A Web Application Widget Library for Scalable Interactive Biological Data Visualization*

Terje André Johansen, Daniel Stødle, Lars Ailo Bongo
Department of Computer Science, University of Tromsø, Norway
terje.andre.johansen@gmail.com, daniels@cs.uit.no, larsab@cs.uit.no

Abstract

The life sciences are facing extreme scales of data due to the rapid technology development of scientific, storage, and computer technologies. Integrated analysis of such data has great potential for novel biological insight. Such analysis requires visualization tools that meet the following five requirements: (i) scalability to millions of datasets, (ii) interactive performance, (iii) ease of implementation, (iv) portability to all common platforms, and (v) single-click installation. To our knowledge, no current biological data visualization tools meet all five requirements. We present a web application widget library that satisfies all of the above requirements. It provides widgets that implement efficient communication and rendering to make it easy to build scalable interactive visualizations. For our experimental evaluation we have used the widgets to build a graphical user interface for the Spell data exploration system. We have demonstrated that by utilizing new HTML5 features our widgets provide interactive scalable visualizations for analysis of a genomics compendium with 400.000.000 individual samples. Our main conclusion is that web applications are the best choice for implementing visualization tools for biological data due to their portability and ease-of-installation. Our widget library will be an important contribution for building these tools.

1. Introduction

Many scientific disciplines are facing extreme scales of data due to the rapid technology development of scientific, storage, and computer technologies [11]. A key research challenge in sciences such as biology is therefore to explore, analyze, and interpret all the data [26]. Many supervised and unsupervised biological data analysis techniques have been developed [4, 13], and the majority of these techniques share a common need for visual, interactive evaluation of results to examine important patterns, explore interesting genes, or consider key predictions and their biological context.

Such data analysis requires building graphical user interfaces for interactive visualization of a particular analysis method, data type, or application domain [20]. With the rapid increase of data, these tools must visualize increasingly larger data sets, and hence the implementation of the tools becomes increasingly challenging. To make the development of novel data analysis methods efficient, there is a need for easy-to-use building blocks that genomics analysis model developers can use to rapidly build an interactive visualization tool for the new methods they develop.

We believe that a framework providing building blocks for interactive biology data visualization should satisfy the following requirements. First, the visualization should scale to the millions of genomics datasets that are predicted to be produced in the near future. For example, it is likely that it will become common practice to sequence the genes of each of the 27.500 people in Norway that get cancer every year [10]. Second, the analysis must be interactive as argued above. Third, it should be easy to implement new tools using the framework such that method developers can focus on providing novel biological insight rather than performance tuning of their tools. Fourth, the developed tools should be portable to all popular platforms including Windows, Mac OS X, Linux, and mobile phones. Fifth, the developed tools should be easy to install,

* This student paper is based on the master thesis of Terje André Johansen [15].

This paper was presented at the NIK 2011 conference. For more information see http://www.nik.no/

since biology end-users are not likely to use tools with complicated installation procedures. In addition, the framework should support integrated visualizations since it is common in biology to analyze multiple data types at multiple levels simultaneously [6]. Finally, but still important, the tools should have a nice look and feel in order for end-users to use them.

Existing frameworks for building interactive visualization tools do not satisfy all of the above requirements ([7] provides a comprehensive list of available visualization tools). Biological visualization tools are often desktop applications implemented in Java for increased portability [12, 24, 25]. An emerging trend in biology and other application domains is to replace desktop applications with web applications that are encoded in JavaScript, downloaded when a user visits a web site, and run in a web browser. These solve the portability and installation problems of desktop applications. In addition they offer software maintenance benefits since updates to application code or dataset content will be available the next time a user visits the web page and starts the application. However, web applications have limitations in that they are encoded in a high-level language, have very limited access to the file system and graphical hardware accelerators, and in addition most browsers only support single-threaded web applications making it hard to exploit modern multi-core processors. It has therefore been necessary to use desktop applications to visualize very large biological datasets that are computationally intensive to analyze and render.

Due to the above described advantages, a lot of effort has been put into making browsers and in particular JavaScript faster. In addition the emerging HTML5 standard [9] reduces many of the performance limitations of current browsers as demonstrated by the recently, implemented computationally intensive demo applications such as the Mandelbrot [18], Quake [21], and AngryBirds [1]. Although, these demonstrate that web applications support computably intensive workloads, they do not require accessing as large datasets as biological applications do. Web applications such as Google Maps are capable of exploring very large datasets by pre-rendering the maps to be displayed and storing these on the server such that the client only needs to display retrieved images [2]. Biological visualizations differ in that the visualizations often depend on user input, such as queries or algorithm parameters, and must therefore be calculated and rendered in real-time.

This paper describes the design and implementation of a web application widget library that satisfies the above requirements. It contains scalable and useful visualization widgets for presenting biological data. These can be used as building blocks to easily develop web applications without handling performance and scalability issues. The widgets can also be integrated using a genomics data aware event bus.

Our contributions are twofold. First we present design principles and a reference implementation of scalable interactive widgets for genomics data analysis web applications. Second, we provide a performance evaluation of widgets implemented with and without HTML5 features.

We have demonstrated that with the new HTML5 features supported by web applications, widgets can be implemented that provide the required scalability and interactive performance for building visualizations for analysis of a 400.000.000 value compendium. Our main conclusion is therefore that web applications are currently the best choice for implementing visualization tools for biological data due to their portability, ease-of-installation, and ease of code and data update. Our widget library will be an important contribution for building web applications for novel biological data analysis models.

2. Widget Library Overview

We have built a widget library that consists of reusable, scalable, integrated components. Applications built using our widgets will use the typical three-tier architecture often used by web applications (Figure 1). The client is run in the web browser on the user's computer. The client sends asynchronous HTTP requests to a Java servlet in a web server or container, which uses search and persistent data storage services run on a backend cluster. The user interface is run in a browser, and consists of HTML, JavaScript and CSS.

The widgets consist of the three layer structure commonly used by web applications. The first, process layer, implements the business logic of the application and is split between the client and the server. The second, data layer, is initially only on the server, but as the client caches data some of it will be moved over to save bandwidth and improve performance. Finally, the interface layer implements the user interface and is only on the client.

The widgets are integrated using an event bus that the widgets use to subscribe to and broadcast high-level events. A widget can therefore implement an event handler that is called when for example a gene is selected in another widget. We also provide a simple window manager for organizing the layout of integrated widgets.

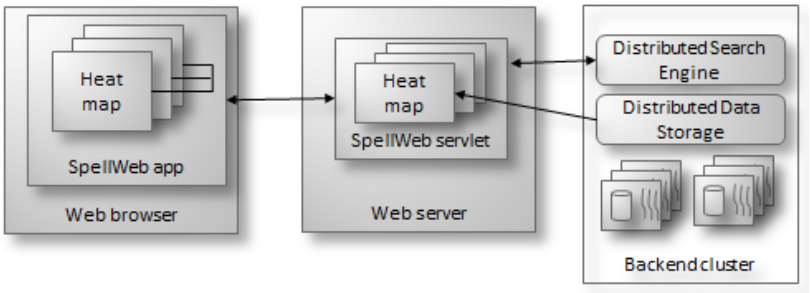

Figure 1: Architecture for an application using the widget library (SpellWeb; described in Section 4.1).

Currently the widget library consists of two different heat map implementations and a dendrogram implementation. The event bus supports two types of events: *geneclick* and *datasetclick*. We have used the widget library to implement a web application for the Spell genomics data exploration algorithm. The library is designed to be easily extended with new widgets or events.

2.1 Google Web Toolkit (GWT)

The widget library implements widgets for Google Web Toolkit (GWT) [8]. GWT is a popular toolkit for developing web applications. Both the server-side servlet and the client-side user interface are implemented in Java, but the client-side Java code is compiled to JavaScript code targeted for different browsers. For the implementation presented in this paper we used version 2.3 which supports some HTML5 features [5].

There are three main reasons why we selected GWT. First, the code is portable, since GWT is already used by major Google applications that run on most devices with a JavaScript capable browser. Second, GWT has good interactive performance since the user interaction code is run in the browser. Finally, the code is easy to understand and extend for developers since everything is programmed in Java, which is the language used for the other systems in our genomics data exploration infrastructure.

2.2 Widget integration

Biological data analysis often requires integrating several visualizations. For example it may be of interest to compare a view of similarity search results with clustering results. In such a case it is necessary to locate the same genes in the different visualizations, by for example highlighting these.

We have implemented such integration using an event bus that supports two genomics data specific events; gene click and dataset click. In our experience these two low-level events provide the right abstraction level for implementing the needed integration in our applications. For example, “select a set of genes” can be implemented by sending one gene-click event for each of the selected genes. The event bus does not maintain any global state, such that additional widgets can easily be added without modifying the event bus.

3. Heat map widget

Heat maps (Figure 2) are one of the most frequently used biological visualizations, especially for analyzing expression data from microarray or next-generation sequencing instruments. Expert users explore the heat map visualizations to detect and investigate interesting patterns indicating a biological signal in the data. For example, to find a set of candidate cancer genes for an expensive and time consuming wet-lab experiment, a genomics researcher may use a data exploration system that ranks the genes and datasets according to their functional similarity to previously known cancer genes.

In order to identify the biological signal en expert must find an ordering of genes and datasets such that significant patterns emerge. Such an ordering is typically done using various clustering algorithms [4] or ranking results obtained using similarity search algorithms [13]. The resulting visualizations can be very large. For example, humans have about 25.000 genes and there are already ten thousands of datasets with tens to hundreds of samples. Furthermore, the number of datasets is rapidly increasing.

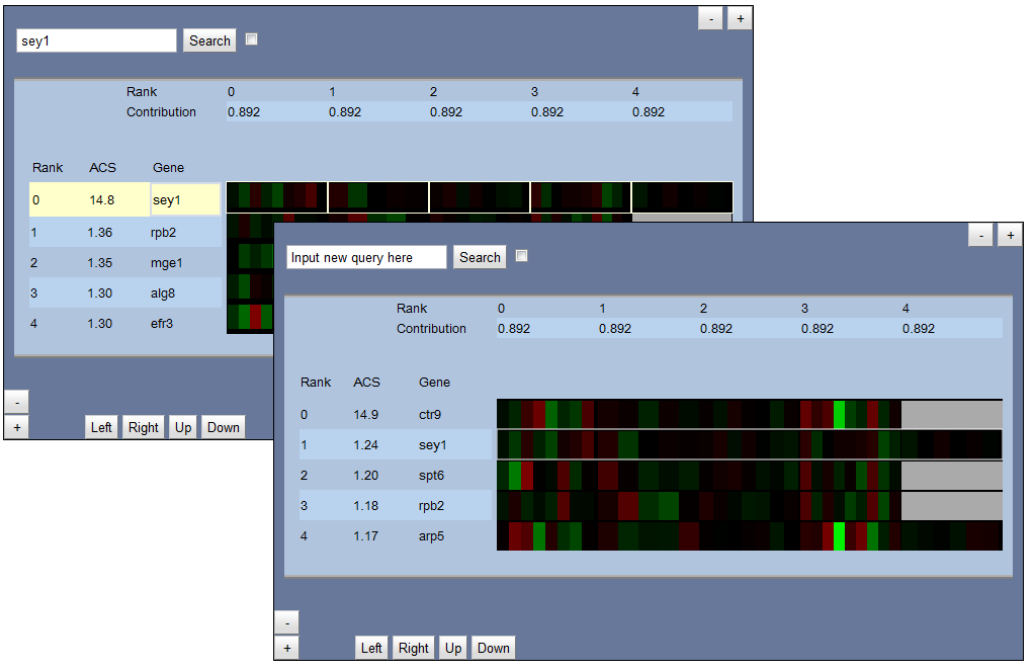

Figure 2: Two integrated heat map visualization widgets used by the SpellWeb application (described in Section 4.1). In biology a heat map is used to visualize gene expressions as measured by for example microarray instruments. The genes are placed in rows, while different datasets are in the columns. The

columns are further split into cells that represents expression values measured in different experiments for the particular gene. An expression value is a single color, green or red. The intensity of the color shows how expressed (green) or non-expressed (red) a gene is for a particular experiment. The grey cells are datasets which does not include measurements for that particular gene. An expert user can identify patterns indicating a biological signal such as indications for a disease using heat maps.

3.1 Communication protocol

The ordering of genes and datasets in heat map visualizations are typically determined by a similarity search or clustering algorithm. The result of these algorithms depends on a set of query genes that are selected by the user at query time. The resulting heat map visualizations can therefore not use an approach similar to Google Maps [3], where a scalable visualization is implemented by combining pre-computed pictures cached on the server side. In addition, desktop applications that use heat maps typically support interaction in the form of highlighting selected genes, color scheme changes, and hiding/showing datasets. Such operations are most efficiently implemented on the client side. For these two reasons, the heat map widget implements the following communication protocol to generate and display visualizations at query-time:

1. The client-side widget sends an application specific query request to the server that may contain a set of query gene names and query datasets IDs.
2. The server-side servlet receives and parses the query, then requests a ranking (or ordering) of the genes and datasets from a backend server. This server can for example be a distributed search engine, a database, or a clustering algorithm.
3. The servlet receives an ordered list of genes and an ordered list of datasets from the backend service, which it forwards to the client.
4. The client uses the ordered list of genes and datasets to requests expression data and meta-data to create a visual representation as a heat map.

The expression values to visualize are split into multiple messages sent on-demand, due to the large size of the visualization. In addition to avoid an extra network roundtrip, the gene and dataset identifiers are piggybacked on the first data message.

3.2 Pagination and client-side caching

The heat map view is implemented using a pagination design pattern. The pagination mechanism uses caching and pre-fetching of data such that the data to be rendered can read from an in-memory data structure instead of being requested over a high-latency wide area network. The cache is used to save the data to be visualized. The pre-fetching policy assumes that the next page the user will view is in close proximity of the current page (either vertically or horizontally). Therefore the client can pre-fetch data for adjacent pages concurrently by rendering of the current page.

The cache implementation takes advantage of GWT provided classes but is also limited by restrictions imposed by these classes. The heat map was initially implemented using a GWT CellTable (as described below), which required using one of two list based data provider classes that provide the data for each row in the CellTable. To reduce the time for the frequently used gene and dataset ID lookups for data within a row, we used a hash table lookup. The hash tables also simplify the cache implementation since it is easy to identify duplicate entries by comparing their keys, and the GWT hash table implements a threshold that can be set to automatically evict old entries using a least-recently-used (LRU) policy. However, the cache and pagination implementation would further have been simplified if GWT provided a hash table based data provider class as described in Section 3.4.

3.3 HTML table implementation

The heat map is rendered using two different approaches. The first approach is to use the GWT "lightweight" CellTable, which creates a HTML table that is shown by the browser. The second approach, described in the next section, is to use a HTML5 canvas. These two represent respectively the traditional approach for implementing heat maps in web applications, and what we believe will be the future approach.

The advantage of the table based implementation is that it is supported by all browsers, the layout can be defined using CSS, and that it is easy to implement since the browser automatically resizes the table and individual cells to fit the size of a browser window. This implementation works well for small views with less than 50 genes and 50 datasets (50 rows, 50 columns, 10 cells per column). However, as the view increases in size, the performance for a HTML table decreases due to a large increase of objects in the browsers Document Object Model (DOM).

The CellTable has two additional problems. First, adding or removing columns requires re-drawing the entire table. Second, we cannot explicitly control its rendering. The browser will therefore not paint the table before all inner tables are created.

3.4 HTML5 canvas implementation

The second approach utilizes the HTML5 canvas which is supported in GWT2.3 to natively draw to a bitmap. This implementation reduces the number of DOM elements up to 80 percent which significantly increases performance.

The canvas relies on our pager to keep track of both row and column offset of the rows that are visible. Compared to the table implementation, we could not utilize GWT provided classes. But by implementing everything from scratch, we had detailed control over rendering, enabling us to support incremental rendering by using timeouts to render the heat-map piece by piece. The rendering is therefore smoother than for the table based implementation since the browser does not freeze while the JavaScript is running, and the user can see the table being rendered.

Double buffering is used to optimize resizing of the canvas, since any changes to the canvas width or height will reset the canvas content. We therefore save the previous view in a temporary canvas, and copy the old bitmap into the new canvas during resize. This improves performance since it is much quicker to copy a bitmap than to run the render method for the whole view over again. The disadvantage of double buffering is that the memory requirements are doubled, since both buffers remain in memory until the old buffer is garbage collected by the JavaScript engine.

4. Evaluation

In our experimental evaluation we determine the maximum visualization size that supports interactive performance, and compare the scalability and performance of the table based heat map with the HTML5 canvas heat map implementation.

4.1 SpellWeb application

We have implemented a web application user interface for the SPELL query-driven search engine for large gene expression microarray compendia. Given a set of query genes, the algorithm finds the datasets that are most informative for these genes and then within those datasets genes that are most similar to the query set [13].

SPELL has a web application for interactive data visualization called SpellWeb [23] that uses heat maps to visualize the search results. SpellWeb was built using the Ruby

on Rails [22] framework, with a Java backend to perform the searches. We have ported SpellWeb to GWT and used our widget library to implement the heat map visualization.

We have used a human dataset with 725 microarray datasets with 15.859 individual samples. The total size of the dataset is 934 MB, and there are 24.410 unique gene identifiers in the datasets. The datasets have therefore 387.118.190 expression values (included null values) which results in a 244.410x158.590 pixel bitmap assuming that a 10x10 pixel area is used to represent a single expression value.

Our experiments focus on the performance of the visualization so we did not use a SPELL search engine, but instead returned results for a predefined query (a Spell search for the two human genes BRCA1 and BRCA2). The expression values where stored in a data structure that fit into the main memory of the server machine.

4.2 Experiment setup and methodology

For the experiments we used a server in Princeton University, New Jersey, USA and a client computer at the University of Tromsø, Norway. The round-trip "ping" latency between these machines is about 133ms.

The client is a desktop computer with an Intel i5 2500K 3.33GHz quad core processor and 8GB main memory. It is running 64-bit Windows 7, and the experiments are run in Google Chrome (only available in 32-bit mode). We have monitored resource usage using the Windows 7 Performance monitor. The only applications on the client computer during the experiments were the Chrome browser and the monitoring tools.

The server has an Intel Xeon E5430 2.66GHz quad core processor with 32GB main memory. The server is running CentOS 5.3, and we use Apache Tomcat 6.0.32 as web-server. To our knowledge there were no other computationally or memory intensive processes running during the experiments.

To fully understand the performance of dynamic web pages it is not enough to just instrument the JavaScript code with timers since this does not measure the performance of the other browser tasks such as building the render tree or calculating the page layout. We therefore use Chrome Developer Tools [3] to inspect the DOM, get statistics about network communication, profile the JavaScript code and identify performance bottlenecks.

To clarify the result discussion we assume a visualization of one gene in one dataset requires 180x30 pixels. This is a good approximation to the actual size used for the average 18 expression values per dataset (standard deviation 22), but the actual size depends on the ranking of genes to datasets where datasets with many expression values tend to be ranked higher than datasets with fewer values.

4.3 Memory usage

The scalability of previous web applications were often limited by browser memory usage [16]. We therefore start the evaluation by answering the following question: *what is the maximum heat map size that can be displayed on current browsers?*

We find that the for the table heat map implementation the maximum view size is limited by the browser memory usage. Since Chrome is currently not available in 64-bit code it has only a 2GB user-level address space (default size in Windows, but it can be increased to 3GB). Hence, the heat-map size is limited to 300 genes by 300 datasets large which uses about 1.6GB of memory (Table 1).

We also find that for the canvas implementation the maximum size is limited by a (non-specified) maximum canvas size (Table 1). The table heat map can therefore only be 200 genes by 200 datasets large. The canvas implementation also uses less memory

than the table implementation, especially for larger visualizations. But, as described above double buffering may for a short period of time double the memory requirements.

We expect future browsers to be 64-bit and hence have a larger user-level address space, and we expect the maximum canvas size to increase. Hence, even large visualizations are likely to be supported in future browsers. But even a 200x200 view is about 36.000x6.000 pixels in size, which in our opinion is too large for a user to explore on an ordinary desktop or laptop monitor. We therefore believe that for a better user experience it is necessary to use pagination, aggregation, or scaling to reduce the amount of data shown at once.

Table 1: Memory usage (in MB) for different view sizes (in pixels).

View size	**900x150**	**4500x750**	**9000x1500**	**18000x3000**	**36000x6000**	**54000x9000**
Canvas	28	40	77	191	582	-
Table	24	41	87	257	772	1578

Table 2: Render time (in milliseconds) for heat map visualizations using different view sizes.

View size	**900x150**	**4.500x750**	**9.000x1.500**	**18.000x3.000**	**36.000x6.000**
Canvas	35	192	575	2371	7500
Table	41	410	1254	5182	18143

Table 3: JavaScript heap inspection for a 36.000x6.000 pixel view.

	Count			**Size (MB)**		
	Code	Objects	Total	Code	Objects	Total
Canvas	7885	1.110.319	1.118.204	1,84	38,24	40,07
Table	7887	5.459.826	5.467.713	1,86	121,93	123,79

Table 4: Render time (in milliseconds) breakdown for table based heat map widget implementation.

View size	**900x150**	**4.500x750**	**9.000x1.500**	**18.000x3.000**	**36.000x6.000**
Parsing	1	171	567	2333	8350
Layout	2	45	127	546	1947
Total	41	410	1254	5182	18143

4.4 Render time

In the previous section we demonstrated that web browsers are capable of displaying very large visualizations, but we did not take into account the time to render and display the data. We therefore answer the question: *what is the maximum heat map visualization size that can be rendered with interactive performance?*

We define interactive performance to be results rendered in less than one second, which is the upper bound for the time users are willing to wait for a search query [19]. Our results show that for the canvas implementation we achieve a render time less than 5 seconds up to 18.000x3.000 pixels, and a render time less than one second for visualizations with 9.000x1.500 pixels (Table 2).

Our results also demonstrate that the canvas is about two times faster than the table implementation, and that the difference increases with larger visualizations. The main difference between the canvas and heat map implementations is the number of objects on the JavaScript heap (Table 3). The table has about 5 times more objects on the heap, due to the additional HTML tags in the DOM required to represent the tables used to draw the expression values for each gene-dataset pair (the canvas still has meta-data headers in tables which contribute significantly to the canvas DOM object count). The performance difference between the canvas and table implementations of the widget can therefore be attributed to the time the browser spends parsing the DOM tree and calculating the layout (Table 4). Previous works have shown that especially DOM tree

parsing is slow [27]. In comparison, the canvas implementation requires less than 10 milliseconds for parsing and less than 1 millisecond for layout even for 36.000x6.000 pixel visualizations.

Rendering using the canvas should be easy to parallelize since different parts of the heat map can be rendered independently. We therefore expect linear speedup for a multi-threaded (web workers) implementation run on a multi-core processor, or a similar speedup when utilizing a massively parallel processor such as a GPU.

4.5 Communication overhead

In the previous section we demonstrated that the canvas implementation of the heat map widget can achieve interactive performance for 9.000x1.500 pixel views. In this section we answer the question: *what is the maximum block size that can be requested to still achieve interactive performance?*

To answer this question we measured the data transfer time between the client and server (on two different continents). The results show that the transfer time is less than one second for a 4.500x750 pixel view (Table 5). A breakdown of the transfer time shows that the achieved throughput increases with larger datasets due to the slow-start used in the TCP protocol. We also calculated the compression ratio achieved by the gzip compression method used by GWT to be about 5x for realistic view sizes (Table 5).

Communication performance can be improved by tuning the protocol for wide area networks by for example increasing the TCP window size, or by using a faster (such as LZO [17]) or more efficient (such as LZMA) compression algorithm. But currently the communication wait time can be overlapped with render time.

Table 5: Transfer time (in milliseconds), communication volumes (in KB), and compression ratio.

View size	**900x150**	**4.500x750**	**9.000x1.500**	**18.000x3.000**	**36.000x6.000**
Total transfer time	266	962	1697	3517	9433
Data size	20	333	1170	4900	16430
Compressed size	5	66	232	962	3180
Compression ratio	4.0	5.05	5.04	5.09	5.17

4.6 Incremental updates

In the previous sections we measured the time to display view of a fixed size. However, it is often of interest to incrementally increase the size of a view either due to user input, or as a latency hiding technique. In this section we answer the question: *can incremental increases of the heat map size be used to hide render overhead?*

To answer this question we start with a view size of 900x750 pixel view, and then add 5 rows at a time until the view size is 36.000x750 pixels, and then a similar experiment where we add columns. All data was pre-fetched to the client such that a wait did not occur any network wait time.

Our results show that the canvas is significantly faster (Figure 3). This is due to the double buffering used in the canvas implementation. Not included in Figure 3 is the time to re-parse the DOM structure and re-layout the table required after each change to the table based implementation. Including these increases the total time for all canvas increases from 4.3 seconds to 11.8 seconds, and the total time for all table widget updates from 8.5 seconds to 45.2 seconds. The results demonstrate that the rendering of the heat map can be started with a small heat map that renders in less than 100ms which is then gradually increased in size by adding rows and columns.

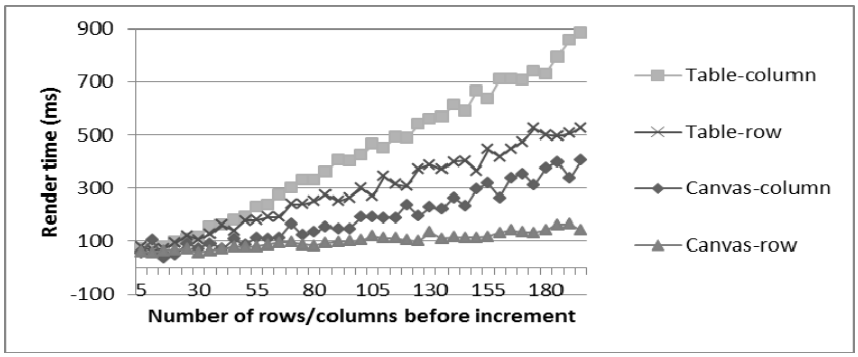

Figure 3: Render time for incremental addition of rows and columns for the table and canvas based heat map implementations.

5. Discussion

In this section we discuss how our results can be used to guide the design and implementation of scalable interactive biological data visualizations, the impact of emerging HTML5 features for such visualizations, and some challenges and possible solutions for large scale visualizations that cannot be paginated.

5.1 Scalability of interactive web application visualizations

We found visualizations with resolution similar to the size of current desktop screens to give smooth interactive performance (25 genes by 25 datasets which is about 4.500x750 pixels). This visualization can be rendered in less than 200 milliseconds, and the first block of data can be transferred in less than one second, and the subsequent data transfers can be overlapped with user think time. If the canvas size is increased to 18.000x3.000 pixels (100 genes by 100 datasets), it will take about 2.3 seconds to render and 3.5 seconds to transfer the data. But the user experience can be improved by incrementally updating the canvas and by pre-fetching the data.

5.2 HTML5 benefits

We have implemented the same visualization using HTML tables and using the new HTML5 canvas. We found the canvas to perform better, use less memory, and significantly reduce the number of objects on the browser DOM, and it offers greater control over the rendering. Another issue with the table-based heat map is the list-based data provider which makes it hard to share data between views and therefore increases the memory requirements of the cache for multi-view visualizations. A hash table based data provider would have made such sharing trivial to implement, especially with the automatic LRU element eviction provided by GWT hash tables.

We expect most modern browsers to support most HTML5 features within a few years and many be compiled in 64-bit mode, and hence: (i) user-level address space and main memory on the client problem will not be a major scalability limitation, (ii) parallel rendering using multi-threading (web workers) and GPU acceleration will reduce the render time by two orders of magnitude, and (iii) network bandwidth will improve by a few factors. The main challenge will therefore be to transfer the large datasets to be visualized from the server to the client computer. Efficient compression of network data, large client-side caches, and pre-fetching will therefore be important solutions.

5.3 Pagination alternatives

The heat map visualizations evaluated in this paper rely on pagination to reduce the memory size of the visualization and to pre-fetch data. For example the popular dendrogram visualizations used by desktop applications such as Java TreeView [25] and HIDRA [12] require displaying the dendrogram tree integrated with all data values in a dataset. We have implemented a prototype dendrogram for our widget library using the GWT canvas API. We found the large size of the resulting bitmaps to be the greatest challenge. In particular humans have about 25.000 genes, resulting in a bitmap with a height of 750.000 pixels when using standard size rectangles for expression values. This is larger than currently supported by the GWT canvas, and too large to be practical for a user to view. It is therefore necessary to scale the bitmap and provide zooming support in the visualization. Another challenge is to cache the data necessary. Since dendrograms are often used to view a one or a few datasets simultaneously we assume it should be possible to achieve interactive network transfer time and render time, but we have not validated this claim experimentally.

5.4 Alternative widget toolkits

There are several widget toolkits for building web applications. A popular toolkit for biological web applications is Ruby on Rails [22] where application are written in the Ruby programming language and where most of the application logic is run on the server. A tool similar to GWT is Microsoft's recently presented script# (script sharp) tool which compiles .NET to JavaScript [28]. To our knowledge, the toolkits provide similar scalability and performance, such that the choice of toolkit to use depends on the preferred programming language and supported developer tools. There are compilers that optimize JavaScript code [14] but the GWT compiler can analyze all code (HTML, CSS and JavaScript) which enables it to do optimizations JavaScript compilers cannot.

6. Conclusion and Future Work

We have presented a widget library for building scalable interactive biological web application visualizations. The library provides two implementations of a heat map visualization that we have used in our experimental evaluation, where we implemented a web application for visualizing the results of the Spell data exploration. Our results demonstrate that the widgets can visualize a compendium with 400.000.000 data points with interactive performance. Our main conclusion is therefore that web applications are currently the best choice for implementing visualization tools for biological data due to their portability, ease-of-installation, and ease of code and data update. Our widget library will be an important contribution for building web applications for novel biological data analysis models.

As future work we intend to add additional widgets to our widget library, and evaluate the dendrogram widget by porting the HIDRA [12] desktop application to GWT. We will also experiment with additional HTML5 features and to improve rendering performance. Finally, we predict network transfer time to be a bottleneck so we intend to implement genomics data specific compression for web applications.

Acknowledgments

Thanks to Olga Troyanskaya and Qiuan Zhu at Princeton University for their help regarding the requirements for the widgets. Kai Li for allowing us to use the cluster at Princeton, and Wei Dong and Zhe Wang for help using the cluster. Also thanks to Otto Anshus and the students at the display wall laboratory for discussions.

References

[1] AngryBirds for Chrome: 2011. http://chrome.angrybirds.com/.

[2] Chang, F. et al. 2008. BigTable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems. 26, 2 (Jun. 2008), 1-26.

[3] Chrome Developer Tools: http://code.google.com/chrome/devtools/docs/overview.html. Accessed: 2011-07-24.

[4] Datta, S. 2003. Comparisons and validation of statistical clustering techniques for microarray gene expression data. Bioinformatics. 19, 4 (Mar. 2003), 459-466.

[5] GWT + HTML5: A web developers dream!: http://www.google.com/events/io/2011/sessions/gwt-html5-a-web-developers-dream.html. Accessed: 2011-07-24.

[6] Gehlenborg, N. et al. 2010. Visualization of omics data for systems biology. Nature methods. 7, 3 Suppl (Mar. 2010), S56-68.

[7] Gene Ontology Tools: http://www.geneontology.org/GO.tools.shtml. Accessed: 2011-10-02.

[8] Google Web Toolkit: 2011. http://code.google.com/webtoolkit/. Accessed: 2011-07-25.

[9] HTML5 - A vocabulary and associated APIs for HTML and XHTML - W3C Working Draft 25 May 2011: 2011. http://www.w3.org/TR/html5/. Accessed: 2011-07-25.

[10] Haldorsen, T. 2011. Cancer in Norway 2009.

[11] Hey, A.J.G. et al. 2009. The fourth paradigm: data-intensive scientific discovery. Microsoft Research Redmond, WA.

[12] Hibbs, M. et al. 2007. Viewing the Larger Context of Genomic Data through Horizontal Integration. 2007 11th International Conference Information Visualization (IV '07) (Jul. 2007), 326-334.

[13] Hibbs, M.A. et al. 2007. Exploring the functional landscape of gene expression: directed search of large microarray compendia. Bioinformatics (Oxford, England). 23, 20 (Oct. 2007), 2692-9.

[14] Introducing Closure Tools: 2009. http://googlecode.blogspot.com/2009/11/introducing-closure-tools.html. Accessed: 2011-07-24.

[15] Johansen, T.A. 2011. A scalable, interactive widget library for visualizing biological data. Master thesis. University of Tromsø. http://www.ub.uit.no/munin/handle/10037/3523

[16] Johansen, T.A. 2010. Interactive Data Exploration of Very Large Biological Datasets using Web Applications. Project report. University of Tromsø.

[17] LZO real-time data compression library: http://www.oberhumer.com/opensource/lzo/. Accessed: 2011-07-24.

[18] Mandelbrot Set in HTML 5 Canvas: http://www.atopon.org/mandel/. Accessed: 2011-07-24.

[19] Nielsen, J. 1993. Usability Engineering. Morgan Kaufmann.

[20] O'Donoghue, S.I. et al. 2010. Visualizing biological data-now and in the future. Nature methods. 7, 3 Suppl (Mar. 2010), S2-4.

[21] Quake II GWT Port: 2011. http://code.google.com/p/quake2-gwt-port/. Accessed: 2011-07-15.

[22] Ruby on Rails: http://rubyonrails.org/. Accessed: 2011-07-24.

[23] SPELL- S. cerevisiae: http://imperio.princeton.edu:3000/yeast/. Accessed: 2011-07-24.

[24] Saeed, A.I. et al. 2003. TM4: a free, open-source system for microarray data management and analysis. BioTechniques. 34, 2 (Feb. 2003), 374-8.

[25] Saldanha, A.J. 2004. Java Treeview--extensible visualization of microarray data. Bioinformatics (Oxford, England). 20, 17 (Nov. 2004), 3246-8.

[26] Schuster, S.C. 2008. Next-generation sequencing transforms today's biology. Nature methods. 5, 1 (Jan. 2008), 16-8.

[27] Tak Lam Ding and J.J. Jyh-Charn Liu 2008. XML Document Parsing: Operational and Performance Characteristics. Computer. 41, 9 (2008), 8.

[28] script#: compiling c# to javascript using visual studio: http://channel9.msdn.com/events/MIX/MIX11/HTM16. Accessed: 2011-07-24.