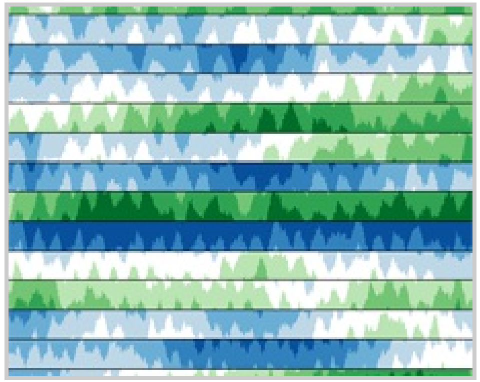
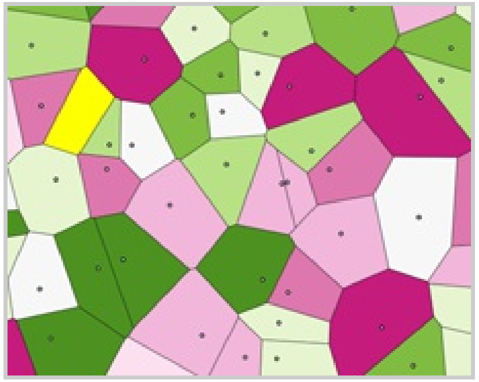
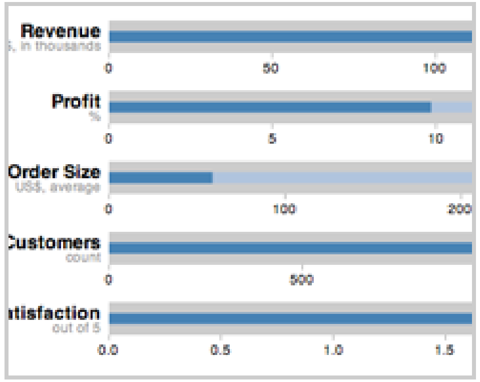
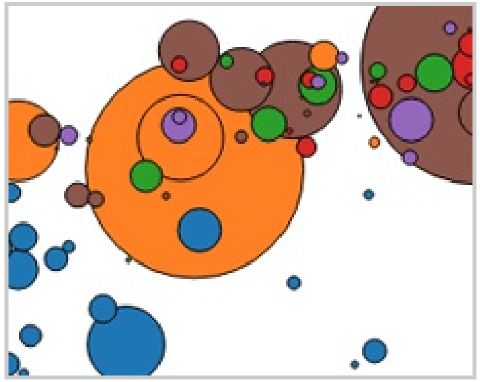
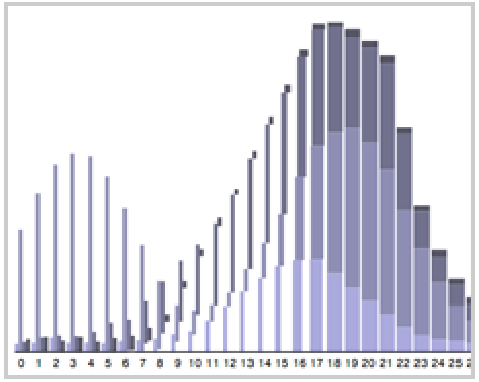
Data-Driven Documents



D3.js is a JavaScript library for manipulating documents based on data. **D3** helps you bring data to life using HTML, SVG and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

See more examples.

Download the latest version here:

- d3.v3.zip

Or, to link directly to the latest release, copy this snippet:

```
<script src="http://d3js.org/d3.v3.min.js"></script>
```

The full source and tests are also available for download on GitHub.

#Introduction

Read more tutorials.

D3 allows you to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document. For example, you can use D3 to generate an HTML table from an array of numbers. Or, use the same data to create an interactive SVG bar chart with smooth transitions and interaction.

D3 is not a monolithic framework that seeks to provide every conceivable feature. Instead, D3 solves the crux of the problem: efficient manipulation of documents based on data. This avoids proprietary representation and affords extraordinary flexibility, exposing the full capabilities of web standards such as CSS3, HTML5 and SVG. With minimal overhead, D3 is extremely fast, supporting large datasets and dynamic behaviors for interaction and animation. D3’s functional style allows code reuse through a diverse collection of components and plugins.

#Selections

Read more about selections.

Modifying documents using the W3C DOM API is tedious: the method names are verbose, and the imperative approach requires manual iteration and bookkeeping of temporary state. For example, to change the text color of paragraph elements:

```
var paragraphs = document.getElementsByTagName("p");
for (var i = 0; i < paragraphs.length; i++) {
  var paragraph = paragraphs.item(i);
  paragraph.style.setProperty("color", "white", null);
}
```

D3 employs a declarative approach, operating on arbitrary sets of nodes called *selections*. For example, you can rewrite the above loop as:

```
d3.selectAll("p").style("color", "white");
```

Yet, you can still manipulate individual nodes as needed:

```
d3.select("body").style("background-color", "black");
```

Selectors are defined by the W3C Selectors API and supported natively by modern browsers. Backwards-compatibility for older browsers can be provided by Sizzle. The above examples select nodes by tag name ("p" and "body", respectively). Elements may be selected using a variety of predicates, including containment, attribute values, class and ID.

D3 provides numerous methods for mutating nodes: setting attributes or styles; registering event listeners; adding, removing or sorting nodes; and changing HTML or text content. These suffice for the vast majority of needs. Direct access to the underlying DOM is also possible, as each D3 selection is simply an array of nodes.

#Dynamic Properties

Readers familiar with other DOM frameworks such as jQuery or Prototype should immediately recognize similarities with D3. Yet styles, attributes, and other properties can be specified as *functions of data* in D3, not just simple constants. Despite their apparent simplicity, these functions can be surprisingly powerful; the d3.geo.path function, for example, projects geographic coordinates into SVG path data. D3 provides many built-in reusable functions and function factories, such as graphical primitives for area, line and pie charts.

For example, to randomly color paragraphs:

```
d3.selectAll("p").style("color", function() {
  return "hsl(" + Math.random() * 360 + ",100%,50%)";
});
```

To alternate shades of gray for even and odd nodes:

```
d3.selectAll("p").style("color", function(d, i) {
  return i % 2 ? "#fff" : "#eee";
});
```

Computed properties often refer to bound data. Data is specified as an array of values, and each value is passed as the first argument (d) to selection functions. With the default join-by-index, the first element in the data array is passed to the first node in the selection, the second element to the second node, and so on. For example, if you bind an array of numbers to paragraph elements, you can use these numbers to compute dynamic font sizes:

```
d3.selectAll("p")
    .data([4, 8, 15, 16, 23, 42])
    .style("font-size", function(d) { return d + "px"; });
```

Once the data has been bound to the document, you can omit the data operator; D3 will retrieve the previously-bound data. This allows you to recompute properties without rebinding.

#Enter and Exit

Read more about data joins

Using D3’s *enter* and *exit* selections, you can create new nodes for incoming data and remove outgoing nodes that are no longer needed.

When data is bound to a selection, each element in the data array is paired with the corresponding node in the selection. If there are fewer nodes than data, the extra data elements form the enter selection, which you can instantiate by appending to the enter selection. For example:

```
d3.select("body").selectAll("p")
    .data([4, 8, 15, 16, 23, 42])
  .enter().append("p")
    .text(function(d) { return "I’m number " + d + "!"; });
```

Updating nodes are the default selection—the result of the data operator. Thus, if you forget about the enter and exit selections, you will automatically select only the elements for which there exists corresponding data. A common pattern is to break the initial selection into three parts: the updating nodes to modify, the entering nodes to add, and the exiting nodes to remove.

```
// Update…
var p = d3.select("body").selectAll("p")
    .data([4, 8, 15, 16, 23, 42])
    .text(String);

// Enter…
p.enter().append("p")
    .text(String);

// Exit…
p.exit().remove();
```

By handling these three cases separately, you specify precisely which operations run on which nodes. This improves performance and offers greater control over transitions. For example, with a bar chart you might initialize entering bars using the old scale, and then transition entering bars to the new scale along with the updating and exiting bars.

D3 lets you transform documents based on data; this includes both creating and destroying elements. D3 allows you to change an existing document in response to user interaction, animation over time, or even asynchronous notification from a third-party. A hybrid approach is even possible, where the document is initially rendered on the server, and updated on the client via D3.

#Transformation, not Representation

D3 is not a new graphical representation. Unlike Processing, Raphaël, or Protovis, the vocabulary of marks comes directly from web standards: HTML, SVG and CSS. For example, you can create SVG elements using D3 and style them with external stylesheets. You can use composite filter effects, dashed strokes and clipping. If browser vendors introduce new features tomorrow, you'll be able to use them immediately—no toolkit update required. And, if you decide in the future to use a toolkit other than D3, you can take your knowledge of standards with you!

Best of all, D3 is easy to debug using the browser's built-in element inspector: the nodes that you manipulate with D3 are exactly those that the browser understands natively.

#Transitions

D3's focus on transformation extends naturally to animated transitions. Transitions gradually interpolate styles and attributes over time. Tweening can be controlled via easing functions such as “elastic”, “cubic-in-out” and “linear”. D3's interpolators support both primitives, such as numbers and numbers embedded within strings (font sizes, path data, *etc.*), and compound values. You can even extend D3's interpolator registry to support complex properties and data structures.

For example, to fade the background of the page to black:

```
d3.select("body").transition()
    .style("background-color", "black");
```

Or, to resize circles in a symbol map with a staggered delay:

```
d3.selectAll("circle").transition()
    .duration(750)
    .delay(function(d, i) { return i * 10; })
    .attr("r", function(d) { return Math.sqrt(d * scale); });
```

By modifying only the attributes that actually change, D3 reduces overhead and allows greater graphical complexity at high frame rates. D3 also allows sequencing of complex transitions via events. And, you can still use CSS3 transitions; D3 does not replace the browser’s toolbox, but exposes it in a way that is easier to use.

Want to learn more? Read these tutorials.