APPRENEZ d3.js

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#d3.js

Table des matières

À propos ... 1
Chapitre 1: Démarrer avec d3.js ... 2
Remarques ... 2
Versions ... 2
Examples ... 2
Installation ... 3
Téléchargement direct du script ... 3
MNP ... 3
CDN ... 3
GITHUB ... 3
Diagramme à barres simple ... 3
index.html ... 3
chart.js ... 4
Bonjour le monde! ... 5
Qu'est ce que D3? Documents basés sur les données. ... 5
Simple D3 Chart: Bonjour tout le monde! ... 7
Chapitre 2: Approches pour créer des graphiques réactifs d3.js ... 10
Syntaxe ... 10
Examples ... 10
Utiliser le bootstrap ... 10
index.html ... 10
chart.js ... 10
Chapitre 3: Cartes SVG utilisant D3 js ... 12
Examples ... 12
Utilisation de D3 js pour créer des éléments SVG ... 12
Chapitre 4: Concepts de base SVG utilisés dans la visualisation D3.js ... 13
Examples ... 13
Système de coordonnées ... 13
le Élément ... 13
le Élément ... 14

le Élément 14
Ajouter correctement un élément SVG 15
SVG: l'ordre de dessin 17
Chapitre 5: Envoi d'événements avec d3.dispatch 20
Syntaxe 20
Remarques 20
Examples 20
usage simple 20
Chapitre 6: modèle de mise à jour 21
Syntaxe 21
Examples 21
Mise à jour des données: exemple de base des sélections d'entrée, de mise à jour et de sor 21
Fusion de sélections 23
Chapitre 7: Projections D3 26
Examples 26
Projections Mercator 26
Projections Albers 31
Les propriétés générales 31
Choisir Parallels 34
Centrage et Rotation 34
Paramètres par défaut 38
Résumé 38
Projections Equidistantes Azimutales 38
Les propriétés générales: 38
Centrage et Rotation: 40
Chapitre 8: Rendre robuste, réactif et réutilisable (r3) pour d3 43
Introduction 43
Examples 43
Scatter Plot 43
Qu'est-ce qui fait un graphique? 43
Installer 44

Configuration 44
Fonctions d'aide 44
index.html 45
Faire notre nuage de points 45
make_margins (dans make_margins.js) 45
make_buttons (dans make_buttons.js) 46
make_title (dans make_title.js) 46
make_axes (dans make_axes.js) 46
Enfin notre nuage de points 46
Tableau des boîtes et moustaches 47
Diagramme à bandes 47
Chapitre 9: Sélections 48
Syntaxe 48
Remarques 48
Examples 48
Sélection de base et modifications 48
Différents sélecteurs 49
Sélection limitée des données simples 49
Le rôle des espaces réservés dans les sélections "enter" 49
Utiliser "this" avec une fonction de flèche 52
La fonction flèche 53
Les deuxième et troisième arguments combinés 53
Chapitre 10: Sur les événements 55
Syntaxe 55
Remarques 55
Examples 55
Joindre des événements de base sur des sélections 55
Supprimer un écouteur d'événement 56
Chapitre 11: Utiliser D3 avec d'autres frameworks 57
Examples 57
Composant D3.js avec ReactJS 57

d3_react.html 57

d3_react.js 57

D3js avec angulaire 59

Diagramme D3.js avec Angular v1 61

Chapitre 12: Utiliser D3 avec JSON et CSV 63

Syntaxe 63

Examples 63

Chargement de données à partir de fichiers CSV 63

Un ou deux paramètres dans le rappel - gestion des erreurs dans d3.request () 64

Crédits 67

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: d3-js

It is an unofficial and free d3.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official d3.js.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec d3.js

Remarques

D3.js est une bibliothèque JavaScript permettant de manipuler des documents à partir de données. D3 vous aide à donner vie aux données en utilisant HTML, SVG et CSS. L'accent mis par D3 sur les normes Web vous offre toutes les fonctionnalités des navigateurs modernes, sans vous lier à une infrastructure propriétaire, en combinant des composants de visualisation puissants et une approche de la manipulation DOM basée sur les données.

Le site officiel: https://d3js.org/ De nombreux exemples ici: http://bl.ocks.org/mbostock

Versions

Version	Date de sortie
v1.0.0	2011-02-11
2.0	2011-08-23
3.0 "Baja"	2012-12-21
v4.0.0	2016-06-28
v4.1.1	2016-07-11
v4.2.1	2016-08-03
v4.2.2	2016-08-16
v4.2.3	2016-09-13
v4.2.4	2016-09-19
v4.2.5	2016-09-20
v4.2.6	2016-09-22
v4.2.7	2016-10-11
v4.2.8	2016-10-20
v4.3.0	2016-10-27

Examples

Installation

Il existe de nombreuses manières de télécharger et d'utiliser D3.

Téléchargement direct du script

1. Télécharger et extraire d3.zip
2. Copiez le dossier résultant dans lequel vous conserverez les dépendances de votre projet.
3. Référence d3.js (pour le développement) ou d3.min.js (pour la production) dans votre code HTML: `<script type="text/javascript" src="scripts/d3/d3.js"></script>`

MNP

1. Initialisez NPM dans votre projet si vous ne l'avez pas déjà fait: `npm init`
2. NPM installe D3: `npm install --save d3`
3. Référence d3.js (pour le développement) ou d3.min.js (pour la production) dans votre code HTML: `<script type="text/javascript" src="node_modules/d3/build/d3.js"></script>`

CDN

1. Référence d3.js (pour le développement) ou d3.min.js (pour la production) dans votre code HTML: `<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>`

GITHUB

1. Obtenez n'importe quelle version de d3.js (pour le développement) ou d3.min.js (pour la production) à partir de Github: `<script type="text/javascript" src="https://raw.githubusercontent.com/d3/d3/v3.5.16/d3.js"></script>`

Pour lier directement à la dernière version, copiez cet extrait:

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

Diagramme à barres simple

index.html

```
<!doctype html>
<html>
  <head>
    <title>D3 Sample</title>
  </head>
  <body>
    <!-- This will serve as a container for our chart. This does not have to be a div, and can
in fact, just be the body if you want. -->
    <div id="my-chart"></div>

    <!-- Include d3.js from a CDN. -->
```

```
    <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>

    <!-- Include our script that will make our bar chart. -->
    <script type="text/javascript" src="chart.js"></script>
  </body>
</html>
```

chart.js

```
// Sample dataset. In a real application, you will probably get this data from another source
such as AJAX.
var dataset = [5, 10, 15, 20, 25]

// Sizing variables for our chart. These are saved as variables as they will be used in
calculations.
var chartWidth = 300
var chartHeight = 100
var padding = 5

// We want our our bars to take up the full height of the chart, so, we will apply a scaling
factor to the height of every bar.
var heightScalingFactor = chartHeight / getMax(dataset)

// Here we are creating the SVG that will be our chart.
var svg = d3
  .select('#my-chart')           // I'm starting off by selecting the container.
    .append('svg')               // Appending an SVG element to that container.
    .attr('width', chartWidth)   // Setting the width of the SVG.
    .attr('height', chartHeight) // And setting the height of the SVG.

// The next step is to create the rectangles that will make up the bars in our bar chart.
svg
  .selectAll('rect')                                          // I'm selecting all of the
rectangles in the SVG (note that at this point, there actually aren't any, but we'll be
creating them in a couple of steps).
  .data(dataset)                                              // Then I'm mapping the dataset
to those rectangles.
  .enter()                                                    // This step is important in
that it allows us to dynamically create the rectangle elements that we selected previously.
    .append('rect')                                           // For each element in the
dataset, append a new rectangle.
      .attr('x', function (value, index) {                    // Set the X position of the
rectangle by taking the index of the current item we are creating, multiplying it by the
calculated width of each bar, and adding a padding value so we can see some space between
bars.
          return (index * (chartWidth / dataset.length)) + padding
        })
      .attr('y', function (value, index) {                    // Set the rectangle by
subtracting the scaled height from the height of the chart (this has to be done becuase SVG
coordinates start with 0,0 at their top left corner).
        return chartHeight - (value * heightScalingFactor)
      })
      .attr('width', (chartWidth / dataset.length) - padding) // The width is dynamically
calculated to have an even distribution of bars that take up the entire width of the chart.
      .attr('height', function (value, index) {               // The height is simply the
value of the item in the dataset multiplied by the height scaling factor.
        return value * heightScalingFactor
      })
```

```
      .attr('fill', 'pink')                              // Sets the color of the bars.

/**
 *  Gets the maximum value in a collection of numbers.
 */
function getMax(collection) {
  var max = 0

  collection.forEach(function (element) {
    max = element > max ? element : max
  })

  return max
}
```

Exemple de code disponible sur https://github.com/dcsinnovationlabs/D3-Bar-Chart-Example

Démo disponible sur https://dcsinnovationlabs.github.io/D3-Bar-Chart-Example/

Bonjour le monde!

Créez un fichier `.html` contenant cet extrait:

```
<!DOCTYPE html>
<meta charset="utf-8">
<body>
<script src="//d3js.org/d3.v4.min.js"></script>
<script>

d3.select("body").append("span")
    .text("Hello, world!");

</script>
```

Voir cet extrait en action sur ce JSFiddle .

Qu'est ce que D3? Documents basés sur les données.

Nous sommes tellement habitués au nom **d3.js** qu'il est possible d'oublier que D3 est en fait **DDD (D** ata- **D** Riven **D** ocuments). Et c'est ce que D3 fait bien, une approche pilotée par les données de la manipulation des DOM (Document Object Model): D3 lie les données aux éléments DOM et manipule ces éléments en fonction des données limitées.

Voyons une fonctionnalité très basique de D3 dans cet exemple. Ici, nous n’ajouterons aucun élément SVG. Au lieu de cela, nous utiliserons un SVG déjà présent sur la page, quelque chose comme ceci:

```
<svg width="400" height="400">
    <circle cx="50" cy="50" r="10"></circle>
    <circle cx="150" cy="50" r="10"></circle>
    <circle cx="210" cy="320" r="10"></circle>
    <circle cx="210" cy="30" r="10"></circle>
    <circle cx="180" cy="200" r="10"></circle>
</svg>
```

C'est un SVG assez basique, avec 5 cercles. À l'heure actuelle, ces cercles ne sont liés à aucune donnée. Vérifions cette dernière allégation:

Dans notre code, nous écrivons:

```
var svg = d3.select("svg");
var circles = svg.selectAll("circle");
console.log(circles.nodes());
```

Ici, `d3.select("svg")` renvoie un objet d3 contenant la `<svg width="400" height="400"></svg>` et toutes les balises enfants, les `<circle>` s. Notez que si plusieurs balises `svg` existent sur la page, seule la première est sélectionnée. Si vous ne le souhaitez pas, vous pouvez également sélectionner par tag id, comme `d3.select("#my-svg")` . L'objet d3 a des propriétés et des méthodes intégrées que nous utiliserons beaucoup plus tard.

`svg.selectAll("circle")` crée un objet à partir de tous les éléments `<circle></circle>` depuis la `<svg>` . Il peut effectuer une recherche parmi plusieurs couches, peu importe si les balises sont des enfants directs.

`circles.nodes()` renvoie les balises de cercle avec toutes leurs propriétés.

Si nous regardons la console et choisissons le premier cercle, nous verrons quelque chose comme ceci:

```
▼ [circle, circle, circle, circle, circle]
  ▼ 0: circle
    ▶ attributes: NamedNodeMap
      baseURI: "https://fiddle.jshell.net/_display/"
      childElementCount: 0
    ▶ childNodes: NodeList[0]
    ▶ children: HTMLCollection[0]
    ▶ classList: DOMTokenList[0]
    ▶ className: SVGAnimatedString
      clientHeight: 0
      clientLeft: 0
      clientTop: 0
      clientWidth: 0
```

Tout d'abord, nous avons des `attributes` , puis `childNodes` , puis des `children` , etc. mais pas de données.

Lions quelques données

Mais que se passe-t-il si nous lions des *données* à ces éléments DOM?

Dans notre code, il y a une fonction qui crée un objet avec deux propriétés, `x` et `y` , avec des valeurs numériques (cet objet est dans un tableau, vérifiez le violon ci-dessous). Si nous lions ces données aux cercles ...

```
circles.data(data);
```

Voici ce que nous allons voir si nous inspectons la console:

```
▼ [circle, circle, circle, circle, circle]
  ▼ 0: circle
    ▼ __data__: Object
        x: 2.9844424316350615
        y: 9.929545206204867
      ▶ __proto__: Object
    ▶ attributes: NamedNodeMap
      baseURI: "https://fiddle.jshell.net/_display/"
      childElementCount: 0
    ▶ childNodes: NodeList[0]
    ▶ children: HTMLCollection[0]
    ▶ classList: DOMTokenList[0]
    ▶ className: SVGAnimatedString
```

Nous avons quelque chose de nouveau juste avant les `attributes` ! Quelque chose nommé `__data__` ... et regardez: les valeurs de `x` et `y` sont là!

Nous pouvons, par exemple, modifier la position des cercles en fonction de ces données. Regardez ce violon .

C'est ce que D3 fait de mieux: lier les données aux éléments DOM et manipuler ces éléments DOM en fonction des données limitées.

Simple D3 Chart: Bonjour tout le monde!

Collez ce code dans un fichier HTML vide et exécutez-le dans votre navigateur.

```
<!DOCTYPE html>

<body>

<script src="https://d3js.org/d3.v4.js"></script>    <!-- This downloads d3 library -->

<script>
//This code will visualize a data set as a simple scatter chart using d3. I omit axes for
simplicity.
var data = [        //This is the data we want to visualize.
                    //In reality it usually comes from a file or database.
  {x: 10,    y: 10},
  {x: 10,    y: 20},
  {x: 10,    y: 30},
  {x: 10,    y: 40},
  {x: 10,    y: 50},
  {x: 10,    y: 80},
  {x: 10,    y: 90},
  {x: 10,    y: 100},
  {x: 10,    y: 110},
  {x: 20,    y: 30},
  {x: 20,    y: 120},
  {x: 30,    y: 10},
  {x: 30,    y: 20},
  {x: 30,    y: 30},
  {x: 30,    y: 40},
  {x: 30,    y: 50},
  {x: 30,    y: 80},
  {x: 30,    y: 90},
  {x: 30,    y: 100},
  {x: 30,    y: 110},
```

```
    {x: 40,    y: 120},
    {x: 50,    y: 10},
    {x: 50,    y: 20},
    {x: 50,    y: 30},
    {x: 50,    y: 40},
    {x: 50,    y: 50},
    {x: 50,    y: 80},
    {x: 50,    y: 90},
    {x: 50,    y: 100},
    {x: 50,    y: 110},
    {x: 60,    y: 10},
    {x: 60,    y: 30},
    {x: 60,    y: 50},
    {x: 70,    y: 10},
    {x: 70,    y: 30},
    {x: 70,    y: 50},
    {x: 70,    y: 90},
    {x: 70,    y: 100},
    {x: 70,    y: 110},
    {x: 80,    y: 80},
    {x: 80,    y: 120},
    {x: 90,    y: 10},
    {x: 90,    y: 20},
    {x: 90,    y: 30},
    {x: 90,    y: 40},
    {x: 90,    y: 50},
    {x: 90,    y: 80},
    {x: 90,    y: 120},
    {x: 100,    y: 50},
    {x: 100,    y: 90},
    {x: 100,    y: 100},
    {x: 100,    y: 110},
    {x: 110,    y: 50},
    {x: 120,    y: 80},
    {x: 120,    y: 90},
    {x: 120,    y: 100},
    {x: 120,    y: 110},
    {x: 120,    y: 120},
    {x: 130,    y: 10},
    {x: 130,    y: 20},
    {x: 130,    y: 30},
    {x: 130,    y: 40},
    {x: 130,    y: 50},
    {x: 130,    y: 80},
    {x: 130,    y: 100},
    {x: 140,    y: 50},
    {x: 140,    y: 80},
    {x: 140,    y: 100},
    {x: 140,    y: 110},
    {x: 150,    y: 50},
    {x: 150,    y: 90},
    {x: 150,    y: 120},
    {x: 170,    y: 20},
    {x: 170,    y: 30},
    {x: 170,    y: 40},
    {x: 170,    y: 80},
    {x: 170,    y: 90},
    {x: 170,    y: 100},
    {x: 170,    y: 110},
    {x: 170,    y: 120},
    {x: 180,    y: 10},
```

```
  {x: 180,    y: 50},
  {x: 180,    y: 120},
  {x: 190,    y: 10},
  {x: 190,    y: 50},
  {x: 190,    y: 120},
  {x: 200,    y: 20},
  {x: 200,    y: 30},
  {x: 200,    y: 40},
  {x: 210,    y: 80},
  {x: 210,    y: 90},
  {x: 210,    y: 100},
  {x: 210,    y: 110},
  {x: 210,    y: 120},
  {x: 220,    y: 80},
  {x: 220,    y: 120},
  {x: 230,    y: 80},
  {x: 230,    y: 120},
  {x: 240,    y: 90},
  {x: 240,    y: 100},
  {x: 240,    y: 110},
  {x: 270,    y: 70},
  {x: 270,    y: 80},
  {x: 270,    y: 90},
  {x: 270,    y: 100},
  {x: 270,    y: 120}
];

//The following code chains a bunch of methods. Method chaining is what makes d3 very simple
and concise.
d3.select("body").append("svg").selectAll()  //'d3' calls the d3 library
                                             //'.select' selects the object (in this case the
body of HTML)
                                             //'.append' adds SVG element to the body
                                             //'.selectAll()' selects all SVG elements
    .data(data)                              //'.data' gets the data from the variable 'data'
  .enter().append("circle")                  //'.enter' enters the data into the SVG
                                             //the data enter as circles with
'.append("circle")'
    .attr("r", 3)                            //'.attr' adds/alters atributes of SVG,
                                             //such as radius ("r"), making it 3 pixels
    .attr("cx", function(d) { return d.x; }) //coordinates "cx" (circles' x coordinates)
    .attr("cy", function(d) { return d.y; }) //coordinates "cy" (circles' y coordinates)
    .style("fill", "darkblue");              //'.style' changes CSS of the SVG
                                             //in this case, fills circles with "darkblue"
color

</script>
```

Voici un JSFiddle du graphique.

Vous pouvez également télécharger le fichier HTML déjà créé depuis GitHub .

La prochaine étape dans l'apprentissage de d3 peut être de suivre le tutoriel de Mike Bostock (le créateur de d3) pour créer un graphique à barres à partir de zéro .

Lire Démarrer avec d3.js en ligne: https://riptutorial.com/fr/d3-js/topic/876/demarrer-avec-d3-js

Chapitre 2: Approches pour créer des graphiques réactifs d3.js

Syntaxe

- var width = document.getElementById ('chartArea'). clientWidth;
- var height = width / 3.236;
- window.onresize = resizeFunctionCall;

Examples

Utiliser le bootstrap

Une approche souvent utilisée consiste à utiliser le cadre quadrillé de bootstrap afin de définir la zone dans laquelle le graphique existera. En utilisant ceci en conjonction avec la variable `clientWidth` et l'événement `window.onresize` , il est très facile de créer des SVG d3 réactifs. .

Créons d'abord une ligne et une colonne dans lesquelles notre graphique sera intégré.

index.html

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="col-xs-12 col-lg-6" id="chartArea">
      </div>
    </div>
  </div>
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>
<script src="chart.js"></script>
</body>
</html>
```

Cela va créer une colonne qui sera en plein écran sur un appareil mobile et l'autre moitié sur un grand écran.

chart.js

```
var width = document.getElementById('chartArea').clientWidth;
//this allows us to collect the width of the div where the SVG will go.
var height = width / 3.236;
```

```
//I like to use the golden rectangle ratio if they work for my charts.

var svg = d3.select('#chartArea').append('svg');
//We add our svg to the div area


//We will build a basic function to handle window resizing.
function resize() {
    width = document.getElementById('chartArea').clientWidth;
    height = width / 3.236;
    d3.select('#chartArea svg')
      .attr('width', width)
      .attr('height', height);
}

window.onresize = resize;
//Call our resize function if the window size is changed.
```

Ceci est un exemple extrêmement simplifié, mais couvre les concepts de base de la configuration des graphiques pour qu'ils soient réactifs. La fonction de redimensionnement devra appeler votre fonction de mise à jour principale pour redessiner tous les chemins, axes et formes, comme si les données sous-jacentes avaient été mises à jour. La plupart des utilisateurs de d3 concernés par les visualisations réactives sauront déjà comment créer leurs événements de mise à jour en fonctions faciles à appeler, comme indiqué dans la rubrique d'introduction et dans cette rubrique .

Lire Approches pour créer des graphiques réactifs d3.js en ligne: https://riptutorial.com/fr/d3-js/topic/4312/approches-pour-creer-des-graphiques-reactifs-d3-js

Chapitre 3: Cartes SVG utilisant D3 js

Examples

Utilisation de D3 js pour créer des éléments SVG

Bien que D3 ne soit pas spécifique au traitement des éléments SVG, il est largement utilisé pour créer et manipuler des visualisations de données complexes basées sur SVG. D3 fournit de nombreuses méthodes puissantes qui permettent de créer facilement diverses structures SVG géométriques.

Il est recommandé de comprendre d'abord les concepts de base des spécifications SVG, puis d'utiliser des exemples étendus de D3 js pour créer des visualisations.

D3 js exemples

Les bases de SVG

Lire Cartes SVG utilisant D3 js en ligne: https://riptutorial.com/fr/d3-js/topic/1538/cartes-svg-utilisant-d3-js

Chapitre 4: Concepts de base SVG utilisés dans la visualisation D3.js

Examples

Système de coordonnées

Dans un système de coordonnées mathématiques normal, le point x = 0, y = 0 est situé dans le coin inférieur gauche du graphique. Mais dans le système de coordonnées SVG, ce point (0,0) se trouve dans le coin supérieur gauche du «canvas», il est en quelque sorte similaire à CSS lorsque vous spécifiez la position absolue / fix et utilisez top et left pour contrôler le point exact de l'élément.

Il est essentiel de garder à l’esprit que plus les valeurs en SVG augmentent, plus les formes diminuent.

Disons que nous allons créer un diagramme de dispersion avec chaque point correspondant à la valeur de l'axe et à la valeur y. Pour mettre à l'échelle la valeur, vous devez définir le domaine et la plage comme suit:

```
d3.svg.scale()
  .range([0, height])
  .domain([0,max])
```

Cependant, si vous gardez les paramètres comme ceci, les points seront basés sur le bord horizontal supérieur au lieu de la ligne horizontale inférieure comme prévu.

La bonne chose à propos de d3 est que vous pouvez facilement changer cela par un simple ajustement dans le paramétrage du domaine:

```
d3.scale.linear()
  .range([height, 0])
  .domain([0, max])
```

Avec le code ci-dessus, le point zéro du domaine correspond à la hauteur du SVG, qui est la ligne inférieure du graphique dans les yeux du spectateur, tandis que la valeur maximale des données source correspondra au point zéro du SVG. système de coordonnées, dont la valeur maximale pour les téléspectateurs.

le Élément

`<rect>` représente un rectangle, en dehors des propriétés esthétiques telles que le trait et le remplissage, le rectangle doit être défini par son emplacement et sa taille.

Quant à l'emplacement, il est déterminé par les attributs x et y. L'emplacement est relatif au parent

du rectangle. Et si vous ne spécifiez pas l'attribut x ou y, la valeur par défaut sera 0 par rapport à l'élément parent.

Après avoir spécifié l'emplacement, ou plutôt le "point de départ" du rect, la prochaine chose à faire est de spécifier la taille, ce qui est essentiel si vous voulez réellement dessiner sth sur le canevas, c'est-à-dire si vous ne le faites pas. spécifiez les attributs de taille ou la valeur est définie sur 0, vous ne verrez rien sur le canevas.

Cas: graphique à barres

Continuez avec le premier scénario, les axes y, mais cette fois, essayons de dessiner un graphique à barres.

En supposant que le paramètre d'échelle y est identique, que l'axe y est correctement défini, la seule différence entre le diagramme de dispersion et ce diagramme à barres est que nous devons spécifier la largeur et la hauteur, en particulier la hauteur. Pour être plus précis, nous avons déjà le «point de départ», le reste consiste à utiliser des choses comme pour la hauteur:

```
.attr("height", function(d){
  return (height - yScale(d.value))
})
```

le Élément

`<svg>` élément `<svg>` est l'élément racine, ou le canevas tel que nous le dessinons.

Les éléments SVG peuvent être imbriqués les uns dans les autres et, de cette manière, les formes SVG peuvent être regroupées, tandis que toutes les formes imbriquées dans un élément seront positionnées par rapport à son élément englobant.

Une chose pourrait devoir être mentionnée, c'est que nous ne pouvons pas nous installer dans un autre, cela ne fonctionnera pas.

Cas: plusieurs graphiques

Par exemple, ce graphique à plusieurs donuts est composé de plusieurs éléments, qui contiennent respectivement un graphique en anneau. Cela peut être réalisé en utilisant l'élément, mais dans ce cas, nous ne voulons que mettre un tableau à côté l'un de l'autre, ce qui est plus pratique.

Une chose à garder à l'esprit est que nous ne pouvons pas utiliser l'attribut transform mais nous pouvons utiliser x, y pour la position.

le Élément

SVG ne prend actuellement pas en charge les sauts de ligne automatiques ou le retour à la ligne, c'est à ce moment-là qu'il faut venir en aide. élément positionne de nouvelles lignes de texte par rapport à la ligne de texte précédente. Et en utilisant dx ou dy dans chacun de ces segments, nous pouvons positionner le mot par rapport au mot qui le précède.

Cas: Annotation sur les axes

Par exemple, lorsque vous souhaitez ajouter une annotation sur y Ax:

```
svg.append("g")
  .attr("class", "y axis")
  .call(yAxis)
.append("text")
  .attr("transform", "rotate(-90)")
  .attr("y", 6)
  .attr("dy", ".71em")
  .style("text-anchor", "end")
  .text("Temperature (°F)");
```

Ajouter correctement un élément SVG

C'est une erreur relativement courante: vous avez créé un élément `rect` , par exemple dans un graphique à barres, et vous souhaitez ajouter un libellé (par exemple, la valeur de cette barre). Ainsi, en utilisant la même variable que vous avez utilisée pour ajouter le `rect` et définir sa position `x` et `y` , vous ajoutez votre élément de `text` . Très logique, vous pouvez penser. Mais cela ne fonctionnera pas.

Comment cette erreur se produit-elle?

Voyons un exemple concret, un code très basique pour créer un graphique à barres (violon ici):

```
var data = [210, 36, 322, 59, 123, 350, 290];

var width = 400, height = 300;

var svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height);

var bars = svg.selectAll(".myBars")
    .data(data)
    .enter()
    .append("rect");

bars.attr("x", 10)
    .attr("y", function(d,i){ return 10 + i*40})
    .attr("width", function(d){ return d})
    .attr("height", 30);
```

Ce qui nous donne ce résultat:

Mais vous voulez ajouter des éléments de texte, peut-être une simple valeur à chaque barre. Donc, vous faites ceci:

```
bars.append("text")
   .attr("x", 10)
   .attr("y", function(d,i){ return 10 + i*40})
   .text(function(d){ return d});
```

Et voilà: rien ne se passe! Si vous en doutez, voici le violon .

"Mais je vois l'étiquette!"

Si vous inspectez le SVG créé par ce dernier code, vous verrez ceci:

```
▼<rect x="10" y="250" width="290" height="30">
    <text x="10" y="250">290</text>
  </rect>
```

Et à ce stade, beaucoup de gens disent: "Mais je vois le tag texte, il est ajouté!". Oui, mais ça ne veut pas dire que ça va marcher. Vous pouvez ajouter *n'importe quoi* ! Voir ceci, par exemple:

```
svg.append("crazyTag");
```

Il vous donnera ce résultat:

```
<svg>
    <crazyTag></crazyTag>
</svg>
```

Mais vous ne vous attendez à aucun résultat simplement parce que le tag est là, n'est-ce pas?

Ajouter des éléments SVG dans le bon sens

Découvrez quels éléments SVG peuvent contenir des enfants, en lisant les spécifications . Dans notre dernier exemple, le code ne fonctionne pas car les éléments `rect` ne peuvent pas contenir

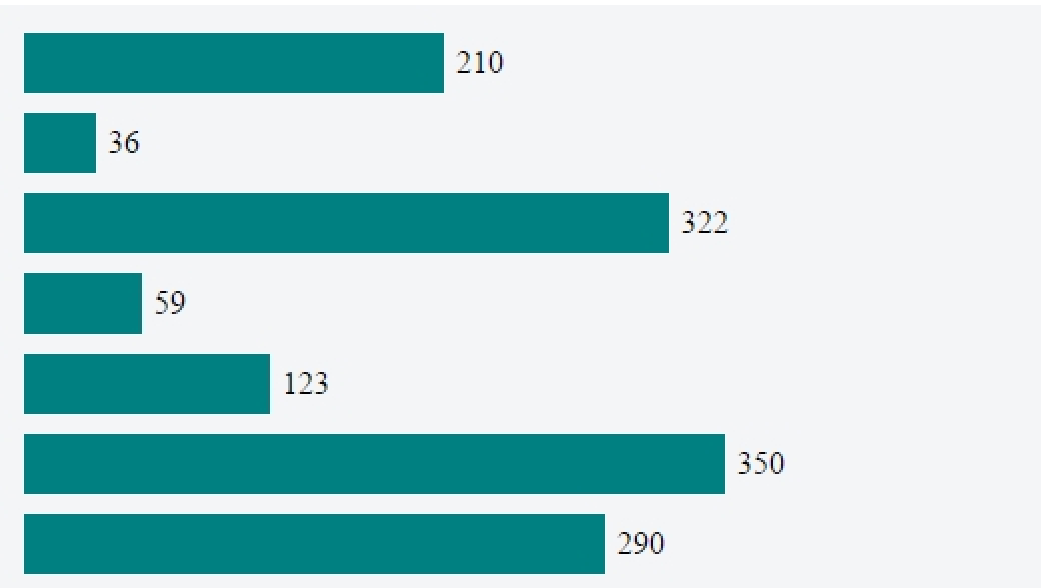
d’éléments de `text` . Alors, comment afficher nos textes?

Créez une autre variable et ajoutez le `text` au SVG:

```
var texts = svg.selectAll(".myTexts")
    .data(data)
    .enter()
    .append("text");

texts.attr("x", function(d){ return d + 16})
    .attr("y", function(d,i){ return 30 + i*40})
    .text(function(d){ return d});
```

Et c'est le résultat:



Et voici le violon

SVG: l'ordre de dessin

C'est quelque chose qui peut être frustrant: vous faites une visualisation en utilisant D3.js mais le rectangle que vous voulez en haut est caché derrière un autre rectangle, ou la ligne que vous avez prévu d'être derrière un cercle est en fait dessus. Vous essayez de résoudre ceci en utilisant le *z-index* dans votre CSS, mais cela ne fonctionne pas (dans SVG 1.1).

L'explication est simple: dans un SVG, l'ordre des éléments définit l'ordre du "tableau", et l'ordre du tableau définit qui va en haut.

> Les éléments d'un fragment de document SVG ont un ordre de dessin implicite, les premiers éléments du fragment de document SVG étant d'abord "peints". Les éléments suivants sont peints sur des éléments déjà peints.

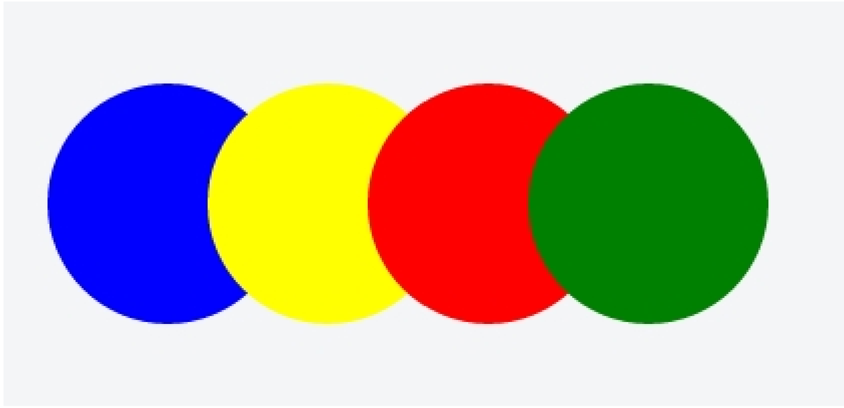
Donc, supposons que nous ayons ce SVG:

```
<svg width="400" height=200>
    <circle cy="100" cx="80" r="60" fill="blue"></circle>
```

```
    <circle cy="100" cx="160" r="60" fill="yellow"></circle>
    <circle cy="100" cx="240" r="60" fill="red"></circle>
    <circle cy="100" cx="320" r="60" fill="green" z-index="-1"></circle>
</svg>
```

Il a quatre cercles. Le cercle bleu est le premier "peint", donc il sera inférieur à tous les autres. Ensuite, nous avons le jaune, puis le rouge et enfin le vert. Le vert est le dernier et il sera en haut.

Voici à quoi ça ressemble:



Modification de l'ordre des éléments SVG avec D3

Alors, est-il possible de changer l'ordre des éléments? Puis-je faire le cercle rouge devant le cercle vert?

Oui. La première approche à prendre en compte est l'ordre des lignes dans votre code: dessinez d'abord les éléments de l'arrière-plan, puis dans le code les éléments du premier plan.

Mais nous pouvons changer de manière dynamique l'ordre des éléments, même après qu'ils ont été peints. Il existe plusieurs fonctions JavaScript simples que vous pouvez écrire pour cela, mais D3 a déjà 2 fonctionnalités intéressantes: `selection.raise()` et `selection.lower()` .

Selon l'API:

> selection.raise (): Réinsère chaque élément sélectionné, dans l'ordre, comme le dernier enfant de son parent. selection.lower (): Réinsère chaque élément sélectionné, dans l'ordre, comme premier enfant de son parent.

Donc, pour montrer comment manipuler l'ordre des éléments dans notre SVG précédent, voici un très petit code:

```
d3.selectAll("circle").on("mouseover", function(){
    d3.select(this).raise();
});
```

Qu'est ce que ça fait? Il sélectionne tous les cercles et, lorsque l'utilisateur survole un cercle, il sélectionne ce cercle et le place au premier plan. Très simple!

Et voici le JSFiddle avec le code live.

Lire Concepts de base SVG utilisés dans la visualisation D3.js en ligne:
https://riptutorial.com/fr/d3-js/topic/2537/concepts-de-base-svg-utilises-dans-la-visualisation-d3-js

Chapitre 5: Envoi d'événements avec d3.dispatch

Syntaxe

- d3. **dispatch** - crée un répartiteur d'événements personnalisé.
- envoi. **on** - enregistre ou désinscrit un écouteur d'événement.
- envoi. **copier** - créer une copie d'un répartiteur.
- envoi. **call** - envoie un événement aux auditeurs enregistrés.
- envoi. **apply** - envoie un événement aux auditeurs enregistrés.

Remarques

La répartition est un mécanisme pratique pour séparer les problèmes avec du code à couplage lâche: enregistrez les callbacks nommés et appelez-les ensuite avec des arguments arbitraires. Une variété de composants D3, tels que d3-request, utilisent ce mécanisme pour émettre des événements pour les écouteurs. Pensez à ceci comme EventEmitter de Node, sauf que chaque écouteur a un nom bien défini, il est donc facile de les supprimer ou de les remplacer.

Lectures connexes

- Dispatching Events par Mike Bostock
- d3.dispatch Documentation
- Événement d'expédition dans NPM

Examples

usage simple

```
var dispatch = d3.dispatch("statechange");

dispatch.on('statechange', function(e){ console.log(e) })

setTimeout(function(){dispatch.statechange('Hello, world!')}, 3000)
```

Lire Envoi d'événements avec d3.dispatch en ligne: https://riptutorial.com/fr/d3-js/topic/3399/envoi-d-evenements-avec-d3-dispatch

Chapitre 6: modèle de mise à jour

Syntaxe

- selection.enter ()
- selection.exit ()
- selection.merge ()

Examples

Mise à jour des données: exemple de base des sélections d'entrée, de mise à jour et de sortie

La création d'un graphique affichant un jeu de données statique est relativement simple. Par exemple, si nous avons ce tableau d'objets comme données:

```
var data = [
    {title: "A", value: 53},
    {title: "B", value: 12},
    {title: "C", value: 91},
    {title: "D", value: 24},
    {title: "E", value: 59}
];
```

Nous pouvons créer un graphique à barres où chaque barre représente une mesure, nommée "title", et sa largeur représente la valeur de cette mesure. Comme ce jeu de données ne change pas, notre diagramme à barres ne comporte qu'une sélection "enter":

```
var bars = svg.selectAll(".bars")
    .data(data);

bars.enter()
    .append("rect")
    .attr("class", "bars")
    .attr("x", xScale(0))
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", 0)
    .attr("height", yScale.bandwidth())
    .transition()
    .duration(1000)
    .delay(function(d,i){ return i*200})
    .attr("width", function(d){ return xScale(d.value) - margin.left});
```

Ici, nous réglons la largeur de chaque barre à 0 et, après la transition, à sa valeur finale.

Cette entrée, seule, suffit à créer notre tableau, que vous pouvez voir dans ce violon .

Mais que se passe-t-il si mes données changent?

Dans ce cas, nous devons changer dynamiquement notre graphique. La meilleure façon de le faire est de créer des sélections "enter", "update" et "exit". Mais avant cela, nous devons apporter des modifications au code.

Tout d'abord, nous déplacerons les parties changeantes dans une fonction nommée `draw()` :

```
function draw(){
    //changing parts
};
```

Ces "pièces changeantes" comprennent:

1. Les sélections d'entrée, de mise à jour et de sortie;
2. Le domaine de chaque échelle;
3. La transition de l'axe;

Dans cette fonction `draw()` , nous appelons une autre fonction, qui crée nos données. Ici, c'est juste une fonction qui retourne un tableau de 5 objets, en choisissant au hasard 5 lettres sur 10 (tri par ordre alphabétique) et, pour chacune, une valeur entre 0 et 99:

```
function getData(){
    var title = "ABCDEFGHIJ".split("");
    var data = [];
    for(var i = 0; i < 5; i++){
        var index = Math.floor(Math.random()*title.length);
        data.push({title: title[index],
            value: Math.floor(Math.random()*100)});
        title.splice(index,1);
    }
    data = data.sort(function(a,b){ return d3.ascending(a.title,b.title)});
    return data;
};
```

Et maintenant, passons à nos sélections. Mais avant cela, un mot de prudence: pour maintenir ce que nous appelons *la constance des objets* , nous devons spécifier une fonction clé comme second argument de selection.data:

```
var bars = svg.selectAll(".bars")
    .data(data, function(d){ return d.title});
```

Sans cela, nos barres ne se transforment pas en douceur, et il serait difficile de suivre les changements dans l'axe (vous pouvez voir que supprimer le second argument dans le violon ci-dessous).

Ainsi, après avoir réglé correctement nos `var bars` , nous pouvons traiter nos sélections. Ceci est la sélection de sortie:

```
bars.exit()
    .transition()
    .duration(1000)
    .attr("width", 0)
    .remove();
```

Et ce sont les sélections d'entrée et de mise à jour (dans D3 v4.x, la sélection de mise à jour est fusionnée avec la sélection d'entrée à l'aide de la `merge`):

```
bars.enter()//this is the enter selection
    .append("rect")
    .attr("class", "bars")
    .attr("x", xScale(0) + 1)
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", 0)
    .attr("height", yScale.bandwidth())
    .attr("fill", function(d){ return color(letters.indexOf(d.title)+1)})
    .merge(bars)//and from now on, both the enter and the update selections
    .transition()
    .duration(1000)
    .delay(1000)
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", function(d){ return xScale(d.value) - margin.left});
```

Enfin, nous appelons la fonction `draw()` chaque fois que le bouton est cliqué:

```
d3.select("#myButton").on("click", draw);
```

Et c'est le violon qui montre toutes ces 3 sélections en action.

Fusion de sélections

Le modèle de mise à jour dans la version D3 3

Une compréhension correcte de la façon dont les sélections «enter», «update» et «exit» fonctionnent est fondamentale pour changer correctement le dataviz en utilisant D3.

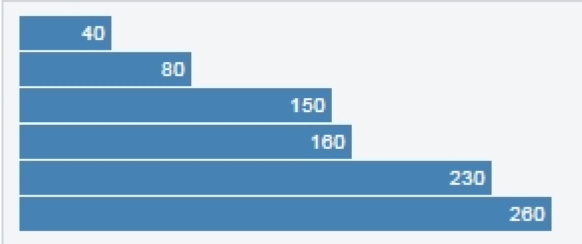
Depuis la version 3 de D3 (en fait, depuis la version 2), cet extrait pouvait définir les transitions pour les sélections «enter» et «update» (démonstration en direct ici):

```
var divs = body.selectAll("div")
    .data(data);//binding the data

divs.enter()//enter selection
    .append("div")
    .style("width", "0px");

divs.transition()
    .duration(1000)
    .style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Donner ce résultat après la transition:



Mais que se passe-t-il avec le même code si nous utilisons la version D3 4? Vous pouvez le voir dans cette démo en direct : *rien* !

Pourquoi?

Changements dans le modèle de mise à jour dans D3 version 4

Vérifions le code. Premièrement, nous avons des `divs` . Cette sélection lie les données au `<div>` .

```
var divs = body.selectAll("div")
    .data(data);
```

Ensuite, nous avons `divs.enter()` , qui est une sélection qui contient toutes les données avec des éléments sans correspondance. Cette sélection contient tous les divs de la première fois que nous appelons la fonction `draw` , et nous définissons leur largeur à zéro.

```
divs.enter()
    .append("div")
    .style("width", "0px");
```

Ensuite, nous avons `divs.transition()` , et ici nous avons le comportement intéressant: dans D3 version 3, `divs.transition()` rend toutes les `<div>` dans la sélection "enter" à leur largeur finale. Mais cela n'a aucun sens! `divs` ne contient pas la sélection “enter” et ne doit modifier aucun élément DOM.

Il y a une raison pour laquelle ce comportement étrange a été introduit dans D3 version 2 et 3 (source ici), et il a été «corrigé» dans la version D3 4. Ainsi, dans la démonstration ci-dessus, rien ne se passe, et c'est prévu! De plus, si vous cliquez sur le bouton, toutes les six barres précédentes apparaissent, car elles sont maintenant dans la sélection «mise à jour», et non plus dans la sélection «entrée».

Pour la transition agissant sur la sélection «enter», nous devons créer des variables distinctes ou, plus facilement, fusionner les sélections :

```
divs.enter()//enter selection
    .append("div")
    .style("width", "0px")
    .merge(divs)//from now on, enter + update selections
    .transition().duration(1000).style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Maintenant, nous avons fusionné les sélections “enter” et “update”. Voyez comment cela

fonctionne dans cette démo en direct .

Lire modèle de mise à jour en ligne: https://riptutorial.com/fr/d3-js/topic/5749/modele-de-mise-a-jour

Chapitre 7: Projections D3

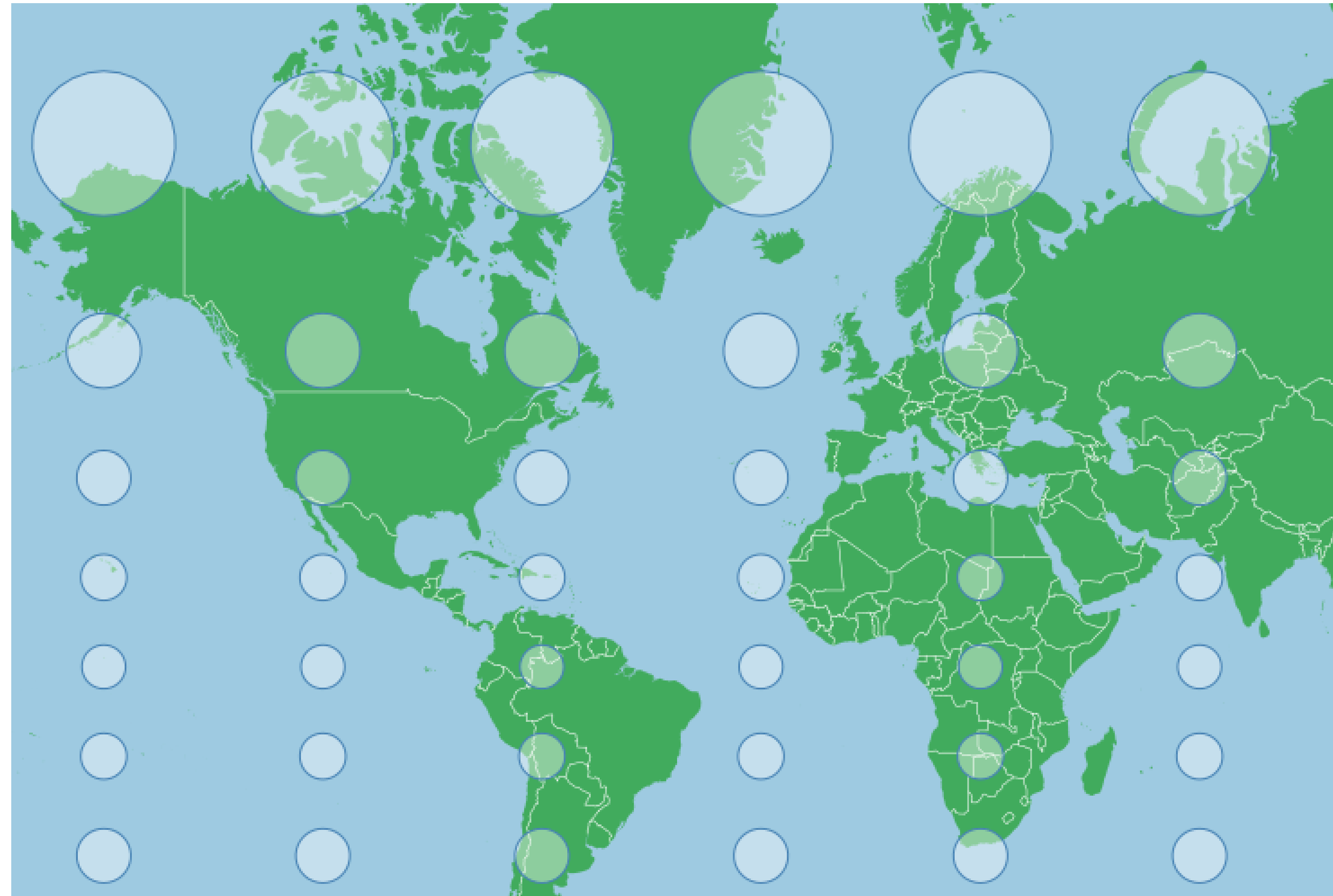
Examples

Projections Mercator

Une projection de Mercator est l'une des projections les plus reconnaissables utilisées dans les cartes. Comme toutes les projections cartographiques, elle présente des distorsions et, pour un Mercator, la projection est particulièrement visible dans les régions polaires. C'est une projection cylindrique, les méridiens se déplacent verticalement et les latitudes sont horizontales.

L'échelle dépend de la taille de votre svg. Pour cet exemple, toutes les échelles utilisées ont un svg de 960 pixels de large sur 450 pixels de haut.

La carte ci-dessous montre un Indicatrix de Tissot pour une projection de Mercator, chaque cercle étant en réalité de la même taille mais la projection en montre évidemment plus que d'autres:



Cette distorsion est due au fait que la projection tente d'éviter un étirement unidimensionnel de la carte. Lorsque les méridiens commencent à fusionner aux pôles Nord et Sud, la distance entre eux commence à se rapprocher de zéro, mais la surface de la projection est rectangulaire (et non la carte, même si elle est rectangulaire) et ne permet pas de modifier la distance. entre les méridiens dans la projection. Cela étirerait les traits le long de l'axe x près des pôles, déformant

leur forme. Pour contrer cela, un Mercator étire l'axe des y et on s'approche des pôles, ce qui rend les indicateurs circulaires.

La projection de la carte ci-dessus est essentiellement la projection par défaut de Mercator légèrement décalée:

```
var projection = d3.geoMercator()
    .scale(155)
    .center([0,40]) // Pan north 40 degrees
    .translate([width/2,height/2]);
```

Pour centrer la projection sur un point donné avec une latitude connue et une longitude connue, vous pouvez vous déplacer facilement vers ce point en spécifiant le centre:

```
var projection = d3.geoMercator()
    .center([longitude,latitude])
```

Cela vous ramènera à cette fonction (mais pas au zoom) sur la surface projetée (qui ressemble à la carte ci-dessus).

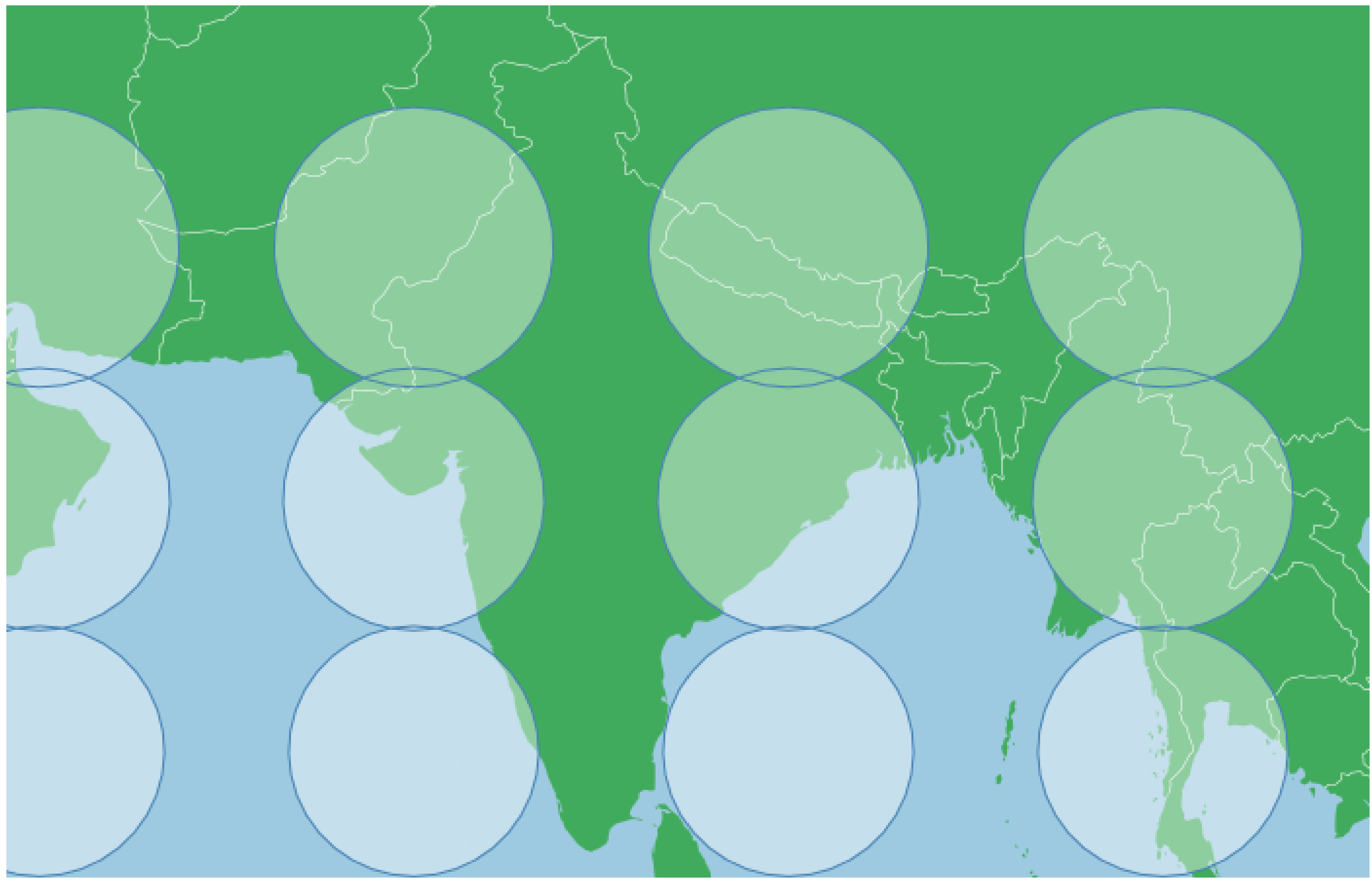
Les échelles devront être adaptées à la zone d'intérêt, les nombres plus grands équivalant à des caractéristiques plus importantes (degré de zoom plus élevé), les nombres plus petits au contraire. Faire un zoom arrière peut être un bon moyen de vous repérer pour voir où vos caractéristiques sont relatives au point sur lequel vous vous êtes concentré - si vous rencontrez des difficultés pour les trouver.

En raison de la nature de la projection de Mercator, les zones proches de l’équateur ou des basses latitudes feront de leur mieux avec ce type de projection, tandis que les zones polaires peuvent être très déformées. La distorsion est même le long de n'importe quelle ligne horizontale, de sorte que les zones larges mais pas grandes peuvent également être bonnes, tandis que les zones très différentes entre leurs extrémités nord et sud présentent davantage de distorsion visuelle.

Pour l'Inde, par exemple, nous pourrions utiliser:

```
var projection = d3.geoMercator()
    .scale(800)
    .center([77,21])
    .translate([width/2,height/2]);
```

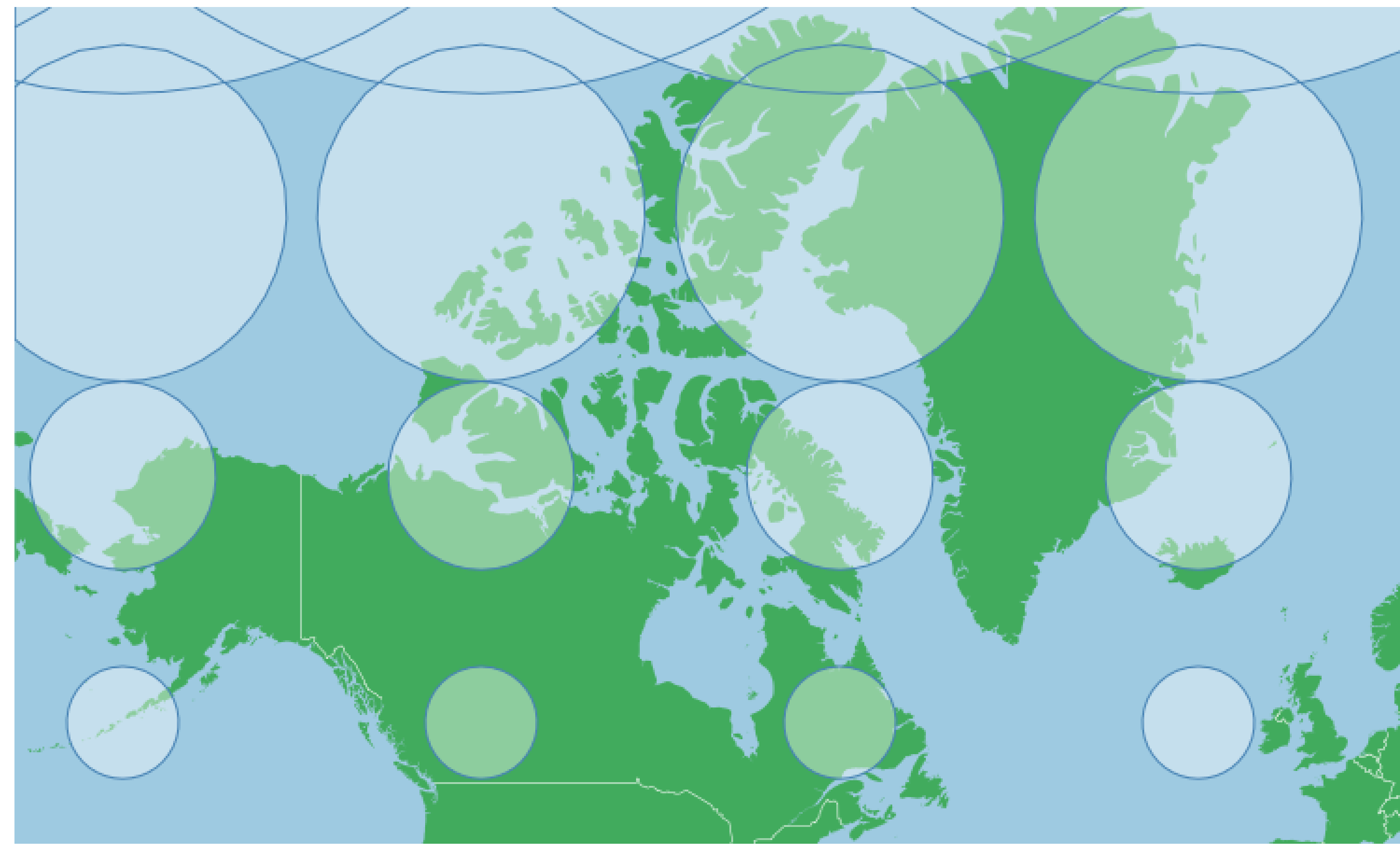
Ce qui nous donne (encore une fois avec l'indicatrice de Tissot pour montrer la distorsion):



Cela a un faible niveau de distorsion, mais les cercles ont la même taille (vous pouvez voir un plus grand chevauchement entre les deux premières lignes que les deux dernières, de sorte que la distorsion est visible). Dans l'ensemble, la carte montre une forme familière pour l'Inde.

La distorsion dans la zone n’est pas linéaire, elle est fortement exagérée vers les pôles, donc le Canada avec des extrêmes nord et sud assez éloignés et une position assez proche des pôles signifie que la distorsion peut être insoutenable:

```
var projection = d3.geoMercator()
    .scale(225)
    .center([-95,69.75])
    .translate([width/2,height/2]);
```

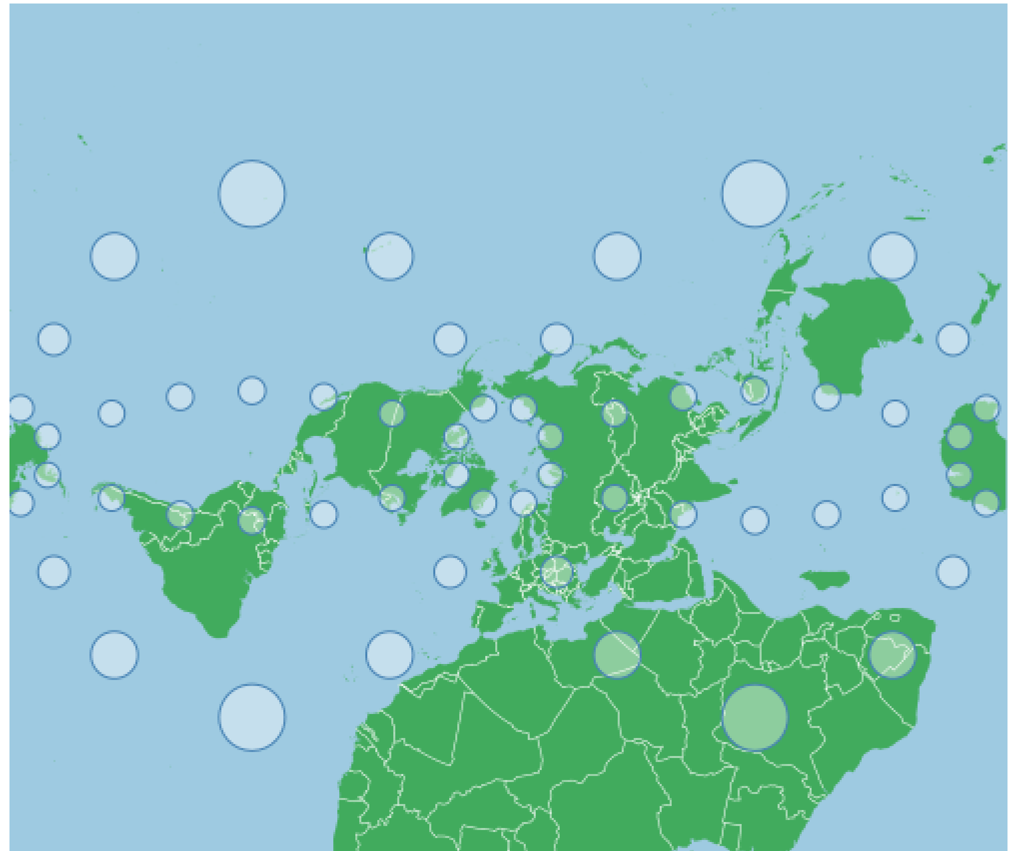


Cette projection rend le Groenland aussi grand que le Canada, alors qu'en réalité, le Canada est près de cinq fois plus grand que le Groenland. C'est tout simplement parce que le Groenland est plus au nord que la majeure partie du Canada (désolé l'Ontario, il me semble avoir coupé une partie de votre extrémité sud).

Comme l'axe des y est considérablement étendu près des pôles dans un Mercator, cette projection utilise un point situé considérablement au nord du centre géographique du Canada. Si vous traitez des zones de haute latitude, vous devrez peut-être adapter votre point de centrage pour tenir compte de cet étirement.

Si vous avez besoin d'une projection Mercator pour les zones polaires, vous pouvez réduire la distorsion et utiliser une projection Mercator. Vous pouvez y parvenir en faisant tourner le globe. Si vous faites pivoter l'axe x sur le Mercator par défaut, il apparaît que vous le déplacez vers la gauche ou la droite (vous faites simplement tourner le globe dans le cylindre sur lequel vous projetez), mais si vous modifiez l'axe y tourner la terre sur le côté ou à un autre angle. Voici un Mercator avec une rotation de -90 degrés:

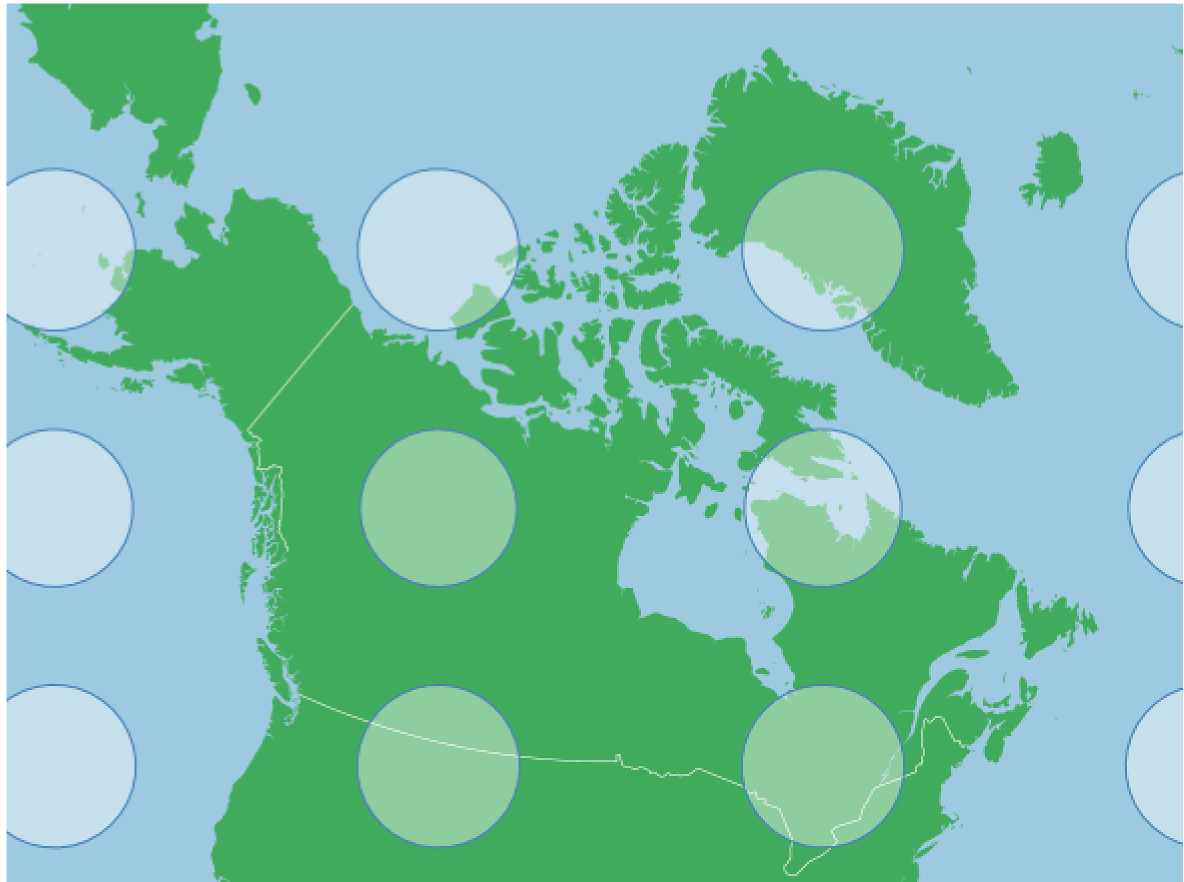
```
var projection = d3.geoMercator()
.scale(155)
.rotate([0,-90]);
.translate([width/2,height/2]);
```



Les points indicatrice sont aux mêmes endroits que la première carte ci-dessus. La distorsion augmente toujours à mesure que l'on atteint le haut ou le bas de la carte. Voici comment une carte Mercator par défaut apparaîtrait si la terre tournait autour d'un pôle Nord à [0,0] et un pôle Sud à [180,0], la rotation a fait pivoter le cylindre sur lequel nous projetons de 90 degrés par rapport à les pôles. Notez que les pôles ne présentent plus de distorsion insoutenable, ceci présente une méthode alternative pour projeter des zones proches des pôles sans trop de distorsion dans la zone.

En utilisant à nouveau le Canada comme exemple, nous pouvons faire pivoter la carte vers une coordonnée centrale, ce qui minimisera la distorsion dans la zone. Pour ce faire, nous pouvons retourner à un point de centrage, mais cela nécessite une étape supplémentaire. En nous centrant sur un élément, avec la rotation nous déplaçons la terre sous nous, nous avons donc besoin du négatif de notre coordonnée de centrage:

```
var projection = d3.geoMercator()
    .scale(500)
    .rotate([96,-64.15])
    .translate([width/2,height/2]);
```



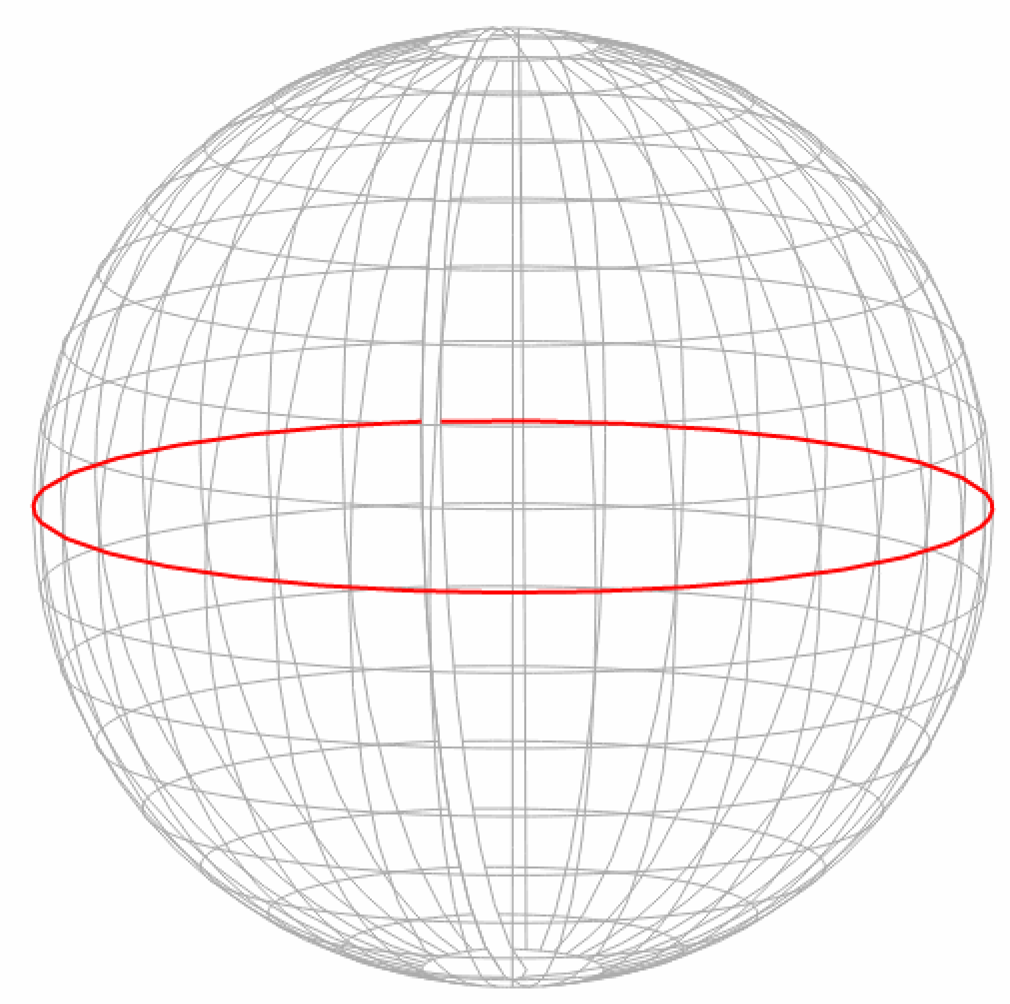
Notez que l'indicatrice de Tissot montre une faible distorsion dans la zone maintenant. Le facteur d'échelle est également beaucoup plus grand qu'avant, car ce Canada est maintenant à l'origine de la projection et le long du milieu de la carte, les caractéristiques sont plus petites que celles du haut ou du bas (voir première matrice ci-dessus). Nous n'avons pas besoin de centrer car le point central ou l'origine de cette projection est à `[-96,64.15]` , le centrage nous `[-96,64.15]` de ce point.

Projections Albers

Une projection d'Albers ou, plus exactement, une projection conique à surface égale d'Albers est une projection conique commune et une projection officielle d'un certain nombre de pays et d'organisations comme le bureau de recensement des États-Unis et la province de la Colombie-Britannique au Canada. Il préserve la zone au détriment des autres aspects de la carte, comme la forme, l'angle et la distance.

Les propriétés générales

La transformation générale est capturée dans le gif suivant:



(Basé sur le bloc de Mike Bostock)

La projection d'Albers minimise la distorsion autour de deux parallèles standard. Ces parallèles représentent l'endroit où la projection conique coupe la surface de la Terre.

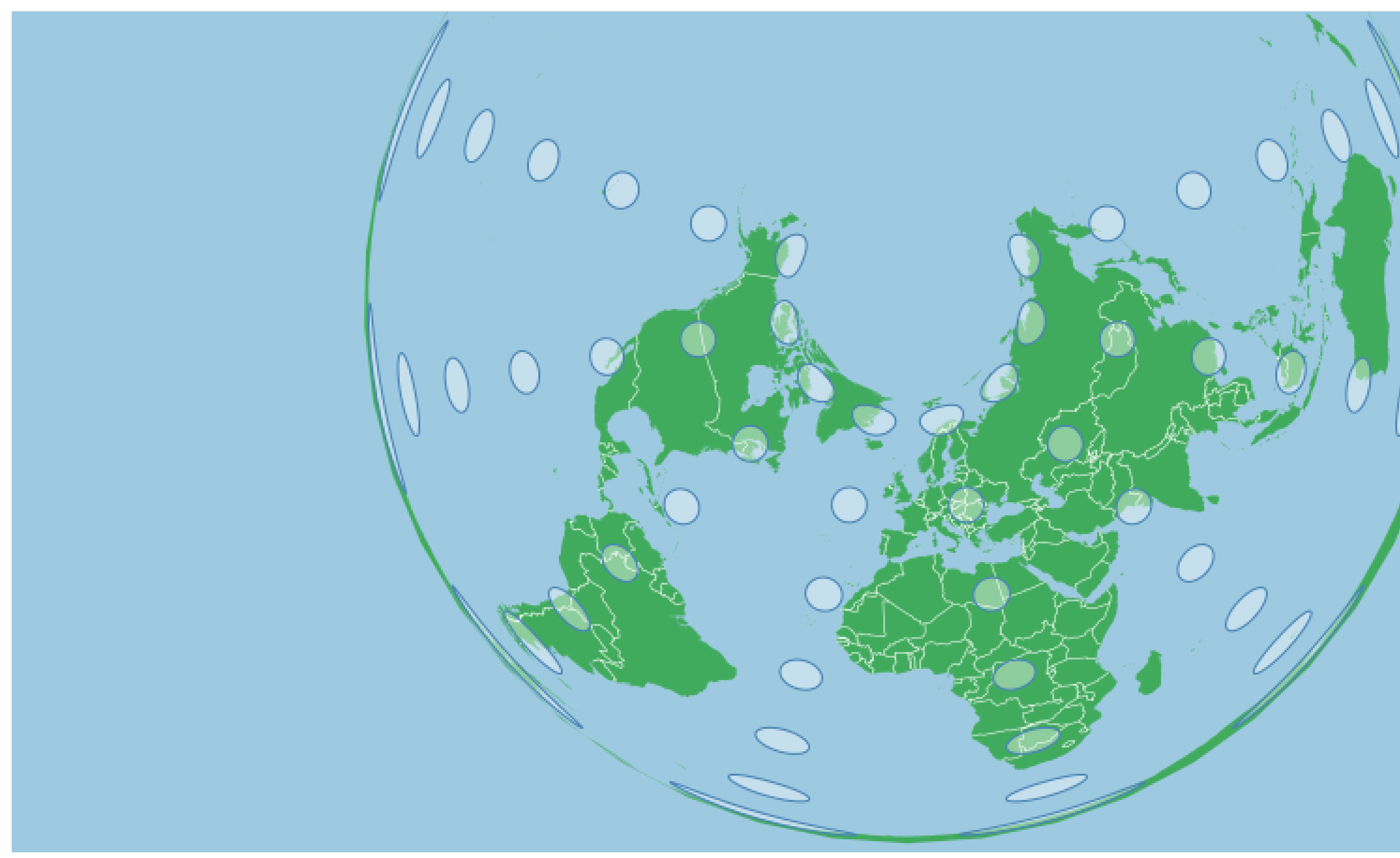
Pour cet exemple, toutes les balances sont utilisées avec des dimensions en svg de 960 pixels de largeur sur 450 pixels de hauteur, l'échelle changera avec ces dimensions

La carte ci-dessous montre un Indicatrix de Tissot pour une projection d'Albers avec des parallèles standard de 10 et 20 degrés nord. Chaque cercle est en réalité de la même taille et de la même forme, mais la projection cartographique les déformera (pas la zone). Notez qu'à environ 10 à 20 degrés nord, les indicateurs sont plus arrondis qu'ailleurs:

Ceci a été créé avec la projection suivante:

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,0])
    .rotate([0,0])
    .parallels([10,20])
    .translate([width/2,height/2]);
```

Si nous utilisons des parallèles à des altitudes plus élevées, le degré de formation d'arcs dans la projection augmente. Les images suivantes utilisent les parallèles de 50 et 60 degrés nord:



```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,70]) // shifted up so that the projection is more visible
    .rotate([0,0])
    .parallels([40,50])
    .translate([width/2,height/2]);
```

Si nous avions des parallèles négatifs (méridionaux), la carte devrait être concave vers le bas plutôt que vers le haut. Si un parallèle est nord et un sud, la carte sera concave vers le parallèle plus haut / plus extrême, si elle est à la même distance de l'équateur, la carte ne sera concave dans aucune direction.

Choisir Parallels

Comme les parallèles marquent les zones ayant le moins de distorsion, elles doivent être choisies en fonction de votre domaine d'intérêt. Si votre zone d'intérêt s'étend de 10 degrés nord à 20 degrés nord, alors choisir des parallèles de 13 et 17 minimisera la distorsion sur toute la carte (car la distorsion est minimisée de chaque côté de ces parallèles).

Les parallèles ne devraient pas être les limites nord et sud extrêmes de votre zone d'intérêt. *Les parallèles peuvent tous deux avoir la même valeur si vous voulez seulement que la projection coupe une fois la surface de la terre.*

Les références et définitions de projection incluent des données parallèles que vous pouvez utiliser pour recréer des projections standardisées.

Centrage et Rotation

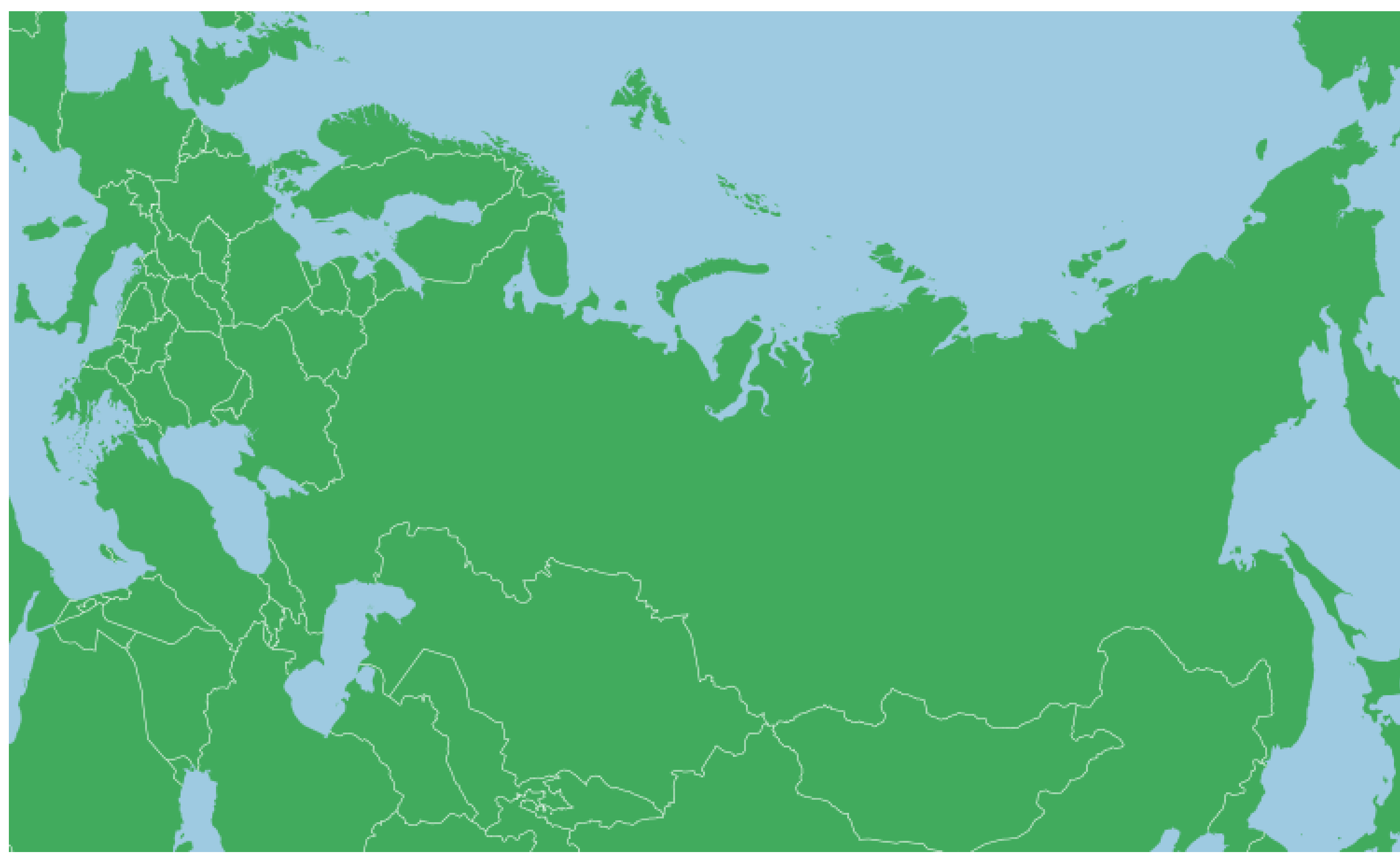
Une fois les parallèles sélectionnés, la carte doit être positionnée de manière à aligner correctement la zone d'intérêt. Si vous utilisez simplement `projection.center([x,y])` , la carte sera simplement déplacée vers le point sélectionné et aucune autre transformation n'aura lieu. Si la zone cible est la Russie, le panoramique ne serait peut-être pas idéal:

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,50]) // Shifted up so the projection is more visible
    .rotate([0,0])
    .parallels([50,60])
    .translate([width/2,height/2]);
```

Le méridien central d'une projection d'Albers est vertical et nous devons faire pivoter la terre sous la projection pour changer le méridien central. La rotation pour la projection d'Alber est la méthode de centrage d'une projection sur l'axe des x (ou par la longitude). Et comme la Terre tourne sous la projection, nous utilisons le négatif de la longitude que nous voulons centrer. Pour la Russie, cela pourrait être d'environ 100 degrés Est, alors nous tournerons le globe à 100 degrés dans l'autre sens.

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,60])
    .rotate([-100,0])
    .parallels([50,60])
```

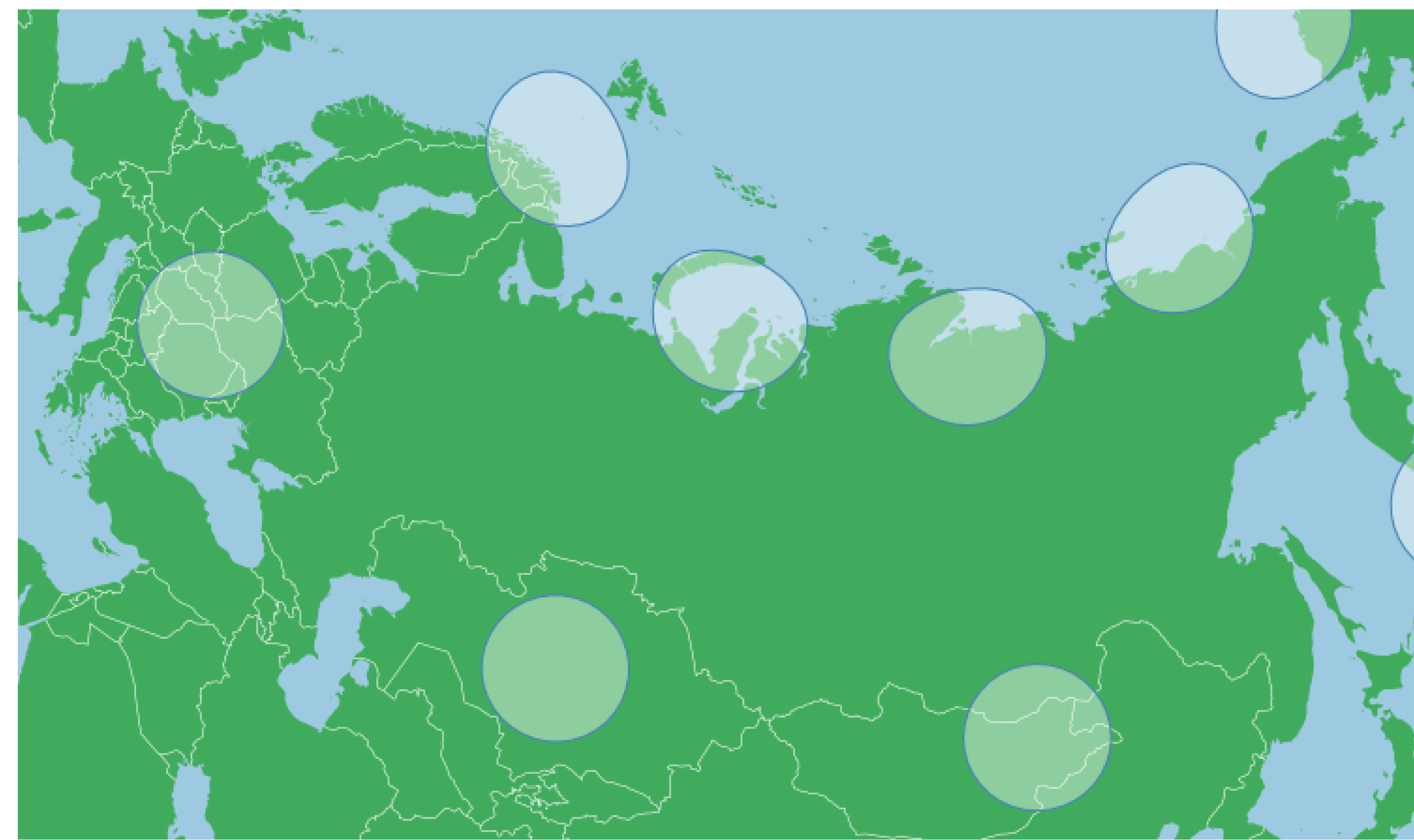
Maintenant, nous pouvons nous déplacer de haut en bas et les caractéristiques le long et près du méridien central seront debout. *Si vous `.center()` sur l'axe des x, votre centrage sera relatif au méridien central défini par la rotation* . Pour la Russie, nous pourrions vouloir aller un peu plus au nord et zoomer un peu:



```
var projection = d3.geoAlbers()
    .scale(500)
    .center([0,65])
    .rotate([-100,0])
    .parallels([50,60])
```

Pour une fonctionnalité comme la Russie, la voûte de la carte signifie que les extrémités du pays seront étendues autour du poteau, ce qui signifie que le point de centrage ne sera peut-être pas le centroïde de votre équipement. nord ou sud que d'habitude.

Avec les Tissots Indicatrix, on peut voir un aplatissement près du pôle lui-même, mais cette forme est assez vraie dans toute la zone d'intérêt (rappelez-vous que pour la taille de la Russie, la distorsion est plutôt minime:



Paramètres par défaut

Contrairement à la plupart des autres projections, la projection d3.geoAlbers comprend des paramètres par défaut qui ne sont pas .rotate ([0,0]) et .center ([0,0]), la projection par défaut est centrée et pivotée pour les États-Unis. Cela est également vrai pour `.parallels()` . Donc, si l'un de ces éléments n'est pas défini, les valeurs par défaut seront nulles.

Résumé

Une projection d'Albers est généralement définie avec les paramètres suivants:

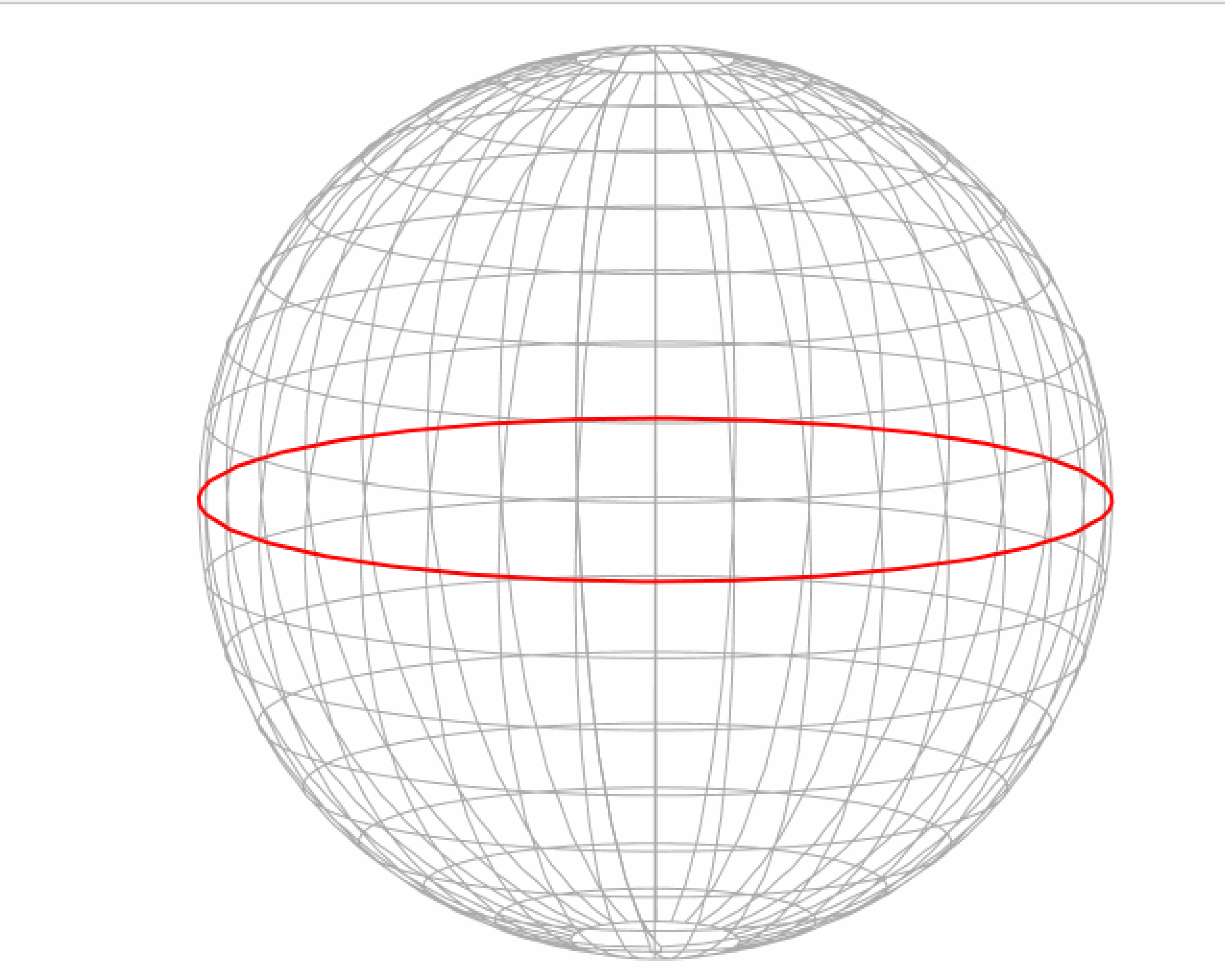
```
var projection = d3.geoAlbers()
    .rotate([-x,0])
    .center([0,y])
    .parallels([a,b]);
```

Où a et b égalent les deux parallèles.

Projections Equidistantes Azimutales

Les propriétés générales:

Une projection équidistante azimutale est mieux reconnue lorsqu'elle est utilisée dans les zones polaires. Il est utilisé dans l' emblème de l' ONU . Du point central, l'angle et la distance sont préservés. Mais la projection faussera la forme et la surface pour y parvenir, en particulier lorsque l'on s'éloigne du centre. De même, la distance et l'angle ne sont pas vrais dans des endroits autres que le centre. La projection se situe dans la catégorie azimutale (plutôt que cylindrique (Mercator) ou conique (Albers). Cette projection projette la terre comme un disque:



(Basé sur le bloc de Mike Bostock. *Centré sur le pôle Nord, Ignorer l'artefact triangulaire au-dessus de l'image une fois plié*)

L'échelle dépend de la taille de votre svg, pour cet exemple, toutes les échelles utilisées se situent dans un svg de 960 pixels de large sur 450 pixels de haut (et l'écran est découpé pour un carré si nécessaire) - sauf indication contraire.

La carte ci-dessous montre un Indicatrix de Tissot pour une projection équidistante azimutale:

Ceci a été créé avec la projection suivante:

```
var projection = d3.geoAzimuthalEquidistant()
   .scale(70)
   .center([0,0])
   .rotate([0,0])
   .translate([width/2,height/2]);
```

Centrage et Rotation:

Le centrage consiste simplement à effectuer un panoramique sur une carte, mais pas à modifier sa composition globale. Centrer un azimutal équidistant sur le pôle Nord tout en laissant les autres paramètres inchangés ou à zéro déplacera le pôle nord au centre de l'écran - mais ne changera pas la carte ci-dessus.

Pour centrer correctement une zone, vous devez la faire pivoter. Comme pour toute rotation de d3, il est préférable de penser que la terre se déplace sous la projection, donc la rotation de la terre de -90 degrés sous la carte sur l'axe des y placera le pôle nord au centre:

```
var projection = d3.geoAzimuthalEquidistant()
   .scale(70)
   .center([0,0])
   .rotate([0,-90])
   .translate([width/2,height/2]);
```

De même, la rotation sur l'axe des x se comporte de la même manière. Par exemple, pour faire pivoter la carte de manière à ce que l'Arctique canadien soit en position verticale, tout en centrant sur le pôle nord, nous pouvons utiliser une projection comme celle-ci:

```
var projection = d3.geoAzimuthalEquidistant()
     .scale(400)
     .center([0,0])
     .rotate([100,-90])
     .translate([width/2,height/2]);
```

Cette carte utilisait un svg 600x600

Dans l'ensemble, cette simplicité rend l'équation azimutale plus facile à définir, il suffit d'utiliser la rotation. Une projection typique ressemblera à:

```
var projection = d3.geoProjection()
    .center([0,0])
    .rotate([-x,-y])
    .scale(k)
    .translate([width/2,height/2]);
```

Lire Projections D3 en ligne: https://riptutorial.com/fr/d3-js/topic/9001/projections-d3

Chapitre 8: Rendre robuste, réactif et réutilisable (r3) pour d3

Introduction

d3 est une bibliothèque puissante pour créer des graphiques interactifs; cependant, cette puissance découle du fait que les utilisateurs doivent travailler à un niveau inférieur à celui des autres bibliothèques interactives. Par conséquent, de nombreux exemples de diagrammes D3 sont conçus pour montrer comment produire une chose particulière - par exemple, les moustaches pour une boîte et un diagramme à moustaches - tout en codant souvent les paramètres, ce qui rend le code rigide. L'objectif de cette documentation est de démontrer comment créer plus de code réutilisable pour gagner du temps à l'avenir.

Examples

Scatter Plot

Cet exemple contient plus de 1000 lignes de code au total (trop à incorporer ici). Pour cette raison, tout le code est accessible sur http://blockbuilder.org/SumNeuron/956772481d4e625eec9a59fdb9fbe5b2 (également hébergé sur https://bl.ocks.org/SumNeuron/956772481d4e625eec9a59fdb9fbe5b2) . Notez que bl.ocks.org utilise l'iframe donc pour voir le redimensionnement, vous devrez cliquer sur le bouton ouvert (coin inférieur droit de l'iframe). Comme il y a beaucoup de code, il a été divisé en plusieurs fichiers et le segment de code correspondant fera référence à la fois par nom de fichier et par numéro de ligne. Veuillez ouvrir cet exemple au fur et à mesure.

Qu'est-ce qui fait un graphique?

Il existe plusieurs composants de base qui entrent dans un tableau complet; à savoir ceux-ci comprennent:

- Titre
- les axes
- étiquettes d'axes
- les données

D'autres aspects pourraient être inclus en fonction du graphique - par exemple, une légende de graphique. Cependant, bon nombre de ces éléments peuvent être contournés avec une info-bulle. Pour cette raison, il existe des éléments spécifiques au graphique interactif - par exemple des boutons pour basculer entre les données.

Puisque le contenu de notre graphique sera interactif, il serait approprié que le graphique lui-

même soit dynamique - par exemple, redimensionne lorsque la taille de la fenêtre change. SVG est évolutif, vous pouvez donc simplement adapter votre graphique en conservant la perspective actuelle. Cependant, en fonction de la perspective définie, le graphique peut devenir trop petit pour être lisible même s'il reste suffisamment d'espace pour le graphique (par exemple, si la largeur est supérieure à la hauteur). Par conséquent, il peut être préférable de simplement redessiner le graphique dans la taille restante.

Cet exemple explique comment calculer de manière dynamique le placement des boutons, des titres, des axes, des étiquettes d'axes, ainsi que gérer des jeux de données de quantités variables de données.

Installer

Configuration

Comme nous visons la réutilisation du code, nous devrions créer un fichier de configuration contenant des options globales pour certains aspects de notre graphique. Un exemple d'un tel fichier de configuration est `charts_configuration.json` .

Si nous regardons ce fichier, nous pouvons voir que j'ai inclus plusieurs éléments qui devraient déjà avoir une utilisation claire pour notre graphique:

- fichiers (stocke la chaîne où se trouve notre graphique)
- document_state (quel bouton est actuellement sélectionné pour notre graphique)
- chart_ids (identifiants HTML pour les graphiques que nous allons faire)
- svg (options pour le svg, par exemple la taille)
- plot_attributes
  - title (définir divers attributs de police)
  - infobulle (définir diverses propriétés de style d'info-bulle)
  - axes (définir divers attributs de police)
  - boutons (définir divers attributs de police et de style)
- parcelles
  - diffusion (définir divers aspects de notre nuage de points, par exemple rayon de point)
- couleurs (une palette de couleurs spécifique à utiliser)

Fonctions d'aide

En plus de définir ces aspects globaux, nous devons définir certaines fonctions d'assistance. Ceux-ci peuvent être trouvés sous `helpers.js`

- `ajax_json` (charge les fichiers json de manière synchrone ou asynchrone)
- `keys` (renvoie les clés de l'objet donné - équivalent à d3.keys ())
- `parseNumber` (une analyse générale des nombres au cas où nous ne saurions pas quel type ou quel nombre est)
- `typeofNumber` (renvoie le type de numéro)

index.html

Enfin, nous devrions configurer notre fichier HTML. Dans le cadre de cet exemple, nous placerons notre graphique dans une balise de `section` où l' `id` correspond à l'identifiant fourni dans le fichier de configuration (ligne 37). Comme les pourcentages ne fonctionnent que s'ils peuvent être calculés à partir de leur membre parent, nous incluons également des styles de base (lignes 19-35).

Faire notre nuage de points

Ouvrons `make_scatter_chart.js` . Maintenant, portons une attention particulière à la ligne 2, où plusieurs des variables les plus importantes sont prédéfinies:

- svg - d3 selection du svg du graphique
- chart_group - sélection d3 du groupe dans le svg dans lequel les données seront placées
- canvas - aspects fondamentaux de l'extrait de svg pour plus de commodité
- marges - les marges à prendre en considération
- maxi_draw_space les plus grandes valeurs de x et y dans lesquelles nous pouvons dessiner nos données
- doc_state - l'état actuel du document si nous utilisons des boutons (dans cet exemple, nous sommes)

Vous avez peut-être remarqué que nous n'avons pas inclus le svg dans le HTML. Par conséquent, avant de pouvoir faire quelque chose avec notre graphique, nous devons ajouter svg à `index.html` s'il n'existe pas encore. Ceci est réalisé dans le fichier `make_svg.js` par la fonction `make_chart_svg` . En regardant `make_svg.js` nous voyons que nous utilisons la fonction d'assistance `parseNumber` sur la configuration du graphique pour la largeur et la hauteur de svg. Si le nombre est un flottant, la largeur et la hauteur du svg sont proportionnelles à la largeur et à la hauteur de la section. Si le nombre est un nombre entier, il sera simplement défini sur ces nombres entiers.

Lignes 6 - 11 teste pour voir - en effet - s'il s'agit du premier appel ou non et définit le `chart_group` (et l'état du document s'il s'agit du premier appel).

La ligne 14-15 extrait les données actuellement sélectionnées par le bouton cliqué. la ligne 16 définit `data_extent` . Alors que d3 a une fonction pour extraire l'étendue des données, *je* préfère stocker l'étendue des données dans cette variable.

Les lignes 27 à 38 contiennent la magie qui définit notre graphique en faisant les marges, les boutons, le titre et les axes. Celles-ci sont toutes déterminées dynamiquement et peuvent sembler un peu complexes, nous allons donc les examiner à tour de rôle.

make_margins (dans make_margins.js)

Nous pouvons voir que l'objet des marges prend en compte de l'espace à gauche, à droite, en haut et en bas du graphique (respectivement x.left, x.right, y.top, y.bottom), le titre, les boutons et

les axes.

Nous voyons également que les marges des axes sont mises à jour à la ligne 21.

Pourquoi faisons-nous cela? Eh bien, à moins de spécifier le nombre de tics, les étiquettes à cocher la taille à cocher et la taille de la police à cocher, nous n'avons pas pu calculer la taille nécessaire aux axes. Même dans ce cas, il faudrait encore évaluer l'espace entre les étiquettes de tiques et les tiques. Par conséquent, il est plus facile de créer des axes fictifs à l'aide de nos données, de voir la taille des éléments svg correspondants, puis de renvoyer la taille.

En fait, nous n'avons besoin que de la largeur de l'axe y et de la hauteur de l'axe x, ce qui est stocké dans axes.y et axes.x.

Avec nos marges par défaut définies, nous calculons alors l' `max_drawing_space` (lignes 29-34 dans make_margins.js)

make_buttons (dans make_buttons.js)

La fonction crée un groupe pour tous les boutons, puis un groupe pour chaque bouton, qui à son tour stocke un cercle et un élément de texte. La ligne 37 - 85 calcule la position des boutons. Pour ce faire, il vérifie si le texte à droite de la longueur de chaque bouton est plus long que l'espace autorisé (ligne 75). Si c'est le cas, il laisse tomber le bouton sur une ligne et met à jour les marges.

make_title (dans make_title.js)

`make_title` est similaire à make_buttons dans la mesure où il divisera automatiquement le titre de votre graphique en plusieurs lignes et coupera le trait si nécessaire. C'est un peu piraté car il ne possède pas la sophistication du schéma de césure de TeX, mais il fonctionne suffisamment bien. Si nous avons besoin de plus de lignes, les marges sont mises à jour.

Avec les boutons, le titre et les marges définis, nous pouvons créer nos axes.

make_axes (dans make_axes.js)

La logique de make_axes reflète celle du calcul de l'espace requis par les axes factices. Ici, cependant, il ajoute des transitions pour changer entre les axes.

Enfin notre nuage de points

Avec tous ces réglages effectués, nous pouvons enfin créer notre nuage de points. Comme nos ensembles de données peuvent comporter un nombre de points différent, nous devons en tenir compte et tirer parti des événements d'entrée et de sortie de d3 en conséquence. L'obtention du nombre de points déjà existants est faite à la ligne 40. L'instruction if de la ligne 45 - 59 ajoute plus d'éléments de cercle si nous avons plus de données, ou transpose les éléments supplémentaires dans un coin et les supprime s'il y en a trop.

Une fois que nous savons que nous avons le bon nombre d'éléments, nous pouvons transférer tous les éléments restants à leur position correcte (ligne 64)

Enfin, nous ajoutons une info-bulle aux lignes 67 et 68. La fonction tooltip se trouve dans `make_tooltip.js`

Tableau des boîtes et moustaches

Pour montrer la valeur de faire des fonctions généralisées comme celles de l'exemple précédent (make_title, make_axes, make_buttons, etc.), considérez cette case et cette grille: https://bl.ocks.org/SumNeuron/262e37e2f932cf4b693f241c52a410ff

Bien que le code pour créer les boîtes et les moustaches soit plus intense que le simple placement des points, nous constatons que les mêmes fonctions fonctionnent parfaitement.

Diagramme à bandes

https://bl.ocks.org/SumNeuron/7989abb1749fc70b39f7b1e8dd192248

Lire Rendre robuste, réactif et réutilisable (r3) pour d3 en ligne: https://riptutorial.com/fr/d3-js/topic/9849/rendre-robuste--reactif-et-reutilisable--r3--pour-d3

Chapitre 9: Sélections

Syntaxe

- d3. **sélectionnez** (sélecteur)
- d3. **selectAll** (sélecteur)
- *sélection* **sélectionnez** (sélecteur)
- *sélection* **selectAll** (sélecteur)
- *sélection* **filtre** (filtre)
- *sélection* **fusionner** (autre)

Remarques

Lectures connexes:

- Comment fonctionnent les sélections - Mike Bostock
- d3-selection README

Examples

Sélection de base et modifications

Si vous êtes familier avec la syntaxe jQuery et Sizzle, les sélections d3 ne devraient pas être très différentes. d3 imite l'API des sélecteurs du W3C pour faciliter les interactions avec les éléments.

Pour un exemple élémentaire, sélectionnez tous `<p>` et ajoutez une modification à chacun d'eux:

```
d3.selectAll('p')
  .attr('class','textClass')
  .style('color', 'white');
```

En bref, c'est relativement le même que dans jQuery

```
$('p')
  .attr('class','textClass')
  .css('color, 'white')
```

En général, vous commencerez par une seule sélection sur votre div pour ajouter un élément SVG qui sera assigné à une variable (le plus souvent appelée svg).

```
var svg = d3.select('#divID').append('svg');
```

De là, nous pouvons faire appel à `svg` pour faire nos sous-sélections de plusieurs objets (même s'ils n'existent pas encore).

```
svg.selectAll('path')
```

Différents sélecteurs

Vous pouvez sélectionner des éléments avec des sélecteurs différents:

- par tag: `"div"`
- par classe: `".class"`
- par id: `"#id"`
- par attribut: `"[color=blue]"`
- sélecteurs multiples (OR): `"div1, div2, class1"`
- plusieurs sélecteurs (AND): `"div1 div2 class1"`

Sélection limitée des données simples

```
var myData = [
    { name: "test1", value: "ok" },
    { name: "test2", value: "nok" }
]

// We have to select elements (here div.samples)
// and assign data. The second parameter of data() is really important,
// it will determine the "key" to identify part of data (datum) attached to an
// element.
var mySelection = d3.select(document.body).selectAll("div.samples") // <- a selection
                   .data(myData, function(d){ return d.name; }); // <- data binding

// The "update" state is when a datum of data array has already
// an existing element associated.
mySelection.attr("class", "samples update")

// A datum is in the "enter" state when it's not assigned
// to an existing element (based on the key param of data())
// i.e. new elements in data array with a new key (here "name")
mySelection.enter().append("div")
    .attr("class", "samples enter")
    .text(function(d){ return d.name; });

// The "exit" state is when the bounded datum of an element
// is not in the data array
// i.e. removal of a row (or modifying "name" attribute)
// if we change "test1" to "test3", "test1" bounded
//          element will figure in exit() selection
// "test3" bounded element will be created in the enter() selection
mySelection.exit().attr("class", "samples remove");
```

Le rôle des espaces réservés dans les sélections "enter"

Qu'est-ce qu'une sélection entrée?

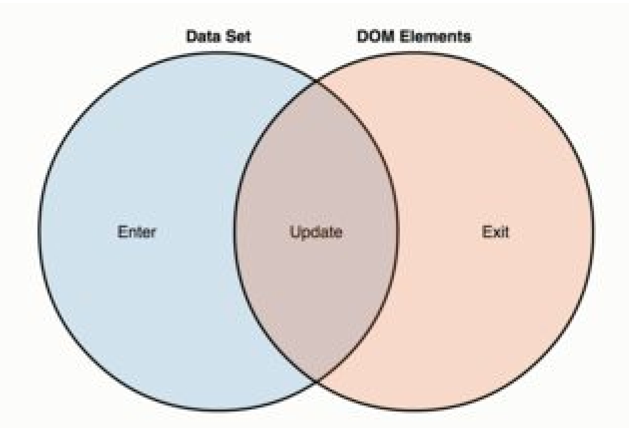
Dans D3.js, quand on lie des données à des éléments DOM, trois situations sont possibles:

1. Le nombre d'éléments et le nombre de points de données sont les mêmes.
2. Il y a plus d'éléments que de points de données.

3. Il y a plus de points de données que d'éléments.

Dans la situation n ° 3, tous les points de données sans élément DOM correspondant appartiennent à la sélection *entrée* . Ainsi, dans D3.js, les sélections de *saisie* sont des sélections qui, après avoir joint des éléments aux données, contiennent toutes les données qui ne correspondent à aucun élément DOM. Si nous utilisons une fonction d' `append` dans une sélection d' *entrée* , D3 créera de nouveaux éléments liant ces données pour nous.

Ceci est un diagramme de Venn expliquant les situations possibles concernant le nombre de points de données / nombre d'éléments DOM:



Comme on peut le voir, la sélection d' *entrée* est la zone bleue à gauche: les points de données sans éléments DOM correspondants.

La structure de la sélection d'entrée

En règle générale, une sélection de *saisie* comporte les 4 étapes suivantes:

1. `selectAll` : sélectionne les éléments dans le DOM;
2. `data` : Compte et analyse les données;
3. `enter` : La comparaison de la sélection avec les données crée de nouveaux éléments.
4. `append` : Ajoute les éléments réels dans le DOM;

Ceci est un exemple très basique (regardez les 4 étapes des `var divs`):

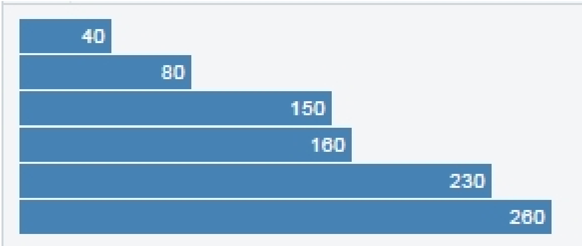
```
var data = [40, 80, 150, 160, 230, 260];

var body = d3.select("body");

var divs = body.selectAll("div")
    .data(data)
    .enter()
    .append("div");

divs.style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Et voici le résultat (jsfiddle ici):



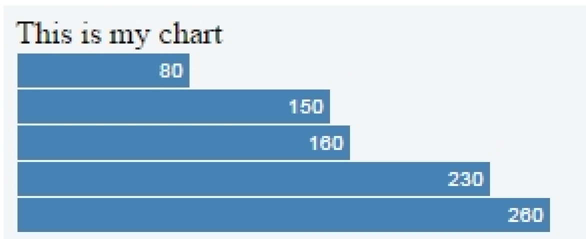
Notez que, dans ce cas, nous avons utilisé `selectAll("div")` comme première ligne de notre variable de sélection "enter". Nous avons un jeu de données avec 6 valeurs et D3 a créé 6 divs pour nous.

Le rôle des espaces réservés

Mais supposons que nous ayons déjà une div dans notre document, quelque chose comme `<div>This is my chart</div>` en haut. Dans ce cas, quand on écrit:

```
body.selectAll("div")
```

nous sélectionnons cette div existante. Ainsi, notre sélection d'entrée n'aura que 5 données sans éléments correspondants. Par exemple, dans ce jsfiddle , où il existe déjà un div dans le HTML ("Ceci est mon graphique"), ce sera le résultat:



Nous ne voyons plus la valeur "40": notre première "barre" a disparu, et la raison en est que notre sélection "enter" ne comporte plus que 5 éléments.

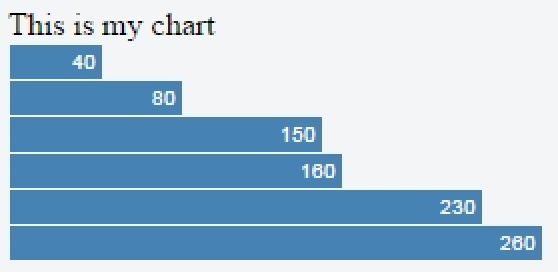
Ce que nous devons comprendre ici, c'est que dans la première ligne de notre variable de sélection d'entrée, `selectAll("div")` , ces divs ne sont que des *espaces réservés* . Nous n'avons pas à sélectionner tous les `divs` si nous ajoutons des `divs` , ou tout le `circle` si nous ajoutons des `circle` . Nous pouvons sélectionner différentes choses. Et, si nous ne prévoyons pas de "mise à jour" ou de "sortie", nous pouvons sélectionner *n'importe quoi* :

```
var divs = body.selectAll(".foo")//this class doesn't exist, and never will!
    .data(data)
    .enter()
    .append("div");
```

En faisant cela, nous sélectionnons tous les ".foo". Ici, "foo" est une classe qui non seulement n'existe pas, mais qui n'est jamais créée nulle part ailleurs dans le code! Mais peu importe, ce n'est qu'un espace réservé. La logique est la suivante:

> Si dans votre sélection "enter" vous sélectionnez quelque chose qui n'existe pas, votre sélection "enter" contiendra *toujours* toutes vos données.

Maintenant, en sélectionnant `.foo` , notre sélection "enter" a 6 éléments, même si nous avons déjà une div dans le document:



Et voici le jsfiddle correspondant .

Sélectionner `null`

Le meilleur moyen de garantir que vous ne sélectionnez rien est de sélectionner `null` . Non seulement cela, mais cette alternative est beaucoup plus rapide que tout autre.

Ainsi, pour une sélection de saisie, il suffit de faire:

```
selection.selectAll(null)
    .data(data)
    .enter()
    .append(element);
```

Voici un violon de démonstration: https://jsfiddle.net/gerardofurtado/th6s160p/

Conclusion

Lorsque vous utilisez les sélections "enter", prenez soin de ne pas sélectionner quelque chose qui existe déjà. Vous pouvez utiliser n'importe quoi dans votre `selectAll` , même les choses qui n'existent pas et n'existeront jamais (si vous ne prévoyez pas d'avoir une sélection de "mise à jour" ou de "sortie").

Le code dans les exemples est basé sur ce code par Mike Bostock: https://bl.ocks.org/mbostock/7322386

Utiliser "this" avec une fonction de flèche

La plupart des fonctions de D3.js acceptent une fonction anonyme comme argument. Les exemples communs sont `.attr` , `.style` , `.text` , `.on` et `.data` , mais la liste est beaucoup plus grande que cela.

Dans de tels cas, la fonction anonyme est évaluée pour chaque élément sélectionné, dans l'ordre, en passant:

1. Le datum actuel (`d`)
2. L'index actuel (`i`)
3. Le groupe actuel (`nodes`)
4. `this` comme l'élément DOM actuel.

Le datum, l'index et le groupe courant sont passés en arguments, les fameux premier, deuxième et troisième arguments de D3.js (dont les paramètres sont traditionnellement nommés `d` , `i` et `p` dans D3 v3.x). Pour l' utilisation de `this` , cependant, on n'a pas besoin d'utiliser aucun argument:

```
.on("mouseover", function(){
    d3.select(this);
});
```

Le code ci-dessus sélectionnera `this` quand la souris est sur l'élément. Vérifiez qu'il fonctionne dans ce violon: https://jsfiddle.net/y5fwgopx/

La fonction flèche

En tant que nouvelle syntaxe ES6, une fonction de flèche a une syntaxe plus courte par rapport à l'expression de fonction. Cependant, pour un programmeur D3 qui utilise `this` constante, il y a un piège: une fonction de flèche ne crée pas son propre `this` contexte. Cela signifie que, dans une fonction de flèche, `this` a sa signification originale à partir du contexte englobant.

Cela peut être utile dans plusieurs circonstances, mais il est un problème pour un codeur habitué à utiliser `this` dans D3. Par exemple, en utilisant le même exemple dans le violon ci-dessus, cela ne fonctionnera pas:

```
.on("mouseover", ()=>{
    d3.select(this);
});
```

Si vous en doutez, voici le violon: https://jsfiddle.net/tfxLsv9u/

Eh bien, ce n'est pas un gros problème: on peut simplement utiliser une expression de fonction classique, à l'ancienne, si nécessaire. Mais que faire si vous voulez écrire tout votre code en utilisant les fonctions de flèche? Est - il possible d'avoir un code avec des fonctions de direction **et** toujours utiliser correctement `this` dans D3?

Les deuxième et troisième arguments combinés

La réponse est **oui** , car `this` va de même pour les `nodes[i]` . L'indice est effectivement présent partout dans l'API D3, lorsqu'il décrit ceci:

> ... avec `this` comme élément DOM actuel (`nodes[i]`)

L'explication est simple: puisque `nodes` est le groupe actuel d'éléments dans le DOM et que `i` est l'index de chaque élément, les `nodes[i]` font référence à l'élément DOM actuel. C'est `this` .

Par conséquent, on peut utiliser:

```
.on("mouseover", (d, i, nodes) => {
    d3.select(nodes[i]);
});
```

Et voici le violon correspondant: https://jsfiddle.net/2p2ux38s/

Lire Sélections en ligne: https://riptutorial.com/fr/d3-js/topic/2135/selections

Chapitre 10: Sur les événements

Syntaxe

- .on ('mouseover', fonction)
- .on ('mouseout', fonction)
- .on ('clic', fonction)
- .on ('mouseenter', fonction)
- .on ('mouseleave', fonction)

Remarques

Pour un exemple plus détaillé où les événements personnalisés sont définis, reportez-vous ici .

Examples

Joindre des événements de base sur des sélections

Souvent, vous voudrez avoir des événements pour vos objets.

```
function spanOver(d,i){
  var span = d3.select(this);
  span.classed("spanOver",true);
}

function spanOut(d,i){
  var span = d3.select(this);
  span.classed("spanOver", false);
}

var div = d3.select('#divID');

div.selectAll('span')
  .on('mouseover' spanOver)
  .on('mouseout' spanOut)
```

Cet exemple ajoutera la classe `spanOver` lors du `spanOver` un intervalle à l'intérieur du div avec l'id `divID` et le supprimera lorsque la souris quittera le span.

Par défaut, d3 transmettra le datum de la plage actuelle et de l'index. Il est très pratique que `this` le contexte actuel, de sorte que nous puissions y effectuer des opérations, comme ajouter ou supprimer des classes.

Vous pouvez également simplement utiliser une fonction anonyme sur l'événement.

```
div.selectAll('span')
  .on('click', function(d,i){ console.log(d); });
```

Des éléments de données peuvent également être ajoutés à l'objet sélectionné.

```
div.selectAll('path')
  .on('click', clickPath);

function clickPath(d,i) {
  if(!d.active) {
    d.active = true;
    d3.select(this).classed("active", true);
  }
  else {
    d.active = false;
    d3.select(this).classed("active", false);
  }
}
```

Dans cet exemple, active n'est pas définie sur la sélection avant que l'événement click ne soit déclenché. Si vous deviez revenir sur la sélection du chemin, tous les objets cliqués contiendraient la clé `active` .

Supprimer un écouteur d'événement

d3.js n'a pas de méthode `.off()` pour détecter les écouteurs d'événements existants. Pour supprimer un gestionnaire d'événement, vous devez le définir sur `null` :

```
d3.select('span').on('click', null)
```

Lire Sur les événements en ligne: https://riptutorial.com/fr/d3-js/topic/2722/sur-les-evenements

Chapitre 11: Utiliser D3 avec d'autres frameworks

Examples

Composant D3.js avec ReactJS

Cet exemple est basé sur un article de blog de Nicolas Hery . Il utilise les classes ES6 et les méthodes de cycle de vie de ReactJS pour maintenir le composant D3 à jour

---

d3_react.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Hello, d3React!</title>
  <style>
    .d3Component {
      width: 720px;
      height: 120px;
    }
  </style>
</head>
<script src="https://fb.me/react-15.2.1.min.js"></script>
<script src="https://fb.me/react-dom-15.2.1.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
<script src="https://d3js.org/d3.v4.min.js"></script>

<body>
  <div id="app" />
  <script type="text/babel" src="d3_react.js"></script>
</body>

</html>
```

d3_react.js

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      d3React: new d3React()
    };
    this.getd3ReactState = this.getd3ReactState.bind(this);
  }
```

```
  getd3ReactState() {
    // Using props and state, calculate the d3React state
    return ({
      data: {
        x: 0,
        y: 0,
        width: 42,
        height: 17,
        fill: 'red'
      }
    });
  }

  componentDidMount() {
    var props = {
      width: this._d3Div.clientWidth,
      height: this._d3Div.clientHeight
    };
    var state = this.getd3ReactState();
    this.state.d3React.create(this._d3Div, props, state);
  }

  componentDidUpdate(prevProps, prevState) {
    var state = this.getd3ReactState();
    this.state.d3React.update(this._d3Div, state);
  }

  componentWillUnmount() {
    this.state.d3React.destroy(this._d3Div);
  }

  render() {
    return (
      <div>
        <h1>{this.props.message}</h1>
        <div className="d3Component" ref={(component) => { this._d3Div = component; } } />
      </div>
    );
  }
}

class d3React {
  constructor() {
    this.create = this.create.bind(this);
    this.update = this.update.bind(this);
    this.destroy  = this.destroy.bind(this);
    this._drawComponent = this._drawComponent.bind(this);
  }

  create(element, props, state) {
    console.log('d3React create');
    var svg = d3.select(element).append('svg')
      .attr('width', props.width)
      .attr('height', props.height);

    this.update(element, state);
  }

  update(element, state) {
    console.log('d3React update');
    this._drawComponent(element, state.data);
```

```
  }

  destroy(element) {
    console.log('d3React destroy');
  }

  _drawComponent(element, data) {
    // perform all drawing on the element here
    var svg = d3.select(element).select('svg');

    svg.append('rect')
      .attr('x', data.x)
      .attr('y', data.y)
      .attr('width', data.width)
      .attr('height', data.height)
      .attr('fill', data.fill);
  }
}

ReactDOM.render(<App message="Hello, D3.js and React!"/>, document.getElementById('app'));
```

Placez le contenu de `d3_react.html` et `d3_react.js` dans le même répertoire et naviguez dans un navigateur Web vers le fichier d3React.html. Si tout se passe bien, vous verrez un en-tête indiquant `Hello, D3.js and React!` rendu du composant React et un rectangle rouge ci-dessous du composant D3 personnalisé.

React utilise des références pour "atteindre" l'instance du composant. Les méthodes de cycle de vie de la classe `d3React` nécessitent cette référence pour ajouter, modifier et supprimer des éléments DOM. La classe `d3React` peut être étendue pour créer plus de composants personnalisés et insérée partout où un `div.d3Component` est créé par React.

D3js avec angulaire

L'utilisation de D3js avec Angular peut ouvrir de nouvelles perspectives, comme la mise à jour en direct des cartes dès la mise à jour des données. Nous pouvons encapsuler une fonctionnalité de graphique complète dans une directive angulaire, ce qui la rend facilement réutilisable.

index.html >>

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  <script src="https://d3js.org/d3.v4.min.js"></script>
  <script data-require="angular.js@1.4.1" data-semver="1.4.1"
src="https://code.angularjs.org/1.4.1/angular.js"></script>
  <script src="app.js"></script>
  <script src="bar-chart.js"></script>
</head>

<body>

  <div ng-controller="MyCtrl">
    <!-- reusable d3js bar-chart directive, data is sent using isolated scope -->
    <bar-chart data="data"></bar-chart>
  </div>
```

```
</body>
</html>
```

Nous pouvons transmettre les données au graphique à l'aide du contrôleur et surveiller tout changement dans les données pour permettre la mise à jour en direct du graphique dans la directive:

app.js >>

```
angular.module('myApp', [])
  .controller('MyCtrl', function($scope) {
    $scope.data = [50, 40, 30];
    $scope.$watch('data', function(newVal, oldVal) {
      $scope.data = newVal;
    }, true);
  });
```

Enfin, la définition de la directive. Le code que nous écrivons pour créer et manipuler le graphique sera placé dans la fonction de lien de la directive.

Notez que nous avons également mis une scope. $ Watch dans la directive pour mettre à jour dès que le contrôleur passe de nouvelles données. Nous réaffectons de nouvelles données à notre variable de données s'il y a un changement de données et appelons ensuite la fonction repaintChart (), qui effectue le rendu du graphique.

bar-chart.js >>

```
angular.module('myApp').directive('barChart', function($window) {
  return {
    restrict: 'E',
    replace: true,
    scope: {
      data: '='
    },
    template: '<div id="bar-chart"></div>',
    link: function(scope, element, attrs, fn) {

      var data = scope.data;
      var d3 = $window.d3;
      var rawSvg = element;

      var colors = d3.scale.category10();

      var canvas = d3.select(rawSvg[0])
        .append('svg')
        .attr("width", 300)
        .attr("height", 150);

      // watching for any changes in the data
      // if new data is detected, the chart repaint code is run
      scope.$watch('data', function(newVal, oldVal) {
        data = newVal;
        repaintChart();
      }, true);
```

```
      var xscale = d3.scale.linear()
        .domain([0, 100])
        .range([0, 240]);

      var yscale = d3.scale.linear()
        .domain([0, data.length])
        .range([0, 120]);

      var bar = canvas.append('g')
        .attr("id", "bar-group")
        .attr("transform", "translate(10,20)")
        .selectAll('rect')
        .data(data)
        .enter()
        .append('rect')
        .attr("class", "bar")
        .attr("height", 15)
        .attr("x", 0)
        .attr("y", function(d, i) {
          return yscale(i);
        })
        .style("fill", function(d, i) {
          return colors(i);
        })
        .attr("width", function(d) {
          return xscale(d);
        });

      // changing the bar widths according to the changes in data
      function repaintChart() {
        canvas.selectAll('rect')
          .data(data)
          .transition()
          .duration(800)
          .attr("width", function(d) {
            return xscale(d);
          })
      }
    }
  }
});
```

Voici le JSFiddle qui fonctionne.

Diagramme D3.js avec Angular v1

HTML:

```
<div ng-app="myApp" ng-controller="Controller">
    <some-chart data="data"></some-chart>
</div>
```

Javascript:

```
angular.module('myApp', [])
.directive('someChart', function() {
    return {
```

```
        restrict: 'E',
        scope: {data: '=data'},
        link: function (scope, element, attrs) {
        var chartElement = d3.select(element[0]);
            // here you have scope.data and chartElement
            // so you may do what you want
        }
    };
});

function Controller($scope) {
    $scope.data = [1,2,3,4,5]; // useful data
}
```

Lire Utiliser D3 avec d'autres frameworks en ligne: https://riptutorial.com/fr/d3-js/topic/3733/utiliser-d3-avec-d-autres-frameworks

Chapitre 12: Utiliser D3 avec JSON et CSV

Syntaxe

- d3.csv (URL [[, ligne], rappel])
- d3.tsv (url [[, ligne], rappel])
- d3.html (url [, callback])
- d3.json (url [, callback])
- d3.text (url [, callback])
- d3.xml (url [, callback])

Examples

Chargement de données à partir de fichiers CSV

Il existe plusieurs façons d'obtenir les données que vous allez lier aux éléments DOM. Le plus simple est d'avoir vos données dans votre script en tant que tableau ...

```
var data = [ ... ];
```

Mais **D3.js** nous permet de charger des données à partir d'un fichier externe. Dans cet exemple, nous verrons comment charger et traiter correctement les données d'un fichier CSV.

Les fichiers CSV sont *des valeurs séparées par des virgules* . Dans ce type de fichier, chaque ligne est un enregistrement de données, chaque enregistrement consistant en un ou plusieurs champs, séparés par des virgules. Il est important de savoir que la fonction que nous allons utiliser, `d3.csv` , utilise la première ligne du `d3.csv` CSV comme en- **tête** , c'est-à-dire la ligne contenant les noms des champs.

Alors, considérez ce CSV, nommé "data.csv":

```
city,population,area
New York,3400,210
Melbourne,1200,350
Tokyo,5200,125
Paris,800,70
```

Pour charger "data.csv", nous utilisons la fonction `d3.csv` . Pour simplifier, supposons que "data.csv" se trouve dans le même répertoire que notre script et que son chemin relatif est simplement "data.csv". Donc, nous écrivons:

```
d3.csv("data.csv", function(data){
    //code dealing with data here
});
```

Notez que, dans le rappel, nous avons utilisé des `data` comme argument. C'est une pratique

courante dans D3, mais vous pouvez utiliser n'importe quel autre nom.

Que fait `d3.csv` avec notre CSV? Il convertit le CSV dans un tableau d'objets. Si, par exemple, nous `console.log` nos données:

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

C'est ce que nous allons voir:

```
[
    {
        "city": "New York",
        "population": "3400",
        "area": "210"
    },{
        "city": "Melbourne",
        "population": "1200",
        "area": "350"
    },{
        "city": "Tokyo",
        "population": "5200",
        "area": "125"
    },{
        "city": "Paris",
        "population": "800",
        "area": "70"
    }
]
```

Nous pouvons maintenant lier ces données à nos éléments DOM.

Notez que, dans cet exemple, la `population` et la `area` sont des chaînes. Mais, probablement, vous voulez les traiter en chiffres. Vous pouvez les changer dans une fonction dans le callback (en tant que `forEach`), mais dans `d3.csv` vous pouvez utiliser une fonction "accessor":

```
d3.csv("data.csv", conversor, function(data){
    //code here
});

function conversor(d){
    d.population = +d.population;
    d.area = +d.area;
    return d;
}
```

Vous pouvez également utiliser des accesseurs dans `d3.tsv` , mais pas dans `d3.json` .

Remarque: `d3.csv` est une fonction asynchrone, ce qui signifie que le code après sera exécuté immédiatement avant même le chargement du fichier CSV. Donc, une attention particulière pour l'utilisation de vos `data` dans le rappel.

Un ou deux paramètres dans le rappel - gestion des erreurs dans d3.request ()

Lorsque vous utilisez `d3.request()` ou l'un des constructeurs de commodité (d3.json , d3.csv , d3.tsv , d3.html et d3.xml), il existe de nombreuses sources d'erreur. Il peut y avoir des problèmes avec la demande émise ou sa réponse en raison d'erreurs de réseau, ou l'analyse peut échouer si le contenu n'est pas bien formé.

Dans les rappels passés à l'une des méthodes mentionnées ci-dessus, il est donc souhaitable de mettre en œuvre une gestion des erreurs. À cette fin, les rappels peuvent accepter deux arguments, le premier étant l'erreur, le cas échéant, le second étant les données. Si une erreur est survenue lors du chargement ou de l'analyse, des informations sur l'erreur seront transmises en tant que première `error` argument `error` les `data` étant `null` .

```
d3.json{"some_file.json", function(error, data) {
  if (error) throw error;   // Exceptions handling
  // Further processing if successful
});
```

Vous n'êtes pas obligé de fournir deux paramètres. Il est parfaitement correct d'utiliser les méthodes de requête avec un rappel ne comportant qu'un seul paramètre. Pour gérer ces types de rappels, il existe une fonction privée `fixCallback()` dans request.js, qui ajuste la façon dont les informations sont transmises à l'argument unique de la méthode.

```
function fixCallback(callback) {
  return function(error, xhr) {
    callback(error == null ? xhr : null);
  };
}
```

Ceci sera invoqué par D3 pour tous les rappels ayant un seul paramètre, qui par définition sont les données.

Quel que soit le nombre de paramètres fournis au rappel de la méthode de requête, la règle pour le paramètre `data` est la suivante:

- si la requête échoue, les `data` seront `null`
- si la demande réussit, les `data` contiendront le contenu chargé (et analysé)

La seule différence entre la version à un paramètre et la version à deux paramètres réside dans la manière dont les informations sont fournies sur l'erreur pouvant survenir. Si le paramètre `error` est omis, la méthode échouera silencieusement, laissant les `data` à la valeur `null` . Si, par contre, le callback est défini pour avoir deux arguments, les informations sur une erreur lors du chargement ou de l'analyse seront transmises au premier paramètre pour vous permettre de le gérer correctement.

Les quatre appels suivants à `d3.json` illustrent les scénarios possibles pour les fichiers existants / non existants par rapport à un paramètre / deux paramètres de rappel:

```
// FAIL: resource not available or content not parsable
// error contains information about the error
// data will be null because of the error
d3.json("non_existing_file.json", function(error, data) {
```

```
  console.log("Fail, 2 parameters: ", error, data);
});

// FAIL: resource not available or content not parsable
// no information about the error
// data will be null because of the error
d3.csv("non_existing_file.json", function(data) {
  console.log("Fail, 1 parameter: ", data);
});

// OK: resource loaded successfully
// error is null
// data contains the JSON loaded from the resource
d3.json("existing_file.json", function(error, data) {
  console.log("OK, 2 parameters: ", error, data);
});

// OK: resource loaded successfully
// no information about the error; this fails silently on error
// data contains the JSON loaded from the resource
d3.json("existing_file.json", function(data) {
  console.log("OK, 1 parameter: ", data);
});
```

Lire Utiliser D3 avec JSON et CSV en ligne: https://riptutorial.com/fr/d3-js/topic/5201/utiliser-d3-avec-json-et-csv

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec d3.js	4444, Adam, altocumulus, Arthur Tarasov, Community, davinov, Fawzan, Gerardo Furtado, Ian H, jmdarling, kashesandr, Marek Skiba, Maximilian Kohl, Rachel Gallen, Ram Visagan, RamenChef, Ruben Helsloot, TheMcMurder, Tim B, winseybash
2	Approches pour créer des graphiques réactifs d3.js	Ian H
3	Cartes SVG utilisant D3 js	rajarshig
4	Concepts de base SVG utilisés dans la visualisation D3.js	fengshuo, Gerardo Furtado
5	Envoi d'événements avec d3.dispatch	aug, kashesandr, Ram Visagan
6	modèle de mise à jour	Gerardo Furtado
7	Projections D3	Andrew Reid
8	Rendre robuste, réactif et réutilisable (r3) pour d3	SumNeuron
9	Sélections	Ashitaka, aug, Carl Manaster, Gerardo Furtado, Ian, Ian H, JulCh, Tim B
10	Sur les événements	aug, Ian H, Kemi, Tim B
11	Utiliser D3 avec d'autres frameworks	Adam, kashesandr, Rishabh
12	Utiliser D3 avec JSON et CSV	altocumulus, Gerardo Furtado