Harnessing WebGL and WebSockets for a Web-Based Collaborative Graph Exploration Tool

Björn Zimmer(✉) and Andreas Kerren

Department of Computer Science, ISOVIS Group, Linnaeus University,
Vejdes Plats 7, SE-35195 Växjö, Sweden
{bjorn.zimmer,andreas.kerren}@lnu.se

Abstract. The advancements of web technologies in recent years made it possible to switch from traditional desktop software to online solutions. Today, people naturally use web applications to work together on documents, spreadsheets, or blogs in real time. Also interactive data visualizations are more and more shared in the web. They are thus easily accessible, and it is possible to collaboratively discuss and explore complex data sets. A still open problem in collaborative information visualization is the online exploration of node-link diagrams of graphs (or networks) in fields such as social sciences or systems biology. In this paper, we address challenges related to this research problem and present a client/server-based visualization system for the collaborative exploration of graphs. Our approach uses WebGL to render large graphs in a web application and provides tools to coordinate the analysis process of multiple users in synchronous as well as asynchronous sessions.

Keywords: Collaboration · Web user interfaces · WebGL · WebSockets · Network visualization · Graph drawing

1 Introduction

The advent of the Web 2.0 introduced a vast amount of interactive web applications such as text editors, online drawing tools or more advanced office suites (e.g., Google Docs). The online nature of these applications makes them feasible for collaboration, and an increasing amount of users spread across the globe work together to synchronously create and edit documents in web-based tools. These collaborative applications should assist users by providing insights about other users who are working on the same document to help everyone establishing a "common ground" (cf. Section 2 and [1,2]) in the shared workspace. Users should be able to notice if another user joins or leaves a session and be aware of the changes that other users are applying to a document. In addition, collaborative systems should also utilize concurrency control to solve arising conflicts when different users want to apply changes to the same part of a document simultaneously.

Information visualization researchers also use the web for making collaborative data visualizations available. The biggest advantage of these web-based applications is their convenient usage. Users do not have to download and install additional software packages or plug-ins to collaborate. It suffices to share a URL to start a collaborative session with other colleagues. This tackles a problem which was mentioned by Isenberg et al. [3] as one of the ongoing challenges in collaborative information visualization.

In this work, we focus on the cooperative analysis of complex node-link diagrams. These diagrams are usually explored by various domain experts, who would like to work together to improve the quality of the analysis process. This process can take place in a joint online session where everybody works *simultaneously* on one data set and discusses possible changes and insights in real-time with other users. In this case, an expert might want to see what the others are doing and if there are possibilities to coordinate the efforts and find a common ground. In another scenario, experts would like to work on the data set whenever time permits. In such an *asynchronous* setting, it would be interesting to quickly perceive changes that were performed by former analysts or to find out which parts of the data set were already explored by others.

We designed our visualization tool OnGraX to address special challenges for *synchronous* and *asynchronous* collaborative network analysis in a web-based environment. The graphs which are analyzed with the help of our system usually have a complex topology and various node attributes. But the available libraries for the visualization of graphs in web browsers are not sufficient enough, since their ability to render several thousand labeled nodes with different shapes and colors is quite limited. In this paper, we want to discuss the technical design decisions of our tool which enable us to provide an easy accessible web-based tool for the collaborative exploration of graphs. Note that all visualization details were published separately [4,5]. Here, we concentrate on the technical and engineering aspects of OnGraX. We will discuss the following points in detail:

– the utilization of WebGL to render graphs with several thousand labeled nodes of different shapes and color,
– the usage of WebSockets to provide real-time mouse and viewport positions of other users in synchronous sessions as well as fast response times for event synchronization, and
– efficiently storing the graph information and concurrency handling during collaborative sessions.

OnGraX is capable of visualizing graphs from various application domains, such as social networks (e.g., Facebook), networks in software engineering (UML-diagrams among other things), or biochemical networks. Currently, our tool is used for the collaborative analysis and data cleaning [6] of metabolic networks due to long lasting cooperations with biologists/bioinformaticians at several research institutions. This specific application serves as use case throughout this paper and focuses on the interactive exploration and analysis of biological networks based on the so-called Kyoto Encyclopedia of Genes and Genomes pathway database [7]. Building these biological networks is often based on complex experiments. In consequence, biologists of different domains and experience levels would like to explore the resulting networks together and check them for wrong entries or missing data and revise the networks wherever it is necessary.

The remainder of this paper discusses our solution for these challenges. The next section covers related work in graph visualization and web-based collaborative visualization. Our requirements are described in Section 3. An overview of our tool is given in Section 4, and the implementation as well as technical details are covered in Section 5. We discuss the benefits of our approach and future work in Section 6 and conclude in Section 7.

2 Related Work

Desktop applications for single users with the possibility to visualize graphs, such as Cytoscape [8], Gephi [9], Pajek [10], or Tulip [11], have been used before the advent of the Web 2.0. Available methods for graph visualizations are comprehensively overviewed in several surveys and books, for instance [12–14]. However, the idea of working collaboratively on complex data sets is becoming more and more attractive with the rapidly growing amount of data available. Thus, a good number of web-based visualizations were already introduced that support the public exploration of complex data sets [15]. Users are able to add comments on interesting visualizations and also discuss new insights with other users over the web. Several systems support those features, such as ManyEyes [16] or Sense.us [17]. In these web-based visualization systems, users are able to save bookmarks of specific views on a data set, and it is possible to add graphical annotations directly to the visualization. Dashiki [18] enables users to collaboratively build visualization dashboards with the help of a wiki-like syntax and interactive editors. However, the aforementioned systems are not suitable for our tasks, since they do not support the interactive visualization of node-link diagrams in a web browser. And, they do not provide any features for real-time interactions during synchronous collaboration settings.

Drawing graphs in a web browser is usually done with the help of already existing JavaScript libraries. Arbor.js [19] and Sigma.js [20] render the input graphs on a HTML5 canvas or via SVG-images, but they do not support OpenGL-enhanced rendering which limits the number of nodes and edges that can be rendered with a suitable frame rate. A faster solution is provided by VivaGraphJS [21]: this library uses a WebGL renderer to draw graphs and provides high performance during rendering, but gives only limited support for different and more complex node shapes. Additionally, these libraries are not able to efficiently render a lot of text, e.g., node labels for all nodes visible at the same time, which is one of the requirements for our tool as described in the next section.

A good overview of the field of collaborative visualization is given in the article of Isenberg et al. [3]. We follow their terminology and classify our approach as distributed, synchronous/asynchronous collaborative visualization method. In this work, we restrict ourselves on the collaborative visual analysis of networks.

For further readings on general aspects of visualization, human-computer interaction, computer-supported collaborated work, and collaborative visualization of other data types, we refer to the standard literature, for instance [2,15,22–25] (this list does not claim to be exhaustive).

The benefits of collaborative work are also discussed in an article on social navigation presented by Dieberger et al. [26]. Being able to see the usage history and annotations of former users might help analysts to filter and find relevant information more quickly. In order to be able to work together during a synchronous session, users have to know each other's interactions and views on the data set, usually referred to as "common ground" [1,2]. To find a common ground in node-link visualizations, we apply the techniques from the work of Gutwin and Greenberg [27] and show the viewports of other users as rectangles in the graph visualization.

There are also existing groupware frameworks and libraries to handle collaborative editing in the web with concurrency control, such as ShareJS [28] or Apache Wave [29]. They usually center on manipulating the DOM of collaborative websites and editing text in online documents without creating conflicts. Our approach concentrates on collaboration and awareness in a node-link based graph visualization which does not require to edit a lot of textual data, i.e., we focus on changing the structure and attributes of the graph instead. For this case, it suffices to use a traditional lock-based approach.

3 Requirements

Our goal was to design a system for analyzing complex graphs in distributed collaborative sessions. The initial process of beginning a collaboration should be as fast and easy as possible, thus the tool should also be available on the fly without requiring users to install specific software, plug-ins, or Java applets first. Additionally, users should be able to drop in and out of ongoing collaborative work without having to setup and plan each session individually—experts would like to work on a data set whenever they find the time and do not want to wait for others to join before they can start their analysis process. In this case, users who are joining an already ongoing session should be able to quickly catch up on what has been done before. Hence, the system should support logging all actions that are performed in a session and provide a way to quickly retrieve and analyze them to show important information to subsequent users. These actions include not only changes performed on a graph (such as deleting or changing the attributes of a node), but also the tracking and logging of every user's camera position. This assists subsequent analysts to identify regions of a graph that other users were already interested in if they work on the data set asynchronously. During a synchronous session however, changes applied to the graph should be distributed to all connected clients as fast as possible and users should be able to track each other's mouse and camera positions in real time to establish a common ground and be aware of everything that is going on in a session. The system also has to store the complete graph structure and additional graph attributes in an efficient way. And, it should handle conflicts that could arise during a collaborative session if two users want to change the same object in the graph at the same time.

In addition to the graph structure itself, nodes usually hold attributes (e.g., age, gender, or income for social networks). Rendering the nodes as simple dots is therefore not sufficient enough for our application areas since the shape, size, and color of nodes may have specific meanings; nodes may also have labels attached. Thus, our system should be able to render graphs with a considerable number of nodes and edges of different shapes and sizes with additional text labels on every node in an acceptable frame rate. Moreover, running computationally expensive tasks—such as calculating the layout of a graph, computing additional graph metrics, or aggregating the camera positions to show interesting regions—should not have a negative impact on the rendering performance on the clients. We summarize the technical requirements (TRs) of our tool as follows:

- TR 1. It should be possible to start a collaborative session on the fly without having to install software or plug-ins first.
- TR 2. The system should be able to render graphs with up to 6,000 nodes and edges with a good performance on standard computers.
- TR 3. It should be able to render nodes in various shapes and colors, with additional text labels on every node.
- TR 4. Changes to the graphs and mouse positions as well as viewports of other users should be distributed to all connected clients in (soft) real time.
- TR 5. The server should efficiently store the graph structure as well as the complete action history of a graph session in order to use this data for subsequent analysis processes.
- TR 6. The system should be able to manage concurrent graph changes during a collaborative session.
- TR 7. Computationally expensive processes, such as calculating the layout of a graph, should not interfere with the rendering performance of the graph visualization.
- TR 8. As the most graphs come with node and/or edge labels, the system should support fast and efficient text rendering.

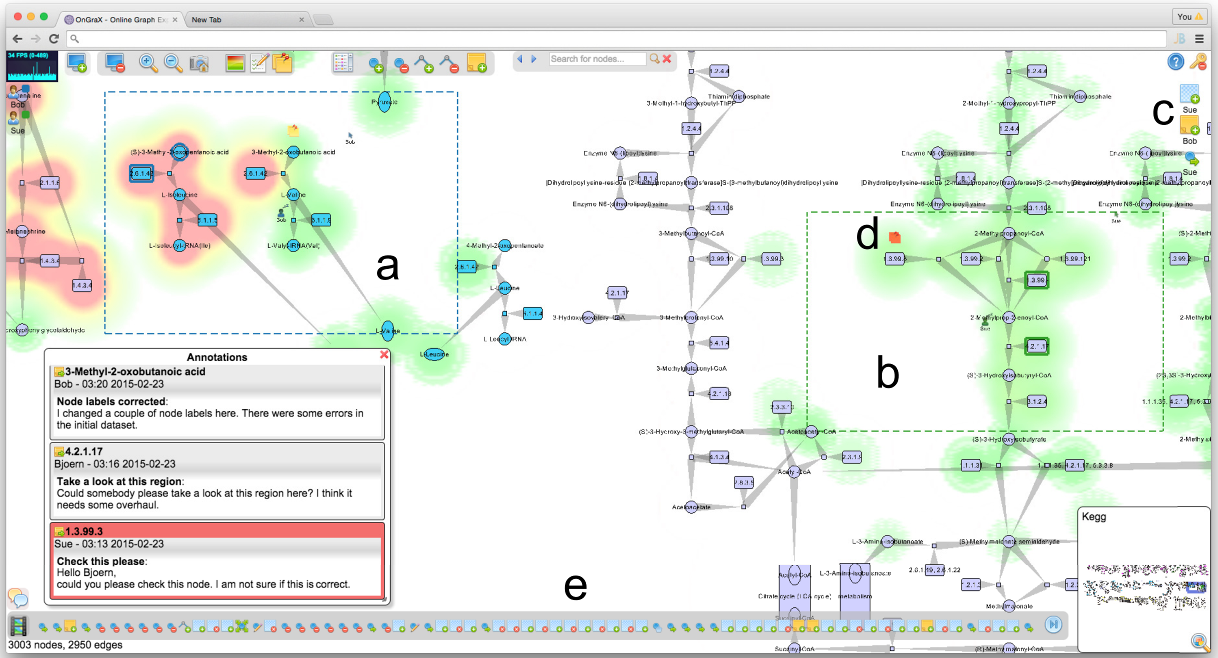
4 General System Overview

OnGraX visualizes graphs as interactive node-link diagrams. Fig. 1 shows an overview of the tool right after joining an ongoing graph analysis session with two other users: Bob and Sue. Their viewports are represented as two dashed rectangles. Bob's position is shown in blue and Sue's position is shown in green (see Fig. 1(a) and (b)). The rectangles also contain their mouse cursors which are updated in real time. This makes it possible to point at interesting objects and coordinate collaborative work. Short animations (created with the tween.js [30] library) are used to facilitate user awareness about ongoing changes during a session. For instance, if nodes are deleted they will slowly grow in size and fade out instead of just disappearing. Recent actions which are performed during a session are tracked as small icons on the right-hand side of the screen (see Fig. 1(c)). These icons can be clicked to move the camera back and forth between the current camera position and the position where the event took place, enabling users to quickly check the ongoing work of other users. To not get overwhelmed with notifications, an additional dialog box allows users to configure which types of actions are tracked.

Apart from a simple chat window where users can discuss changes, it is also possible to pin textual annotations to nodes and edges. This gives analysts the possibility to pass information about performed changes in the graph to subsequent users, ask questions about specific objects, or delegate work to other users. In Fig. 1(d), an annotation that is pinned to a node is highlighted, along with the respective text in the annotations dialog in the bottom right corner. The text in the annotations dialog can also be clicked to move the camera to the respective annotated object in the graph if it is not in the current view. One problem with textual annotations is, that the original context in which an annotation was initially written could get lost if the respective graph region—where the annotation is pointing to—is changed during the course of a session or if the object with this annotation is deleted. We solve this problem by enabling analysts to temporarily revert the complete graph to an old state, giving them the possibility to view the graph in a state in which the annotation was originally written. This is done by clicking on any icon in the timeline at the bottom of the screen (see Fig. 1(e)). This action will undo all subsequent changes that were performed on the graph. While viewing an old state, icons to the right can be clicked to replay already performed changes and go forward in time, and icons on the left are used to undo additional changes to view even older states of the graph.

Analysts who join a session would like to be able to quickly find out which graph regions were already viewed or changed by other users, for instance to decide if they should work on another part of a graph or if they should double check on specific regions. In our case, we had to use a visualization method that does not interfere with the original graph visualization. Changing the colors or the size of nodes in the graph was not an option for us, because these properties are already mapped to other attributes. To still be able to show additional user data without affecting the original graph visualization, we use a heat map-based visualization approach. The heat map is drawn in the background of the graph visualization. It can be configured to only show data for specific users or to show data for all users together. Furthermore, it is possible to select a time frame (e.g., the last five minutes of the current analysis session) or a specific start and end date. This enables an analyst to review changes done in a collaborative session during a specific time frame or to check the work of a single user. The heat map can be configured to show two different data sources:

User viewports. Here, nodes are highlighted that were in the viewing area of all users during their analysis process. The server aggregates the logs of all node movements and user views to visualize the amount of seconds that nodes were in the viewports of all or specific users. This helps analysts to



Annotations

3-Methyl-2-oxobutanoic acid
Bob - 03:20 2015-02-23
Node labels corrected:
I changed a couple of node labels here. There were some errors in the initial dataset.

4.2.1.17
Bjoern - 03:16 2015-02-23
Take a look at this region:
Could somebody please take a look at this region here? I think it needs some overhaul.

1.3.99.3
Sue - 03:13 2015-02-23
Check this please:
Hello Bjoern,
could you please check this node. I am not sure if this is correct.

Fig. 1. Overview of our system. The image shows a part of a biochemical network with 3,003 nodes and 2,950 edges. The blue and green dashed rectangles (see (a) and (b)) are the viewing areas (viewports) of two other users who are exploring this graph simultaneously. The viewports and the mouse cursor of every user in a graph session are updated in (soft) real time. Here, the heat map in the background assists the analyst to identify regions of the graph that were of interest to the other users by highlighting all nodes based on their logged viewports. The symbols in the top-right corner of the screen (c) assist the analyst to keep track of recent actions performed by other users, such as adding or deleting nodes. Text annotations can be pinned to nodes and edges to discuss tasks, insights and questions with other users (d). The timeline (e) can be used to revert the graph to a previous state and replay applied changes.

quickly identify in which part of a graph another user was interested in and also to find out if there are any regions that were not viewed at all.

Graph changes. This option calculates a heat map based on the number of changes that have been performed on graph objects, such as modifying the shape or color of a node, adding a new node, or adding edges to a node. A multiplier can be specified for each individual action type to give it more or less weight during the calculation. This gives analysts the possibility to search for renamed and moved nodes only, for instance.

5 System Setup and Architecture

We decided to implement OnGraX as a web-based tool to address our first technical requirement (TR 1). OnGraX' client/server-based architecture enables us to provide an easily accessible tool for graph exploration over the web which allows analysts to simply open a web browser to start a collaborative session. Figure 2 illustrates the basic architecture of our system. On the client side, all graphs are rendered with the help of the client computer's GPU by using WebGL [31], a JavaScript API for rendering 3D graphics natively in modern web browsers. For our special case, this is faster than using SVG-based node-link visualization approaches, such as the d3.js visualization library [32]. To ease the process of low-level OpenGL programming, we utilize the JavaScript library three.js [33]. This approach tackles the second and third technical requirements (TR 2-3) and enables us to achieve a high-performance rendering of node-link diagrams.

The server side of OnGraX is implemented in Java EE and currently runs as a web application on an Apache Tomcat server. Communication between server and clients is done with the relatively new WebSocket protocol [34]. It suffers less from network overhead since it is layered over TCP instead of HTTP which removes the overhead from HTTP header fields and allows for a low

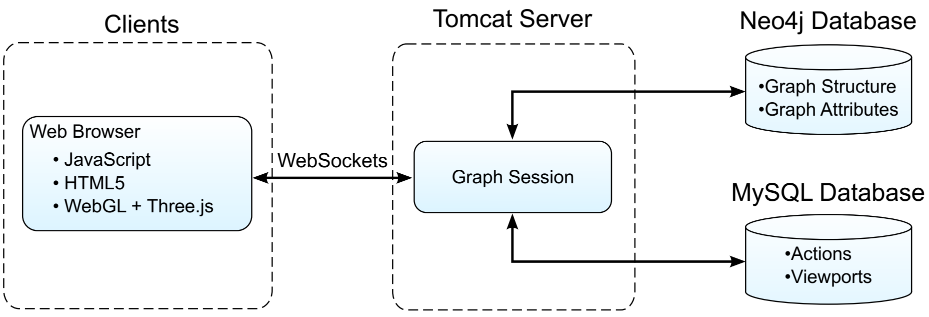


Fig. 2. General architecture of our system. Sessions are initialized on demand whenever a user joins a graph analysis session. The client-server communication is done via WebSockets. Every graph is stored in a separate Neo4j database, whereas all performed actions of a session are logged in a MySQL table.

latency two-way communication. Clients do not have to poll the server in regular intervals anymore to ask for updates. Instead, the server can send a message to all connected clients whenever an update on the client side is required. This allows our system to track the viewports and mouse positions of other users and distribute this information among all clients in real time and addresses our fourth technical requirement (TR 4).

Each graph is stored in a separate Neo4j [35] database. As soon as a user joins a graph session, the respective Neo4j database is initialized as an embedded database service on the server. As a graph-based database, Neo4j offers a convenient way to store our graphs together with all of their node/edge attributes and supports graph-like queries, such as shortest path calculations. It also simplifies other graph-related queries for community detection or applying clustering algorithms. For the remaining data, such as user data and login information, our system uses a MySQL database. Performed actions and all camera positions that are generated by the users while they are exploring a graph are also stored in a MySQL table. This data can be used later to visualize regions of a graph that were modified or viewed by other users. Using Neo4j to efficiently store the graph structure in conjunction with a MySQL database to log all events enables us to tackle our fifth technical requirement (TR 5).

Technical requirement number seven is addressed by using the server side part of our system for complex processes (TR 7). The Tomcat server currently runs on a Dell PowerEdge R720 with two Intel Xeon E5-2650 2.00GHz processors (eight cores each), 128GB RAM, and a Value MLC 3G SSD hard drive. This configuration provides more than enough computing power for our current purpose and future extensions like the calculation of complex graph layouts or running graph analysis algorithms on the server and distributing the results to all clients. Right now, all graphs analyzed with our tool already have precomputed layouts, but computing a layout for other graphs could easily be achieved by using a Java-based graph layout library, such as Jung [36].

5.1 Action and Conflict Handling

Since our system focuses on the collaborative exploration of graphs and not in the collaborative editing of text documents, we do not need sophisticated concurrency control systems which are usually used in such a case (e.g., operational transformation [37]). A pessimistic locking approach is sufficient enough for our scenario, since changes are usually performed directly on one or a couple of single nodes and node attributes. To address our sixth technical requirement (TR 6), we use a simple server-side queue to ensure that all clients visualize the same data structure. Whenever a user performs an action that would change the graph structure or any of the graph object's attribute—such as moving a node, changing the shape of a node, or adding a new edge between two nodes—an action request event is sent to the server. The server uses Neo4j's transaction system to open a new transaction and apply the changes to the stored graph. Only if the transaction is successful, the server reports this back to all connected clients, including the client who initiated the action. Figure 3 illustrates this approach.

All incoming actions are handled in a server-side queue, and an event is only applied if it is not in conflict with a previous event. This could happen if a user deletes a node, while a second user tries to add an edge to this node at exactly the same time. If the server processes the node delete action first, the second user's action will not be performed on the graph structure, and the user will receive a short notification instead, while his/her local graph visualization is updated to reflect the changes that where performed by the first user.

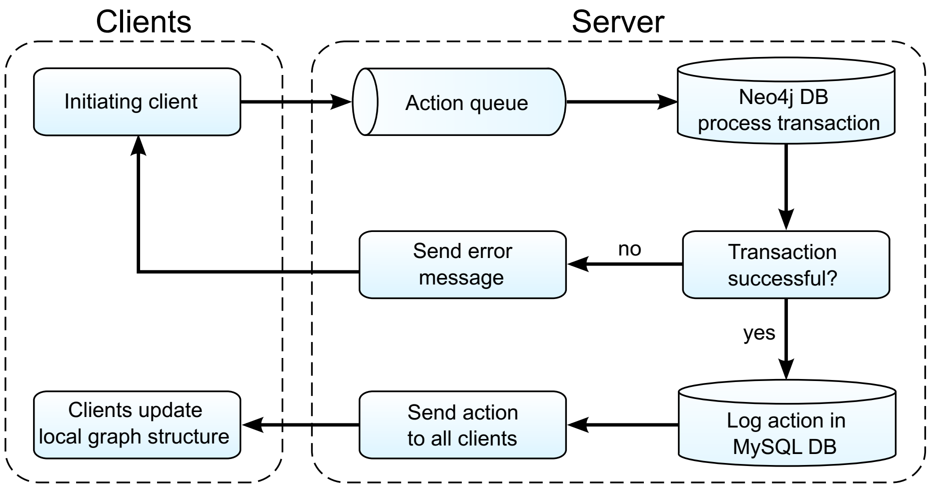


Fig. 3. Action handling between clients during a graph session. To avoid concurrent changes, actions that would affect the graph structure are sent as an action request event to a queue on the server. All clients only update their local graph visualization, if the transaction was applied successfully to the Neo4j database. In case of a conflict the action is not applied, and the initiating client receives an error message.

5.2 Calculation and Visualization of the Heat Map

As discussed in Section 4, we use a heat map-based visualization in the background of the node-link diagram (cf. Figure 1) to either show aggregated values of all logged user views to find regions of interest or applied node changes to get an overview of changes that were performed on a graph. If the heat map configuration is set to show the logged user views, the server calculates the heat map values by aggregating all logged user views with all node positions from all move actions that have been performed previously on the nodes. This process results in a heat map visualization that is robust against layout changes of the graph.

After the values are calculated and sent to the requesting client, they have to be visualized in the web browser without negative performance effects on the interactive graph visualization. To avoid having to render a lot of additional objects, we first draw the heat map values on an off-screen canvas element and create a single OpenGL texture from this canvas afterwards. The canvas represents the heat map values as an alpha map—for every node, a circular gradient is drawn which is based on the size and position of its related node. This creates an image with grayscale values ranging from 0 to 255. To create the actual colors for the heat map, each pixel value is used to lookup the color from a 1×256 pixel-wide color gradient. The gradient colors range from white, over green to red. Based on these color values, an OpenGL texture is created and put on a mesh in the background of the graph visualization. By using an alpha map as basis, we could also draw a heat map based on mouse positions or eye tracking data easily.

Getting the heat map values for all actions that were applied to all nodes of a graph only takes around 4 milliseconds on the server for a typical graph analysis session. Calculating the heat map based on user views and node positions takes considerably more time, because the server has to correlate all stored views with all logged node positions to get the number of seconds every node of the graph was viewed by a user while also considering layout changes. During our test sessions, the server calculated those values in about 60 milliseconds if the complete time frame was selected. The time also depends on the amount of stored user actions and node positions and will of course increase slightly if a graph analysis session is used for a longer time period and more actions are logged. However, since analysts are usually only interested in specific time frames which span over the course of a few days, this is not an important issue. Right now, the bottleneck during the heat map generation lies on the client side. The largest graph that is currently visualized with our system spans an area of 5,800×3,600 pixels. Drawing the alpha map on a canvas with the same size would take approximately 20 seconds, which is not adequate for generating a real-time heat map. Another problem occurs during the creation of the OpenGL texture, as the maximum texture size is limited by the client's graphic card. Current graphic cards support textures with up to 16,384×16,384 pixels, whereas older computers are limited to 2,048×2,048 pixels. We avoid this performance problem on the client side and achieve a real-time rendering for the heat map by drawing a scaled-down version of the alpha map if the graph area exceeds the size of 2,048×2,048 pixels. The resulting texture is then put on a mesh in the background of the visualization and stretched to the appropriate size of the graph.

An alternative and faster approach for generating the heat map would be to create the alpha map and the texture array directly on the server and send the array for the texture to the requesting client(s). But this would drastically increase the amount of traffic between server and clients. Therefore, we decided against this idea and settled for the slightly slower approach of drawing the heat map on the client side.

5.3 Rendering Text with WebGL

Drawing a large number of node labels was another challenge and requirement (TR 8) that we had to solve. The most convenient way—which can also be found in three.js forums—is to draw each node label on an offscreen canvas, render this canvas to an image and use this image as a sprite texture. This works well for a small amount of text strings. But for graphs with more than just a couple of hundred nodes, the JavaScript engine would have to create and render a large amount of textures, which is not fast enough and will eventually crash the JavaScript engine. To solve this, we adapted and modified the JavaScript-based solution from an online article of Heikkinen [38] for bitmap fonts, a common technique used in standard OpenGL rendering. Instead of creating a texture for each label, we just use one texture with all required letters on it. For each letter in a node label, two triangles are drawn and only the part of the texture with the position of the letter is mapped to this geometry. This is an extremely fast solution and the only drawback is, that the fonts do not scale nicely at very high zoom levels.

5.4 Performance

Upon joining a graph session, the complete graph structure is transferred to the client. This approach is fast enough for graphs with up to 6,000 nodes and edges. The download and initialization of all graphical objects on the client takes around three seconds. Unfortunately, this approach does not work for graphs with more than 6,000 nodes as it takes simply too long to transfer the whole data set to the client and generate all graphical objects in one single step. This may eventually lead to an error message in the client's browser stating that JavaScript is not responding anymore. Streaming the graph to the client by only sending an initial part and expanding the information as soon the user zooms out or further explores the graph would be a more convenient step and is planned for a future version of our tool. Picking up the idea of Gretarsson et al. [39], it would also be possible to render parts of the graph on the server and send them as images to the client where they are visualized until the complete data is loaded.

After the initialization phase, our tool renders a zoomed-out overview of our test graphs—which usually contain about 3,000 labeled nodes and edges—with 20 frames per second and 30-40 frames per second in a standard zoomed-in view on an early 2011 MacBook Pro (2,2GHz i7, 8GB memory, Radeon HD 6750M with 1,024MB video memory) with a screen resolution of 1,680×1,050 pixels. We also did a performance test with the same computer on a 4K monitor with a resolution of 3,840×2,160 pixels. Here, the overview is rendered with 10 frames per second and 20-28 frames per second for the zoomed-in view. The biggest graph visualized by with our system by now had 3,700 nodes and 7,500 edges. The initial data transfer while joining the session took about ten seconds. While rendering the complete overview of the graph was only possible with five frames per second, the standard zoom level of our users while working within the graph was rendered at around 10 to 15 frames per second with a 1,680×1,050 resolution. While this is not an optimal frame rate, the visualization still proved to be useable by our test users. Increasing the performance for bigger graphs is one of the next planned steps in our future work as discussed in the following section.

6 Discussion

We made our tool available to various experts in systems biology and bioinformatics at Monash University, Australia, and received mostly positive and also constructive feedback. They liked the idea of working together on their data sets by simply opening the visualization in a browser window. Seeing each others camera position in a synchronous session made it a lot easier for them to discuss and change specific parts of the graph, although one group of experts missed a voice chat directly in the tool. They would have preferred to be able to talk to each other directly without having to fall back to other programs, such as Skype or Google Hangouts. This could be addressed in a future version of our tool with the help of the new WebRTC standard [40] for real-time communications in browsers. Another group of biologists found the heat map visualization of user behavioral data quite useful. They would like to use OnGraX for the education of their students. A use case here would be to give students an already edited graph and ask them to revise the data set further as well as to verify the changes that have already been performed by previous users. Afterwards, the supervisors could review the steps that the students performed and also discuss and reflect the process online together with the students in a collaborative session.

There are still some technical issues that have to be addressed to improve the performance and usability of OnGraX. One task is to utilize Web Workers [41] to speed up the client-side part of the heat map generation. The user interface sometimes becomes unresponsive—depending on the client's hardware—for up to two seconds while the JavaScript engine generates the heat map texture. This caused a small inconvenience among some of our test users. Web Workers could be used to finish this calculation in a thread-like manner in the background of the web application, allowing the web page and user interface to remain responsive all the time.

In general, our tool is able to handle graphs with 10,000 nodes and edges on faster computers as we stated in [4], but some users with slower computers had performance issues during the exploration of graphs with more than 6,000 nodes. Using streaming techniques together with the idea of WiGis [39] to render parts of the graph on the server and only show them as images on the clients would be another possibility to improve the rendering performance and initialization times. Another technique would be to still send the whole data to the clients, but to render specified areas of a graph only once into a texture instead of rendering all graph objects in every frame. Users could then manually select areas to be rendered as a texture and only switch to full rendering on demand. If a subgraph that is rendered as a texture for one user is changed on another client, the changes only have to be rendered once again into the texture on the first user's client to update the information. Adapting this technique would enable us to visualize even bigger graphs with more than 10,000 nodes.

After implementing the aforementioned improvements, the next step will be to conduct a detailed user study to improve the user interface, find missing features and get detailed feedback about the usability of our system.

7 Conclusion

In this paper, we presented a collaborative system for visualizing graphs with several thousands of nodes and edges in a web-based environment. By using WebGL, the system is able to provide an interactive graph visualization with up to 6,000 labeled nodes and edges. With the help of our client/server-based system, analysts do not have to install any additional applications or browser plug-ins anymore. The start of a collaborative analysis session is simply done by opening a URL in a browser window. With this fundamental property of our visualization environment, we address one of the research challenges given in the article of Isenberg et al. [3]. Another challenge named by these authors is the need to develop so-called hybrid collaboration scenarios. Here, we provide users visualization and interaction techniques for analyzing data sets synchronously *and* asynchronously in a distributed environment. With the help of our tool, users can seamlessly drop in and out of ongoing sessions and do not have to wait for other users to start or finish their work. All actions performed during a session as well as the users' camera positions are tracked and can be visualized along with the graph data by using underlying heat map representations. This helps experts to analyze regions of a graph that were of interest or have been edited by former users, i.e., social navigation and guidance is maintained by the aggregated user activity in form of heat maps. Here, we address the "Collaboration & Awareness" challenge in web-based collaborative visualization that was raised by Heer et al. [15]. In the synchronous collaboration case, we provide another contribution (cf. "Pointing & Reference" challenge in [15]) namely that participants of an analysis session can follow the activities of the others (shared viewing areas) and point to specific nodes or edges by using brushing (shared node markers and mouse cursors). The use of WebSockets enables us to distribute this user data in real time among all connected clients.

Acknowledgments. The authors would like to thank Falk Schreiber and his colleagues at Monash University in Australia for constructive discussions and valuable feedback, as well as all students at Linnaeus University in Sweden who participated in our initial user studies.

References

1. Chuah, M., Roth, S.: Visualizing common ground. In: Proceedings of the International Conference on Information Visualization (IV 2003), pp. 365–372. IEEE (2003)
2. Heer, J., Agrawala, M.: Design Considerations for Collaborative Visual Analytics. Information Visualization **7**(1), 49–62 (2008)

3. Isenberg, P., Elmqvist, N., Cernea, D., Scholtz, J., Ma, K.-L., Hagen, H.: Collaborative Visualization: Definition, Challenges, and Research Agenda. Information Visualization **10**(4), 310–326 (2011)
4. Zimmer, B., Kerren, A.: Applying heat maps in a web-based collaborative graph visualization. In: Poster Abstracts, IEEE Information Visualization (InfoVis 2014), France, Paris (2014)
5. Zimmer, B., Kerren, A.: Sensemaking and provenance in distributed collaborative node-link visualizations. In: Abstract Papers, IEEE VIS 2014 Workshop: Provenance for Sensemaking, France, Paris (2014)
6. Kandel, S., Heer, J., Plaisant, C., Kennedy, J., van Ham, F., Riche, N.H., Weaver, C., Lee, B., Brodbeck, D., Buono, P.: Research directions in data wrangling: Visualizations and transformations for usable and credible data. Information Visualization **10**(4), 271–288 (2011)
7. KEGG: Kyoto Encyclopedia of Genes and Genomes. http://www.genome.jp/kegg/ (accessed July 10, 2014)
8. Shannon, P., Markiel, A., Ozier, O., Baliga, N.S., Wang, J.T., Ramage, D., Amin, N., Schwikowski, B., Ideker, T.: Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome Research **13**(11), 2498–2504 (2003)
9. Bastian, M., Heymann, S., Jacomy, M.: Gephi: an open source software for exploring and manipulating networks. In: International AAAI Conference on Weblogs and Social Media (2009)
10. Batagelj, V., Mrvar, A.: Pajek - analysis and visualization of large networks. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 477–478. Springer, Heidelberg (2002). http://dx.doi.org/10.1007/3-540-45848-4_54
11. Auber, D.: Tulip: data visualization software. In: Graph Drawing, pp. 435–437 (2001)
12. von Landesberger, T., Kuijper, A., Schreck, T., Kohlhammer, J., van Wijk, J., Fekete, J.-D., Fellner, D.: Visual analysis of large graphs: State-of-the-art and future research challenges. Computer Graphics Forum **30**(6), 1719–1749 (2011). http://dx.doi.org/10.1111/j.1467-8659.2011.01898.x
13. Kerren, A., Purchase, H.C., Ward, M.O.: Multivariate Network Visualization, ser. Lecture Notes in Computer Science, vol. 8380. Springer (2014)
14. Kerren, A., Schreiber, F.: Network visualization for integrative bioinformatics. In: Chen, M., Hofestädt, R. (eds.) Approaches in Integrative Bioinformatics - Towards the Virtual Cell, pp. 173–202. Springer, Heidelberg (2014)
15. Heer, J., van Ham, F., Carpendale, S., Weaver, C., Isenberg, P.: Creation and collaboration: engaging new audiences for information visualization. In: Kerren, A., Stasko, J.T., Fekete, J.-D., North, C. (eds.) Information Visualization. LNCS, vol. 4950, pp. 92–133. Springer, Heidelberg (2008)
16. Viégas, A.B., Wattenberg, M., Ham, F.V., Kriss, J., Mckeon, M.: Many eyes: A site for visualization at internet scale. IEEE Transactions on Visualization and Computer Graphics **13**(6), 1121–1128 (2007)
17. Heer, J., Viégas, F., Wattenberg, M.: Voyagers and voyeurs: supporting asynchronous collaborative information visualization. In: ACM Human Factors in Computing Systems (CHI), pp. 1029–1038 (2007)
18. McKeon, M.: Harnessing the Information Ecosystem with Wiki-based Visualization Dashboards. IEEE Transactions on Visualization and Computer Graphics **15**(6), 1081–1088 (2009)
19. Samizdat Drafting Co. Arbor.js. http://arborjs.org (accessed January 2015)

20. Jacomy, A.: sigma.js. http://sigmajs.org (accessed January 2015)
21. Kashcha, A.: Vivagraphjs. https://github.com/anvaka/VivaGraphJS (accessed January 2015)
22. Kerren, A., Ebert, A., Meyer, J. (eds.): Human-Centered Visualization Environments, ser. LNCS Tutorial, vol. 4417. Springer, Heidelberg (2007)
23. Dix, A., Finlay, J.E., Abowd, G.D., Beale, R.: Human-Computer Interaction, 3rd edn. Prentice Hall, London (2003)
24. Baecker, R.M.: Readings in GroupWare and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (1994)
25. Kerren, A., Stasko, J.T., Fekete, J.-D., North, C. (eds.): Information Visualization, Human-Centered Issues and Perspectives, ser. Lecture Notes in Computer Science, vol. 4950. Springer (2008)
26. Dieberger, A., Dourish, P., Höök, K.: Social Navigation: Techniques for Building more Usable Systems. Interactions **7**(6), November 2000
27. Gutwin, C., Greenberg, S.: Design for individuals, design for groups: tradeoffs between power and workspace awareness. In: Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work, ser. CSCW 1998, pp. 207–216. ACM, New York (1998)
28. Gentle, J.: ShareJS - Live concurrent editing in your app. http://sharejs.org (accessed January 2014)
29. The Apache Software Foundation. Apache Wave. http://sharejs.org (accessed January 2014)
30. TweenJS. http://www.createjs.com/TweenJS (accessed January 2015)
31. Khronos Group. WebGL Specification. Editor's Draft 1, July 2014. http://www.khronos.org/registry/webgl/specs/latest (accessed January 2015)
32. Bostock, M.: D3 - data-driven documents. http://threejs.org (accessed January 2015)
33. Cabello, R.: Three.js. http://threejs.org (accessed January 2015)
34. World Wide Web Consortium. The WebSocket API. http://dev.w3.org/html5/websockets/ (accessed January 2015)
35. Neo Technology, Inc., Neo4j. http://neo4j.com (accessed January 2015)
36. O'Madadhain, J., Fisher, D., Nelson, T.: JUNG - Java Universal Network/Graph Framework. http://jung.sourceforge.net/ (accessed January 2015)
37. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. SIGMOD Rec. **18**(2), 399–407 (1989)
38. Animating a Million Letters Using Three.js. http://www.html5rocks.com/en/tutorials/webgl/million_letters/ (accessed January 2015)
39. Gretarsson, B., Bostandjiev, S., O'Donovan, J., Höllerer, T.: WiGis: a framework for scalable web-based interactive graph visualizations. In: Eppstein, D., Gansner, E.R. (eds.) GD 2009. LNCS, vol. 5849, pp. 119–134. Springer, Heidelberg (2010)
40. World Wide Web Consortium. WebRTC. http://www.w3.org/TR/2015/WD-webrtc-20150210/ (accessed January 2015)
41. Web Workers. http://www.w3.org/TR/workers/ (accessed January 2015)