Behind the Scenes Look at mbtaviz August 13th, 2014

Mike Barry msb5014
Brian Card bmcard

project: mbtaviz.github.io notes: mbtaviz.github.io/medialab-handout.pdf

Background

Visualizing MBTA Data is an interactive report of the performance and behavior of Boston's subway system over the month of February 2014. The visualizations focus on the performance and behavior of the subway system which differs from traditional train visualizations that focus on the train schedule. This started as the term project for a graduate course in data visualization. We collected the data in February, made mockups and prototypes in March and built project in April. After the course was over we spent May fine-tuning it and published the end product on June 10, 2014.

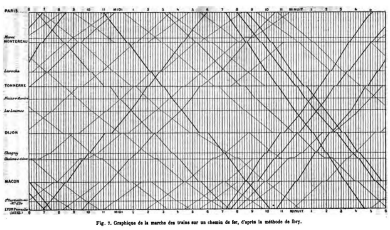
The Design Process

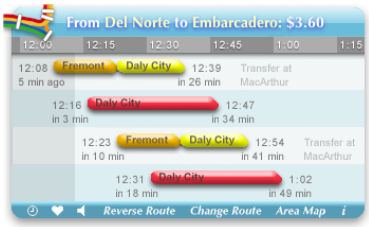
We started by trying to understand what information would be interesting to people and then narrowing down our focus based on the data that we could gather. We outlined three questions to focus on: **When and where are the trains crowded or delayed? How do snowstorms or special events affect the train system? How congested or delayed is my route?**

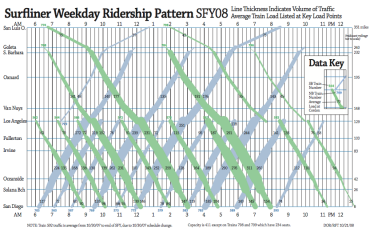
The MBTA's realtime data feed provides data for the red, orange and blue lines, but not green or silver, so the report focuses on those lines. Data is provided in JSON format and contains the current location of each train as well as the predicted time to the next stop. The real time data allowed us to see how delayed the trains were, the MBTA also provided station entry and exit data that we used to understand where the system was congested.

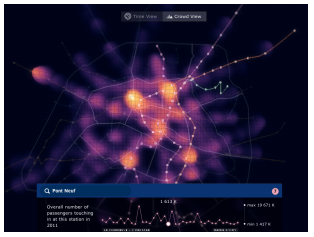
Existing Works

Below is a sampling of existing train visualizations. From left to right, Étienne-Jules Marey's schedule from 1885 [1], Bret Victor's Bart Widget for trip scheduling [2], State of California Department of Transportation report on ridership [3], metropolitan.io a visualziation of the Paris train system [4].







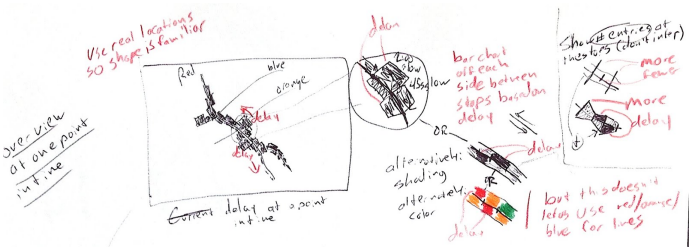


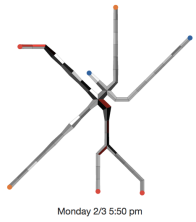
Both Marey and Victor focus on scheduling and not on train system performance. Tufte [5] provides many visualizations of train schedules, all based off of Marey's original design. The State of California also applies Marey's layout and additionally encodes the number of riders on each segment over the year. Metropolitan.io is based of of similar data to what the MBTA provides but uses different visualization techniques.

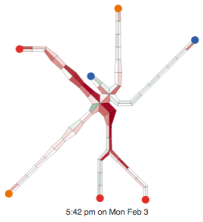
Prototypes

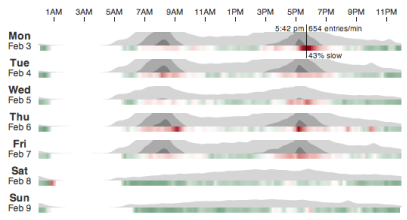
The project started with more than a dozen mock-ups, of these 6 to 8 made it into the prototype phase where we made samples with real data. Prototypes helped us validate ideas and find trends in the data to explore further. Several ideas such as the congestion and delay visualization only came after significant investment in prototypes.

Mockup | Prototype | Final Implementation



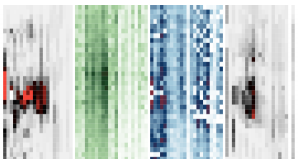


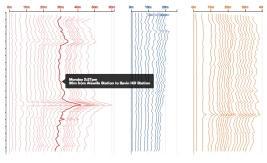


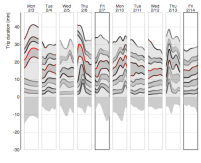


Iterate, Iterate, Iterate

We went through several iterations of each prototype before settling on the versions shown in the report. Several other prototypes were useful in exploring the data but couldn't be integrated cleanly. The image on the left shows congestion and transit times together over time but was hard to follow. The middle show delay from each line, a first attempt at the Your Commute visualization. The one on the right shows a more complex Your Commute visualization that we left out because it didn't reveal much more than a scatter plot, and the scatter plot was more obvious.

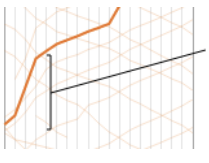






Layout and Publishing

Once we discovered the interesting features of our dataset we created the narrative and organized the visualizations into the different sections. We added the interactive text and annotations and iterated on the finished reported until we covered all of the major areas. After we were comfortable with the content we optimized the website and published it online.



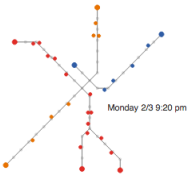
Average Number of Tr

	Weekdays	Satu
Red	450	350
Orange	320	260
Blue	380	260
Total	**1150**	**870**

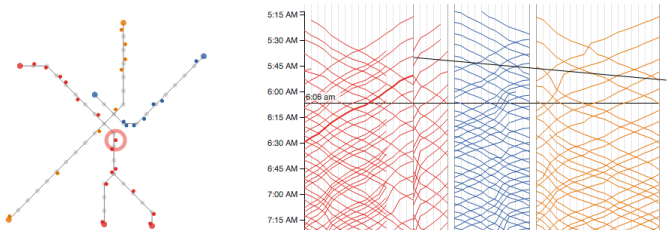
Our Design Concepts

Levels of Abstraction

Bret Victor's *Up and Down the Ladder of Abstraction* [6] influenced many parts of the report. The model for our system is train locations on subways lines and we represent this using a small map that closely resembles the subway map seen on most trains www.mbta.com/schedules_and_maps/subway.



An abstraction one level up is the trains on subway lines moving over time. The Marey diagram represents this abstraction, but is difficult to read and interpret. Victor's advice is to tie these two together using lightweight interaction which lets the user understand the abstraction in terms of the model. This way the higher level behaviors and patterns that are clear in the abstraction can be more easily related back to the model that they represent. In our case it's easy to see the delay caused by the trains with mechanical failures in the Marey diagram because the slope of the lines change drastically. However it's difficult to see that the trains buch up and slow down in the diagram. The tight feedback loop of hovering over a time on the Marey diagram and seeing where the trains are at that time (as well as lightweight interaction to link the highlight corresponding parts on hover) reinforces the connection the levels of abstraction and helps make the Marey diagram more understandable.

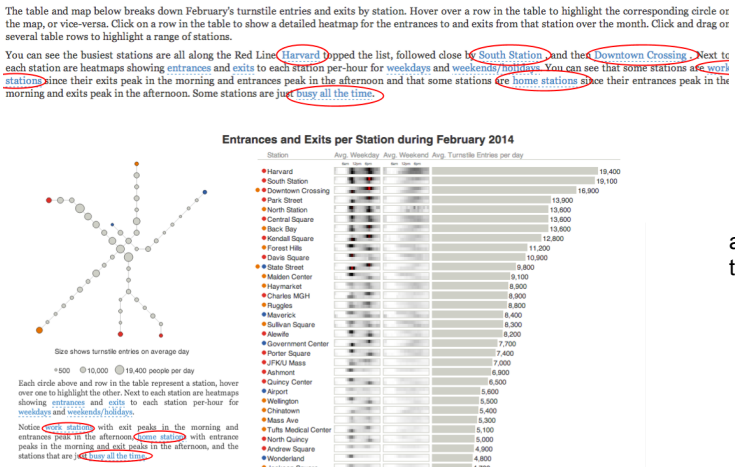


Avoiding Administrative Debris

A few things you won't find in our visualization are checkboxes, sliders, or buttons (aka "administrative debris").

At first we were not going to include the Marey Diagram, we just wanted to show trains on a map and let the user control time using a slider. We realized though that the Marey Diagram could serve as an information-rich interface for controlling the visualization while avoiding typical administrative widgets.

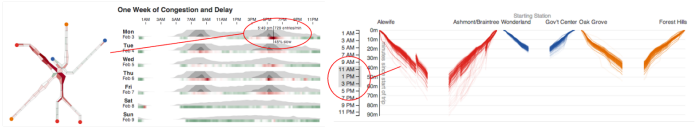
Instead of buttons with short, cryptic labels we allow you to click on or hover over linked words to control the visualization. These words are inside of sentences organized into paragraphs that put the point of interaction into context with the rest of the narrative.



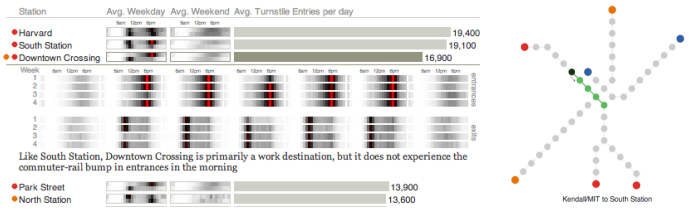
Interaction Considered Harmful

We avoided mandatory interaction as much as possible. Even though we provide details on demand through tooltips and dynamic annotations, chart keys and labels provide the same information statically. The fact that the entire visualization is readable as an image also makes it easier to explore on mobile devices.

We did compromise on this in a few places. We wanted to use small multiples for the lined-up Marey and congestion diagrams, but due to the low resolution of computer screens we could not get enough detail out of a smaller version of each. Instead of small multiples we allow you to hover over a particular time to change the data shown in the graphic



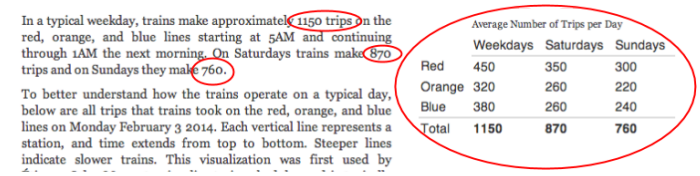
Also, the turnstile heatmaps and your commute scatterplots were expensive to render so we require the user to select which instance they want to drill down into to see more details, otherwise the page would take too long to load.



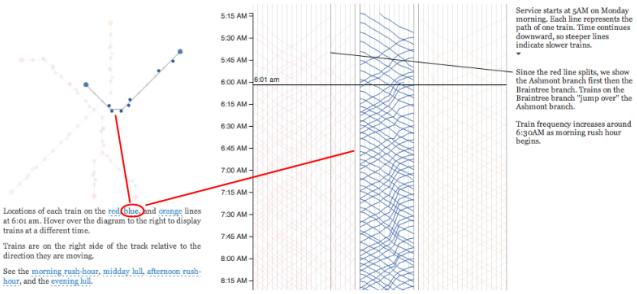
In all of these cases we start with initial default values so the user is never presented with an empty chart prior to interacting with it.

Integrating Words, Numbers, and Graphics

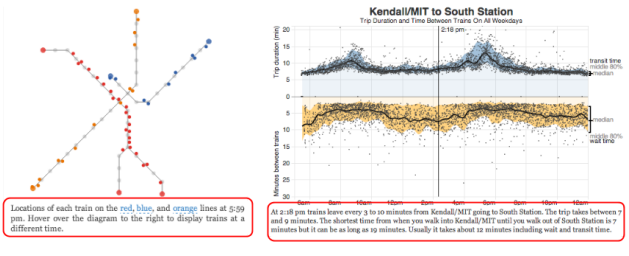
Data does not always require a visualization and visualizations do not always need to be interactive. We use sentences to show datasets with only a few numbers, tables to explain datasets with more numbers, and graphics to explain datasets with hundreds or thousands of numbers.



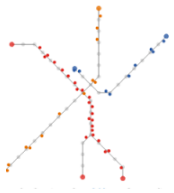
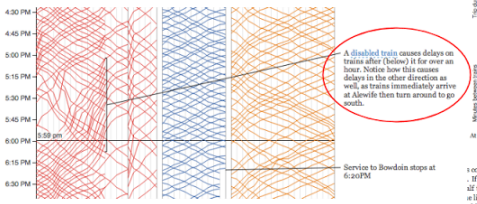
As a twist on this concept, we also link hovering over or clicking on words to changing associated graphics:

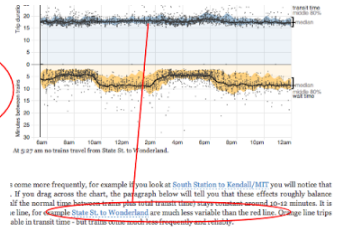
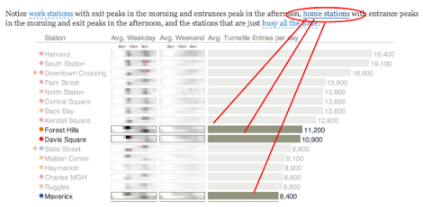


and we link hovering over graphics to changing the words and numbers that explain the graphic:



Show Me What's Interesting
We spent a lot of time analyzing the data to pick out the interesting points and showed them off with liberal use of annotations. This lets people who spend a short amount of time playing with the visualization get the highlights while users who spend more time can explore and dig into the details. If there are anomalies in the data explaining the cause of the anomalies is an easy way to show the viewer something interesting.

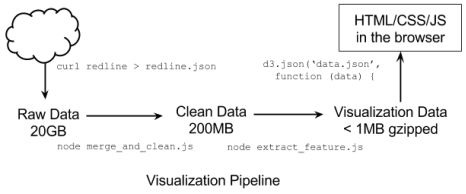



What We Didn't Do
We wanted to show how crowded each individual train was and also to come to a sweeping conclusion about the train system, but the data wasn't there to do either, so we ended up just showing the data. We also wanted a "grand visualization" that showed all of the variables together in a single view, but we couldn't quite pull that off. Instead we went with separate views, each of which showcased a particular aspect of the data.

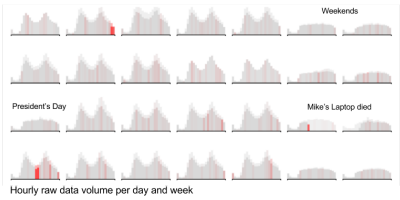
Gathering Data

We wrote simple bash scripts and scheduled them using cron jobs on old laptops in parallel for the month of February to get the realtime data files. We also pulled the MBTA GTFS zip file which contained the scheduling data for all trips and worked with a contact at the MBTA to get per-minute entry and exit counts at each station from their turnstile data.



Visualization Pipeline

Merging and Cleaning the Data
We experimented with python, scala, and node.js for data pre-processing. Python was an order of magnitude slower than scala or node.js which were comparable. Since only one of us was comfortable with Scala and we both knew JavaScript, we chose node.js for continued work. We found the bulk of the time spent running our scripts was from opening files from disk (12 files per minute of gathered data) so our preprocessing scripts merged the two datasets, removed redundant data, and wrote the data out to hourly gzipped data files with a single json blob per line, for example `redline/yyyy/mm/dd/hh.json.gz`. This gave the optimal performance for downstream feature-extraction scripts, while still allowing us to process date ranges at hour granularity.



Hourly raw data volume per day and week

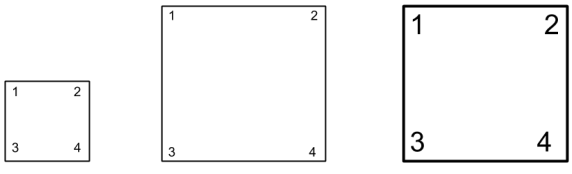
Extracting Visualization Data Files
We wrote a node.js script for to generate a minimal JSON data file for each visualization embedded within our larger web page with the goals of minimizing file size and reducing the amount of work done in the browser at runtime. Almost every script ran inside a reusable helper that took a date range, opened the necessary `.json.gz` files and called a function with each line of that file.

Building the Web App

The web application is a single HTML file with descriptions and explanation surrounding placeholder empty `<div>` elements for our visualizations. Our JavaScript files load at the end, read in the data files, render SVG dynamically into the placeholders, and setup listeners to handle interaction. We used a few open source projects to make development easier including D3.js (d3js.org) for mapping our data to DOM elements that create the visualization, underscore.js (underscorejs.org) for collection manipulation, jQuery (jquery.com) for additional DOM manipulation utilities, moment.js (momentjs.com) for date and time formatting, and d3-tip (labratrevenge.com/d3-tip) for creating simple tooltips. Below we describe how we solved several problems along the way. See the github repo for the full details (github.com/mbtaviz/mbtaviz.github.io)

decoupling the visualizations We wanted to make sure our visualizations did not interact in undesired ways when placed on the same page. We used the less CSS preprocessor (lesscss.org), which allowed us to nest selectors and ensure that all rules for a visualization would apply only to that section.. Each visualization got its own copy of the data it needed and code had to be explicitly put into a common location to share between visualizations.

responsive SVG using D3? This is tricky, since D3 guides you down the path of using fixed pixel sizes for all of your elements. To make your visualization grow or shrink with the screen size you need to programmatically resize it. A few options are:



We use the re-render option in most places to preserve text size at different scales, but for the turnstile heatmap which is expensive to render, and the text feels small at the default size, we use the transform=scale(...) option to make our text bigger as well.

data loading Since we needed to load many data files and render several different visualizations we created a utility that requests several files asynchronously and coordinates progress and completion events, bubbling them up to user-defined listeners

```
VIZ.requiresData([
  'json!data/file1.json', 'json!data/file2.json'
]).progress(function (percent) {
  d3.selectAll(".progress").text('Loading data... ' + percent + '%');
}).onerror(function () {
  d3.selectAll(".progress").text('Failed to load data');
}).done(function (dataFromFile1, dataFromFile2) {
  // use data
});
```

screen size/responsive design We developed on 1200px wide macbooks and optimized for that, but some people we showed this to viewed on iPhones, some viewed on tablets, and Bill Shander viewed it on a large, high-resolution monitor. After this feedback we pulled in Twitter's Bootstrap (getbootstrap.com) and customized its responsive design utilities to create 3 responsive width breakpoints: 789px, 990px, and 1200px. We set `<meta name="viewport" content="width=768">` in our header to ensure that phones and tablets rendered the page using the minimum fixed width. This allowed us to not worry about rendering our visualizations any narrower than 768px and also looked acceptable on a range of screen sizes.

browser compatibility Underscore, D3, and jQuery work across modern browsers and we used an es5/6 shim to let us use newer language features in older browsers. Still, ie8 does not support SVG. Instead of bending over backwards to support them, we used phantomjs (phantomjs.org) to render our page to a png file and show that instead.

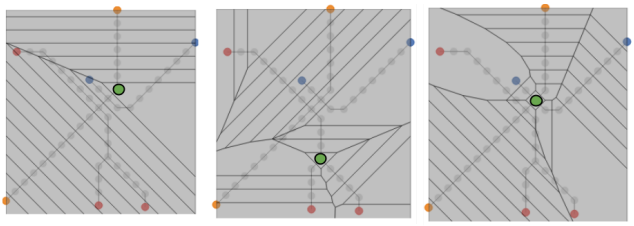
```
<!--[if lt IE 9]>
  <img src="ie.png">
<![endif]-->
<!--[if gte IE 9]><!-->
  The visualization
<!--<![endif]-->
```

scalability & performance As our visualizations grew in size and the number of DOM nodes increased, we had to optimize a few things to get acceptable performance. Adding mouseover D3 event listeners to thousands of DOM elements was slow, so instead we added a single listener to the parent element that handled mouseover events bubbled up from its children. We pushed all calculations for each data point into the pre-processing step to avoid slowing down the browser.

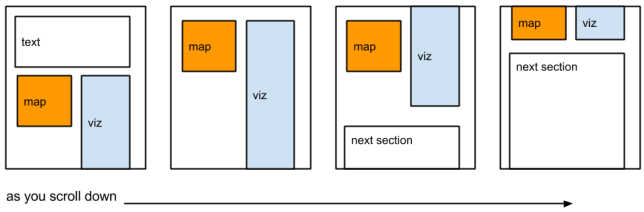
To optimize further the first thing to try would be rendering parts of our visualization involving many small DOM elements that don't require interaction to canvas instead of SVG. Alternatively, we could just render those section to images and load those instead of rendering each time.

fonts We chose Helvetica for our headings to match the MBTA's sign style. For our body text we wanted to go with Garamond but after some reading learned that Georgia is better since it is optimized for computer screens, whereas Garamond is optimized for print.

voronoi picker Our final visualization allows you to click and drag between two stops to see a scatter plot of delay between them. To implement our rules for valid destinations from a source efficiently as you drag, we used D3's voronoi utility which takes a set of points and renders a polygon for each where all points inside that polygon are closest to the point. We render voronoi polygons for the set of valid destinations every time you click and start dragging, and add our mouseover listeners to those polygons as shown below.



fixing maps in place as you scroll We wanted some fixed context while scrolling to navigate the visualization, so we made a utility that listens on scroll events and programmatically sets the `top` and `left` css attributes of the map as you scroll to make the map appear to stay fixed to the top. Since scroll events fire on iOS devices only when the scroll finishes, we disable this entirely for iPhones and iPads.



Performance was slow on firefox and safari at first, but we added `transform: translate3d(0,0,0)` CSS style which causes browsers to hardware accelerate animations and the improvement was noticeable

Conclusions about the T

Through our analysis we found that mechanical issues account for the worst, most unpredictable delays. Localized events like hockey and basketball games cause temporary spikes in ridership but have very little effect on delay. Snow storms drastically reduce ridership, which improves performance more than the weather hurts it. Neglecting these sporadic events, increased transit times around rush-hour are mostly offset by the fact that trains run more frequently so you need to spend less time waiting for the next train.

Each line has its own unique characteristics of ridership and delay. The blue line experiences lowest ridership and also delay. The orange line has rock-solid transit times but wait times vary much more drastically. The red line is the busiest and has the worst delays.

The MBTA gave feedback that this aligns with what they know about the system. They explained that the cause of congestion-related delay is the time it takes to close the subway doors at each stop as people scramble to squeeze into an over-crowded train. The tracks are divided into zones and only one train is allowed in each zone at a time so when one train stays at a station for too long it has ripple effects on trains close behind it.

Lessons Learned about Building Interactive Data Visualizations

While trying to understand our large dataset, we avoided using complex statistical techniques to over-reduce our data and instead did the minimal reduction necessary to fit it into high-resolution images. When we could see the data and all of its intricacies, our eyes drew their own conclusions.

Mockups and prototypes helped us formulate ideas but we found that nothing beat iterating on working code. When we had something concrete to talk to we came up with new ideas that we would never have thought of in a mockup or prototype. We experienced first hand that higher levels of abstraction allow for powerful reasoning, but you need to start at a level the audience is used to. We had success using words, numbers, and images to explain our data with no preference towards one or the other. We mostly avoided gratuitous interaction and color but found that subtle use of both allowed us to better link words to images. We wanted to use more small multiples, but had to compromise since the resolution of computer screens is still much lower than paper and instead used interaction to change a single larger graphic. When necessary, lightweight interaction with words and images served as a much richer alternative to typical administrative debris like buttons, checkboxes and sliders.

References

[1] Marey, Étienne Jules. "La Méthode Graphique Dans Les Sciences Expérimentales Et Principalement En Physiologie Et En Médecine. 2. tirage augm. d'un supplément sur le Développement de la méthode graphique par la photographie; avec 383 figures dans le texte." Paris: G. Masson, 1885. archive.org/details/lamthodegraphiq00maregoog
[2] Victor, Bret. "Magic Ink: Information Software and the Graphical Interface." Online. 2006. worrydream.com/MagicInk/
[3] Nguyen, Lam. "Surfliner Weekday Ridership Pattern." Memorandum to Chairs and Commissioners regarding Follow-up on September and December 2008 Rail Items. State of California Department of Transportation. January 14, 2009.
[4] metropolitian.io. "Metropolitain.io; Paris never sleeps." Online. 2013. dataveyes.com/#!/en/case-studies/metropolitain
[5] Tufte, Edward. "Envisioning Information." Graphics Press, Cheshire, CT, USA. 1990.
[6] Victor, Bret. "Up and Down the Ladder of Abstraction: A systematic approach to interactive visualization." Online. October 2011. worrydream.com/LadderOfAbstraction/