Simple collaborative visualizations

Patrick Landis (pdlandis@ucsc.edu)

CMPS161 Final Project

03/20/2017

Abstract

Collaborative visualization tools have become increasingly easy to create and use as the related technologies improve and become popularized. While it becomes easier to develop these systems, the demand for intuitive data visualization and collaborative techniques is growing rapidly, as computer hardware enables the collection and storage of enormous amounts of data. This paper describes the technical details and development process of a simple collaborative tool to highlight some of the challenges and opportunities related to these circumstances.

1. Introduction

Collaborative visualization software is not a particularly new field of study. Data visualizations are generally intended to be shared between users in a way that assists an understanding of the data being viewed. While visualizations based on a single individual's understanding of a dataset, and the relevant features to be highlighted, can be incredibly useful, there are many applications in which the benefits of additional perspectives are clear. Medical visualizations, such as x-ray images, have specific areas of interest that can benefit from a collaborative environment where multiple doctors can quickly and efficiently identify these critical points, without significantly overlapping in work. Prior to collaborative medical imaging, these documents would be sent to each doctor individually, resulting in significantly slower communication and a delayed diagnosis [1]. There are now many software solutions to the specific problem of radiological collaboration [1][2][3], from small, academic-lead initiatives like radiollaboration [1], to fully-featured commercial products such as Terarecon's Overlay PACS™ [3]. For very similar reasons, collaboration software in an architectural, engineering and construction context has become commonplace, with solutions like Gehry Technologies' GTeam™[4] offering collaborative visualizations of 2D and 3D architectural designs. Other, more generalized collaborative visualization tools

Collaboration tools like these are intended to offer solutions for professionals in specific fields and industries. Because of this, the tools often come with significant price tags, hardware requirements, and training requirements: hurdles that are difficult, if not impossible, to overcome for general users. Open-source collaborative visualization systems designed for general data

sets, such as *Paraview*, solve the issues of specialization and monetary cost, but come at the price of requiring even more technical expertise in order to be used effectively[7]. While non-specialist participation may not be an issue for many uses, certain collaborative projects have much to gain from increased ease of access that allows and encourages users who otherwise would not meet the requirements of specialized products.

Isenberg et al described "engaging new audiences" as a primary goal for the preferred future of collaborative visualizations, and the importance of expanding collaborative tools beyond the realm of the scientific community [5]. This paper describes the development of a collaborative visualization tool that attempts to provide a collaborative and intuitive environment that allows sending and receiving simple drawn input while maximizing ease-of-use, both in terms of program interactivity and setup. The program was created with the goal of learning more about the process and challenges facing developers attempting to pursue these new audiences.

2. Technical Detail

Both client and server components of the program are written almost entirely in the JavaScript[9] programming language. The server portion is built in the Node.js [10] JavaScript runtime, and uses the Express[11] package for serving templated HTML and static script files to clients. The client architecture uses JavaScript, as implemented by any major modern web browser, in conjunction with an HTML5 canvas[12] element. The jQuery[13] JavaScript library is used for simplified DOM[14] manipulation on client side elements, and the jscolor[15] JavaScript widget is used for the client color selection input. To connect clients to the server, WebSockets[16] manipulated by the Socket.io[17] JavaScript library are used. The program is built on the *Dead Simple Screen Sharing* [18] Chrome extension developed by Mohammed Lakkadshaw, who provided a server framework and an implementation of mutation-based website screen sharing previously described by Chrome developer Rafael Weinstein.

2.1 Screen Sharing

In mutation-based screen sharing, the host user has event handlers created to watch DOM elements for changes. When a change is detected, visual information of the user's active tab is saved, then propagated to connected clients. For most screen sharing purposes, this is an extremely efficient way to collect only relevant data. If we were watching the web browser from outside the application, and were unable to watch the DOM, we would have to constantly process image data for changes, or worse, maintain a constant stream of unaltered images whenever visual content on the page is not changing. For systems intending to support large numbers of users, sending this much data would be a significant waste of resources. Using Weinstein's method, as primarily implemented by

Lakkadshaw, the program avoids excessive packet transmission by only sending visual data when applicable changes have been made to the viewed tab. Visual data is captured from the host user using the captureVisibleTab() Chrome extension API call[19], which provides a dataURL[20] representation of a JPEG screenshot of the active tab. As it is already effectively compressed, this dataURL is sent to the server, and rebroadcast to each connected client.

2.2 Client Curve Drawing

```
// Defines a node point in the curve.
class CurveNode {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.n = null; // Link to next CurveNode.
  }
}

// Defines a root node for a curve.
class CurveRoot {
  constructor(x, y, color, size, xm, ym) {
    this.x = x;
    this.y = y;
    this.c = color;
    this.s = size;
    this.n = null; // Link to first CurveNode.
    this.isDrawn = false;
    this.xm = xm;    // Maximum x - used for sharing curves
    this.ym = ym;    // Maximum y - used for sharing curves
    this.hostTopOffset = 0;
  }
```

Class variable member definitions for CurveNode and CurveRoot. CurveNodes are stored in a linked-list style data structure.

User-generated curves are stored in a linked list style data structure, allowing for efficient traversal while drawing a curve on the display. CurveRoots serve as the initial node for a curve and also contain additional information about size, color, and drawing conditions, as well as member functions for manipulating the nodes. While this structure is highly efficient for client processing, it causes some issues with networking using standardized data serialization. The most widely used standard for transmitting JavaScript objects in networked applications is through JSON encoding [21]. However, for the purpose of saving a linked list data structure, the JSON standard requires the inclusion of a large amount of redundant data, significantly increasing the bandwidth requires of a server sharing CurveNode information. To solve this issue, I wrote a minimalistic encoding/decoding specification for the CurveNode class, substantially decreasing the size of data payloads being sent and received.

Standard JSON (456 bytes)	Custom Data String (162 bytes)
`{"x":436,"y":309,"c":"rgb(0, 0, 0)","s":6,"n":{"x":437,"y":309,"n"`	`rgb(0,0,0).6.436.309.1371.700.0.437.309.440.309.442.309.444.309.445.`

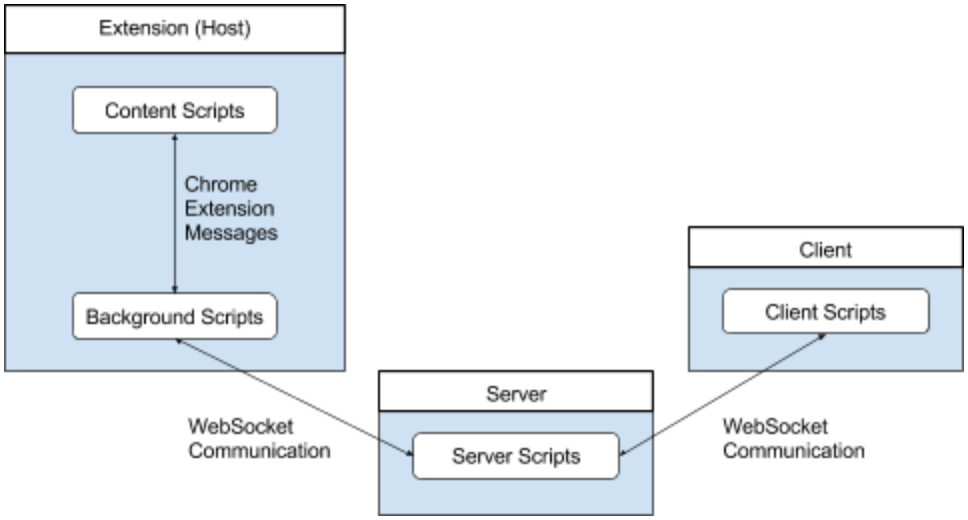
:{"x":440,"y":309,"n":{"x":442,"y":309,"n":{"x":444,"y":309,"n":{"x":445,"y":309,"n":{"x":446,"y":309,"n":{"x":447,"y":309,"n":{"x":448,"y":309,"n":{"x":450,"y":309,"n":{"x":451,"y":309,"n":{"x":453,"y":309,"n":{"x":454,"y":309,"n":{"x":456,"y":309,"n":{"x":457,"y":309,"n":{"x":457,"y":309,"n":{"x":457,"y":309,"n":null}}}}}}}}}}}}}}}},"isDrawn":false,"xm":1371,"ym":700,"hostTopOffset":0}	309.446.309.447.309.448.309.450.309.451.309.453.309.454.309.456.309.457.309.457.309.457.309.

Comparison of standard JSON encoding and implemented custom method for a simple curve.

The drawing interface is implemented on clients through JavaScript event handlers[ref], which interact with the browser to provide triggers for specific events. When the screen drawing area is initially clicked, a CurveRoot object is generated with the user's current option selections. While the mouse button is held, another event handler is triggered when the user drags the mouse across the surface. At regular intervals during this event, CurveNodes, containing the current mouse position within the canvas element, are appended to the list maintained by the original CurveRoot object.

2.3 Host Extension

The host's view is controlled by JavaScript code found within a Google Chrome extension. Background scripts are used to create a constant WebSocket connection to the server, while content scripts inject JavaScript code onto each visited website, such as the mutation observers described in section 2.1. A particularly time-consuming issue that arose in using Chrome extensions was the way in which inter-script communication was handled. Background scripts are loaded just once, when the extension is activated by the user, and continue running as long as the extension is active. The content scripts for this project must be re-injected into each page visited by the host, in order to manipulate the DOM of that page. Due to the nature of this program, constant communication between scripts able to modify the visual content of the host's active tab, and the scripts maintaining the WebSocket connection was necessary. Although the Chrome extension API provides excellent options for handling this situation, it adds a significant layer of complexity to the program.



Inter-script communication diagram.

3. Results

Despite many setbacks and technical issues, as described above, the current state of the program meets the initial goals of the project. Without any additional downloads or configuration, users with any popular modern web browser can enter a simple URL to begin participating. Input is intuitive and smooth, allowing even inexperienced users to collaborate on a shared document. Barriers of entry for host users are only slightly greater, with the requirement of a specific web browser (Google Chrome) and an extension. Viewer clients can draw on the shared screen images and have their input shared quickly and efficiently with all connected users.

Some users, while testing, have mentioned that the lack of host input options reduces the usefulness of the program. I agree, and wish I had more time to implement an interface for this behavior.

Pang and Wittenbrink wrote about encountering significant latency issues while testing the networked collaborative visualization system *CSpray* using a standard Internet connection in 1994 [6]. During the initial phases of developing my program, these issues were mostly disregarded as being predicated on the low-bandwidth technology available at the time. However, while bandwidth issues have been greatly alleviated by improvements in network infrastructure, experiences in developing this program have shown that network resource management is still an important issue when building collaborative tools that rely on such communications. Tests using remote web hosting resulted in latency issues that created frustrating user experiences, such as a curve not appearing on the host for several seconds. Implementing the data compression techniques described in Technical Detail reduced, but did not eliminate, those problems.

Although the excellent frameworks and tools mentioned in Technical Detail made it possible to complete this project in such a short time, there are still many issues with using them. In particular, the Chrome extension system of inter-script communication necessitated a much larger investment of time to implement ostensibly simple features, although these problems were greatly exacerbated by my own inexperience with the Chrome API.

4. Conclusion

Modern computers and software have greatly increased our ability to collect, store and visualize data sets. These advances have lead not only to the collection of enormous amounts of data, but to mass audience understanding of the importance and usefulness of data. One only needs to think about the term "big data" and its implications to understand the prevalence of information collection and processing in our modern world. In addition to the massive data collections built by Internet companies, smaller businesses and organizations are taking advantage of the increasing accessibility of data collection and presentation systems [22]. As these systems increase in availability and popularity, simple and intuitive means of presenting and interacting with data will become more important.

5. Related Works

There are many projects attempting one or both of the major components of this program. Simple web-based collaborative whiteboard applications include *A Web Whiteboard* [23] and *Ziteboard* [24]. Web-based screen sharing projects that only require the host user to install software include Mohammad Lakkadshaw's *Dead Simple Screen Sharing* [18], the code of which was used for this project, and the commercial product *Screenleap* [25].

References

[1] Walsh, John. "X-Ray Collaboration." John Carroll University. http://sites.jcu.edu/magazine/2012/01/06/x-ray-collaboration/

[2] BRIT Systems. "Collaborative Radiology Pool." BRIT Systems. http://www.brit.com/productsandservices/services/technology.html

[3] Lugo, Kevin. "Collaborative tools for Radiologists." Terarecon, Inc. http://www.terarecon.com/blog/collaborative-tools-for-radiologists

[4] Gehry Technologies. http://www.gteam.com/

[5] Isenberg, Petra et al. “Collaborative Visualization: Definition, Challenges, and Research Agenda.” French Institute for Research in Computer Science and Automation. http://www.umiacs.umd.edu/~elm/projects/collabvis/collabvis.pdf

[6] Pang, Alex and Wittenbrink, Craig. “Collaborative 3D Visualization with CSpray.” Links to paper at: http://avis.soe.ucsc.edu/cspray.html

[7] Kitware Inc. “The ParaView Guide: Community Edition” http://www.paraview.org/paraview-guide/

[8] Weinstein, Rafael. “DOM Mutation Observers and the Mutation Summary Library.” https://www.youtube.com/watch?v=eRZ4pO0gVWw

[9] Mozilla Developer Network. “JavaScript.” https://developer.mozilla.org/en-US/docs/Web/JavaScript

[10] Node.js. https://nodejs.org/en/

[11] Express. https://expressjs.com/

[12] Mozilla Developer Network. “Canvas API.” https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

[13] The jQuery Foundation. https://jquery.com/

[14] Mozilla Developer Network. “Introduction to the DOM.” https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[15] Odvarko, Jan. jscolor. http://jscolor.com

[16] Fette, I. and Melnikov, A. “The WebSocket Protocol.” Internet Engineering Task Force. https://tools.ietf.org/html/rfc6455

[17] Socket.io. https://socket.io/

[18] Lakkadshaw, Mohammad. “Dead Simple Screen Sharing - Overview.” http://mohammedlakkadshaw.com/blog/Deadsimplescreensharing_overview.html

[19] Google Chrome Developer Documentation. https://developer.chrome.com/extensions/tabs#method-captureVisibleTab

[20] Masinter, L. “The ‘data’ URL scheme.” The Internet Society. https://tools.ietf.org/html/rfc2397

[21] Brey, T. “The JavaScript Object Notation (JSON) Data Interchange Format.” Internet Engineering Task Force. https://tools.ietf.org/html/rfc7159

[22] Wall, Stuart. “How and Why Data Will Save Small Business.” Small Business Trends. https://smallbiztrends.com/2015/03/small-business-data-collection.html

[23] A Web Whiteboard. https://awwapp.com/

[24] Ziteboard. https://app.ziteboard.com/

[25] Screenleap. http://www.screenleap.com/