Exploring 3D GPU-accelerated graph visualization with time-traveling virtual camera

Peter Kapec, Michal Paprčka, Adam Pažitnaj, Vladimír Polák
Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Slovakia
kapec@fiit.stuba.sk, michal.paprcka@ynet.sk, librus44@gmail.com, xpolakv@is.stuba.sk

Abstract: Graph visualization is an ongoing research area with many open problems. Graphs are often visualized in 2D space and recently also 3D visualizations emerge. However, the added third dimension adds additional problems that make the graph comprehension more difficult. In this paper we focus on navigating and exploring 3D graph visualizations. We present our approach for the automation of virtual camera movement for better graph exploration. This camera movement is enhanced with experimental exploration recording and play-back that allows to fork exploration paths at any time and to switch between them. We also present how graph layout can be accelerated with GPU in combination with scene graph structures. These features were added into our graph visualization system that we use for software visualization. We present several visualizations of the structure and the evolution of software systems.

Keywords: graph visualization, software visualization, exploration, playback, recording, GPU, scene graph

1. Introduction

Graph visualizations have recently been in focus of many research projects, because various information can be effectively modeled by graphs. Graph visualizations aim to help with graph comprehension by providing graphical views that may reveal hidden structures and other interesting topological features. The problematics of graph visualizations addresses graph complexity (graph size and density) and graph layout algorithms often utilize the well know force-based approaches. Still the aesthetic criteria and comprehension of the visualization are open problems. Exploring graphs involves examining graph nodes and their content. However, the relations between nodes that are represented trough edges are often more important. Navigation between nodes can be done either freely or by inspecting edges from a starting node and following edges of interest. Often such edge following leads to other interesting nodes and this process is repeated. During the exploration process the users may ask what information could we obtain if we explored the graph in another way.

In this paper we present our approach for 3D graph exploration that uses a virtual camera that calculates its motion path so that all nodes the user is interested in are always visible, thus preserving contextual information. We enhanced this approach by providing exploration history that can be played back. The user can fork other exploration paths from the stored exploration history. These new approaches were integrated into our generic 3D graph visualization system that we use for software visualization, as various software aspect can be modeled with graphs – several examples are show in later sections.

The next section mentions related work that inspired us and we briefly mention the main
differences of our work. The study presented here is an extension of our previous work presented in [1]. Therefore, in Sections 3 and 4 we briefly repeat parts of the previous material. Section 3 presents our algorithm for intelligent camera movement with illustrations and evaluation for several kinds of graphs. The explanations are extended for better clarity, including the addition of new figures. The Section 4 describes the time-traveling features of our graph visualization system. The new contributions made here are described in Sections 5 and 6. In Section 5 we discuss the implementation details of the used graph layout algorithm based on scene graph representations, which is accelerated with GPU. The following Section 6 demonstrates how we utilized graph visualization to visualize the structure and development history of existing software projects, which is followed by conclusions.

2. Related work

Graph visualization is a large problem area [2]. We have used the force-directed graph layout proposed in [3] with modifications proposed in [4]. The presented algorithm for camera movement is related to finding the best viewpoints problematics [5, 6, 7]. A similar approach to ours has been proposed in [8]. However, our approach considers also additional points of interest that may not be visible from initial camera viewpoint as opposed to [8], which tries to maximally preserve the initial visible area. In addition, we used several graph metrics mentioned in [9] to select graph’s nodes of interest, which are used as control parameters for the proposed camera movement algorithm.

Many research projects focus on time-series data visualization [10], or displaying and managing histories of actions done during data visualization [11]. Several time-line interfaces for visualizing undo, browsing and time travel functionality have been discussed in [12]. However our method for 3D graph exploration with time-traveling features can be considered as a new and unique approach, especially in the graph visualization field. Our approach not only records user actions and exploration, but records also the graph layout process.

A good overview of the software visualization research area can be found in [13, 14, 15]. Our software structure visualization approach is based on hypergraph-based software representations developed in our previous work [16]. Our approach for software evolution visualization was inspired by existing projects [17, 18], but offers different ways of visualization and exploration.

3. Automated camera movement preserving best view

Graphs can be explored in various ways, but the simplest approach for exploring 3D graph visualizations is to move the virtual camera along a straight line between camera’s current position and camera’s target position. The camera’s target position should be near the user selected graph node. This simple approach has however several open problems: a) what is a suitable camera position and orientation near the selected node? (actually what the camera will be actually looking at) b) where should be the camera facing during movement? (except the target node there might be other interesting graph parts).

These questions are fundamental to the user, because when the movement and final position of the camera is not clear, the user may lose orientation and become confused. To preserve user’s mental model it is suitable that the camera’s movement is smooth, without abrupt changes, and that the virtual camera at the final position shows also the starting position. During camera movement the user is often interested not only in the node’s final position,
but also in several other graph nodes. Therefore it is also suitable that the camera shows also graph nodes of user interest during movement.

![Diagram](image)

(a) Camera movement between two positions (b) Camera’s position $C_T$ and focus point $C_F$

Figure 1. Illustration of the camera movement problematic

### 3.1. Camera path algorithm

We address the above mentioned situation and our approach for camera movement is illustrated in Figure 1a: the virtual camera starts at position at time $T_1 = 0$ and is oriented towards some graph node. The user then selects another graph node and the camera starts moving towards the target node, which is reached at time $T_3 = 1$. However, the user is interested in several nodes from the graph, highlighted as red nodes (their selection can be done in various ways), and these nodes should be visible during camera movement (shown at $T_2 = 0.5$ and also from camera’s final position.\(^1\) The view from camera’s final position should also cover camera’s starting position, so the user can easily identify the position where he/she came from.

The points of interest our algorithm takes into account are therefore: the initial starting node, the target node and a cluster of relevant nodes\(^2\). These points form the control points of a Bezier curve that will be used for camera movement.

Our camera model is not only defined by position and orientation, but also has an additional parameter: the position of a focus point that lies along camera’s view axis. Similarly to the control points for camera movement, we define the control points for the camera’s focus point. This way we can independently manipulate camera’s position and orientation and also change the point in space the camera is facing to. Figure 1b illustrates our camera model: $C_{P_1}$ and $C_{P_2}$ are the start and end camera positions, the $C_{T_1}$ and $C_{T_2}$ are the start and end positions of camera’s focus point. Camera’s position and focus point can move along independent paths, as shown in Figure 1b.

The Figure 2 shows two blue curves – the calculated camera trajectory for two different graphs. The figure also shows two red curves – the calculated trajectory where the camera should be looking. As can be seen, these two trajectories are different.

---

1. Time $T_2$ is only illustrative, the nodes of interest should be in camera’s field-of-view most of the time.
2. We use the geometric center of these nodes and apply it with a negative weight.
The eight images in Figure 3 show the first graph in Figure 2 from camera’s viewpoint during camera movement. The camera starts with the red node in focus and then the user selected the green node in the left lower part of the image. The following images show how the camera moved along the calculated trajectory to the selected green node. As can be seen in the final image, the selected green node is in camera center and the starting red node is also visible.

The Algorithm 1 summarizes our approach. The algorithm takes as input:

- **CameraPositions** – camera’s position
- **CameraTarget** – camera’s focus point
- **TargetCameraPos** – camera’s target position near the selected node
- **NodesOfInterest** – the list of nodes of interest
- **Speed** – speed of interpolation

The algorithm calculates the center of the nodes of interest and sets control points for the Bezier curves along which the camera will move (CameraPositions, CameraTargets). Then we initialize weights for the Bezier curves and calculate the camera’s forward vector, which is used to move the camera backward until all nodes of interest are in camera’s field-of-view. Similarly we increase the curvatures of camera’s trajectory.

Then we calculate the individual positions on both trajectories (the Bezier curves) using the mentioned control points and set the new camera position and orientation (using NewPos, and NewTarget). The camera’s up vector does not change during camera motion to minimize confusion.

### 3.2. Evaluating camera movement

We have tested our approach on two graphs called *veolia* (containing 639 nodes and 917 edges) and *genealogy* (containing 515 nodes and 1072 edges). For selecting nodes of user’s interest we used two approaches. The first was to let the user manually select nodes by mouse and the second was based on graph metrics. We used following four graph metrics [9]: 1.) matching node’s content 2.) the degree of nodes 3.) the summation of length of shortest paths 4.) distance between nodes. These graph metrics, together with user selected nodes, can be combined using weights to form various complex node selections.

To verify that our algorithm displays all nodes of interest, we have measured the number of nodes visible on screen during camera movement. The Figure 4 shows the number of visible nodes during camera movement for movement interpolation steps.

The total number of nodes of interest was up to 50. The Figure 4 shows that the algorithm was able to display all nodes of interest. These three runs for different starting and ending camera positions also reveal, that the time all nodes of interest are visible, vary – especially at the beginning and at the end of camera movement not all nodes of interest are visible. Therefore it is important to evaluate not only if all nodes of interest are visible but also the amount of time they are visible. The Table 1 shows the results for the mentioned two graphs for various metrics and for a selected number of nodes of interest. The number of camera movement interpolation steps was set to 2000. The first column shows the total number of camera movement steps in which the nodes of interest were visible.

As can be seen the percentage ranges from 48.8% to 99% of time the nodes were visible. The best results were obtained for user selected nodes and worse for metrics-based selections. The explanation of these results can be seen in the fact, that the user usually selects nodes by
Figure 2. The blue and red curves display the calculated trajectories for two different graphs.

**Require:** \texttt{CameraPosition}, \texttt{CameraTarget}, \texttt{TargetCameraPos}, \texttt{NodesOfInterest}, \texttt{Speed}

1. \( t_1 \leftarrow 0, t_2 \leftarrow 0 \) \{comment: interpolation steps\}
2. \( \texttt{PointOfInterest} \leftarrow \texttt{GetCenter} (\texttt{NodesOfInterest}) \)
3. \( \texttt{center} \leftarrow \texttt{GetCenter} (\texttt{EyePosition}, \texttt{TargetCameraPos}, \texttt{PointOfInterest}) \)
4. \( \texttt{CameraPositions} \leftarrow \{\texttt{CameraPosition}, \texttt{PointOfInterest}, \texttt{TargetCameraPos}\} \)
5. \( \texttt{CameraTargets} \leftarrow \{\texttt{CameraTarget}, \texttt{PointOfInterest}, \texttt{center}\} \)
6. \( w_1 \leftarrow \{1, -0.1, 1\} \) \{comment: weights for camera’s positions\}
7. \( w_2 \leftarrow \{1, 0.5, 1\} \) \{comment: weights for camera’s target positions\}
8. \( \vec{a} \leftarrow \texttt{TargetCameraPos} - \texttt{center} \)
9. \( \textbf{while} \ NOT \texttt{IsVisible}(\texttt{CameraPosition}) \ AND \ NOT \texttt{IsVisible}(\texttt{PointOfInterest}) \textbf{do} \)
10. \( \textbf{end while} \)
11. \( \textbf{while} NOT \texttt{IsVisible}(\texttt{PointsOfInterest}) \textbf{do} \)
12. \( \textbf{end while} \)
13. \texttt{Increase trajectory curvature} \texttt{Algorithm 1: Algorithm to calculate camera’s motion path}
Figure 3. These eight images (from left to right and top to bottom) show the camera view during movement for the first graph in Figure 2.
Figure 4. The number of visible nodes depending on time

Table 1. Number of interpolation steps in which all nodes of interest were visible. The total number of interpolation steps was 2000.

<table>
<thead>
<tr>
<th>Steps with visible nodes of interest</th>
<th>Percentage</th>
<th>Nodes</th>
<th>Nodes of interest</th>
<th>Metrics</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>1467</td>
<td>73.35 %</td>
<td>515</td>
<td>50</td>
<td>combined</td>
<td>Genealogy</td>
</tr>
<tr>
<td>1352</td>
<td>67.6 %</td>
<td>639</td>
<td>50</td>
<td>combined</td>
<td>Veolia</td>
</tr>
<tr>
<td>1663</td>
<td>83.15 %</td>
<td>515</td>
<td>20</td>
<td>combined</td>
<td>Genealogy</td>
</tr>
<tr>
<td>1634</td>
<td>81.7 %</td>
<td>639</td>
<td>20</td>
<td>combined</td>
<td>Veolia</td>
</tr>
<tr>
<td>1516</td>
<td>75.8 %</td>
<td>515</td>
<td>50</td>
<td>node degree</td>
<td>Genealogy</td>
</tr>
<tr>
<td>976</td>
<td>48.8 %</td>
<td>639</td>
<td>50</td>
<td>node degree</td>
<td>Veolia</td>
</tr>
<tr>
<td>1973</td>
<td>98.65 %</td>
<td>515</td>
<td>40</td>
<td>user selected nodes</td>
<td>Genealogy</td>
</tr>
<tr>
<td>1980</td>
<td>99 %</td>
<td>639</td>
<td>30</td>
<td>user selected nodes</td>
<td>Veolia</td>
</tr>
</tbody>
</table>

mouse that are placed very near to each other and the nodes picked by metrics-based selections are usually spread over the whole graph.

4. Time-traveling graph exploration

It often happens that the user is interrupted at work and can forget where the work stopped. A possible solution to this problem is to provide a feature that allows to record the work in the form of a recording and then to replay it. The exploration of graph structures often involves user’s navigation decisions depending on user’s current interests. However, sometimes the user would like to return to previous graph nodes and start exploring the graph in new ways. Exploring the graph in new ways may lead to different understandings of different graph parts. It would be beneficial if the user could explore the graph in some way, then return back
to some point in his exploration history, and to start exploration again changing the previous exploration history.

In our graph visualization system we provide even more complex recording of user’s exploration allowing sci-fi like time-travel exploration. The user explores the graph by navigating the virtual camera in the scene containing the visualized graph and examines the content of nodes. Graph exploration may also involve manually changing positions of selected graph nodes and/or node highlighting. This exploration is recorded and the user can return to previous camera positions, either playing back the exploration history or by jumping to desired time. After returning to the desired time position, the user can start recording his exploration again.

The most important feature is however that the new exploration recording creates actually a new time-fork from the previous recording, thus preserving the previous recording. This way the user can create a tree of graph exploration histories. Several interesting questions arise: how to effectively manipulate this tree of graph exploration histories and how to present these recorded histories. For manipulating and playback of the tree containing recordings of graph explorations we have developed a custom user interface widget.

The Figure 5 shows this widget with an exploration tree containing four recordings for four exploration forks. These four exploration forks are controlled by four individual sliders. Below these four exploration forks is a slider covering the whole exploration time. To visually compare different graph exploration recordings, our graph visualization system can open a separate window for each graph exploration fork.

The Figure 6 shows this situation for four selected graph exploration forks – each window is highlighted with a different frame color. Using sliders the user can playback or jump to specific time for each exploration recording individually and the playback is displayed its own window. The bottom slider controls all exploration recordings and allows to play them back simultaneously – this allows the user to visually compare these exploration recordings which may be interesting for future evaluation of different graph exploration methods.

Currently the exploration recordings are stored in files on disk in XML format. However, the amount of stored information (especially the progress of graph layout) is vast – this is a major issue especially for larger graphs and needs to be addressed in the future, e.g. utilizing binary formats and/or storing only important layout events.

5. Accelerating graph layout using GPU

Currently many approaches and algorithms for graph layout exist that produce pleasing layouts. Many approaches are based on the force directed approach [3], but their inherent problem is theirs high computational complexity that is often $O(V^2 + E)$ for each iteration. Generating layouts for large graphs using these approaches can take tens of seconds and for very large graphs even minutes.
General Purpose Graphics Processing Unit (GPGPU) techniques present a suitable way how to deal with this problem. However, common GPU graph layout implementations use simple graphical primitives for visualization, e.g. points or point-sprites for graph nodes and lines for edges, and are not capable to visualize nodes as complex, probably 3D, objects – this might be useful when the nodes contain complex information that we want to visualize [4].

5.1. Integrating GPU-based graph layout into scene graph

Our own developed visualization system utilizes a scene graph to manage and to draw graph visualizations in 3D space. Integrating GPGPU functionality into this kind of data structure brings new problems and challenges to overcome. The work presented in [19] deals with this problem in a very effective way that preserves the scene graph’s model of abstraction and usability and also respects its extension mechanisms.

The aforementioned work integrates GPU compute API into a scene graph by introducing computation nodes that can be inserted at any place within the scene graph. These computation nodes work as container classes for application specific modules and resources. The modules wrap user defined parallel programs (kernels) to be executed on the GPU and resources that represent data the kernels operate on. The Figure 7a shows our scene graph structure with a Computation Node. The attached Layout Module wraps the parallel version of the layout algorithm.

The parallel layout algorithm needs several inputs to work on. The data required by these inputs are located in particular nodes of the scene subgraph and need to be collected first. Our solution utilizes a special class extended from a Resource Visitor class presented in [19]. The Resource Visitor traverses the scene graph in two steps. In the first traversal it collects all required data (vertex positions of nodes and neighborhood information) and in the second
Exploring 3D GPU-accelerated graph visualization with time-traveling virtual camera

traversal it distributes these data to the respective computation nodes (the Computation Node with the Layout Module in our case).

The Figure 7a illustrates how the Resource Visitor traverses our scene graph. Our parallel layout algorithm is based on a modified version of Fruchterman–Reingold layout algorithm [3]. The main difference between the modified and original version is in the calculation of node’s new positions. The modified version takes also into account the forces applied to nodes in previous iterations (node velocity). The parallel algorithm is implemented using the CUDA programming model.

To fully leverage the potential of GPUs, the implementation of the graph layout algorithm needs to be adapted to GPU’s architecture/programming model. To calculate graph layout the algorithm uses following data structures: location data, neighborhood data, velocities. The Figure 7b shows these structures as rectangles. Data that are accessed only for read during kernel execution are bound to textures for faster access and thus better performance.

The algorithm consists of 3 kernels (in Figure 7b shown as circles) that are executed in sequence:

1. **repulse kernel** – calculates repulsive forces exerted on each node. It reads information from the location data structure and this information is copied into shared memory for faster access.
2. **attract kernel** – calculates attractive forces exerted on each node and adds them to repulsive forces. It reads information from the location and neighborhood data structures. It processes each graph edge in parallel and uses atomic add operations.
3. **apply kernel** – calculates new node positions based on forces exerted on nodes in the current iteration and velocities, that represents forces applied to the nodes in previous iterations. New values are updated into the location and velocity data structures. This algorithm is executed during the scene graph update traversal and its results (new nodes positions) are updated into corresponding scene graph nodes.

![Figure 7. Implementation details of GPU-based graph layout algorithm](image)
5.2. Results for test graphs

We have tested our GPU version of the layout algorithm on several graphs. The graphs veolia\(^3\), genealogy\(^4\), wc\(^5\) represent real-world data, the others are synthetic cubes generated just for this purpose (cube\(x\) is a cube with \(x\) nodes on its edge). The synthetic cube-like graphs were used, because it is easy to visually verify the working of the graph layout algorithm – the final graph layout should have a cube-like shape. Figure 8 shows the final layouts generated for the tested graphs. For performance test we have used a computer with an Intel Core i7 2630QM processor with four 2GHz cores. The NVidia GeForce GT 550M GPU has been used. The CPU implementation of graph layout algorithm is single-threaded. Using the CUDA compute API, all operations work with float data type. The Table 2 compares the CPU and GPU versions of the same layout algorithm using several graphs. The table shows the number of nodes and edges, the time for the algorithm to reach final graph layout, and CPU usage during layout computation.

![Figure 8. Graph layouts calculated with GPU](image)

Table 2. Comparison of CPU and GPU versions of the layout algorithm for different graphs

| Graph   | \(|V|\) | \(|E|\) | Time | CPU usage | Time | CPU usage |
|---------|--------|--------|------|-----------|------|-----------|
| veolia  | 725    | 1003   | 19s  | 21%       | 16s  | 9%        |
| genealogy | 521 | 1078 | 32s  | 21%       | 56s  | 9%        |
| wc      | 1539   | 2920   | 82s  | 23%       | 43s  | 12%       |
| cube\_8 | 512    | 1344   | 16s  | 21%       | 30s  | 9%        |
| cube\_12 | 1728 | 4752 | 149s | 23%       | 145s | 13%       |
| cube\_16 | 4096 | 11520 | > 600s | 24% | 219s | 13% |

The results show that the GPU version decreases CPU usage as was intended. Concerning the time to reach the final layout, the CPU version achieves better results with smaller graphs, but as the amount of nodes and edges increases, the GPU version reaches the final layout faster.

---


\(^4\) Genealogy of Influence, available at http://www.mike-love.net/genealogy

This can be due to the fact, that not all principles and techniques of the layout algorithm can be directly ported to GPU, because of GPUs special architecture and programming constraints.

As can be seen from the Table 2, the graph layout time even on GPU is in tens of seconds. Current state-of-the-art graph layout algorithms are capable to layout significantly larger graphs in a similar amount of time [20], however as mentioned above, they do not operate on a scene graph, thus limiting graph visualization to simple graphical elements. We are currently addressing this issue and search for better implementation. However, the graph layout algorithm is effective for smaller graphs that were used to visualize the structure and evolution of software systems at an interactive frame-per-second level as discussed in the following section.

6. Applications in software visualization

We have utilized the work mentioned in previous sections in our general purpose graph visualization system to visualize the structure and evolution of software. Our visualization system supports not only common simple graphs, but can be used on various graph types and graph structures, including generalized graphs like hypergraphs and hierarchical graphs. Especially generalized graph structures have been used in our previous work for software visualization [16]. The following section briefly mentions our previous results.

6.1. Visualizing software structure

We have used the visualization system to visualize the structure of a small open-source project that deals with the management, automatic building and deployment of modules and general libraries6. We used hypergraph structures to represent the software artifacts and their relations.

We focused on obtaining not directly visible relations and artifacts: we were searching for just eight different node types and seven hyperedge types. The extraction resulted in 1233 nodes and 459 hyperedges, what are for such a small project relatively high numbers, considering the small amount of different node and hyperedge types. Searching for other node or hyperedge types would certainly dramatically increase the number of extracted artifacts and relations. For large-scale projects we can expect very high numbers of artifacts and relations.

The Figure 9a shows nearly the whole hypergraph – as can be seen, the visualization is very complex and difficult to visually analyze. Therefore we have developed a filtering mechanism based on a query language that can be used to filter the visualization and show only hypergraph parts that are of user’s interest. The Figure 9b shows all the modules and their interfaces obtained by a simple textual query that is based on a hypergraph query language – more details and results can be found in [16].

6.2. Visualizing software evolution

In addition to software structure visualization we also focused on software evolution visualization. Analyzing changes in software made by developers during the whole development process is a tedious work. Visualizing software changes using animation may provide a better overview of the development progress – rapid changes in software structure, e.g. adding new source code files, can be easily visually detected.

6 The LuaDist project, available at: www.luadist.org
Figure 9. Visualization of the structure of a software system [16]

Figure 10. Sequence of the evolution of source code file hierarchy
We have utilized our graph visualization system to visualize various Git repositories storing versions of source code files. For each commit we extracted the folder hierarchy with individual files and identified changes (e.g., addition/removal of files/folders). The folder hierarchy was then visualized. Figure 10 shows a part of the animated visualization.

The first image shows the file hierarchy before a commit. The second and third image show the user connected to all files that the commit affected. The fourth image shows the situation after the commit as the graph layout approaches final layout.

Our approach for visualizing the evolution of source code files and their hierarchical organization is very similar to the Codeswarm [18] and Gource [17] visualization systems, but our approach uses a 3D graph layout and the user can utilize the automated camera movement mentioned in previous sections. Our visualization steps over individual commits as opposed to continuous time playback in the mentioned systems.

7. Conclusions

In this paper we have presented our approach for virtual camera movement that considers user’s points of interest when moving between two positions. The evaluation of the proposed algorithm for camera movement shows that it is capable to show points of interest most of the time the camera moves. We enriched this virtual camera for graph exploration with time-travel functionality, which allows to play back exploration time-lines and start new graph exploration paths. We also discussed our approach for GPU-based graph layout that operates on a scene graph. We briefly presented two examples of software visualizations based on graph visualizations that can directly benefit from the proposed method for camera motion and exploration recording and playback.

Future work can be directed to several aspects of the presented work. Although the camera path algorithm produces good results, there is still room to improve the number of visible nodes, especially in the beginning and at the end of camera movement. User evaluation of the time-travel functionality needs to be addressed, especially in concrete situations to verify benefits of this approach. The implemented graph layout algorithm and the measured results show that a graph layout algorithm operating over a scene graph is complicated and we need to find a better solution to compete with state-of-the-art layout algorithms.

Acknowledgment. This work was supported by the grant KEGA 068UK-4/2011: Integration of visual information studies and creation of comprehensive multimedia study materials. We would like to thank Lenka Baková for her work on the software evolution visualization part.

References


