Drawing in the browser

How HTML5 canvas can be great, easy & performant for data visualization

February 8th, 2018

Nadieh Bremer

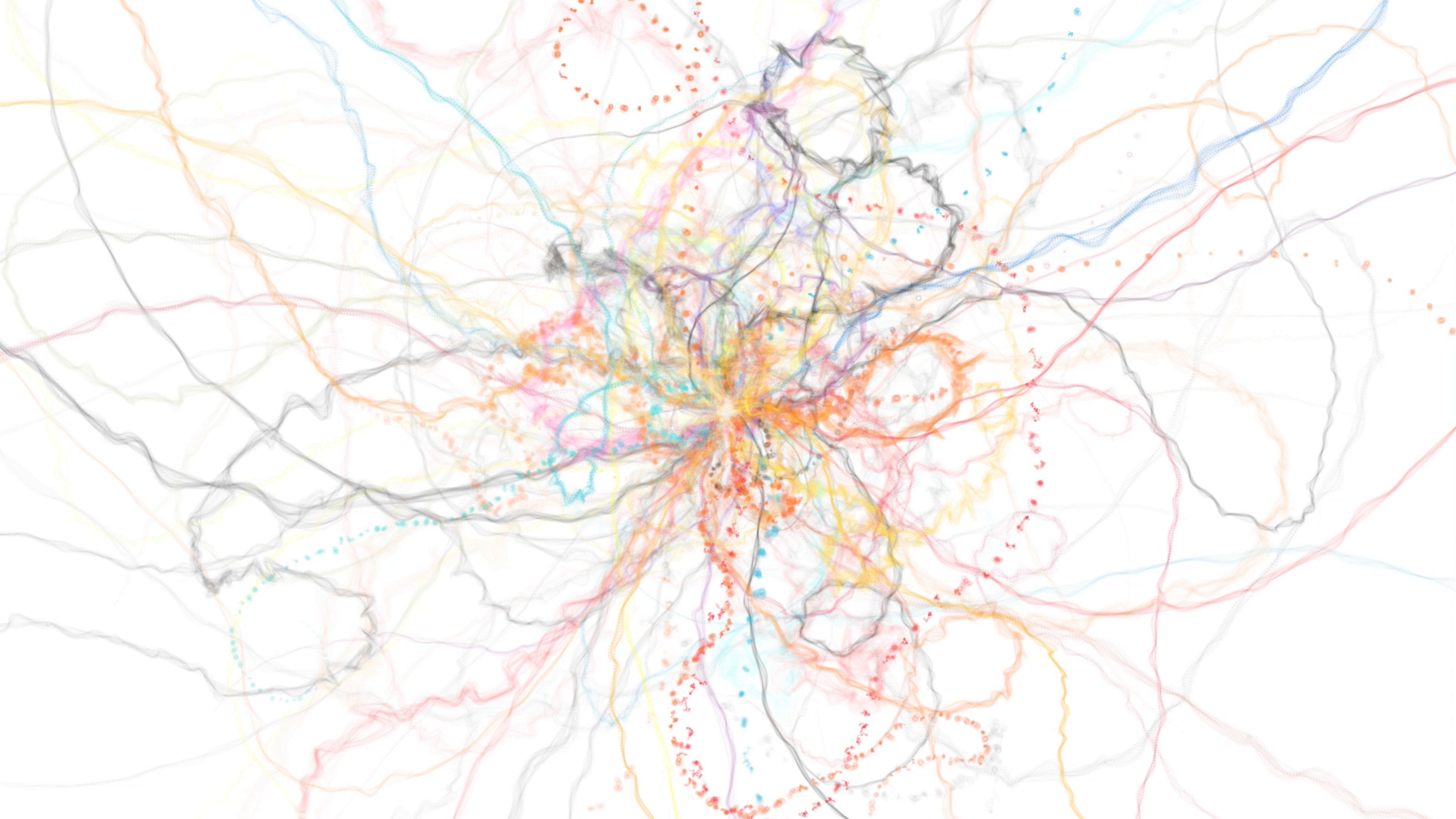
Visual Cinnamon

Canvas

“ **<canvas>** is an HTML element which can be used to **draw** graphics using JavaScript, on the fly! ”

You’re creating an actual image-like rectangle,
with pixels, just like a png or jpg

https://github.com/nbremer/canvastutorial



SVG vs canvas

When to use

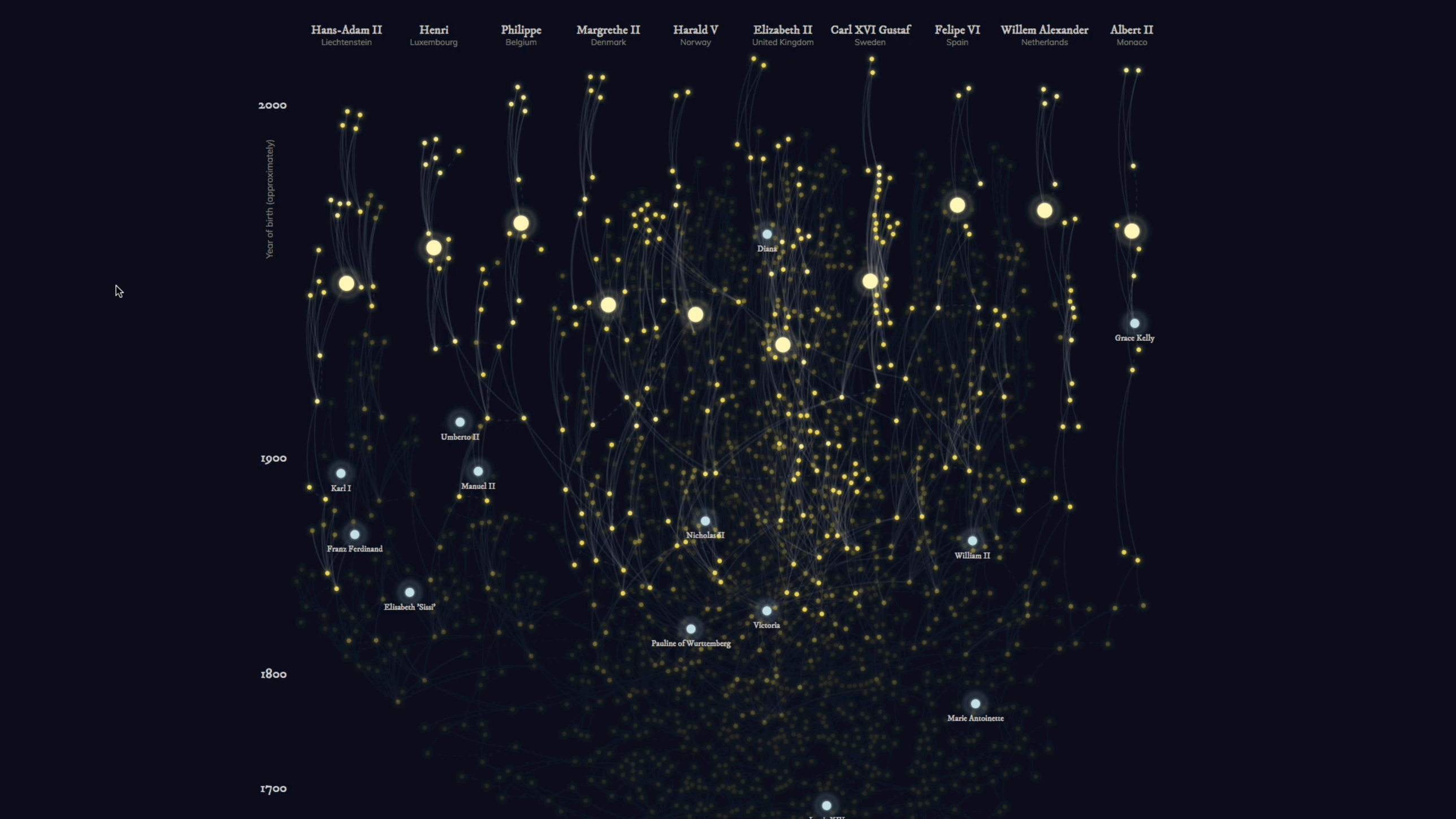
and when not

YAY

- When you want to show thousands of different data points (that don't change/move (too much))
  - Each SVG element creates a DOM element, which can severely slow down the browser when you get >> 1000 elements
  - For canvas only the JS calculation and drawing takes "time", the number of items you draw doesn't matter, it's "just a canvas"
- When speed of showing "something" to the user is important, also on mobile
  - Typically drawing with canvas works wonderfully fast in practically all browsers and devices

NAY

- If perfect crispness is key & you want to adjust the final result in Illustrator
  - If you zoom in, you'll start to see the separate pixels. They're not vectors | But there's a practical solution!
- If you're planning many interactions
  - There are no *mouseover*, *mouseout* like event listeners. There are other ways though, but not as easy as ".on"
- When you have (many different) animations and gradual changes
  - D3 makes changes of state/animations super easy. All of that is lost in canvas. You have to do more effort to "fake" an animation


Hans-Adam II
Liechtenstein
Henri
Luxembourg
Philippe
Belgium
Margrethe II
Denmark
Harald V
Norway
Elizabeth II
United Kingdom
Carl XVI Gustaf
Sweden
Felipe VI
Spain
Willem Alexander
Netherlands
Albert II
Monaco
Year of birth (approximately)
2000
1900
1800
1700
Diana
Grace Kelly
Umberto II
Karl I
Manuel II
Nicholas II
Franz Ferdinand
William II
Elisabeth 'Sissi'
Victoria
Pauline of Württemberg
Marie Antoinette

When to use

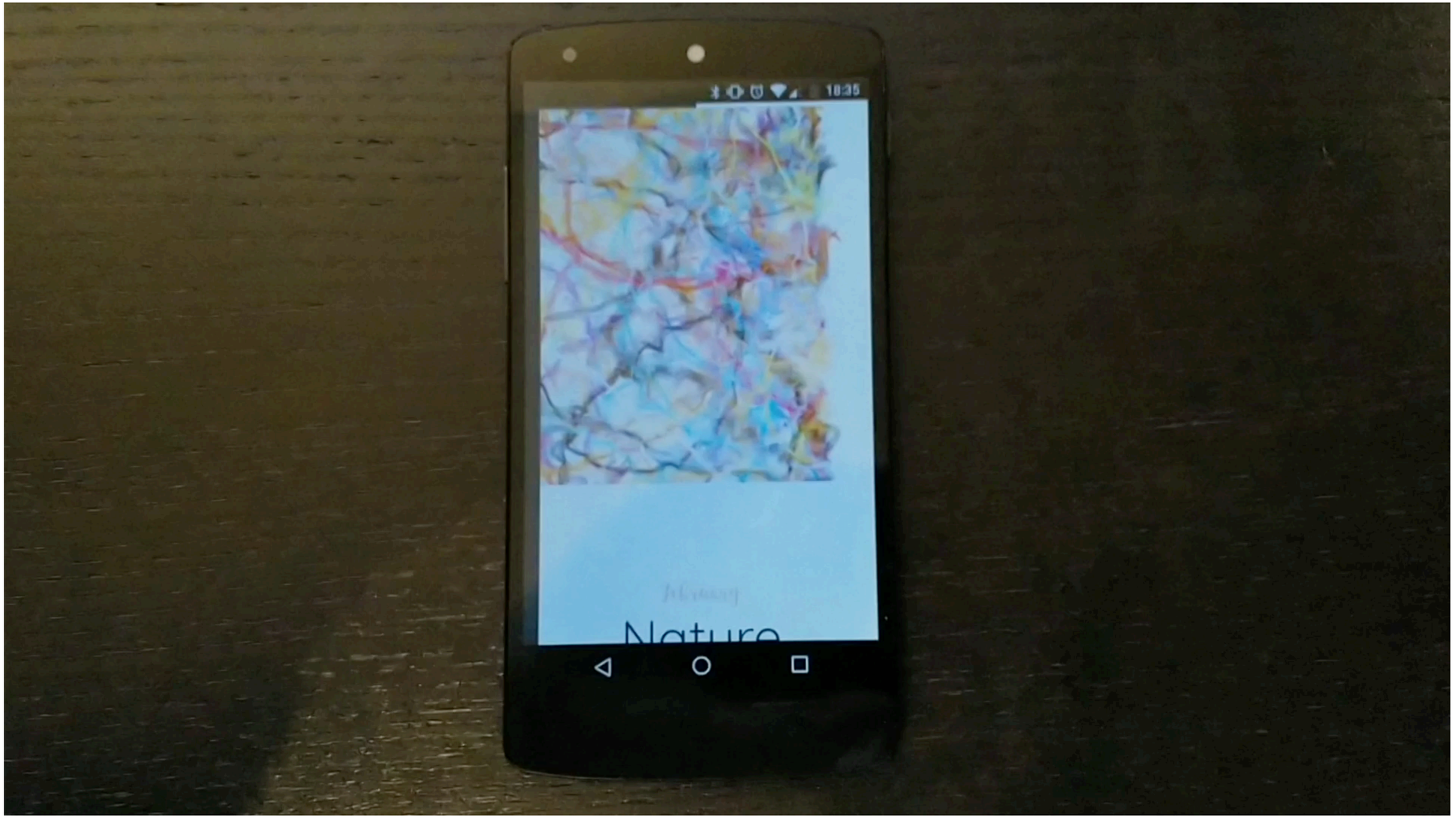
and when not

YAY

- When you want to show thousands of different data points (that don't change/move (too much))
  - Each SVG element creates a DOM element, which can severely slow down the browser when you get >> 1000 elements
  - For canvas only the JS calculation and drawing takes "time", the number of items you draw doesn't matter, it's "just a canvas"
- When speed of showing "something" to the user is important, also on mobile
  - Typically drawing with canvas works wonderfully fast in practically all browsers and devices

NAY

- If perfect crispness is key & you want to adjust the final result in Illustrator
  - If you zoom in, you'll start to see the separate pixels. They're not vectors | But there's a practical solution!
- If you're planning many interactions
  - There are no *mouseover*, *mouseout* like event listeners. There are other ways though, but not as easy as ".on"
- When you have (many different) animations and gradual changes
  - D3 makes changes of state/animations super easy. All of that is lost in canvas. You have to do more effort to "fake" an animation


10:35
Nature

When to use

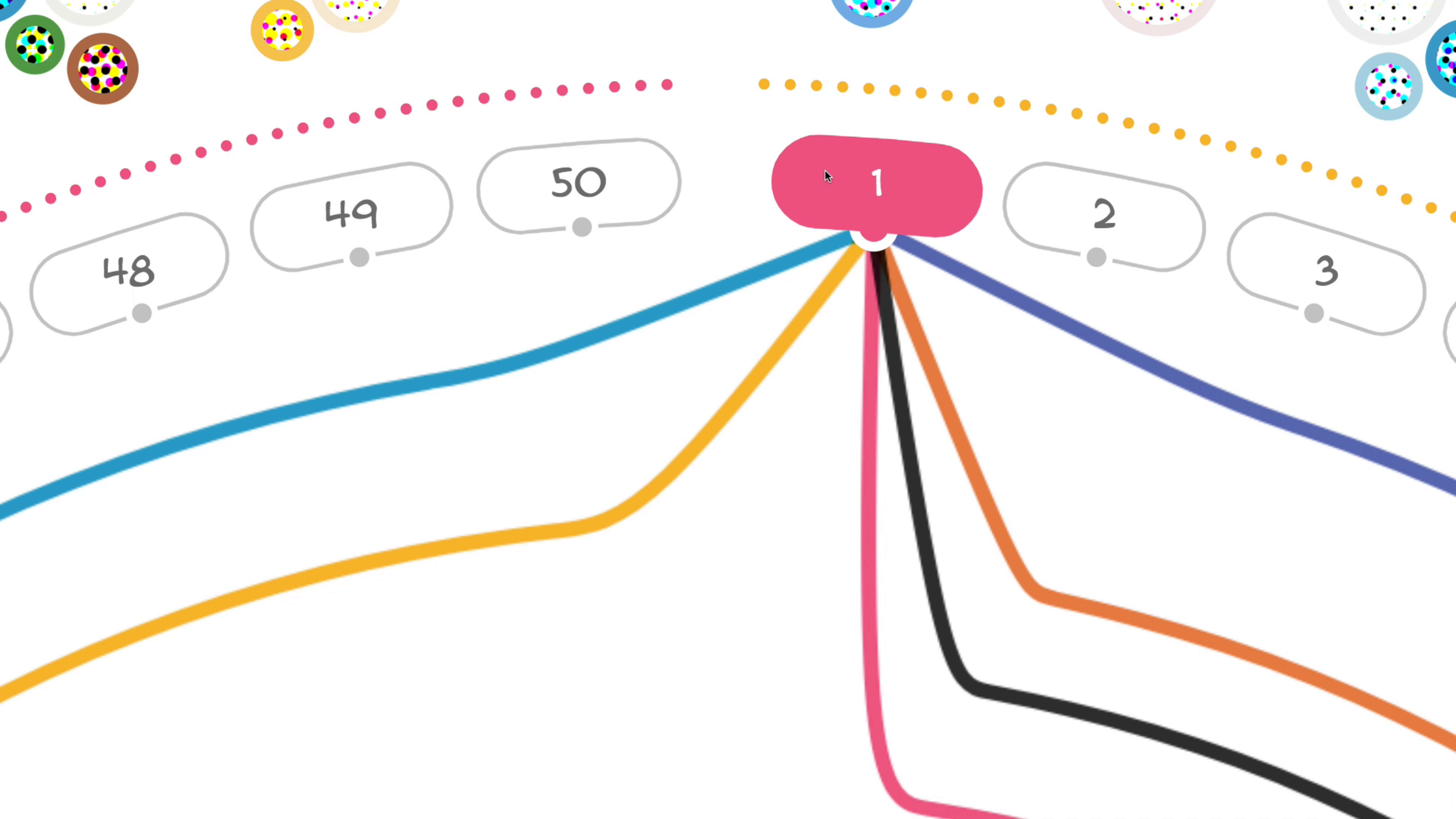
and when not

YAY

- When you want to show thousands of different data points (that don't change/move (too much))
  - Each SVG element creates a DOM element, which can severely slow down the browser when you get >> 1000 elements
  - For canvas only the JS calculation and drawing takes "time", the number of items you draw doesn't matter, it's "just a canvas"
- When speed of showing "something" to the user is important, also on mobile
  - Typically drawing with canvas works wonderfully fast in practically all browsers and devices

NAY

- If perfect crispness is key & you want to adjust the final result in Illustrator
  - If you zoom in, you'll start to see the separate pixels. They're not vectors | But there's a practical solution!
- If you're planning many interactions
  - There are no *mouseover*, *mouseout* like event listeners. There are other ways though, but not as easy as ".on"
- When you have (many different) animations and gradual changes
  - D3 makes changes of state/animations super easy. All of that is lost in canvas. You have to do more effort to "fake" an animation


48
49
50
1
2
3

When to use

and when not

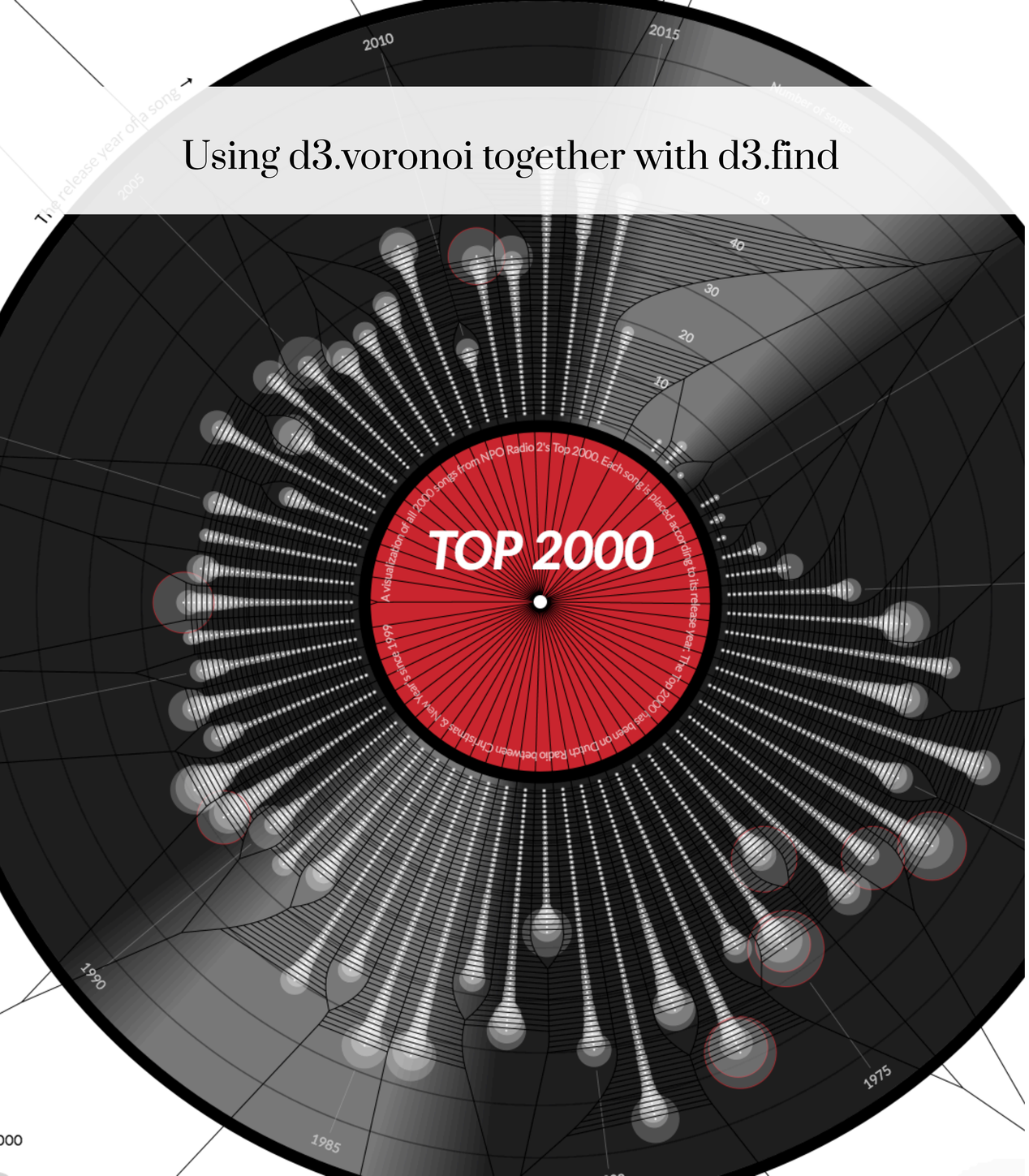
YAY

- When you want to show thousands of different data points (that don’t change/move (too much))
  - Each SVG element creates a DOM element, which can severely slow down the browser when you get >> 1000 elements
  - For canvas only the JS calculation and drawing takes “time”, the number of items you draw doesn’t matter, it’s “just a canvas”
- When speed of showing “something” to the user is important, also on mobile
  - Typically drawing with canvas works wonderfully fast in practically all browsers and devices

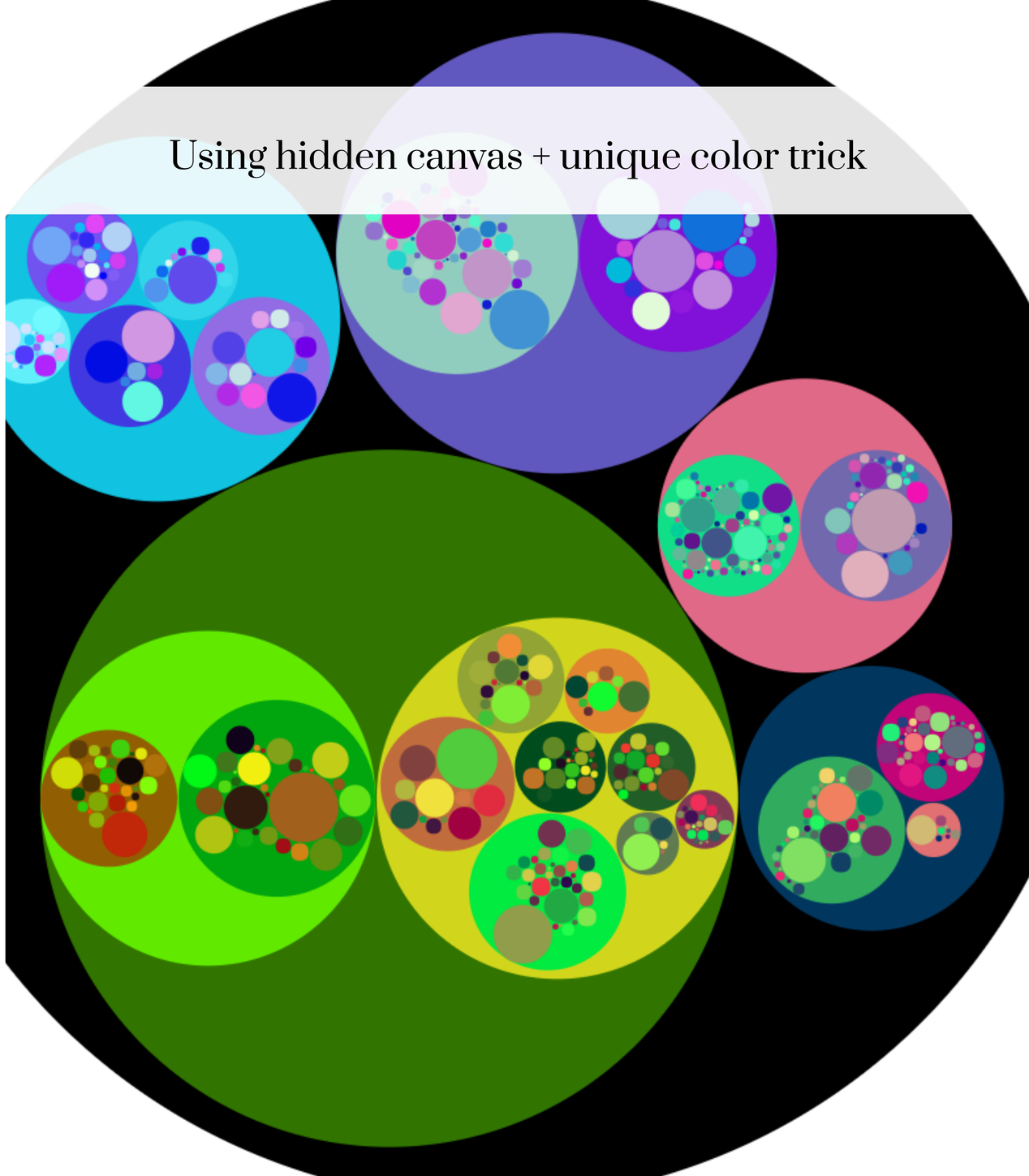
NAY

- If perfect crispness is key & you want to adjust the final result in Illustrator
  - If you zoom in, you’ll start to see the separate pixels. They’re not vectors | But there’s a practical solution!
- If you’re planning many interactions
  - There are no *mouseover*, *mouseout* like event listeners. There are other ways though, but not as easy as “.on”
- When you have (many different) animations and gradual changes
  - D3 makes changes of state/animations super easy. All of that is lost in canvas. You have to do more effort to “fake” an animation

Using d3.voronoi together with d3.find



Using hidden canvas + unique color trick



When to use

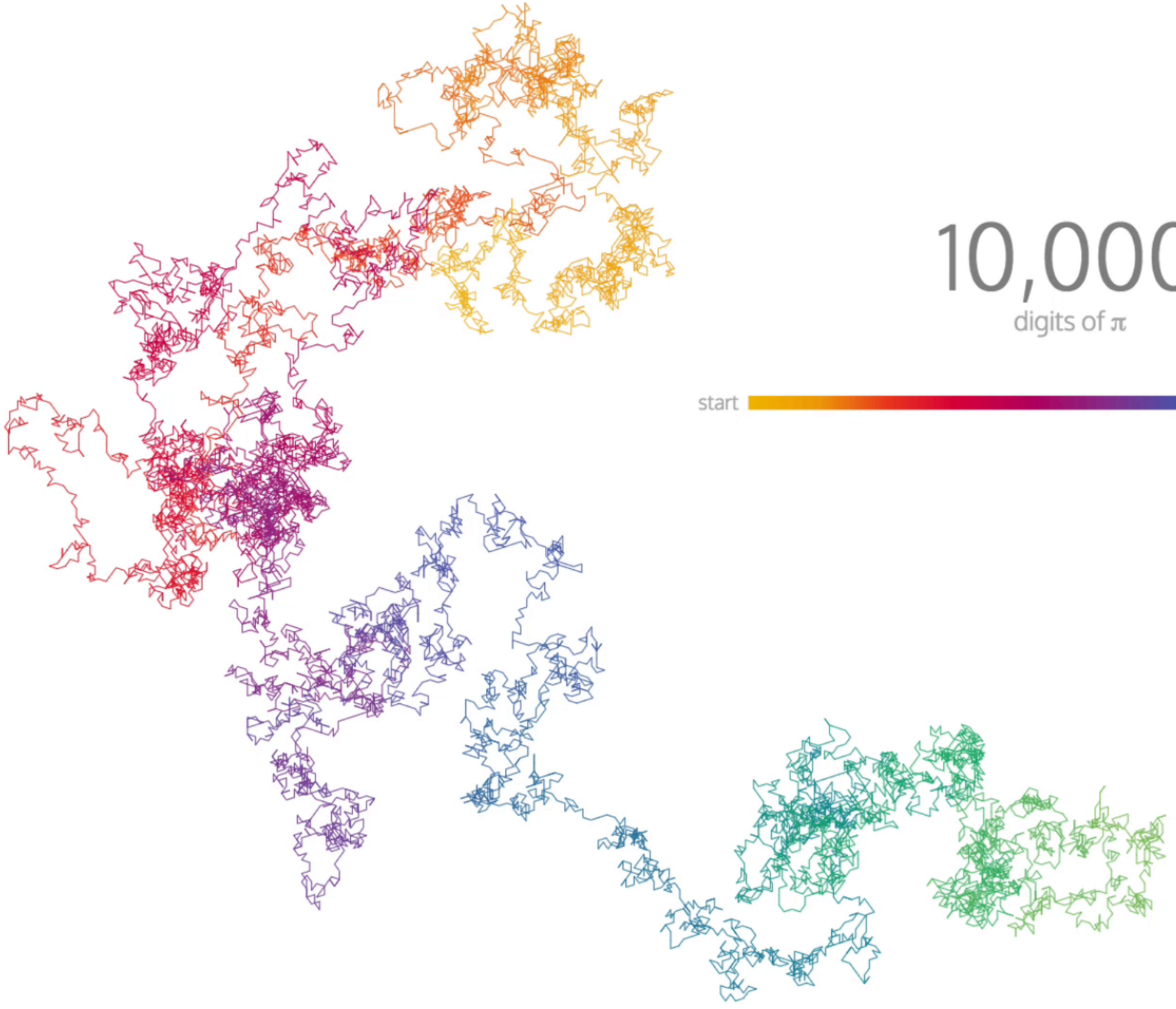
and when not

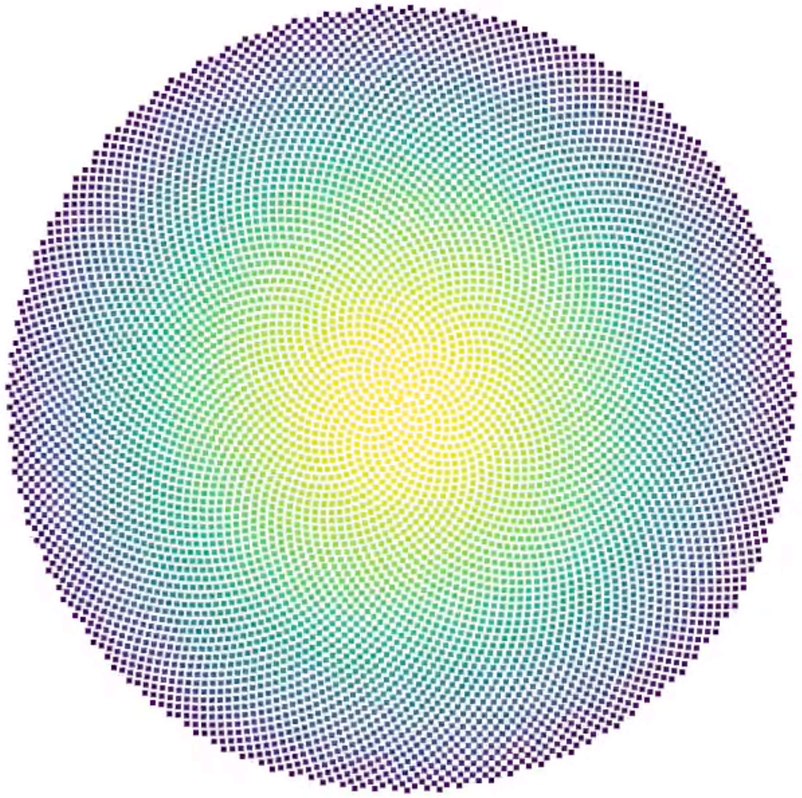
YAY

- When you want to show thousands of different data points (that don't change/move (too much))
  - Each SVG element creates a DOM element, which can severely slow down the browser when you get >> 1000 elements
  - For canvas only the JS calculation and drawing takes "time", the number of items you draw doesn't matter, it's "just a canvas"
- When speed of showing "something" to the user is important, also on mobile
  - Typically drawing with canvas works wonderfully fast in practically all browsers and devices

NAY

- If perfect crispness is key & you want to adjust the final result in Illustrator
  - If you zoom in, you'll start to see the separate pixels. They're not vectors | But there's a practical solution!
- If you're planning many interactions
  - There are no *mouseover*, *mouseout* like event listeners. There are other ways though, but not as easy as ".on"
- When you have (many different) animations and gradual changes
  - D3 makes changes of state/animations super easy. All of that is lost in canvas. You have to do more effort to "fake" an animation


10,000
digits of π
start
end



https://bocoup.com/blog/smoothly-animate-thousands-of-points-with-html5-canvas-and-d3

The basics

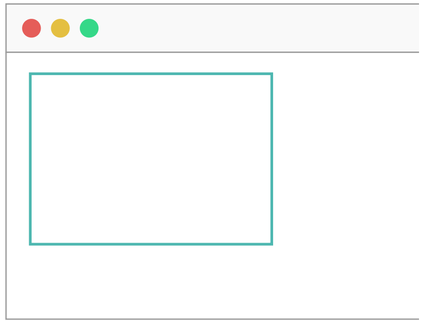
Creating a canvas

Select the body of your page and append the **canvas**

```
//Append a canvas element to the body
let canvas = d3.select("body")
    .append("canvas")
    .attr("width", 1000)
    .attr("height", 700)
```

To quickly check if the canvas is there, you can give it a color border (as if you were styling an `<img>` element)

```
//You can style the canvas like an <img>
canvas.style("border", "1px solid #4DB7B0")
```



It's the **context** that you can start "drawing" on

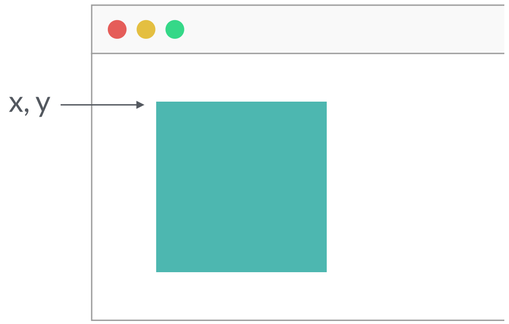
```
//Get the 2D context from the canvas element
let ctx = canvas.node().getContext("2d")
```

Draw a rectangle

First set the color to use for the **fill**

Define the **rectangle** and fill it with `.fillRect()`

```
//Define the color to be used when something is filled
ctx.fillStyle = “#4DB7B0"

//Draw a rectangle with x, y, width, height
//x & y define the position of the top-left corner
ctx.fillRect(10, 10, 100, 100) //x, y, width, height
```



Transparency can be created by using an `rgba()` color

Expert note | Set the overall transparency with `ctx.globalAlpha = 0.5`

```
//Draw another one that is a bit transparent
ctx.fillStyle = "rgba(228, 31, 104, 0.5)"
ctx.fillRect(50,50,150,150)
```

Instead of a fill, you can also **stroke** a rectangle

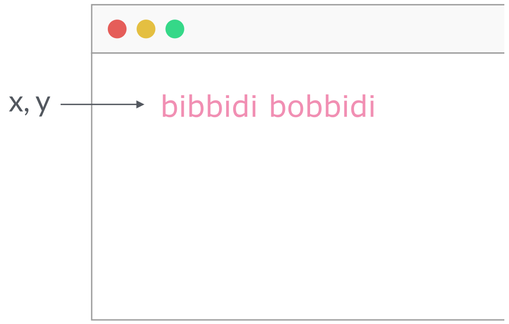
```
//Create a rectangle that is stroked
ctx.strokeStyle = "#e42068"
ctx.lineWidth = 3
ctx.strokeRect(30,80,100,100)
```

Draw text

Define the **font size** and font **family**

Draw the **text** at the x & y location with `.fillText()`

```
//Define the font size and font family
ctx.font = "20px Verdana"

//Draw the font with text, start-x & start-y
ctx.fillText("bibbidi bobbidi", 250, 100)
```



Expert point | Several font settings can be defined

As with the rectangle, you can also create a **stroked text**

```
//Adjust the font and create a stroked text
ctx.font = "italic bold 16px Verdana"
ctx.strokeText("boo", 250, 150)
```

However | It's better to always use a SVG to draw text

Crisper text, selectable, searchable (also for Google)

Draw a circle

A circle is seen as a **path**

There are a wide variety of path types available

Any path starts with `.beginPath()`

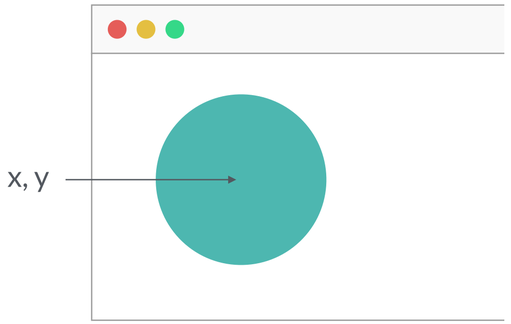
A circle is created by drawing the **outline** of a complete circle and filling it with a color

The **fill** of a path is done separately with `.fill()`

And a stroke with `.stroke()`

A path ends with `.closePath()`

```
//Adjust the fill style
ctx.fillStyle = "#4DB7B0"

//A circle is created with one of the "path" commands
//Start any path with beginPath()
ctx.beginPath()

//Draw the circle with: center-x, center-y, radius,
start-angle, end-angle
ctx.arc(500, 100, 50, 0, 2*Math.PI)

//Fill the path
ctx.fill()

//Close a path with closePath()
ctx.closePath()
```



Draw a line

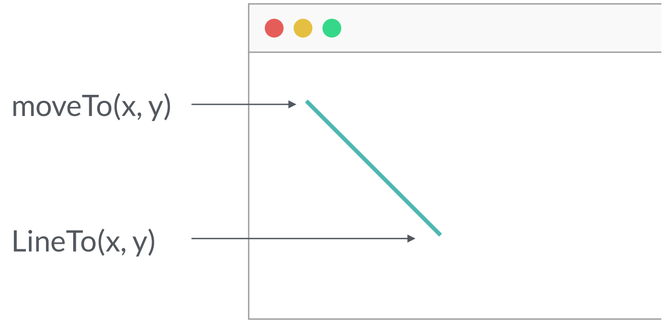
A line is the most basic **path**

Most paths begin with `.moveTo()`, where should the “pen” start to draw its path

The `.lineTo()` command draws a **straight line** from the previously declared point to the x & y within `.lineTo()`

Instead of fill, we stroke this path with `.stroke()`

```
ctx.beginPath()
//Start drawing a line from the point x, y
//In essence, move your pen to this point
ctx.moveTo(600, 50)
//Draw a line to this point with x, y
ctx.lineTo(700, 150)
//Stroke the path
ctx.stroke()
ctx.closePath()
```



Many styling that you can give a (SVG) line in CSS can also be created in canvas, although the wording is often different

Creating a rounded-edge-dashed line

```
//You can use many CSS stylings, e.g. stroke-linecap
ctx.lineCap = "round"
//or stroke-dasharray
ctx.setLineDash([5, 15])
ctx.moveTo(650, 50)
ctx.lineTo(750, 150)
ctx.stroke()
```

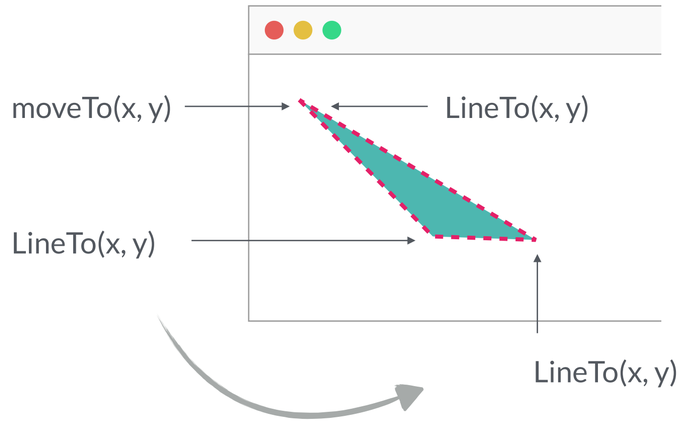
Combining paths

Combine paths to create more complex shapes

The next path step will always use the previous point as the starting point of the next step

Expert note | This isn't true for the `.arc()` path. To combine a circle into a complex path, use `.arcTo()`

```
//Drawing multiple line segments creates a path
ctx.beginPath()
ctx.moveTo(700,50)
ctx.lineTo(800,150)
ctx.lineTo(900,150)
ctx.lineTo(700,50)
//You can fill it, or stroke it, or both
ctx.fill()
ctx.stroke()
ctx.closePath()
```



Draw a curved line

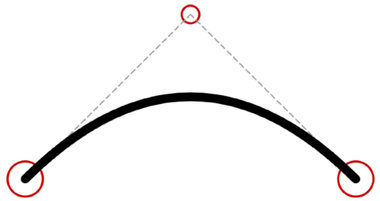
The Quadratic and Cubic Bézier curve path commands give you the option to create complex curved lines

Besides a start and end point, you also define 1 (quadratic) or 2 (cubic) ‘anchor’ points that define the curve of the line

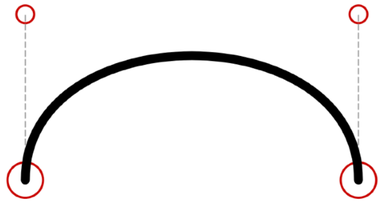
These anchor points are not visible

```
ctx.beginPath()
//Start drawing a line from the point x, y
ctx.moveTo(50, 100)
//Create a quadratic curve, with 1 anchor point
//anchor-x, anchor-y, end-x, end-y
ctx.quadraticCurveTo(150, 0, 250, 100)
//Stroke the path
ctx.stroke()
ctx.closePath()
```

Quadratic Bézier Curve



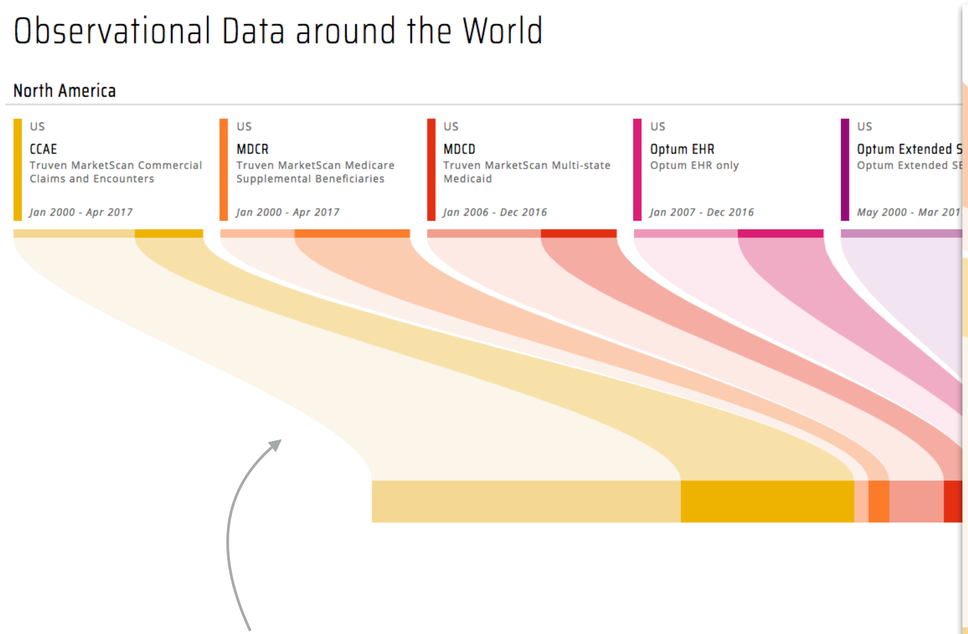
Cubic Bézier Curve



```
ctx.beginPath()
//Start drawing a line from the point x, y
ctx.moveTo(300, 100)
//Create a cubic curve, with 2 anchor points
//start-anchor-x, start-anchor-y, end-anchor-x, end-
anchor-y, end-x, end-y
ctx.bezierCurveTo(400, 200, 400, 0, 500, 100)
//Stroke the path
ctx.stroke()
ctx.closePath()
```

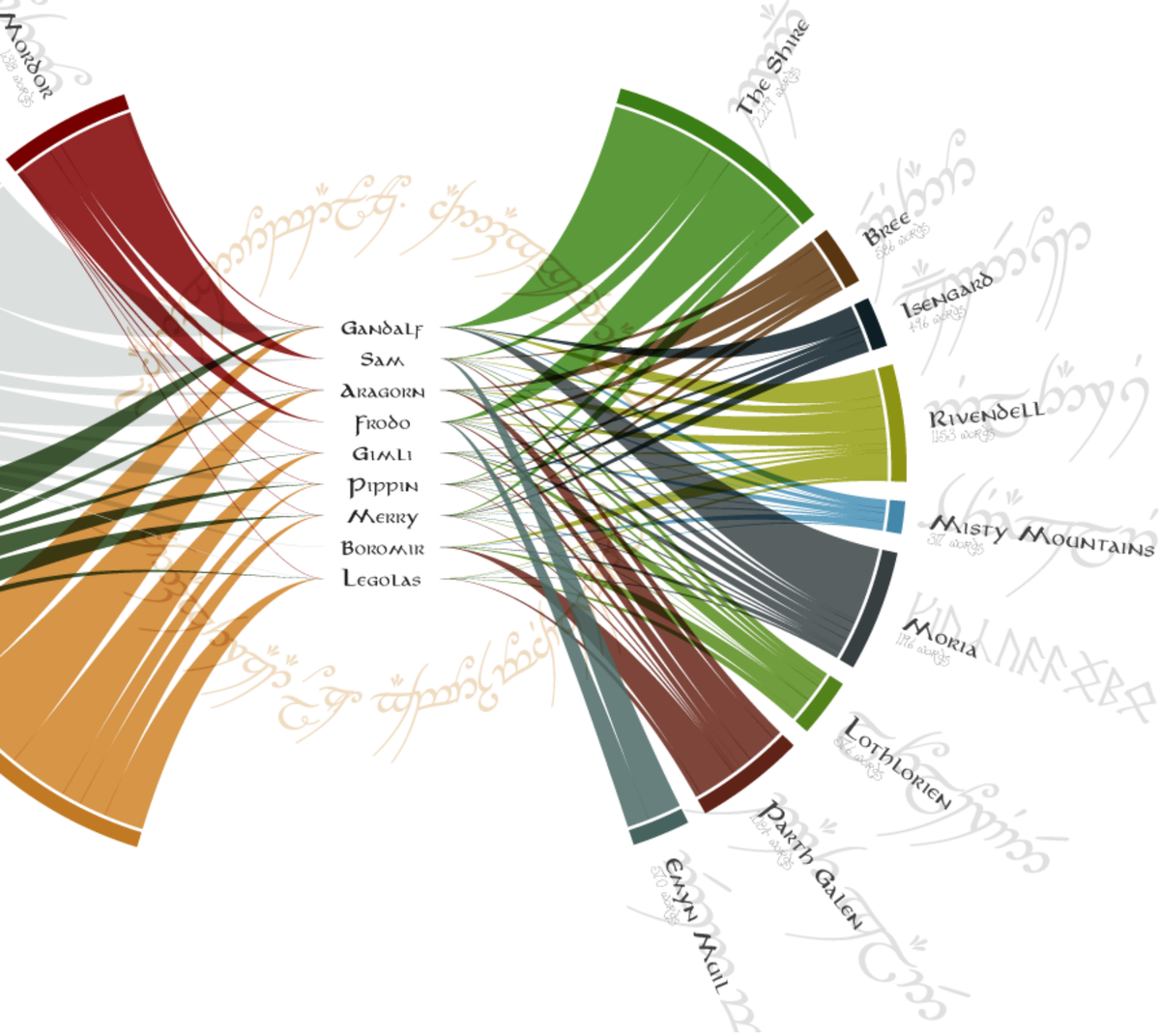
DataViz

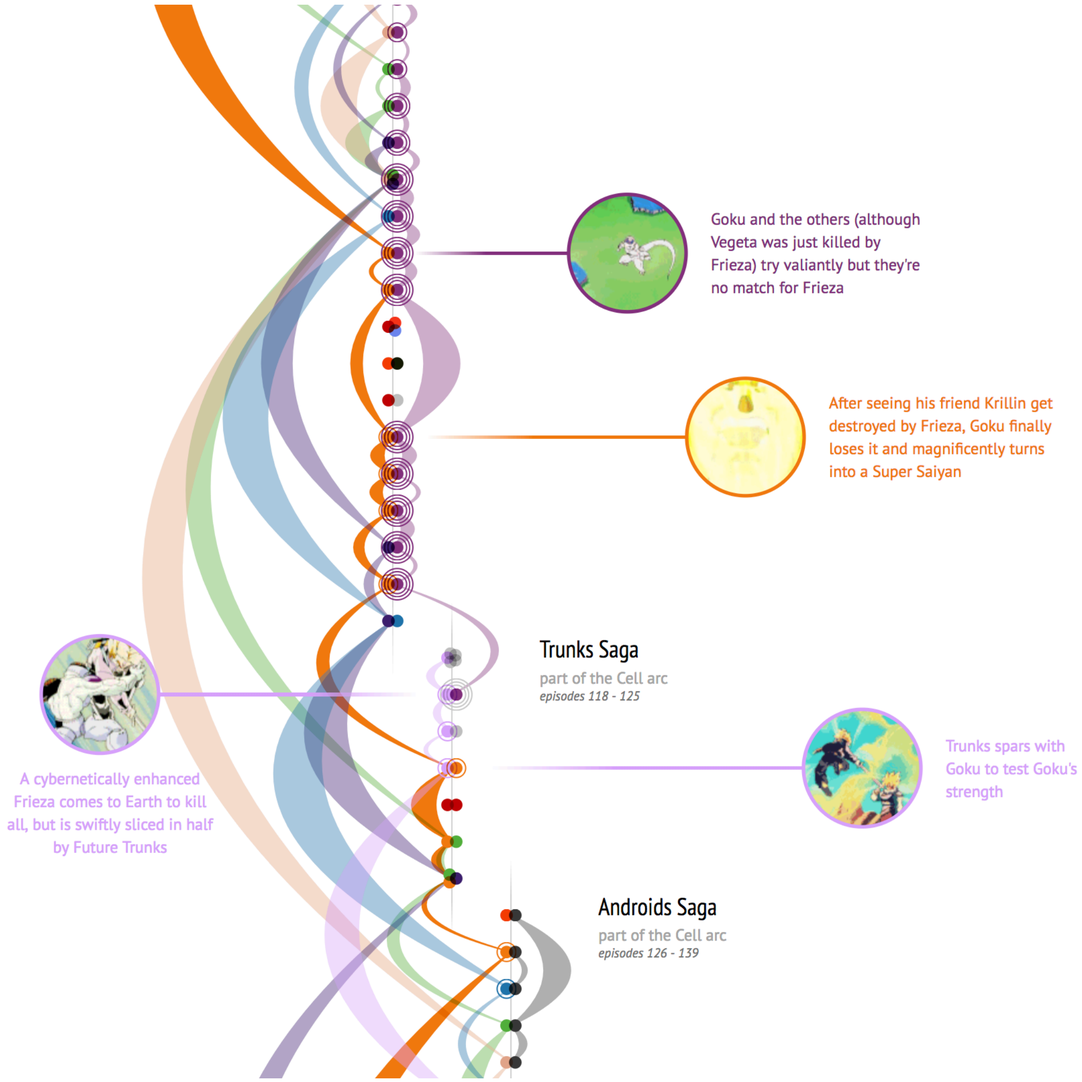
Complex shapes in dataviz



Data based complex shapes:
line - cubic curve - line - cubic curve

Complex shapes in dataviz





Creating a map out of circles

We have a dataset with 50,000 x & y points of the landmass across the Earth, and their "**amount of greenness**" as measured from space

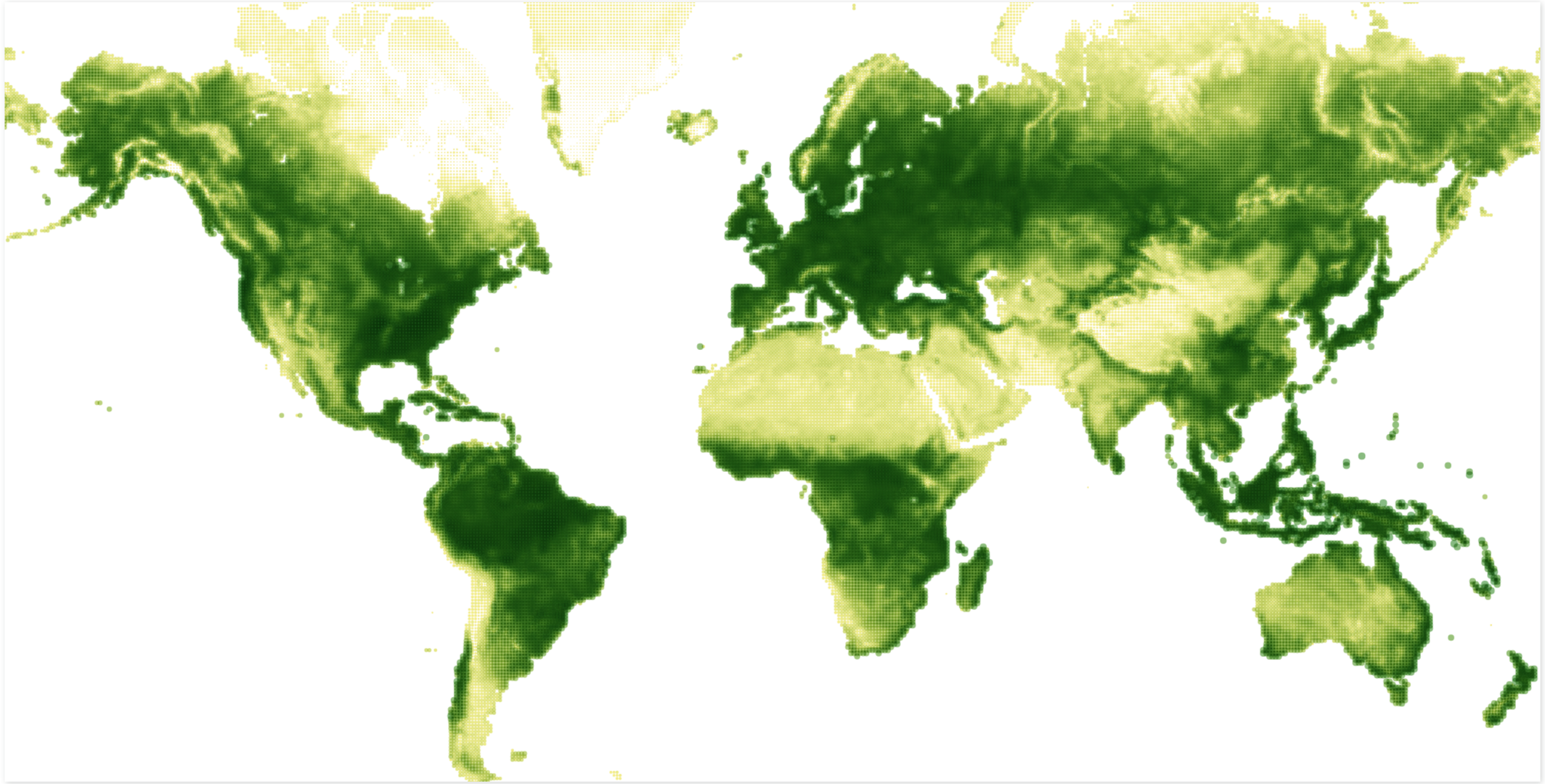
```
{"x":1,"y":210,"greenness":0.28},
{"x":1,"y":211,"greenness":0.29},
{"x":1,"y":212,"greenness":0.31},
...
```

Loop over each datapoint, and draw a circle at the right x & location with a color, opacity and radius defined by the greenness

Expert note | For a nicer effect, set the overall color blending to multiply, using `.globalCompositeOperation`

```
//Color blend mode of multiply for a nicer effect
ctx.globalCompositeOperation = "multiply"

d3.json("greenness-map-data.json", function(error, map_data) {

    if (error) throw error

    /Draw the map by looping over each circle
    map_data.forEach(function (d, i) {
        //The fill color, opacity and radius of each circle
        //are all based on the greenness value
        ctx.fillStyle = greenColor(d.greenness)
        ctx.globalAlpha = opacityScale(d.greenness)
        let r = radiusScale(d.greenness)

        //Draw the circle and fill it
        ctx.beginPath()
        ctx.arc(xScale(d.x), yScale(d.y), r, 0, 2*Math.PI)
        ctx.closePath()
        ctx.fill()
    })//forEach

})//d3.json
```

Creating a map out of circles



Putting it all together

EXERCISE

Creating a scatterplot

Turn this scatterplot about movie ratings vs budgets, which is created with d3.js into one using canvas for the visual plotting

You can still use d3's function for all none drawing things, such as the existing d3.scaleLinear, d3.select, etc.

Casual

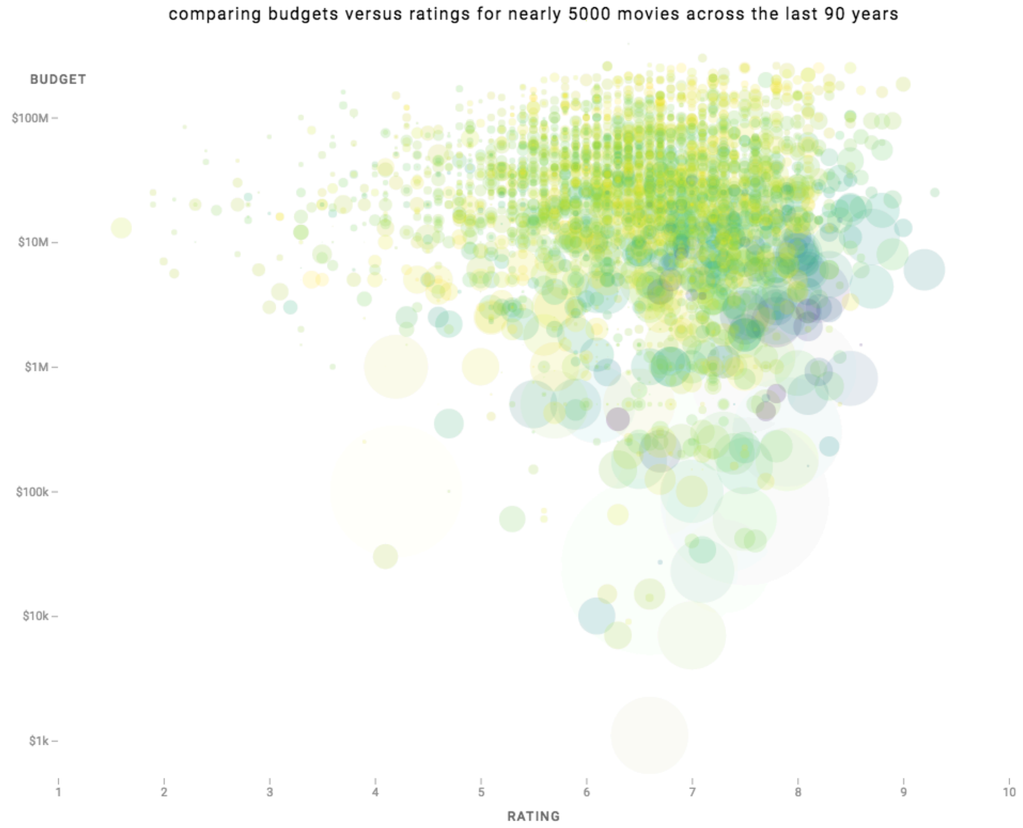
- Create a **canvas** & define the **context**
- **Loop** over all the datapoints (using a for loop, or .forEach()) and draw a **circle** for each datapoint with the correct: x & y location, radius & color

Medium

- Keep the existing **SVG** there, for the axes and **text** only, with the canvas underneath for the circles

Extreme

- level 1 | On a hover, instantaneously **change the color** of the hovered circle to black & full opacity (and back to the original color when moving out of the circle)
- level 2 | On a hover, **linearly increase** the hovered circle's **size** to 1.5 times its original radius and **opacity** of 1 in 1000ms (and back again to normal size on a mouseout)
- level 3 | The same as level 2, but use an **easing function** for the radius animation



- Each circle is a movie | ±3700 in total
- x axis: (IMDB) rating
- y axis: budget (in US dollars)
- radius: profit ratio (revenue / budget)
- color: release year (from dark purple in 1930 tot light yellow today)

Appendix

Creating a retina-crisp canvas

On Macs with retina screens, canvas looks a bit blurry. I therefore actually always set up my canvas with this “hack”

It creates a canvas that twice as wide/high, but squishes it down to the intended size

Expert note | You can use methods such as `window.devicePixelRatio` to see if you’re dealing with a retina screen and only do this method if it is

```
//Define the width and height
let w = 1000,
    h = 700

//Create the canvas
let canvas = d3.select("body")
    .append("canvas")
    //Make the canvas twice as wide & high
    .attr("width", 2 * w)
    .attr("height", 2 * h)
    //But fit this within an area that is
    //the intended width and height
    .style("width", w + "px")
    .style("height", h + "px")

//Finally scale the canvas, so you can keep using your
original coordinate system that has a width of "w" and
height of "h"
let ctx = canvas.node().getContext("2d")
ctx.scale(2,2)
```

Animations

Creating movement

Creating **animations** on canvas follows the same idea as film: creating "**frames**" in which the new state is slightly updated

Continuously call a function that updates your visual

With `.clearRect()` you can clear away the previous "drawing" on the canvas. Then draw the visual again

E.g. draw a rectangle at a slightly different location

Use `requestAnimationFrame()` to call the same function again, once it's done

```
let change = 5 //Amount of change on each loop
let x = 0 //Starting x position

function loop() {
    //Clear the area between the x, y, width & height
    //The x & y define the top left of the rectangle
    //This clears the entire canvas
    ctx.clearRect(0, 0, width, height)

    //Add the "change" onto the previous x location
    x = x + change
    //If the edges are reached, flip the direction
    if(x <= 0 || x >= width-50) change = change * -1

    //Draw the rectangle
    ctx.fillRect(x, 100, 50, 50)

    //Call the next loop
    requestAnimationFrame(loop)
}//function loop

//Start the 1st time
requestAnimationFrame(loop)
```

Creating “eased” movement

Mimicking non-linear movement (e.g. ease-in-out) can be done by using d3’s easing and interpolators

Instead of adding a change to x, add a change to a “progress” variable (running from 0 to 1 -> 0% to 100%)

Feed that variable to d3’s ease function to get back an “eased” number between 0 and 1

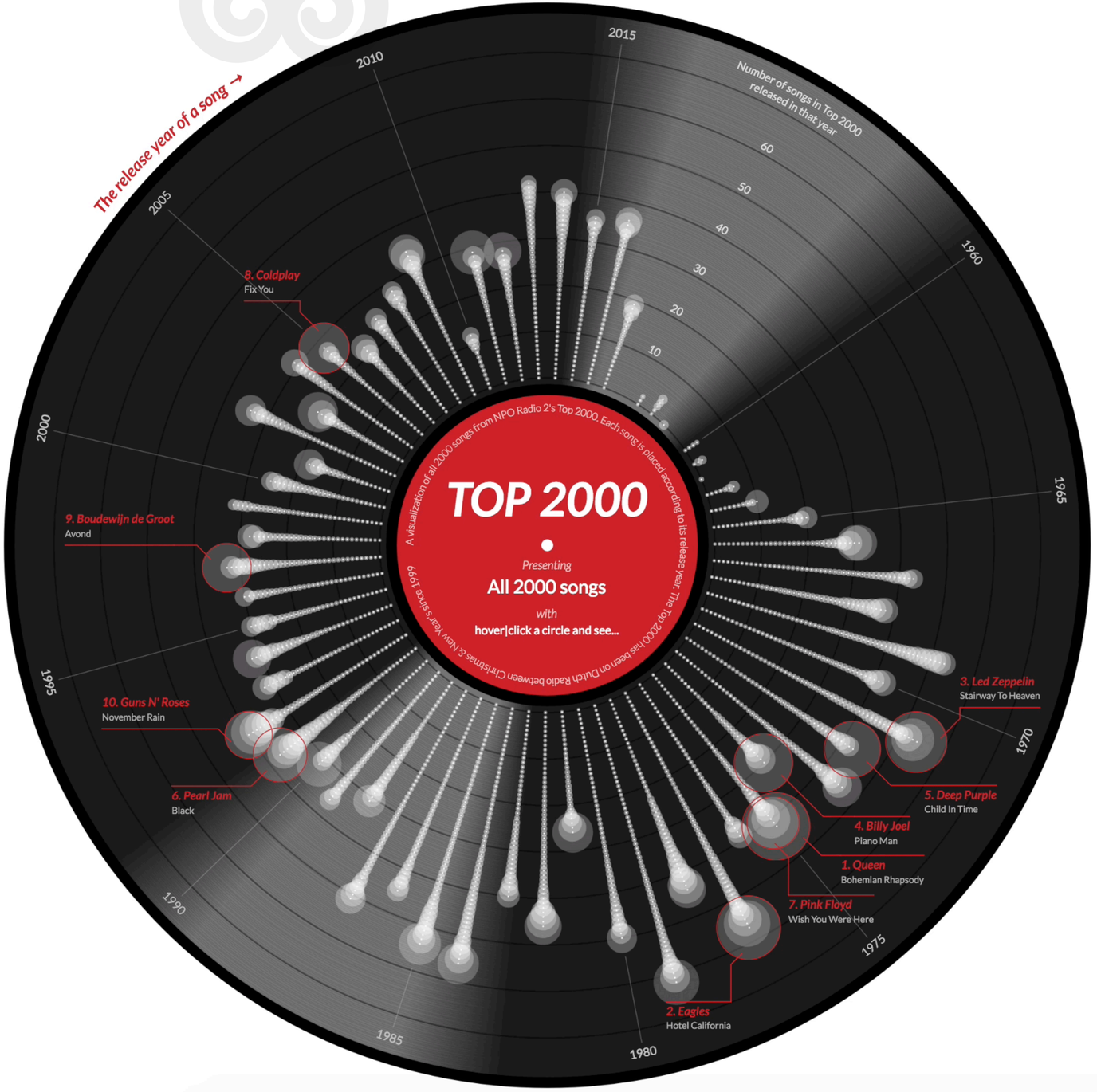
Used the “eased” value to calculate the new x position

```
let change = 0.01
const total = width - 50 //The movement length
let progress = 0 //Will track progress along the total
const ease = d3.easeSinInOut

function loop() {
    //Clear the area between the x, y, width & height
    ctx.clearRect(0, 0, width, height)

    //Add a little onto the change
    progress = progress + change
    //If the edges are reached, reverse the change
    if(progress <= 0 || progress >= 1) change *= -1

    //Calculate the x new position
    x = total * ease(progress)

    //Draw the rectangle
    ctx.fillRect(x, 100, 50, 50)
    requestAnimationFrame(loop)
}//function loop

requestAnimationFrame(loop)
```

Interactions

OPTION 1 | VORONOI

Apart from the annotations & text (which are SVG) this visual is **completely** made in **canvas**



The hover interaction is “captured” by an **underlying voronoi grid** and the `.find()` function of `d3.voronoi()`

Capture a hover with voronoi

Initialize a `d3.voronoi()`

Once the data is read in, calculate the **voronoi grid** and save

Attach a `mousemove` event listener to the canvas

`mouseover` and `mouseout` don't really make sense here

Get the **mouse location** with `d3.mouse(this)`

Use the voronoi's `.find()` function to see if the mouse is near any of the points

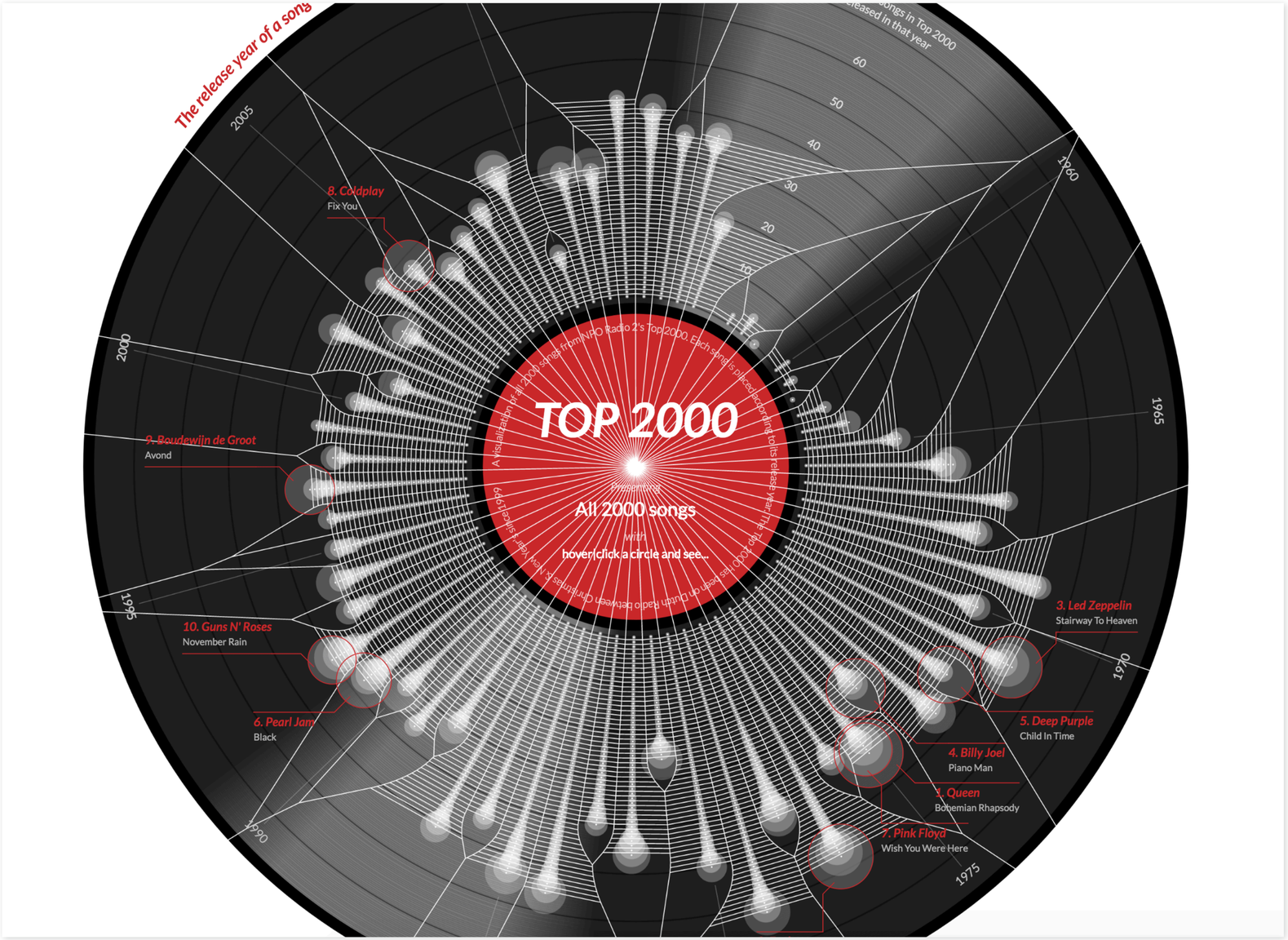
If found is defined, run your "hover" function

```
//Initialize the voronoi within the rectangle of
//width x height
const voronoi = d3.voronoi()
    .x(function(d) { return d.x })
    .y(function(d) { return d.y })
    .extent([[0,0], [width, height]])

//Once the data is read in, create the voronoi grid
diagram = voronoi(data)

//When a mouse is moved over the canvas, see if it's
//near enough to a "song"
canvas.on("mousemove", function() {

    //Get the x and y location of the mouse
    //over the canvas
    let m = d3.mouse(this)

    //Find the nearest song to the mouse
    //within a max distance of 50 pixels
    let found = diagram.find(m[0], m[1], 50)

    //If found is defined, run a function to do stuff
    if (found) { show_highlight_artist(found.data)
    } else { reset_chart() }

})//on mousemove
```

Capture a hover with voronoi

The **voronoi grid** made visible

although not taking into account the max radius



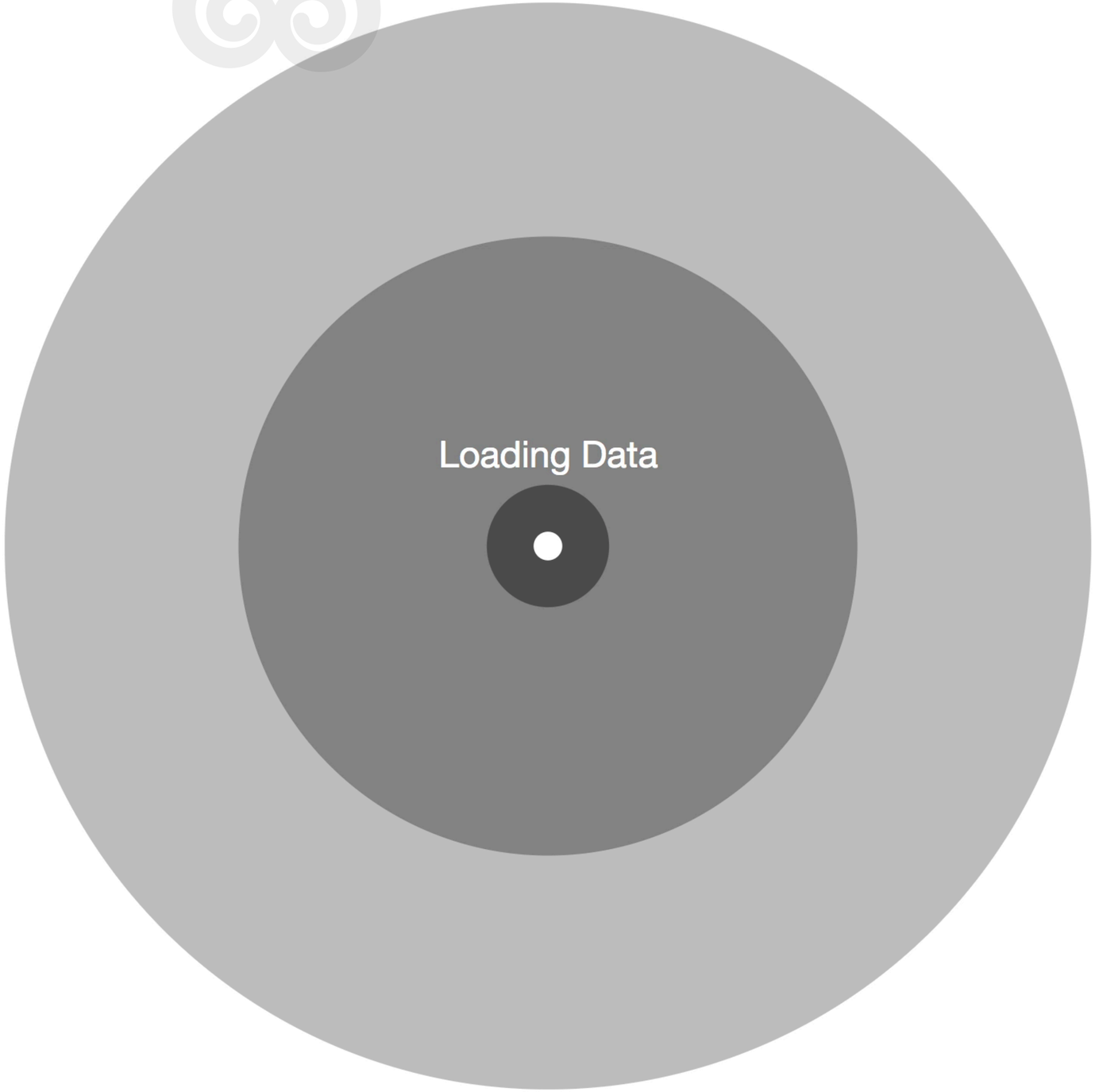
also see: https://bl.ocks.org/Fil/1b7ddbcd71454d685d1259781968aefc

Interactions

OPTION 2 | HIDDEN LAYER

This visual about labour division (& age division within that) is **completely** made in

SVG



It’s excruciatingly slow ;____;

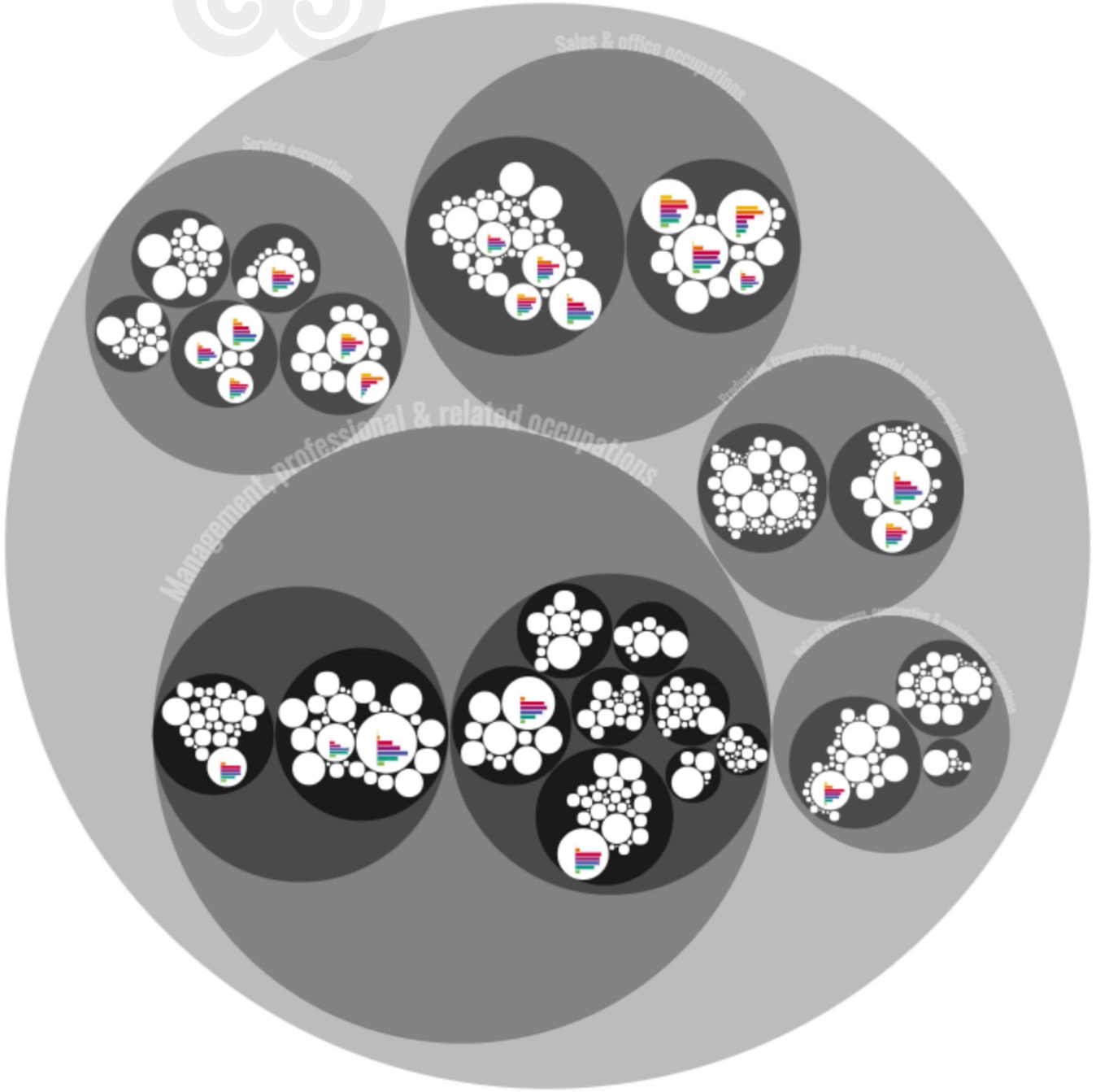
This visual about labour division (& age division within that) is **completely** made in

canvas

Practically instantaneous loading!

But fuzzy on a Mac, because I didn't do the scaling trick

The click interaction is "captured" by an **underlying hidden canvas** in which each clickable circle has its own unique rgb color



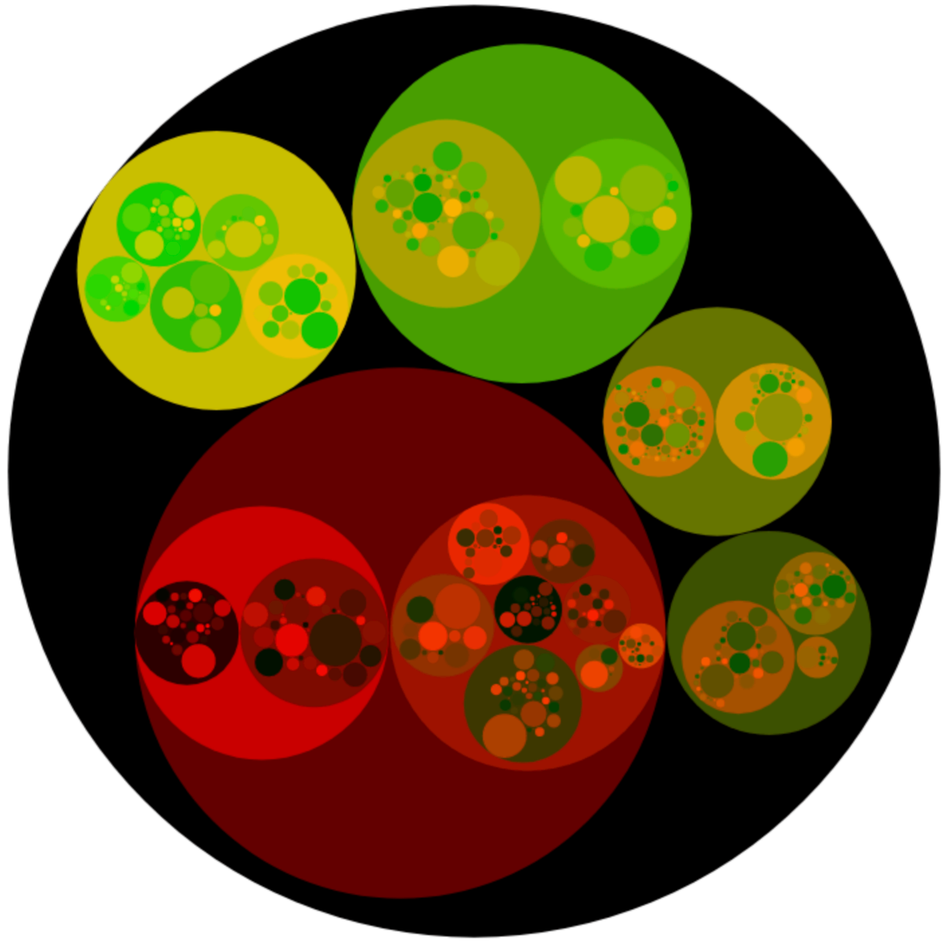
Capture a click with a “hidden” canvas

Create a **second canvas**, positioned absolutely to lie exactly on top of the “normal” canvas. Set its display to none to **hide it**

Give each clickable element a unique rgb color

E.g. rgb(100,100,100), rgb(100,100,101), rgb(100,100,102), etc.

Map the unique rgb color to the data. When the canvas is clicked, request the rgb color of the clicked circle and request the data that is connected to that rgb color



Based on “Needles, Haystacks, and the Canvas API” by Yannick Assogba

Capture a click with a “hidden” canvas

Create a **second canvas**, positioned absolutely to lie exactly on top of the “normal” canvas. Set its display to none to **hide it**

Give each clickable element a unique rgb color

E.g. rgb(100,100,100), rgb(100,100,101), rgb(100,100,102), etc.

Map the unique rgb color to the data. When the canvas is clicked, request the rgb color of the clicked circle and request the data that is connected to that rgb color

```
d3.select("#canvas").on("click", click_to_zoom)

//Function to run if a user clicks on the canvas
function click_to_zoom(e) {

    //Figure out where the mouse click occurred
    let mouseX = e.offsetX
    let mouseY = e.offsetY

    //Return the clicked pixel's color
    let col = hidden_ctx.getImageData(mouseX, mouseY,
1, 1).data
    //Turn into an rgb string
    let colString = "rgb(" + col[0] + "," + col[1] +
","+ col[2] + “)"
    //Find that rgb string in the color–circle mapping
    let circle = colToCircle[colString]

    //If there was a circle clicked on, zoom into this
    if(circle) {
        //Perform the zoom
        zoomToCanvas(circle)
    }//if

}//function click_to_zoom
```

Based on “Needles, Haystacks, and the Canvas API” by Yannick Assogba

Solutions

Creating a scatterplot

Full links to all the levels and their solutions

Casual

- SVG version | http://blockbuilder.org/nbremer/0ed7cfdca6487b8cda79af3aaedf2ec5
- Solution | https://bl.ocks.org/nbremer/6177a48b75a47c745707efae5e60d6f4

Medium

- SVG version | http://blockbuilder.org/nbremer/aae35b358c7450b70cc12b27e48c6196
- Solution | https://bl.ocks.org/nbremer/9ea9967adfe7849f9fef22eea1b884de

Extreme

- SVG version | http://blockbuilder.org/nbremer/7eda7f35215d6a5eb2a485f645844676
- Solution to level 1 | https://bl.ocks.org/nbremer/9a2fdead4297d2747df3d60aa48f64f1

Visual Cinnamon