Whatever the Weather
Basil Siddiqui and Kevin Schmid

With our project, we aim to utilize visualization techniques to promote explanation of weather data, and related data about the environment (for example, population size). We aim to design an intuitive visualization that will provoke users to explore trends in climate data, and investigate correlations, look at change over time, and be able to answer their own questions about weather trends. One of most immediate applications of this, we feel, is to examine claims made about global warming in greater details. Websites like NASA's offer, to some extent, visualizations of data for variables like "global surface temperature." But these visualizations have limited interactivity: while there is, for example, a world map with temperature changes, it doesn't support zooming, details-on-demand, or any of the useful features that make visualizations dynamic and engaging. As a result, the website gives a straightforward visualization that tells a one-sided story. We think that there's a lot more to the global warming story, and we hope that we can produce a visualization that is effective in letting users piece together the story themselves. Primarily, we will limit our region of interest to the United States and work to enable the following tasks:

- What portions (cities, states, regions) of the United States have experienced the largest changes (both increases and decreases!) in temperature (and any other factors that the API offers, such as dew point, humidity, air pressure, wind chill index, etc.)?
- How has the population of a city/state/region changed over time? Did the temperature change accelerate when the population growth in a region accelerated?
- Be able to compare this data (and more, on-demand) in a detailed way for two or more cities, on a line graph. Or more generally, a rectangular region of the United States that the user can select.

We will obtain the data for our visualization from the following sources:

- **The wunderground.com Weather API** offers historical weather data for cities. The data is available in both JSON and XML formats. Currently, we have registered for an API key, and we are able to obtain the weather data from this API. We did notice that there is an API limit of 500 calls per day; we will probably have to fetch some of this data in advance (instead of fetching it as the user requests it, which may not even be possible without doing something like this). This shouldn't be a problem, since one API call can give us the historical data for an entire state's worth of city temperature data for a particular day. So, we can choose some established days in the year (a summer date, winter date, etc.) and then accomplish this in about several hundred API calls using a Python script. Other than the API key, no authentication is required, so we don't expect this to be that complicated.
- **Various census data sets** to extract population information for cities. There appears to be population data available for the 1980-2010 censuses, right around the global warming hot spot[1]. Datasets appear to be unorganized and available in inconsistent formats. Hopefully, with Google Refine and Google Fusion Tables, it won't be difficult to grab

[1] No pun intended.

corresponding population data for each of the cities.

We would like our main visualization to convey data in a color-colored, interactive map. Let's stop there first: independent of the data we've collected, we need to have the framework set up for this map. Ideally, the map will be zoomable, have labels, and be reasonably accurate in terms of scale and depiction of landmarks. We have considered using the Google Maps API, which can embed a full screen map on a page. The map can be represented as a JavaScript object, which has various methods that allow you to customize the appearance and behavior of the map, specifically with event handlers. However, there are some downsides to this: namely, the customization that can be done is limited by a high-level API, which could restrict the possible visualization techniques we could employ. (That said, the API's markers, events, and polygons aren't bad!) To make this more in-line with a project for this class, we will probably use D3 and its lightweight abstractions over DOM manipulations instead to create a map, which won't be difficult after completing HW6, according to feedback from our project TF.

Once we have the map, we need to decide on how to initially present the data to the user, and then further decide what kind of interactive components to add. An earlier homework assignment included readings that advised against the use of a rainbow color map, so we'll stay away from that, but we will probably want to use some sort of color scale to indicate temperature, and color the map in accurately based on this scale. When you load the visualization, the information displayed could pertain to the most recent decade (2010 temperature data), but there could be a slider on the bottom of the map (similar to the NASA visualization referenced earlier) that allows you to select a specific time period. Moving this slider back and forth will presumably change the colors shown on the map and enable the viewer to get a good sense of the change over time for a certain place (or just the entire US in general) on the map. The behavior would be the same when one zooms in on a particular spot: you would just see the color change for that particular region.

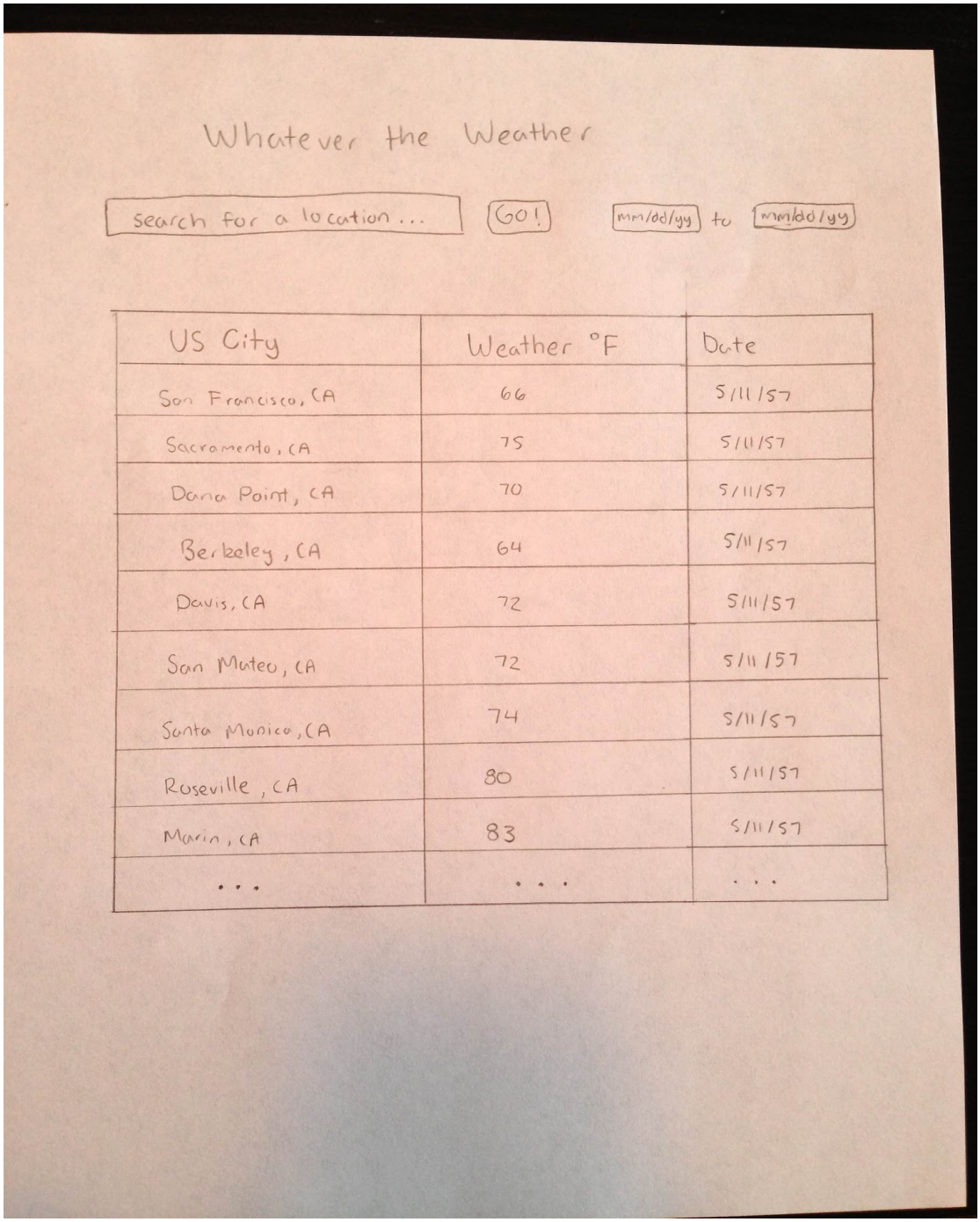
One possible interactive component of this map would be the ability to select a specific city. When you select a specific city, a modal window would appear, displaying specific information for that city (such as the population data, climate data, etc.), as well as any other visualizations we may want to display on a per-city basis. This could be a simple line graph, showing the trends in the data mentioned earlier. We feel that this series of visualizations would promote a very natural and intuitive way to explore the data: yes, there's a lot of data available, but not all of it is shown at once, it's not too "busy," and it doesn't stray too far away from familiar visualizations. At the same time, we intend to add "features" to our visualization to make it more engaging, but keep it not overwhelming for viewers. One of these features is a tool that allows the viewer to select a rectangular region from the map, and then view specific details about the cities contained in this region, including the averages of all the statistics from the API, as well as the trends for the averages over time. At the same time, the user may be interested in customizing exactly what kind of data is displayed. Currently, that functionality is limited; the map merely provides a slider for selecting a particular target year. We can extend this to allow for removal of outliers and to limit the data shown for cities with a value for a certain data type in between a certain range, etc. Form elements (especially with a library like JQuery UI, or Bootstrap) would make this easy to implement on the interface side, and we could just add a layer of abstraction in our JavaScript functions that makes the rendering functions accept range

information.

Table of Contents

Project II

Visualization Sketches

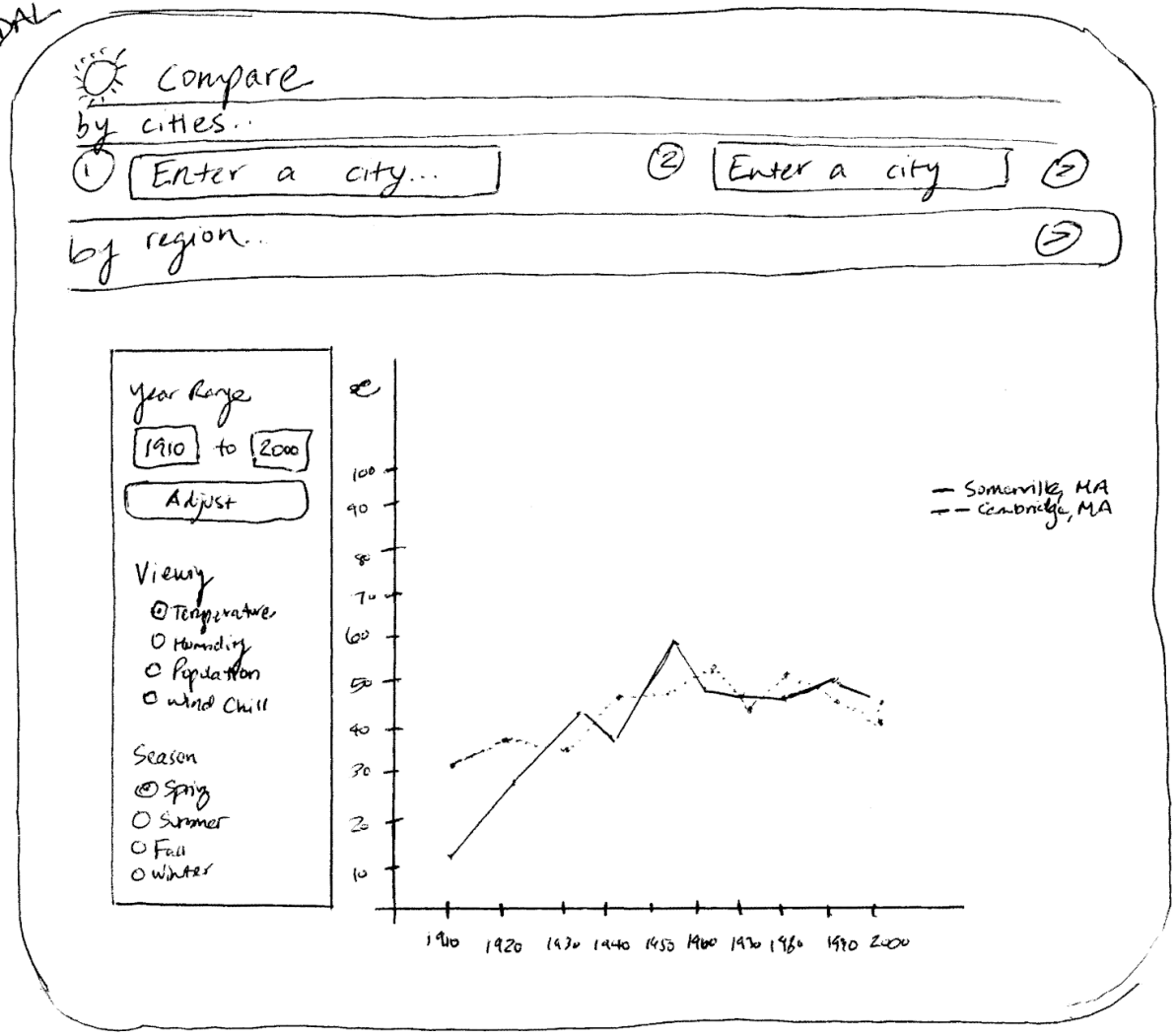


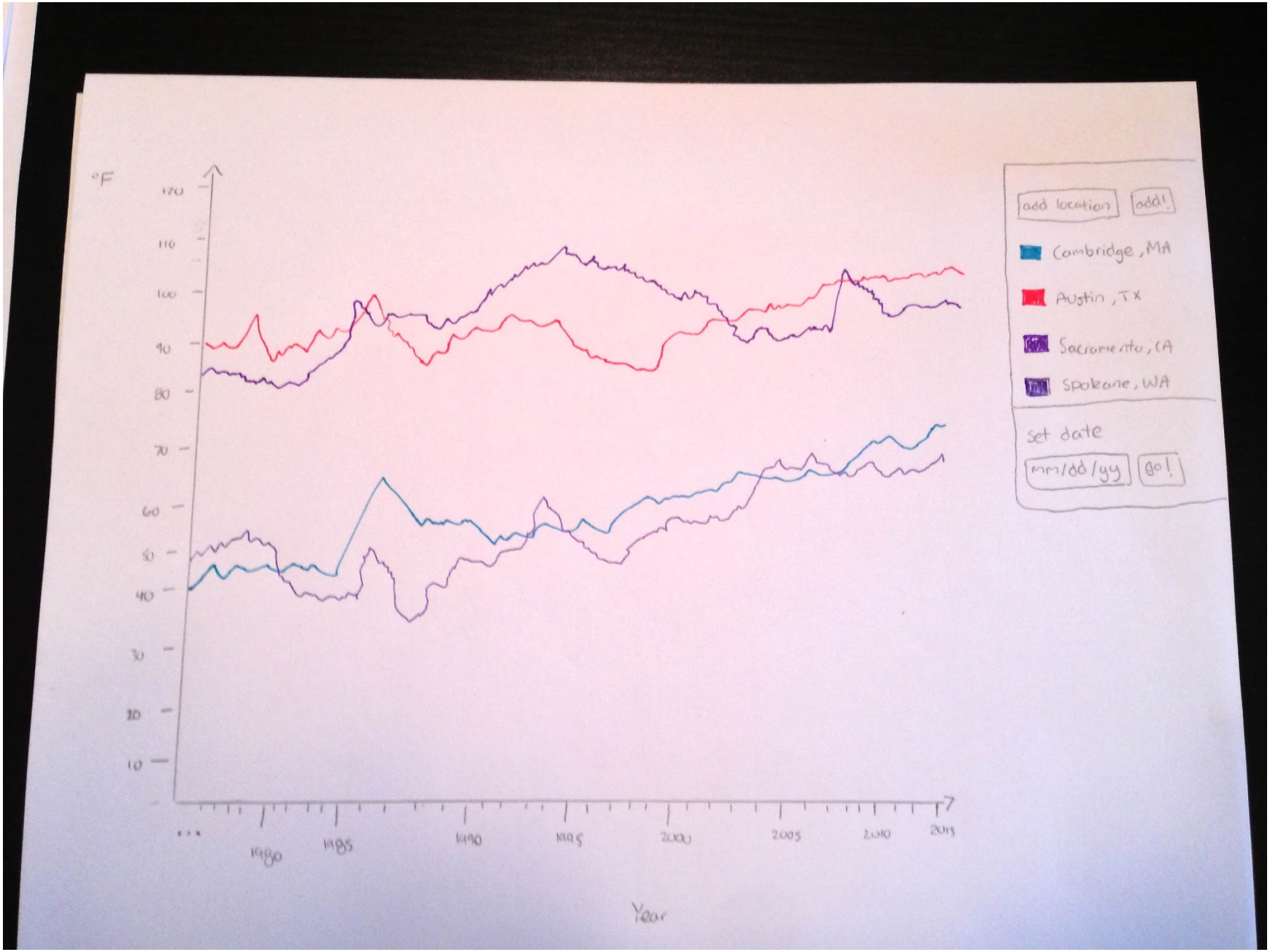
Whatever the Weather

search for a location ... | GO! | mm/dd/yy to mm/dd/yy

US City	Weather °F	Date
San Francisco, CA	66	5/11/57
Sacramento, CA	75	5/11/57
Dana Point, CA	70	5/11/57
Berkeley, CA	64	5/11/57
Davis, CA	72	5/11/57
San Mateo, CA	72	5/11/57
Santa Monica, CA	74	5/11/57
Roseville, CA	80	5/11/57
Marin, CA	83	5/11/57
...	...	...

COMPARE MODAL

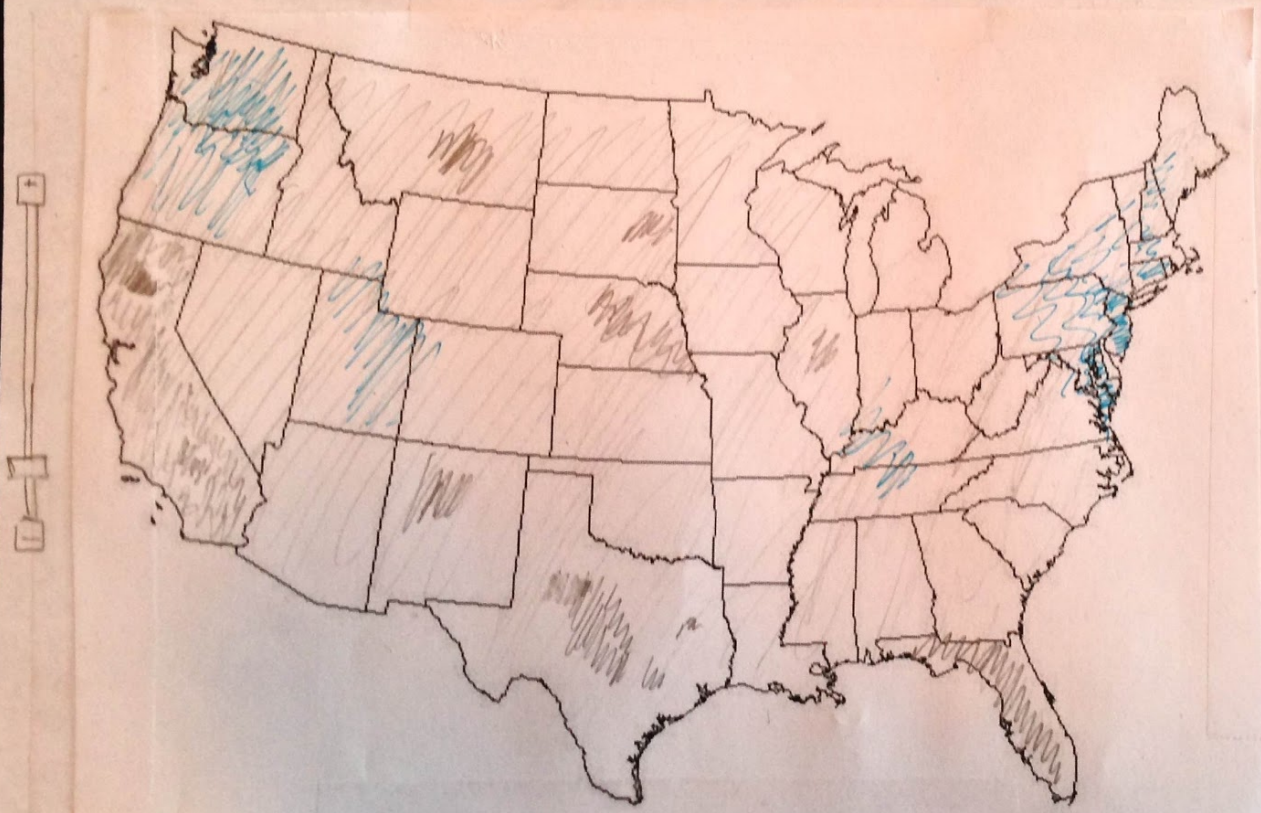



°F
120
110
100
90
80
70
60
50
40
30
20
10
1980
1985
1990
1995
2000
2005
2010
2015
Year
add location
add!
Cambridge, MA
Austin, TX
Sacramento, CA
Spokane, WA
set date
mm/dd/yy
go!

Whatever the Weather

Search a specific area or event . . .

GO!



Control Panel

animate visualization

1957

slice through the years

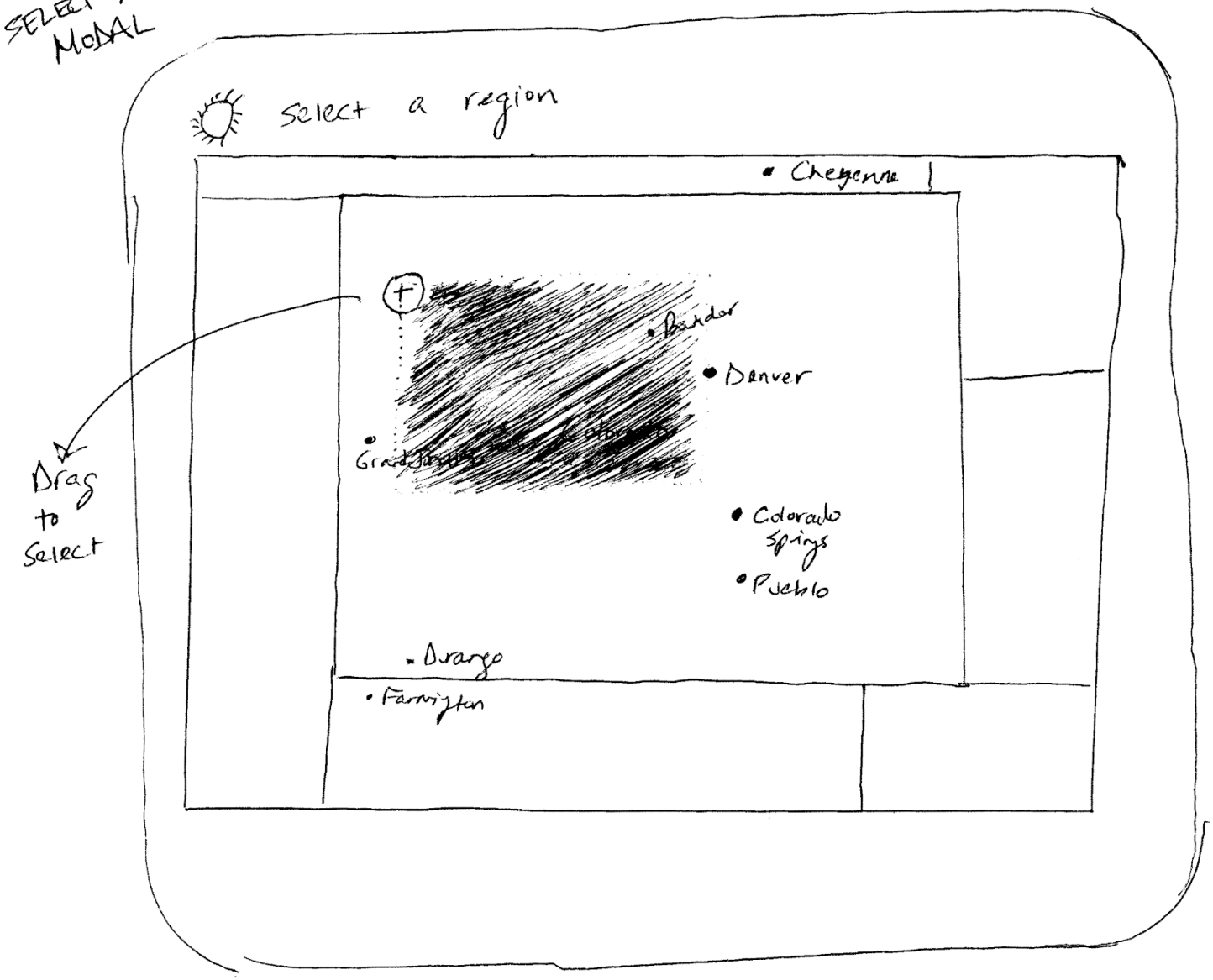
mm/dd/yy to mm/dd/yy

display specific dates

weather index

10 30 50 70 90 110

SELECT A REGION MODAL



Tasks

1. Develop a JSON structure for representing weather data. It should be easy to filter out weather data for a particular year, and then by city, so something like this

   ```
   {"2000": {"San Francisco": {/* data */}}}
   ```

   would be reasonable.

2. Get some data.
   a. Ideally one or two cities at first, with a bunch of dates for each, to take full advantage of the JSON structure
3. Implement a "template" CSS/HTML page for the visualization.
   a. Whoever's doing this may want to use Bootstrap for the modal views.
   b. Need not be interactive or include maps - just the basic structure.
4. Implement basic components of the visualization
   a. Start with DataMaps, and the weather for one or two cities.
   b. Once DataMaps is fully functional, try Google Maps. (This could almost be done independently of 4a)
5. Add the key deliverable interactive features:
   a. Brushing
   b. Linking
   c. Cross filter
6. (In the mean time) Get more data
   a. It should be in the format decided by the person who does #1.
7. (Always) Document things in the process book

A JavaScript Interface for Retrieving Map Data

```
function filterByYear(year)
{
     // filterByYear allows for the retrieval of all weather data for
        a particular year.

     // It returns an array, where each element corresponds to a
        specific city.  Each element is structured in the following
        way:

        {
          "name": "San Francisco",
```

```
            "state": "CA",
            "data": {
                  // temperature data here
            }
          }

      // If the year is invalid, it throws an exception.
      // If the year is valid, but no data is located, it returns
          the empty array.

}

function filterByCity(city, state)
{
      // filterByCity allows for the retrieval of all historical
         weather data for a given city.

      // It returns an array, where each element is structured in the
         following way:

         {
            "year": 2000,
            "data": {
                  // temperature data here
            }
         }

      // If no city is given and the function is called, an
         exception is thrown.

      // If a city is specified but not found, an empty array is
         returned.

      // Same behavior with the state argument.
}
```

Scraping

Kevin

Today, I started to scrape data for usage in our visualization. Early into the process, I realized something: the Wunderground API we decided to use had a rate limit that was really going to

interfere with our scraping. We were limited to making 500 API calls per day. On a single API call, the most information we could get was the weather for a single city on a particular day. The API did not offer the averages, either. That's a lot of API calls that would be necessary!

I contacted Basil, and we began exploring other options. We considered sites like:

- http://weather-warehouse.com/WeatherHistoryListing/monthlyWeatherDataStart.html
- http://www.nws.noaa.gov/pa/climate_data.php
- The list of sites at http://droughtmonitor.unl.edu/source.html

But none of them were very organized and cost-effective.

However, then we had an idea! We knew one place that offered this weather data - namely, Weather Underground. So maybe there was another way to access this data. Sure enough, there was! For a sample page, check out http://www.wunderground.com/history/airport/KBOS/2013/3/27/DailyHistory.html?req_city=Cambridge&req_state=MA&req_statename=Massachusetts.

So we have now modified our scraping data source - instead of using the WUnderground API, we will use the WUnderground website, and use Python's Pattern library to do the scraper.

Regardless of how we carried out the scraping, I wanted to determine a reliable and consistent JSON representation of the scraped data, so that I could write the helper JavaScript functions that operate on the dataset independently (and test them on smaller datasets, etc.)

Here's the format that we came up with:

```
{
      "2010": [
            {
                  "city": "Cambridge",
                  "state": "MA",
                  "data": {
                        "temperature": "23.4"
                        /* other key-value pairs can go here */
                  }
            },
            {
                  "city": "North Babylon",
                  "state": "NY",
                  "data": ...
            }
      ]
```

```
}
```

There are a couple of things I would like to note about this specific representation of the weather data we scraped. First, that it is, at the top level, a JavaScript object whose keys are the years. (The corresponding values are arrays.) This was done deliberately, to make it easy to get all of the weather data for a given year. I specifically chose this organization (rather than organizing by city, then year) because I think that a common usage of our visualization is going to be to view the changes in weather data over time for a given map view.

As a negative, though, now, it makes gathering all of the data for a particular city somewhat inefficient, in that we have to loop through each of the years, and then perform linear search for the city. If this proves to be too slow in practice, I am thinking of some possible workarounds:

- Store a pre-processed copy of the data that was scraped, organized by city.
- During the initialization of the visualization, create a new version of the dataset that is organized by city.

Also, there is something of an asymmetry in the way we note the year (as a key of the object), and the way we note the city/state (a value). This was intentional, to avoid ambiguity in using the city name alone (i.e., it's possible for two states to have identically named cities), and also to avoid having the city and state combined. (We wanted to be able to access both separately.)

Scraping went pretty well overall. I started with the HW1 example to refresh myself on the usage of the Pattern library, and then wrote a scraper that could get one field from one specific website, just to make sure I was accessing the right DOM element on the page. Then, I factored that out into a function, and added more looping logic, so that it would proceed through a set of web pages!

To make the scraper more configurable, I have created a csv file (currently called `cities`) that holds a list of cities to scrape.

JavaScript Functionality for Accessing Data

Kevin

I worked on implementing the interface proposed earlier for retrieving map data. Given the way we structured the JSON data, this was pretty straightforward, and most of the implementations were done by accessing array elements in a loop, and comparing strings in the search part.

We also implemented a function called `getCities`, which returns an array of all of the cities in the dataset. This was useful later on for some tasks in the JavaScript code (specifically, implementing the `typeahead` functionality on a search box), so I added it to the interface.

```
function getCities() {
      // returns list of cities
}
```

Also, I required that clients of this API call `initWithData`, passing in the JSON data, before using the functions.

Google Maps API

Kevin

Now that I had some preliminary data, it was time to build some of the initial features of the visualization. I emailed our TF, Azalea, to ask about whether or not we could use the Google Maps API for one component of the visualization. This turned out to be a very helpful conversation about the various options for the map visuals (thanks!)

For now, we decided to start with Google Maps API for the homepage's visualization, and then use D3 to generate more complex visualizations for the individual city and "selection of cities" views.

The Google Maps API is very well documented. I started by looking at the simplest map example available in the documentation, to learn how to instantiate the map. Then, I worked with a simple example of a Bootstrap CSS layout (available in Bootstrap's documentation) and merged the two to create an initial page:

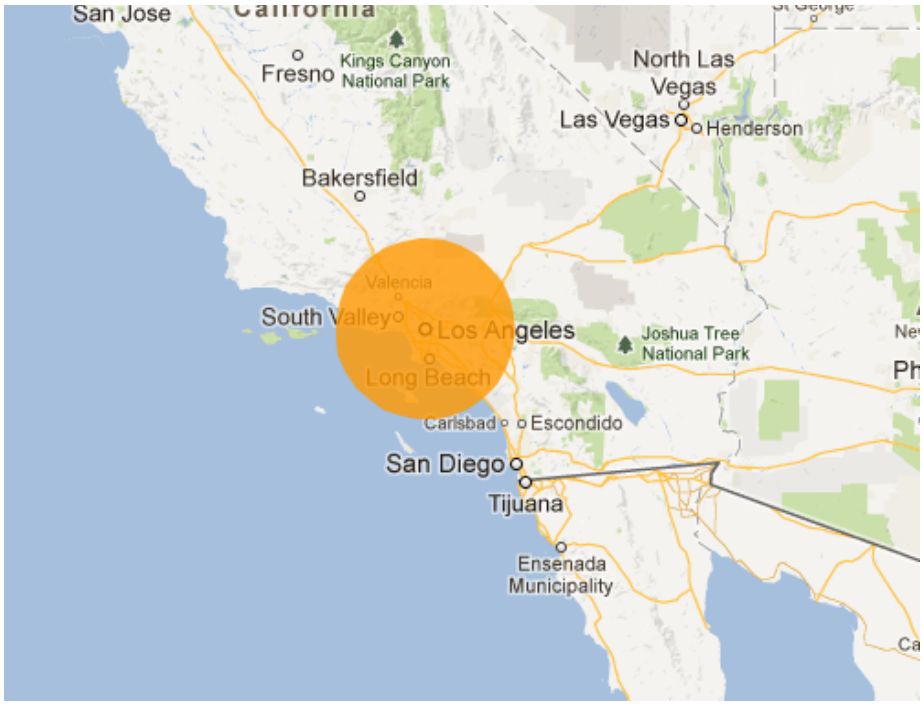
Looking at the documentation for Google Maps API's `MapOptions`, I learned how to configure the initial zoom level, center, type of map, and available of controls. Here are some choices we made with regard to this:

- The `center` is set to coordinates in the middle of the United States, since that's where all our data is from.
- `mapTypeId` is set to `google.maps.MapTypeId.ROADMAP`. We chose not to use `google.maps.MapTypeId.SATELLITE`, because we felt that with `.ROADMAP`, the range of colors that the map imagery would be limited, and we could make more informed color choices. Also, the `ROADMAP` type seems brighter and looks better.
- We decided to disable the `panControl`, the `streetViewControl`, the `overviewMapControl`, and the `mapTypeControl`. We felt that all of these were features that provide nice functionality, but are unnecessary and take the focus off of the visualization itself.

Now, it was time to add some data to the visualization. In doing this, I decided to explicitly avoid using the default markers that come with the Google Maps API. These don't really offer a great way to visually encode the notion of a "data point" and a lot of them at once can appear overwhelming. Even with custom icons. Instead, we noticed that Datamaps (one of the map

visualization technologies we were considering using and the one that HW 6 used) generates nice, clean, well-encoded visualizations using “bubbles.” So our objective was to replicate this concept in Google Maps. For that, the `Circles` class (and the corresponding `CircleOptions` class) came in handy.

Duplicating this concept was pretty straightforward - we wrote a `chooseColor` function that chooses a color for the circle based on a data value, and customized the circles so that they are somewhat transparent.



It’s unclear whether we would want to change this particular encoding (maybe add a border, or decrease the radius!) but we think that this is a good start.

Adding Controls to the Layout

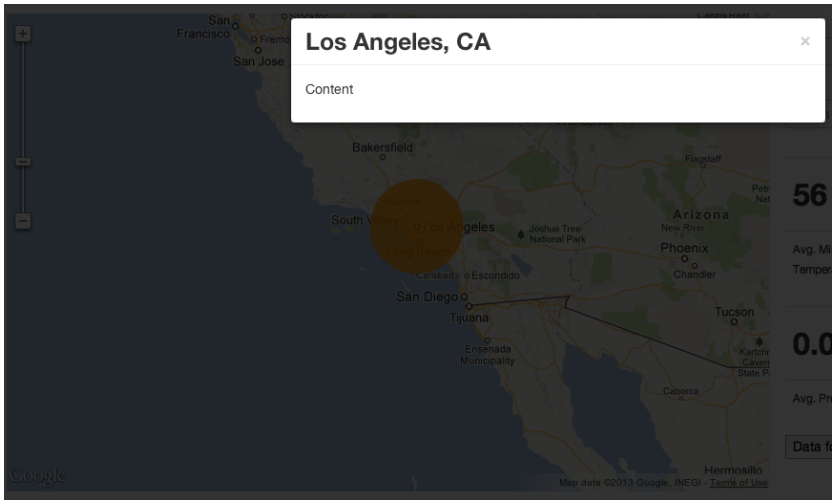
Kevin

Out of the box, the Google Maps API gave us an interactive, dynamic map that supports panning. We were also able to successfully add circles to the map to represent our particular data points. However, it’s hard to classify the visualization itself as “interactive” at this point - there’s no easy way to view data in more detail and filter. That’s what my next goal was in working with the visualization.

Our original vision of the visualization was such that, when a user clicked on one of the circles, a modal with more details about the particular city selected would appear. Inside this modal would

be a comprehensive set of visualization for that particular city's historical data.

Implementing this feature was pretty straightforward, thanks to `google.maps.event.addListener` and Bootstrap's modal functionality.



Above is the modal view: when you click on the orange click, the modal appears, shading everything and bringing the window into focus. In our final visualization, we intend to make the modal much larger and include actual visualizations!

Another interactive layer to our visualization was that our users would be able to select the year for which they wanted to view the weather data. In our original sketches, we represented this feature with a slider on the bottom on the map. For now, though, we're choosing to implement this feature with a text box, that can allow the user to just type in the new year value. This is because we'd like to get the visualization in a working state before we begin to use a fancy slider or something like that; the textbox is easy to setup and pretty well documented.

So we added a textbox. When the textbox is changed, an event listener gets called, which updates the map to have the appropriate data for the given year (using `filterByYear`). If no data is available, no data is shown.

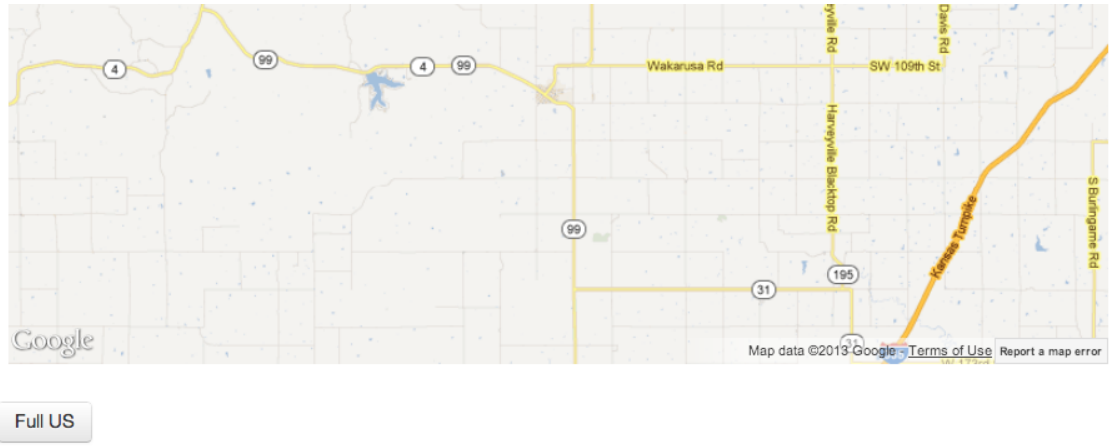
Similarly, we wanted users to be able to find a particular city on the map. Using the `getCities` function described earlier, we are able to obtain a list of the dataset's cities, which we used to fill a Bootstrap typeahead box with the list of the cities, so that they appear when the user types them. The typeahead documentation permits us to specify some callbacks (one in particular is called `updater` - it gets called when the user selects something), so we used the `updater` callback to adjust the zoom of the map.

This raises the question of "how does the user zoom back out?" since we disabled the zoom controls from the map. Here is what we have decided to do as an alternative:

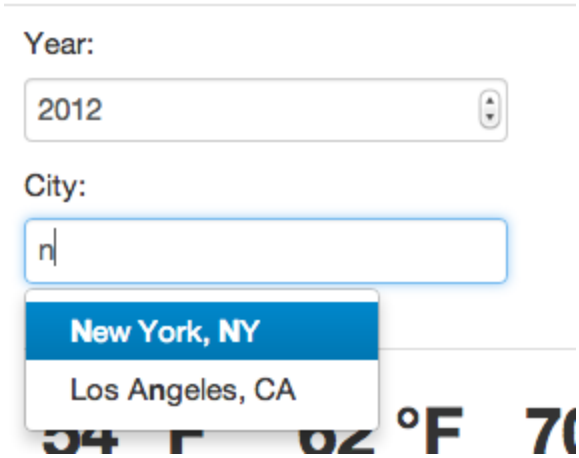
- When the zoom level is changed, and it is above a certain threshold, a button will appear (an event listener is responsible for doing this) that allows the user to restore the original

zoom mode, which is the United States.

- Outside of that threshold, no button will be shown.
- The user is still allowed to zoom in on the map by double clicking. We think that this is okay, since the button will appear in this case, too.



Notice the "Full US" button that appears when zoomed in! Currently, we've positioned it below the map.

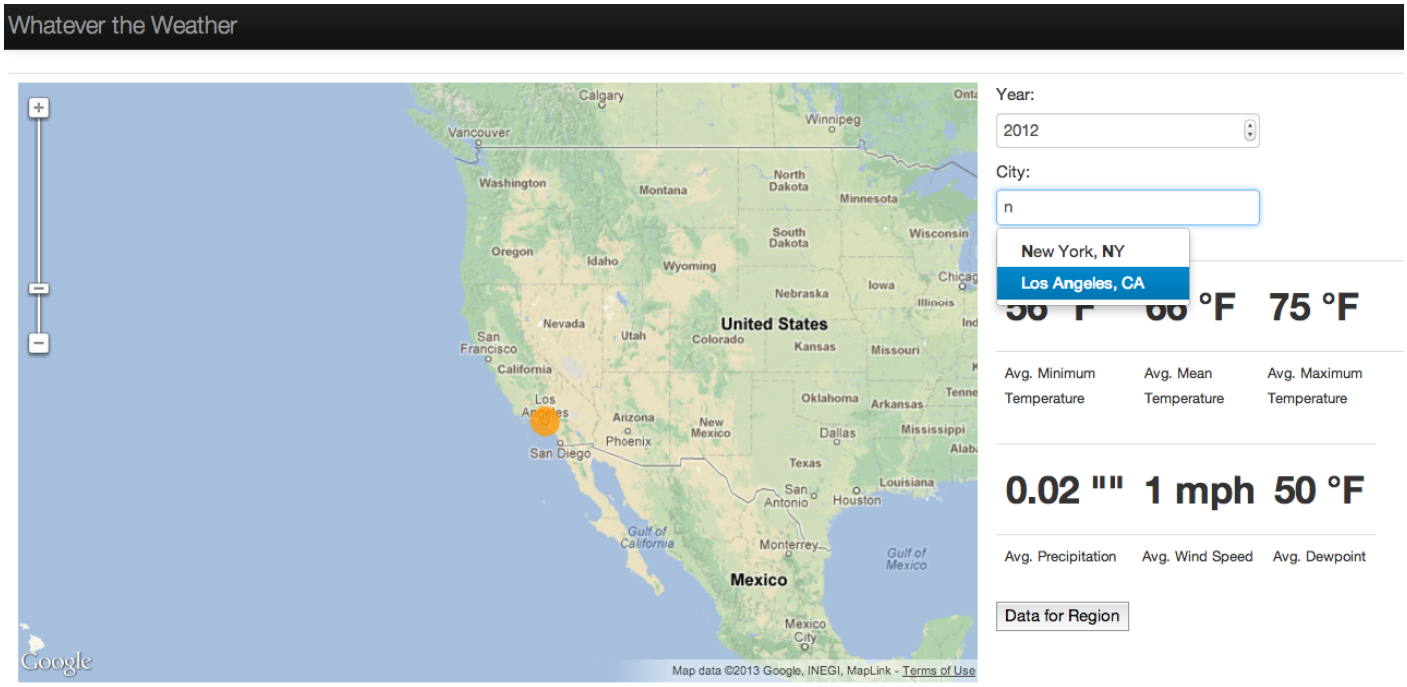


The map control pane, positioned to the right of the map itself.

Before we began to work with `d3` to generate per-city (or per-city-selection visualizations), we wanted to try out one more thing without `d3`. We were interested in showing some statistics about the user's current selection (average temperature of cities in the selection, say) on the sidebar with the control pane, that would be updated when the user panned around the map. We thought this would be neat, non-intrusive, and just a "cool" component of the visualization. We also felt that it would be a good starting point for implementing the per-city-selection visualizations, since those rely on being able to determine what is in the current map view.

Tools used to implement this:

- Our `filterByYear` function, to grab the selection of cities for the current year.
- The `getBounds` method on the Google Maps Map object.
- Event listeners.
- DOM updating (to push the newly computed statistics to the DOM).



The full map view. You can see a video of our map view (with panning in action) at http://www.youtube.com/watch?v=Cyrk7pdGNBk

Implementing a Line Graph with D3

The most intuitive visual encoding to display weather trends was a line chart. I decided to take a simple line chart example and modify it for our purposes. During the testing period I hard coded values and created a separate view.

I discussed possible implementation methods with Kevin and we decided to integrate the graph in a modal for each city. I refactored the code for the line graph and created our graphs.js file, a file for all our graphing functions.

```
function lineChart(city, state, datapoint){
      // draws D3 line chart for the given city and datapoint
}
```

Adding NVD3

While I was learning about D3 and working through some tutorials, I found a great graphing library built on D3 called NVD3. According to the NVD3 website's description:

> "This project is an attempt to build re-usable charts and chart components for d3.js without taking away the power that d3.js gives you. This is a very young collection of components, with the goal of keeping these components very customizeable, staying away from your standard cookie cutter solutions."

After looking through some of examples on the NVD3 website, I decided to add the library to our project for use with future encodings. All of the examples were documented well, had great interactivity, and were fully customizable.

Getting Data Ready for NVD3

To use the built-in graphs provided by NVD3, we had to reformat our JSON in NVD3's preferred structure:

```
[{
  "key" : "Cambridge",
  "values" : [
  ["2000", 32],
  ["2001", 37],
  ["2002", 29]
  ]
},
{
  "key" : "New York",
  "values" : [
  ["2000", 35],
  ["2001", 40],
  ["2002", 39]
  ]
}]
```

I wrote a simple function in graph.js for all our NVD3 graphing functions to use to get the proper JSON format:

```
function compareCitiesData(city1, state1, city2, state2, datapoint){
     // returns JSON of reformatted relevant data for two cities
}
```

Implementing a Stacked Area Chart / Compare Line Chart

Once I got the data in the proper format, I began testing the various graphs provided in the NVD3 library:

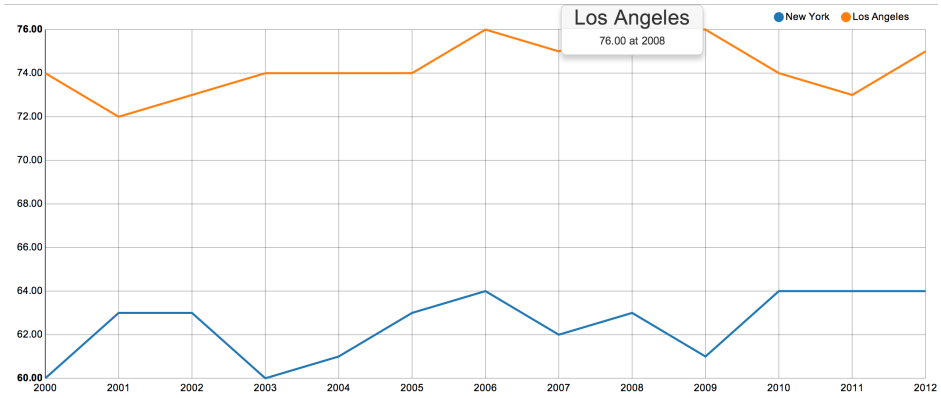
We met to discuss the graphs we would like to include in our comparison tool and ultimately decided to go with a Stacked Area Chart and Multiple Line Graph.

From there I wrote two functions in graph.js to simplify the creation of these graphs:

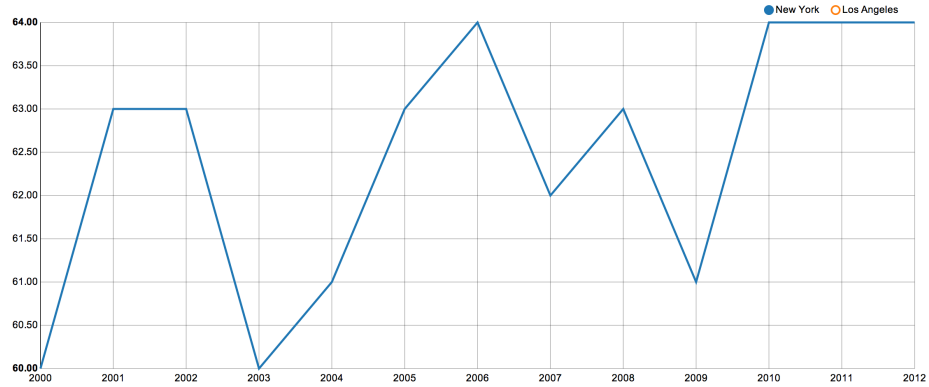
1) compareLineChart

```
function compareLineChart(city1, state1, city2, state2, datapoint){
      // draws a D3 a line chart of the two cities with interactivity
}
```

When you mouse over a given data point, a pop-up is displayed with information about the datapoint:



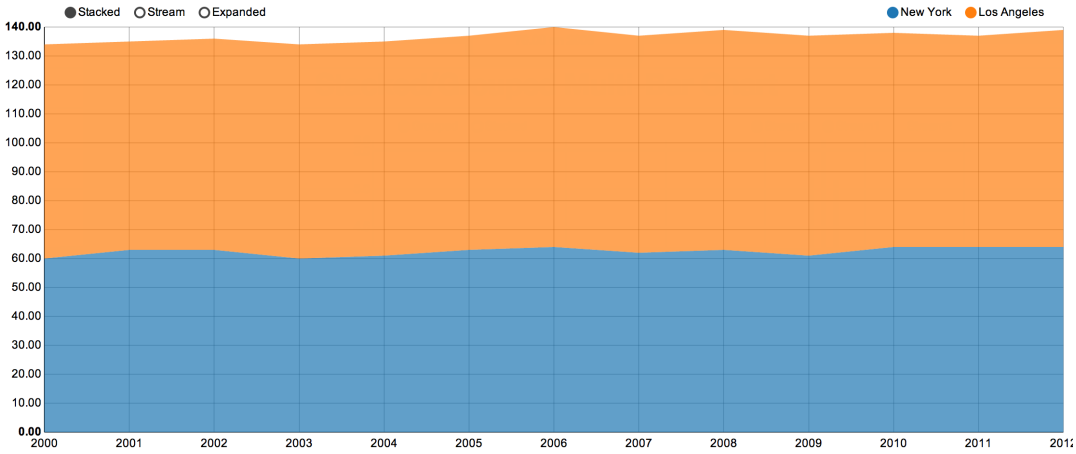
You can also choose to toggle which cities are displayed:



2) compareStackedAreaChart

```
function compareStackedAreaChart(city1, state1, city2, state2, datapoint){
      // draws a D3 stacked area chart of the two cities with interactivity
}
```

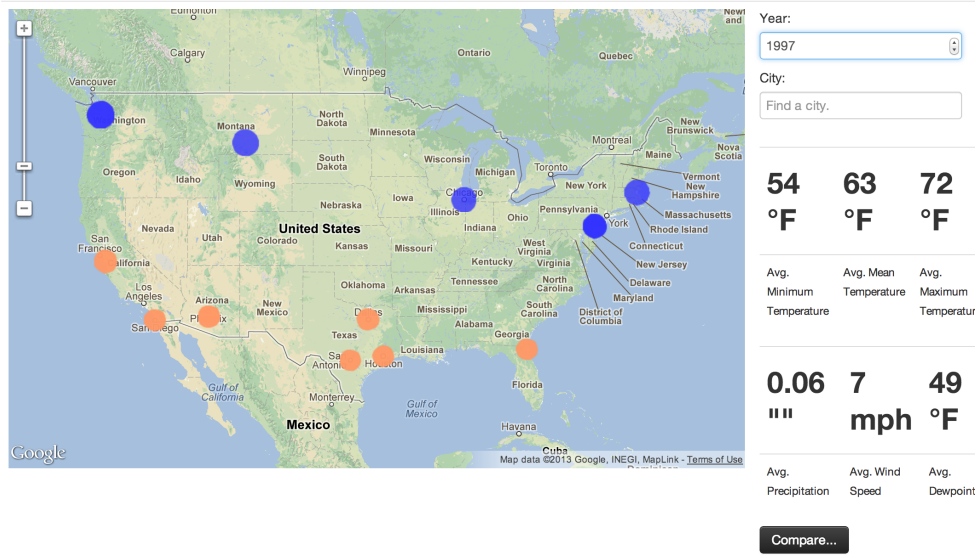
The Stacked Area Chart has the same mouseover and toggling features as the Multiple Line Chart:



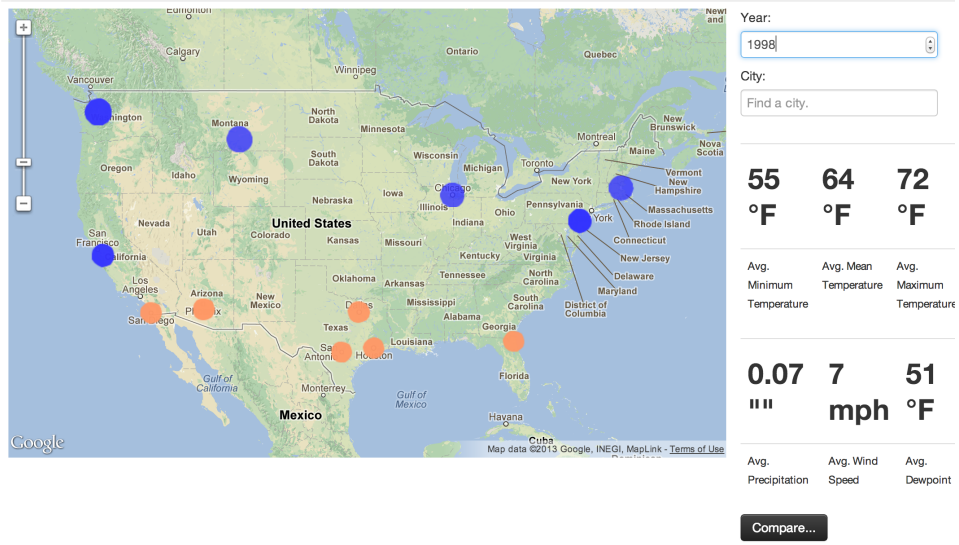
Global Warming and Beyond

Our visualization provides a method for us to track global warming trends in a few ways:

1) Search by year - users can see how weather trends have changed over time by clicking through the various years. If they find something interesting, they can further investigate. If, for example, a user was clicking from 1997-1998 they would see the following:



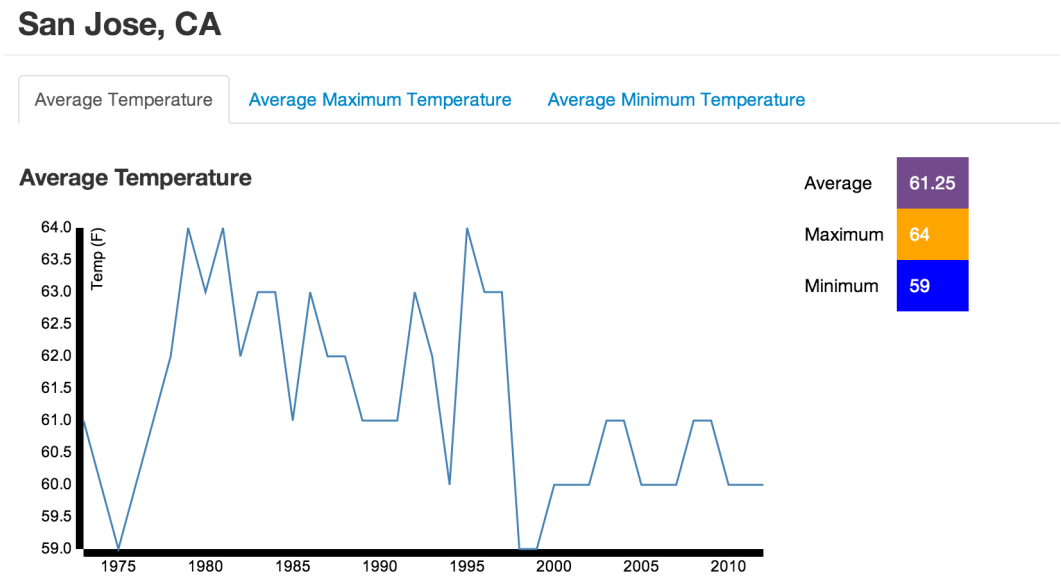
1997



1998

Just from looking at the map, they can see that some city in CA (San Jose, CA) has changed colors, implying a temperature drop. To further investigate, they can click on the bubble of interest to gather more data.

The line chart provided when click on a city bubble would let the user know if the change in temperature was an anomaly or an overarching trend.



Continuing with the example of San Jose, we can see that, starting around 1995, the weather began to drop and hasn't really come back up to its previous highs since then.

2) City comparison - a lot of the debate surrounding global warming has to do with the effect of development on the environment. Using our city comparison view, we can see how the weather trends of developmentally heavy cities compares to that of more rural cities.

Now that we have completed our data exploration and have a better idea of what visual encodings to use, what trends may exist, etc we can build on our foundation. Most of our current data is from larger US cities (by population) but we plan on incorporating data from a variety of cities. Also, we may include other datapoints that are relevant for understanding the effects of global warming. Data on precipitation, humidity, and event tags can all be incorporated into our existing foundation to take make our data visualization an even greater resource in the future for those interested in weather trends.

Reflections on Choice of Visual Encodings

To summarize, our visualization utilizes a variety of visual encodings to represent data:

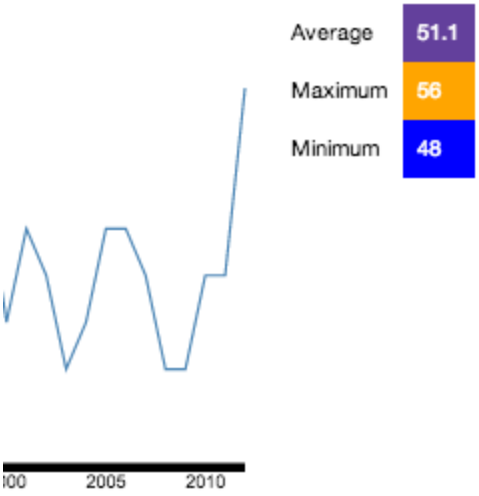
- In the overview map view, we use circles to show the average temperature of a city in a given year. Given the nature of the the data, which is quantitative in form, we felt that this would be an appropriate encoding. The color scale used is very coarse grained - this is also intentional, to allow the viewer to quickly discern which territories and cities are "cold" and which are "warm" without being overwhelmed by a myriad of colors. It also promotes, we feel, exploration on behalf of the user: if they can easily determine the temperature range of a particular city (or even region), they can further explore these

insights with the more fine grained tools and the line graphs.

- In the detailed view, we use a single series line graph to show the trends in weather data (average temperature, average maximum temperature, average minimum temperature... all per-year statistics) over time. We think this is a reasonable encoding for this kind of data; there is minimal chart junk using D3's line graph path generators, the contrast is strong enough, and the scale is automatically determined by D3's library functions, so it is a reasonable choice for the data.

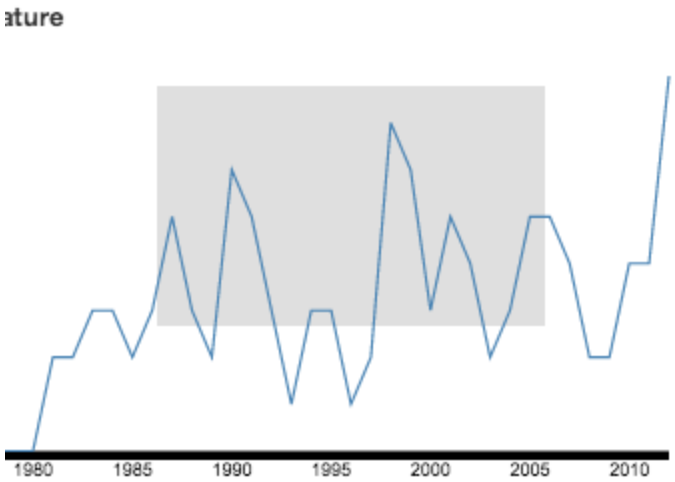
  The line graph also proved to be a good choice for permitting brushing: at first, we display approximately 40 years worth of data (40 data points) to the user. This may make it hard for the user to learn about a particular time period, since a span of 40 years can have a lot of ups and downs that may be interesting to explore in detail. To accommodate this, we've introduced the following:

  - **"Average" statistics shown on the side of the graph**. At first, these are just average statistics for the entire graph's worth of data. However, if you brush on the graph (implemented by using D3's brush functionality), you can make a selection on a particular region, and view the average temperature information for that particular region. (Then, also in tandem with D3's brushing functionality, you can pan the brush from side to side. Hopefully, this will allow users to explore change in a fixed window over time.)



    The average statistics, shown on the right hand side. The colors are chosen by D3, and range from blue to yellow, which we felt was an appropriate way to represent temperature data. We also think that keeping it off to the side like that allows it to not interfere with the line graph.

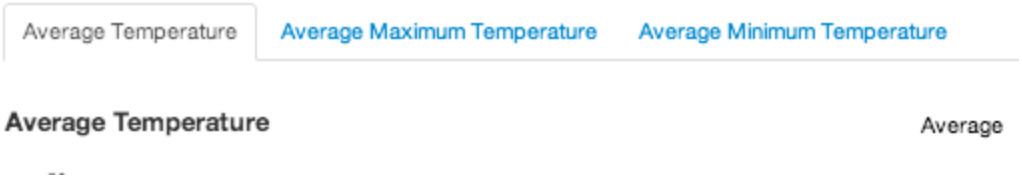
  - **Ability to focus on a selection**. When you brush on the graph, like this,



a button appears on the right hand pane of the modal window, called *Focus on Selection*. *Focus on Selection* allows the user to trim the dimensions of the graph to meet their needs. We hope that this will actually allow users to see a more fine grained trend at large. (At the same time, the user can restore the original view with another button, *Reset View*, that appears.)

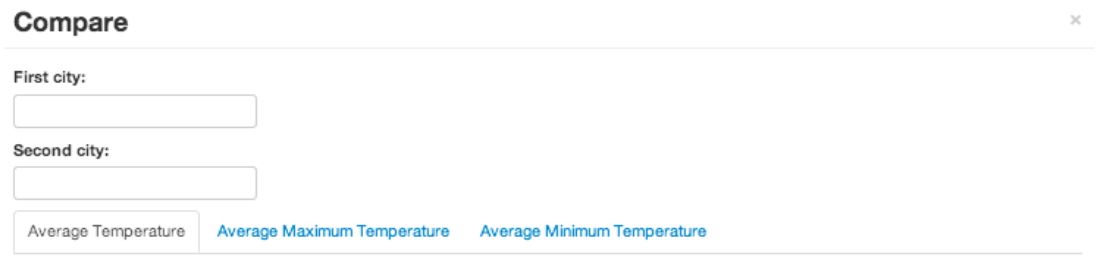
We think that this harnesses interactivity in an effective way; it's not overly "showy" or "gaudy," and it hopefully allows for realtime data analysis.

- In order to minimize the "busy-ness" of the modal visualization, we used Bootstrap's tabs to allow the user to hop from place to place:

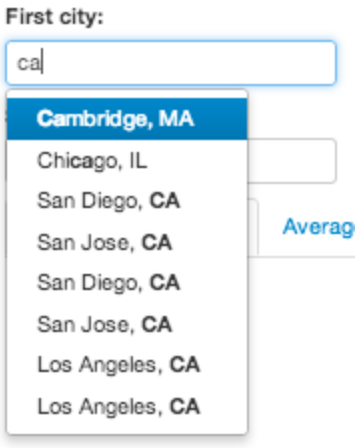


State is saved when you visit and return to and from each of the tabs. Each of the visualizations shown is a line graph.

- The other major component of our visualization is the Comparison Tool, which is a subtle but powerful feature, available in the bottom right hand corner of the main visualization. When you click on the *Compare...* button, a modal window will appear that will prompt the user to supply two or more cities:



Now, you can type two cities, using the familiar typeahead-powered interface we saw earlier in the main map visualization:



Upon typing the cities, you can now see the multiseries line chart and stacked area chart mentioned by Basil earlier! The interface allows the user to toggle between the two views.

We recognize that we offer two different visualizations for the same exact data, in almost the same exact form, but we think that is okay for a few reasons:

- There are benefits to each of the representations of the data. For example, the multiseries line graph enables tasks such as comparing the temperature of two cities at a particular point in time. Obviously, this is useful for answering questions like, “Which city was generally hotter in decade X?” and “Is city A generally hotter than city B?” However, we think that this is not the only question somebody might be wondering about the relative trends of two cities. For example, as silly as it sounds, the stacked area chart might be useful in seeing when both cities were collectively “hottest.” Of course, the stacked area chart has

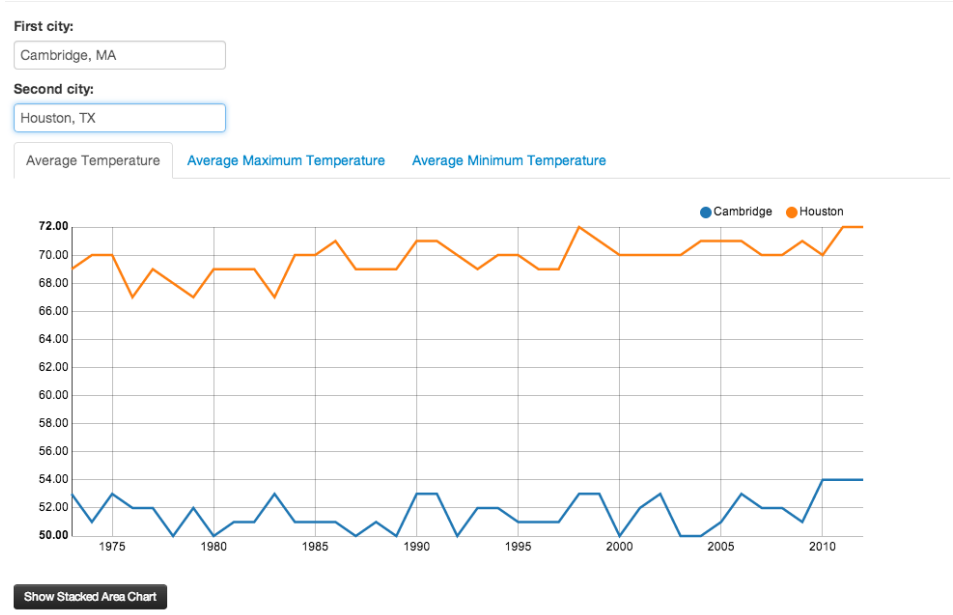
some disadvantages too, in that in would be more natural for data that is broken into a whole over time. But we thought we would include both kinds of visualizations, and allow the user to choose between the two.

So, what can you learn about global warming from this visualization?

1. The coarse-grained homepage view reveals that not much has *drastically* changed over the past forty years. If you adjust the Year field on the homepage, and glance at the map while adjusting it, you'll see that the colors remain relatively stable over time, with a few minor hue adjustments over time, but these are generally very rare.

   It's also mentioned that global warming is associated with an increase in preciptation (http://www.climatehotmap.org/global-warming-effects/rain-and-snow.html, http://thehill.com/blogs/e2-wire/e2-wire/291867-federal-study-global-warming-means-stronger-extreme-rains). But if you glance at the statistics on the right hand side, the average precipitation over time remains pretty stable over a period of forty years. We are aware of the possible misleading-ness associated with using an average to make this conclusion, but it still provides a decent estimate of the precipitation.

   If you're interested in a particular region's average daily precipitation, it's just a zoom away on our visualization, and so if you are skeptical about averages and want to check out a particular area, feel free to go ahead! :-) (Panning the map so that only the bottom portion is shown reveals a similar trend over time, although there is a spike in the mid-1980s, so this is important!!)
2. The more fine-grained views provide some insights too: for example, consider this view of Cambridge and Houston's average temperature over time (period from 1970-2010):



   Here are two cities, extremely disparate in terms of their climate, that show *fluctuation* in their average daily temperature per year.

The temperatures did ultimately increase, and that's certainly something that can't be ignored. But the increase was pretty smaller in both situations, and there was a lot of fluctuation in that time period between the start and end!

3. In some ways, our visualization doesn't answer *all* the questions one might have about global warming. "More extreme winters" (http://www.independent.co.uk/news/science/expect-more-extreme-winters-thanks-to-global-warming-say-scientists-2168418.html) and "more extreme summers" (http://www.nasa.gov/topics/earth/features/warming-links.html) are both attributed to global warming in reliable media outlets (NASA!!), but our visualization currently does not provide a way to compare two cities on a basis as fine-grained as a month (or even day.) As an alternative, we may consider providing the following:
    a. Standard deviation / variance for the temperature data, to show how much it has fluctuated within a year. Our data source (Wunderground) does not supply this information up front, but it does supply daily temperature data, so we could obtain the daily temperature data and then calculate it ourselves.
    b. Just, plainly put, the daily temperature data. Finding a way to appropriate encode daily temperature data is an interesting challenge, since there is a lot more of it this time around.
4. Allows the viewer to get a sense of variation in temperature data around the country in the year specified in the textbox. (Specify a year, and then zoom in and fly around!) This puts the "1 degree increase per year" claim (http://www.thedailygreen.com/living-green/definitions/global-warming) in perspective, somewhat.
5. It's also hard to draw conclusions from this temperature data alone, especially in cases where the fluctuations are so small, because the error range on equipment has changed over a period of 40 years.

Improvements for Project 3

While we are proud to submit this work for project 2, we can think of a few things we'd like to improve upon for project 3:

- Improve some of the user interface / interactivity, especially involved in brushing the line graph visualizations
- Add more fine-grained, monthly data to allow the comparisons mentioned in the global warming analysis section
- Consider switching to DataMaps to harness their bubble encodings (we used Google Maps for its ability to specifically plot a point at a given latitude and longitude)

Acknowledgements

We would like to thank...

- mbostock for his D3 examples, that we used
- The NV library, for its abstraction of D3 that permits the drawing of fluid multiseries and stacked area line graphs
- The Google Maps API

- Wunderground, for its historical data
- The Pattern scraping library
- John Mercer, for his inspirational Skype chat
- Our project TF, Azalea Vo, for her advice

Project III

In Project III, we will attempt to build on our Project II visualization by adding storytelling, more interactivity, more data, a different palette of visualizations, and repairing some bugs from the Project 2 submission. Here's a full list of things we intend to do.

Bugfixes

We intend to fix the following issues with our Project II submission:

- Line graph size
- The legend in the line graph moves to bottom sometimes (just regular line chart)
- On the stacked area chart, the color scheme changes.
- Can focus-in to the point of no data on the graph.
- It's **hard to distinguish between circles** in areas with concentrated circles (i.e. Texas). Perhaps this can be addressed by making the circles smaller, or making a zoom threshold, or replacing them with a different visual encoding completely.
- It's **hard to click on circles** that are covered by multiples other circles (for example, Arlington, TX).
- **Not all of the cities have data** (for example, Arlington, TX).
    - We will have to check WUnderground and see if the data is actually available there and fix the bug in our scraper. If it's not, though, we can try what Azalea suggested for these cities, which was to "grey" them out on the map.
- **If you compare cities and then uncheck the node in the tool-tip to hide the city, you can't recheck** the node to display the city again (line chart only).
- **Sometimes names of cities in the compare cities tooltip overlap each other.**

Features

We intend to add the following features to enhance our current visualization.

- We will **scrape more data**, and definitely include new cities in our visualization. Also, we will possibly scrape on a more fine grained scale (months, seasons). As discussed in our meeting, this may reveal more interesting and conclusive trends.
- We will avail the user to **graphs of other data points** - such as humidity or precipitation.

- We will change the interface slightly to **remove modals completely**, and integrate the visualizations on the map page.
- We will feature **better chronological browsing** or just a more i**ntuitive slider** that allows you to browse through the span of data that we have.
    - We discussed using an HTML 5 slider (“range” input type) at the meeting.
- We will try to improve the map component. A few things:
    - Use a more fine grained color encoding - the current one divides temperatures into 20 degree buckets!
    - Use smaller circles, and also consider other markers
- We will enhance the comparison tool - perhaps allowing one to compare 2 or more cities.
- We will consider **replacing the stacked area chart** with a more relevant visual encoding
    - Potentially a color-coded, searchable data table (like those generated by Tableau)
    - Would use d3’s color scale
- Change layout - consider changing modal so we can see changes

Proposed scope of storytelling:
For the storytelling aspect of Project III, we will add a feature to our web visualization that allows the user to “play” a chronology of weather data. The user can press a button to automate scrolling through the weather data within a given time period.

This feature could allow one to tell a story kind of like this: http://vimeo.com/27376376

Here’s a possible view for the slider, made in Microsoft PowerPoint.

Proposed implementation plan:

- For the enhanced visualization, we plan to use D3 strictly, instead of an external library like we used to generate the stacked area chart.
- Otherwise, we will continue using the set of technologies we’ve used.
- We may need to write some new scraping code if we consider adding population and other environment-based events.
- We may need to use more advanced CSS or JQuery UI if we consider adding a more aesthetically appealing slider.
    - For the play feature, we would look at JavaScript’s `window.setTimeout` to update the position of the time data incrementally
- To implement the more fine-grained color encoding, we would consider using d3’s color scale libraries

- Scraping more fine-grained data would be a matter of slightly tweaking our scraper code.

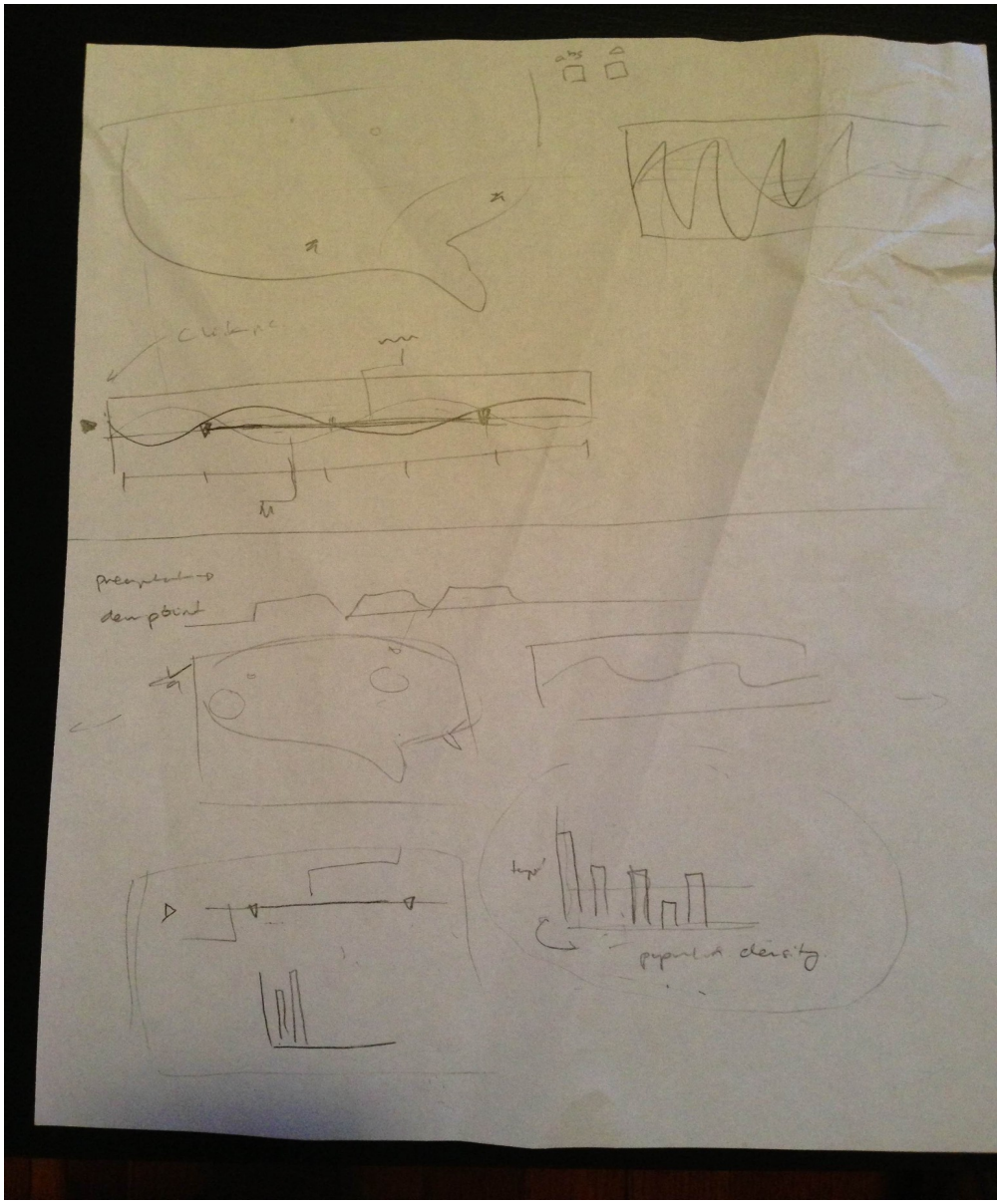
Proposed timeline for implementation:

Improvement:	Date:
fix Project II bugs / start scraping more data	ASAP (since scraping took so long last time) / By Sunday, April 21
tweak interface to avoid modals	Tuesday, April 23
add graphs of other data points	“”
better color encodings	“”
better visual encoding	Wednesday, April 24
play feature / better sliding	Friday, April 25
census data	Saturday - Sunday April 26 / 27
project due	Thursday, May 2

TF Meeting

Here are Azalea’s priorities for our project III improvements and additions.:

- Get more data
- Add 2 new linked views
    - Show data in map in a more quantitative way
- Implement timeline
- Storytelling - guiding user through important findings



The Slider

One of the features of Project 3 that we're most excited to create is the **slider**, which will serve two purposes:

- Allow a user of the visualization to browse the history of the weather for a given map view
- Allow a user of the visualization to play a "time lapse" view of the weather map
    - Given some start and end dates, the weather map automatically advances one month at a time when the user hits play.

We hope that introducing this feature will, first of all, better enable the task of looking at the

weather for a given year - currently, a textbox (with a restriction to allow only numbers) is used, and this is not as intuitive or pretty as we would like.

Additionally, we hope that the time lapse provides the visualization with a storytelling feature - that is, a way to make the data truly come alive. We hope that, coupled with the interface redesign, the user can really start to observe trends in the data and form their own conclusions and test their own hypotheses about the weather.

At first, we considered using an HTML input for the slider. One of the attributes of the `input` tag, the `type`, allows one to specify that the input will allow the user to select from a range - which, in our case, meant that the user could select a date from a range of years.

Here's an example of how to use the HTML5 slider. I created a JSFiddle to demonstrate this with the following:

HTML

```
<input id="someSlider" type="range" min="1980" max="2000"></input>
<br>
Slider value:
<div id="sliderVal">
</div>
```

Notice that you can specify a `min` and a `max`.

JS

```
$("#someSlider").change(function() {
    $("#sliderVal").html($(this).val());
});
```

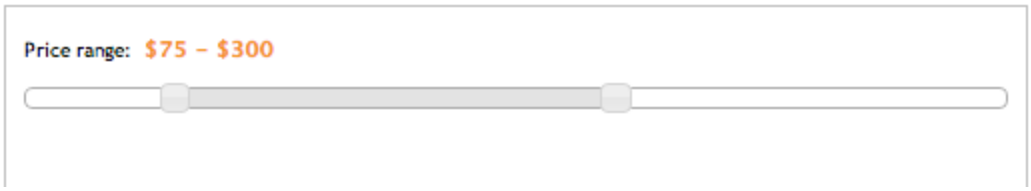
This code adds an event listener to the slider's `onchange` event, which updates a `div` on the page to contain the value of the slider.

View

Slider value:
1993

That is what the page looks like. The year at the bottom gets updated when the user changes the slider.

On one hand, it is very nice that this functionality is available in HTML5 without need for external libraries. However, the slider itself is not readily customizable: to implement storytelling, we'll need to have two pieces on the slider to allow the user to select a range. According to StackOverflow posts like http://stackoverflow.com/questions/4753946/html5-form-slider-with-two-inputs-possible and http://stackoverflow.com/questions/10015455/html5-slider-input-have-range (ignore the wrong answer given), this cannot be done easily. So, UI libraries to the rescue! I found that JQuery UI offers a wonderful range slider plugin. Here are some screenshots:



You can specify that the slider will allow you to give a range (two points), or a single point from a range (one point). You can also specify the step!

This was perfect; my only concern is that the JQuery UI CSS styles (and possibly, its JavaScript extensions) would interfere with the Bootstrap CSS and JS plugins.

Luckily, it did not! Adding the slider to the page was as easy as including a new script file and stylesheet. The slider can be triggered to appear with JavaScript; I made two functions, one called `initializeSlider` and another called `initializeRangeSlider`. `initializerSlider` is a function that prepares a single point slider; it's called at the beginning and when the user switches from the two point slider, which can be triggered and set up by `initializeRangeSlider`. Factoring these out like this proved to be beneficial since the code was removed multiple times.

Implementing the single point slider was pretty straightforward, since we already had a function that could update the map to reflect the data we gathered for a given year. I just bound an event listening function that called this function. The event listening function also adjusted the text on the visualization that dispalyed the current date.

The two-point slider was an interesting challenge to tackle, too. Luckily, setting up the two-point slider was a cinch thanks to the JQuery documentation. Then, implementing the "play" feature was reasonably straightfoward: I just used JavaScript's `window.setInternval` (https://developer.mozilla.org/en-US/docs/DOM/window.setInterval) to automatically advance the slider every second by a month. Then, I called my event handler that updates the map and everything, and voila! This was a very exciting milestone to achieve. Some limitations of the current implementation, though, include:

- While advancing from one month to another seems to be automated properly, this general process causes the map to flicker. This is not a bug - it's expected behavior, since we need to remove the circles from the map and then add them back, but it's not

very aesthetically appealing. I looked online from some ways to make this transition cleaner, and I found a StackOverflow post that looks like it will help: http://stackoverflow.com/questions/14052580/how-do-i-fade-out-a-circle-in-a-google-map-x-seconds-after-ive-added-it-to-the.

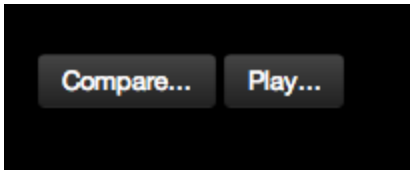
The accepted answer there provides a function that allows you to fade out or fade in a circle on a Google Map.

I will talk to Basil about what he thinks since he is much better at interface design than I am.

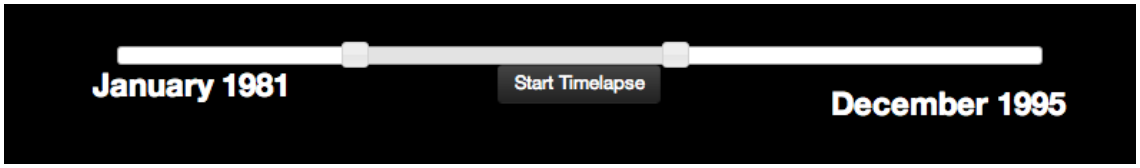
- We don't have month data, so currently, given a request for month *M* and year *Y*, our data source interface just returns the data for the year. Basil is currently working on scraping the data, and we hope to integrate this soon.
- We need to think about what parts of the interface we should disable and which parts of the interface we should let the user still use when the timelapse is playing. I think that the user should be able to still pan around in the map and pretty much do everything else, maybe except for launching the Comparison modal (from project II).

Here is a walkthrough of the current process for playing a timelapse.

1) First, click on the play button, which is located on the right pane of the visualization.



2) The slider on the bottom will change into a two point slider, ready for you to manipulate:



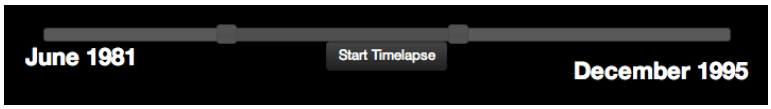
The beginning and end month and years are shown on the left and right hand sides respectively. It is important to note that hopefully, in our final Project III submission, I will learn CSS enough to fix the horrible alignment exhibited above.

3) Click the awkwardly-positioned "Start Timelapse" button.

This is actually a very significant button, and it ought to be placed in a better position than it currently is, and it possibly should be identified with an icon (like a “play” icon or something) rather than text. You can see its current position relative to the timeline in the preceding step.

On another note, since we changed the background to black, we might want to consider changing the color of the buttons :-).

4) The slider will fade to grey, making it inactive. The map will begin to advance at a rate of one month per second, and you will see the updated weather data for each advancement of the slider.



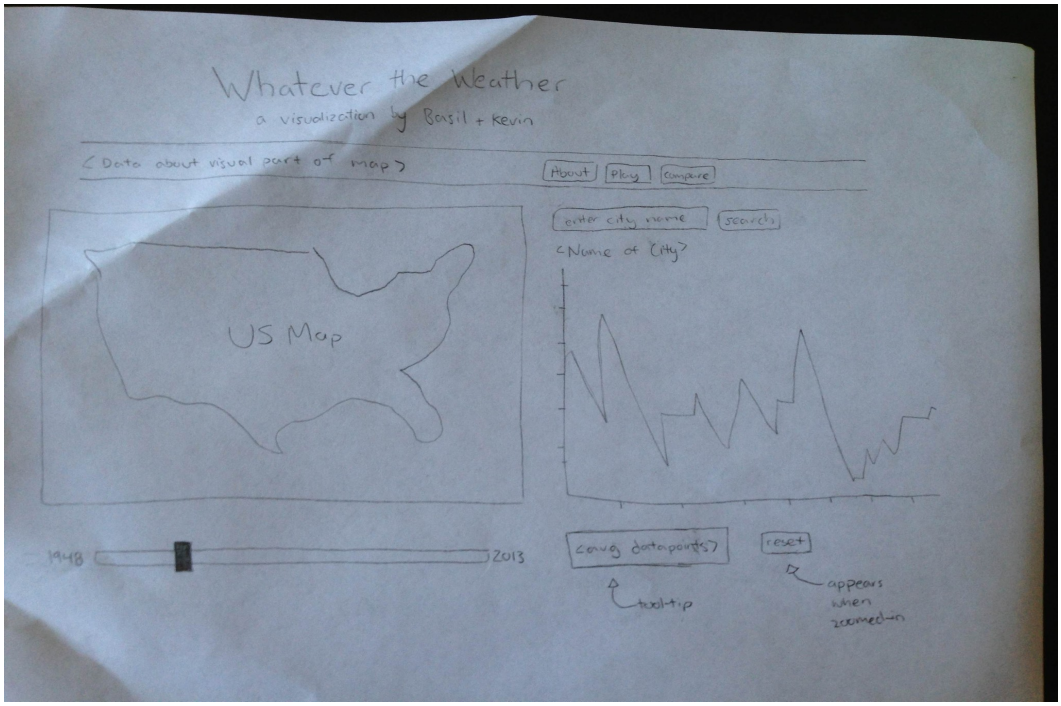
The slightly greyed-out slider bar, 5 seconds later.

TODOs for the slider

- Talk to Basil about the interface and the positioning of the associated buttons. Is it intuitive enough? Stylish? Etc.
- Add a speed-up feature and slow-down feature, perhaps. Maybe hitting a key on the keyboard, etc. Need to provide user-friendly instructions for this that don’t invade the visualization with text.
- Work on the map flickering (this is not a timelapse issue, but more of a general slider issue.)

Interface Sketch

Before redesigning the interface, we drew the following sketch:



The major changes to the interface is the side-by-side view (instead of modal) to display the line chart. After we implement this layout we will re-assess to see if we want to stick with it or try something else out.

The Interface Re-design

One of the things that I've noticed about nice web visualizations, like the ones posted on the class Piazza thread for these, is that they have nice title art. Since we're using Bootstrap, we have been working with the default black navigation bar. I wanted to change this as part of our interface re-design.

Around the same time, I also discussed over GChat with Basil the possibility of changing the background color of the visualization from white to black. My motivation for doing this was not very well-grounded in design principles or anything; I really just wanted to try it out. I also felt that changing the background color would give the entire visualization a new feel. He agreed that we could give it a shot, and so now I started to design title art with a black background in mind.

Here are some of the iterations of logo choices that I went through. I made these in Microsoft PowerPoint using WordArt:

I wasn't crazy about the font choice when this one was added to the main visualization page; it didn't seem to go well with Arial, which is what Bootstrap uses.

As a result I changed it to Arial. Basil and I both agreed that the gradient and the color weren't strong suits of this one, so...

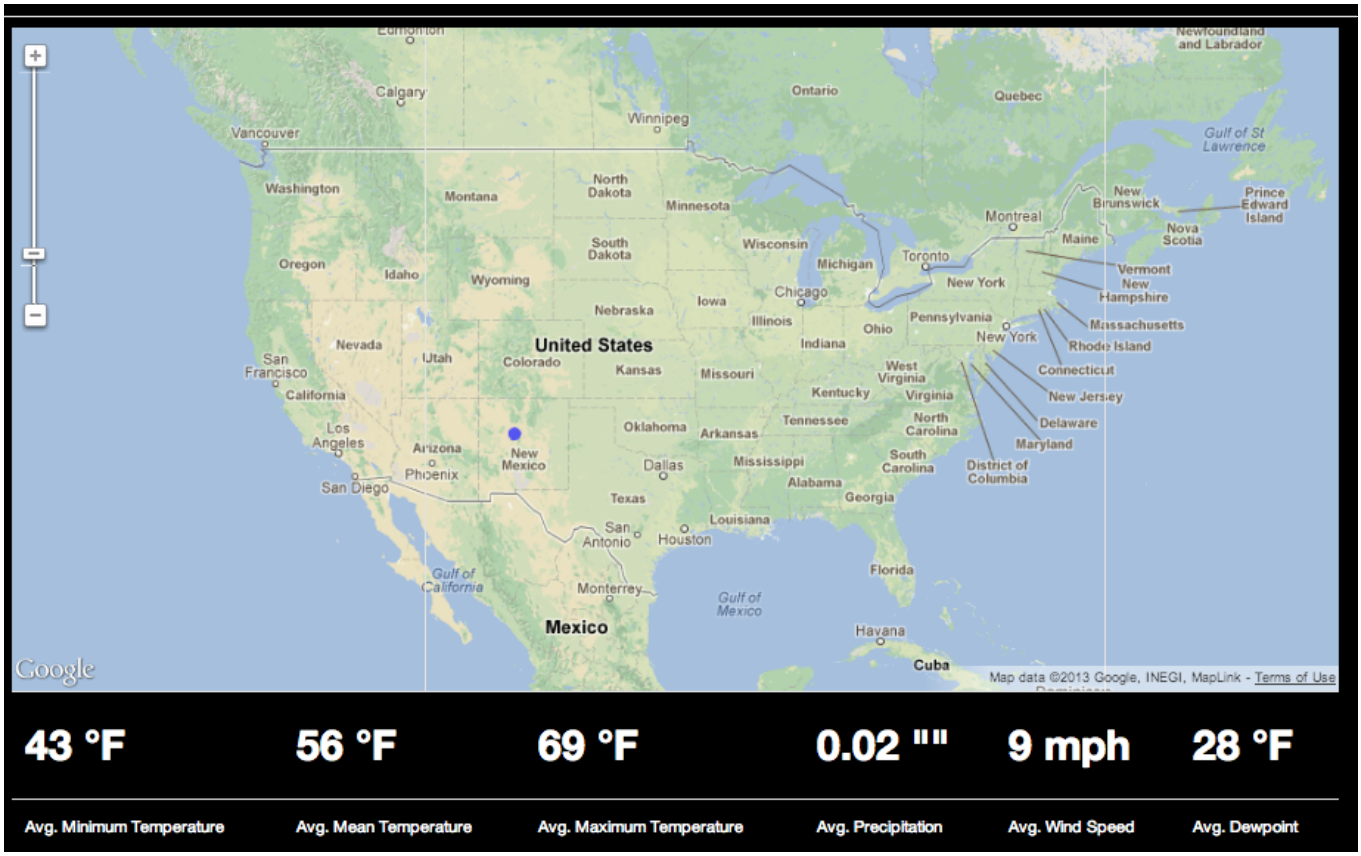
… is what I tried next. This might be a little bland. We'll think about tweaking it in the future.

As discussed in our project meeting last week, one of our objectives for Project III was to limit the usage of modals, and try to consolidate (neatly, of course!) the visualizations onto one main view. I think this would make the visualization feel more integrated.

I started by removing both modals for the single city (average temperature, average max temperature, etc. line graphs) and the compare view (two cities - stacked area and line graphs). Of course, this begs the question of, "Where in the main view should we add these?" For the single city view, that's somewhat straightforward - since the single city in question is tied very strongly to the map view selection, we could probably sketch out a few ways to integrate this chart somewhere in the view, after shuffling a few elements around.

However, the compare view is an entirely separate utility. It doesn't *really* depend on the map selection or anything like that - the user just inputs two cities in some autocomplete search boxes and views the graph. For this reason, we decided to actually leave the compare view inside of the modal.

We still needed to figure out where we could stash the single city view. On our current visualization at the time, we had aggregate data displayed on the right hand side. We decided to experiment with one approach for getting some space back (at least vertically): move that data off to directly below, and make it stretch horizontally instead of vertically. After a lot of CSS tweaking, it came out like this:



I have to talk to Basil about this (and we have to flesh out the rest of the main view together), but I think that I really like the feel of this - it reminds me of the speed data in a car, stretching right below the windshield. I think that the spacing is clean and it gives the user some useful information without totally consuming the space, which is now really precious.

Scraping New Data

As we mentioned earlier in the Process Book and in our meeting, we ran into some trouble with the data scraping for Project II:

- A few cities didn't have any data or were missing data for certain years
- All of the cities for which we had data were high population cities
- All of the data was averaged over the year (eliminating some of the variations over time)
- Collecting data was difficult because of scraping limits, URL timeouts, and other errors
- Data collection was painfully slow

With this in mind, we decided to update our list of cities and our scraper.

First, I added ~50 cities with lower population to our Cities CSV file and then I went to work on the scraper:

1. Error handling: Originally we were using a "`try, except URLTimeout`" block to fetch the weather data to handle URL timeouts (the most common error). I found that we would occasionally get a different error so I changed to to be "`try, except URLTimeout, except`" so that we would have a catch-all to continue scraping in the case of a random error.
2. Refactoring the data fetch for monthly data: I changed the urlForAirportAndYear function and all the function calls to also take a month parameter and to fetch data from the "Monthly History" section of the Wunderground website. Also, I added a level to our for loop that fetches data to go through each month.
3. Refactoring the data format for monthly data: I changed the format of the data JSON to incorporate monthly data

```
{
    "1948": [
        {
            "Salem": {
                "city": "Salem",
                "state": "OR",
                "data": {
                    "avg_wind": "8",
                    "avg_precipitation": "0.00",
                    "avg_max_temp": "48",
                    "avg_temp": "40",
                    "avg_min_temp": "32",
                    "avg_sea_level_pressure": "30.23",
                    "avg_gust_wind": "",
                    "avg_dew_point": "32"
                },
                "month": "1"
            }
        }
    ]
}
```

I started to run the scraper and it wasn't failing with errors but there were several cities where the scraper was failing to get data (especially in the earlier years, i.e. 1948-1990). After looking at the web pages for those cities and consulting our TF, Azalea, I realized that our scraper was getting an empty array back from some of the cities that had missing data, causing an `IndexError`. The reasons why we were getting an empty array back was because our DOM scraping method used bold class to pull data:

```
avg_gust_wind =
dom.by_class("contentData")[0].by_tag("table")[0].by_tag("tr")[1
6].by_class("b")[1].content
```

For pages that didn't have some data (i.e. some datapoint, like avg_gust_wind, was missing) an empty array would be returned because there was nothing at `("b")[1].content`, causing us to lose all data points from that page. To resolve the issue, I modified the scraping of each datapoint to use an "`if, else`" block:

```
avg_gust_wind =
dom.by_class("contentData")[0].by_tag("table")[0].by_tag("tr")[1
6].by_class("b")
if avg_gust_wind:
     avg_gust_wind = avg_gust_wind[1].content
else:
     avg_gust_wind = ""
```

So, this block would try and get datapoint but if the datapoint pull returns an empty array, it would assign an empty value to that datapoint. This prevented the `IndexError`, allowing us the data pull on the page to complete.

Once the scraper was fully functional, the only left to do was to run it. Knowing Project II that the scraping process takes a long time, I broke the cities up and ran several data pulls concurrently. The results from each of the data pull can be found in the "data/dataPulls" folder. Finally, once all the dataPulls were complete, I wrote a javascript function (located in "html/js/dataCombine.js") to combine the various dataPulls into one, final, 10 MB JSON.

Refactoring for the New Data Format

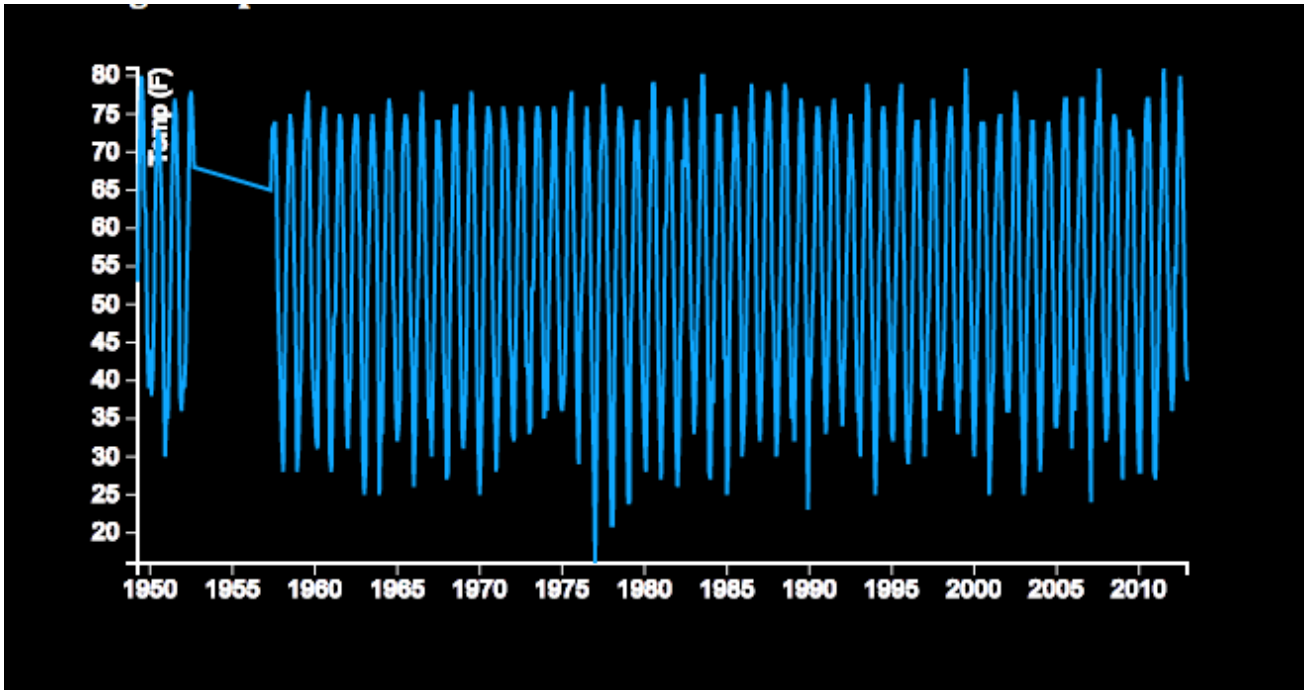
Integrating the new scraped data into our codebase was an interesting design challenge, namely because:

- The JSON data was in a different format than it was before, since it was organized by month and year.
- Modifying our codebase to work with this new data proved to test our interface that we wrote for Project 2, and we gleaned a few design lessons in the progress.

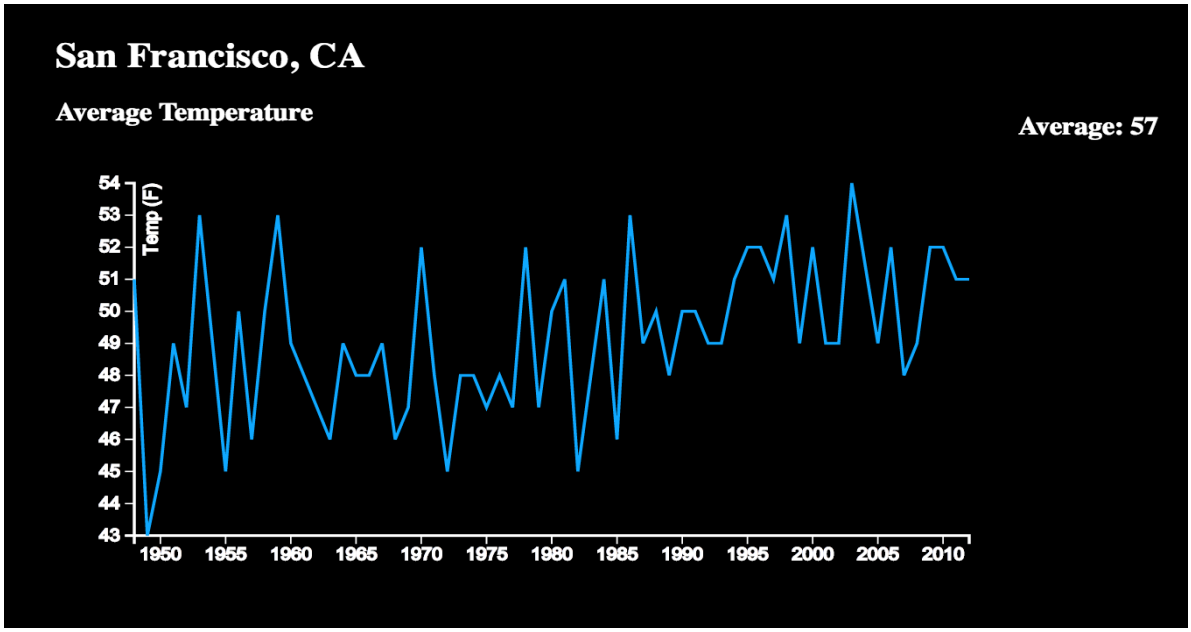
Design Decisions

As we begun restructuring our visualization with monthly data, there were a lot of design decisions we had to make.

1. Our line chart looked erratic:



Because we had added much more data (data for each month instead of getting only data from January each year) the line chart shows very high highs and very low lows; something we should have expected from seasonal data. To resolve this issue, we decided to add a month drop-down. The data for the map would show only data from that month each year, ensuring that relevant trends could be seen. The end result was this:



2. There was a lack of coherence between the data displayed on the map and the chart on the right.

A user could, for example, be looking at a line chart of New York, NY (which appears when user clicks on a bubble) while looking at a zoomed-in map view of Los Angeles, CA (which appears when a user uses the search bar).

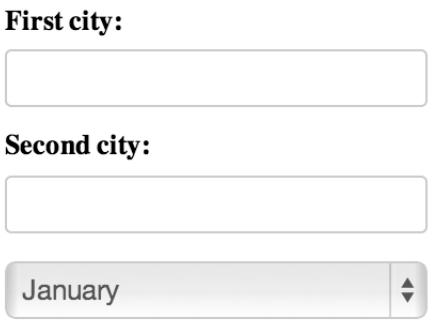
We ultimately decided to tie it all together to ensure that the user has a seamless experience search for data. If they search for San Francisco, CA, the map zooms-in and the line chart also switches.

3. The Data Stream feature didn't reveal anything interesting.

We found that streaming all the data simply showed that weather gets cold in winter and warms up in summer; a fact we hardly needed a visualization to showcase. To solve this problem, we decided to only stream data by month. If, for example, presses play with the slider on January, the player would only stream data from January. After implementing this change, we were able to begin seeing changes in weather patterns over time.

4. Our Line Chart in the Comparison Tool displayed the same erratic data the other Line Chart originally was showing.

This was a slightly easier fix because we already knew that the best design decision was to show only data for one month. To fix this we added a simple month drop down in the Comparison Tool, like this:



Enhancing our Color Scale

Our map in Project II had a color scale that prevented us from seeing most variations because each level of the scale spanned ~20 degrees. We originally had planned to use D3 Color Scales to revamp our map. Upon further investigation, we found the following options:

Categorical Colors

d3.scale.**category10**()

Constructs a new ordinal scale with a range of ten categorical colors: **#1f77b4** **#ff7f0e** **#2ca02c** **#d62728** **#9467bd** **#8c564b** **#e377c2** **#7f7f7f** **#bcbd22** **#17becf**.

d3.scale.**category20**()

Constructs a new ordinal scale with a range of twenty categorical colors: **#1f77b4** **#aec7e8** **#ff7f0e** **#ffbb78** **#2ca02c** **#98df8a** **#d62728** **#ff9896** **#9467bd** **#c5b0d5** **#8c564b** **#c49c94** **#e377c2** **#f7b6d2** **#7f7f7f** **#c7c7c7** **#bcbd22** **#dbdb8d** **#17becf** **#9edae5**.

d3.scale.**category20b**()

Constructs a new ordinal scale with a range of twenty categorical colors: **#393b79** **#5254a3** **#6b6ecf** **#9c9ede** **#637939** **#8ca252** **#b5cf6b** **#cedb9c** **#8c6d31** **#bd9e39** **#e7ba52** **#e7cb94** **#843c39** **#ad494a** **#d6616b** **#e7969c** **#7b4173** **#a55194** **#ce6dbd** **#de9ed6**.

d3.scale.**category20c**()

Constructs a new ordinal scale with a range of twenty categorical colors: **#3182bd** **#6baed6** **#9ecae1** **#c6dbef** **#e6550d** **#fd8d3c** **#fdae6b** **#fdd0a2** **#31a354** **#74c476** **#a1d99b** **#c7e9c0** **#756bb1** **#9e9ac8** **#bcbddc** **#dadaeb** **#636363** **#969696** **#bdbdbd** **#d9d9d9**.

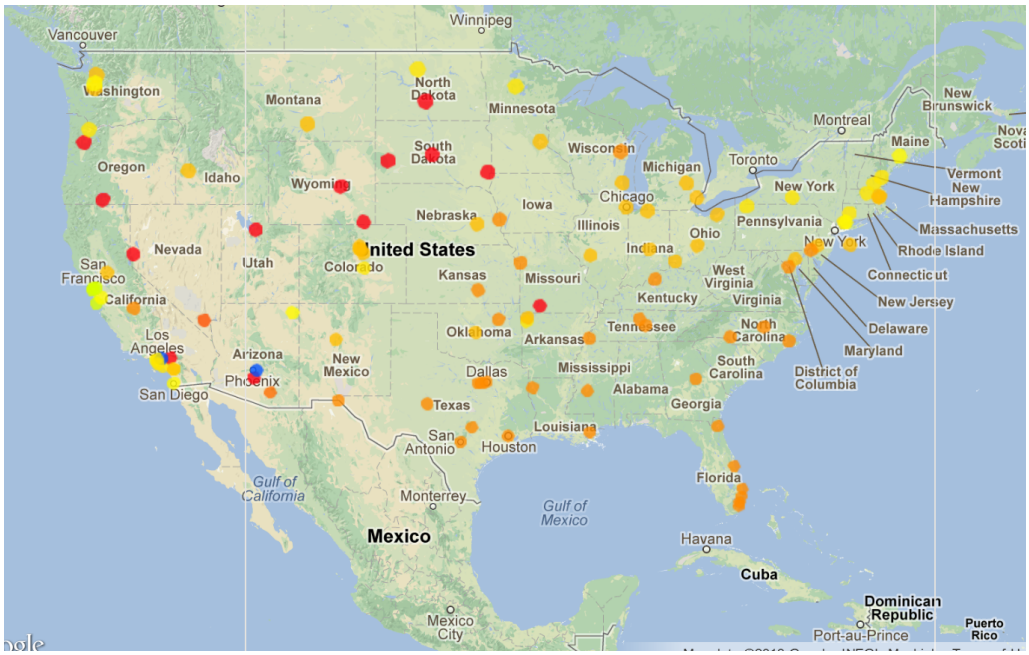
After discussing the scale categories, we decided that we didn't find any of the scales fitting for weather data. Category10, for example, starts with a blue color, moves to pink, then brown, mustard, and back to blue. In contrast, we wanted to start with dark blue for cold, transition to green/yellow for mild temperatures, and finally to a dark red to represent warm temperatures. Using this blue to red scale, which matches a thermometer, would be most intuitive to a user. Since people already associate blue with cold and red with hot, it would approximate weather obvious to the user from just looking at the map without the need for an extensive legend.

We looked around the web for some options but ultimately decided to roll-out our own color scale. Basically, we wrote a function that would take a temperature and return a hex value for to encode the color bubble based on our parameters:

```
function chooseColor(temp)
{
      if (temp < 0)
      {
            return "#0022ff";
      }
      else if (temp < 5)
      {
            return "#0054ff";
      }
      else if (temp < 10)
      {
            return "#0074ff";
      }
      …
```

```
        …
        else if (temp < 95)
        {
              return "#FF5000";
        }
        else
        {
              return "#FF0010";
        }
}
```

In the end, our map looked much better! Here is view of the map in July 2002:

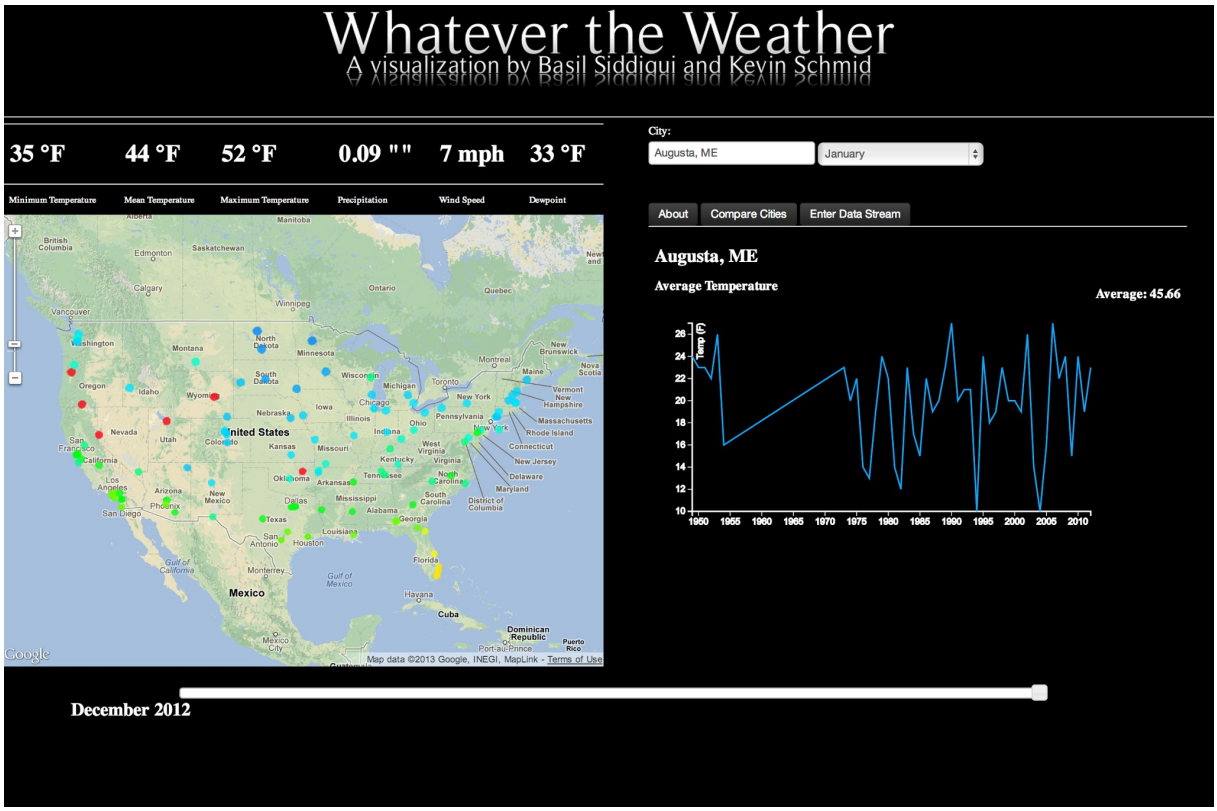


Redesigning the Title

After some discussion, we decided the WordArt was too bulky and explored some other options. We ultimately ended up going with this new design:

Redesigned Interface

After much trial and error, we came to an interface that we're quite happy with:



Audio Tour

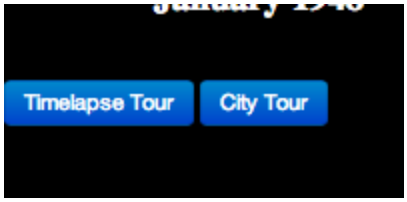
For this project, we really wanted to add a feature that we thought would make the project

resonate with viewers. We hoped that we could find some way to use the storytelling features that we learned about in the lectures to entice viewers to - put plainly, explore their own stories. So we considered adding a couple of audio tours to the visualization, that would provide the user with a walkthrough of not only how to use the visualization, but also how to glean insights and explore further.

In the audio tours, we pointed out several things along the way, including temperature fluctuations and outliers that we noticed. During the tours, we highlighted specific pieces (buttons, table views, etc.) on the interface, such as buttons that displayed information.

Here's how we implemented the audio feature:

- I went off into a quiet room in Lamont Library and recorded some sound snippets. For each tour we created, we used about four clips.
- We then added these clips to our HTML document, using `audio` tags.
- We created two functions, one for each tour, that kicked off the audio player. We then used event handlers and JavaScript's `setTimeout`, along with JQuery CSS updating, to highlight and time the playback of our audio.
- Here are the buttons, at the bottom of the visualization, that you can use to play the tours:



I honestly had so much fun making this feature; I wish we thought of it earlier in the process so we could have added more!

Here are the **scripts** from the tour:

Tour 1: Timelapse Feature

Here, we're going to use our visualization's play feature to view the change in mean temperature over time for winters.

First, adjust the slider on the bottom so that the beginning year is January 1948.

Awesome! Thanks! Now that we're in play mode, just hit the Start Timelapse button to begin the timelapse through January's, since 1948 to present.

As we walk through time here, notice that the mean temperature fluctuates pretty steadily over the years in question, staying at around 40 degrees, bouncing up and down by about 5 degrees depending on which year. While you watch the mean temperature change, notice that the colors

remain consistent on the map, showing that the individual temperatures stay pretty consistent too. You might even want to try zooming in on a particular region or city, to compare the effect there also.

Tour 2: City Feature

We can also use the Whatever the Weather visualization to find trends for a particular region, in case we're curious. For example, let's look at the Central United States. Adjust the slider on the bottom so that we're taking a look at January 1990.

Now, adjust the end slider on the right to be January 1997.

Now, hit the Start Timelapse button so we can begin to see the climate changes over time.

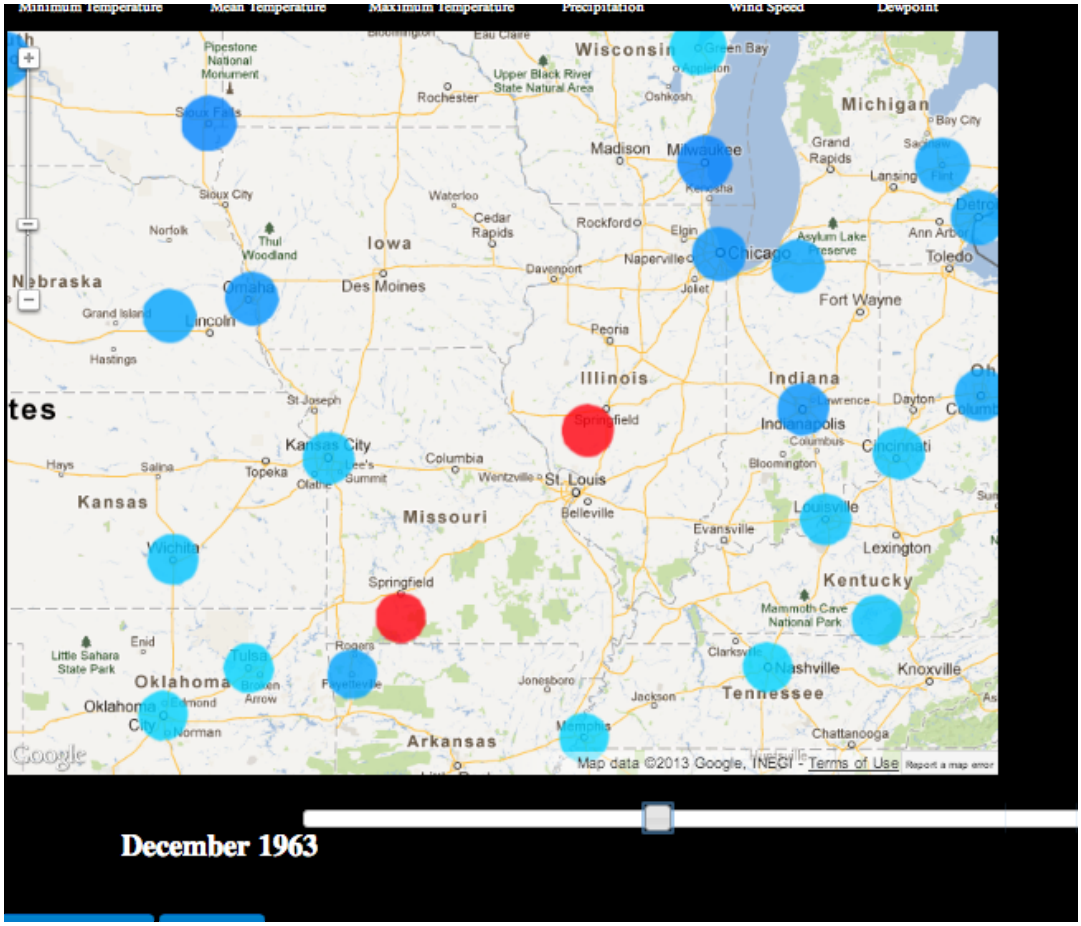
As you watch this timelapse, pay close attention to the middle of the United States - right where Illinois is. Hmm. It looks like it got unseasonably warm there - what could have caused this? Let's explore further. Let's lookup Modesto in the search box.

Insights Gained

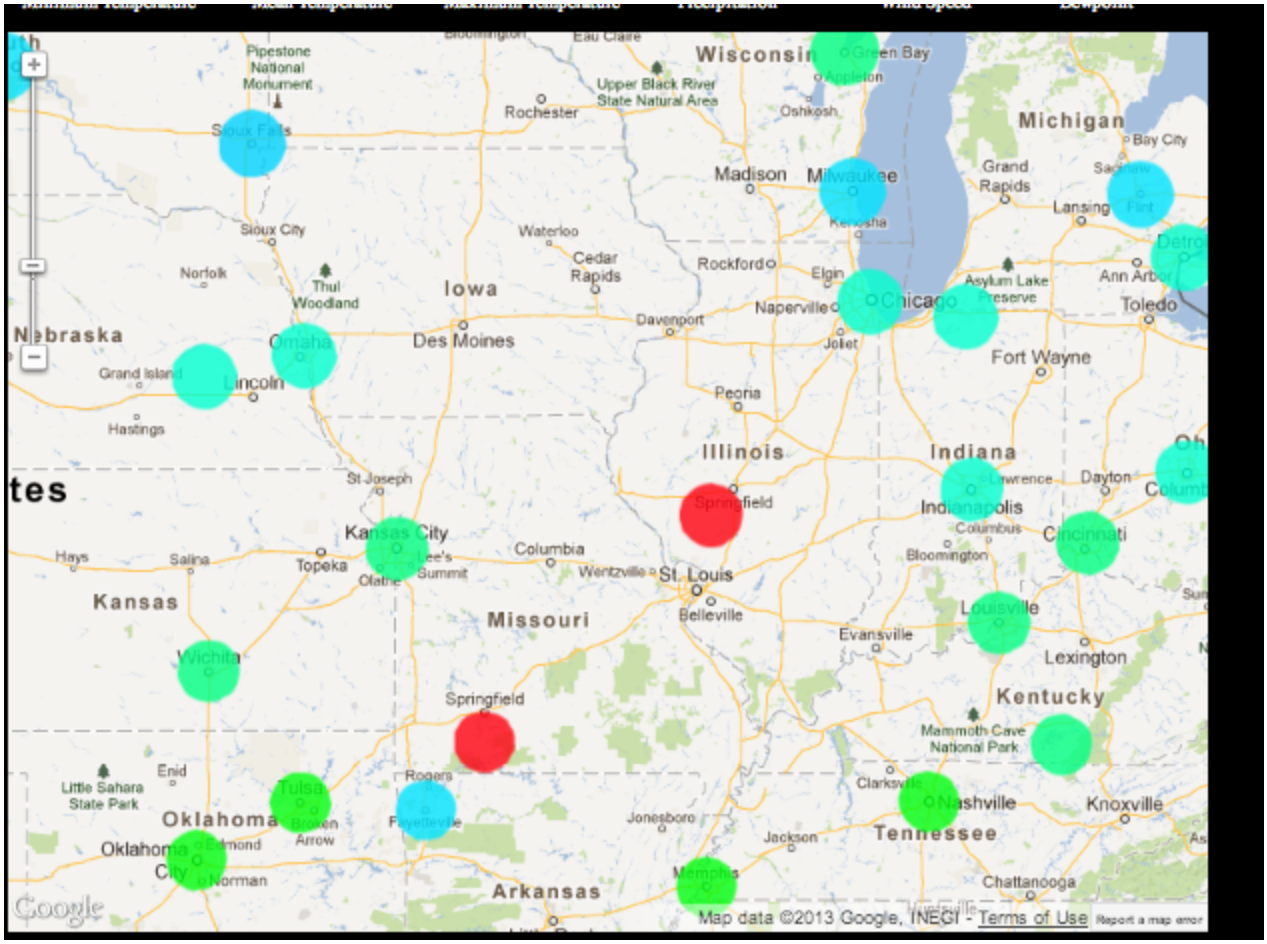
Being able to access data on a month-to-month basis makes a huge difference in our analysis! We edited all of the graphs and charts to allow the user to specify a month, because we think that stepping through and analyzing data on a monthly basis is more revealing than just seeing aggregate data for a year. We wanted to make the user experience promote this as much as we possibly could.

Some notable things we saw in the visualization:

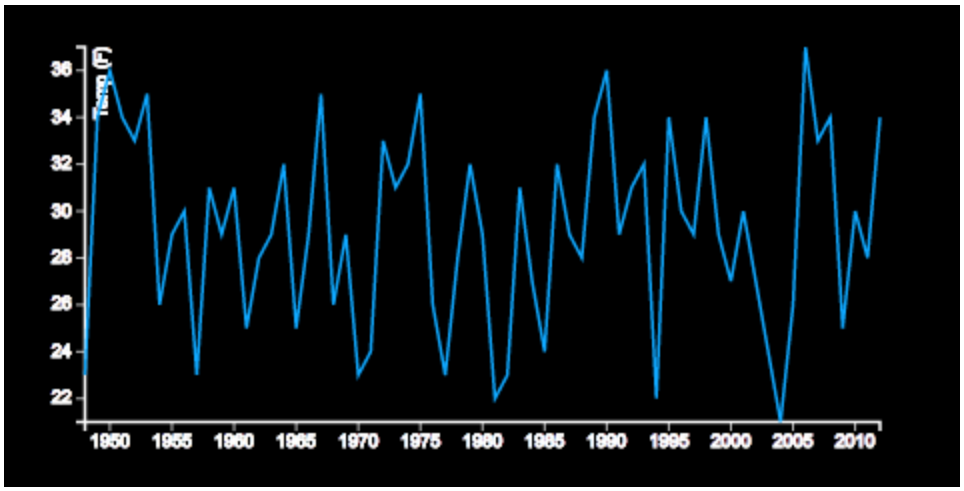
- For the most part, summers and winters don't seem to be more extreme. You can gain a visual intuition for this by looking at the colors and kicking off the timelapse. You can also gain a more quantitative backing for this by using the brushing and linking offered by our line graph. Just select a particular city, and use the brushing and linking to highlight a region and see the average. Even in the extreme situations, the temperature is quite stable over time. Trends certainly did not change as much as we expected.
- There are years where some regions experience increased warm (take January 1990 - January 1997, Central United States), but given enough time (consider January 2003), things seem to level off. Another thing to note is that these spikes (or lows) don't seem to be particularly extreme, lending credence to the notion that global warming isn't as extreme as it is portrayed in the media.
- Also, the hot Illinois temperatures were there in 1963, too:



Also, it's a bit unusual given the surroundings - so this is perhaps a result of inaccurate instrumentation.
And this trend persists even in Fall months:



- Seasonable temperatures seem to be getting slightly more extreme in recent years compared to immediately previous years, but not in an alarming way, and it's not inconsistent with the remote past.
  Here's Boston, for example:



Resources Used

We used the NVD3.js library (http://nvd3.org/) for our compare line chart visualization.
We used the d3 library for (http://d3js.org/) our color scale and line chart and brushing and it is a dependency of D3.