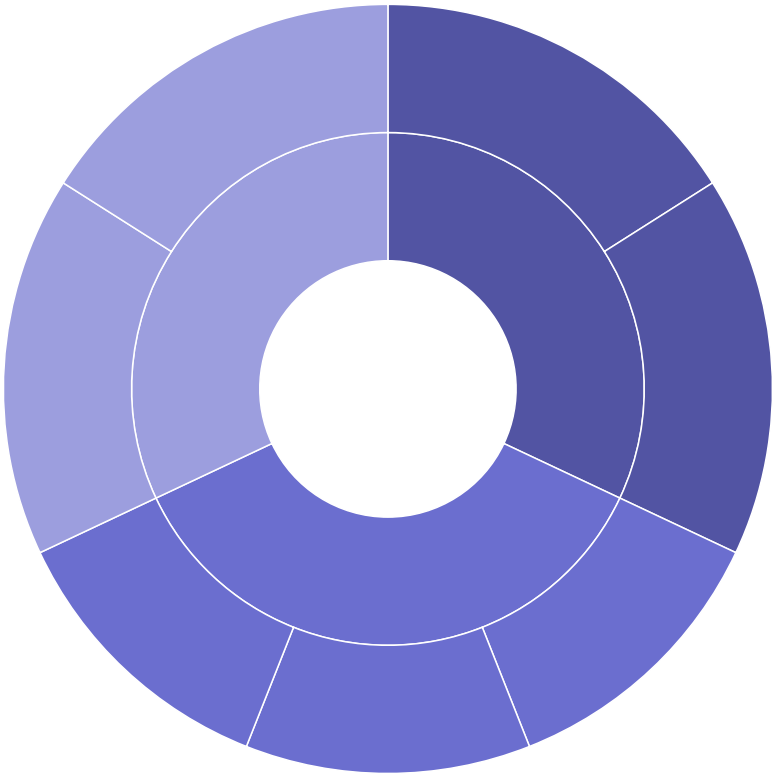
David Richards's Block e1cdbbe586ac31747b4a304f8f86efa5
Updated July 7, 2017

Sunburst Tutorial (d3 v4), Part 1



Open

A No Frills Sunburst

In this page, we'll do a detailed walk-through of a basic "no frills" d3 Sunburst. We'll add features in future tutorials. I strive to explain each line. If I don't explain it, or explain it well, I welcome your input. For each titled section, we'll begin with the code and then explain it. Maybe it'll help you solve a problem in your own code or build something that you're proud of.

On the bl.ocks.org page, scroll to the bottom to see the uninterrupted code of a Sunburst visual, based on d3 version 4. Part 2 builds on this tutorial, but adds labels and pulls the data from a separate json file.

Sunbursts are great for explaining relationships in hierarchical data. But the code can get confusing as we mix html, css, svg, json, javascript, and d3. And, bounce between radians and degrees.

Do good! —David Richards

The Web Page

Create a bare-bones web page that references the d3 framework and holds our sunburst viz.

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

```
    <script>
    <!— d3 logic goodness here —>
    </script>
</body>
```

This very basic web page has includes 2 <script> sections 1. In the <head>: points the browser to our d3 library. 2. In the <body>: will hold all of the code shared below.

The <body> section also contains a <svg> element. This is where our d3 visualization will actually get drawn.

The Data

Create some data to be presented in our sunburst.

```
var nodeData = {
    "name": "TOPICS", "children": [{
        "name": "Topic A",
        "children": [{"name": "Sub A1", "size": 4}, {"name": "Sub A2", "siz
    }, {
        "name": "Topic B",
        "children": [{"name": "Sub B1", "size": 3}, {"name": "Sub B2", "siz
            "name": "Sub B3", "size": 3}]
    }, {
        "name": "Topic C",
        "children": [{"name": "Sub A1", "size": 4}, {"name": "Sub A2", "siz
    }]
};
```

JSON for a sunburst is structured as a hierarchy. This JSON contains data about 11 **nodes**. (We'll call these **arcs** when we calculate each node's size in d3 code. And we sometimes call them **slices** when we're looking at our visualization.). The very first node is called the **root** node (in our code above: `"name": "TOPICS"`). The root node is a sort of anchor for our data and visualization, and we often treat it differently since it's the center of or sunburst. We define each node in the above data in 1 of 2 ways:

1. `{ "name": "abc", "children": [] }` describes a node that has children. Size isn't defined for these nodes, because it'll be adopted (calculated by d3) based on children nodes. Children will either be more nodes like this one, with children of their own, or nodes with a "size" when it has no children.

2. `{ "name": "zyz", "size": 4 }` describes an end node with no children. The hierarchy does not need to be symmetrical in any way if. Nodes can have differing numbers of children, or have "sibling" nodes that have no children at all).

Initialize Variables

We'll set variables for common values here at the top.

```
var width = 500;  // <-- 1
var height = 500;
var radius = Math.min(width, height) / 2;  // < -- 2
var color = d3.scaleOrdinal(d3.schemeCategory20b);   // <-- 3
```

We'll set 4 variables that we can use throughout our code: 1. `var width = 500` creates a variable set to 500; it does not actually set the width of anything, yet. Below we'll apply this variable to the <svg> element's width attribute. We could set width directly (`<svg width=500>`). But we'll use this values a few times. If we coded it directly, we'd then need to change each occurrences every time. Mistakes will happen.

1. `var radius = Math.min(width, height) / 2` determines which is smaller (the `min`), the width or height. Then it divides that value by 2 (since the radius is 1/2 of the circle's diameter). Then we store that value as our radius. This optimizes the size of our viz within the <svg> element (since we don't want to leak past the edges, but we also don't want a bunch of wasted white space). Since width and height are both 500, the radius variable will equal 250.

2. `d3.scaleOrdinal`: d3 scales help us map our data to something in our visual. Outside of d3, *ordinal scales* indicate the direction of the underlying data and provide nominal information (e.g., low, medium, high). In the same way, scaleOrdinal in d3 allows us to relate a part of our data to something that has a series of named values (like an array of colors).

    - `schemeCategory20b` is a d3 command that returns an array of colors. d3 has several similar options that are specifically designed to work with `d3.scaleOrdinal()`. The result of this line is that we'll have a variable ("color") that will return a rainbow of options for our sunburst.

Setting up our SVG workspace

Begin by getting a handle on our SVG and beginning to get things set up.

```
var g = d3.select('svg')  // <-- 1
    .attr('width', width)  // <-- 2
    .attr('height', height)
    .append('g')  // <-- 3
    .attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')');
```

1. `d3.select('svg')` selects our <svg></svg> element so that we can work with it. The `d3.select()` command finds the first element (and only the first, if there are multiple) that matches the specified string. If the select command does not find a match, it returns an empty selection.

2. `.attr('width', width)` sets the width attribute of our <svg> element.

3. `.append('g')` adds a <g> element to our SVG. <g> does not do much directly, it's is a special SVG element that acts as a container; it groups other SVG elements. And transformations applied to the <g> element are performed on all of its child elements. And its attributes are inherited by its children. That'll be helpful later.

4. `.attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')')` sets the value for the `transform` attribute (as we did with `width` above). SVG's transform attribute allows us to scale, translate (move), and rotate our <g> element (and it's children). There's a longer conversation to be had about the SVG coordinate system (See Sara Soueidan's article helps clarify the mechanics.). For now, we'll simply say that we'll use this transform attribute to move the "center" [0,0] of our <g> element from the upper-left to the actual center of our <svg> element:

    - `'translate(' + width / 2 + ',' + height / 2 + ')'` will resolves to `translate(250, 250)`. This command moves our coordinate system (for <g>) 250

units right (x-axis) and 250 units down (y-axis).

Method Chaining & the HTML

Method chaining allows us to connect multiple commands together with periods between into a single statement, like we've done above. It's important to recognize that each d3 method returns something, and the next method in the chain applies to that something. Here's the above code, with a note about what each line returns:

```
var g = d3.select('svg')  // returns a handle to the <svg> element
    .attr('width', width)  // sets the width of <svg> and then returns the
    .attr('height', height)  // (same as width)
    .append('g')  // adds a <g> element to the <svg> element. It returns th
    .attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')');
```

Method chaining is key to understanding what's going on in most all d3 code. To fully "get" the meaning of a code block, we must understand both what the method does and what it returns. (Want more? See Scott Murray's Chaining methods article.)

Another way to think about the progression of our d3 is to see our html elements grow through each step. Thinking about the same 1-5 steps above, we'd see the following happen:

```
var g = d3.select('svg')  // --> <svg></svg>
    .attr('width', width)  // --> <svg width="500"></svg>
    .attr('height', height)  // --> <svg width="500" height="500"></svg>
    .append('g')  // --> <svg width="500" height="500"><g></g></svg>
    .attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')');
```

Formatting the Data

Like we prepped our SVG above, we'll create a space for our data to go that's specifically formatted for hierarchical visualizations.

```
var partition = d3.partition()  // <-- 1
    .size([2 * Math.PI, radius]);  // <-- 2
```

1. The `partition` command is a special tool that will help organize our data into the sunburst pattern, and ensure things are properly sized (e.g., that we use all 360 degrees of the circle, and that each slice is sized relative to the other slices.) So far, this is about structure, since we haven't linked it to our actual data yet.

2. `size` sets this partition's overall size "width" and "height". But we've shifted from an [x,y] coordinate system (where a box could be 25 by 25] to a system where we size each part of our sunburst in radians (how much of the 360 the shape will consume) and depth (distance from center to full radius):

    1. `2 * Math.PI` tells d3 the number of **radians** our sunburst will consume. Remember from middle-school geometry that a circle has a circumference of 2πr (2 * pi * r). This coordinate tells d3 how big our sunburst is in "radiuses". The answer is that it's 2π radiuses (or *radians*). So it's a full circle.

        - Want a sunburst that's a ½ circle? Delete the `2 *`.
        - Want to better understand radians and how they map to degrees? Try mathisfun:

radians or Intuitive Guide to Angles, Degrees and Radians.

2. `radius` takes our variable, set above, and tells d3 that this is the distance from the center to the outside of the sunburst.

Find the Root Node

The first thing we need to do with our data is to find the **root** node--that's the very first node in our hierarchical data.

```
var root = d3.hierarchy(nodeData)  // <-- 1
    .sum(function (d) { return d.size});  // <-- 2
```

1. The sunburst layout (or any hierarchical layout in d3) needs a **root** node. Happily, our data is already in a hierarchical pattern and has a root node ("name" : "TOPICS"). So we can pass our data directly to d3.partition with no preliminary reformatting. We use `d3.hierarchy(nodeData)` to tell d3, "Hey d3, our data is stored in the nodeData variable. It's already shaped in a hierarchy."

2. `sum()` iterates through each node in our data and adds a "value" attribute to each one. The value stored in the "value" attribute is the combined sum of d.size (since that's what we've asked for) for this node and all of its children nodes. It will be used later to determine arc / slice sizes.

    - Example: If the current node has no size attribute of its own, but it has 2 children, each size = 4, then .sum() will create a "value = 8" attribute for this node.
    - See the d3 documentation for node.sum(value)

d3 has a specific pattern for retrieving your data and applying it to d3 commands, a pattern that you'll see repeatedly: `function(d) { return d }`. This functions accepts "d", which represents your data, and returns a value, or an array of values, based on your data. The "return d" part can get intricate. In our code, we're returning the size (d.size) to the sum function. We defined "size" in our JSON, so it's often available for a node. When size isn't defined, this function returns 0. Two specific examples from our data: * "Sub A1" will have a has a size of 4. * "Topic A" has a size of 8, the sum of "Sub A1" and "Sub A2".

Calculate each arc

Now that we've got our data, we should calculate the size of each node for our sunburst.

```
partition(root);  // <-- 1
var arc = d3.arc()  // <-- 2
    .startAngle(function (d) { return d.x0 })
    .endAngle(function (d) { return d.x1 })
    .innerRadius(function (d) { return d.y0 })
    .outerRadius(function (d) { return d.y1 });
```

1. `partition(root)` combines our partition variable (which creates the data structure) with our root node (the actual data). This line sets us up for the arc statement. Advanced Note: Inspecting "d" in our functions (e.g., `function (d) { return d.x0 }`) before and after this partition line yields an interesting finding:

    - Before this line, "d" for a particular node returns an object that looks just like our

underlying json: `{name: "Sub A1", size: 4}`.
- After this partition line, "d" for a particular node returns a d3-shaped object: `{data: Object, height: 0, depth: 2, parent: qo, value: 4…}`. And our json attributes are tucked into the *data* attribute.

2. `d3.arc()` calculates the size of each arc based on our JSON data. Each of the 4 variables below are staples in d3 sunbursts. They define the 4 outside lines for each arc.

    - d.x0 is the radian location for the start of the arc, as we traverse around the circle.
    - d.x1 is the radian location for the end of the arc. If x0 and x1 are the same, our arc will be invisible. If x0 = 0 and x1 = 2, our arc will encompass a bit less than 1/3 of our circle.
    - d.y0 is the radian location for the inside arc.
    - d.y1 is the radian location for the outside arc. If y0 and y1 are the same, our arc will be invisible.

Putting it all together

We've got a palette (the SVG), data, and some calculated arcs. Let's put it all together and give it some color.

```
g.selectAll('path')  // <-- 1
    .data(root.descendants())  // <-- 2
    .enter()  // <-- 3
    .append('path')  // <-- 4
    .attr("display", function (d) { return d.depth ? null : "none"; })  //
    .attr("d", arc)  // <-- 6
    .style('stroke', '#fff')  // <-- 7
    .style("fill", function (d) { return color((d.children ? d : d.parent).
```

This final block of code takes everything we've built so far and writes it to our `<svg><g></g></svg>` element, using a series of `<path>` elements.

d3's "update pattern" operates as following: 1. `g.selectAll('path')` starts with the g variable that we created way above; it references the <g> element that we originally appended to our `<svg>` element. `selectAll` gets a reference to all existing `<path>` elements within our <g> element. "That's odd," you say, "since we know that there are no `<path>` elements in <g>." You are right. They don't yet exist. For now, we'll just say that d3 uses this step to establish where the new `<path>` elements will fit on the page (in the svg object model).

1. `.data(root.descendants())` tells d3 what about the `<path>` elements that we want to exist by passing it our data. We're passing in our root variable with its descendants.

2. `.enter()` tells d3 to "connect" the originally selected `<path>` element with our data so that we can...

3. `.append('path')` actually creates one new, but empty, `<path>` element for each node under our <g> element. See Chris Givens' Update Pattern tutorial for another look at steps 1-4 above.

4. `.attr("display", function (d) { return d.depth ? null : "none"; })` sets the display attribute of the `<path>` element for our *root* node to "none". (`display="none"` tells SVG that this element will not be rendered.)

    - d.depth will equal 0 for the root node, 1 for its children, 2 for "grandchildren", etc.
    - Want more layers in your sunburst? This visualization will add as many layers as you have

in your data. We've limited to just 2 layers for simplicity in our explanation. (Advanced Idea: Imagine you've added many additional layers in the json, and maybe you don't always want to show all of those layers. You could use a d.depth test to limit the number of rings that actually appear on your viz.)

5. `.attr("d", arc)` fills in all the "d" attributes of each <path> element with the values from our arc variable. Two important notes here:
    - The d attribute contains the actual directions for each line of this svg <path> element, see the example below.
    - Don't confuse the the <path d=""> attribute with the d variable that represents the data within or d3 script.

If you stopped here, you'd see a sunburst with each slice drawn, but all black with barely visible gray lines separating the slices. Let's add some color:

1. `.style('stroke', '#fff')` add `style="stroke: rgb(255, 255, 255);"` to our <path> element. Now the lines between our slices are white.
2. `.style("fill", function (d) { return color((d.children ? d : d.parent).data.name); })` combines the `color` variable we defined at the beginning (which returns an array of colors that we can step through) with our data.
    - `(d.children ? d : d.parent)` is a javascript inline if in the form of (condition ? expr1 : expr2) that says, if the current node has children, return the current node, otherwise, return its parent.
    - That node's name will be passed to our color variable and then returned to the style attribute within each <path> element.

In the end, this section of our HTML will look something like this (ellipsis indicates details that I haven't included):

```
<g transform="translate(250,250)">
    <path display="none" d=". . ." style="stroke: rgb(255, 255, 255); fill:
    <path style="stroke: rgb(255, 255, 255); fill: rgb(82, 84, 163);" d="M1
    150.80450874433657,70.96321526084546L75.40225437216829,35.4816076304227
    0,0,0,5.102694996447305e-15,-83.33333333333333Z">
    </path></path> . . . <path></path>
</g>
```

Voilà! Great job on creating your first, well-understood hierarchical visualization in d3. We've just scratched the surface. If you're ready, join me for Tutorial 2.

index.html

```
<!DOCTYPE html>
<head>
    <script src="https://d3js.org/d3.v4.min.js"></script>
</head>
<body>
    <svg></svg>
</body>

<script>
    // JSON data
    var nodeData = {
```

```
        "name": "TOPICS", "children": [{
            "name": "Topic A",
            "children": [{"name": "Sub A1", "size": 4}, {"name": "Sub A2", "size": 4}]
        }, {
            "name": "Topic B",
            "children": [{"name": "Sub B1", "size": 3}, {"name": "Sub B2", "size": 3}, {
                "name": "Sub B3", "size": 3}]
        }, {
            "name": "Topic C",
            "children": [{"name": "Sub A1", "size": 4}, {"name": "Sub A2", "size": 4}]
        }]
    };

    // Variables
    var width = 500;
    var height = 500;
    var radius = Math.min(width, height) / 2;
    var color = d3.scaleOrdinal(d3.schemeCategory20b);

    // Create primary <g> element
    var g = d3.select('svg')
        .attr('width', width)
        .attr('height', height)
        .append('g')
        .attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')');

    // Data strucure
    var partition = d3.partition()
        .size([2 * Math.PI, radius]);

    // Find data root
    var root = d3.hierarchy(nodeData)
        .sum(function (d) { return d.size});

    // Size arcs
    partition(root);
    var arc = d3.arc()
        .startAngle(function (d) { return d.x0 })
        .endAngle(function (d) { return d.x1 })
        .innerRadius(function (d) { return d.y0 })
        .outerRadius(function (d) { return d.y1 });

    // Put it all together
    g.selectAll('path')
        .data(root.descendants())
        .enter().append('path')
        .attr("display", function (d) { return d.depth ? null : "none"; })
        .attr("d", arc)
        .style('stroke', '#fff')
        .style("fill", function (d) { return color((d.children ? d : d.parent).data.name); });
</script>
```

LICENSE

Released under the GNU General Public License, version 3.