xBoard: A Recordable HTML5 Canvas Based Virtual Whiteboard

Ernest Park
Massachusetts Institute of Technology
6.UAP Final Report

Supervisor: Anant Agarwal
May 17, 2012

Source: http://github.com/eipark/xboard, @commit a0a199c
Demo: http://erniepark.com/xboard/index.html

1: Introduction and Motivation

Improvements in internet accessibility and technology have allowed innovators to push the boundaries of education over the past few years. Websites such as EdX and the Khan Academy have made heavy use of video based lectures and tutorials to transform the educational experience into one that is on-demand and free. Videos (and tablet sketches a la Khan Academy) are a great tool in that they can replicate a lesson while giving learners the flexibility to learn at their own pace by pausing, rewinding, and fast-forwarding. However, recording videos is often a resource intensive effort in terms of time, money, and equipment. It is also very high bandwidth and requires a strong internet connection. As a result, video creation is often limited to instructors with these resources and users with poor internet connections cannot view the videos.

This project, xBoard, aims to make video-like media easier to create and more accessible. xBoard is an HTML5 canvas and Javascript based virtual whiteboard that can be recorded and played back. Anyone with a modern browser can create and embed sketches that can be used

anywhere on the web. One compelling use case would be in a student forum. If one student asks a question regarding an exercise, another student could answer by creating and embedding an xBoard video of them working through the problem. Since only the browser is needed, there is a very low barrier to creation, and a several minute sketch is under one megabyte, so anyone should be able to playback the video as well.

2: Overview

xBoard consists of a standard video player interface. If the user is creating a video, they also see a red record button and a tool palette. When recording, the user can draw using a pencil and different colors, erase, or clear the canvas. When not in recording mode, the user can play back what they created just like a standard video and skip to different points. The video can also be embedded elsewhere on the internet in which case the recording features wouldn't be visible. Figure 1a shows xBoard with recording tools, and Figure 1b shows it without.

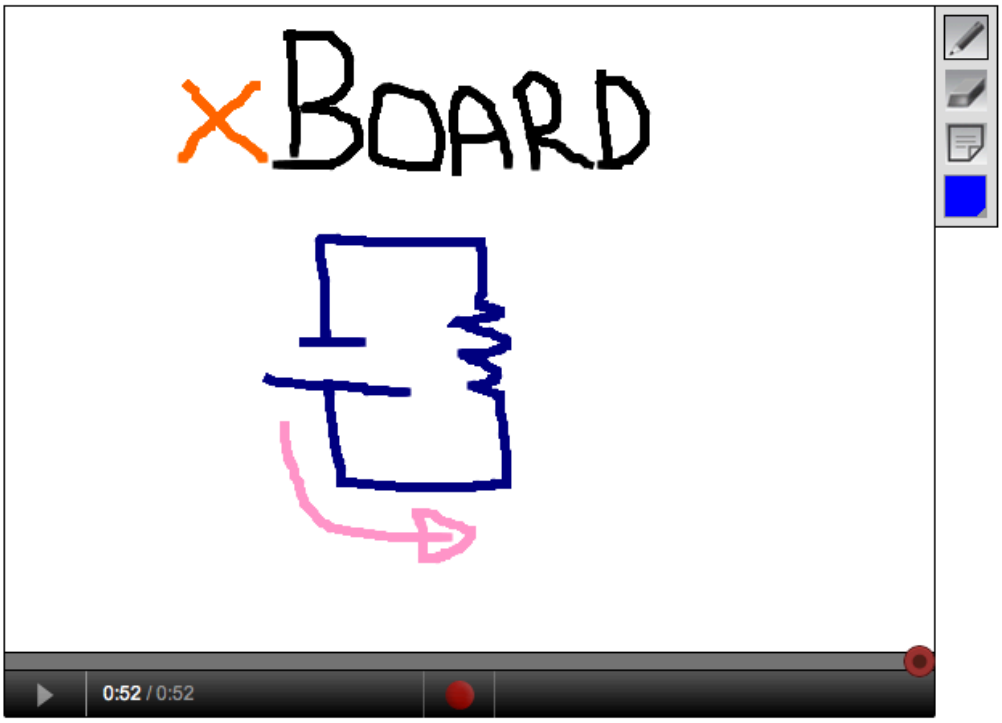

Figure 1a: xBoard with recording tools shown.

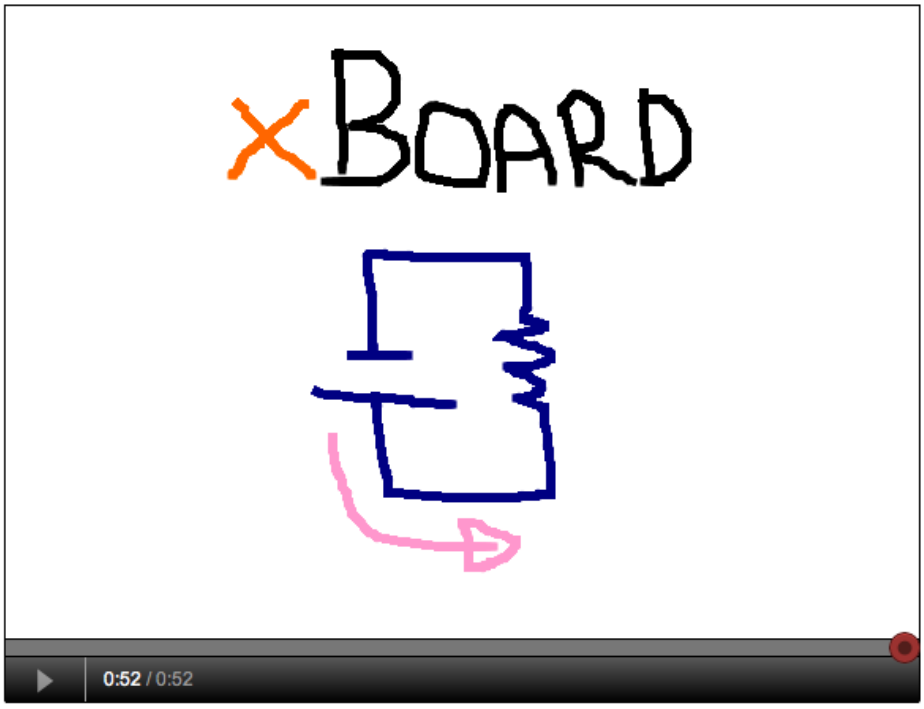

Figure 1b: xBoard as an embeddable element. Recording tools are hidden.

3: Architecture

xBoard consists of three main components: xBoard logic (xboard.js which has the `XB` variable), the visible page, and the xBoard UI logic (xboardui.js which has the `XBUI` variable). They interact in an MVC-like pattern, respectively. However it is not strictly MVC as the model can update the view without the controller. Figure 2 details the high-level architecture and some example interactions between the components.

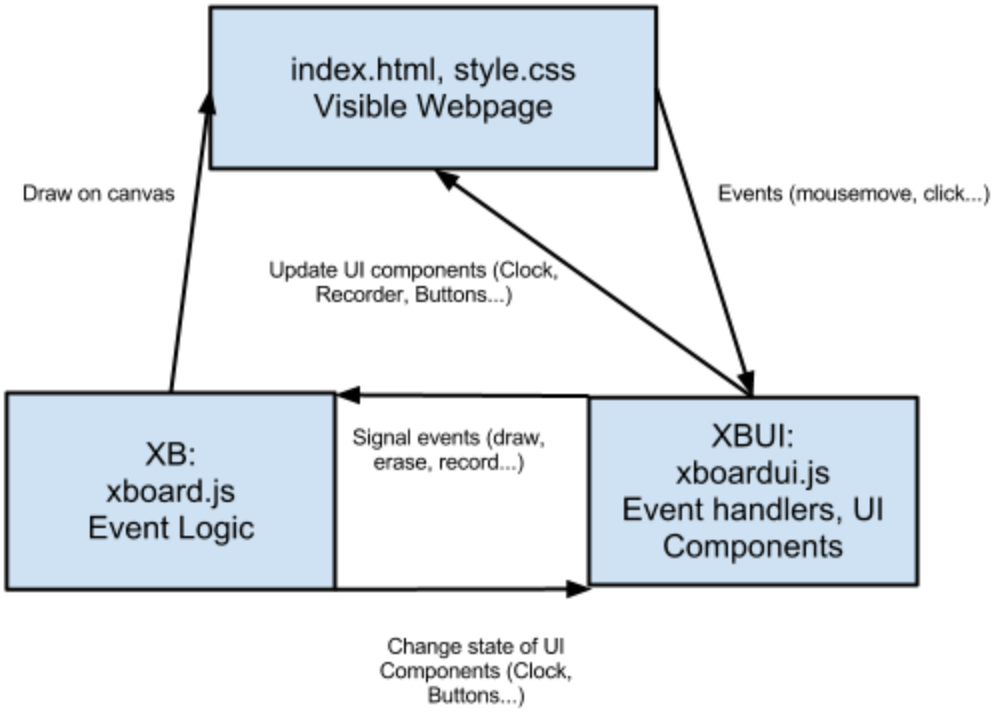

Figure 2: xBoard's High-Level Architecture and Interactions

For an example full-stack interaction, take when a user uses the pencil to draw on the canvas. When the user moves his/her mouse after already clicking down, an `onmousemove` event fires from the web page and a listener in `XBUI` picks it up. That listener calls the appropriate drawing function for the pencil in `XB`. `XB.execute(event)` is called on the pencil event which pushes the event onto the `XB.events` array, and then tells the page to draw the stroke.

xBoard was forked off of a project called HTML5 Canvas Whiteboard (http://code.google.com/p/html-5-canvas-whiteboard/) developed at the Aalto University School of Science and Technology in Finland. This project featured an HTML5 canvas whiteboard that could be sketched on and then animated. However, this project lacked any features to make playback video-like. There were no time controls and sketches could not be saved. You could simply draw,

and click an “animate” button that would play back the entirety of what you drew. From this good base, extraneous features were gutted out and the focus was placed on making recording and playback work properly along with basic editing tools.

4: Features and Components

4.1: Canvas

HTML5’s canvas element API has several drawing features built in. xBoard uses `beginPath()`, `lineTo()`, and `closePath()` to draw lines, as well as `strokeStyle()` for switching to different colors, and `clearRect()` for the eraser and clear. We will refer to these functions as “events” for the rest of this paper. Each event is wrapped in an object that stores various meta-data necessary for recording and playback such as a timestamp or x-y coordinates. When a user starts drawing in recording mode, `XBUI` listens to the events and tells `XB` which event has occurred. `XB` will push the event onto the `XB.events` array and execute the proper function in `XB.execute()` which will do the actual drawing.

The canvas element has no notion of specific pixels being drawn in. All drawings are a sequence of events, which necessitates the recording of every single event. This leads to many complications with recording and playback as is detailed in later sections.

4.2: Recording

The original Aalto University project stores a UNIX timestamp in each event object so that they can be played back at the proper intervals. However, there is no explicit recording state. This means that any time the web page is open, time is passing by and being recorded. It is also not possible to manipulate the time of playback. Once you click “animate” the entire drawing has to be drawn.

To make xBoard more video-like, a recording state was introduced. Only when in the recording state does the `XB.execute()` function push drawing events onto the events array. Otherwise, `execute()` will draw on the canvas but not save the events. This prevents playback from being recorded and getting stuck in a record - playback loop. The timestamps Javascript provides are standard UNIX epoch timestamps, so we must keep track of how much time to subtract in order for our video to start at time zero and to account for pauses in recording. This is stored in `XB.subtractTime` which is the sum of the UNIX timestamp at the first time of recording, and the total time in the non-recording state.

4.3: Timing

Timing was one of the trickiest parts of implementing xBoard. Because Javascript is single threaded, implementing a timer meant setting many timeouts and keeping track of them across many different states and sequences of events. Javascript's `setTimeout()` function executes a function after a designated delay. By recursively calling a function with `setTimeout()`, we can initiate a repetitive timer. Javascript's native `setInterval()` implements a regular timer with less mechanism, but it add events to the Javascript queue regardless of whether the previous function has completed which can lead to the page hanging as events are being added faster than they can be executed.

xBoard maintains two primary clocks, the total length of the video (`total_timer`), and the current playback position (`elapsed_timer`). Both are updated with timeouts every `XB.sampleRate` milliseconds. When recording, both clocks increment regularly and the timeouts are cleared whenever recording is paused.

Playback is more complicated. The `total_timer` does not change while the `elapsed_timer` must increment if we are in play mode, or remain unchanged when paused. If playback is paused, when a user scrubs to a different point in the video, the `elapsed_timer` can simply be explicitly set to the proper time. If we are recording when the user scrubbs, we must save that state in `XBUI`, pause playback (which clears all timeouts), set the clock, and resume playback (which sets the `elapsed_timer` timeout again). Pausing playback allows xBoard to recreate the state of the board at the new time without the clock incrementing - there are all sorts of bugs that arise when this is not the case. We also need to account for the edge case where the `elapsed_timer` will exceed the `total_timer` during one interval at the end of playback. When this occurs, we reset the `elapsed_timer` back down to equal the `total_timer` and pause the video. There is also an animation clock used in playback which is discussed in the section 4.4.

It is important to note that because of Javascript's asynchronous, single-threaded nature, there is a level of imprecision in the timing. For instance, when jumping around, an event may happen at 0:07 once, but 0:08 another time. Since events happen fairly quickly and in succession, this was judged to be a reasonable margin of error, but future work should improve upon the current timing scheme. Ideally, the play clock should be updated when an event is executed and in times where there is no event, a dummy incrementing clock should take over.

4.4: Playback

Playback in xBoard requires iterating through the `XB.events` array and re-executing events in order, but not pushing them onto the array again. This is done in the `XB.animate()` and `XB.animateNext()` functions. Each iteration of `XB.animateNext()` executes the `XB.event`

element at index `XB.animIndex`. Playback has its own timer, `XB.animateTimeout`, which ensures the proper time between each event is maintained. Whenever we try to execute an event, the `XB.animateTimeout` is set with a delay equal to the difference in time between the event and the previous event, or, if a user has just seeked to a new spot in the video, the difference in time between the event and the current playback time. Note that there is no explicit linking between the `elapsed_timer` and `total_timer` clocks and playback. Given Javascript's constructs, this appeared to be the most sensible and simple design.

4.5: Saving and Restoring

Saving an xBoard video requires storing the XB.events array, as well as other variables important to the state of the canvas - `XB.recordingTime`, `XB.subtractTime`, `XB.lastEndTime`, and `XB.context.strokeStyle(color)`. All these variables are put into one dictionary and then compressed using CJSON and LZW (more details in section 4.7a). `XB.restore()` simply does the reverse of `XB.save()` to restore the proper state. The compressed data can be stored in and retrieved from a database.

xBoard is not currently connected to any backend datastore. However, two functions, `XB.saveToDatabase(data)`, and `XB.restoreFromDatabase(uniqueID)`, are provided for developers to extend and connect to their databases. The functions are called from within `XB.save()` and `XB.restore()`, respectively. Ideally, these functions should be separated from xboard.js for good modularity, but for simplicity, they are included. The code also includes some basic error handling to help developers ensure they are saving and retrieving the data from their data stores correctly.

4.6: Embedding

In order to embed xBoard videos in different web pages, an approach similar to YouTube's was taken. Embeds are done with an HTML iframe tag which loads a web page within a web page. An example embed would look like:

```
<iframe src="http://www.urlofxboard.com?embed=DVie92hgzLe" width="600"
height="450" frameborder="0"></iframe>
```

"DVie92hgzLe" is the embed code which is an 11 character unique ID generated randomly from all 10 digits and both the alphabet in both upper and lower cases. With 62^11 combinations of unique ID's, we are highly confident we will never repeat an embed code. Developers should store xBoard videos keyed by their unique ID.

xBoard detects when it is being loaded from an iFrame and accordingly removes all recording elements so only playback is allowed. This is done in Javscript using jQuery's `.remove()` function. This removes the HTML elements from the viewer, but since the original code was all sent to the client, the recording elements can easily be recovered to manipulate the video by a decent hacker. It is up to the developer who implements `XB.saveFromDatabase()` to ensure that any modifications remain on the client side and cannot be written back to a database with some sort of authentication.

4.7: Performance (Speed/Memory)

4.7a: Compression

While not necessarily the fairest comparison, one significant advantage xBoard has over traditional video is that it takes up considerably less space. A one minute standard video of decent quality may take up several megabytes, but an xBoard video between 1000-2000 events

(about one minute) will be only around 20 kilobytes.

As the general array/dictionary format is very inefficient spacewise, several steps decrease the space consumption of an xBoard video and allow it to be saved in a database. The lowest hanging fruit was shortening common repetitive keys such as event type to just one character.

For storage, the data needed to be serialized. The natural choice for converting the data to a string was JSON. However, JSON is extremely inefficient in its repetition of keys. CJSON, a more efficient alternative to JSON, was a perfect fit to stringify the data array.

In terms of compression, research showed that JSON.HPack (https://github.com/WebReflection/json.hpack) was the most efficient algorithm, but it was not able to support the nested nature of xBoard data. In the end, LZW compression was chosen for compression because it was the easiest to adapt and develop from open source code while still providing strong gains in space efficiency.

Figure 3 compares the size of an xBoard video after using JSON, CJSON, and CJSON with LZW. CJSON with LZW leveled off to being about 28% of the data size of its JSON counterpart. The speeds of compression and decompression were not issues as even 10,000 events took only about a second to compress and decompress.

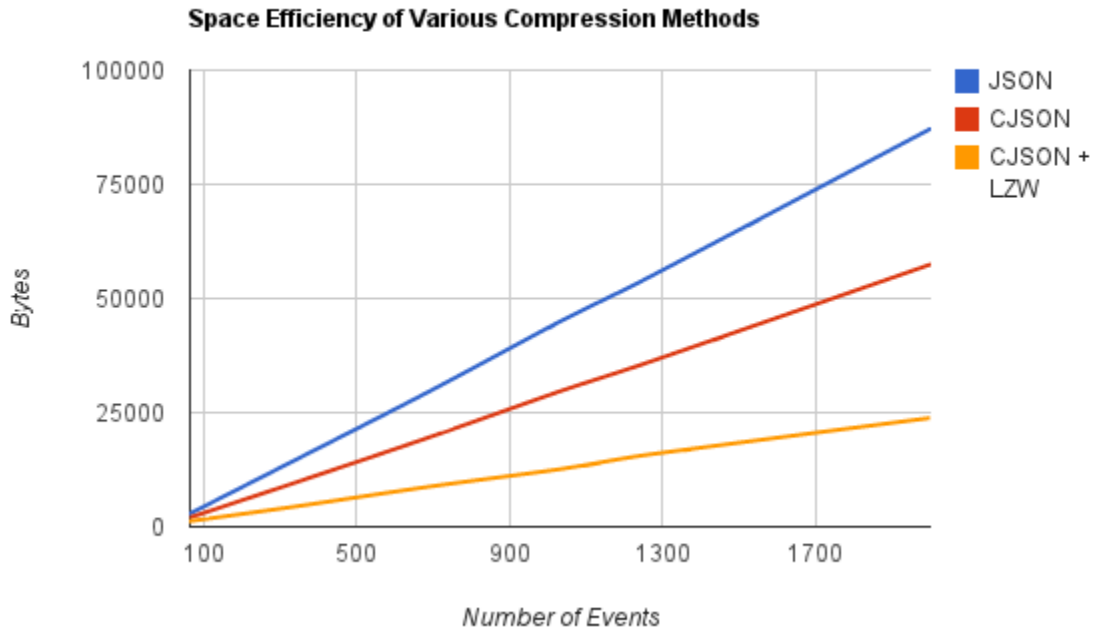

Figure 3: CJSON + LZW compressed data was ~28% the size of JSON formatted data.

4.7b: Speed

Recording and playback in xBoard generally had no issues with speed while running on a 2011 Macbook Air with a 1.8GHz Intel i7 Processor running the latest Google Chrome browser.

Speed is only an issue when the user jumps around to different points of the video which requires redrawing the entire canvas. If a large number of events must be redrawn (on the order of several thousand), it can often take up to two or three seconds to redraw. The type of events also make a difference as 1000 events made up of many short strokes is faster to redraw than a few long strokes. For reference, in standard usage while drawing leisurely, about 1500 events are executed every minute. Therefore, for a delay to be noticed, the video must be several minutes long and the user must be seeking backwards to a point in the later parts of the video.

`XB.redraw()` was slightly optimized by first eliminating redundant branching in the for loop that executes the events. In addition, the canvas is only redrawn when seeking backwards from the current playback position since canvas has no built in notion of undo. When seeking forward, the events are simply fast forwarded on top of the current state of the canvas.

While not much of an issue, speed could be improved further in future development as detailed on **http://www.html5rocks.com/en/tutorials/canvas/performance/**. Most notably, batching line events like `lineTo()` together before calling `stroke()` (which does the actual drawing) could improve speed considerably.

4.8: UI/UX

xBoard tries to be as simple to use and familiar to the user as possible. By using a familiar YouTube like video interface as well as standard recording and tool palette paradigms, the user should have no problems recognizing how to use xBoard. There are also many subtle cues that help the user use the tool. When components such as the play button or recording button are disabled, they are darkened more heavily. If the user still tries to hover over the buttons, the cursor changes to a slashed out circle to indicate the button is not active.

Below are additional UI/UX features:

- the canvas glows red when in a recording state
- the selected color and tool is outlined
- the cursor changes to a pointer when hovering over clickable elements
- the cursor corresponds to your current tool
- there are tool tips for some UI elements

5: Future Work

The original intent of xBoard was to have a transcript play along with the video to account for the lack of audio. Unfortunately, due to the scope of this project, it could not be implemented in time. A transcript would greatly increase xBoard's utility. Users could then have the context to accompany what they are watching being drawn. Although not in the current build, the code should be extensible enough that adding transcripts should be an endeavor taking only a few days.

The following would also be useful additional features:

- implement save/load button in interface
- iPad (iOS) compatibility - requires updating listeners to recognize touches as mouse events rather than scroll actions
- different brush sizes - supported by canvas
- shapes - supported by canvas and is in original Aalto University code
- integration of external images/documents that can be drawn on, for instance a homework problem with a diagram from a PDF
- improving performance with some notion of snapshotting to prevent full redraws, or going back to the last ‘Clear’ event.
- implementing better compression algorithms
- Custom icons under an MIT license

6. Conclusion

Based on my research, xBoard is the first video-like implementation of canvas. It is fast, simple, lightweight, and easy to use. Easy content creation combined with a familiar embed mechanism

will hopefully encourage widespread adoption. In addition integration with the EdX platform could also be a great testing ground for xBoard. With better cross-browser compatibility and the addition of a couple features, namely a time-synchronized transcript, xBoard can become much more useful.

I will continue to develop xBoard beyond 6.UAP. xBoard is under an MIT license and will be announced publicly in order to encourage developer collaboration through GitHub. It is my hope that xBoard can play a part in continuing to make online education more robust and accessible.

A. Miscellaneous

- xBoard was optimized for Google Chrome 18. It has not been tested in Internet Explorer. Safari seems to work perfectly as it has a webkit engine like Chrome. Firefox has a few bugs likely due to irregularities in how it handles Javascript.
- All code and images, including libraries, is less than 400 kilobytes before minifying Javascript files.
- Small bugs, improvements, and other issues are documented in the GitHub issue queue at https://github.com/eipark/xboard/issues.
- The embed mode demo playback is not LZW decompressed since this would require pulling the data from a data store since plaintext compressed data cannot be put directly into the Javascript to mimic the data being pulled. It is only parsed from a stored CJSON format for example purposes.

Data used in **Figure 3:**

Space Efficiency of Various Compression/Storage Methods

Number Events	JSON (bits)	CJSON (bits)	CJSON + LZW (bits)	CJSON + LZW versus JSON
66	2890	2037	1238	0.428373702422145
336	14405	9534	4372	0.303505727178063
694	29763	19674	8817	0.29624029835702
1039	45221	29845	12690	0.280621834988169
1240	53543	35336	15517	0.28980445623144
2000	87099	57420	23821	0.273493381095076

B. External Libraries Reference

- jQuery 1.7.2 (http://jquery.com)
- jQuery UI 1.8.19 (http://jqueryui.com)
- Really-Simple-Color-Picker (https://github.com/laktek/really-simple-color-picker)
- HTML5 Canvas Whiteboard (http://code.google.com/p/html-5-canvas-whiteboard/)
- CJSON.js (http://stevehanov.ca/blog/cjson.js): released to the public domain
- Tool palette icons (http://www.comoyodsg.com/portfolio.html#item): under CC License 3.0 for Personal Use