Towards Cross-Platform Collaboration - Transferring Real-Time Groupware To The Browser

Matthias Wenzel, Lutz Gericke, Raja Gumienny, Christoph Meinel
Hasso Plattner Institute Potsdam
Prof. Dr. Helmert Str. 2-3, Potsdam, Germany
Email: {firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—Mobile devices such as smartphones and tablets play an increasing role in today's working environment. The variety of computer platforms increased in the same way, which makes the development of cross-platform applications even more challenging. Tele-Board is a real-time remote collaboration system based on the Java programming language. Therefore, it cannot be run on most mobile devices. In order to overcome this limitation, we redeveloped the system on the basis of HTML5 technology. We present an approach for combining web based networking and rendering in a single application for real-time collaboration based on SVG, HTML5 Canvas, Websockets, and Web workers. In our prototype we implemented optimization mechanisms leveraging the Canvas API's rendering flexibility. This way, our canvas based rendering performs better than a respective SVG version. Moreover, our solution integrates server communication effectively so that the rendering performance is hardly influenced by user input.**

I. Introduction

With their growing maturity, mobile devices become increasingly capable of hosting technically demanding applications. This way, opportunities arise to seamlessly use tools such as real-time groupware systems on mobile as well as desktop platforms. There are different approaches for developing cross-platform software applications. Either creating native programs for each specific operating system or using a platform-independent programming environment provided by e.g. the Java virtual machine. Due to the increasing platform variety, applying these concepts is becoming more difficult. The effort for development and maintenance of native platform specific applications rises significantly. On the other hand, platform- and vendor-independent runtime environments are not available across desktop and mobile devices.

Web browsers have evolved to central software systems that even enable advanced applications [1][2] such as web based games or office software suites [3] on most desktop and mobile systems. Many of those web based applications are relying on browser plugins. The wide spread plugin technologies Adobe Flash[1] and Microsoft Silverlight[2] are hardly available on mobile platforms such as iOS and Android. Moreover, Adobe announced that they will no longer continue to develop Flash on mobile devices [4]. Plugin technology in the web browser is therefore not an option for cross-platform development.

[1]http://www.adobe.com/software/flash/about/
[2]http://www.microsoft.com/silverlight/

In this paper, we present and evaluate a browser based prototype using HTML5 technology for replacing a Java based component of our Tele-Board [5] collaboration system. We created a web based application that runs on desktop as well as mobile devices. Our system is therefore available to a broader range of users while keeping maintenance effort limited. Other research in the field of real-time groupware and web technologies focused either on web based networking *or* rendering techniques that can be used in a web browser. In our work, we integrate technologies for web based networking and rendering in a single application. We elaborate on techniques for decoupling network communication and user input handling. Besides implementing a browser based solution, we also tested our prototype's performance on common mobile and desktop platforms.

II. Overview of the Tele-Board system

Tele-Board is a digital whiteboard system allowing creative teams to work together over geographical and temporal distances. This collaboration can be synchronous, when people are able to work at the same time, or asynchronous, which means people can build upon the content of their co-workers.

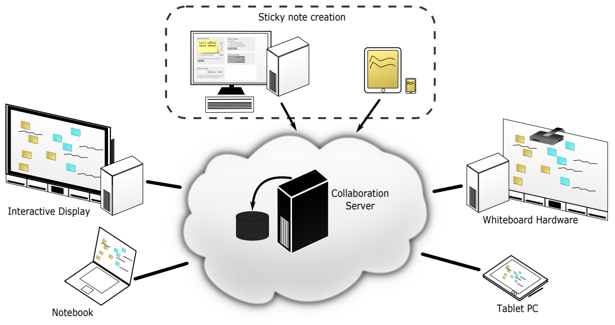

Fig. 1. The Tele-Board software system architecture

The existing system consists of different components (see Figure 1), each especially designed for fulfilling a specific task:
Whiteboard Client: Serves as the main user interface by presenting the digital whiteboard drawing surface. It enables people to do the work they are used to from traditional whiteboards by using common metaphors such as written notes or sketches. The client runs on any Java-enabled computer,

while often a digital whiteboard is used as an input device. During development, we had different kinds of hardware in mind, which we evaluated and prototyped for. When we started the development, 95% of all computers had a Java runtime environment installed. Furthermore, Java applications were reliable and very common on different platforms.
The communication server: Every whiteboard client connects to this server and thereby subscribes to updates of a specific panel. Content is organized in projects and panels and a project can consist of multiple panels. A panel is a whiteboard session, where people are working on, including the whole history of interaction. Changes made on one location are automatically synchronized to all other locations.
Tele-Board History: This component is embedded into the server. Every bit of synchronization communication is captured and immediately stored in a central database. This allows to keep track of the whole process of interaction and not only certain snapshots. *Every* single operation is stored, which allows applications to resume from any point in time during the past lifetime of the panel, copy sessions, or run statistics on the usage.
Mobile devices: Although content of any kind can be created directly in the whiteboard client, we want to supply people with a variety of mobile devices realizing an efficient workflow that includes (digital) sticky note pads. Optimally, they can use their own devices. Therefore, we developed applications for different platforms. People can use their iOS or Android based devices and can take pictures, make drawings or just input text. The content is sent to a panel and can be arranged on the whiteboard surface.

III. Related work

As we focus on browser based real-time groupware systems, we provide a short overview of evaluations concerning HTML5 standard based web technologies.

Gutwin et. al [6] deal with web based networking for real-time groupware. Many networking techniques that can be used in web browsers such as *Asynchronous JavaScript and XML* (AJAX) and also *Websockets*[3] are described in detail. Moreover, the performance of these methods is measured. They recommend the browser as a suitable environment for real-time groupware. Additionally, they show that Websockets have the highest performance among standards based networking web technolgy. This way, Websocket is the technolgy of choice in our implementation.

In contrast to that, Hoetzlein [7] focuses on rendering technologies such as *HTML5 canvas*[4], Flash and *WebGL*[5]. He develops a test suite for measuring graphics performance. In his tests the hardware accelerated WebGL technology for 3D graphics performs best. Flash and Canvas have similar test results. Johnson et. al [8] compare different data visualizations with the help of Java applets, HTML5 Canvas, *Scalable Vector Graphics*[6] (SVG) and HTML elements. Java based rendering delivers best results in all measured test cases. Canvas performs better than SVG when there is a large amount of data to be displayed. They show that pure HTML rendering is not suitable for larger data.

Relying on web standards implemented in most modern browsers, SVG and Canvas represent good approaches for rendering. WebGL is not part of the HTML5 specification and not available in all browsers. Additionally, there is no W3C WebGL recommendation. For these reasons, WebGL is not an option for our implementation. The mentioned related work focuses on just one aspect of real-time groupware systems, either networking or rendering. The goal of our approach is to combine these two components in a single prototypical implementation in order to give information about how they can be integrated in one application.

IV. Tele-Board in the web browser - implementation and evaluation

We developed an HTML5 standards based prototype which resembles the functionality of the above mentioned Java based Tele-Board whiteboard client. We identified three main aspects that had to be transferred from the Java application to a browser based solution:

- Networking - bi-directional communication with the Tele-Board server component keeps all connected whiteboard clients synchronized
- Rendering - a virtual whiteboard surface displays whiteboard content such as drawings, sticky notes and images
- Threading - networking and rendering have to run in parallel, i.e. a threading mechanism prevents interruption of user interaction when whiteboard content is synchronized simultaneously

A. Networking with websockets

The traditional way of client-server communication in the World Wide Web is based on the Hypertext Transfer Protocol (HTTP) and its underlying request-response paradigm. Even more sophisticated methods such as AJAX and *HTTP server push*, that allow a more flexible communication handling, follow this pattern. In relation to the size of the message data to be transmitted, these techniques produce a considerable data overhead caused by HTTP headers [9].

A web technology facilitating a *persistent*, *bi-directional* Transmission Control Protocol (TCP) based communication is called Websocket. Websocket came up in the course of HTML5 and has been standardized by the Internet Engineering Task Force (IETF). The technology reduces the message overhead to just two bytes per frame [10]. Websocket functionality is implemented in many browsers[7] and can be accessed by an API from within JavaScript.

Building on the results and recommendations from Gutwin et al. [6] we use Websockets as the communication technology

[3]http://www.w3.org/TR/websockets/
[4]http://www.w3.org/TR/html51/
[5]http://www.khronos.org/webgl/
[6]http://www.w3.org/Graphics/SVG/
[7]http://caniuse.com/

for our prototype implementation. This way, we implemented a JavaScript Websocket client component. Whenever an element (e.g. sticky note or drawing) is created, moved or deleted on the whiteboard surface a corresponding message is sent to the server. Prior to that, the respective object and its properties are serialized using JavaScript Object Notation (JSON).

Since we use Websocket for client-server communication, a corresponding web server which supports the Websocket protocol is required. Currently, the synchronization server is implemented using Java. We therefore decided to use the Java based Jetty web server[8] in version 8.1.2. This way, we implemented a prototypical communication server running on the Jetty web server while preserving the existing code base.

B. HTML5-Canvas vs. Inline-SVG

As a first step, we implemented a subset of the whiteboard client's features in our prototype in order to evaluate available techniques. It supports drawing, creation, and arrangement of sticky notes as well as a zooming and panning capability. HTML5 offers several technologies for client-side rendering. The specification allows the usage of SVG as part of an HTML document, which is often called *inline SVG*. Another standards based technology is HTML5 canvas. Canvas represents a rectangular area where graphics can be drawn onto. Both technologies can be accessed by a JavaScript API in the browser. The ways of using these graphics APIs differ fundamentally. For comparing the different rendering techniques, we implemented two versions in our prototype utilizing the respective API.

1) SVG-Rendering: SVG is based on XML and was developed for decribing 2D vector graphics. Transformations such as scaling can be done without image quality loss. SVG is a *retained-mode* graphics model that holds an internal model with all rendering objects. When calling the API, the internal model is updated. The graphics library initiates respective drawing commands, i.e. the actual drawing is done by the library. Every SVG element (e.g. rectangle, line or text) has a representation in the Document Object Model (DOM) tree and can therefore handle user input on its own. Furthermore, Cascading Style Sheets (CSS) can be used to style SVG elements which makes it very convenient to set appearance attributes such as color, gradients or shadows.

The virtual whiteboard surface has a zooming and panning functionality. In our SVG rendering version we use the `svg` element's `viewBox`[9] attribute for scaling and panning. All other whiteboard elements such as sticky notes and drawings are represented by SVG's `rect` and `path` elements. For whiteboard element transformation such as translation and scaling we use the `transform` attribute of the respective SVG element. For example, if there is a drawing on a sticky note (see listing 1) the respective `rect` and `path` elements are grouped with the help of the SVG `g` element. The `g` element defines a transformation matrix. Changing the matrix' data, included elements can be scaled and translated together. Therefore, the path's coordinates are stored relative to the `rect` element.

```
<g transform="matrix(1 0 0 1 1052.67 1365.8)">
    <rect id="0:8" height="60" y="0" x="0" class="
        sticky green" width="90"></rect>
    <path id="0:10" class="scribble black" d="M
        10.32 39.70 ... L 70.32 43.70"></path>
</g>
```

Listing 1. SVG representation of a sticky note with a drawing on it. The `g` element is used for grouping the included elements. Both elements can be easily scaled and moved by updating `g` element's transformation matrix

Since the virtual whiteboard surface is typically larger than user's screen resolution, the position on the screen where a user clicks on a whiteboard element does not necessarily match the position where the element is placed on the whiteboard surface. To obtain the corresponding whiteboard surface position, the SVG API offers methods (e.g. `matrixTransform`) for translating different coordinate spaces, which is needed for setting the position and scaling in the whiteboard coordinate space. Within the application, no further programming effort is required for rendering. The actual drawing is initiated by the SVG API. This makes its usage rather convenient compared to the Canvas based method.

2) Canvas-Rendering: In contrast to SVG, Canvas is pixel based, i.e. a Canvas represents basically an image where graphics primitives such as rectangles, lines or polygons can be drawn onto. In addition, Canvas uses an *immediate-mode* API. Immediate-mode rendering means that the graphics library does not store any internal model of the objects to be drawn. The model has to be saved inside the application. Furthermore, the application controls when to draw the scene and what area needs to be redrawn when the internal model has changed. Graphical objects drawn on a canvas surface cannot individually handle user input such as click events. Instead, the position on the canvas surface has to be used to find the respective element manually, based on the internal model of the application. This way, each internal object has to be tested whether it contains the given position. Though this rendering method requires more implementation effort, it also provides a large amount of flexibility.

In order to offer the same zooming and panning functionality as the SVG version, our Canvas based rendering implementation is using two canvas elements. The first canvas is declared in the document markup. When the page is loaded the canvas' `width` and `height` attributes are set to the screen dimension. The second canvas element is created dynamically in the memory but is not appended to the document markup. It serves as an image buffer. The size of this second element is set to the virtual whiteboard size which can be larger than screen resolution. All whiteboard elements are drawn on the second canvas element. Afterwards, the whole image content of this background canvas is drawn onto the front element. The API function `drawImage` can use a canvas element as an input parameter, i.e. the function just renders the whole content of the background canvas as one single image. In particular, for zooming and panning this is very

[8]http://jetty.codehaus.org/jetty/

[9]http://www.w3.org/TR/SVG/coords.html#ViewBoxAttribute

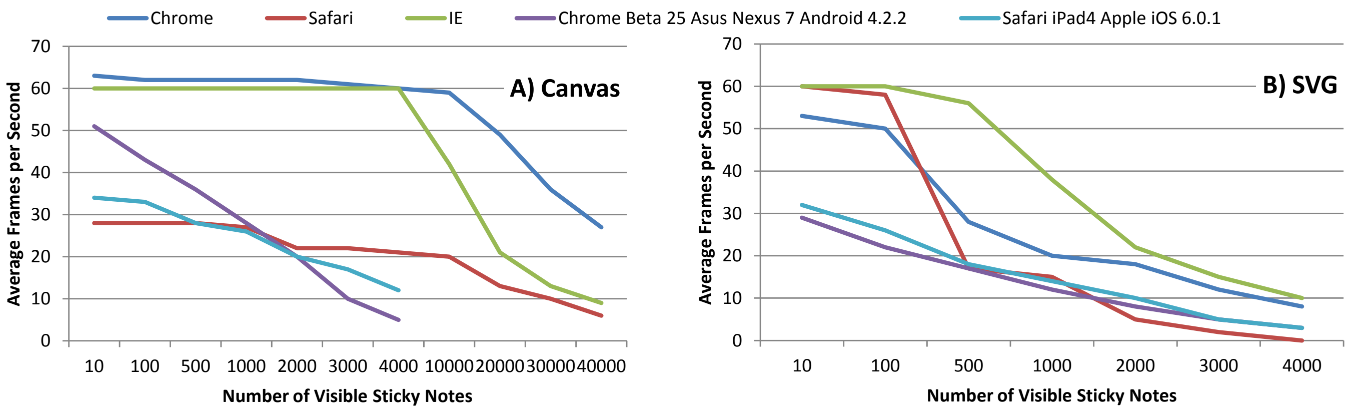

Fig. 2. Canvas and SVG based rendering performance. Rendering and whiteboard content synchronization is executed simultaneously using web workers

efficient since the rendering of all whiteboard content can be done in a constant time period, no matter how many elements are contained. These elements are already pre-rendered in the background canvas. Before rendering the background canvas, a transformation matrix can be defined in the front canvas. This way, zooming and panning is realized. The canvas element's position and scaling is defined by transformation matrices similar to the approach we use in our SVG implementation. For the respective matrix calculations we use the SVG API in our approach. The API defines methods to efficiently process matrix multiplications for translation and scaling[10].

In immediate-mode rendering, the application is responsible for redrawing a scene. When moving a sticky note on the whiteboard surface, the scene has to be updated. One approach could be to redraw the whole scene, i.e. to redraw every element on the surface. However, the more elements exist on the surface, the slower the rendering works. Therefore, we had to find a way to reduce redrawing costs. When an element is updated, e.g. changes its position, only the affected area has to be redrawn which includes the element itself and all elements overlapping this region. Our implementation makes use of the *R-tree* data structure where data items are stored according to their spatial location [11]. R-trees can be searched very efficiently. On the basis of given coordinates an R-tree's search algorithm delivers all data items that overlap with the respective area. We use an open source JavaScript R-tree implementation[11]. This way, we can find those objects that have to be redrawn. In the next step we define a *clipping region* with the help of Canvas API `clip` method. Afterwards, all drawing commands only affect the given area. We calculate the clipping region as the minimum bounding box of an elements prior and actual dimension. Providing this approach, the rendering runs much more efficient.

The effort using Canvas API for rendering is higher than in the SVG version. However, Canvas based rendering performs better as we will show in section IV-D.

[10]http://www.w3.org/TR/SVG/coords.html#InterfaceSVGMatrix
[11]https://raw.github.com/imbcmdth/RTree/master/src/rtree.js

C. *Multithreading - HTML5 Web workers*

Most JavaScript implementations in today's browsers are single threaded [12][13]. The computation in this thread includes tasks such as user input handling, repainting or animating [14]. As a result, scripts that run for a long time can block processing of other tasks so that the respective website becomes unresponsive. With the help of the two `window` object methods `setTimeout` and `setInterval` one can try to simulate a parallel execution of tasks. Following this approach, the execution of the respective method is actually just enqueued by the browser. These execution requests are processed sequentially in the main thread. Thus, long running code inside such method calls still compromises website's responsiveness.

As a solution for this problem and to leverage the computational power of today's multi core processors, the new *Web worker* API for parallel task execution has been added to modern browsers with HTML5. Web workers run in separate threads. Since they have a high memory consumption and their creation is expensive in terms of processor load [15], they are not well suited for short running, fine grained computational tasks. A major difference between Web workers and the threading model in other programming languages (e.g. Java) is that there is no concept of shared memory. The only way of communicating between worker and main thread is provided by message interchange based on an event model where the passed data is copied. This implies that Web workers are not allowed to access the DOM [15] (notable exceptions are `setTimeout`, `setInterval`, and `XMLHttpRequest`). Most native browser properties are not accessible. Therefore, some tasks to be processed in a web worker are either not possible (e.g. file operations) or must be built almost exclusively on own JavaScript code or appropriate libraries (e.g. XML data processing). These limitations exclude related challenges such as data synchronization, locks or race conditions. However, this new concept of having real threads within web browsers is strictly limited in its usefulness by the lacking resource accessibility.

A suitable use case for Web workers is the processing of background I/O as it is examplary mentioned in the Web workers specification [15]. The synchronization of whiteboard content and therefore required network communication is such a case. Our implementation requires the processing of a large number of messages to be sent to the server. For example, in order to allow a traceable arranging of sticky notes on the whiteboard surface, a smooth movement animation is required instead of just setting its final position after moving. The message data is passed from the main thread to the worker thread and is added to a queue which is constantly processed inside the worker thread.

D. Evaluation

In the following, we describe how the different rendering and threading approaches influence the graphical performance according to the amount of whiteboard elements that have to be rendered and synchronized to the server. We do not concentrate on network performance measurement since Gutwin et. al [6] extensively tested this aspect. We focus on rendering performance while taking network communication into account as well. Therefore, we measure graphics performance in terms of frames per second (FPS) on different platforms. The impression of image movement appears around 20 FPS [16]. The evaluation goal is to identify to what amount of data our prototype allows a convenient user interaction.

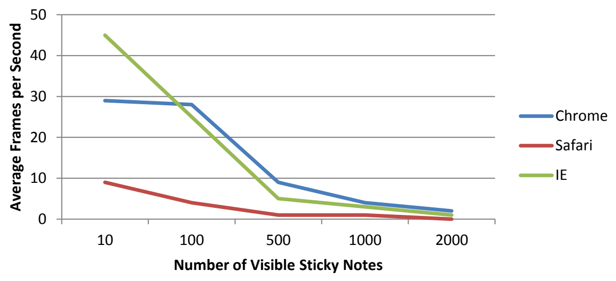

Fig. 3. Canvas based rendering performance with whiteboard content synchronization. No rendering optimizations

1) Test setup: The most common operating systems (Windows, Android, iOS) on desktop and mobile computers are used for our test setup. The evaluation consists of graphics performance tests of our browser based prototypical whiteboard client implementation. A predefined number of sticky notes is drawn randomly on our virtual whiteboard surface in different browsers. We measure FPS while one sticky note is moved a predefined distance over the screen. The sticky movement causes concurrent rendering and network load, since every movement initiates a corresponding network message. On the desktop system we use the JavaScript library *stats.js*[12] revision 11 for measuring FPS in the web browsers *Internet Explorer 10 (IE) Platform Preview*, *Chrome 23*, *Safari 5.1* and *Firefox 17*. Tests are run in full screen mode with a screen resolution of 1920x1080 pixels on a Windows PC with the following configuration: CPU: Corei5 750@2.66 GHz, RAM: 6GB, Graphics: Nvidia GeForce 310, OS: Windows 7 Enterprise SP1 x64. Futhermore, the two mobile devices Apple iPad 4 (Safari, iOS 6.0.1) and Asus Nexus 7 (Chrome Beta 25, Android 4.2.2) are used for testing. On these systems, vendor based frame profiling tools were used.

The following two scenarios will reveal insights regarding the identified main aspects mentioned in section IV with focus on the combination of rendering and networking. The first test scenario evaluates the different approaches of Canvas based rendering described in section IV-B2 and SVG.

The second scenario looks into browser based threading on the basis of HTML5 Web workers compared to the traditional way of "simulating" multithreading using the `setTimeout` method. Though it is officialy claimed that the Firefox[13] web browser supports Web workers and Websockets [15][17], we determined that Websockets are not available inside Web workers. Therefore, only in this scenario tests were additionally run in the Firefox browser.

2) Scenario 1 - SVG and Canvas rendering optimization capabilities: In our implementation based on the Canvas API, we implemented a mechanism for optimizing rendering performance regarding the region that has to be redrawn when an element changes. Looking at figure 3 it can be seen that frame rates are falling rapidly to a level less than 10 FPS when there are 500 or more elements on the surface. In this plain approach all elements are redrawn regardless of the affected area. This is the case if you do not think about any optimizations using the Canvas API, which performs worse than SVG based implementation (see Figure 2b). Compared to that, Canvas based rendering heavily profits from the optimization we implemented by setting a clipping region with the help of the R-tree data structure (see section IV-B2) as it is shown in Figure 2a. Even when there are 10,000 elements on the surface, Internet Explorer 10 and the webkit based browsers Chrome and Safari run at high frame rates around 50 FPS. In the SVG version, on average just 50% of the frame rates are reached when there are only 2,000 elements. On mobile systems, the behavior is similar. Though the frame rates are lower than on the desktop system, which is most likely caused by the generally weaker mobile hardware, it can be seen that canvas based rendering is nearly twice as fast as the SVG version. Obviously it is worthwhile to invest some effort on implementing mechanisms that leverage the Canvas API's rendering flexibility.

3) Scenario 2 - Parallel tasks using web workers or setTimeout method: The actual scenario focuses on the question whether our prototype implementation can benefit from utilising Web workers instead of the traditional way of avoiding user interface blocking based on the `setTimeout` method. Figures 4 and 2a reveal an advantage for the web worker approach, when there is a very large number of elements on the surface. Since we focus on a realistic usage, a mainly constant network load is generated, which explains the little difference between both techniques. However, when there

[12]https://github.com/mrdoob/stats.js

[13]http://www.mozilla.org/en-US/firefox/fx/

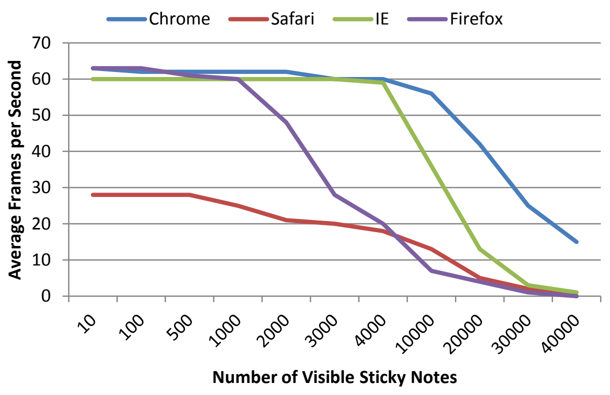

Fig. 4. Performance of optimized Canvas based rendering with whiteboard content synchronization using traditional setTimeout method

are larger messages to be processed, we expect to see a bigger difference between our implemented solutions. The performance bottleneck is therefore the rendering of large amounts of elements rather than their synchronization. That is the reason for only showing the results for the desktop browsers. On mobile systems, FPS are decreasing much faster before synchronization has any effect.

V. Conclusion and Outlook

We present a graphical real-time groupware software prototype based on modern standard browser technology. The system combines technical aspects for rendering and networking in a single application that runs on desktop and mobile hardware as well. Through performance measurements it became clear that computational power on mobile devices is still too low for using our application on a satisfactory level. However, due to more powerful devices, we expect notable performance improvements within the next year. On desktop computers, performance is already sufficient for convenient usage. The good performance results from optimizations we did on the basis of HTML5 canvas, leveraging its rendering flexibility. We propose an approach for redrawing user affected regions instead of redrawing the whole canvas during user interaction. This method relies on the R-tree data structure that allows to efficiently search in spatial data. For comparing different rendering technologies we implemented an SVG based version which in particular performs equal to canvas when there are no canvas rendering optimizations. Although there is less programming effort, SVG proved that it is not suitable for high interaction with large amount of data within our system. Canvas based rendering is therefore the most promising approach.

In order to facilitate effective network communication for synchronizing remote collaboration clients we implemented an HTML5 Websocket based networking. With regard to parallel processing in the browser, we compared the traditional way of simulating parallel tasks with the help of the `setTimeout` method to multithreading offered by the Web worker API. Though there are strict conceptual limitations, usage of the Web worker API has an advantage when the respective task consumes much computational power.

Future tests have to reveal more precisely at which processing load Web workers significantly improve the application's performance. Additionally, we want to investigate whether caching whiteboard content on temporary canvas elements in memory can help further enhancing Canvas based rendering performance. We also have to analyze the effect of increased memory consumption when applying this approach.

Acknowledgment

The authors would like to thank the HPI-Stanford Design Thinking Research Program.

References

[1] M. Anttonen and A. Salminen, "Transforming the web into a real application platform: new technologies, emerging trends and missing pieces," *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 800–807, 2011.

[2] T. H. McMullen, K. A. Hawick, V. D. Preez, and B. Pearce, "Graphics on web platforms for complex systems modelling and simulation," in *Proc. International Conference on Computer Graphics and Virtual Reality (CGVR'12)*. Las Vegas, USA: WorldComp, 16-19 July 2012, pp. 83–89.

[3] A. Wright, "Ready for a Web OS?" *Communications of the ACM*, vol. 52, no. 12, p. 16, Dec. 2009.

[4] D. Winokur, "Flash to Focus on PC Browsing and Mobile Apps; Adobe to More Aggressively Contribute to HTML5," http://blogs.adobe.com/conversations/2011/11/flash-focus.html/, retrieved November 26, 2012.

[5] R. Gumienny, L. Gericke, M. Quasthoff, C. Willems, and C. Meinel, "Tele-board: Enabling efficient collaboration in digital design spaces," in *Proc. of the 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2011)*. Lausanne, Switzerland: IEEE Press, 6 2011, pp. 47–54.

[6] C. A. Gutwin, M. Lippold, and T. C. N. Graham, "Real-time groupware in the browser: testing the performance of web-based networking," in *Proc. of the ACM 2011 conference on Computer supported cooperative work*, ser. CSCW '11. New York, NY, USA: ACM, 2011, pp. 167–176.

[7] R. Hoetzlein, "Graphics Performance in Rich Internet Applications," *Computer Graphics and Applications, IEEE*, vol. 32, pp. 98–104, 2012.

[8] D. W. Johnson and T. J. Jankun-Kelly, "A scalability study of web-native information visualization," *Proc. of graphics interface 2008*, pp. 163–168, May 2008.

[9] P. Lubbers and F. Greco, "HTML5 Web Sockets: A Quantum Leap in Scalability for the Web," http://www.websocket.org/quantum.html, retrieved November 27, 2012.

[10] Internet Engineering Task Force (IETF), "RFC 6455: The WebSocket Protocol," http://tools.ietf.org/html/rfc6455/, retrieved November 27, 2012.

[11] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*. ACM, 1984, pp. 47–57.

[12] J. Edwards, "Multi-threading in JavaScript," http://www.sitepoint.com/multi-threading-javascript/, 10/24/2008.

[13] L. Wagner, "JSRuntime is now officially single-threaded," http://blog.mozilla.org/luke/2012/01/24/jsruntime-is-now-officially-single-threaded/, 01/24/2012.

[14] Rousset, David, "Introduction to HTML5 Web Workers: The JavaScript Multi-threading Approach," http://blogs.msdn.com/b/davrous/archive/2011/07/15/introduction-to-the-html5-web-workers-the-javascript-multithreading-approach.aspx, 07/15/2011.

[15] W3C, "Web Workers W3C Candidate Recommendation 01 May 2012," http://www.w3.org/TR/workers/, retrieved November 28, 2012.

[16] S. K. Card, A. Newell, and T. P. Moran, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983.

[17] Mozilla Foundation, "Firefox - Built on Open Technology - Powerful new innovations that change the way you build the Web," http://www.mozilla.org/en-US/firefox/technology/, retrieved November 29, 2012.