Visualize History

Computer Science 490

Sam Strasser

December 14, 2007

Introduction

Visualize History is a web-based software application that enables easy presentation of historical data. It allows a user to navigate through time and space and to imagine and recall the complex relationships between historical events.

History is naturally presented visually. Its various parts, interlinked and roughly ordered, often connect in ways too complex to be expressed with words alone, and instead require a picture to capture the content. The problem of visualizing historical data is in no way new. David Staley predicts that computers will change the way history is taught (Staley, 2003). In his book, he gives philosophical arguments and predictions for the evolution of historians and historiography, as well as examples of new ways to present data. Edward Tufte, perhaps the most famous person to visualize data, gives example after example of historical data that was properly presented, and of some that was not (Tufte, 1997). He tells of the cholera outbreak in London in 1854 near the Broad Street pump, and the role of the diagram in evaluating the cause of the outbreak. The list of examples, even as given only by these two authors, is endless.

The problem of visualizing historical data has not, however, been addressed in general. Each of the example diagrams had its own special considerations that factored into the final product. Each case was to be considered separately, and often yielded vastly different diagrams. Visualize History takes historical data and defines it in a general way that allows for meaningful display of the relationships within the topic of history.

The application aims at non-technical users who should need to know as little as possible to begin using and to appreciate the core features of the site. Development will use junior high and high school students and teachers as the main audience. The key problem the application hopes to solve is to make learning and teaching history as simple as possible for both the learner and the teacher.

To achieve its most important goal, the site must have as intuitive a user interface as possible. It should be familiar enough that any user can immediately start using it without any explanation even as to its purpose. That is, if someone accidentally typed in http://visualizehistory.com, he should be able to start exploring a topic without any further information. I focused entirely on making an intuitive framework to enable the display of user information during the first semester.

However the framework that displays interactive data is useless without data. The back end of the site is responsible for the abstract historical concepts that I mention above. It should be entirely responsible for retrieving data for the user interface, which should have no knowledge of the location, format or

content of the saved historical data. While the front end and back end of the site must communicate, one main goal of the project was to minimize the amount either side had to know about the other.

Front End

Overview

The goal of the front end of the web site is to allow average, non-technical users to navigate through history. It needs to support at least the two major browsers, Mozilla Firefox and Microsoft Internet Explorer. The user interface should be simple and recognizable, though not exactly like anything that exists already, since there is no interface for an application like Visualize History.

The user interface should allow for navigation through space using the familiar drag-and-drop interface that most mapping applications currently use, and it should allow for navigation through time, using its own addition to the existing mapping interface. The piece that allows a user to scroll forward and back in time is called the Time Slider and is the key UI piece in Visualize History. As part of this interface, the front end displays the actual historical data. It shows a title located at the time and place that the event took place. When the user focuses on the event by putting their mouse on top of it, the UI displays more information about the event, including its date, location and a short description.

A brief understanding of web programming will be extremely helpful in understanding the rest of the details of the project. Before about 2003, web applications had often been criticized as impractical. For any meaningful action, the application would have to make an entire round trip to and from the server, and then (re)render the page on the user's machine. JavaScript allowed for some tricks to avoid this problem. Animations and presentations were shown with the script, which only was loaded once, and ran embedded in page. Yet it lacked a general ability to save the user's information back to the server, making web applications last only as long as the user remained on the page.

In 2003, Gmail's introduction made an idea called AJAX popular. AJAX, or **A**synchronous **J**avaScript **a**nd **X**ml, is not a language but a combination of existing technologies that fix the main problem web applications face. In an AJAX application, a page loads once, taking care of relatively static things like menus, the header and images. Then when the page needs to change, a call is made to the server in the background. While the user waits for the server to return, he can continue working on the page in any way he likes. When the server returns, a JavaScript function is called (hence the "asynchronous") and then JavaScript manipulates the page to reflect the change made. To do so, JavaScript uses the browser's representation of the page, which is called the Document Object Model, or DOM.

Different browsers have historically had very different implementations and interpretations of the DOM, and there is a large community around creating and enforcing these standards. Until the differences in browsers are fixed, though, companies have developed libraries that abstract away some of the differences. Third-party APIs are available for the most common AJAX tasks including Events, DOM manipulation and display of Cascading Style sheets (css).

Visualize History uses two APIs: Google Maps API and the Yahoo! User Interface (YUI) API[1]. The Google Maps API provides a map that enables dragging and zooming. Customized markers, lines and polygons can be added to the map at certain geographical points. The markers are themselves DOM nodes so any HTML and JavaScript can be used. The maps API also provides a way to add custom controls that lay on top of the map and control the map. Visualize History uses the YUI's slider component to implement the Time Slider which controls the user's navigation through time.

JavaScript is an event-driven language; functions traditionally register themselves as callbacks for actions that the user might do, whether it is click a link or move the mouse. Both Google and Yahoo! Provide an API to standardize these events, registering callbacks for them and de-allocating the memory required after an event has fired. Visualize History does not user either one yet, but will probably choose one of the two APIs and use the Event class from that API.

Implementation

At first, I designed the implementation to use a loosely object oriented structure. The goal was to separate logic from presentation as much as possible, so I had objects that represented concepts, and those objects would point to the API objects that represented presentation. For instance, there was a Map object, and the Map object had an instance of a GoogleMap object as one of its variables. The script would call the Map object and only the Map object would call the GoogleMap object. I would register a logical object's methods as callbacks to some standard events, like when data had successfully loaded from the server or when the user clicked a link. After coding up the whole framework, I ran it and encountered a significant bug. When the data loaded from the server asynchronously, it would call back the objects that had registered themselves. The objects were no longer in the scope, though, so none of their variables could be accessed (see example on the right). To solve the problem, I changed all of my objects and all references to be from one global object. That way no matter what the scope was when a function was called, the variables would be in the right scope.

```
// JavaScript scoping example: Out of scope
function Object{
        this.variable = 'Value of variable';

        this.callback = function(argument){
                alert(this.variable);
        };

        Server.registerCallback(this.callback);
}

// Registering this.callback with Server
// does not work as expected
// When this.callback is called 'this'  will refer
// to the global scope, so this.variable is undefined
```

Of course using global variables for everything was not feasible for the long term as all the regular problems arose. All of the memory was allocated all the time, two functions with the same name overwrote each other, and the code became impossible to maintain, since any reference to a variable had to know exactly where it lived in the global scope. It was very hard to direct DOM manipulation since the DOM itself also lives in the global scope, creating more conflicts.

[1] For more on the APIs, see the References section

I threw away the code and started on try 2. After researching JavaScript scoping for some time, I learned that it is a lexically scoped and not a dynamically scoped language.[2] In other words, an object's scope is defined based on where it is defined in the code and not on the scope when it is running. JavaScript including something called function closures which are functions that maintain the local variables of the scope in which they were defined, even after that scope goes out of focus. I designed and implemented the same object oriented ideas using closures instead of object methods. Any function that would need an object's variables was a closure so that it could access the local variables when it was called back by the JavaScript events.

```
// JavaScript scoping example: Closures
function Object{
        var variable = 'Value of variable';

        var callback = function(argument){
                alert(variable);
        };

        Server.registerCallback(callback);
}
// Registering 'callback' works as expected
// When this.callback is called 'this'  will refer
// to the global scope, so this.variable is undefined
```

By the end of the semester, it was clear using closures for everything would not suffice. It had not fixed the problem of memory, since all closures stored the values of all private variables. Memory management is a fairly low priority for a web application like Visualize History which takes up a tiny portion of memory relative to the browser in which it runs. The code was slightly more manageable, but inserting nodes into the DOM remained difficult. The final problem with using closures was that closures could not access the objects using the 'this' keyword and the objects could not access the closures once they were out of scope (after callbacks).

One common JavaScript practice is to subclass[3] objects from APIs, thereby linking the object and the parent objects. Subclassing the objects in Visualize History would break the main goal of separating logic form presentation, which was the initial reason not to use this common paradigm. It would on the other hand solve the memory footprint problem since there would not need to be a logical layer separating the presentation objects and since the objects would not all have to be maintained in the global scope. Subclassing would improve the manageability of code drastically as each logical piece would be exactly one object (just Map, not Map and GoogleMap), assuming that the APIs do not change drastically. Lastly, it is the job of the front end to display the data, and all of its logic is inherently linked to the presentation portions. The back end-to-front end separation[4] will be maintained, and the back end will be responsible for the complex logical computations. The next version of the front end code will reevaluate which objects are in use and in what ways, and then inherit from those objects to make future code changes easier and more logical.

[2] For more on JavaScript scoping, see the References section

[3] JavaScript objects are not subclassed in the traditional sense but instead use prototype-based inheritance. It is often convenient and sufficient to consider them in the more traditional language of object-oriented languages.

[4] Visualize History makes all its calls to the server asynchronously, and the abstract class responsible for making the calls is also responsible for converting the backend data format to the front end format.

Future Goals

The first goal for the next semester will be to fix the bugs left from the fall semester. The two major fixes are in the slider's UI, which allows overlaps and in the displaying of events, which does not currently use the DOM at all.

There is a long feature list in the future. The key features are:

- Display more than one topic at one time
- Display relationships between events, with arrows and lines
- Display regions as highlighted polygons
- Assigns different colors and icons to topics and events

I will focus mainly on the back end, and when it is clear that the data allows for some of the above features, I will sketch them out and implement them. The biggest limitation in the design of the UI phase this semester was the lack of data to explore. As more data becomes available, the UI will add more features.

Back End

Overview

According to the original proposal, I would spend second semester creating an automated way to mine historical data from various places on the internet and to then upload that data to the visualizehistory.com servers. By controlling the data, the site would have fine-grained control on exactly what data was stored and its format.

Implementation

Currently there is a MySQL database running on visualizehistory.com that stores events, locations, titles and descriptions. All the data on the site right now has been uploaded by hand. The server loads and returns the data and does not know anything about the JavaScript representation. A key component of the site is and will be for a user to upload any historical data, so the database schema has to allow for any abstract historical data to be uploaded.

Future Goals

During the semester, I modified the original plan slightly. The back end of the project will now link with external sites to provide the data. The server will still need to manipulate and sanitize that data into a format that can be presented in this visual, historical manner. Many trustworthy sites exist with historical data, and those sites are maintained and updated for more frequently than Visualize History could hope to be. By leveraging the data from trusted sites, Visualize History can lift the burden of acquiring and storing the massive amount of historical data that is available. Wikipedia and the Library of Congress are the two sites that I will start with, but an emphasis will be placed on the ability to extend the list of extended sites.

Another place for research to start will be on file formats and standards. One such format is called Keyhole Markup Language (KML), which includes the idea of a place with a title, geographic location and a description, much like in Visualize History. It may have to be extended to include time, which will be part of the research of the upcoming semester. Other places to start research include Geographic Information Systems, geodatabases, and Geographic Markup Language (GML). It would be better to use one of these standardized formats than to try to scrape and interpret html from existing web pages.

References

Web sites

- http://en.wikipedia.org/wiki/HGIS
- http://www.timemap.net/
- http://www.gapminder.org/world/

Books

- Knowles, Anne Kelly (editor). Past Time, Past Place
- Staley, David. Computers, Visualization and History
- Tufte, Edward. The Visual Display of Quantitative Information

My URLs

- Project home page: http://code.google.com/p/vishis/
- Source code browser: http://vishis.googlecode.com/svn/
- Bug list: http://code.google.com/p/vishis/issues/list
- Home page: http://www.visualizehistory.com/
- Blog: http://blog.visualizehistory.com/

Coding References and APIs

- JavaScript scoping: http://www.windley.com/archives/2006/03/introduction_to.shtml
- Google Maps API: http://code.google.com/apis/maps/index.html
- Yahoo! User Interface API: http://developer.yahoo.com/yui/
- Gmaps Util Library: http://code.google.com/p/gmaps-utility-library/