Changes in D3 7.0

Released June 11, 2021.

This document covers only major changes. For minor and patch changes, please see the release notes.

D3 now ships as pure ES modules and requires Node.js 12 or higher. For more, please read Sindre Sorhus's FAQ.

d3.bin now ignores nulls. d3.ascending and d3.descending no longer consider null comparable.

Ordinal scales now use InternMap for domains; domain values are now uniqued by coercing to a primitive value via *object*.valueOf instead of coercing to a string via *object*.toString.

Array-likes (*e.g.*, a live NodeList such as *element*.childNodes) are converted to arrays in d3.selectAll and *selection*.selectAll.

Changes in D3 6.0

Released August 26, 2020.

D3 now **uses native collections** (Map and Set) and **accepts iterables**. d3.group and d3.rollup are powerful new aggregation functions that replace d3.nest and work great with d3-hierarchy and d3-selection. There are lots of new helpers in d3-array, too, such as d3.greatest, d3.quickselect, and d3.fsum.

D3 now **passes events directly to listeners**, replacing the d3.event global and bringing D3 inline with vanilla JavaScript and most other frameworks.

d3-delaunay (based on Vladimir Agafonkin's excellent Delaunator) replaces d3-voronoi, offering dramatic improvements to performance, robustness, and search. And there's a new d3-geo-voronoi for spherical (geographical) data! **d3-random** is greatly expanded and includes a fast linear congruential generator for seeded randomness. **d3-chord** has new layouts for directed and transposed chord diagrams. **d3-scale** adds a new radial scale type.

… and a variety of other small enhancements. More than 450 examples have been updated to D3 6.0!

d3-array

- Accept iterables.
- Add d3.group.
- Add d3.groups.
- Add d3.index.
- Add d3.indexes.
- Add d3.rollup.
- Add d3.rollups.
- Add d3.maxIndex.
- Add d3.minIndex.
- Add d3.greatest.
- Add d3.greatestIndex.
- Add d3.least.
- Add d3.leastIndex.
- Add d3.bin.
- Add d3.count.
- Add d3.cumsum.

- Add d3.fsum.
- Add d3.Adder.
- Add d3.quantileSorted.
- Add d3.quickselect.
- Add *bisector*.center.
- Allow more than two iterables for d3.cross.
- Accept non-sorted input with d3.quantile.
- Fix a *array*.sort bug in Safari.
- Fix bin thresholds to ignore NaN input.
- Fix d3.ticks to not return ticks outside the domain.
- Improve the performance of d3.median.

See https://observablehq.com/@d3/d3-array-2-0 for details.

d3-brush

- Add *event*.mode.
- Change *brush*.on to pass the *event* directly to listeners.
- Improve multitouch (two-touch) interaction.

d3-chord

- Add d3.chordDirected.
- Add d3.chordTranspose.
- Add d3.ribbonArrow.
- Add *ribbon*.padAngle.
- Add *ribbon*.sourceRadius.
- Add *ribbon*.targetRadius.

d3-delaunay

- Add d3.Delaunay.

d3-drag

- Change *drag*.on to pass the *event* directly to listeners.

d3-force

- Add *iterations* argument to *simulation*.tick.
- Add *forceCenter*.strength.
- Add *forceSimulation*.randomSource.
- All built-in forces are now fully deterministic (including “jiggling” coincident nodes).
- Improve the default phyllotaxis layout slightly by offsetting by one half-radius.
- Improve the error message when a link references an unknown node.
- *force*.initialize is now passed a random source.
- Fix bug when initializing nodes with fixed positions.

d3-format

- Change the default minus sign to the minus sign (−) instead of hyphen-minus (-).
- Fix decimal `d` formatting of numbers greater than or equal to 1e21.

d3-geo

- Fix clipping of some degenerate polygons.

d3-hierarchy

- Accept iterables.
- Add *node*[Symbol.iterator]; hierarchies are now iterable.
- Add *node*.find.
- Change *node*.each to pass the traversal index.
- Change *node*.eachAfter to pass the traversal index.
- Change *node*.eachBefore to pass the traversal index.
- Fix d3.packSiblings for huge circles.
- Fix divide-by-zero bug in d3.treemapBinary.
- Fix divide-by-zero bug in d3.treemapResquarify.

d3-interpolate

- Add *interpolateZoom*.rho. (#25)
- Allow d3.piecewise to default to using d3.interpolate. #90
- Change d3.interpolateTransformCss to use DOMMatrix and require absolute units. #83

d3-quadtree

- Fix an infinite loop when coordinates diverge to huge values.

d3-random

- Add d3.randomLcg.
- Add d3.randomGamma.
- Add d3.randomBeta.
- Add d3.randomWeibull.
- Add d3.randomCauchy.
- Add d3.randomLogistic.
- Add d3.randomPoisson.
- Add d3.randomInt.
- Add d3.randomBinomial.
- Add d3.randomGeometric.
- Add d3.randomPareto.
- Add d3.randomBernoulli.
- Allow d3.randomBates to take fractional *n*.
- Allow d3.randomIrwinHall to take fractional *n*.
- Don't wrap Math.random in the default source.

Thanks to @Lange, @p-v-d-Veeken, @svanschooten, @Parcly-Taxel and @jrus for your contributions!

d3-scale

- Accept iterables.
- Add *diverging*.rangeRound.
- Add *sequential*.range (for compatibility with d3-axis).
- Add *sequential*.rangeRound.
- Add *sequentialQuantile*.quantiles.
- Add d3.scaleRadial.
- *diverging*.range can now be used to set the interpolator.
- *sequential*.range can now be used to set the interpolator.
- d3.scaleDiverging can now accept a range array in place of an interpolator.
- d3.scaleSequential can now accept a range array in place of an interpolator.
- Fix *continuous*.nice to ensure that niced domains always span ticks.
- Fix *log*.ticks for small domains.

- Fix *log*.ticks for small domains. #44
- Fix *scale*.clamp for sequential quantile scales. Thanks, @Fil!
- Fix *scale*.clamp for continuous scales with more domain values than range values.
- Fix diverging scales with descending domains.
- Remove deprecated *step* argument from *time*.ticks and *time*.nice.

d3-selection

- Add *selection*.selectChild.
- Add *selection*.selectChildren.
- Add d3.pointer.
- Add d3.pointers.
- Add *selection*[Symbol.iterator]; selections are now iterable!
- Accept iterables with *selection*.data.
- Accept iterables with d3.selectAll.
- Change *selection*.on to pass the *event* directly to listeners.
- Remove index and group from *selection*.on listeners!
- Remove d3.event!
- Remove d3.mouse.
- Remove d3.touch.
- Remove d3.touches.
- Remove d3.customEvent.
- Remove d3.clientPoint.
- Remove d3.sourceEvent.
- Fix *selection*.merge(*transition*) to error.

For an overview of changes, see https://observablehq.com/@d3/d3-selection-2-0.

d3-shape

- Accept iterables.
- Add d3.line(*x*, *y*) shorthand.
- Add d3.area(*x*, *y0*, *y1*) shorthand.
- Add d3.symbol(*type*, *size*) shorthand.

d3-time-format

- Add ISO 8601 “week year” (`%G` and `%g`).

d3-timer

- Fix *interval*.restart to restart as an interval.

d3-transition

- Add *transition*.easeVarying.
- Add *transition*[Symbol.iterator]; transitions are now iterable.
- Fix *selection*.transition to error if the named transition to inherit is not found.k
- Fix *transition*.end to resolve immediately if the selection is empty.

d3-zoom

- Add *zoom*.tapDistance.
- Change *zoom*.on to pass the *event* directly to listeners.
- Change the default *zoom*.filter to observe *wheel* events if the control key is pressed.
- Change the default *zoom*.wheelDelta to go faster if the control key is pressed.

- Don’t set touch-action: none.
- Upgrade to d3-selection 2.

Breaking Changes

D3 6.0 introduces several non-backwards-compatible changes.

- Remove d3.event.
- Change *selection*.on to pass the *event* directly to listeners.
- Change *transition*.on to pass the *event* directly to listeners.
- Change *brush*.on to pass the *event* directly to listeners.
- Change *drag*.on to pass the *event* directly to listeners.
- Change *zoom*.on to pass the *event* directly to listeners.
- Remove d3.mouse; use d3.pointer.
- Remove d3.touch; use d3.pointer.
- Remove d3.touches; use d3.pointers.
- Remove d3.clientPoint; use d3.pointer.
- Remove d3.voronoi; use d3.Delaunay.
- Remove d3.nest; use d3.group and d3.rollup.
- Remove d3.map; use Map.
- Remove d3.set; use Set.
- Remove d3.keys; use Object.keys.
- Remove d3.values; use Object.values.
- Remove d3.entries; use Object.entries.
- Rename d3.histogram to d3.bin.
- Rename d3.scan to d3.leastIndex.
- Change d3.interpolateTransformCss to require absolute units.
- Change d3.format to default to the minus sign instead of hyphen-minus for negative values.

D3 now requires a browser that supports ES2015. For older browsers, you must bring your own transpiler.

Lastly, support for Bower has been dropped; D3 is now exclusively published to npm and GitHub.

See our migration guide for help upgrading.

Changes in D3 5.0

Released March 22, 2018.

D3 5.0 introduces only a few non-backwards-compatible changes.

D3 now uses Promises instead of asynchronous callbacks to load data. Promises simplify the structure of asynchronous code, especially in modern browsers that support async and await. (See this introduction to promises on Observable.) For example, to load a CSV file in v4, you might say:

```
d3.csv("file.csv", function(error, data) {
  if (error) throw error;
  console.log(data);
});
```

In v5, using promises:

```
d3.csv("file.csv").then(function(data) {
  console.log(data);
});
```

Note that you don't need to rethrow the error—the promise will reject automatically, and you can *promise*.catch if desired. Using await, the code is even simpler:

```
const data = await d3.csv("file.csv");
console.log(data);
```

With the adoption of promises, D3 now uses the Fetch API instead of XMLHttpRequest: the d3-request module has been replaced by d3-fetch. Fetch supports many powerful new features, such as streaming responses. D3 5.0 also deprecates and removes the d3-queue module. Use Promise.all to run a batch of asynchronous tasks in parallel, or a helper library such as p-queue to control concurrency.

D3 no longer provides the d3.schemeCategory20* categorical color schemes. These twenty-color schemes were flawed because their grouped design could falsely imply relationships in the data: a shared hue can imply that the encoded data are part of a group (a super-category), while relative lightness can imply order. Instead, D3 now includes d3-scale-chromatic, which implements excellent schemes from ColorBrewer, including categorical, diverging, sequential single-hue and sequential multi-hue schemes. These schemes are available in both discrete and continuous variants.

D3 now provides implementations of marching squares and density estimation via d3-contour! There are two new d3-selection methods: *selection*.clone for inserting clones of the selected nodes, and d3.create for creating detached elements. Geographic projections now support *projection*.angle, which has enabled several fantastic new polyhedral projections by Philippe Rivière.

Lastly, D3's package.json no longer pins exact versions of the dependent D3 modules. This fixes an issue with duplicate installs of D3 modules.

Changes in D3 4.0

Released June 28, 2016.

D3 4.0 is modular. Instead of one library, D3 is now many small libraries that are designed to work together. You can pick and choose which parts to use as you see fit. Each library is maintained in its own repository, allowing decentralized ownership and independent release cycles. The default bundle combines about thirty of these microlibraries.

```
<script src="https://d3js.org/d3.v4.js"></script>
```

As before, you can load optional plugins on top of the default bundle, such as ColorBrewer scales:

```
<script src="https://d3js.org/d3.v4.js"></script>
<script src="https://d3js.org/d3-scale-chromatic.v0.3.js"></script>
```

You are not required to use the default bundle! If you're just using d3-selection, use it as a standalone library. Like the default bundle, you can load D3 microlibraries using vanilla script tags or RequireJS (great for HTTP/2!):

```
<script src="https://d3js.org/d3-selection.v1.js"></script>
```

You can also `cat` D3 microlibraries into a custom bundle, or use tools such as Webpack and Rollup to create optimized bundles. Custom bundles are great for applications that use a subset of D3's features; for example, a React chart library might use D3 for scales and shapes, and React to manipulate the DOM. The D3 microlibraries are written as ES6 modules, and Rollup lets you pick at the symbol level to produce smaller bundles.

Small files are nice, but modularity is also about making D3 more *fun*. Microlibraries are easier to understand, develop and test. They make it easier for new people to get involved and contribute. They reduce the distinction between a "core module" and a "plugin", and increase the pace of development in D3 features.

If you don't care about modularity, you can mostly ignore this change and keep using the default bundle. However, there is one unavoidable consequence of adopting ES6 modules: every symbol in D3 4.0 now shares a flat namespace rather than the nested one of D3 3.x. For example, d3.scale.linear is now d3.scaleLinear, and d3.layout.treemap is now d3.treemap. The adoption of ES6 modules also means that D3 is now written exclusively in strict mode and has better readability. And there have been many other significant improvements to D3's features! (Nearly all of the code from D3 3.x has been rewritten.) These changes are covered below.

Other Global Changes

The default UMD bundle is now anonymous. No `d3` global is exported if AMD or CommonJS is detected. In a vanilla environment, the D3 microlibraries share the `d3` global, even if you load them independently; thus, code you write is the same whether or not you use the default bundle. (See Let's Make a (D3) Plugin for more.) The generated bundle is no longer stored in the Git repository; Bower has been repointed to d3-bower, and you can find the generated files on npm or attached to the latest release. The non-minified default bundle is no longer mangled, making it more readable and preserving inline comments.

To the consternation of some users, 3.x employed Unicode variable names such as λ, φ, τ and π for a concise representation of mathematical operations. A downside of this approach was that a SyntaxError would occur if you loaded the non-minified D3 using ISO-8859-1 instead of UTF-8. 3.x also used Unicode string literals, such as the SI-prefix μ for 1e-6. 4.0 uses only ASCII variable names and ASCII string literals (see rollup-plugin-ascii), avoiding encoding problems.

Table of Contents

- Polygons
- Quadtrees
- Queues
- Random Numbers
- Requests
- Scales
- Selections
- Shapes
- Time Formats
- Time Intervals
- Timers
- Transitions
- Voronoi Diagrams
- Zooming

Arrays (d3-array)

The new d3.scan method performs a linear scan of an array, returning the index of the least element according to the specified comparator. This is similar to d3.min and d3.max, except you can use it to find the position of an extreme element, rather than just calculate an extreme value.

```
var data = [
  {name: "Alice", value: 2},
  {name: "Bob", value: 3},
  {name: "Carol", value: 1},
  {name: "Dwayne", value: 5}
];

var i = d3.scan(data, function(a, b) { return a.value - b.value; }); // 2
data[i]; // {name: "Carol", value: 1}
```

The new d3.ticks and d3.tickStep methods are useful for generating human-readable numeric ticks. These methods are a low-level alternative to *continuous*.ticks from d3-scale. The new implementation is also more accurate, returning the optimal number of ticks as measured by relative error.

```
var ticks = d3.ticks(0, 10, 5); // [0, 2, 4, 6, 8, 10]
```

The d3.range method no longer makes an elaborate attempt to avoid floating-point error when *step* is not an integer. The returned values are strictly defined as *start* + *i* * *step*, where *i* is an integer. (Learn more about floating point math.) d3.range returns the empty array for infinite ranges, rather than throwing an error.

The method signature for optional accessors has been changed to be more consistent with array methods such as *array*.forEach: the accessor is passed the current element (*d*), the index (*i*), and the array (*data*), with *this* as undefined. This affects d3.min, d3.max, d3.extent, d3.sum, d3.mean, d3.median, d3.quantile, d3.variance and d3.deviation. The d3.quantile method previously did not take an accessor. Some methods with optional arguments now treat those arguments as missing if they are null or undefined, rather than strictly checking arguments.length.

The new d3.histogram API replaces d3.layout.histogram. Rather than exposing *bin*.x and *bin*.dx on each returned bin, the histogram exposes *bin*.x0 and *bin*.x1, guaranteeing that *bin*.x0 is exactly equal to *bin*.x1 on the preceding bin. The “frequency” and “probability” modes are no longer supported; each bin is simply an array of elements from the input

data, so *bin*.length is equal to D3 3.x's *bin*.y in frequency mode. To compute a probability distribution, divide the number of elements in each bin by the total number of elements.

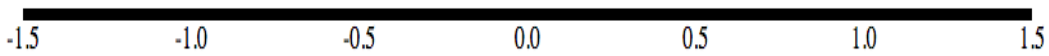
The *histogram*.range method has been renamed *histogram*.domain for consistency with scales. The *histogram*.bins method has been renamed *histogram*.thresholds, and no longer accepts an upper value: *n* thresholds will produce *n* + 1 bins. If you specify a desired number of bins rather than thresholds, d3.histogram now uses d3.ticks to compute nice bin thresholds. In addition to the default Sturges' formula, D3 now implements the Freedman-Diaconis rule and Scott's normal reference rule.

Axes (d3-axis)

To render axes properly in D3 3.x, you needed to style them:

```
<style>

.axis path,
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}

.axis text {
  font: 10px sans-serif;
}

</style>
<script>

d3.select(".axis")
    .call(d3.svg.axis()
        .scale(x)
        .orient("bottom"));

</script>
```

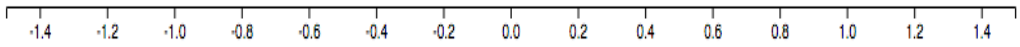
If you didn't, you saw this:



D3 4.0 provides default styles and shorter syntax. In place of d3.svg.axis and *axis*.orient, D3 4.0 now provides four constructors for each orientation: d3.axisTop, d3.axisRight, d3.axisBottom, d3.axisLeft. These constructors accept a scale, so you can reduce all of the above to:

```
<script>

d3.select(".axis")
    .call(d3.axisBottom(x));

</script>
```

And get this:



As before, you can customize the axis appearance either by applying stylesheets or by modifying the axis elements. The default appearance has been changed slightly to offset the axis by a half-pixel; this fixes a crisp-edges rendering issue on Safari where the axis would be drawn two-pixels thick.

There's now an *axis*.tickArguments method, as an alternative to *axis*.ticks that also allows the axis tick arguments to be inspected. The *axis*.tickSize method has been changed to only allow a single argument when setting the tick size. The *axis*.innerTickSize and *axis*.outerTickSize methods have been renamed *axis*.tickSizeInner and *axis*.tickSizeOuter, respectively.

Brushes (d3-brush)

Replacing d3.svg.brush, there are now three classes of brush for brushing along the *x*-dimension, the *y*-dimension, or both: d3.brushX, d3.brushY, d3.brush. Brushes are no longer dependent on scales; instead, each brush defines a selection in screen coordinates. This selection can be inverted if you want to compute the corresponding data domain. And rather than rely on the scales' ranges to determine the brushable area, there is now a *brush*.extent method for setting it. If you do not set the brush extent, it defaults to the full extent of the owner SVG element. The *brush*.clamp method has also been eliminated; brushing is always restricted to the brushable area defined by the brush extent.

Brushes no longer store the active brush selection (*i.e.*, the highlighted region; the brush's position) internally. The brush's position is now stored on any elements to which the brush has been applied. The brush's position is available as *event*.selection within a brush event or by calling d3.brushSelection on a given *element*. To move the brush programmatically, use *brush*.move with a given selection or transition; see the brush snapping example. The *brush*.event method has been removed.

Brush interaction has been improved. By default, brushes now ignore right-clicks intended for the context menu; you can change this behavior using *brush*.filter. Brushes also ignore emulated mouse events on iOS. Holding down SHIFT (⇧) while brushing locks the *x*- or *y*-position of the brush. Holding down META (⌘) while clicking and dragging starts a new selection, rather than translating the existing selection.

The default appearance of the brush has also been improved and slightly simplified. Previously it was necessary to apply styles to the brush to give it a reasonable appearance, such as:

```
.brush .extent {
  stroke: #fff;
```

```
  fill-opacity: .125;
  shape-rendering: crispEdges;
}
```

These styles are now applied by default as attributes; if you want to customize the brush appearance, you can still apply external styles or modify the brush elements. (D3 4.0 features a similar improvement to axes.) A new *brush*.handleSize method lets you override the brush handle size; it defaults to six pixels.

The brush now consumes handled events, making it easier to combine with other interactive behaviors such as dragging and zooming. The *brushstart* and *brushend* events have been renamed to *start* and *end*, respectively. The brush event no longer reports a *event*.mode to distinguish between resizing and dragging the brush.

Chords (d3-chord)

Pursuant to the great namespace flattening:

- d3.layout.chord ↦ d3.chord
- d3.svg.chord ↦ d3.ribbon

For consistency with *arc*.padAngle, *chord*.padding has also been renamed to *ribbon*.padAngle. A new *ribbon*.context method lets you render chord diagrams to Canvas! See also d3-path.

Collections (d3-collection)

The d3.set constructor now accepts an existing set for making a copy. If you pass an array to d3.set, you can also pass a value accessor. This accessor takes the standard arguments: the current element (*d*), the index (*i*), and the array (*data*), with *this* undefined. For example:

```
var yields = [
  {yield: 22.13333, variety: "Manchuria",        year: 1932, site: "Grand Rapids"},
  {yield: 26.76667, variety: "Peatland",         year: 1932, site: "Grand Rapids"},
  {yield: 28.10000, variety: "No. 462",          year: 1931, site: "Duluth"},
  {yield: 38.50000, variety: "Svansota",         year: 1932, site: "Waseca"},
  {yield: 40.46667, variety: "Svansota",         year: 1931, site: "Crookston"},
  {yield: 36.03333, variety: "Peatland",         year: 1932, site: "Waseca"},
  {yield: 34.46667, variety: "Wisconsin No. 38", year: 1931, site: "Grand Rapids"}
];

var sites = d3.set(yields, function(d) { return d.site; }); // Grand Rapids, Duluth,
Waseca, Crookston
```

The d3.map constructor also follows the standard array accessor argument pattern.

The *map*.forEach and *set*.forEach methods have been renamed to *map*.each and *set*.each respectively. The order of arguments for *map*.each has also been changed to *value*, *key* and *map*, while the order of arguments for *set*.each is now *value*, *value* and *set*. This is closer to ES6 *map*.forEach and *set*.forEach. Also like ES6 Map and Set, *map*.set and *set*.add now return the current collection (rather than the added value) to facilitate method chaining. New *map*.clear and *set*.clear methods can be used to empty collections.

The *nest*.map method now always returns a d3.map instance. For a plain object, use *nest*.object instead. When used in conjunction with *nest*.rollup, *nest*.entries now returns {key, value} objects for the leaf entries, instead of {key, values}. This makes *nest*.rollup easier to use in conjunction with hierarchies, as in this Nest Treemap example.

Colors (d3-color)

All colors now have opacity exposed as *color*.opacity, which is a number in [0, 1]. You can pass an optional opacity argument to the color space constructors d3.rgb, d3.hsl, d3.lab, d3.hcl or d3.cubehelix.

You can now parse rgba(…) and hsla(…) CSS color specifiers or the string “transparent” using d3.color. The “transparent” color is defined as an RGB color with zero opacity and undefined red, green and blue channels; this differs slightly from CSS which defines it as transparent black, but is useful for simplifying color interpolation logic where either the starting or ending color has undefined channels. The *color*.toString method now likewise returns an rgb(…) or rgba(…) string with integer channel values, not the hexadecimal RGB format, consistent with CSS computed values. This improves performance by short-circuiting transitions when the element’s starting style matches its ending style.

The new d3.color method is the primary method for parsing colors: it returns a d3.color instance in the appropriate color space, or null if the CSS color specifier is invalid. For example:

```
var red = d3.color("hsl(0, 80%, 50%)"); // {h: 0, l: 0.5, s: 0.8, opacity: 1}
```

The parsing implementation is now more robust. For example, you can no longer mix integers and percentages in rgb(…), and it correctly handles whitespace, decimal points, number signs, and other edge cases. The color space constructors d3.rgb, d3.hsl, d3.lab, d3.hcl and d3.cubehelix now always return a copy of the input color, converted to the corresponding color space. While *color*.rgb remains, *rgb*.hsl has been removed; use d3.hsl to convert a color to the RGB color space.

The RGB color space no longer greedily quantizes and clamps channel values when creating colors, improving accuracy in color space conversion. Quantization and clamping now occurs in *color*.toString when formatting a color for display. You can use the new *color*.displayable to test whether a color is out-of-gamut.

The *rgb*.brighter method no longer special-cases black. This is a multiplicative operator, defining a new color r', g', b' where $r' = r \times pow(0.7, k)$, $g' = g \times pow(0.7, k)$ and $b' = b \times pow(0.7, k)$; a brighter black is still black.

There’s a new d3.cubehelix color space, generalizing Dave Green’s color scheme! (See also d3.interpolateCubehelixDefault from d3-scale.) You can continue to define your own custom color spaces, too; see d3-hsv for an example.

Dispatches (d3-dispatch)

Rather than decorating the *dispatch* object with each event type, the dispatch object now exposes generic *dispatch*.call and *dispatch*.apply methods which take the *type* string as the first argument. For example, in D3 3.x, you might say:

```
dispatcher.foo.call(that, "Hello, Foo!");
```

To dispatch a *foo* event in D3 4.0, you’d say:

```
dispatcher.call("foo", that, "Hello, Foo!");
```

The *dispatch*.on method now accepts multiple typenames, allowing you to add or remove listeners for multiple events simultaneously. For example, to send both *foo* and *bar* events to the same listener:

```
dispatcher.on("foo bar", function(message) {
  console.log(message);
});
```

This matches the new behavior of *selection*.on in d3-selection. The *dispatch*.on method now validates that the specifier *listener* is a function, rather than throwing an error in the future.

The new implementation d3.dispatch is faster, using fewer closures to improve performance. There's also a new *dispatch*.copy method for making a copy of a dispatcher; this is used by d3-transition to improve the performance of transitions in the common case where all elements in a transition have the same transition event listeners.

Dragging (d3-drag)

The drag behavior d3.behavior.drag has been renamed to d3.drag. The *drag*.origin method has been replaced by *drag*.subject, which allows you to define the thing being dragged at the start of a drag gesture. This is particularly useful with Canvas, where draggable objects typically share a Canvas element (as opposed to SVG, where draggable objects typically have distinct DOM elements); see the circle dragging example.

A new *drag*.container method lets you override the parent element that defines the drag gesture coordinate system. This defaults to the parent node of the element to which the drag behavior was applied. For dragging on Canvas elements, you probably want to use the Canvas element as the container.

Drag events now expose an *event*.on method for registering temporary listeners for duration of the current drag gesture; these listeners can capture state for the current gesture, such as the thing being dragged. A new *event*.active property lets you detect whether multiple (multitouch) drag gestures are active concurrently. The *dragstart* and *dragend* events have been renamed to *start* and *end*. By default, drag behaviors now ignore right-clicks intended for the context menu; use *drag*.filter to control which events are ignored. The drag behavior also ignores emulated mouse events on iOS. The drag behavior now consumes handled events, making it easier to combine with other interactive behaviors such as zooming.

The new d3.dragEnable and d3.dragDisable methods provide a low-level API for implementing drag gestures across browsers and devices. These methods are also used by other D3 components, such as the brush.

Delimiter-Separated Values (d3-dsv)

Pursuant to the great namespace flattening, various CSV and TSV methods have new names:

- d3.csv.parse ↦ d3.csvParse
- d3.csv.parseRows ↦ d3.csvParseRows
- d3.csv.format ↦ d3.csvFormat
- d3.csv.formatRows ↦ d3.csvFormatRows
- d3.tsv.parse ↦ d3.tsvParse
- d3.tsv.parseRows ↦ d3.tsvParseRows
- d3.tsv.format ↦ d3.tsvFormat
- d3.tsv.formatRows ↦ d3.tsvFormatRows

The d3.csv and d3.tsv methods for loading files of the corresponding formats have not been renamed, however! Those are defined in d3-request.There's no longer a d3.dsv method, which served the triple purpose of defining a DSV formatter, a DSV parser and a DSV requestor; instead, there's just d3.dsvFormat which you can use to define a DSV formatter and parser. You can use *request*.response to make a request and then parse the response body, or just use d3.text.

The *dsv*.parse method now exposes the column names and their input order as *data*.columns. For example:

```
d3.csv("cars.csv", function(error, data) {
  if (error) throw error;
  console.log(data.columns); // ["Year", "Make", "Model", "Length"]
});
```

You can likewise pass an optional array of column names to *dsv*.format to format only a subset of columns, or to specify the column order explicitly:

```
var string = d3.csvFormat(data, ["Year", "Model", "Length"]);
```

The parser is a bit faster and the formatter is a bit more robust: inputs are coerced to strings before formatting, fixing an obscure crash, and deprecated support for falling back to *dsv*.formatRows when the input *data* is an array of arrays has been removed.

Easings (d3-ease)

D3 3.x used strings, such as “cubic-in-out”, to identify easing methods; these strings could be passed to d3.ease or *transition*.ease. D3 4.0 uses symbols instead, such as d3.easeCubicInOut. Symbols are simpler and cleaner. They work well with Rollup to produce smaller custom bundles. You can still define your own custom easing function, too, if desired. Here’s the full list of equivalents:

- linear ↦ d3.easeLinear[1]
- linear-in ↦ d3.easeLinear[1]
- linear-out ↦ d3.easeLinear[1]
- linear-in-out ↦ d3.easeLinear[1]
- linear-out-in ↦ d3.easeLinear[1]
- poly-in ↦ d3.easePolyIn
- poly-out ↦ d3.easePolyOut
- poly-in-out ↦ d3.easePolyInOut
- poly-out-in ↦ REMOVED[2]
- quad-in ↦ d3.easeQuadIn
- quad-out ↦ d3.easeQuadOut
- quad-in-out ↦ d3.easeQuadInOut
- quad-out-in ↦ REMOVED[2]
- cubic-in ↦ d3.easeCubicIn
- cubic-out ↦ d3.easeCubicOut
- cubic-in-out ↦ d3.easeCubicInOut
- cubic-out-in ↦ REMOVED[2]
- sin-in ↦ d3.easeSinIn
- sin-out ↦ d3.easeSinOut
- sin-in-out ↦ d3.easeSinInOut
- sin-out-in ↦ REMOVED[2]
- exp-in ↦ d3.easeExpIn
- exp-out ↦ d3.easeExpOut
- exp-in-out ↦ d3.easeExpInOut
- exp-out-in ↦ REMOVED[2]
- circle-in ↦ d3.easeCircleIn
- circle-out ↦ d3.easeCircleOut

- circle-in-out ↦ d3.easeCircleInOut
- circle-out-in ↦ REMOVED[2]
- elastic-in ↦ d3.easeElasticOut[2]
- elastic-out ↦ d3.easeElasticIn[2]
- elastic-in-out ↦ REMOVED[2]
- elastic-out-in ↦ d3.easeElasticInOut[2]
- back-in ↦ d3.easeBackIn
- back-out ↦ d3.easeBackOut
- back-in-out ↦ d3.easeBackInOut
- back-out-in ↦ REMOVED[2]
- bounce-in ↦ d3.easeBounceOut[2]
- bounce-out ↦ d3.easeBounceIn[2]
- bounce-in-out ↦ REMOVED[2]
- bounce-out-in ↦ d3.easeBounceInOut[2]

[1] The -in, -out and -in-out variants of linear easing are identical, so there's just d3.easeLinear.
[2] Elastic and bounce easing were inadvertently reversed in 3.x, so 4.0 eliminates -out-in easing!

For convenience, there are also default aliases for each easing method. For example, d3.easeCubic is an alias for d3.easeCubicInOut. Most default to -in-out; the exceptions are d3.easeBounce and d3.easeElastic, which default to -out.

Rather than pass optional arguments to d3.ease or *transition*.ease, parameterizable easing functions now have named parameters: *poly*.exponent, *elastic*.amplitude, *elastic*.period and *back*.overshoot. For example, in D3 3.x you might say:

```
var e = d3.ease("elastic-out-in", 1.2);
```

The equivalent in D3 4.0 is:

```
var e = d3.easeElastic.amplitude(1.2);
```

Many of the easing functions have been optimized for performance and accuracy. Several bugs have been fixed, as well, such as the interpretation of the overshoot parameter for back easing, and the period parameter for elastic easing. Also, d3-transition now explicitly guarantees that the last tick of the transition happens at exactly $t = 1$, avoiding floating point errors in some easing functions.

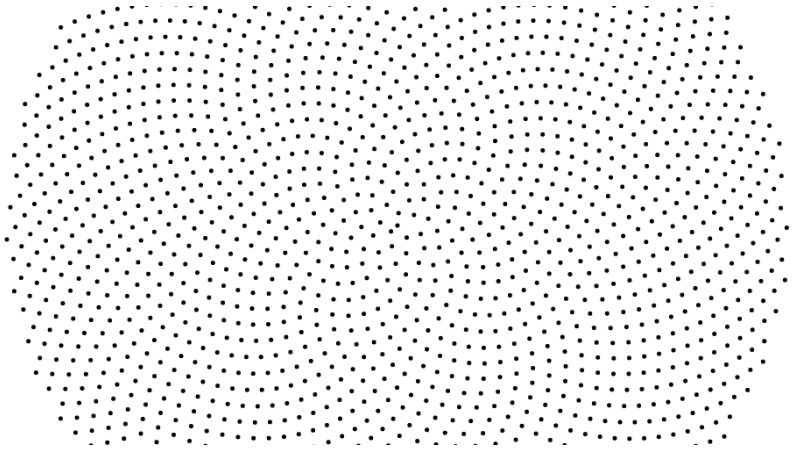
There's now a nice visual reference and an animated reference to the new easing functions, too!

Forces (d3-force)

The force layout d3.layout.force has been renamed to d3.forceSimulation. The force simulation now uses velocity Verlet integration rather than position Verlet, tracking the nodes' positions (*node*.x, *node*.y) and velocities (*node*.vx, *node*.vy) rather than their previous positions (*node*.px, *node*.py).

Rather than hard-coding a set of built-in forces, the force simulation is now extensible: you specify which forces you want! The approach affords greater flexibility through composition. The new forces are more flexible, too: force parameters can typically be configured per-node or per-link. There are separate positioning forces for *x* and *y* that replace *force*.gravity; *x*.x and *y*.y replace *force*.size. The new link force replaces *force*.linkStrength and employs better default heuristics to improve stability. The new many-body force replaces *force*.charge and supports a new minimum-distance parameter and performance improvements thanks to 4.0's new quadtrees. There are also brand-new forces for centering nodes and collision resolution.

The new forces and simulation have been carefully crafted to avoid nondeterminism. Rather than initializing nodes randomly, if the nodes do not have preset positions, they are placed in a phyllotaxis pattern:



Random jitter is still needed to resolve link, collision and many-body forces if there are coincident nodes, but at least in the common case, the force simulation (and the resulting force-directed graph layout) is now consistent across browsers and reloads. D3 no longer plays dice!

The force simulation has several new methods for greater control over heating, such as *simulation*.alphaMin and *simulation*.alphaDecay, and the internal timer. Calling *simulation*.alpha now has no effect on the internal timer, which is controlled independently via *simulation*.stop and *simulation*.restart. The force layout's internal timer now starts automatically on creation, removing *force*.start. As in 3.x, you can advance the simulation manually using *simulation*.tick. The *force*.friction parameter is replaced by *simulation*.velocityDecay. A new *simulation*.alphaTarget method allows you to set the desired alpha (temperature) of the simulation, such that the simulation can be smoothly reheated during interaction, and then smoothly cooled again. This improves the stability of the graph during interaction.

The force layout no longer depends on the drag behavior, though you can certainly create draggable force-directed graphs! Set *node*.fx and *node*.fy to fix a node's position. As an alternative to a Voronoi SVG overlay, you can now use *simulation*.find to find the closest node to a pointer.

Number Formats (d3-format)

If a precision is not specified, the formatting behavior has changed: there is now a default precision of 6 for all directives except *none*, which defaults to 12. In 3.x, if you did not specify a precision, the number was formatted using its shortest unique representation (per *number*.toString); this could lead to unexpected digits due to floating point math. The new default precision in 4.0 produces more consistent results:

```
var f = d3.format("e");
f(42);         // "4.200000e+1"
f(0.1 + 0.2); // "3.000000e-1"
```

To trim insignificant trailing zeroes, use the *none* directive, which is similar `g`. For example:

```
var f = d3.format(".3");
f(0.12345);   // "0.123"
f(0.10000);   // "0.1"
f(0.1 + 0.2); // "0.3"
```

Under the hood, number formatting has improved accuracy with very large and very small numbers by using *number*.toExponential rather than Math.log to extract the mantissa and exponent. Negative zero (-0, an IEEE 754 construct) and very small numbers that round to zero are now formatted as unsigned zero. The inherently unsafe d3.round method has been removed, along with d3.requote.

The d3.formatPrefix method has been changed. Rather than returning an SI-prefix string, it returns an SI-prefix format function for a given *specifier* and reference *value*. For example, to format thousands:

```
var f = d3.formatPrefix(",.0", 1e3);
f(1e3); // "1k"
f(1e4); // "10k"
f(1e5); // "100k"
f(1e6); // "1,000k"
```

Unlike the `s` format directive, d3.formatPrefix always employs the same SI-prefix, producing consistent results:

```
var f = d3.format(".0s");
f(1e3); // "1k"
f(1e4); // "10k"
f(1e5); // "100k"
f(1e6); // "1M"
```

The new `(` sign option uses parentheses for negative values. This is particularly useful in conjunction with `$`. For example:

```
d3.format("+.0f")(-42);  // "-42"
d3.format("(.0f")(-42);  // "(42)"
d3.format("+$.0f")(-42); // "-$42"
d3.format("($.0f")(-42); // "($42)"
```

The new `=` align option places any sign and symbol to the left of any padding:

```
d3.format(">6d")(-42);  // "   -42"
d3.format("=6d")(-42);  // "-   42"
d3.format(">(6d")(-42); // "  (42)"
d3.format("=(6d")(-42); // "(  42)"
```

The `b`, `o`, `d` and `x` directives now round to the nearest integer, rather than returning the empty string for non-integers:

```
d3.format("b")(41.9); // "101010"
d3.format("o")(41.9); // "52"
d3.format("d")(41.9); // "42"
d3.format("x")(41.9); // "2a"
```

The `c` directive is now for character data (*i.e.*, literal strings), not for character codes. The is useful if you just want to apply padding and alignment and don't care about formatting numbers. For example, the infamous left-pad (as well as center- and right-pad!) can be conveniently implemented as:

```
d3.format(">10c")("foo"); // "       foo"
d3.format("^10c")("foo"); // "   foo    "
d3.format("<10c")("foo"); // "foo       "
```

There are several new methods for computing suggested decimal precisions; these are used by d3-scale for tick formatting, and are helpful for implementing custom number formats: d3.precisionFixed, d3.precisionPrefix and d3.precisionRound. There's also a new d3.formatSpecifier method for parsing, validating and debugging format specifiers; it's also good for deriving related format specifiers, such as when you want to substitute the precision automatically.

You can now set the default locale using d3.formatDefaultLocale! The locales are published as JSON to npm.

Geographies (d3-geo)

Pursuant to the great namespace flattening, various methods have new names:

- d3.geo.graticule ↦ d3.geoGraticule
- d3.geo.circle ↦ d3.geoCircle
- d3.geo.area ↦ d3.geoArea
- d3.geo.bounds ↦ d3.geoBounds
- d3.geo.centroid ↦ d3.geoCentroid
- d3.geo.distance ↦ d3.geoDistance
- d3.geo.interpolate ↦ d3.geoInterpolate
- d3.geo.length ↦ d3.geoLength
- d3.geo.rotation ↦ d3.geoRotation
- d3.geo.stream ↦ d3.geoStream
- d3.geo.path ↦ d3.geoPath
- d3.geo.projection ↦ d3.geoProjection
- d3.geo.projectionMutator ↦ d3.geoProjectionMutator
- d3.geo.albers ↦ d3.geoAlbers
- d3.geo.albersUsa ↦ d3.geoAlbersUsa
- d3.geo.azimuthalEqualArea ↦ d3.geoAzimuthalEqualArea
- d3.geo.azimuthalEquidistant ↦ d3.geoAzimuthalEquidistant
- d3.geo.conicConformal ↦ d3.geoConicConformal
- d3.geo.conicEqualArea ↦ d3.geoConicEqualArea
- d3.geo.conicEquidistant ↦ d3.geoConicEquidistant
- d3.geo.equirectangular ↦ d3.geoEquirectangular
- d3.geo.gnomonic ↦ d3.geoGnomonic
- d3.geo.mercator ↦ d3.geoMercator
- d3.geo.orthographic ↦ d3.geoOrthographic
- d3.geo.stereographic ↦ d3.geoStereographic
- d3.geo.transverseMercator ↦ d3.geoTransverseMercator

Also renamed for consistency:

- *circle*.origin ↦ *circle*.center
- *circle*.angle ↦ *circle*.radius
- *graticule*.majorExtent ↦ *graticule*.extentMajor
- *graticule*.minorExtent ↦ *graticule*.extentMinor
- *graticule*.majorStep ↦ *graticule*.stepMajor
- *graticule*.minorStep ↦ *graticule*.stepMinor

Projections now have more appropriate defaults. For example, d3.geoOrthographic has a 90° clip angle by default, showing only the front hemisphere, and d3.geoGnomonic has a default 60° clip angle. The default projection for d3.geoPath is now null rather than d3.geoAlbersUsa; a null projection is used with pre-projected geometry and is typically faster to render.

"Fallback projections"—when you pass a function rather than a projection to *path*.projection—are no longer supported. For geographic projections, use d3.geoProjection or d3.geoProjectionMutator to define a custom projection. For arbitrary geometry transformations, implement the stream interface; see also d3.geoTransform. The "raw" projections (e.g., d3.geo.equirectangular.raw) are no longer exported.

Hierarchies (d3-hierarchy)

Pursuant to the great namespace flattening:

- d3.layout.cluster ↦ d3.cluster
- d3.layout.hierarchy ↦ d3.hierarchy
- d3.layout.pack ↦ d3.pack
- d3.layout.partition ↦ d3.partition
- d3.layout.tree ↦ d3.tree
- d3.layout.treemap ↦ d3.treemap

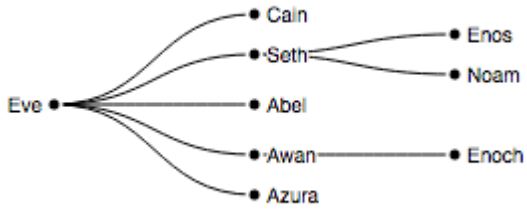
As an alternative to using JSON to represent hierarchical data (such as the "flare.json format" used by many D3 examples), the new d3.stratify operator simplifies the conversion of tabular data to hierarchical data! This is convenient if you already have data in a tabular format, such as the result of a SQL query or a CSV file:

```
name,parent
Eve,
Cain,Eve
Seth,Eve
Enos,Seth
Noam,Seth
Abel,Eve
Awan,Eve
Enoch,Awan
Azura,Eve
```

To convert this to a root *node*:

```
var root = d3.stratify()
    .id(function(d) { return d.name; })
    .parentId(function(d) { return d.parent; })
    (nodes);
```

The resulting *root* can be passed to d3.tree to produce a tree diagram like this:



Root nodes can also be created from JSON data using d3.hierarchy. The hierarchy layouts now take these root nodes as input rather than operating directly on JSON data, which helps to provide a cleaner separation between the input data and the computed layout. (For example, use *node*.copy to isolate layout changes.) It also simplifies the API: rather than each hierarchy layout needing to implement value and sorting accessors, there are now generic *node*.sum and *node*.sort methods that work with any hierarchy layout.

The new d3.hierarchy API also provides a richer set of methods for manipulating hierarchical data. For example, to generate an array of all nodes in topological order, use *node*.descendants; for just leaf nodes, use *node*.leaves. To highlight the ancestors of a given *node* on mouseover, use *node*.ancestors. To generate an array of {source, target} links for a given hierarchy, use *node*.links; this replaces *treemap*.links and similar methods on the other layouts. The new *node*.path method replaces d3.layout.bundle; see also d3.curveBundle for hierarchical edge bundling.

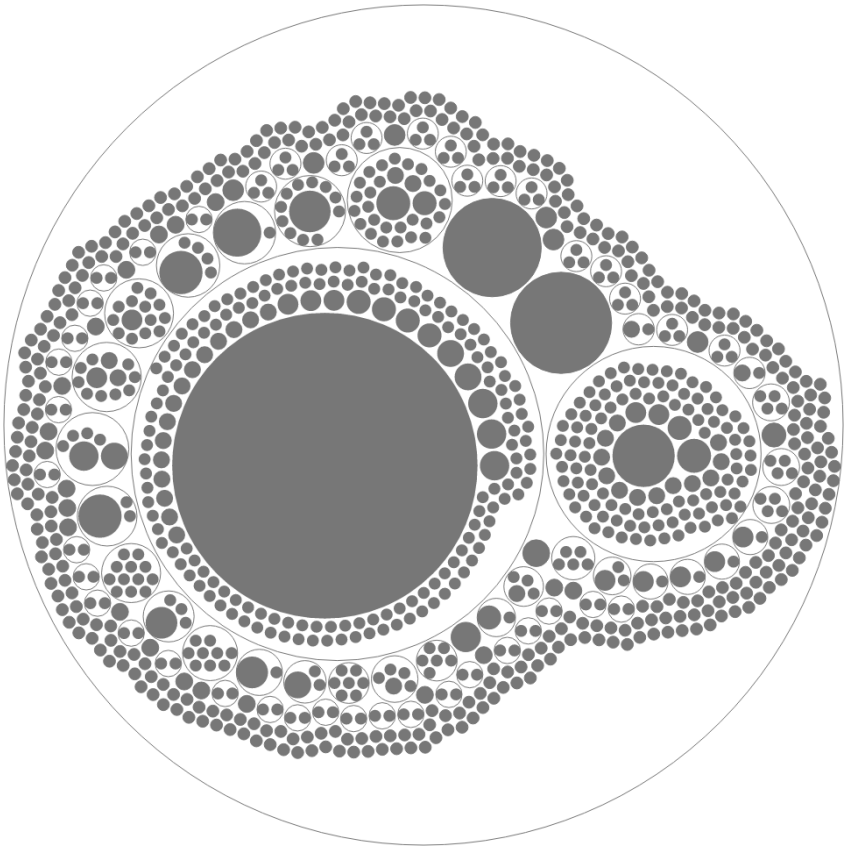
The hierarchy layouts have been rewritten using new, non-recursive traversal methods (*node*.each, *node*.eachAfter and *node*.eachBefore), improving performance on large datasets. The d3.tree layout no longer uses a *node*._ field to store temporary state during layout.

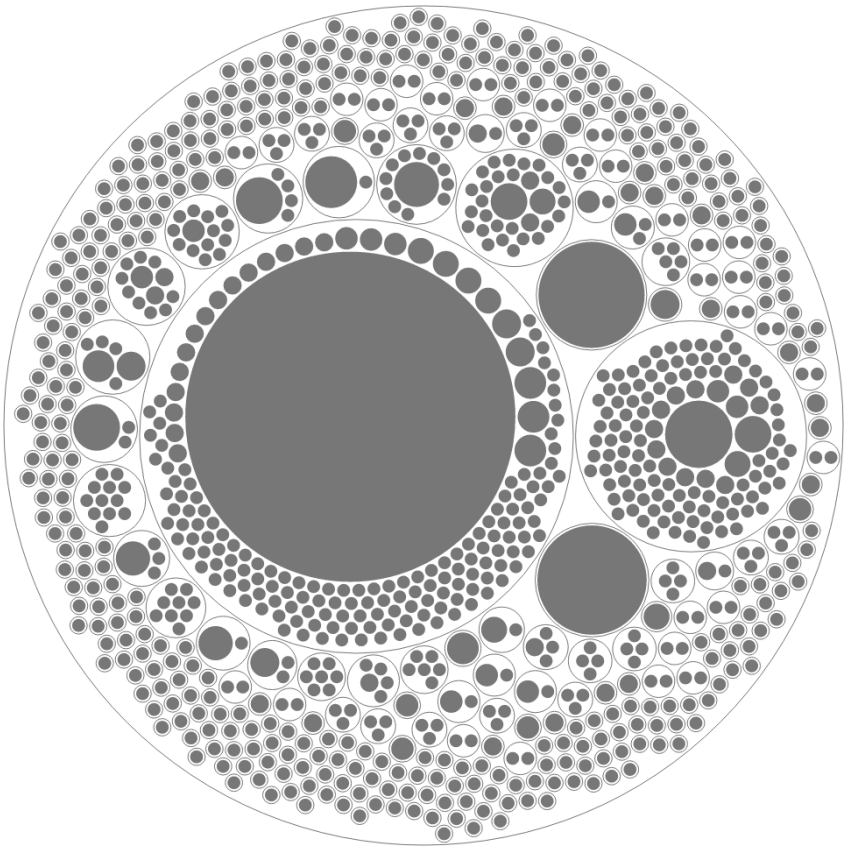
Treemap tiling is now extensible via *treemap*.tile! The default squarified tiling algorithm, d3.treemapSquarify, has been completely rewritten, improving performance and fixing bugs in padding and rounding. The *treemap*.sticky method has been replaced with the d3.treemapResquarify, which is identical to d3.treemapSquarify except it performs stable neighbor-preserving updates. The *treemap*.ratio method has been replaced with *squarify*.ratio. And there's a new d3.treemapBinary for binary treemaps!

Treemap padding has also been improved. The treemap now distinguishes between outer padding that separates a parent from its children, and inner padding that separates adjacent siblings. You can set the top-, right-, bottom- and left-outer padding separately. There are new examples for the traditional nested treemap and for Lü and Fogarty's cascaded treemap. And there's a new example demonstrating d3.nest with d3.treemap.

The space-filling layouts d3.treemap and d3.partition now output *x0*, *x1*, *y0*, *y1* on each node instead of *x0*, *dx*, *y0*, *dy*. This improves accuracy by ensuring that the edges of adjacent cells are exactly equal, rather than sometimes being slightly off due to floating point math. The partition layout now supports rounding and padding.

The circle-packing layout, d3.pack, has been completely rewritten to better implement Wang et al.'s algorithm, fixing major bugs and improving results! Welzl's algorithm is now used to compute the exact smallest enclosing circle for each parent, rather than the approximate answer used by Wang et al. The 3.x output is shown on the left; 4.0 is shown on the right:





A non-hierarchical implementation is also available as d3.packSiblings, and the smallest enclosing circle implementation is available as d3.packEnclose. Pack padding now applies between a parent and its children, as well as between adjacent siblings. In addition, you can now specify padding as a function that is computed dynamically for each parent.

Internals

The d3.rebind method has been removed. (See the 3.x source.) If you want to wrap a getter-setter method, the recommend pattern is to implement a wrapper method and check the return value. For example, given a *component* that uses an internal *dispatch*, *component*.on can rebind *dispatch*.on as follows:

```
component.on = function() {
  var value = dispatch.on.apply(dispatch, arguments);
  return value === dispatch ? component : value;
};
```

The d3.functor method has been removed. (See the 3.x source.) If you want to promote a constant value to a function, the recommended pattern is to implement a closure that returns the constant value. If desired, you can use a helper method as follows:

```
function constant(x) {
  return function() {
    return x;
  };
}
```

Given a value *x*, to promote *x* to a function if it is not already:

```
var fx = typeof x === "function" ? x : constant(x);
```

Interpolators (d3-interpolate)

The d3.interpolate method no longer delegates to d3.interpolators, which has been removed; its behavior is now defined by the library. It is now slightly faster in the common case that *b* is a number. It only uses d3.interpolateRgb if *b* is a valid CSS color specifier (and not approximately one). And if the end value *b* is null, undefined, true or false, d3.interpolate now returns a constant function which always returns *b*.

The behavior of d3.interpolateObject and d3.interpolateArray has changed slightly with respect to properties or elements in the start value *a* that do not exist in the end value *b*: these properties and elements are now ignored, such that the ending value of the interpolator at *t* = 1 is now precisely equal to *b*. So, in 3.x:

```
d3.interpolateObject({foo: 2, bar: 1}, {foo: 3})(0.5); // {bar: 1, foo: 2.5} in 3.x
```

Whereas in 4.0, *a*.bar is ignored:

```
d3.interpolateObject({foo: 2, bar: 1}, {foo: 3})(0.5); // {foo: 2.5} in 4.0
```

If *a* or *b* are undefined or not an object, they are now implicitly converted to the empty object or empty array as appropriate, rather than throwing a TypeError.

The d3.interpolateTransform interpolator has been renamed to d3.interpolateTransformSvg, and there is a new d3.interpolateTransformCss to interpolate CSS transforms! This allows d3-transition to automatically interpolate both the SVG transform attribute and the CSS transform style property. (Note, however, that only 2D CSS transforms are supported.) The d3.transform method has been removed.

Color space interpolators now interpolate opacity (see d3-color) and return rgb(…) or rgba(…) CSS color specifier strings rather than using the RGB hexadecimal format. This is necessary to support opacity interpolation, but is also beneficial because it matches CSS computed values. When a channel in the start color *a* is undefined, color interpolators now use the corresponding channel value from the end color *b*, or *vice versa*. This logic previously applied to some channels (such as saturation in HSL), but now applies to all channels in all color spaces, and is especially useful when interpolating to or from transparent.

There are now "long" versions of cylindrical color space interpolators: d3.interpolateHslLong, d3.interpolateHclLong and d3.interpolateCubehelixLong. These interpolators use linear interpolation of hue, rather than using the shortest path around the 360° hue circle. See d3.interpolateRainbow for an example. The Cubehelix color space is now supported by d3-color, and so there are now d3.interpolateCubehelix and d3.interpolateCubehelixLong interpolators.

Gamma-corrected color interpolation is now supported for both RGB and Cubehelix color spaces as *interpolate*.gamma. For example, to interpolate from purple to orange with a gamma of 2.2 in RGB space:

```
var interpolate = d3.interpolateRgb.gamma(2.2)("purple", "orange");
```

There are new interpolators for uniform non-rational B-splines! These are useful for smoothly interpolating between an arbitrary sequence of values from *t* = 0 to *t* = 1, such as to generate a smooth color gradient from a discrete set of colors. The d3.interpolateBasis and d3.interpolateBasisClosed interpolators generate one-dimensional B-splines, while d3.interpolateRgbBasis and d3.interpolateRgbBasisClosed generate three-dimensional B-splines through RGB color space. These are used by d3-scale-chromatic to generate continuous color scales from ColorBrewer's discrete color schemes, such as PiYG.

There's also now a d3.quantize method for generating uniformly-spaced discrete samples from a continuous interpolator. This is useful for taking one of the built-in color scales (such as d3.interpolateViridis) and quantizing it for use with d3.scaleQuantize, d3.scaleQuantile or d3.scaleThreshold.

Paths (d3-path)

The d3.path serializer implements the CanvasPathMethods API, allowing you to write code that can render to either Canvas or SVG. For example, given some code that draws to a canvas:

```
function drawCircle(context, radius) {
  context.moveTo(radius, 0);
  context.arc(0, 0, radius, 0, 2 * Math.PI);
}
```

You can render to SVG as follows:

```
var context = d3.path();
drawCircle(context, 40);
pathElement.setAttribute("d", context.toString());
```

The path serializer enables d3-shape to support both Canvas and SVG; see *line*.context and *area*.context, for example.

Polygons (d3-polygon)

There's no longer a d3.geom.polygon constructor; instead you just pass an array of vertices to the polygon methods. So instead of *polygon*.area and *polygon*.centroid, there's d3.polygonArea and d3.polygonCentroid. There are also new d3.polygonContains and d3.polygonLength methods. There's no longer an equivalent to *polygon*.clip, but if Sutherland–Hodgman clipping is needed, please file a feature request.

The d3.geom.hull operator has been simplified: instead of an operator with *hull*.x and *hull*.y accessors, there's just the d3.polygonHull method which takes an array of points and returns the convex hull.

Quadtrees (d3-quadtree)

The d3.geom.quadtree method has been replaced by d3.quadtree. 4.0 removes the concept of quadtree "generators" (configurable functions that build a quadtree from an array of data); there are now just quadtrees, which you can create via d3.quadtree and add data to via *quadtree*.add and *quadtree*.addAll. This code in 3.x:

```
var quadtree = d3.geom.quadtree()
    .extent([[0, 0], [width, height]])
    (data);
```

Can be rewritten in 4.0 as:

```
var quadtree = d3.quadtree()
    .extent([[0, 0], [width, height]])
    .addAll(data);
```

The new quadtree implementation is vastly improved! It is no longer recursive, avoiding stack overflows when there are large numbers of coincident points. The internal storage is now more efficient, and the implementation is also faster; constructing a quadtree of 1M normally-distributed points takes about one second in 4.0, as compared to three seconds in 3.x.

The change in internal *node* structure affects *quadtree*.visit: use *node*.length to distinguish leaf nodes from internal nodes. For example, to iterate over all data in a quadtree:

```
quadtree.visit(function(node) {
  if (!node.length) {
    do {
      console.log(node.data);
    } while (node = node.next)
  }
});
```

There's a new *quadtree*.visitAfter method for visiting nodes in post-order traversal. This feature is used in d3-force to implement the Barnes–Hut approximation.

You can now remove data from a quadtree using *quadtree*.remove and *quadtree*.removeAll. When adding data to a quadtree, the quadtree will now expand its extent by repeated doubling if the new point is outside the existing extent of the quadtree. There are also *quadtree*.extent and *quadtree*.cover methods for explicitly expanding the extent of the quadtree after creation.

Quadtrees support several new utility methods: *quadtree*.copy returns a copy of the quadtree sharing the same data; *quadtree*.data generates an array of all data in the quadtree; *quadtree*.size returns the number of data points in the quadtree; and *quadtree*.root returns the root node, which is useful for manual traversal of the quadtree. The *quadtree*.find method now takes an optional search radius, which is useful for pointer-based selection in force-directed graphs.

Queues (d3-queue)

Formerly known as Queue.js and queue-async, d3.queue is now included in the default bundle, making it easy to load data files in parallel. It has been rewritten with fewer closures to improve performance, and there are now stricter checks in place to guarantee well-defined behavior. You can now use instanceof d3.queue and inspect the queue's internal private state.

Random Numbers (d3-random)

Pursuant to the great namespace flattening, the random number generators have new names:

- d3.random.normal ↦ d3.randomNormal
- d3.random.logNormal ↦ d3.randomLogNormal
- d3.random.bates ↦ d3.randomBates
- d3.random.irwinHall ↦ d3.randomIrwinHall

There are also new random number generators for exponential and uniform distributions. The normal and log-normal random generators have been optimized.

Requests (d3-request)

The d3.xhr method has been renamed to d3.request. Basic authentication is now supported using *request*.user and *request*.password. You can now configure a timeout using *request*.timeout.

If an error occurs, the corresponding ProgressEvent of type "error" is now passed to the error listener, rather than the XMLHttpRequest. Likewise, the ProgressEvent is passed to progress event listeners, rather than using d3.event. If d3.xml encounters an error parsing XML, this error is now reported to error listeners rather than returning a null response.

The d3.request, d3.text and d3.xml methods no longer take an optional mime type as the second argument; use *request*.mimeType instead. For example:

```
d3.xml("file.svg").mimeType("image/svg+xml").get(function(error, svg) {
  …
});
```

With the exception of d3.html and d3.xml, Node is now supported via node-XMLHttpRequest.

Scales (d3-scale)

Pursuant to the great namespace flattening:

- d3.scale.linear ↦ d3.scaleLinear
- d3.scale.sqrt ↦ d3.scaleSqrt
- d3.scale.pow ↦ d3.scalePow
- d3.scale.log ↦ d3.scaleLog

- d3.scale.quantize ↦ d3.scaleQuantize
- d3.scale.threshold ↦ d3.scaleThreshold
- d3.scale.quantile ↦ d3.scaleQuantile
- d3.scale.identity ↦ d3.scaleIdentity
- d3.scale.ordinal ↦ d3.scaleOrdinal
- d3.time.scale ↦ d3.scaleTime
- d3.time.scale.utc ↦ d3.scaleUtc

Scales now generate ticks in the same order as the domain: if you have a descending domain, you now get descending ticks. This change affects the order of tick elements generated by axes. For example:

```
d3.scaleLinear().domain([10, 0]).ticks(5); // [10, 8, 6, 4, 2, 0]
```

Log tick formatting now assumes a default *count* of ten, not Infinity, if not specified. Log scales with domains that span many powers (such as from 1e+3 to 1e+29) now return only one tick per power rather than returning *base* ticks per power. Non-linear quantitative scales are slightly more accurate.

You can now control whether an ordinal scale's domain is implicitly extended when the scale is passed a value that is not already in its domain. By default, *ordinal*.unknown is d3.scaleImplicit, causing unknown values to be added to the domain:

```
var x = d3.scaleOrdinal()
    .domain([0, 1])
    .range(["red", "green", "blue"]);

x.domain(); // [0, 1]
x(2); // "blue"
x.domain(); // [0, 1, 2]
```

By setting *ordinal*.unknown, you instead define the output value for unknown inputs. This is particularly useful for choropleth maps where you want to assign a color to missing data.

```
var x = d3.scaleOrdinal()
    .domain([0, 1])
    .range(["red", "green", "blue"])
    .unknown(undefined);

x.domain(); // [0, 1]
x(2); // undefined
x.domain(); // [0, 1]
```

The *ordinal*.rangeBands and *ordinal*.rangeRoundBands methods have been replaced with a new subclass of ordinal scale: band scales. The following code in 3.x:

```
var x = d3.scale.ordinal()
    .domain(["a", "b", "c"])
    .rangeBands([0, width]);
```

Is equivalent to this in 4.0:

```
var x = d3.scaleBand()
    .domain(["a", "b", "c"])
    .range([0, width]);
```

The new *band*.padding, *band*.paddingInner and *band*.paddingOuter methods replace the optional arguments to *ordinal*.rangeBands. The new *band*.bandwidth and *band*.step methods replace *ordinal*.rangeBand. There's also a new *band*.align method which you can use to control how the extra space outside the bands is distributed, say to shift columns closer to the *y*-axis.

Similarly, the *ordinal*.rangePoints and *ordinal*.rangeRoundPoints methods have been replaced with a new subclass of ordinal scale: point scales. The following code in 3.x:

```
var x = d3.scale.ordinal()
    .domain(["a", "b", "c"])
    .rangePoints([0, width]);
```

Is equivalent to this in 4.0:

```
var x = d3.scalePoint()
    .domain(["a", "b", "c"])
    .range([0, width]);
```

The new *point*.padding method replaces the optional *padding* argument to *ordinal*.rangePoints. Like *ordinal*.rangeBand with *ordinal*.rangePoints, the *point*.bandwidth method always returns zero; a new *point*.step method returns the interval between adjacent points.

The ordinal scale constructor now takes an optional *range* for a shorter alternative to *ordinal*.range. This is especially useful now that the categorical color scales have been changed to simple arrays of colors rather than specialized ordinal scale constructors:

- d3.scale.category10 ↦ d3.schemeCategory10
- d3.scale.category20 ↦ d3.schemeCategory20
- d3.scale.category20b ↦ d3.schemeCategory20b
- d3.scale.category20c ↦ d3.schemeCategory20c

The following code in 3.x:

```
var color = d3.scale.category10();
```

Is equivalent to this in 4.0:

```
var color = d3.scaleOrdinal(d3.schemeCategory10);
```

Sequential scales, are a new class of scales with a fixed output interpolator instead of a range. Typically these scales are used to implement continuous sequential or diverging color schemes. Inspired by Matplotlib's new perceptually-motived colormaps, 4.0 now features viridis, inferno, magma, plasma interpolators for use with sequential scales. Using d3.quantize, these interpolators can also be applied to quantile, quantize and threshold scales.

4.0 also ships new Cubehelix schemes, including Dave Green's default and a cyclical rainbow inspired by Matteo Niccoli:

For even more sequential and categorical color schemes, see d3-scale-chromatic.

For an introduction to scales, see Introducing d3-scale.

Selections (d3-selection)

Selections no longer subclass Array using prototype chain injection; they are now plain objects, improving performance. The internal fields (*selection*._groups, *selection*._parents) are private; please use the documented public API to manipulate selections. The new *selection*.nodes method generates an array of all nodes in a selection.

Selections are now immutable: the elements and parents in a selection never change. (The elements' attributes and content will of course still be modified!) The *selection*.sort and *selection*.data methods now return new selections rather than modifying the selection in-place. In addition, *selection*.append no longer merges entering nodes into the update selection; use *selection*.merge to combine enter and update after a data join. For example, the following general update pattern in 3.x:

```
var circle = svg.selectAll("circle").data(data) // UPDATE
    .style("fill", "blue");

circle.exit().remove(); // EXIT

circle.enter().append("circle") // ENTER; modifies UPDATE!
    .style("fill", "green");

circle // ENTER + UPDATE
    .style("stroke", "black");
```

Would be rewritten in 4.0 as:

```
var circle = svg.selectAll("circle").data(data) // UPDATE
    .style("fill", "blue");

circle.exit().remove(); // EXIT

circle.enter().append("circle") // ENTER
    .style("fill", "green")
  .merge(circle) // ENTER + UPDATE
    .style("stroke", "black");
```

This change is discussed further in What Makes Software Good.

In 3.x, the *selection*.enter and *selection*.exit methods were undefined until you called *selection*.data, resulting in a TypeError if you attempted to access them. In 4.0, now they simply return the empty selection if the selection has not been joined to data.

In 3.x, *selection*.append would always append the new element as the last child of its parent. A little-known trick was to use *selection*.insert without specifying a *before* selector when entering nodes, causing the entering nodes to be inserted before the following element in the update selection. In 4.0, this is now the default behavior of *selection*.append; if you do not specify a *before* selector to *selection*.insert, the inserted element is appended as the last child. This change makes the general update pattern preserve the relative order of elements and data. For example, given the following DOM:

```
<div>a</div>
<div>b</div>
<div>f</div>
```

And the following code:

```
var div = d3.select("body").selectAll("div")
  .data(["a", "b", "c", "d", "e", "f"], function(d) { return d || this.textContent;
});

div.enter().append("div")
    .text(function(d) { return d; });
```

The resulting DOM will be:

```
<div>a</div>
<div>b</div>
<div>c</div>
<div>d</div>
<div>e</div>
<div>f</div>
```

Thus, the entering *c*, *d* and *e* are inserted before *f*, since *f* is the following element in the update selection. Although this behavior is sufficient to preserve order if the new data's order is stable, if the data changes order, you must still use *selection*.order to reorder elements.

There is now only one class of selection. 3.x implemented enter selections using a special class with different behavior for *enter*.append and *enter*.select; a consequence of this design was that enter selections in 3.x lacked certain methods. In 4.0, enter selections are simply normal selections; they have the same methods and the same behavior. Placeholder enter nodes now implement *node*.appendChild, *node*.insertBefore, *node*.querySelector, and *node*.querySelectorAll.

The *selection*.data method has been changed slightly with respect to duplicate keys. In 3.x, if multiple data had the same key, the duplicate data would be ignored and not included in enter, update or exit; in 4.0 the duplicate data is always put in the enter selection. In both 3.x and 4.0, if multiple elements have the same key, the duplicate elements are put in the exit selection. Thus, 4.0's behavior is now symmetric for enter and exit, and the general update pattern will now produce a DOM that matches the data even if there are duplicate keys.

Selections have several new methods! Use *selection*.raise to move the selected elements to the front of their siblings, so that they are drawn on top; use *selection*.lower to move them to the back. Use *selection*.dispatch to dispatch a custom event to event listeners.

When called in getter mode, *selection*.data now returns the data for all elements in the selection, rather than just the data for the first group of elements. The *selection*.call method no longer sets the `this` context when invoking the specified function; the *selection* is passed as the first argument to the function, so use that. The *selection*.on method now accepts multiple whitespace-separated typenames, so you can add or remove multiple listeners simultaneously. For example:

```
selection.on("mousedown touchstart", function() {
  console.log(d3.event.type);
});
```

The arguments passed to callback functions has changed slightly in 4.0 to be more consistent. The standard arguments are the element's datum (*d*), the element's index (*i*), and the element's group (*nodes*), with *this* as the element. The slight exception to this convention is *selection*.data, which is evaluated for each group rather than each element; it is passed the group's parent datum (*d*), the group index (*i*), and the selection's parents (*parents*), with *this* as the group's parent.

The new d3.local provides a mechanism for defining local variables: state that is bound to DOM elements, and available to any descendant element. This can be a convenient alternative to using *selection*.each or storing local state in data.

The d3.ns.prefix namespace prefix map has been renamed to d3.namespaces, and the d3.ns.qualify method has been renamed to d3.namespace. Several new low-level methods are now available, as well. d3.matcher is used internally by *selection*.filter; d3.selector is used by *selection*.select; d3.selectorAll is used by *selection*.selectAll; d3.creator is used by *selection*.append and *selection*.insert. The new d3.window returns the owner window for a given element, window or

document. The new d3.customEvent temporarily sets d3.event while invoking a function, allowing you to implement controls which dispatch custom events; this is used by d3-drag, d3-zoom and d3-brush.

For the sake of parsimony, the multi-value methods—where you pass an object to set multiple attributes, styles or properties simultaneously—have been extracted to d3-selection-multi and are no longer part of the default bundle. The multi-value map methods have also been renamed to plural form to reduce overload: *selection*.attrs, *selection*.styles and *selection*.properties.

Shapes (d3-shape)

Pursuant to the great namespace flattening:

- d3.svg.line ↦ d3.line
- d3.svg.line.radial ↦ d3.radialLine
- d3.svg.area ↦ d3.area
- d3.svg.area.radial ↦ d3.radialArea
- d3.svg.arc ↦ d3.arc
- d3.svg.symbol ↦ d3.symbol
- d3.svg.symbolTypes ↦ d3.symbolTypes
- d3.layout.pie ↦ d3.pie
- d3.layout.stack ↦ d3.stack
- d3.svg.diagonal ↦ REMOVED (see d3/d3-shape#27)
- d3.svg.diagonal.radial ↦ REMOVED

Shapes are no longer limited to SVG; they can now render to Canvas! Shape generators now support an optional *context*: given a CanvasRenderingContext2D, you can render a shape as a canvas path to be filled or stroked. For example, a canvas pie chart might use an arc generator:

```
var arc = d3.arc()
    .outerRadius(radius - 10)
    .innerRadius(0)
    .context(context);
```

To render an arc for a given datum *d*:

```
context.beginPath();
arc(d);
context.fill();
```

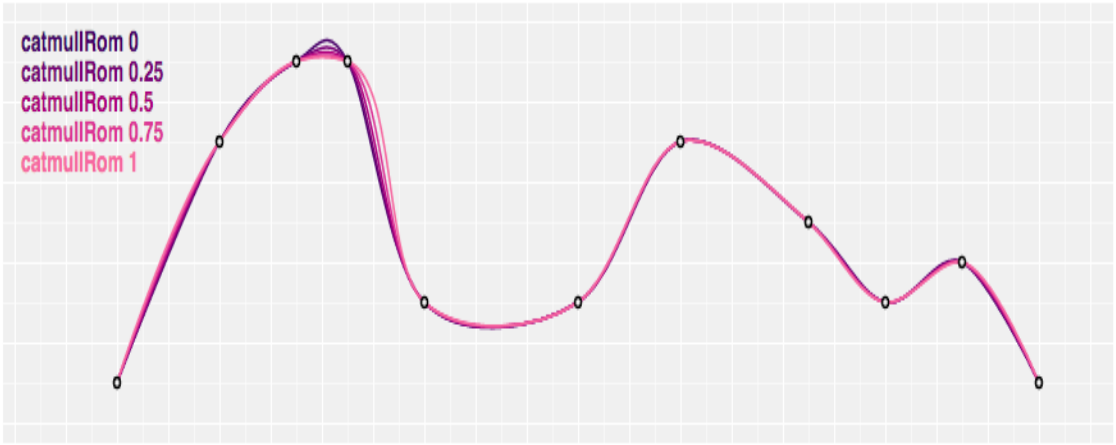
See *line*.context, *area*.context and *arc*.context for more. Under the hood, shapes use d3-path to serialize canvas path methods to SVG path data when the context is null; thus, shapes are optimized for rendering to canvas. You can also now derive lines from areas. The line shares most of the same accessors, such as *line*.defined and *line*.curve, with the area from which it is derived. For example, to render the topline of an area, use *area*.lineY1; for the baseline, use *area*.lineY0.

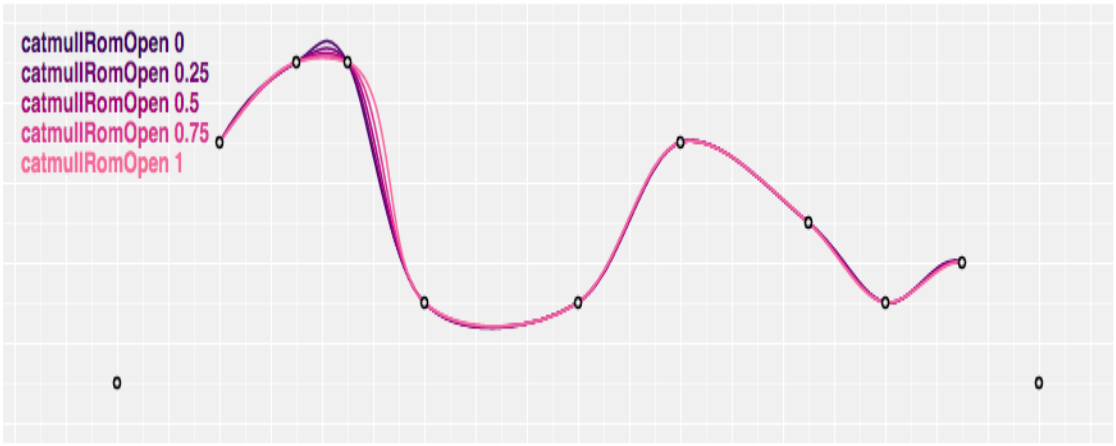
4.0 introduces a new curve API for specifying how line and area shapes interpolate between data points. The *line*.interpolate and *area*.interpolate methods have been replaced with *line*.curve and *area*.curve. Curves are implemented using the curve interface rather than as a function that returns an SVG path data string; this allows curves to render to either SVG or Canvas. In addition, *line*.curve and *area*.curve now take a function which instantiates a curve for a given *context*, rather than a string. The full list of equivalents:

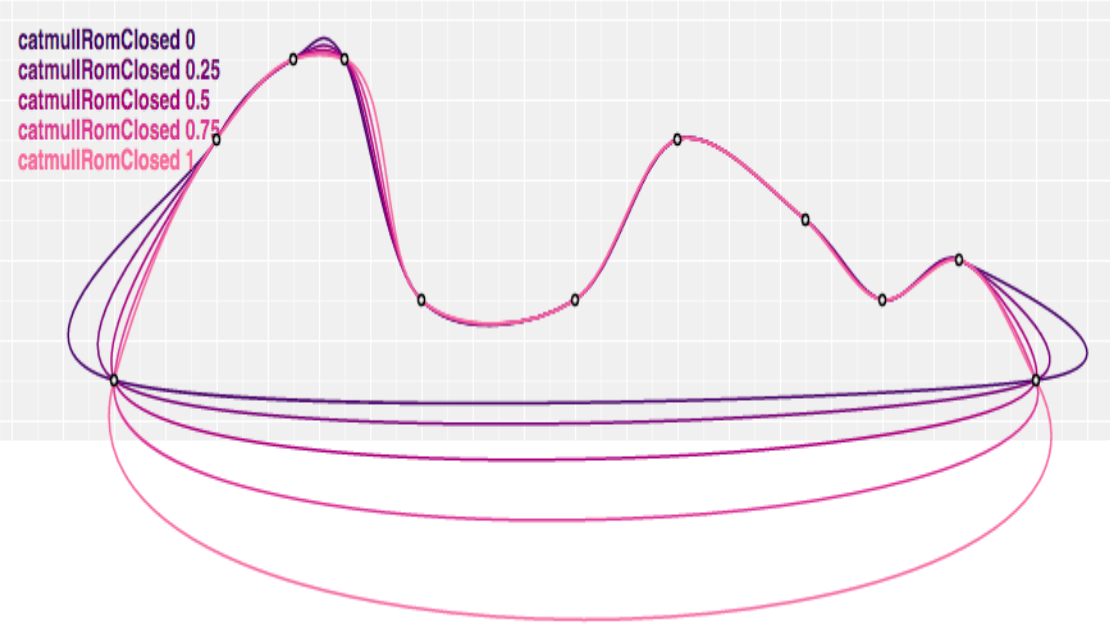
- linear ↦ d3.curveLinear

- linear-closed ↦ d3.curveLinearClosed
- step ↦ d3.curveStep
- step-before ↦ d3.curveStepBefore
- step-after ↦ d3.curveStepAfter
- basis ↦ d3.curveBasis
- basis-open ↦ d3.curveBasisOpen
- basis-closed ↦ d3.curveBasisClosed
- bundle ↦ d3.curveBundle
- cardinal ↦ d3.curveCardinal
- cardinal-open ↦ d3.curveCardinalOpen
- cardinal-closed ↦ d3.curveCardinalClosed
- monotone ↦ d3.curveMonotoneX

But that's not all! 4.0 now provides parameterized Catmull–Rom splines as proposed by Yuksel *et al.*. These are available as d3.curveCatmullRom, d3.curveCatmullRomClosed and d3.curveCatmullRomOpen.







Each curve type can define its own named parameters, replacing *line*.tension and *area*.tension. For example, Catmull–Rom splines are parameterized using *catmullRom*.alpha and defaults to 0.5, which corresponds to a centripetal spline that avoids self-intersections and overshoot. For a uniform Catmull–Rom spline instead:

```
var line = d3.line()
    .curve(d3.curveCatmullRom.alpha(0));
```

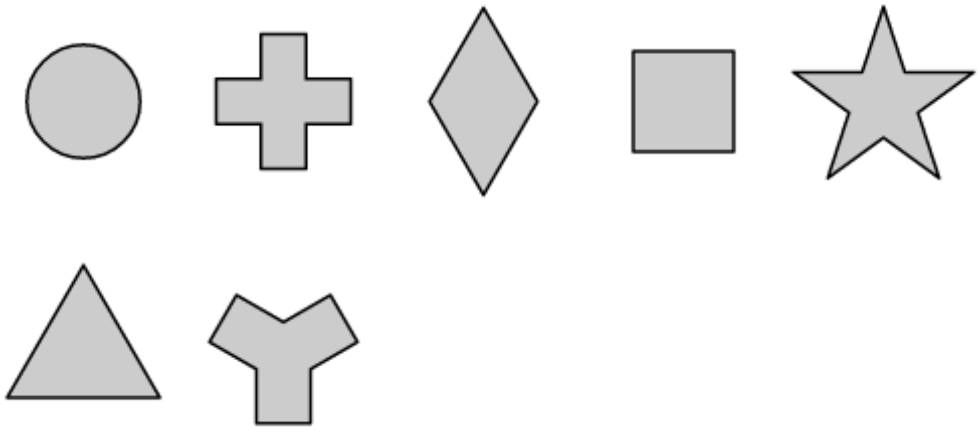
4.0 fixes the interpretation of the cardinal spline *tension* parameter, which is now specified as *cardinal*.tension and defaults to zero for a uniform Catmull–Rom spline; a tension of one produces a linear curve. The first and last segments of basis and cardinal curves have also been fixed! The undocumented *interpolate*.reverse field has been removed. Curves can define different behavior for toplines and baselines by counting the sequence of *curve*.lineStart within *curve*.areaStart. See the d3.curveStep implementation for an example.

4.0 fixes numerous bugs in the monotone curve implementation, and introduces d3.curveMonotoneY; this is like d3.curveMonotoneX, except it requires that the input points are monotone in *y* rather than *x*, such as for a vertically-oriented line chart. The new d3.curveNatural produces a natural cubic spline. The default β for d3.curveBundle is now 0.85, rather than 0.7, matching the values used by Holten. 4.0 also has a more robust implementation of arc padding; see *arc*.padAngle and *arc*.padRadius.

4.0 introduces a new symbol type API. Symbol types are passed to *symbol*.type in place of strings. The equivalents are:

- circle ↦ d3.symbolCircle
- cross ↦ d3.symbolCross
- diamond ↦ d3.symbolDiamond
- square ↦ d3.symbolSquare
- triangle-down ↦ REMOVED
- triangle-up ↦ d3.symbolTriangle
- ADDED ↦ d3.symbolStar
- ADDED ↦ d3.symbolWye

The full set of symbol types is now:



Lastly, 4.0 overhauls the stack layout API, replacing d3.layout.stack with d3.stack. The stack generator no longer needs an *x*-accessor. In addition, the API has been simplified: the *stack* generator now accepts tabular input, such as this array of objects:

```
var data = [
  {month: new Date(2015, 0, 1), apples: 3840, bananas: 1920, cherries: 960, dates:
400},
  {month: new Date(2015, 1, 1), apples: 1600, bananas: 1440, cherries: 960, dates:
400},
  {month: new Date(2015, 2, 1), apples:  640, bananas:  960, cherries: 640, dates:
400},
  {month: new Date(2015, 3, 1), apples:  320, bananas:  480, cherries: 640, dates:
400}
];
```

To generate the stack layout, first define a stack generator, and then apply it to the data:

```
var stack = d3.stack()
    .keys(["apples", "bananas", "cherries", "dates"])
    .order(d3.stackOrderNone)
    .offset(d3.stackOffsetNone);

var series = stack(data);
```

The resulting array has one element per *series*. Each series has one point per month, and each point has a lower and upper value defining the baseline and topline:

```
[
  [[   0, 3840], [   0, 1600], [   0,  640], [   0,  320]], // apples
  [[3840, 5760], [1600, 3040], [ 640, 1600], [ 320,  800]], // bananas
  [[5760, 6720], [3040, 4000], [1600, 2240], [ 800, 1440]], // cherries
  [[6720, 7120], [4000, 4400], [2240, 2640], [1440, 1840]], // dates
]
```

Each series in then typically passed to an area generator to render an area chart, or used to construct rectangles for a bar chart. Stack generators no longer modify the input data, so *stack*.out has been removed.

For an introduction to shapes, see Introducing d3-shape.

Time Formats (d3-time-format)

Pursuant to the great namespace flattening, the format constructors have new names:

- d3.time.format ↦ d3.timeFormat
- d3.time.format.utc ↦ d3.utcFormat
- d3.time.format.iso ↦ d3.isoFormat

The *format*.parse method has also been removed in favor of separate d3.timeParse, d3.utcParse and d3.isoParse parser constructors. Thus, this code in 3.x:

```
var parseTime = d3.time.format("%c").parse;
```

Can be rewritten in 4.0 as:

```
var parseTime = d3.timeParse("%c");
```

The multi-scale time format d3.time.format.multi has been replaced by d3.scaleTime's tick format. Time formats now coerce inputs to dates, and time parsers coerce inputs to strings. The `%Z` directive now allows more flexible parsing of time zone offsets, such as `-0700`, `-07:00`, `-07`, and `Z`. The `%p` directive is now parsed correctly when the locale's period name is longer than two characters (*e.g.*, "a.m.").

The default U.S. English locale now uses 12-hour time and a more concise representation of the date. This aligns with local convention and is consistent with *date*.toLocaleString in Chrome, Firefox and Node:

```
var now = new Date;
d3.timeFormat("%c")(new Date); // "6/23/2016, 2:01:33 PM"
d3.timeFormat("%x")(new Date); // "6/23/2016"
d3.timeFormat("%X")(new Date); // "2:01:38 PM"
```

You can now set the default locale using d3.timeFormatDefaultLocale! The locales are published as JSON to npm.

The performance of time formatting and parsing has been improved, and the UTC formatter and parser have a cleaner implementation (that avoids temporarily overriding the Date global).

Time Intervals (d3-time)

Pursuant to the great namespace flattening, the local time intervals have been renamed:

- ADDED ↦ d3.timeMillisecond
- d3.time.second ↦ d3.timeSecond
- d3.time.minute ↦ d3.timeMinute
- d3.time.hour ↦ d3.timeHour
- d3.time.day ↦ d3.timeDay
- d3.time.sunday ↦ d3.timeSunday
- d3.time.monday ↦ d3.timeMonday

- d3.time.tuesday ↦ d3.timeTuesday
- d3.time.wednesday ↦ d3.timeWednesday
- d3.time.thursday ↦ d3.timeThursday
- d3.time.friday ↦ d3.timeFriday
- d3.time.saturday ↦ d3.timeSaturday
- d3.time.week ↦ d3.timeWeek
- d3.time.month ↦ d3.timeMonth
- d3.time.year ↦ d3.timeYear

The UTC time intervals have likewise been renamed:

- ADDED ↦ d3.utcMillisecond
- d3.time.second.utc ↦ d3.utcSecond
- d3.time.minute.utc ↦ d3.utcMinute
- d3.time.hour.utc ↦ d3.utcHour
- d3.time.day.utc ↦ d3.utcDay
- d3.time.sunday.utc ↦ d3.utcSunday
- d3.time.monday.utc ↦ d3.utcMonday
- d3.time.tuesday.utc ↦ d3.utcTuesday
- d3.time.wednesday.utc ↦ d3.utcWednesday
- d3.time.thursday.utc ↦ d3.utcThursday
- d3.time.friday.utc ↦ d3.utcFriday
- d3.time.saturday.utc ↦ d3.utcSaturday
- d3.time.week.utc ↦ d3.utcWeek
- d3.time.month.utc ↦ d3.utcMonth
- d3.time.year.utc ↦ d3.utcYear

The local time range aliases have been renamed:

- d3.time.seconds ↦ d3.timeSeconds
- d3.time.minutes ↦ d3.timeMinutes
- d3.time.hours ↦ d3.timeHours
- d3.time.days ↦ d3.timeDays
- d3.time.sundays ↦ d3.timeSundays
- d3.time.mondays ↦ d3.timeMondays
- d3.time.tuesdays ↦ d3.timeTuesdays
- d3.time.wednesdays ↦ d3.timeWednesdays
- d3.time.thursdays ↦ d3.timeThursdays
- d3.time.fridays ↦ d3.timeFridays
- d3.time.saturdays ↦ d3.timeSaturdays
- d3.time.weeks ↦ d3.timeWeeks
- d3.time.months ↦ d3.timeMonths
- d3.time.years ↦ d3.timeYears

The UTC time range aliases have been renamed:

- d3.time.seconds.utc ↦ d3.utcSeconds
- d3.time.minutes.utc ↦ d3.utcMinutes
- d3.time.hours.utc ↦ d3.utcHours
- d3.time.days.utc ↦ d3.utcDays
- d3.time.sundays.utc ↦ d3.utcSundays
- d3.time.mondays.utc ↦ d3.utcMondays
- d3.time.tuesdays.utc ↦ d3.utcTuesdays

- d3.time.wednesdays.utc ↦ d3.utcWednesdays
- d3.time.thursdays.utc ↦ d3.utcThursdays
- d3.time.fridays.utc ↦ d3.utcFridays
- d3.time.saturdays.utc ↦ d3.utcSaturdays
- d3.time.weeks.utc ↦ d3.utcWeeks
- d3.time.months.utc ↦ d3.utcMonths
- d3.time.years.utc ↦ d3.utcYears

The behavior of *interval*.range (and the convenience aliases such as d3.timeDays) has been changed when *step* is greater than one. Rather than filtering the returned dates using the field number, *interval*.range now behaves like d3.range: it simply skips, returning every *step*th date. For example, the following code in 3.x returns only odd days of the month:

```
d3.time.days(new Date(2016, 4, 28), new Date(2016, 5, 5), 2);
// [Sun May 29 2016 00:00:00 GMT-0700 (PDT),
//  Tue May 31 2016 00:00:00 GMT-0700 (PDT),
//  Wed Jun 01 2016 00:00:00 GMT-0700 (PDT),
//  Fri Jun 03 2016 00:00:00 GMT-0700 (PDT)]
```

Note the returned array of dates does not start on the *start* date because May 28 is even. Also note that May 31 and June 1 are one day apart, not two! The behavior of d3.timeDays in 4.0 is probably closer to what you expect:

```
d3.timeDays(new Date(2016, 4, 28), new Date(2016, 5, 5), 2);
// [Sat May 28 2016 00:00:00 GMT-0700 (PDT),
//  Mon May 30 2016 00:00:00 GMT-0700 (PDT),
//  Wed Jun 01 2016 00:00:00 GMT-0700 (PDT),
//  Fri Jun 03 2016 00:00:00 GMT-0700 (PDT)]
```

If you want a filtered view of a time interval (say to guarantee that two overlapping ranges are consistent, such as when generating time scale ticks), you can use the new *interval*.every method or its more general cousin *interval*.filter:

```
d3.timeDay.every(2).range(new Date(2016, 4, 28), new Date(2016, 5, 5));
// [Sun May 29 2016 00:00:00 GMT-0700 (PDT),
//  Tue May 31 2016 00:00:00 GMT-0700 (PDT),
//  Wed Jun 01 2016 00:00:00 GMT-0700 (PDT),
//  Fri Jun 03 2016 00:00:00 GMT-0700 (PDT)]
```

Time intervals now expose an *interval*.count method for counting the number of interval boundaries after a *start* date and before or equal to an *end* date. This replaces d3.time.dayOfYear and related methods in 3.x. For example, this code in 3.x:

```
var now = new Date;
d3.time.dayOfYear(now); // 165
```

Can be rewritten in 4.0 as:

```
var now = new Date;
d3.timeDay.count(d3.timeYear(now), now); // 165
```

Likewise, in place of 3.x's d3.time.weekOfYear, in 4.0 you would say:

```
d3.timeWeek.count(d3.timeYear(now), now); // 24
```

The new *interval*.count is of course more general. For example, you can use it to compute hour-of-week for a heatmap:

```
d3.timeHour.count(d3.timeWeek(now), now); // 64
```

Here are all the equivalences from 3.x to 4.0:

- d3.time.dayOfYear ↦ d3.timeDay.count
- d3.time.sundayOfYear ↦ d3.timeSunday.count
- d3.time.mondayOfYear ↦ d3.timeMonday.count
- d3.time.tuesdayOfYear ↦ d3.timeTuesday.count
- d3.time.wednesdayOfYear ↦ d3.timeWednesday.count
- d3.time.thursdayOfYear ↦ d3.timeThursday.count
- d3.time.fridayOfYear ↦ d3.timeFriday.count
- d3.time.saturdayOfYear ↦ d3.timeSaturday.count
- d3.time.weekOfYear ↦ d3.timeWeek.count
- d3.time.dayOfYear.utc ↦ d3.utcDay.count
- d3.time.sundayOfYear.utc ↦ d3.utcSunday.count
- d3.time.mondayOfYear.utc ↦ d3.utcMonday.count
- d3.time.tuesdayOfYear.utc ↦ d3.utcTuesday.count
- d3.time.wednesdayOfYear.utc ↦ d3.utcWednesday.count
- d3.time.thursdayOfYear.utc ↦ d3.utcThursday.count
- d3.time.fridayOfYear.utc ↦ d3.utcFriday.count
- d3.time.saturdayOfYear.utc ↦ d3.utcSaturday.count
- d3.time.weekOfYear.utc ↦ d3.utcWeek.count

D3 4.0 now also lets you define custom time intervals using d3.timeInterval. The d3.timeYear, d3.utcYear, d3.timeMillisecond and d3.utcMillisecond intervals have optimized implementations of *interval*.every, which is necessary to generate time ticks for very large or very small domains efficiently. More generally, the performance of time intervals has been improved, and time intervals now do a better job with respect to daylight savings in various locales.

Timers (d3-timer)

In D3 3.x, the only way to stop a timer was for its callback to return true. For example, this timer stops after one second:

```
d3.timer(function(elapsed) {
  console.log(elapsed);
  return elapsed >= 1000;
});
```

In 4.0, use *timer*.stop instead:

```
var t = d3.timer(function(elapsed) {
  console.log(elapsed);
```

```
  if (elapsed >= 1000) {
    t.stop();
  }
});
```

The primary benefit of *timer*.stop is that timers are not required to self-terminate: they can be stopped externally, allowing for the immediate and synchronous disposal of associated resources, and the separation of concerns. The above is equivalent to:

```
var t = d3.timer(function(elapsed) {
  console.log(elapsed);
});

d3.timeout(function() {
  t.stop();
}, 1000);
```

This improvement extends to d3-transition: now when a transition is interrupted, its resources are immediately freed rather than having to wait for transition to start.

4.0 also introduces a new *timer*.restart method for restarting timers, for replacing the callback of a running timer, or for changing its delay or reference time. Unlike *timer*.stop followed by d3.timer, *timer*.restart maintains the invocation priority of an existing timer: it guarantees that the order of invocation of active timers remains the same. The d3.timer.flush method has been renamed to d3.timerFlush.

Some usage patterns in D3 3.x could cause the browser to hang when a background page returned to the foreground. For example, the following code schedules a transition every second:

```
setInterval(function() {
  d3.selectAll("div").transition().call(someAnimation); // BAD
}, 1000);
```

If such code runs in the background for hours, thousands of queued transitions will try to run simultaneously when the page is foregrounded. D3 4.0 avoids this hang by freezing time in the background: when a page is in the background, time does not advance, and so no queue of timers accumulates to run when the page returns to the foreground. Use d3.timer instead of transitions to schedule a long-running animation, or use d3.timeout and d3.interval in place of setTimeout and setInterval to prevent transitions from being queued in the background:

```
d3.interval(function() {
  d3.selectAll("div").transition().call(someAnimation); // GOOD
}, 1000);
```

By freezing time in the background, timers are effectively “unaware” of being backgrounded. It’s like nothing happened! 4.0 also now uses high-precision time (performance.now) where available; the current time is available as d3.now.

Transitions (d3-transition)

The *selection*.transition method now takes an optional *transition* instance which can be used to synchronize a new transition with an existing transition. (This change is discussed further in What Makes Software Good?) For example:

```
var t = d3.transition()
    .duration(750)
    .ease(d3.easeLinear);

d3.selectAll(".apple").transition(t)
    .style("fill", "red");

d3.selectAll(".orange").transition(t)
    .style("fill", "orange");
```

Transitions created this way inherit timing from the closest ancestor element, and thus are synchronized even when the referenced *transition* has variable timing such as a staggered delay. This method replaces the deeply magical behavior of *transition*.each in 3.x; in 4.0, *transition*.each is identical to *selection*.each. Use the new *transition*.on method to listen to transition events.

The meaning of *transition*.delay has changed for chained transitions created by *transition*.transition. The specified delay is now relative to the *previous* transition in the chain, rather than the *first* transition in the chain; this makes it easier to insert interstitial pauses. For example:

```
d3.selectAll(".apple")
  .transition() // First fade to green.
    .style("fill", "green")
  .transition() // Then red.
    .style("fill", "red")
  .transition() // Wait one second. Then brown, and remove.
    .delay(1000)
    .style("fill", "brown")
    .remove();
```

Time is now frozen in the background; see d3-timer for more information. While it was previously the case that transitions did not run in the background, now they pick up where they left off when the page returns to the foreground. This avoids page hangs by not scheduling an unbounded number of transitions in the background. If you want to schedule an infinitely-repeating transition, use transition events, or use d3.timeout and d3.interval in place of setTimeout and setInterval.

The *selection*.interrupt method now cancels all scheduled transitions on the selected elements, in addition to interrupting any active transition. When transitions are interrupted, any resources associated with the transition are now released immediately, rather than waiting until the transition starts, improving performance. (See also *timer*.stop.) The new d3.interrupt method is an alternative to *selection*.interrupt for quickly interrupting a single node.

The new d3.active method allows you to select the currently-active transition on a given *node*, if any. This is useful for modifying in-progress transitions and for scheduling infinitely-repeating transitions. For example, this transition continuously oscillates between red and blue:

```
d3.select("circle")
  .transition()
    .on("start", function repeat() {
        d3.active(this)
            .style("fill", "red")
          .transition()
            .style("fill", "blue")
```

```
      .transition()
        .on("start", repeat);
  });
```

The life cycle of a transition is now more formally defined and enforced. For example, attempting to change the duration of a running transition now throws an error rather than silently failing. The *transition*.remove method has been fixed if multiple transition names are in use: the element is only removed if it has no scheduled transitions, regardless of name. The *transition*.ease method now always takes an easing function, not a string. When a transition ends, the tweens are invoked one last time with *t* equal to exactly 1, regardless of the associated easing function.

As with selections in 4.0, all transition callback functions now receive the standard arguments: the element's datum (*d*), the element's index (*i*), and the element's group (*nodes*), with *this* as the element. This notably affects *transition*.attrTween and *transition*.styleTween, which no longer pass the *tween* function the current attribute or style value as the third argument. The *transition*.attrTween and *transition*.styleTween methods can now be called in getter modes for debugging or to share tween definitions between transitions.

Homogenous transitions are now optimized! If all elements in a transition share the same tween, interpolator, or event listeners, this state is now shared across the transition rather than separately allocated for each element. 4.0 also uses an optimized default interpolator in place of d3.interpolate for *transition*.attr and *transition*.style. And transitions can now interpolate both CSS and SVG transforms.

For reusable components that support transitions, such as axes, a new *transition*.selection method returns the selection that corresponds to a given transition. There is also a new *transition*.merge method that is equivalent to *selection*.merge.

For the sake of parsimony, the multi-value map methods have been extracted to d3-selection-multi and are no longer part of the default bundle. The multi-value map methods have also been renamed to plural form to reduce overload: *transition*.attrs and *transition*.styles.

Voronoi Diagrams (d3-voronoi)

The d3.geom.voronoi method has been renamed to d3.voronoi, and the *voronoi*.clipExtent method has been renamed to *voronoi*.extent. The undocumented *polygon*.point property in 3.x, which is the element in the input *data* corresponding to the polygon, has been renamed to *polygon*.data.

Calling *voronoi* now returns the full Voronoi diagram, which includes topological information: each Voronoi edge exposes *edge*.left and *edge*.right specifying the sites on either side of the edge, and each Voronoi cell is defined as an array of these edges and a corresponding site. The Voronoi diagram can be used to efficiently compute both the Voronoi and Delaunay tessellations for a set of points: *diagram*.polygons, *diagram*.links, and *diagram*.triangles. The new topology is also useful in conjunction with TopoJSON; see the Voronoi topology example.

The *voronoi*.polygons and *diagram*.polygons now require an extent; there is no longer an implicit extent of ±1e6. The *voronoi*.links, *voronoi*.triangles, *diagram*.links and *diagram*.triangles are now affected by the clip extent: as the Delaunay is computed as the dual of the Voronoi, two sites are only linked if the clipped cells are touching. To compute the Delaunay triangulation without respect to clipping, set the extent to null.

The Voronoi generator finally has well-defined behavior for coincident vertices: the first of a set of coincident points has a defined cell, while the subsequent duplicate points have null cells. The returned array of polygons is sparse, so by using *array*.forEach or *array*.map, you can easily skip undefined cells. The Voronoi generator also now correctly handles the case where no cell edges intersect the extent.

Zooming (d3-zoom)

The zoom behavior d3.behavior.zoom has been renamed to d3.zoom. Zoom behaviors no longer store the active zoom transform (*i.e.*, the visible region; the scale and translate) internally. The zoom transform is now stored on any elements to which the zoom behavior has been applied. The zoom transform is available as *event*.transform within a zoom event or by calling d3.zoomTransform on a given *element*. To zoom programmatically, use *zoom*.transform with a given selection or transition; see the zoom transitions example. The *zoom*.event method has been removed.

To make programmatic zooming easier, there are several new convenience methods on top of *zoom*.transform: *zoom*.translateBy, *zoom*.scaleBy and *zoom*.scaleTo. There is also a new API for describing zoom transforms. Zoom behaviors are no longer dependent on scales, but you can use *transform*.rescaleX, *transform*.rescaleY, *transform*.invertX or *transform*.invertY to transform a scale's domain. 3.x's *event*.scale is replaced with *event*.transform.k, and *event*.translate is replaced with *event*.transform.x and *event*.transform.y. The *zoom*.center method has been removed in favor of programmatic zooming.

The zoom behavior finally supports simple constraints on panning! The new *zoom*.translateExtent lets you define the viewable extent of the world: the currently-visible extent (the extent of the viewport, as defined by *zoom*.extent) is always contained within the translate extent. The *zoom*.size method has been replaced by *zoom*.extent, and the default behavior is now smarter: it defaults to the extent of the zoom behavior's owner element, rather than being hardcoded to 960×500. (This also improves the default path chosen during smooth zoom transitions!)

The zoom behavior's interaction has also improved. It now correctly handles concurrent wheeling and dragging, as well as concurrent touching and mousing. The zoom behavior now ignores wheel events at the limits of its scale extent, allowing you to scroll past a zoomable area. The *zoomstart* and *zoomend* events have been renamed *start* and *end*. By default, zoom behaviors now ignore right-clicks intended for the context menu; use *zoom*.filter to control which events are ignored. The zoom behavior also ignores emulated mouse events on iOS. The zoom behavior now consumes handled events, making it easier to combine with other interactive behaviors such as dragging.