D3: EASY AND FLEXIBLE DATA VISUALISATION USING WEB STANDARDS

@JOSDIRKSEN

More information contact me: @josdirksen / jos.dirksen@gmail.com / www.smartjava.org / github.com/josdirksen


WHAT IS D3.JS

WHAT IS D3?

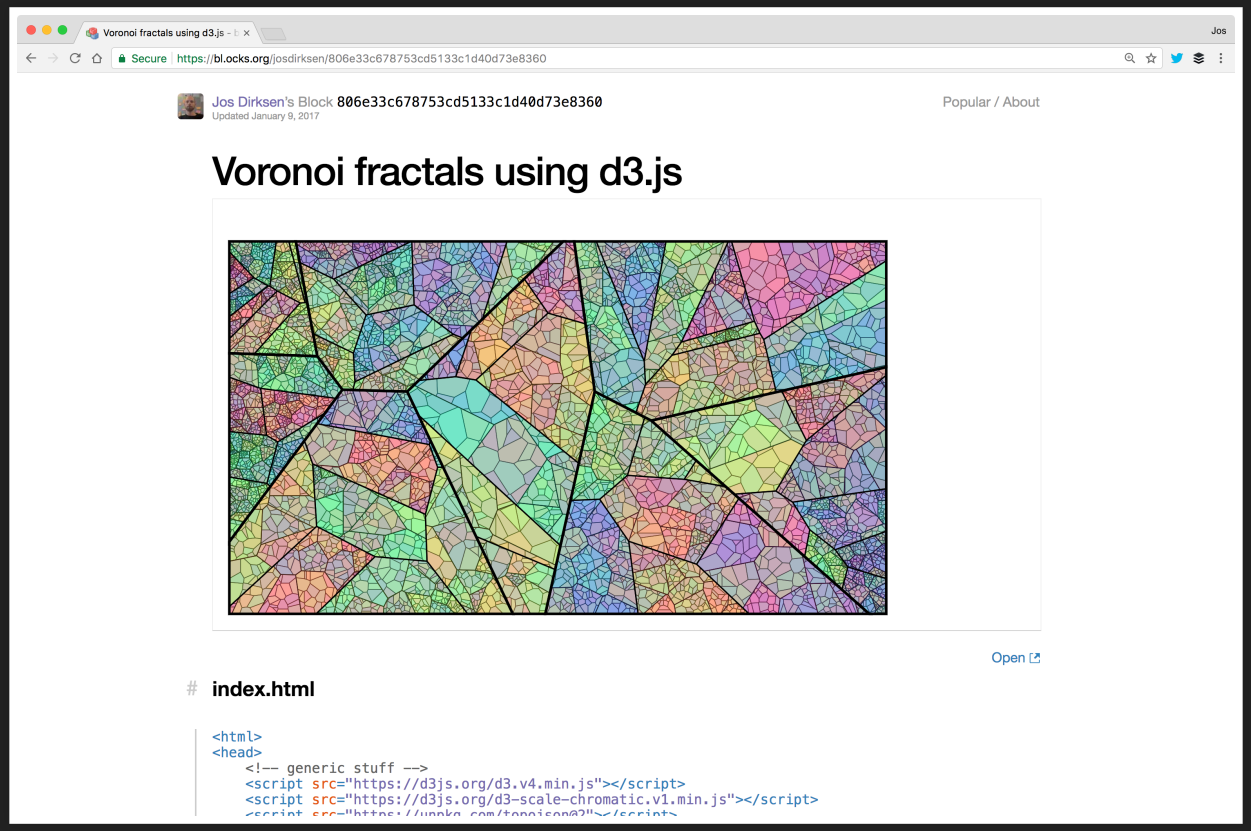
*D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, **SVG**, and **CSS**. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a **proprietary** framework, combining powerful visualization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

MORE INFO ON D3.JS

- Main site: https://d3js.org/
- API: https://github.com/d3/d3/blob/master/API.md
- Examples: https://bl.ocks.org/

Usually this will be all you need

BL.OCKS.ORG IS GREAT



ALSO LARGE NUMBER OF BOOKS

- Be sure to check version: D3 v4 has new API

or one of the many others

WHAT IT IS NOT

- **Not a charting library**: provides building blocks for other libraries (e.g nvd3.js, Rickshaw.js and c3.js)
- **Not an SVG abstraction library**: Manipulates the Dom and SVG directly.
- **Not a Canvas/WebGL library**: Minimal support for Canvas, for 3D use Three.js.
- **Not a framework**: Can be used with Angular, React, Vue.js and anything else.

WHAT ARE WE GOING TO COVER

- Main patterns:
  - Data binding: **select** / **add** / **update** / **remove**
  - Path generators: such as **arcs** and **path**
  - Working with **geo** to visualize map based data
  - Add **interactivity**
  - Working with layouts: **force**, **tree** and others
- Mentions of some other stuff:
  - loading data, working with **Scales**
  - Generative art, transitions

FOLLOW ALONG

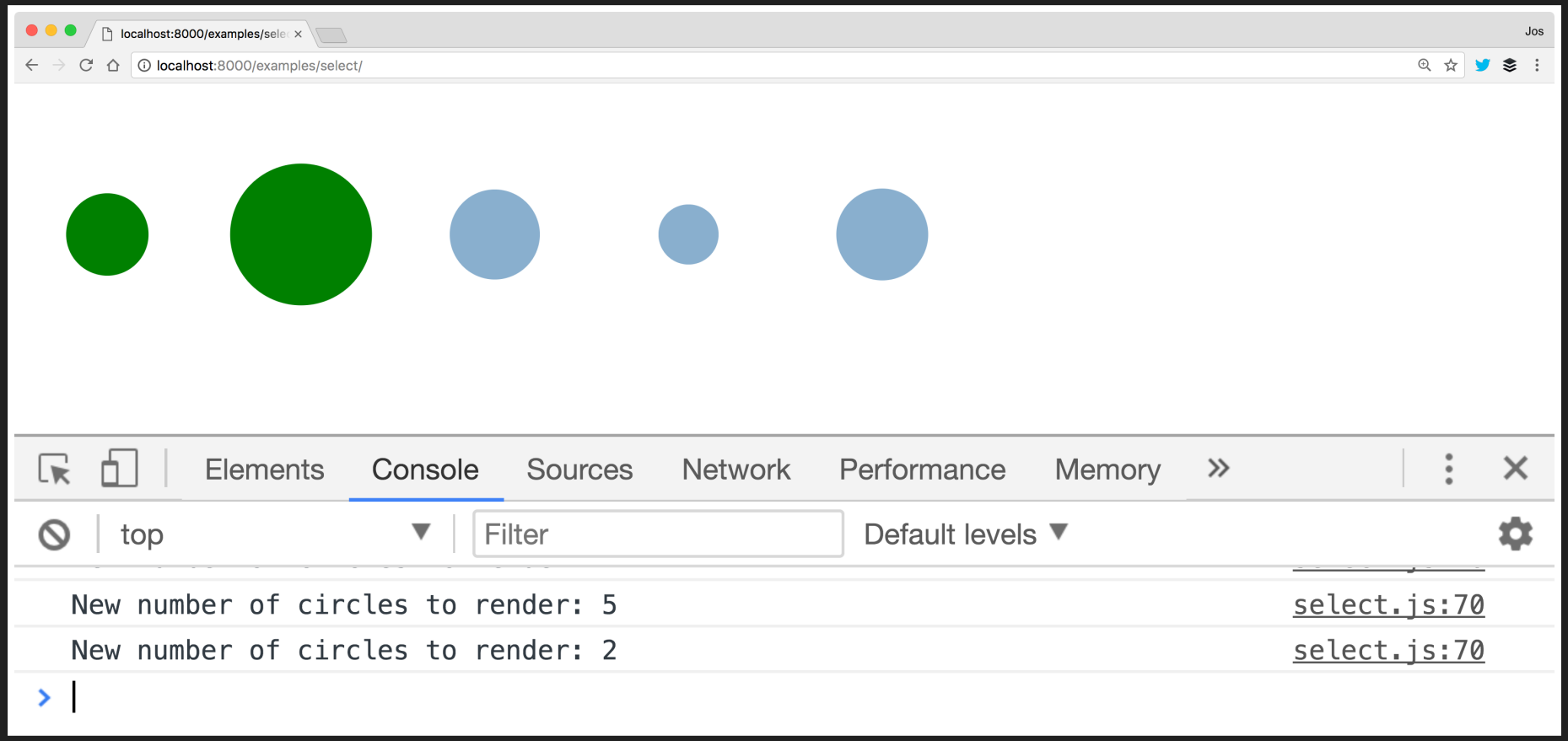
- http://tiny.cc/d3devoxx
- Or last tweet from twitter: **@josdirksen**


PATTERN 1:
SELECT/ADD/UPDATE/REMOVE

SELECT / ADD / UPDATE / REMOVE

Select elements from the DOM, and bind data to them. If we have too little elements, add new ones. Update the datum bound to the existing ones, and remove elements, for which we don't have data ony more.

PATTERN IN ACTION



Select / add / update / remove

SIDESTEP: HTML SETUP

```
<html>
<head>
    <script src="../../libs/d3.v4.js"></script>
    <script src="./js/select.js"></script>
    <link rel="stylesheet" href="./css/style.css" type="text/c
</head>
<body>
<div id="output"><svg class="chart"></svg></div>
<script>
    (function() { show(); })();
</script>
</body>
</html>
```

SIDESTEP: JAVASCRIPT SETUP

```
function show() {

    var margin = { top: 20, bottom: 20, right: 40, left: 40 },
        width = 800 - margin.left - margin.right,
        height = 400 - margin.top - margin.bottom;

    var chart = d3.select(".chart")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

    function update() { ... }

    update();
    d3.interval(function() { update(); }, 5000);
}
```

We'll fill in the update function

STEP 0: THE DATA WE'RE WORKING WITH

- Data is a just an array: `[ ]`
- We'll use sample data:
  - `1` to `10` elements
  - each element random value `20` to `40`
- Handled in this example by `getRandomData()`

```
val data = [26.23690455102537, 25.132792981537815,
            34.363197598539266]
```

STEP 1: SELECT AND BIND DATA

- Select and bind data:

```
var circles = chart.selectAll('circle')
  .data(getRandomData())
```

- Bind data to a selection with `data`
  - By default bound on array index.
  - Can be bound on property of data.
- Select with `d3.select` or `d3.selectAll`

STEP 2: HANDLE NEW ELEMENTS

```
var circles = chart.selectAll('circle')
    .data(getRandomData())
```

- circles is < bound `data`
  - Use `enter()` to create new `circles`

```
circles.enter().append('circle')
.attr("r", function(d) { return d })
.attr("cx", function(d, i) { return i * 100})
.attr("class", "added")
.attr("cy", "50")
```

- Set with: `attr(..)`, `style(..)`, `html(..)`, `text(..)` and `classed(..)`

STEP 3 AND 4: UPDATE AND REMOVE EXISTING ELEMENTS

- Update the data elements bound to existing **circle**s

```
// set the class to updated and update radius
circles.attr("class", "updated")
     .attr("r", function(d) { return d })
```

- Remove the circles without bound data

```
circles.exit().remove();
```

~ And with that ~

SIDESTEP: TRANSITIONS

- Interpolate change on a selection

```
// contains the circles, which are being reused
circles.attr("class", "updated")
     .transition().duration(2000)
     .attr("r", function(d) { return d })
```

- Works out of the box for:
  - `number`, `color`
  - `string`: finds numbers in the string
- Custom using: `attrTween` or `styleTween`
- https://github.com/d3/d3-transition

DATABINDING ON INDEX

```
var input = [
{id: 3, value: 22.639784786233 12}
{id: 6, value: 23.562434755740533}
{id: 8, value: 23.827175330644 51}];

var circles = chart.selectAll('circle').data(input
   , function(d) {return d.id});
```

- With `data` we can provide an accessor function.
- Data is now bound and updated on `d.id`
- E.g useful for sorting and moving elements

CODE: PATTERN, TRANSITIONS AND DATA BINDING

```
var circles = chart.selectAll('circle')
    .data(getRandomData(), function(d) {return d.index});

circles.enter().append('circle').attr("class", "added")
    .attr("cy", "50")
    .transition().duration(2000)
    .attr("cx", function(d, i) { return i * 100})
    .attr("r", function(d) { return d.value });

circles
    .attr("class", "updated")
    .transition().duration(2000)
    .attr("cx", function(d, i) { return i * 100})
    .attr("r", function(d) { return d.value });

circles.exit()
    .transition().duration(2000)
    .attr("r", 0).style("opacity", 0)
    .on('end', function(d) { this.remove() });
```

~ End of pattern 1 ~


PATTERN 2:
GENERATORS

D3: USING GENERATORS

*SVG provides a number of **primitives** (circle, rect, ellipse, line, polyline, polygon) used for drawing. Real power, though, comes from the '**path**' element. Writing '**path**'s by hand is rather complex. With **generators**, D3.js wraps the **complexity** in a simple function.*

THE PATH ELEMENT IS VERSATILE

- the `d` attribute describes a path:

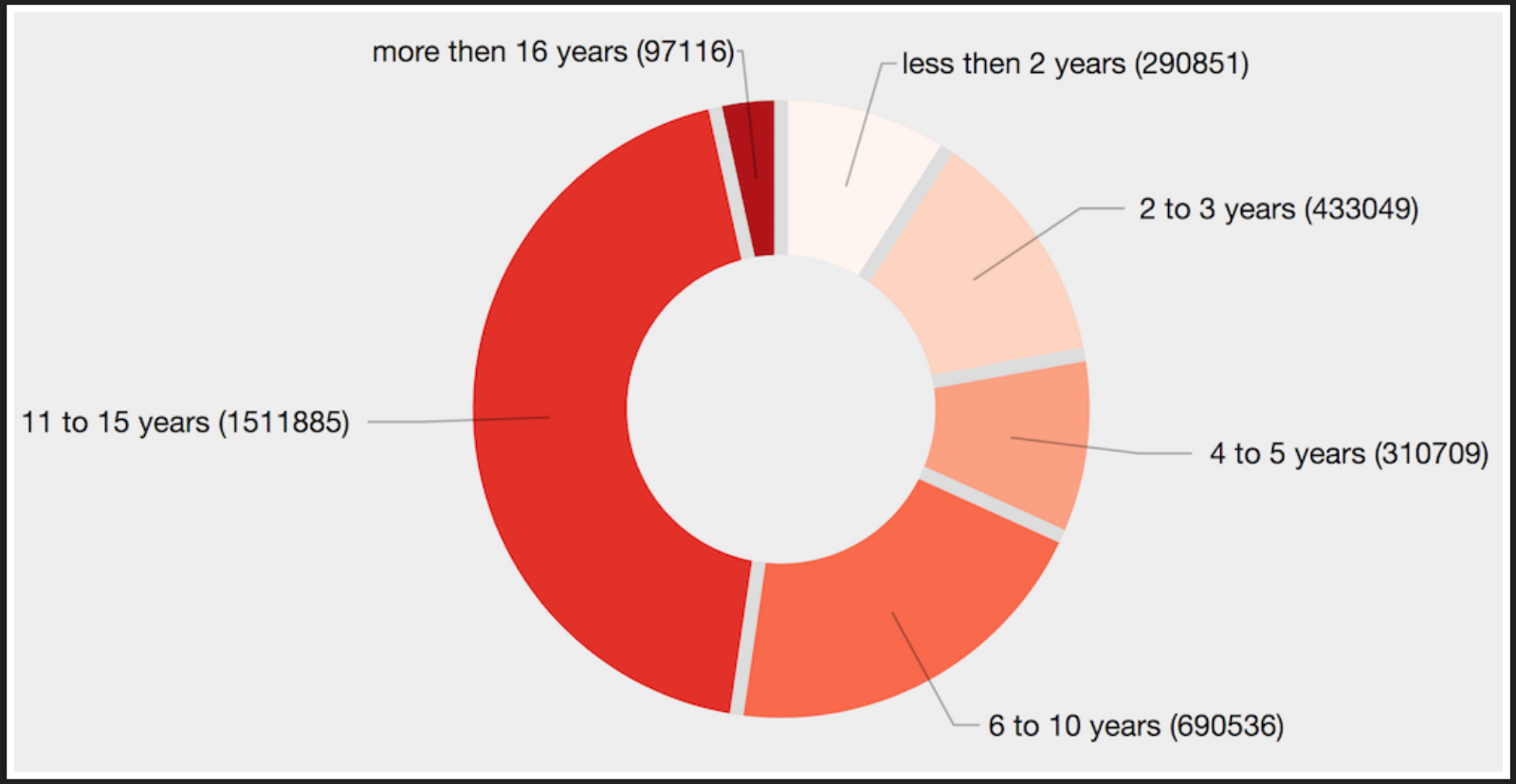
```
<path class="arc" d="M136.86141570725613,-17.6904745726513 7A13
0,0,1,124.60150267951192,59.31665474390461L62.948628653284814,
25721413499303 5A69,69,0,0,0,68.61145448511735,-7.312203049469 5
style="fill: rgb(252, 160, 130);"></path>
```

- `M`, `L`, `A`, `C`, `A`, `Q`, `T` ... for lines, arcs, curves
- Hard to determine the correct values yourself
- D3.js provides generators for complex shapes

THE ARC GENERATOR

```
var arc = d3.arc().outerRadius(200).innerRadius(100);

// generate a path string
var pathString = arc({
  startAngle: 0,
  endAngle: Math.PI
});

// "M1.469576158976824e-14,-240A240,240,0,1,1,1.469576158976824
// 240L7.347880794884120e-15,120A120,120,0,1,0,7.347880794884120
// -120Z"
select(".chart").append("path").attr("d", pathString);
```

NOW IT'S EASY TO CREATE A PIE CHART



Generator

COMBINED WITH THE PIE FUNCTION

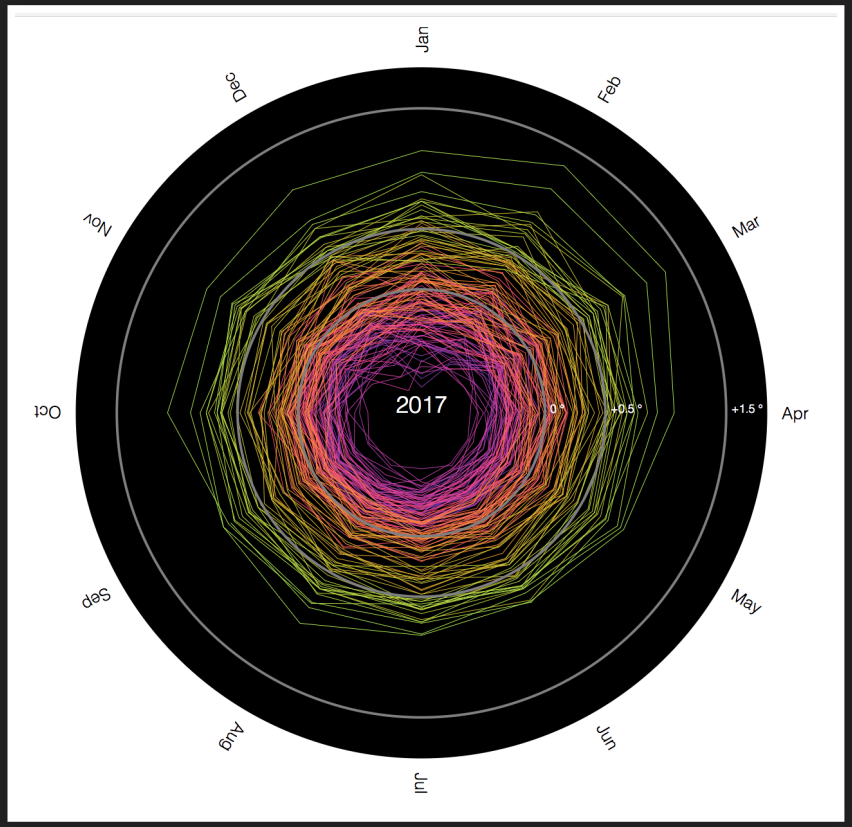
- `d3.pie()` to generate config for `d3.arc`

```
var arc = d3.arc()
    .outerRadius(height/2 * 0.6).innerRadius(height/2 * 0.3);
var data = [{count:10}, {count:20}, {count:30}]

var pie = d3.pie().padAngle(0.04)
.value(function (d) { return d.count; });

var arcs = pie(data)
// "[{"data":{"count":10},"index":2,"value":10,
//    "startAngle":5.215987755982988,
//    "endAngle":6.283185307179586,"padAngle":0.04},
//   {"data":{"count":20},"index":1,"value":20,
//    "startAngle":3.121592653589793,
//    "endAngle":5.215987755982988,"padAngle":0.04} ...
selectAll("path").data(arcs).enter()
   .append("path").attr("d", arc);
```

ANOTHER STANDARD PATTERN

1. (Optionally) Use a generator which creates **config** for other generators (`d3.pie`)
2. Pass initial **data** through step 1, result is **enriched data** with config.
3. Use the normal `selectAll()`, `enter()`, `merge()`, `exit()` pattern
4. Use generator which creates a **path** based on properties. (`d3.arc`)

USING ANOTHER GENERATOR



Temperature spiral

SPIRAL DATA


```
▼ Array(138) i
  ▼ [0 … 99]
    ▼ 0:
      ▼ months: Array(13)
        ▼ 0:
            month: "Jan"
            monthIndex: 0
            originalValue: "-.31"
            value: -0.31
          ▶ __proto__: Object
        ▶ 1: {month: "Feb", monthIndex: 1, originalValue: "-.20", value: -0.2}
        ▶ 2: {month: "Mar", monthIndex: 2, originalValue: "-.12", value: -0.12}
        ▶ 3: {month: "Apr", monthIndex: 3, originalValue: "-.21", value: -0.21}
        ▶ 4: {month: "May", monthIndex: 4, originalValue: "-.13", value: -0.13}
        ▶ 5: {month: "Jun", monthIndex: 5, originalValue: "-.25", value: -0.25}
        ▶ 6: {month: "Jul", monthIndex: 6, originalValue: "-.22", value: -0.22}
        ▶ 7: {month: "Aug", monthIndex: 7, originalValue: "-.11", value: -0.11}
        ▶ 8: {month: "Sep", monthIndex: 8, originalValue: "-.17", value: -0.17}
        ▶ 9: {month: "Oct", monthIndex: 9, originalValue: "-.25", value: -0.25}
        ▶ 10: {month: "Nov", monthIndex: 10, originalValue: "-.21", value: -0.21}
        ▶ 11: {month: "Dec", monthIndex: 11, originalValue: "-.24", value: -0.24}
        ▶ 12: {month: "Jan", monthIndex: 0, originalValue: "-.17", value: -0.17}
          length: 13
        ▶ __proto__: Array(0)
        year: "1880"
```

SPIRAL CODE: SETUP 1/2

```
var minR = 20;
var maxR = height/2;

// determine min and max values to determine range
var minValue = years.reduce ....
var maxValue = years.reduce ....

// set the scale for the radius
var radius = d3.scaleLinear().range([minR, maxR]).domain([minValue, maxValue+0.3]);
var color = d3.scaleSequential(d3.interpolateRainbow).domain([0, years.length]);

// configure the path generator
var radialGenerator = d3.lineRadial();
radialGenerator.radius(function(d) { return radius(d.value); });
radialGenerator.angle(function(d) { return ((2*Math.PI)/12) * d.monthIndex; });
```

SPIRAL: RENDER LOOP 2/2

```
function appendYear() {

    // select and bind the data
    var dataToShow = years.slice(0, year);
    var current = graph.selectAll("path").data(dataToShow);

    // We only add new data, we don't update existing data.
    current.enter()
        .append("path")
        .attr("d", function(d,i) { return radialGenerator(d.months)})
        .attr("stroke", function(d,i) { return color(i)});

    graph.select(".currentYear").text(dataToShow[dataToShow.length-1].year);

    year++;
}
```

SIDESTEP: SVG AND POSITIONING

- Absolute positioning:

```
<rect x="100", y="100"></rect>
<circle cx="100", cy="100"></circle>
```

- Provides a `g` element for grouping
  - `g` has no `x` or `y`, uses `transform`
  - All attributes on this element apply on children

```
<g transform="translate(50 200)">...</g> <!-- positioning -
<g transform="scale(0.5 0.3)">...</g> <!-- sizing -->
<g transform="rotate(45)">...</g> <!-- rotate the g -->
```

- Actions can be combined, still annoying

POSITIONING WITH SCALES

- A scale translates input domain to an output range
- Different scales (https://github.com/d3/d3-scale)
  - **scaleLinear**: continuous input to cont. output
    - scalePow, scaleLog, scaleTime
  - **scaleSequential**: continuous input to interpolator
  - scaleQuantize: continuous input to discrete range
  - scaleQuantile: sampled input to discrete range
  - scaleThreshold: scaleQuantize with thresholds
  - scaleOrdinal: discrete input to discrete range

SCALELINEAR: CONTINUOUS INPUT TO CONTINUOUS OUTPUT

```
var x = d3.scaleLinear()
    .domain([10, 130])   // e.g the min and max of your data
    .range([0, 960]);    // the width of your chart

x(20); // 80
x(50); // 320

var color = d3.scaleLinear()
    .domain([10, 100])
    .range(["brown", "steelblue"]);

color(20); // "#9a3439"
color(50); // "#7b5167"
```

WHAT CAN WE INTERPOLATE?

- Domain is numeric, the range can be any of this:

d3.**interpolate**(*a*, *b*)

Returns an interpolator between the two arbitrary values *a* and *b*. The interpolator implementation is based on the type of the end value *b*, using the following algorithm:

1. If *b* is null, undefined or a boolean, use the constant *b*.
2. If *b* is a number, use interpolateNumber.
3. If *b* is a color or a string coercible to a color, use interpolateRgb.
4. If *b* is a date, use interpolateDate.
5. If *b* is a string, use interpolateString.
6. If *b* is an array, use interpolateArray.
7. If *b* is coercible to a number, use interpolateNumber.
8. Use interpolateObject.

- Easy to create your own
- Also be used in **transitions**


PATTERN 3:
INTERACTION

INTERACTIONS

*Extend the **select** pattern and respond to user interactions with the elements on screen.*

D3 MAKES ADDING LISTENERS EASY

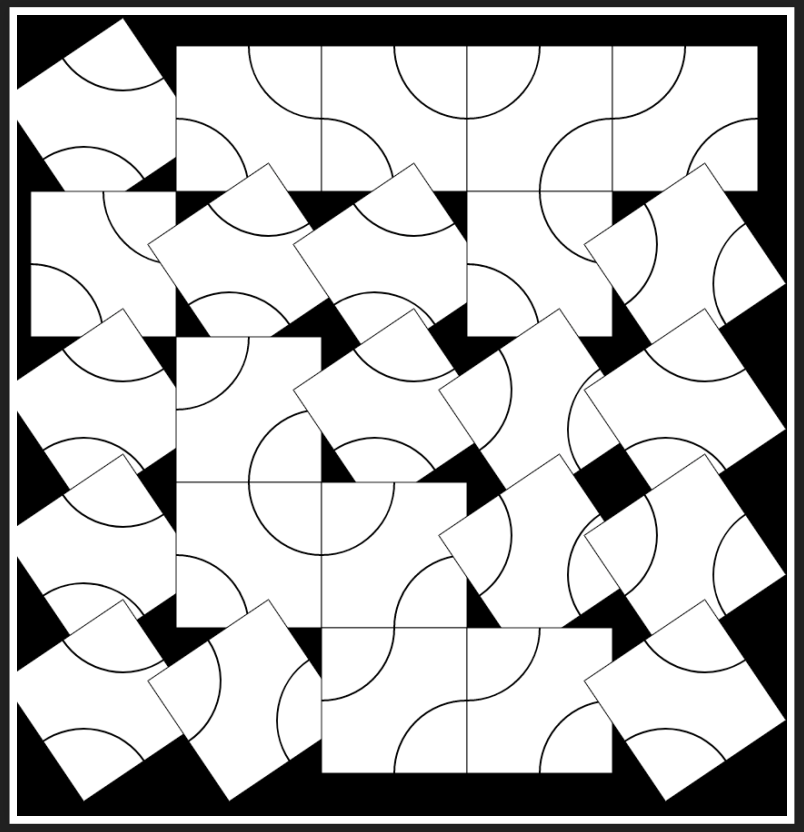
- Part of the **select**/**add**/**update**/**remove** pattern
- Use the `on` function (with any DOM event type):
- `event.name` when multiple listeners for event

```
// on enter() also add event listeners
circles.enter().append('circle').attr("class", "added")
   .attr("cy", "50")
   .attr("cx", function(d, i) { return i * 100})
   .attr("r", function(d) { return d.value })
   .on("mouseover", handleMouseOver)
   .on("mouseup.foo", handleMouseUp1)
   .on("mouseup.bar", handleMouseUp2)
   .on("mouseenter mouseleave", handleMouseEnterMouseLeave)
```

EVENT LISTENERS IN D3

```
function handleMouseOver(d, i, nodes) {
    var boundData = d;
    var index = i;
    var currentElement = nodes(i);  // also bound to this

  // Select element, and add a transition
  d3.select(this)
    .transition()
    .attr({ fill: "orange", r: 100 });
}
```

STATIC EXAMPLE

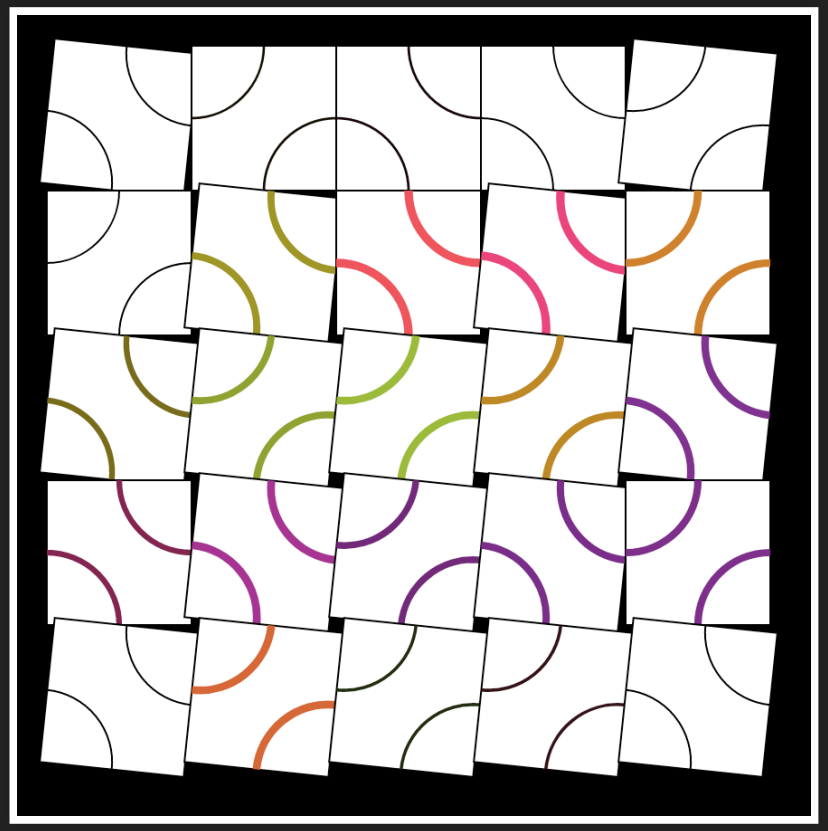


Rotating rectangles

ADD INTERACTIVITY

```
var tiles = chart.selectAll("g").data(data).enter()...

tiles.on("mouseover", function(d, i ,nodes) {
    d3.select(this)
        .selectAll("path")
        .attr("stroke", colorScale(Math.random()))
        .attr("stroke-width", "10")
});

tiles.on("mouseout", function(d, i ,nodes) {
    d3.select(this).selectAll("path")
        .transition().duration(5000)
        .attr("stroke", "#000")
        .attr("stroke-width", "2")
});
```

INTERACTIVE EXAMPLE



Rotating colored rectangles

SOME ADDITIONAL HELPERS

- Helpers to get more information:
  - `d3.event`: the current event
  - `d3.mouse`: `x`,`y` relative to a container
  - `d3.touch`: `x`,`y` of touch relative to a container
  - `d3.touches`: `x`,`y` of touches rel. to a container

THERE IS MORE

- `d3.brush`: select multiple elements by clicking and dragging the mouse.
- `d3.drag`: drag and drop any element to a new location

d3.brush


PATTERN 4:
LAYOUTS

LAYOUTS

*With **generator**s we generate **path** attributes, which we can position ourselves. With a layout we can have D3 determine the **correct positions** of our elements, and we only have to determine how to visualize them.*

D3 AND LAYOUTS

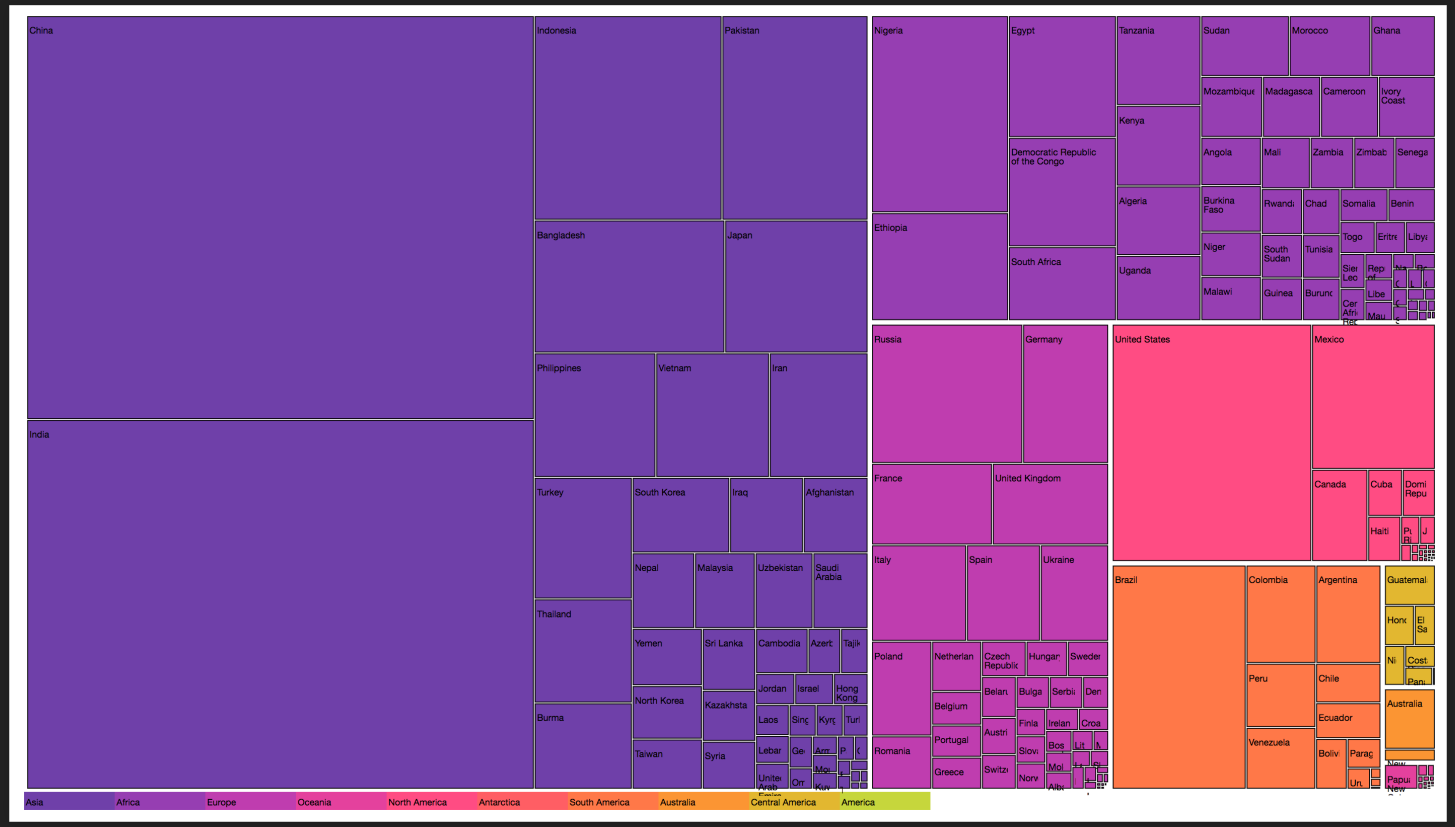
- D3 comes with a standard set of layouts: **Chord**, **Force**, **Cluster**, **Tree**, **Treemap**, **Partion**, **Pack**
- Except **Chord** and **Force** all follow the same pattern (more on this later)

Radial Tree Layout

BASIC LAYOUT PATTERN

1. Load the data (`d3.csv`,`d3.tsv`,`d3.json` etc.)
2. If necessary, convert it to a nested structure
   - D3 has helpers: `d3.stratisfy`, `d3.nest`
3. Convert the nested data into the required stucture (`d3.hierarchy`)
4. Pass it through a generator (e.g `Tree`)
5. Use the output from the generator to draw elements

LETS CREATE A TREEMAP



Sample treeMap

More information contact me: @josdirksen / jos.dirksen@gmail.com / www.smartjava.org / github.com/josdirksen

STEP 1: LOAD THE DATA

```
"Country (en)";"Country code";"Continent";"Population";"Area";
"Afghanistan";"AF";"Asia";32564342;652230;0;
"Egypt";"EG";"Africa";88487396;1001450;2450;
..
```

```
d3.text('./data/countries.csv', function(raw) {
  var data = d3.dsvFormat(";").parse(raw)
  ... // do some data cleanup
```

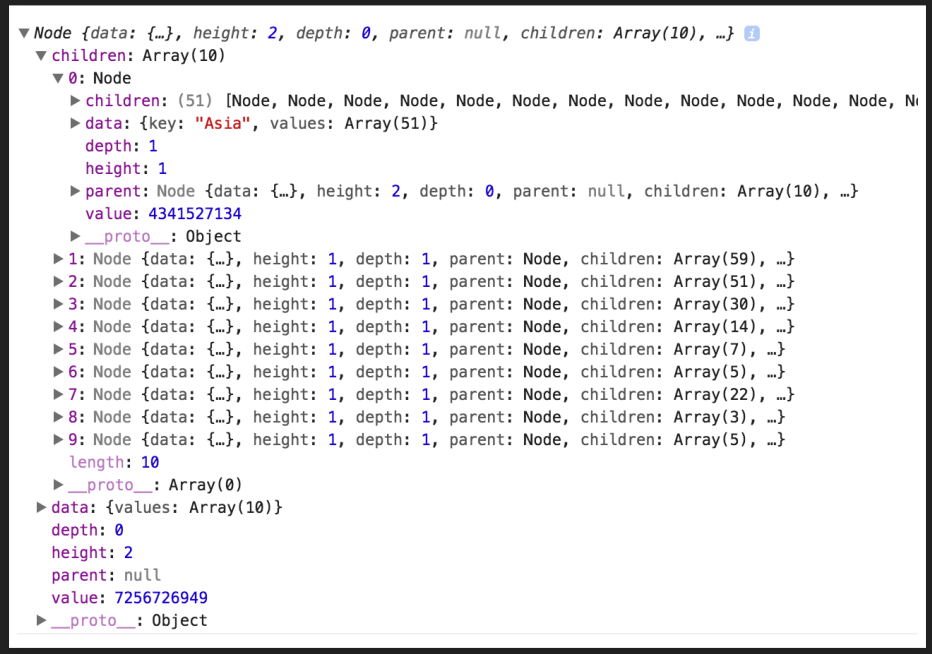
STEP 2: CONVERT TO NESTED

```
// group entries using nest and create hierarchy per continent
var entries = d3.nest()
              .key(function (d) {return d.Continent; })
              .entries(data);
// and create a root node
var source = { values: entries }
```

```
▼ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶ 0: {key: "Asia", values: Array(51)}
  ▶ 1: {key: "Africa", values: Array(59)}
  ▶ 2: {key: "Europe", values: Array(51)}
  ▶ 3: {key: "Oceania", values: Array(22)}
  ▶ 4: {key: "North America", values: Array(30)}
  ▶ 5: {key: "Antarctica", values: Array(5)}
  ▶ 6: {key: "South America", values: Array(14)}
  ▶ 7: {key: "Australia", values: Array(5)}
  ▶ 8: {key: "Central America", values: Array(7)}
  ▶ 9: {key: "America", values: Array(3)}
```

STEP 3: CONVERT TO HIERARCHY

```
// use our nested data, and tell d3 that the
// children can be accessed through the values property
var root = d3.hierarchy(source, function(d){return d.values;})
    // Count the populations to create the hierarchy
    .sum(function(data) { return data['Population']; })
    // sort the hierarchy based on value
    .sort(function(a, b) { return b.value - a.value; });
```



STEP 4: ENHANCE WITH THE TREEMAP LAYOUT

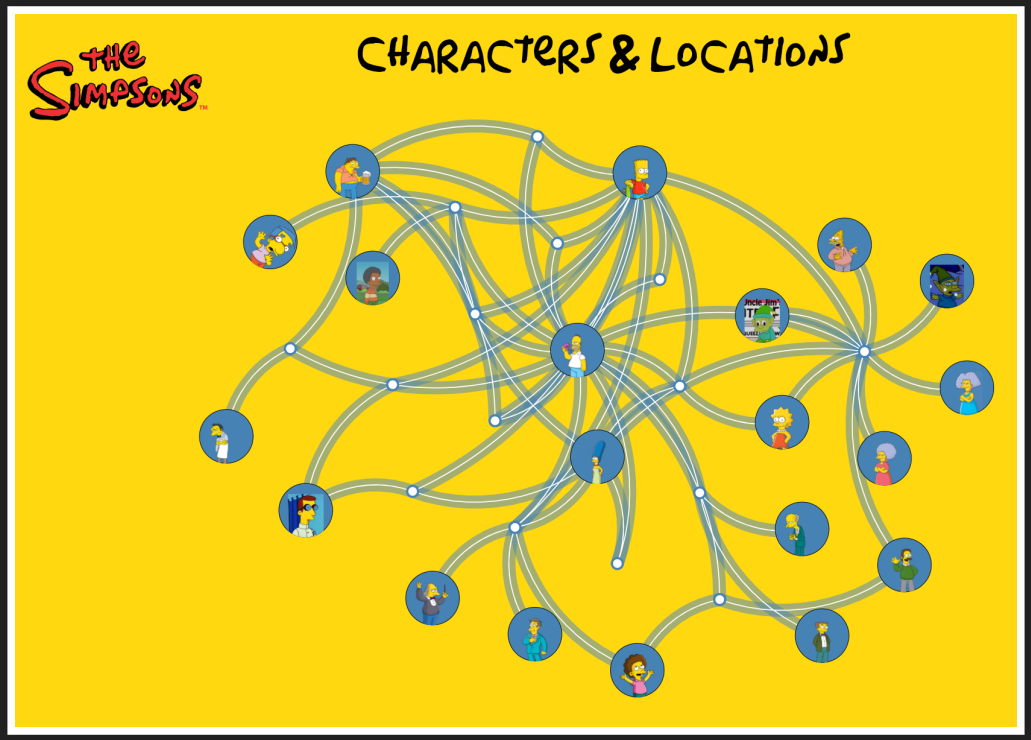
```
var tree = d3.treemap().size([width, height]).padding(2)
tree(root);
```

STEP 5: SELECT/ADD/UPDATE/REMOVE

```
// leaves is returns all the elements in an array
var countries = chart.selectAll(".node")
    .data(root.leaves(), function(d) {return d.data.id})

// append a rectangle for each group and set the color
var newCountries = countries.enter().append("rect")
    .style("fill", function(d) {return colorScale(d.parent.data.key)})
    .style("stroke", "black").attr("class","node")
    .on("click", onclick)

// merge two selections for common functionality
var allGroups = countries.merge(newCountries);

// Step 6. Create a transition in which we update the x, y, widht and height
// of the rectangles based on the d.x0, d.x1, d.y0 and d.y1 values.
allGroups.transition().duration(2000)
    .attr("x", function(d) {return d.x0}).attr("y", function(d) {return d.y0})
    .attr("width", function(d) {return d.x1 - d.x0})
    .attr("height", function(d) {return d.y1 - d.y0});
```

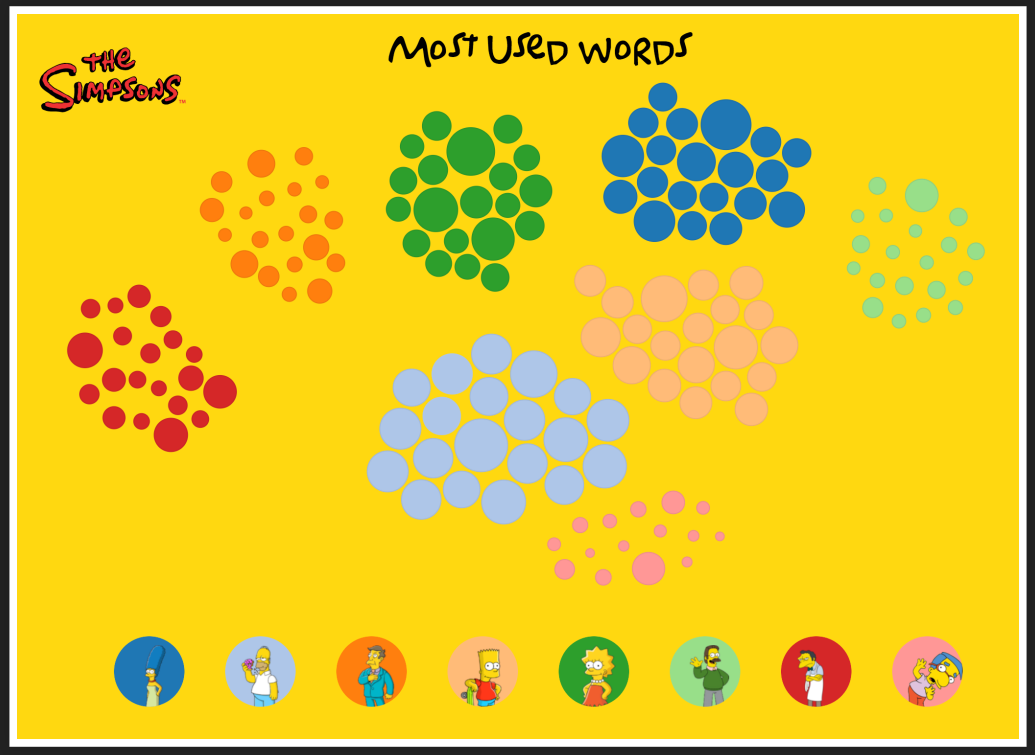
Result

SIDENOTE: FORCE LAYOUT WITH LINKS AND NODES



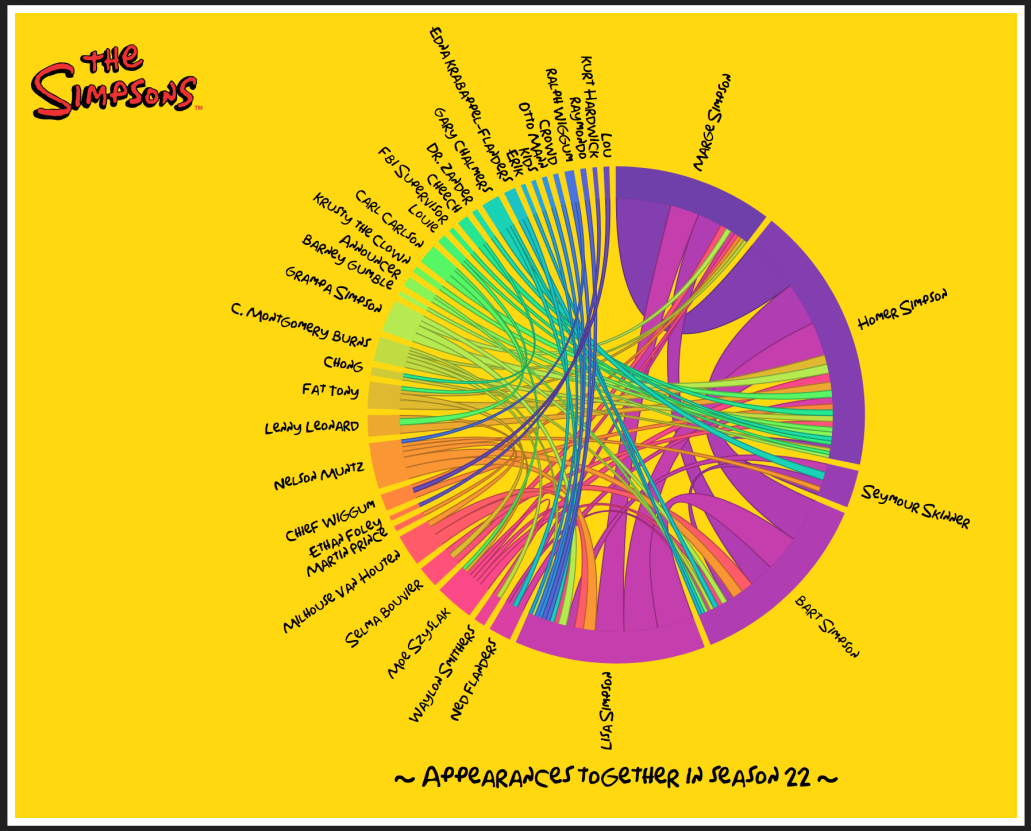
Force layout 1

SIDENOTE: FORCE LAYOUT WITH JUST NODES



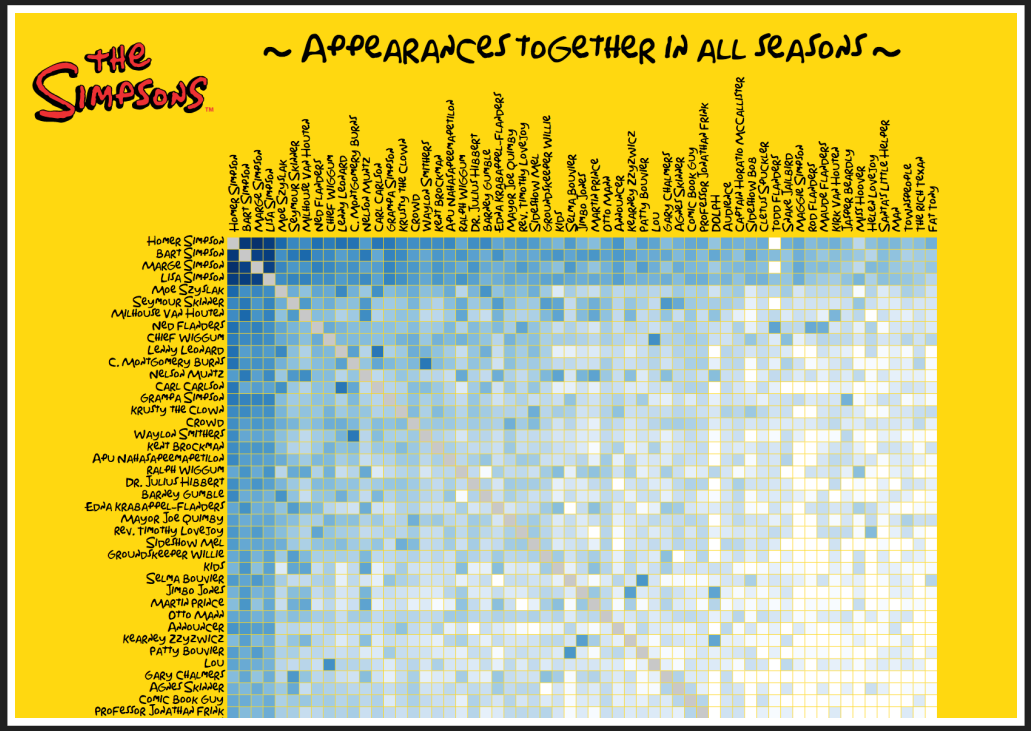
Force layout 2

SIDENOTE: MATRIX DATA WITH CHORD



Chord Layout

SIDENOTE: MATRIX DATA WITH TABLE



Custom Layout


EXTRAS:
MAPS

CREATING MAPS

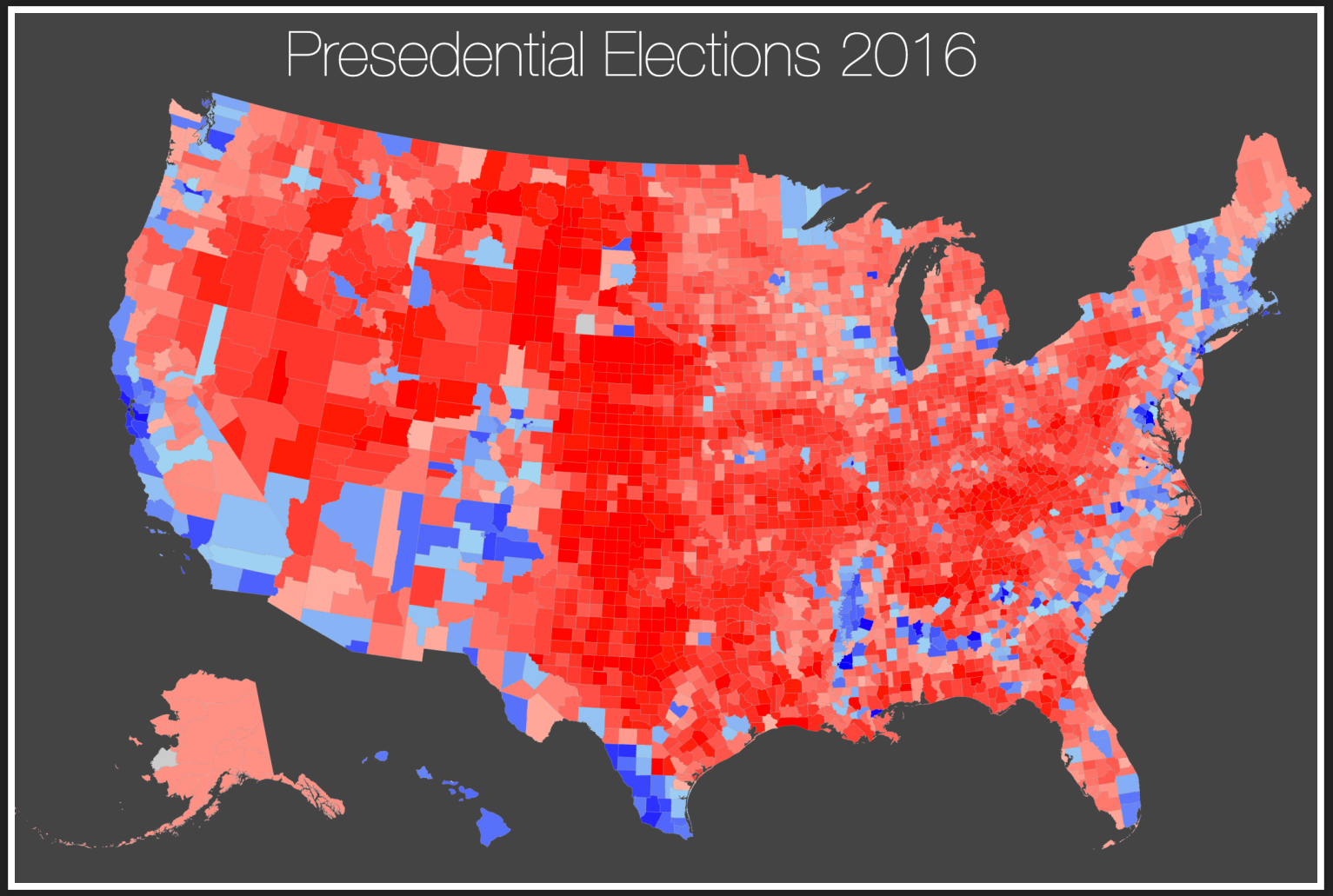
*Often used way to represent data is on **maps**. D3 provides a standard way to easily create **interactive** maps in different **projections**.*

(Not really a pattern, but nicely shows a combination of the other patterns and concepts we've seen)

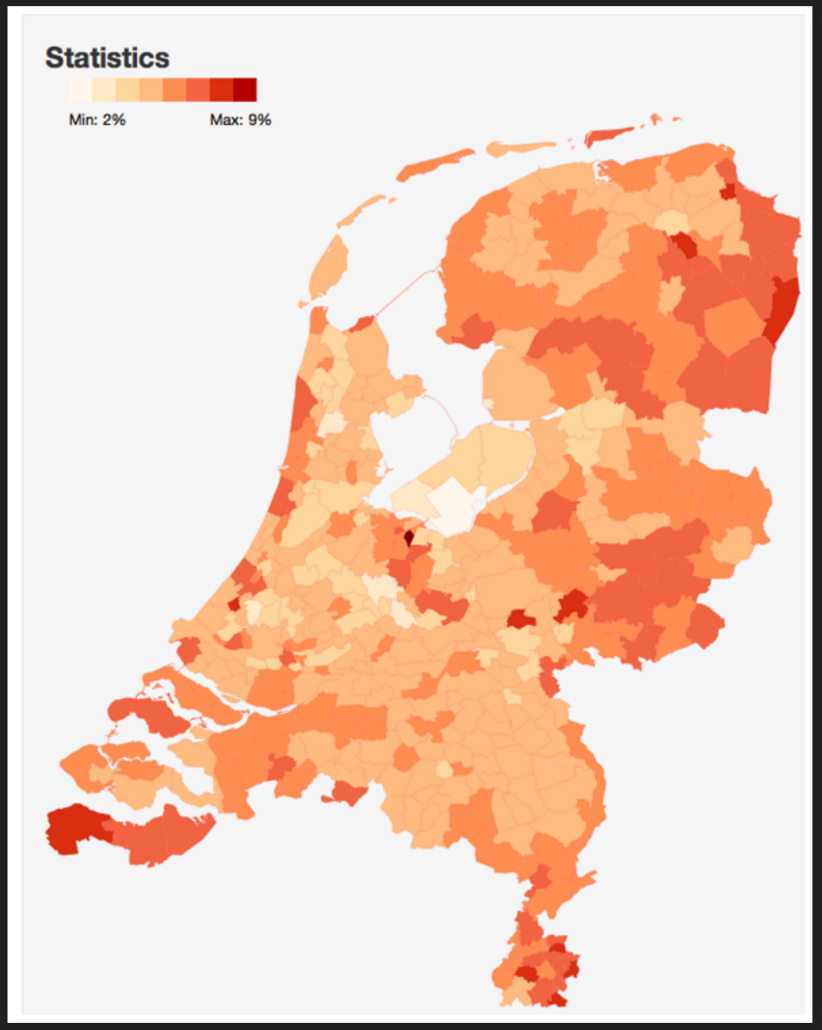
BEST PART OF D3.JS

At least for me.... since I really like maps

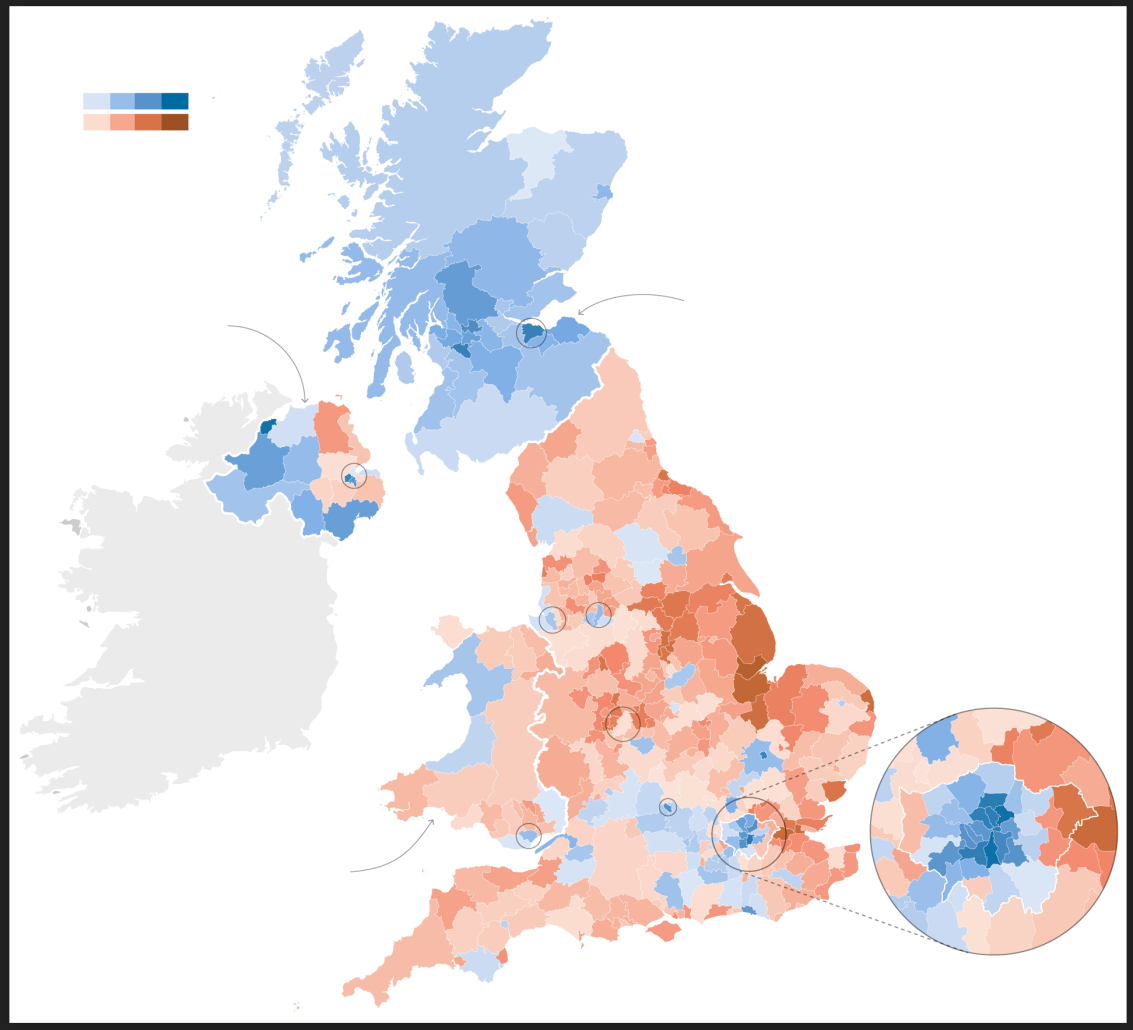
CAUSE IT LOOKS NICE



AND INFORMATIVE

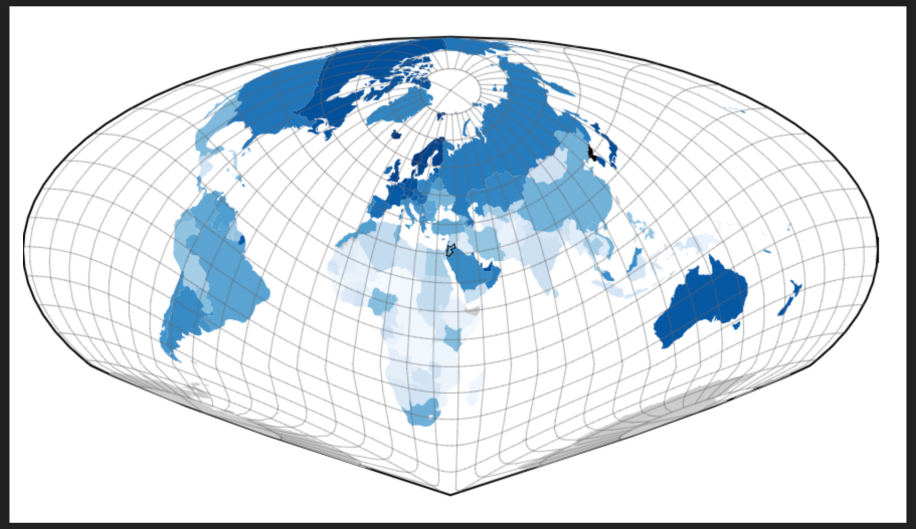


AND BEAUTIFUL



QUICK NOTE ON PROJECTIONS

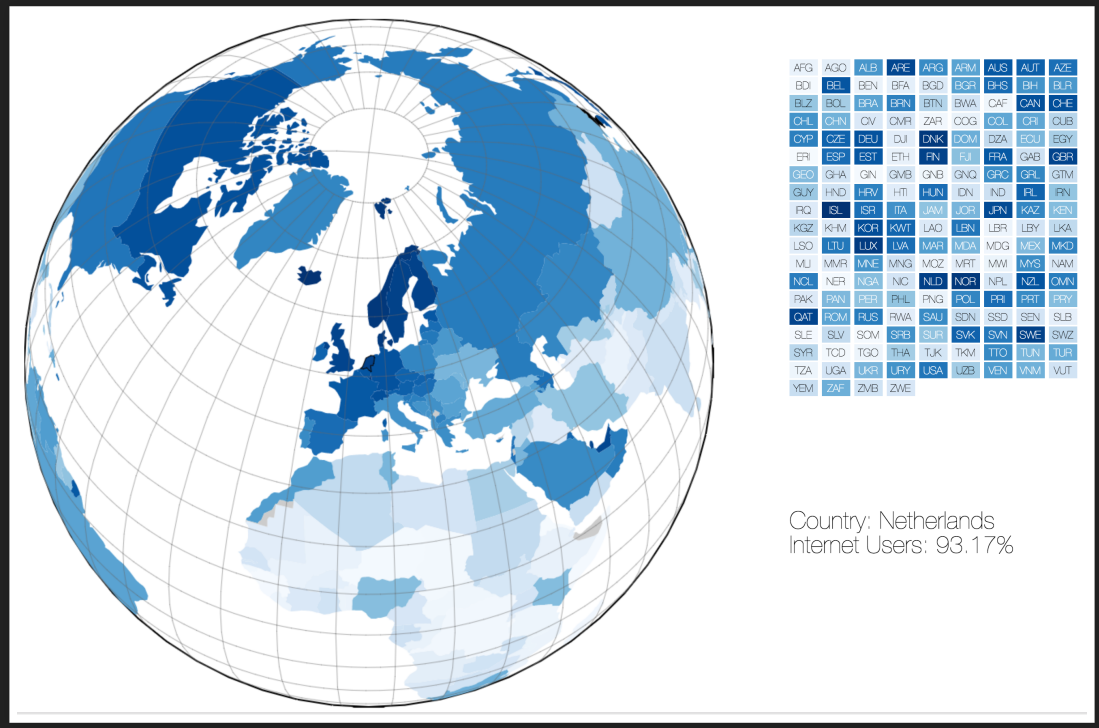
Defines how the source **coordinates** (lat, lon) are projected to a **2D** flat canvas. E.g: **Sinu Mollweide**:



Example

VISUALIZING IS EASY

Lets recreate this:



Example

More information contact me: @josdirksen / jos.dirksen@gmail.com / www.smartjava.org / github.com/josdirksen

NOTE ON DATA

- D3.js can work with:
  - **GeoJSON**: A standard supported by many applications.
  - **TopoJSON**: A specific optimized format for smaller files.
- Most common format is **ArcGIS** shapefile.
  - binary format
  - Use QGis, Mapshaper, OGR to convert

GEOJSON AND TOPOJSON FORMATS

- Can contain multiple geometries.
- Each geometry is described (usually) as a path.
- Can contain additional properties
  - population, unemployment rate, etc.
  - one file for everything
- Try to work with TopoJSON, since it's much smaller


```
{"type":"Topology","objects":{"countries":{"type":"GeometryCollection",
"geometries":[{"type":"Polygon","id":"004","arcs":[[0,1,2,3,4,5]],
"properties":{"value":"7","countryA":"AFG","name":"Afghanistan"}},
{"type":"MultiPolygon","id":"024","arcs":[[[6,7,8,9]],[[10,11,12]]],
"properties":{"value":"10.2"....
```

PATTERN: TOPO DATA AND D3.JS

1. Load the data
2. Setup a path **generator** and projection
3. Use the projection to generate path segments
4. Use **select** / **bind** / **update** / **remove** pattern

SIDESTEP: LOADING DATA

- D3.js provides standard loaders:
  - `d3.csv`, `d3.tsv`
  - `d3.json`, `d3.xml`
  - `d3.text`, `d3.html`

```
d3.csv("somedata.csv", function(rows) {
   // do something with the data
});
```

SETUP THE PATH GENERATOR

```
var projection = d3.geoNaturalEarth()
var path = d3.geoPath().projection(projection)
```

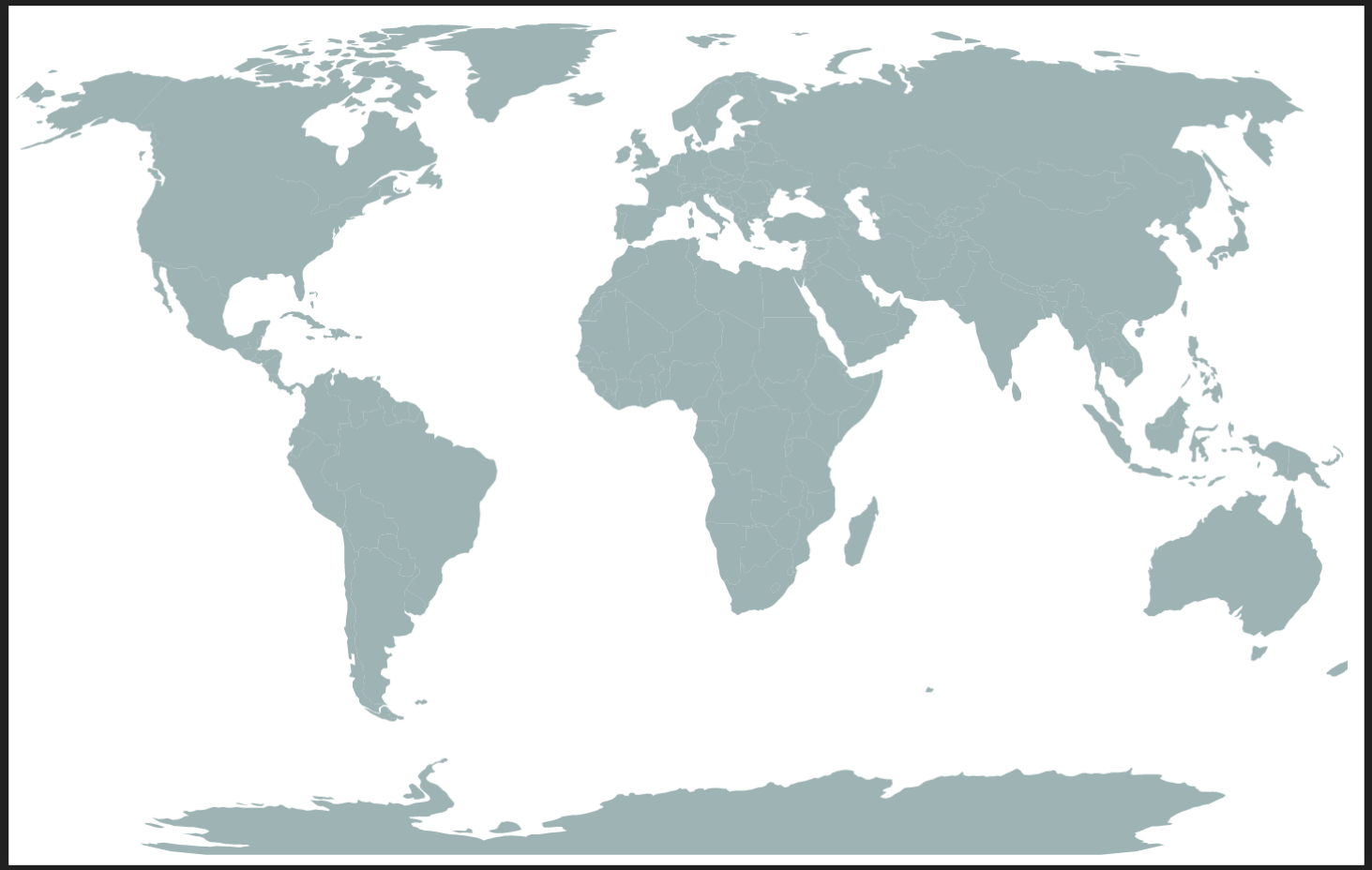
- For all the supported projections see:
  - https://github.com/d3/d3-geo/
  - https://github.com/d3/d3-geo-projection
- Pretty much any projection you can think off
  - Relatively easy to create your own one

WITH A GENERATOR AND THE DATA

```
var projection = d3.geoNaturalEarth()
var path = d3.geoPath().projection(projection)

d3.json("./data/world-110m-inet.json", function(loadedTopo) {
  countries  = topojson.feature(loadedTopo,
                        loadedTopo.objects.countries).features;

  svg.selectAll('.country').data(countries).enter()
     .append("path")
     .classed('country', true)
     .attr("d", path);
}
```

- `topojson.feature`: convert the topoJSON format to geoJSON
- Which can be processed by the path generator

AND YOU'RE DONE!

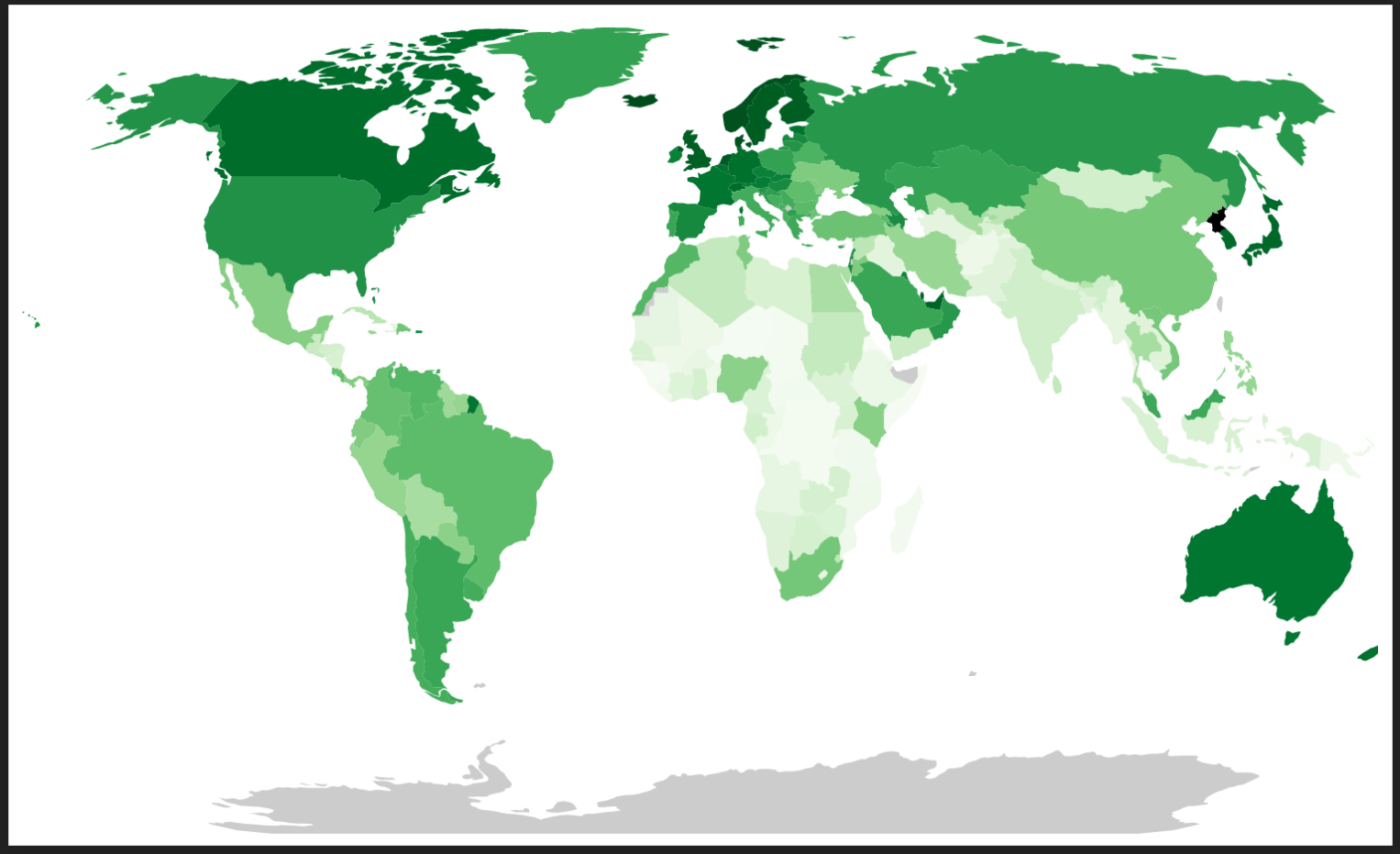


But colors?

ADD A SIMPLE SCALE

```
var color = d3.scaleSequential(d3.interpolateGreens).domain([0,100])
var projection = d3.geoNaturalEarth()
var path = d3.geoPath().projection(projection)

d3.json("./data/world-110m-inet.json", function(loadedTopo) {
  countries  = topojson.feature(loadedTopo,
                      loadedTopo.objects.countries).features;

  svg.selectAll('.country').data(countries).enter()
     .append("path")
     .classed('country', true)
     .attr("d", path);
     .attr("fill", function(d) {return d.properties.value
                ? color(+d.properties.value)
                : '#ccc'});
}
```

EASY RIGHT?



SIDENOTE: SUPPORT OF HTML5 CANVAS

- Also possible to use `<canvas>`

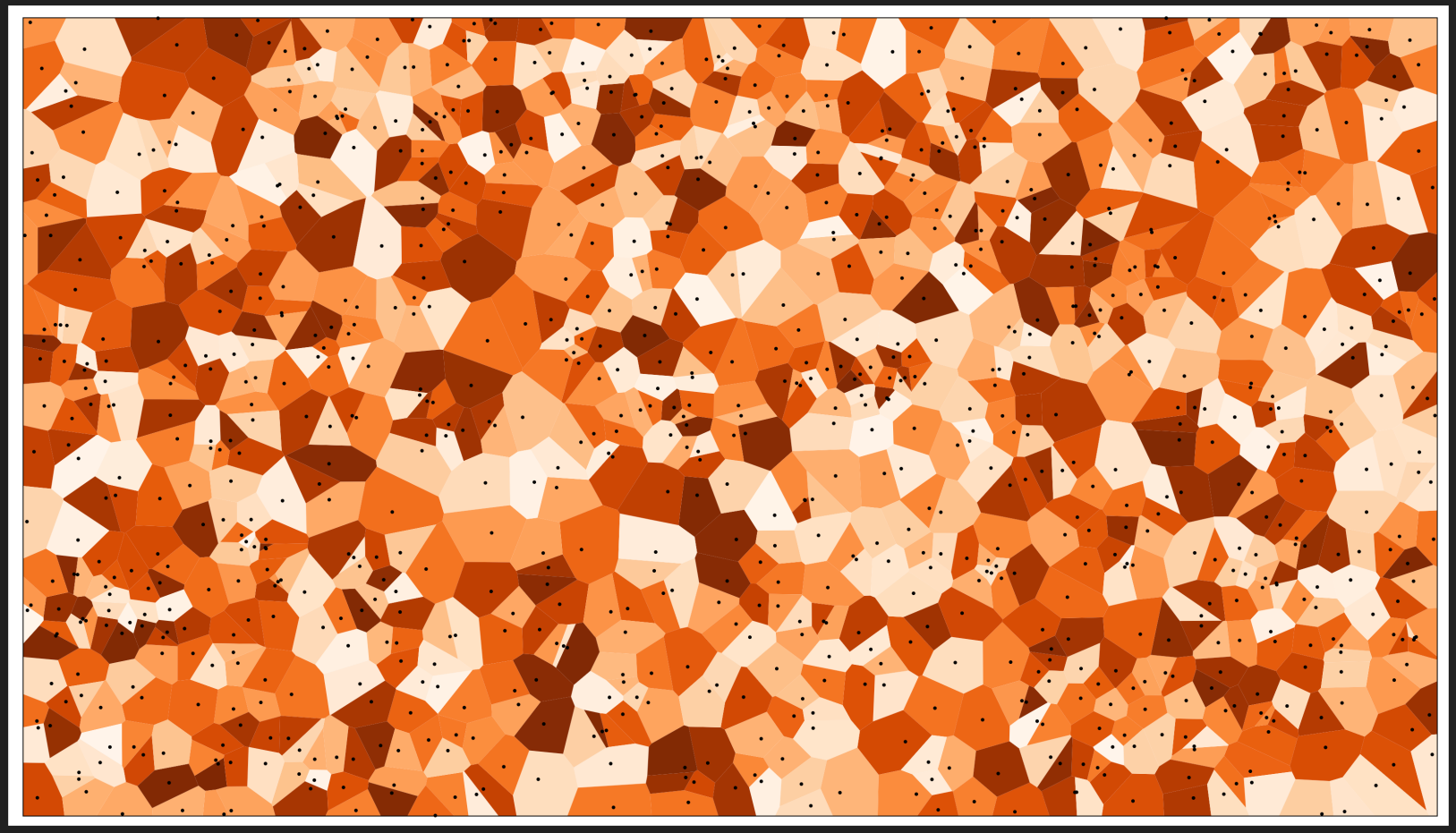
```
var projection = d3.geoNaturalEarth()

// c is a canvas.getContext("2D");
var path = d3.geoPath().projection(projection).context(c);

d3.json("./data/world-110m-inet.json", function(loadedTopo) {
  countries  = topojson.feature(loadedTopo,
                      loadedTopo.objects.countries).features;

  countries.forEach(function(toDraw) {
    toDraw.properties.value ? c.fillStyle = color(toDraw.properties.value)
                            : c.fillStyle = '#ccc';
    c.beginPath();
    path(toDraw);
    c.fill();
  })
});
```


EXTRAS:
GENERATIVE ART

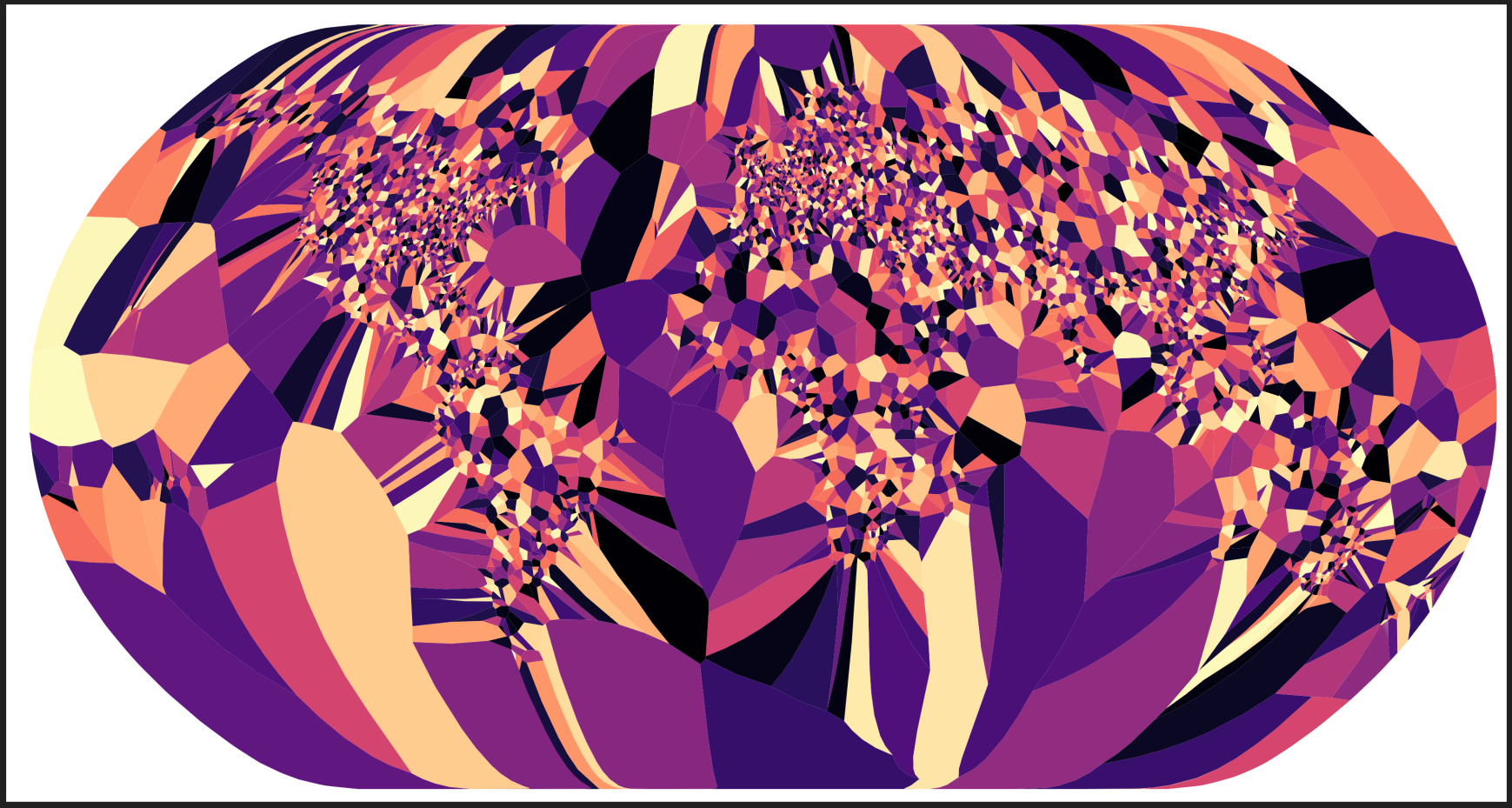
PLAYING WITH VORONOIS

*In mathematics, a **Voronoi** diagram is a partitioning of a plane into regions based on distance to points in a specific subset of the plane. That set of **points** (called seeds, **sites**, or generators) is specified beforehand, and for each seed there is a corresponding **region** consisting of **all points closer to that seed than to any other**. These regions are called Voronoi **cells**. The Voronoi diagram of a set of points is dual to its Delaunay triangulation.*

VORONOI EXAMPLE

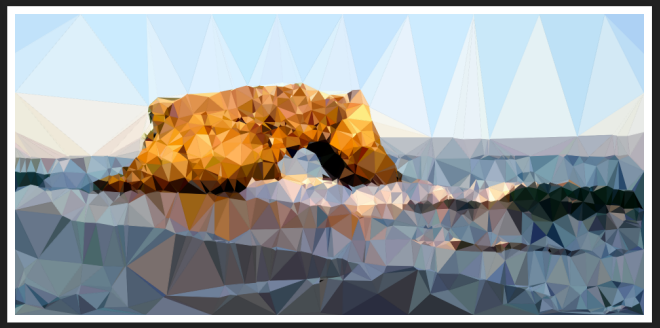


WORLDS AIRPORTS



VORONOI TRIANGULATION

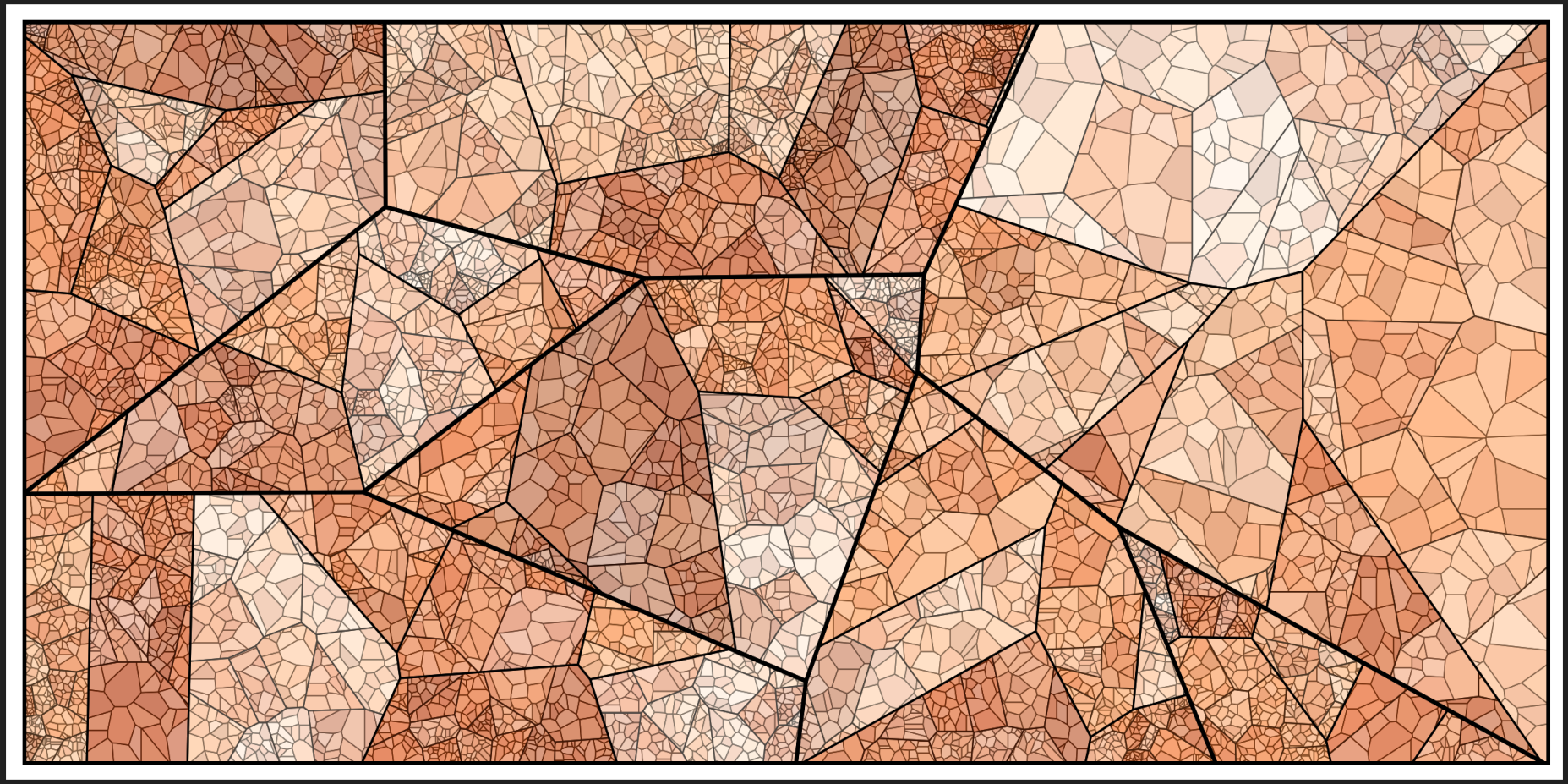





Triangulated using `D3.voronoi` and Delaunay triangulation

More information contact me: @josdirksen / jos.dirksen@gmail.com / www.smartjava.org / github.com/josdirksen

MORE VORONOI: STAINED GLASS



THANK YOU!

- Main d3 sites:
  - http://www.d3js.org
  - https://github.com/d3/d3/blob/master/API.md
- Examples from this presentation:
  - https://github.com/josdirksen/d3exercises
  - https://github.com/josdirksen/d3dv
- This presentation:
  - https://github.com/josdirksen/d3-devoxx-2017
- One of many books:
  - https://www.packtpub.com/web-development/expert-data-visualization