D3 for Mere Mortals

By Luke Francl (look@recursion.org), August 2011

d3.js is a data visualization library by Mike Bostock, who is also the primary creator of Protovis, which D3 is designed to replace.

D3 has a steep learning curve, especially if (like me) you are not used to the pixel-precision of graphics programming. To build a visualization with D3, you need to understand JavaScript objects, functions, and the method-chaining paradigm of jQuery; the basics of SVG and CSS; D3's API; and the principles for designing effective infographics.

The pay off is that you can create some amazing visualizations with D3 (just look at the examples!).

I know very little about D3, but the best way to learn something is to teach it...so here is a very simple introduction to D3 from the beginning.

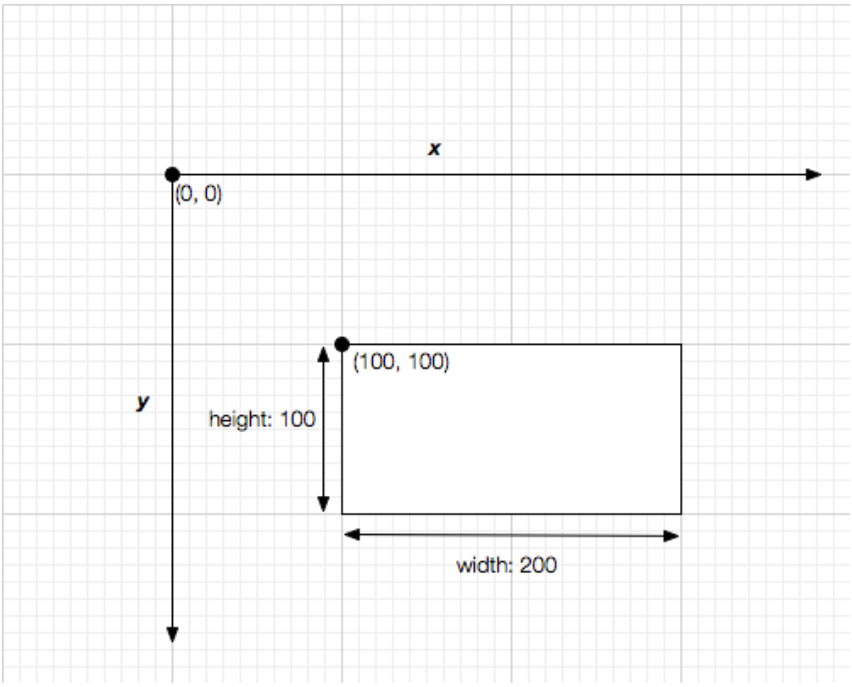
SVG

D3 shares similarities with its predecessor Protovis. However, instead of having its own graphical representation, D3 uses SVG (or HTML for simpler visualizations). This means that using D3 requires a fair amount of understanding of SVG. Fortunately, D3 provides a jQuery-like interface to the DOM, which means you will not have to endure the XML situps.

SVG provides basic shape primitives like `line`, `rect`, and `circle` as well as `text` and `path` to build complicated lines and shapes. If you think about a bar chart, you can see how you could make one of lines and rectangles with text for labels. D3 provides an API to help you place your rectangles in the correct location on the canvas.

To place a rectangle on the canvas, you need to understand that the axis starts in the top left corner and counts up towards the bottom of the screen. An SVG `rect` is positioned from its top-left corner and has a width and height.

Here's a diagram of a rectangle positioned at (100, 100) with a height of 100 pixels and a width of 200 pixels.



To produce this with D3, add a container `div` to the page, and the following JavaScript code.

```
var rectDemo = d3.select("#rect-demo").
  append("svg:svg").
  attr("width", 400).
  attr("height", 300);

rectDemo.append("svg:rect").
  attr("x", 100).
  attr("y", 100).
  attr("height", 100).
  attr("width", 200);
```

The result looks like this.

Not very impressive, eh? However, it does demonstrate how to use the D3 API to build an SVG document. The generated SVG code looks like this:

```
<svg width="400" height="300">
  <rect x="100" y="100" height="100" width="200"></rect>
</svg>
```

Drawing a Bar Chart

A vertical bar chart (or column chart) is basically a series of rectangles with the sizes proportional to the data being represented. D3 provides scales to translate between the input (your data) and the output (an x or y position on the canvas).

Understanding Scales

A linear scale is declared like this:

```
var xScale = d3.scale.linear().
  domain([0, 20]). // your data minimum and maximum
  range([0, 100]); // the pixels to map to, e.g., the width of the diagram.
```

`d3.scale.linear()` returns a function which above is stored in the xScale variable. In use, the xScale function maps between data and pixel positions:

```
xScale(0) // => 0
xScale(10) // => 50
xScale(20) // => 100
```

In D3, scales can be used two ways to set the position of an element. If the data is a simple array of numbers, the scale itself can be passed to the attr method:

```
attr("x", xScale)
```

If the data is more complicated, or if you need to perform additional calculations to adjust the position, you can pass a function to `attr`. It takes two arguments: `datum` and `index` (which are often written as "d" and "i"). Remember that JavaScript functions can accept a variable number of arguments, so if you only need `datum` your function does not need to take `index`.

Inside this function, you can call the scale function, manipulate the result, and return it.

```
attr("x", function(datum, index) {
  return xScale(datum.foobar) + whatever;
})
```

Using a scale reduces the amount of math you need to do to lay out a visualization, but using them is not required.

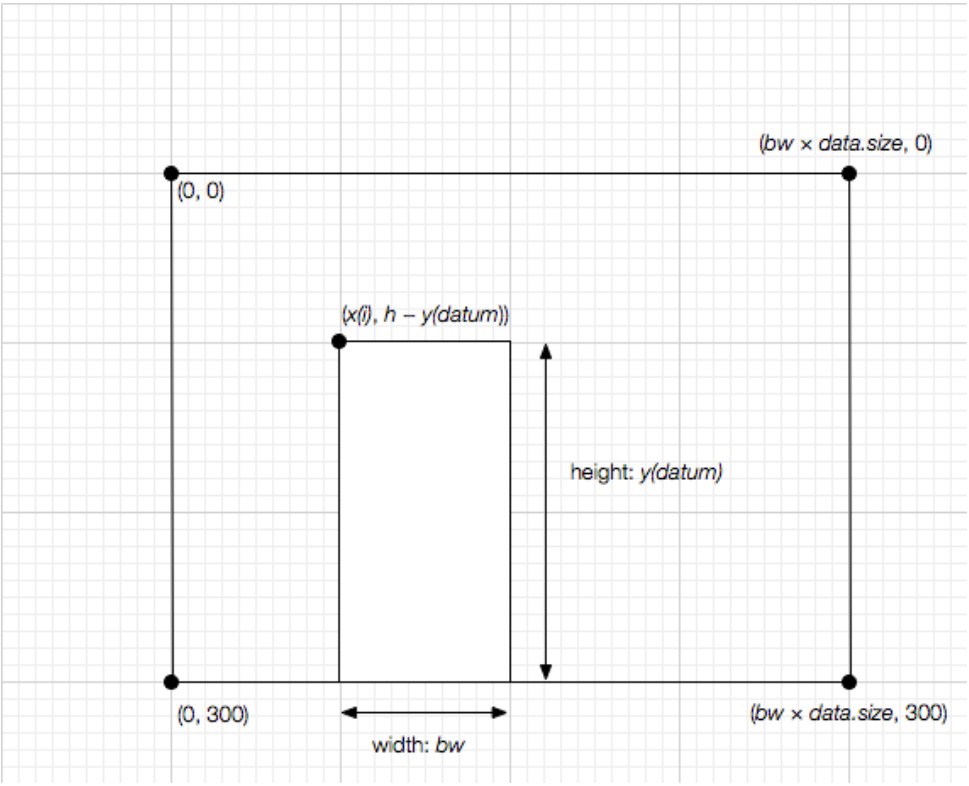
Diagramming a Bar Chart

In the diagram below, a single `rect` from a bar chart is represented. The canvas is 300 pixels tall and the width is the number of data points multiplied by the bar width *bw*. There are two scale functions, *x* and *y*.

The x position of the top left of the rectangle is easy: it is `x(i)` where `x` is the scale function and `i` is the index of the current datum.

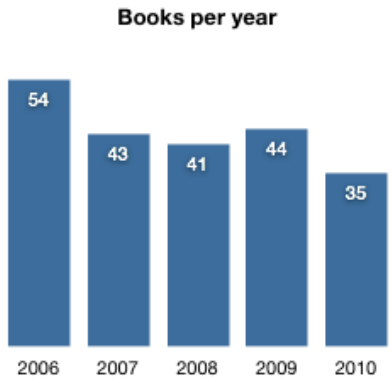
Since the SVG axis is located at the top of the graph, the height *h* is subtracted from the value of *y(datum)* to get the y position.

The width is similarly easy, as it is defined to be *bw*. The height is *y(datum)*.



Drawing the Bars

To give a concrete example, here is a bar graph showing books read per year.



To replicate this in D3, you will create a series of rectangles of the appropriate height along the x axis. D3's linear scale will provide functions used to map a data point on the x and y axis.

The data for the chart is a list of JavaScript objects with properties for year and number of books read. Since the years are continuous, there is no need to track them with the data, but doing so makes the code simpler to understand. The technique of using an array of JavaScript objects will be more useful for more complicated data, as you will see below.

To draw the `rects`, first use the selectAll method returns to create a selector, then use the data method to binds the array of data objects to the selection. Calling enter on the selection will create any objects that don't have corresponding elements in the data array.

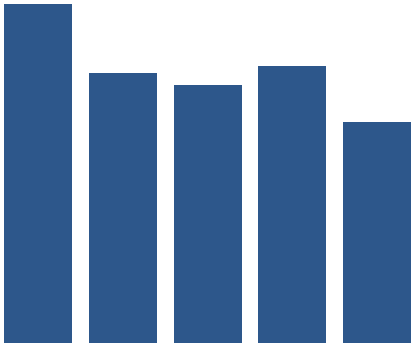
Since there's no existing `rects` returned by the selector, one will be created for each data element, so it may seem superfluous. However, the `selectAll`/`data`/`enter` sequence can be used to add or remove elements from an existing list. Take a look at the Three Little Circles demo for more about `data` and `enter`.

The rest of the diagram all comes down to positioning and formatting.

```
var data = [{year: 2006, books: 54},
            {year: 2007, books: 43},
            {year: 2008, books: 41},
```

```
            {year: 2009, books: 44},
            {year: 2010, books: 35}];

var barWidth = 40;
var width = (barWidth + 10) * data.length;
var height = 200;

var x = d3.scale.linear().domain([0, data.length]).range([0, width]);
var y = d3.scale.linear().domain([0, d3.max(data, function(datum) { return datum.books; })]).
  rangeRound([0, height]);

// add the canvas to the DOM
var barDemo = d3.select("#bar-demo").
  append("svg:svg").
  attr("width", width).
  attr("height", height);

barDemo.selectAll("rect").
  data(data).
  enter().
  append("svg:rect").
  attr("x", function(datum, index) { return x(index); }).
  attr("y", function(datum) { return height - y(datum.books); }).
  attr("height", function(datum) { return y(datum.books); }).
  attr("width", barWidth).
  attr("fill", "#2d578b");
```

The result looks like this:

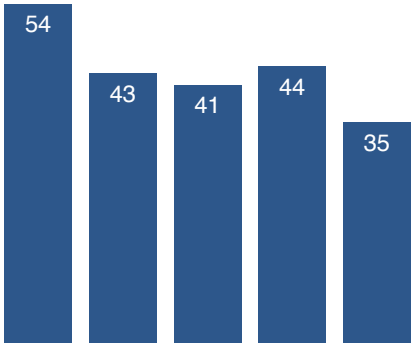


Adding Text and Axes

To add numbers inside the bars, svg:text elements must be positioned in the same place as the top of the bar and padding added to make it look right. The `svg:text` element has a lot of attributes for positioning and formatting. In this example, dx, dy, text-anchor and style are used.

Again, `selectAll` is used get a selection of elements, data is bound to them with `data` and then `enter` is used to add the elements to to the chart.

```
barDemo.selectAll("text").
  data(data).
  enter().
  append("svg:text").
  attr("x", function(datum, index) { return x(index) + barWidth; }).
  attr("y", function(datum) { return height - y(datum.books); }).
  attr("dx", -barWidth/2).
  attr("dy", "1.2em").
  attr("text-anchor", "middle").
  text(function(datum) { return datum.books;}).
  attr("fill", "white");
```

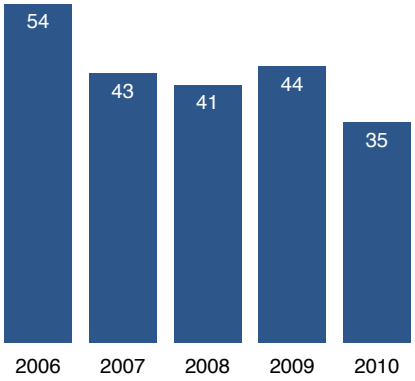


To add an x axis, the height of the chart must be increased to make room. A padding of 30 pixels will work nicely. By keeping the old `height` variable and increasing the height of the `svg:svg` canvas by `padding`, the rest of the code does not need to change. The axis could be positioned by using `height + padding` as the baseline, but using translate makes the x and y attributes simpler.

```
barDemo.selectAll("text.yAxis").
  data(data).
  enter().append("svg:text").
  attr("x", function(datum, index) { return x(index) + barWidth; }).
  attr("y", height).
  attr("dx", -barWidth/2).
  attr("text-anchor", "middle").
  attr("style", "font-size: 12; font-family: Helvetica, sans-serif").
  text(function(datum) { return datum.year;}).
  attr("transform", "translate(0, 18)").
  attr("class", "yAxis");
```

The font has also been changed to closer match the template. This can also be done with CSS, as you'll see in the next example.

And the result:



Drawing a Custom Visualization

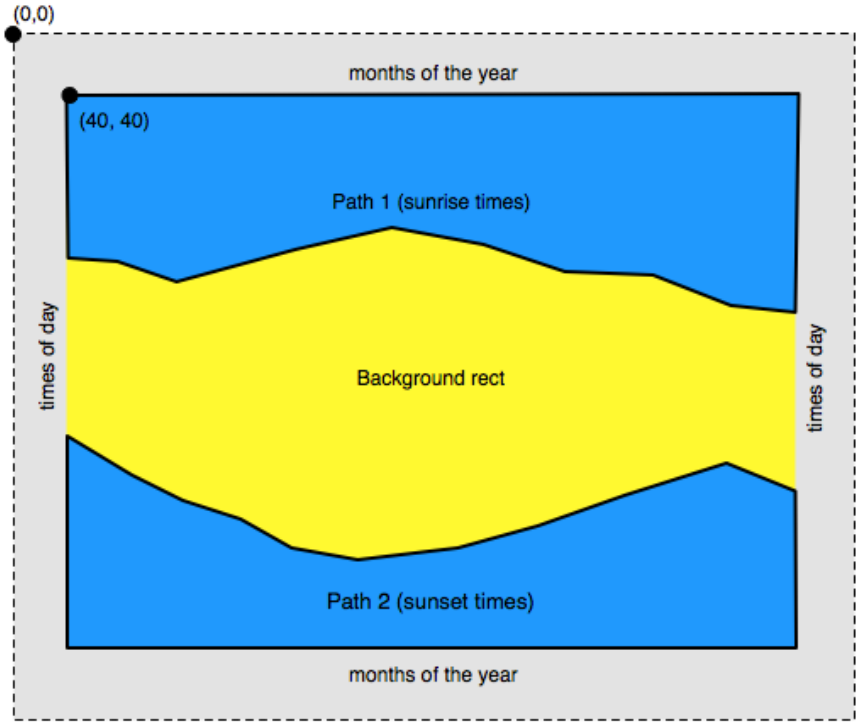
So far, you would have been better off using Google Chart Tools to produce a simple chart like this. But this barely scratches the surface of what D3 is capable of. A more complicated example will show how you can make custom visualizations with D3 that are impossible with standard charting libraries. This is where D3 really shines.

The goal is to make a chart of day lengths throughout the year, like this one drawn by a child from M&M child care.

This chart has several interesting aspects that are worth noting. It is centered on noon and the length of the day is shown with two lines over the central axis, which forms a shape which is colored in and the background is blue to represent night. There are four labeled axes, top and bottom and left and right. Building a visualization like it is not likely to come from a pre-packaged library.

Planning the Visualization

To plan the attack, a diagram of the desired visualization is helpful.



This brings several points to the foreground:

- The visualization will be inset with a margin of 40 pixels to print the axes. Using transform will make the math for this simple.
- Axes will go above and below and left and right of the graphic. Since the axes are all time-related, it makes sense to use time scales.
- Helpfully, the y axis counts up the same way as the SVG coordinate system so you do not need to reverse the axis.
- The sunrise section will be yellow and will be made up of what is left of a rect after drawing shapes with a `svg:path` for sunrise time and sunset time. d3.svg.area makes this easy.
- The points on the path need to be connected into a smooth curve. There are different kinds of interpolation for this.

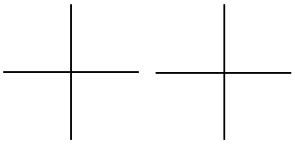
New Concepts

For this example, some new concepts will be introduced:

- Time scales, an extension of linear scales tailored for time values.
- The svg:g element which groups other elements like a `div` and can be transformed as a group.
- Drawing complex shapes with svg:path, which is complicated, but made easy with the d3.svg.area function.
- Using d3.range to create an array of numbers.
- Using scale.ticks to generate tick marks for the axes.
- Using SVG translation to simplify positioning.
- Styling text with CSS.
- Adjusting pixel positions to avoid anti-aliasing

Digression: Anti-aliasing

Anti-aliasing helps make text and curved lines look smoother, but can make straight lines appear fuzzy. If you are using a WebKit-based browser, the lines on the left should be fuzzy, while those on the right should be crisp.



Anti-aliasing often happens when drawing straight lines with D3. If the line is 1 pixel tall and positioned exactly on a pixel, it will straddle the pixel and become fuzzy. To avoid it, add or subtract 0.5 to the pixel position. The example below uses this technique to prevent fuzziness.

However, rendering differs depending on the SVG layout engine. In the example above, Firefox 5 renders fuzzy vertical lines when positioned exactly on a pixel, while the horizontal lines are not.

Setting the SVG `shape-rendering` property to `crispEdges` disables anti-aliasing for some or all lines. However, this may have unwanted side-effects if the visualization has curved lines, making them look jagged.

```
svg {
  shape-rendering: crispEdges;
}
```

A better solution may be to target the `crispEdges` property at certain (vertical or horizontal) lines with a CSS class.

```
line.crisp {
  shape-rendering: crispEdges;
}
```

Using this solution without pixel munging seems to work better across different browsers, but you should test it yourself and see what works best for your audience.

Getting the Data

Having game-planned the graphic, the first step is to collect the data. You can find sunrise and sunset times for cities around the world at timeanddate.com. I could not find a source of sunrise and sunset times in a computer-readable format and didn't feel like writing a scraper, so I selected a few days throughout the year for the selected location (Minneapolis, Minnesota).

The data format is like this:

```
{
 date: new Date(2011, 4, 15), // day of year
 sunrise: [7, 51],            // sunrise time as array of hours and minutes
 sunset: [16, 42]             // likewise, sunset time (24 hour clock)
}
```

Sunrise and sunset times are stored as an array of hour and minute values. These are used to construct a `Date` to represent the time of

day when plotting.

Building the Visualization

The JavaScript code for the visualization is below, with comments in line.

```
var width = 700;
var height = 525;
var padding = 40;

// the vertical axis is a time scale that runs from 00:00 - 23:59
// the horizontal axis is a time scale that runs from the 2011-01-01 to 2011-12-31

var y = d3.time.scale().domain([new Date(2011, 0, 1), new Date(2011, 0, 1, 23, 59)]).range([0, height]);
var x = d3.time.scale().domain([new Date(2011, 0, 1), new Date(2011, 11, 31)]).range([0, width]);

var monthNames = ["Jan", "Feb", "Mar", "April", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];

// Sunrise and sun set times for dates in 2011. I have picked the 1st
// and 15th day of every month, plus other important dates like equinoxes
// and solstices and dates around the standard time/DST transition.

var data = [
  {date: new Date(2011, 0, 1), sunrise: [7, 51], sunset: [16, 42]},
  {date: new Date(2011, 0, 15), sunrise: [7, 48], sunset: [16, 58]},
  {date: new Date(2011, 1, 1), sunrise: [7, 33], sunset: [17, 21]},
  {date: new Date(2011, 1, 15), sunrise: [7, 14], sunset: [17, 41]},
  {date: new Date(2011, 2, 1), sunrise: [6, 51], sunset: [18, 0]},
  {date: new Date(2011, 2, 12), sunrise: [6, 32], sunset: [18, 15]}, // dst - 1 day
  {date: new Date(2011, 2, 13), sunrise: [7, 30], sunset: [19, 16]}, // dst
  {date: new Date(2011, 2, 14), sunrise: [7, 28], sunset: [19, 18]}, // dst + 1 day
  {date: new Date(2011, 2, 14), sunrise: [7, 26], sunset: [19, 19]},
  {date: new Date(2011, 2, 20), sunrise: [07, 17], sunset: [19, 25]}, // equinox
  {date: new Date(2011, 3, 1), sunrise: [6, 54], sunset: [19, 41]},
  {date: new Date(2011, 3, 15), sunrise: [6, 29], sunset: [19, 58]},
  {date: new Date(2011, 4, 1), sunrise: [6, 3], sunset: [20, 18]},
  {date: new Date(2011, 4, 15), sunrise: [5, 44], sunset: [20, 35]},
  {date: new Date(2011, 5, 1), sunrise: [5, 30], sunset: [20, 52]},
  {date: new Date(2011, 5, 15), sunrise: [5, 26], sunset: [21, 1]},
  {date: new Date(2011, 5, 21), sunrise: [5, 26], sunset: [21, 3]}, // solstice
  {date: new Date(2011, 6, 1), sunrise: [5, 30], sunset: [21, 3]},
  {date: new Date(2011, 6, 15), sunrise: [5, 41], sunset: [20, 57]},
  {date: new Date(2011, 7, 1), sunrise: [5, 58], sunset: [20, 40]},
  {date: new Date(2011, 7, 15), sunrise: [6, 15], sunset: [20, 20]},
  {date: new Date(2011, 8, 1), sunrise: [6, 35], sunset: [19, 51]},
  {date: new Date(2011, 8, 15), sunrise: [6, 51], sunset: [19, 24]},
  {date: new Date(2011, 8, 23), sunrise: [7, 1], sunset: [19, 9]}, // equinox
  {date: new Date(2011, 9, 1), sunrise: [7, 11], sunset: [18, 54]},
  {date: new Date(2011, 9, 15), sunrise: [7, 28], sunset: [18, 29]},
  {date: new Date(2011, 10, 1), sunrise: [7, 51], sunset: [18, 2]},
  {date: new Date(2011, 10, 5), sunrise: [7, 57], sunset: [17, 56]}, // last day of dst
  {date: new Date(2011, 10, 6), sunrise: [6, 58], sunset: [16, 55]}, // standard time
  {date: new Date(2011, 10, 7), sunrise: [6, 59], sunset: [16, 54]}, // standard time + 1
  {date: new Date(2011, 10, 15), sunrise: [7, 10], sunset: [16, 44]},
  {date: new Date(2011, 11, 1), sunrise: [7, 31], sunset: [16, 33]},
  {date: new Date(2011, 11, 15), sunrise: [7, 44], sunset: [16, 32]},
  {date: new Date(2011, 11, 22), sunrise: [7, 49], sunset: [16, 35]}, // solstice
  {date: new Date(2011, 11, 31), sunrise: [7, 51], sunset: [16, 41]}
];

function yAxisLabel(d) {
  if (d == 12) { return "noon"; }
  if (d < 12) { return d; }
  return (d - 12);
}

// The labels along the x axis will be positioned on the 15th of the
// month

function midMonthDates() {
  return d3.range(0, 12).map(function(i) { return new Date(2011, i, 15) });
}
```

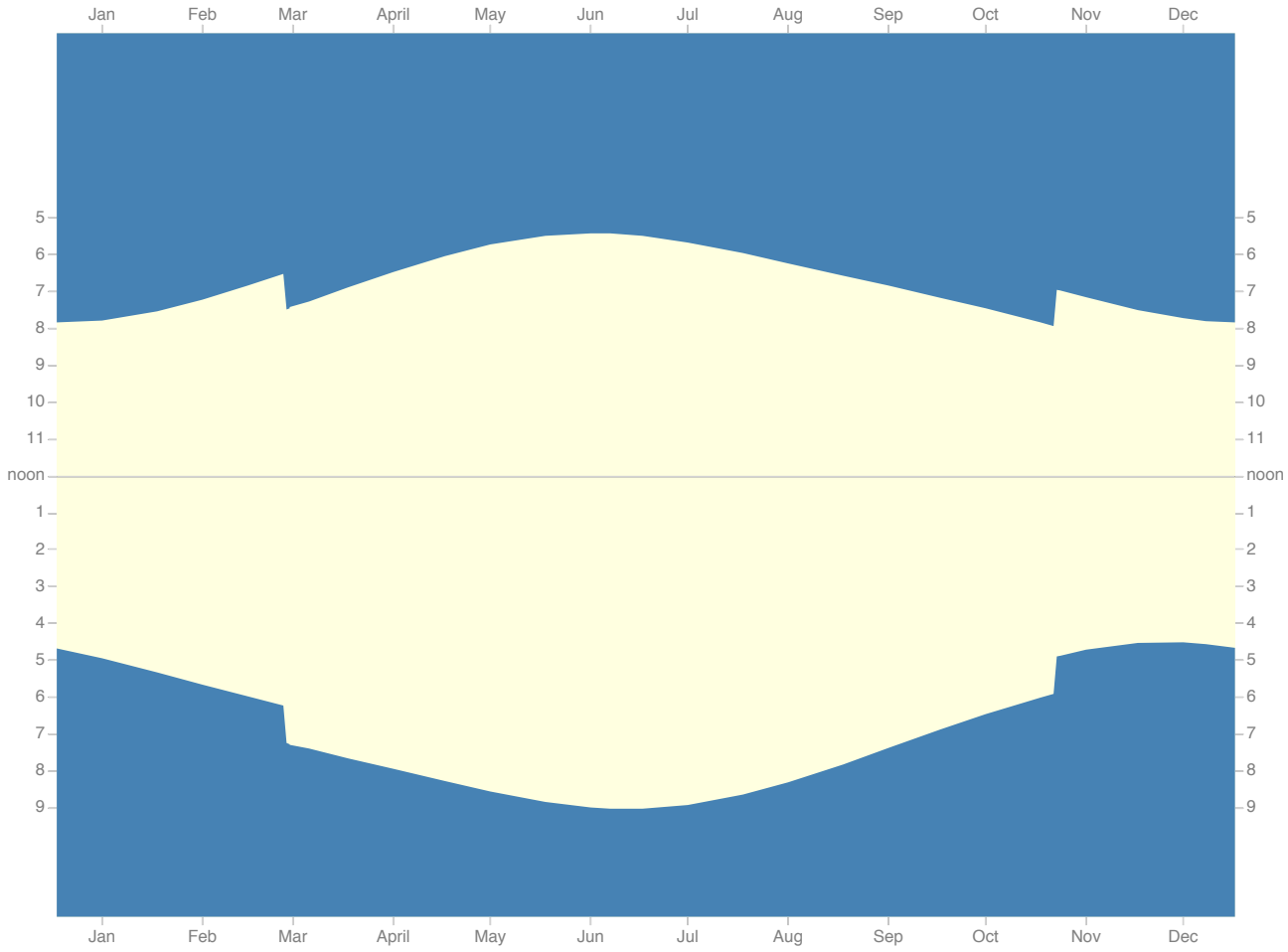
```
var dayLength = d3.select("#day-length").
  append("svg:svg").
  attr("width", width + padding * 2).
  attr("height", height + padding * 2);

// create a group to hold the axis-related elements
var axisGroup = dayLength.append("svg:g").
  attr("transform", "translate("+padding+","+padding+")");

// draw the x and y tick marks. Since they are behind the visualization, they
// can be drawn all the way across it. Because the  has been
// translated, they stick out the left side by going negative.

axisGroup.selectAll(".yTicks").
  data(d3.range(5, 22)).
  enter().append("svg:line").
  attr("x1", -5).
  // Round and add 0.5 to fix anti-aliasing effects (see above)
  attr("y1", function(d) { return d3.round(y(new Date(2011, 0, 1, d))) + 0.5; }).
  attr("x2", width+5).
  attr("y2", function(d) { return d3.round(y(new Date(2011, 0, 1, d))) + 0.5; }).
  attr("stroke", "lightgray").
  attr("class", "yTicks");

axisGroup.selectAll(".xTicks").
  data(midMonthDates).
  enter().append("svg:line").
  attr("x1", x).
  attr("y1", -5).
  attr("x2", x).
  attr("y2", height+5).
  attr("stroke", "lightgray").
  attr("class", "yTicks");

// draw the text for the labels. Since it is the same on top and
// bottom, there is probably a cleaner way to do this by copying the
// result and translating it to the opposite side

axisGroup.selectAll("text.xAxisTop").
  data(midMonthDates).
  enter().
  append("svg:text").
  text(function(d, i) { return monthNames[i]; }).
  attr("x", x).
  attr("y", -8).
  attr("text-anchor", "middle").
  attr("class", "axis xAxisTop");

axisGroup.selectAll("text.xAxisBottom").
  data(midMonthDates).
  enter().
  append("svg:text").
  text(function(d, i) { return monthNames[i]; }).
  attr("x", x).
  attr("y", height+15).
  attr("text-anchor", "middle").
  attr("class", "xAxisBottom");

axisGroup.selectAll("text.yAxisLeft").
  data(d3.range(5, 22)).
  enter().
  append("svg:text").
  text(yAxisLabel).
  attr("x", -7).
  attr("y", function(d) { return y(new Date(2011, 0, 1, d)); }).
  attr("dy", "3").
  attr("class", "yAxisLeft").
  attr("text-anchor", "end");

axisGroup.selectAll("text.yAxisRight").
  data(d3.range(5, 22)).
```

```
  enter().
  append("svg:text").
  text(yAxisLabel).
  attr("x", width+7).
  attr("y", function(d) { return y(new Date(2011, 0, 1, d)); }).
  attr("dy", "3").
  attr("class", "yAxisRight").
  attr("text-anchor", "start");

// create a group for the sunrise and sunset paths

var lineGroup = dayLength.append("svg:g").
  attr("transform", "translate("+ padding + ", " + padding + ")");

// draw the background. The part of this that remains uncovered will
// represent the daylight hours.

lineGroup.append("svg:rect").
  attr("x", 0).
  attr("y", 0).
  attr("height", height).
  attr("width", width).
  attr("fill", "lightyellow");

// The meat of the visualization is surprisingly simple. sunriseLine
// and sunsetLine are areas (closed svg:path elements) that use the date
// for the x coordinate and sunrise and sunset (respectively) for the y
// coordinate. The sunrise shape is anchored at the top of the chart, and
// sunset area is anchored at the bottom of the chart.

var sunriseLine = d3.svg.area().
  x(function(d) { return x(d.date); }).
  y1(function(d) { return y(new Date(2011, 0, 1, d.sunrise[0], d.sunrise[1])); }).
  interpolate("linear");

lineGroup.
  append("svg:path").
  attr("d", sunriseLine(data)).
  attr("fill", "steelblue");

var sunsetLine = d3.svg.area().
  x(function(d) { return x(d.date); }).
  y0(height).
  y1(function(d) { return y(new Date(2011, 0, 1, d.sunset[0], d.sunset[1])); }).
  interpolate("linear");

lineGroup.append("svg:path").
  attr("d", sunsetLine(data)).
  attr("fill", "steelblue");

// finally, draw a line representing 12:00 across the entire
// visualization

lineGroup.append("svg:line").
  attr("x1", 0).
  attr("y1", d3.round(y(new Date(2011, 0, 1, 12))) + 0.5).
  attr("x2", width).
  attr("y2", d3.round(y(new Date(2011, 0, 1, 12))) + 0.5).
  attr("stroke", "lightgray");
```

Using CSS helps remove duplication from the text formatting attributes:

```
div#day-length text {
  fill: gray;
  font-family: Helvetica, sans-serif;
  font-size: 10px;
}
```

And the final result:



Next Steps

Now that the visualization is complete, think of what could be done to improve it. You could label the solstices and equinoxes with exact times, or plot different sunset and sunrise times for different locations against each other, or make a label that shows exact times for the day the user is are mousing over...

The possibilities are endless.

Debugging D3

Lest you assume these examples were coded straight through, let me assure you this was not the case. There was a significant amount of trial-and-error coding needed to get them working.

One of the advantages of D3 using SVG as its native graphical representation is that it is easier to debug the WebKit Inspector or Firebug.

```
▼<div id="bar-demo-part3">
  ▼<svg width="250" height="230">
      <rect x="0" y="20" height="180" width="40" fill="#2d578b"></rect>
      <rect x="50" y="57" height="143" width="40" fill="#2d578b"></rect>
      <rect x="100" y="63" height="137" width="40" fill="#2d578b"></rect>
```

This is incredibly useful. You can try it on this page to see how the SVG code for these examples is generated.

Another technique that has been helpful for me is to use `console.log` in positioning functions. This lets you inspect the datum and index of an element as well as the calculated position for the x or y value. For example:

```
attr("x", function(d, i) {
  console.log(d);
  console.log(i);

  return x(d.whatever);
})
```

Further Reading

Now you know about as much about D3 as I do. Obviously, this barely scratches the surface. For more tutorials and resources about D3, read the following articles:

- The D3 homepage, examples (don't miss the examples only in the source code), and API documentation.
- The D3 tutorials Three Little Circles, A Bar Chart, Part 1, and A Bar Chart, Part 2
- Jan Willem Tulp's D3 blog category especially Tutorial: Introduction to D3 and Tutorial: Line chart in D3.
- D3 scales and interpolation by Nelson Minar.
- D3: Scales and Color by Jerome Cukier (with nice diagrams explaining scales).
- Creating Basic Charts using d3.js by Ben Lorica (view source to see the D3 code).
- Try D3 Now by Christophe Viau.
- D3.js is Not a Graphing Library, Let's Design a Line Graph by Justin Palmer.

You also need to read about SVG to know what's possible – because with D3, you can do anything SVG can do.

- The SVG spec is a good place to start.
- Learn SVG by Jon Frost, Stefan Goesser and Michel Hirtzler provides a gentler introduction.

That's it for now. I hope this tutorial was helpful for you.

Return to recursion. More about me.