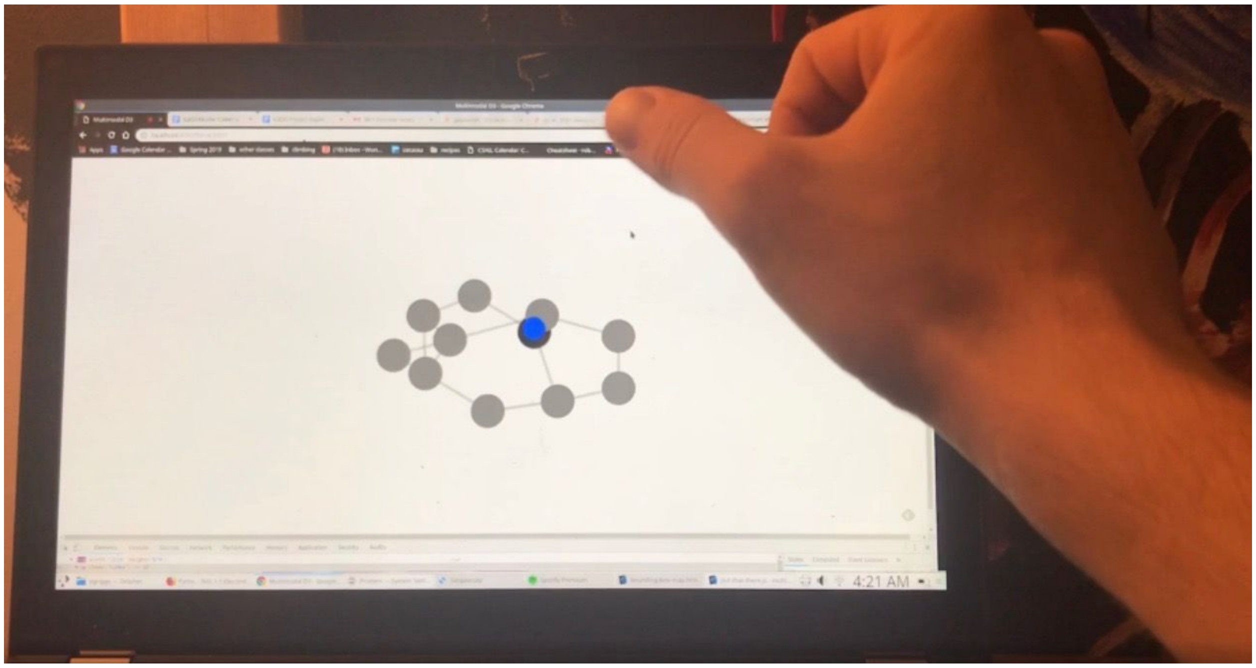


Multimodal D3.js

Peter Griggs, James Rodriguez

We built a system to interact with D3.js visualizations using the Leap Motion sensor to track hand gestures and the Web Speech API to recognize speech commands. The system observes the user's hand position, and detects when a certain gesture (pointing, tapping, grabbing) is made. Also, the system listens for speech commands such as "zoom in". Finally, it will combine the gesture and speech to affect the visualization based on what type of visualization is loaded (force-directed graph, scatter plot, etc.).

The system worked well for interactions involving gestures, but we had a few bugs with multiple similar hand gestures because they were similar enough that the Leap gave incorrect signals, and triggered the wrong interaction with the visualization. Additionally, we were limited in some ways by the accuracy of the Web Speech API, but were still able to incorporate speech into our interactions in a fairly robust way.

Code: https://github.com/peterg17/multimodal-d3

Introduction

D3.js is a popular tool for creating interactive web visualizations, used by a variety of data professionals from Biologists to journalists at the New York Times[1]. One of the main features of D3.js is that it allows a user to create DOM objects in a declarative syntax and bind data to them such that when the data updates, the object will also change, but the D3.js representation of that object won't change. This is a very handy principle because it helps a user understand what objects they are manipulating in their visualization and avoid having to keep track of a bunch of representations of their objects. But, what if the user could get even closer to the visualization layer? Through a multimodal interface, we can further D3.js' goal of bringing the representation level and the visualization level closer by allowing the user to interact with and modify a visualization not with code, but with gestures and speech.

System Description

At a high level, the system takes in a gesture and speech, and converts these inputs into a change in the D3.js visualization. As a more concrete example, one of the actions a user can do on a 2D X-Y graph is to pan around the data points. When the Leap Motion detects a grabbing gesture as seen in Fig. 1(a), it starts a pan event in our X-Y graph visualization, which starts at the viewport in Fig. 1(c). While the hand is moving

[1] https://github.com/d3/d3/wiki/Gallery

during the grab gesture, it will move the viewport in the X-Y graph in the opposite direction of the hand. When the hand stops moving, the viewport also stops moving. In Fig. 1(b), the hand stops moving, which ends the grabbing gesture. Since the hand moved to the right, we can guess that the visualization panned to the left, which is confirmed in Fig. 1(d).

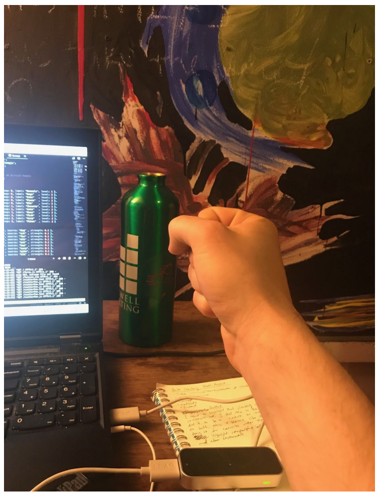


Fig. 1a

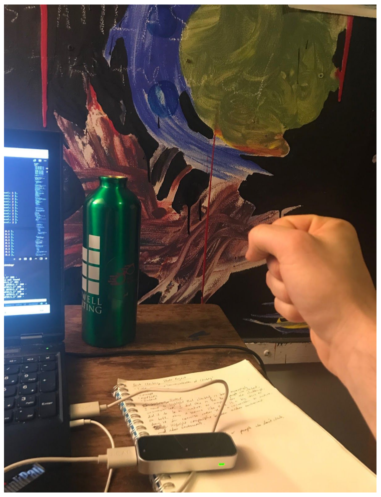
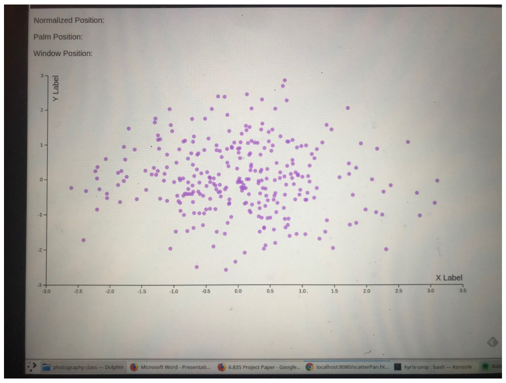


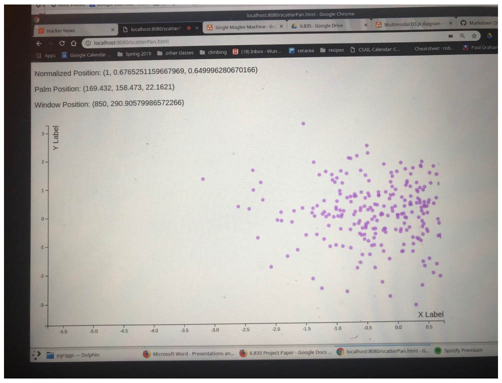
Fig. 1b


Normalized Position:

Palm Position:

Window Position:

Fig. 1c



Normalized Position: (1, 0.6765251159667969, 0.6499996280670166)

Palm Position: (169.432, 158.473, 22.1621)

Window Position: (850, 290.90579986572266)

Fig. 1d

How does it work?

Our system is implemented in vanilla Javascript that runs in the browser, packaged into a bundle with other Javascript dependencies. We maintain two listeners in our code, corresponding to each of the modality inputs we use for our system. For gestures, we have a listener for hand gesture inputs from the Leap Motion, which we implement using the LeapJS library. For speech, we start a Web Speech API listener. We use a different gesture and speech command vocabulary for each type of visualization, because the context of interaction is different for each. For example, I explained how the panning action works using a grabbing gesture through Fig. 1, but the force-directed graph visualization interprets a grabbing gesture as moving one of the nodes in the graph. Also, the put-that-there visualization does not have a grabbing gesture in its gesture vocabulary.

If the user follows the commands we have defined for the interactions, then the system works very well. This rigidness is due mostly to the limitations of our recognition systems - the Leap Motion and the Web Speech API. So, the performance of the system is good, but not very flexible. This was a result of adapting to interesting errors that we found through experimentation. For example, we found that when using the Leap Motion on the scatter plot to pan and zoom, the system would often pan when we weren't trying to pan at all. The reason for this is that the Leap is not good at differentiating pointing with a closed fist and grabbing, when measured by grab strength. The way we zoom is to point at a location on the graph and say "increase zoom", while we pan by simply grabbing at a location on the screen and moving that hand. So, when we point to a location to prepare to zoom, the system thinks we are trying to pan, and the screen does many small pans. We found through experiments that the system will make significantly fewer mistakes when pointing at a location with only 3 fingers in a closed fist. This was interesting because it seemed like a somewhat small difference in hand gesture, but it increased the Leap's accuracy in differentiating between the two gestures. From this example we learned that increasing the size of our gesture vocabulary greatly increased the possibility for conflicts between those gestures, and it was only possible to support more than one gesture if they were different enough.

Discovering how to manually zoom the scatter plot to a specific viewport was much more difficult than anticipated. This is in large part because of the lack of documentation for programatically panning and zooming, since it is an anti-pattern in D3.js. After running into walls getting this to work in D3.js, we developed a work around that used manually triggered DOM events. However, as mentioned in the Modifications section, this was even more difficult to debug, so we ended up returning to implementing manual pan and zoom in D3.js and were able to get it to work quite well.

Another part of the implementation that was harder than expected was using the built-in Leap Motion gestures, specifically the screen tap gesture, which is one of the gestures we used for zooming. We found that the Leap Motion would often mistake the screen tap gesture for the circle gesture. This made it frustrating to use at times, which led us to develop another way to zoom, which combines pointing to a location and giving a speech command.

One of the things that worked easily was adding objects to the DOM in our put-that-there visualization. This worked easily because it is exactly what D3.js was meant to do - declaring objects and binding them to data.

Modifications

One of the big ideas that really drove us was to try to simulate DOM events like mouse events, drag events, etc. at the level of the Window or DOM[2]. We really liked this idea because it required a lot less manipulation of the D3.js representation and seemed like a much more clean solution. However, we ran into a lot of trouble trying to implement interactions where we manually triggered DOM events. We actually implemented the first version of the scatter plot zooming by triggering a DOM double-click when the user does a Leap Motion screen tap gesture. Then, we ran into this problem of the DOM event changing the state of the svg object in ways that we didn't want it to. For example, every time we zoomed into the scatter plot in the svg, the zoom level, k, would increase by a factor of two but didn't decrease when we zoomed out. So, when we zoomed in and zoomed out 3 times so that we were back at the normal view and zoomed in, k would be 16. We never found out why exactly this state wasn't updating correctly, but it is probably because of something that D3.js does under the hood to maintain state.

[2] https://developer.mozilla.org/en-US/docs/Web/API/Event

Additionally, we were never able to get DOM drag events to work. We tried many different ways to trigger these methods such as trying to trigger events in all of the scopes possible - on a specific element, on the DOM, and in the Window. We decided that we had to find a way within D3.js to do what we wanted, which ended up being to manually scale and translate the svg with some simple vector math.

Response to Feedback

One of the pieces of feedback that stood out to us was to show the user in some way that their gestures were being recognized in the system. We implemented this by highlighting areas that the user is interacting with. For example, in the force-directed graph, we color the selected node differently as the user moves the node around in the graph.

User Study

We are planning to conduct usability lab studies, which under the Norman/Nielson framework, falls under a behavioral, qualitative, and scripted user study. The main two qualities of the system we want to collect data on are the learnability and accuracy of the user interactions. In order to test this, we are going to have the user perform a series of tasks for each type of visualization. For example, for the force directed graph visualization, one of the tasks would be “move node A to the top right of the screen”. For the pan and zoom visualization, one of the tasks would be “zoom in once, then zoom out and return to the original screen”. We will have users perform tasks on a visualization, using all of the types of interactions we have created so that we can compare them, and then perform a task on the non-multimodal D3.js version of the visualization as a benchmark. In order to measure learnability, we will measure how much time it takes the user to complete a task given little or no information about how to interact with the system, then measure how much time it takes knowing how to correctly use the system. To measure accuracy, we will record how many times the user must attempt a gesture or voice command once they have declared their intention to do so. The following is the script of what we will say to the subjects:

“This system provides an interface that allows a user to interact with D3.js visualizations using their hand gestures and voice. We will be asking you to perform a series of tasks for several visualizations. At the beginning of each task, we will give you a brief description of the task, then a brief description of how to interact with that type of visualization. If you cannot complete the task within 30 seconds, we will show you how to interact with the system to complete the task. Then you will be asked to complete the task 3 more times. During the last iteration of the task, you will be asked to confirm out loud what you are attempting to do, and then complete that action.”

Performance

Something that we attempted to do but was slightly out of reach is continuous zooming. This is an interesting next step because it takes full effect of the hand’s position in 3d space. To make panning and zooming in our system clean, we had to debug the gestures quite a bit. In order to make a continuous zoom work, the panning and zooming gestures would have to be quite different in order for it to work well. Additionally, the implementer might want to think about a way of smoothing the zoom constant, k. If they were to register any small change in the hand's z coordinate as a small zoom change, then the zoom might not appear smooth and continuous. I think that this wouldn't be too hard to do, but would involve careful thinking about the smoothing and the amount of events sent to the svg.

For speech inputs, we are limited by the accuracy of the Web Speech API, which has limited mileage based on the complexity of the phrase. We found that it would often mistake words in context by producing homophones or near-homophones of a word like “susan” instead of “zoom in”. In order to improve the performance of our system, we might want to train a speech recognizer on our small vocabulary of words and phrases so that we do not face the limitations of the out-of-the-box Web Speech API.

Another improvement that would be helpful would be to implement hand gesture recognition using the webcam, which would remove the need for the Leap Motion and make this tool more accessible. On the scale of straightforward software engineering to creating a sentient AI, this is not the easiest task to do well, but definitely also not as hard as creating a sentient AI to get a decent version working. We can see that there are people already doing hand gesture recognition through web cameras using OpenCV [3].

Tools/Packages/Libraries

Webpack: https://webpack.js.org/guides/installation

We used Webpack to better manage our dependencies, modules in javascript files, and to access local assets. This was extremely helpful as we found that when we ran python simple server, using vanilla javascript we couldn't access json/csv/css files from the local directory without Webpack. Used as is.

D3.js: https://github.com/d3/d3

We used D3.js to build all of the visualizations in our project. It worked well and was a core part of our project, we used as is.

LeapJS: https://github.com/leapmotion/leapjs

We used LeapJS to use the Leap Motion controller in our javascript project. It worked well out of the box.

LeapJS-Plugins: http://leapmotion.github.io/leapjs-plugins/docs/#screen-position

[3] https://medium.com/@muehler.v/simple-hand-gesture-recognition-using-opencv-and-javascript-eb3d6ced28a0

We used LeapJS-Plugins to use the Screen Position, which transforms hand position to a screen position. This worked well during the Battleship lab and we used it initially in our final project, but we ended writing our own code to calculate the position of a hand within the svg object using the Leap's relative position of the hand.

Web Speech API: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API
We used the Web Speech API to recognize speech commands. It didn't work very well for us, and we had to change our vocabulary to be simple phrases and chose phrases that sounded very different to avoid confusion. This API is easy to set up, but given more time it is not ideal. We used it out of the box.

Collaboration

James helped with developing the force directed graph visualization and he added other features (changing colors, removing nodes).

Peter worked on the force directed graph visualization, developed the other visualizations, build system, and testing the visualizations. Peter also prepared the movie and the paper.