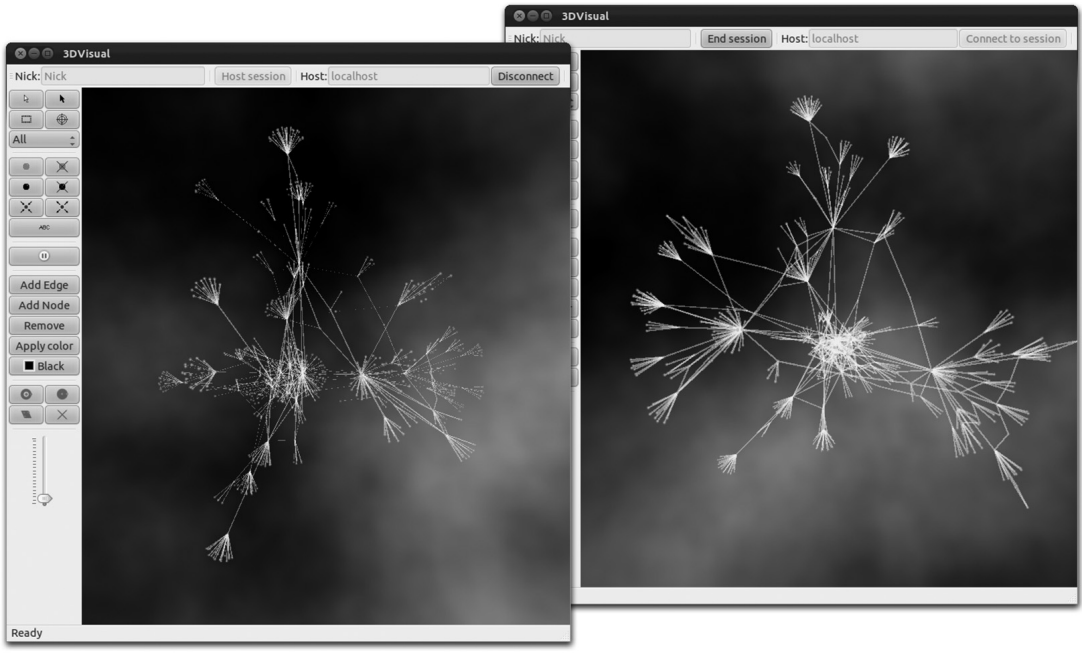


Fig. 7. *Two clients exploring the same hypergraph from different viewpoints.*

Conclusions

In this paper we shortly introduced the problematic of graph and software visualization. We presented interesting hypergraph-based representations of software artifacts and showed the visualization of an existing software system. Our main focus was aimed at lowering visual clutter using magic lenses. Currently the magic lens is static on screen, but we are working on multiple overlapping and user movable lenses. We shortly discussed a prototype collaboration environment in which we are experimenting with interaction methods for collaborative graph exploration.

Acknowledgments

We would like to thank Zuzana Číková and Ivan Pleško for their help with the development of magic lenses and the prototype of collaborative graph exploration.

References

- Bier, E. A., et al (1994). Toolglass and magic lenses: the see-through interface. In CHI '94: Conference companion on Human factors in computing systems, ACM, pp. 445–446
- Bennett, C., Ryall, J., Spalteholz, L., Gooch, A. (2007). The aesthetics of graph visualization. Computational Aesthetics in Graphics, Visualization, and Imaging
- Diehl, S. (2007). Software visualization: visualizing the structure, behaviour, and evolution of software. Springer Verlag

- Eades, P. (1984). A heuristic for graph drawing. Congressus numerantium, 42 (149160), pp. 194-202
- Frishman, Y., Tal, A. (2007). Multi-level graph layout on the GPU. IEEE Transactions on Visualization and Computer Graphics, 13(6), pp. 1310-1319
- Robertson, G.G., Mackinlay, J.D., Card, S.K. (1991). Cone trees: animated 3D visualizations of hierarchical information. In Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology, page 194, ACM
- Herman et al (2000). Graph visualization and navigation in information visualization: a survey. In IEEE Transactions on visualization and computer graphics, volume 6
- Kaufmann, M., Wagner, D. (2001). Drawing graphs: methods and models. Springer Verlag
- Kapec, P. (2010). Visualizing software artifacts using hypergraphs , in proceedings of SCCG'2010 Spring Conference on Computer Graphics in Cooperation with ACM and Eurographics, pp. 33-38
- Šperka, M., Kapec, P. Ruttkay-Nedecký, I. (2010). Exploring and understanding software behaviour using interactive 3d visualization, in proceedings of ICETA, 8th Int. Conference on Emerging eLearning Technologies and Applications, pp. 281-287
- Šperka, M., Kapec, P. (2010). Interactive Visualization of Abstract Data. In: Science & Military.- Vol. 5, No. 1 (2010), pp. 84-90

Ing. Peter Kapec, PhD.
Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 3, 842 16 Bratislava, Slovak Republic
Email address: kapec@fiit.stuba.sk

About Author(s) Peter Kapec received his PhD. in 2011 at the Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, in the field of Applied Informatics. He is currently a research fellow at same faculty and his research is oriented at graph, knowledge and software visualization.