d3-spanish Documentation

Versión 0.0.1

Álvaro Mondéjar Rubio

26 de noviembre de 2018

Conceptos básicos

CAPÍTULO 1

Introducción a D3JS

D3JS es una biblioteca Javascript que permite realizar documentos orientados a datos. Su fuerte es proporcionar maneras de interactuar y visualizar datos usando los navegadores web. A simple vista parece estar orientado a científicos que trabajan sobre datos, pero combinado con el poder de SVG o Canvas puede ser increíblemente útil en la construcción de elementos visuales para aplicaciones web.

- NPM: `npm install d3`

1.1 Historia

D3JS ha sido creado por Mike Bostock, un desarrollador estadounidense especializado en el tratado de datos y su visualización. Trabajó para el New York Times mientras desarrollaba la biblioteca y tiene una vasta cantidad de ejemplos de su uso publicados.

1.2 Consideraciones para novatos

A lo largo de las versiones mayores de D3JS se han introducido diversos cambios no compatibles hacia atrás, los cuales pueden consultarse en el CHANGELOG.

El cambio más drástico ha sido el paso de la versión 3 a la 4.

1.2.1 Estructura del código

A partir de la versión 4, D3JS se ha vuelto una biblioteca modular, es decir, cada módulo se encuentra en un repositorio separado, por lo cual pueden importarse los módulos necesarios para empaquetarlos en un bundle sin depender del resto de códigos no usados.

Por lo tanto si encuentras ejemplos en la web, mira siempre la versión en la que estén diseñados, puesto que si es la versión 3 y estás utilizando la 4, (depende de tu conocimiento de la biblioteca y la complejidad del ejemplo) te puede ser difícil de trascribir y es probable que salten errores.

Nota: Abriendo la consola en cada página de este libro puedes comprobar la versión en la que están desarrollados sus ejemplos.

D3JS ofrece un bundle (archivo) que incluye el núcleo de la biblioteca, que es el que importarás si estás simplemente probando en desarrollo o no usas empaquetado (Webpack, Rollup...). Puedes ver los módulos que incluye el bundle nuclear en el index del repositorio principal. Si encuentras algún ejemplo con módulos no incluídos en el bundle estándar, tendrás que instalarlos o linkearlos por separado.

1.2.2 D3JS es el infierno

Es lo que pensarás al principio y es lo que pensaba yo dándome cabezazos contra los conceptos centrales de la biblioteca. ¿Cómo que puedo crear elementos fantasma con datos que no existen pero que van a entrar? ¿Cómo que los datos van a salir? ¿Por qué todas las funciones se encadenan?

Otra barrera de entrada a esta biblioteca es que debes saber SVG o Canvas, cuanto más mejor. Si estás perdido, échale un ojo a la guía sobre SVG que publiqué hace meses (prometo terminar de incluirla en este libro).

Comprende desde cosas muy básicas como selecciones estilo jQuery a diseños predefinidos donde volcar tus datos los cuales hay que comprender perfectamente antes de ponerse manos a la obra. Lo único que puedo aconsejarte es que estudias bien los ejemplos, luego todo será más sencillo.

Nota: No debes perder nunca de vista la referencia, que está escrita en Markdown en el `README.md` del repositorio de cada módulo.

CAPÍTULO 2

Selecciones — d3-selection

El módulo d3-selection está incluido en la distribución estándar de D3JS y proporciona funciones para seleccionar elementos del DOM. Usa las cadenas de texto de selecciones estandarizadas por la W3C:

```
// Selección de un nodo
var a = d3.select("a");
console.log(a);

// Selección de todos
var lis = d3.selectAll("li");

// Selección anidada
var as = d3.selectAll("li")
    .select("a");
```

Ver también:

- How selections Work - Mike Bostock

Si ves la salida de la consola, puedes comprobar como la selección devuelve un objeto con los atributos `_groups` y `_parents`.

- `_groups`: representa a los grupos de objetos que han sido seleccionados. D3JS almacena agrupados los objetos para ejecutar operaciones sobre ellos. Es una subclase de `Array`.
- `_parents`: representa los padres de los grupos de objetos. Estos dependerán de como se seleccionan. Usando fuciones como `d3.select` en lugar de usar sintaxis encadenada con `.select` sobre otro objeto, devolverá un array con el objeto raíz del documento HTML.

Nota: Con el siguiente código, hemos aumentado el tamaño del título de este capítulo y lo hemos centrado:

```
d3.select("h1")
  .style("font-size", "2em")
  .style("text-align", "center");
```

2.1 Trabajando con datos

D3JS es una biblioteca enfocada a representar datos de forma gráfica. Para ello, cuenta con funciones que están especializadas en enlazar datos y elementos, para ir actualizándolos sincronizados.

2.1.1 Enlazando datos y elementos - selection.data

Jugando con las selecciones podemos hacer lo siguiente:

```
<div id="contenedor-vacio"></div>
<script>
    var spans = d3.select("contenedor-vacio")
        .selectAll("span");

    console.log(spans);
    /**
     * Rt {_groups: Array(0), _parents: Array(0)}
     *   _groups: []
     *   _parents: []
     *   __proto__: Object
     **/
</script>
```

Como puedes ver, devuelve una selección vacía. Esto no parece tener mucho sentido, ¿por qué obtener los elementos `span` de un `div` vacío? Pues en D3JS esto es muy común.

Ver también:

Thinking with joins - Mike Bostock

Supongamos que tenemos los siguientes datos que representan los días de la semana:

```
var data = [
    {day: "Monday", days_until_next_weekend: 5},
    {day: "Tuesday", days_until_next_weekend: 4},
    {day: "Wednesday", days_until_next_weekend: 3},
    {day: "Thursday", days_until_next_weekend: 2},
    {day: "Friday", days_until_next_weekend: 1},
    {day: "Saturday", days_until_next_weekend: 7},
    {day: "Sunday", days_until_next_weekend: 6},
]
```

Podríamos representar esos datos como elementos SVG haciendo lo siguiente:

```
for (var i=0; i<data.length; i++) {
    var g = svg.append("g");
    g.attr("transform", "translate(" + (90*i+30) + "," + height/2 + ")");

    var circle = g.append("circle");
    circle.attr("r", 10);
    circle.attr("fill", "red");
    circle.attr("fill-opacity", 0.2);

    var text = g.append("text");
    text.text(data[i].day);
}
```

Pero, ¿qué pasaría si queremos actualizar esos datos en tiempo real? Tendríamos que atravesar el DOM buscando el dato que queremos y eliminar sus elementos correspondientes. Además, fíjate en la sintaxis. ¿No es demencial?

Ahora, veamos el mismo ejemplo usando D3:

```
var g = svg.selectAll("g")
  .data(data)
  .enter().append("g")
    .attr("transform",function(d, i){
        return "translate(" + (90*i+30) + "," + height/2 + ")";
    })

g.append("circle")
  .attr("r", 10)
  .attr("fill", "red")
  .attr("fill-opacity", 0.2)

g.append("text")
  .text(function(d) { return d.day; });
```

Presta atención a la primera línea: `svg.selectAll("g").` ¡Estamos seleccionando elementos que no existen! Así creamos una selección vacía y en la siguiente línea le pasamos nuestros datos a esa selección con la función selection.data.

Para comprobar lo que hace esta extraña función podemos hacer:

```
var g = svg.selectAll("g").data(data);
console.log(g);

/**
 * Rt {_groups: Array(1), _parents: Array(1), _enter: Array(1), _exit: Array(1)}
 *   _enter: [Array(7)]
 *   _exit: [Array(0)]
 *   _groups: [Array(7)]
 *   _parents: [svg#data2]
 **/
```

Mira que interesante. Tenemos un objeto muy parecido a una selección. Las propiedades `_groups` y `_parents` siguen siendo los grupos de objetos y los padres (en este caso el elemento SVG raíz), que los vimos al principio de este capítulo. Pero ahora han aparecido dos propiedades más: `_enter` y `_exit`:

- `_enter`: Es la propiedad donde se almacena la selección de los datos que han entrado a la selección. Para obtener los elementos de esta selección usamos la función selection.enter.
- `_exit`: Es la propiedad donde se almacena la selección de los datos que existen en los elementos del DOM seleccionados pero ya no existen entre los datos que han entrado a la selección, es decir, correponde a los datos que salen. Para obtener los elementos de esta selección usamos la función selection.exit.

2.1.2 Patrón de actualización

Como dijimos al principio, D3JS no sólo es capaz de enlazar datos a elementos y representarlos, si no que es capaz de actualizarlos en tiempo real. Para ello, es fundamental conocer el patrón de actualización y como implementarlo en D3JS.

Ver también:

- General Update Pattern, I
- General Update Pattern, III

Este sigue los siguientes pasos:

1. Antes de la ejecución del código, no existen elementos en el DOM.
2. Cuando se ejecuta por primera vez una selección vacía con selectAll y le enlazamos datos con selection.data, la selección sigue estando vacía, pero se añaden las propiedades `_enter` y `_exit` y estas esperan a que entren o salgan datos.
3. Cuando se ejecuta la función selection.enter, seleccionamos todos los datos que entraron con selection.data preparados para ser manipulados. En este paso es muy común añadir los elementos correspondientes a los datos al DOM con selection.append o selection.insert.
4. Como estamos en un patrón de actualización, tenemos que ejecutar un timer (ver *Timer — d3-timer*), es decir, tenemos que hacer que se establezca algún tipo de bucle para que se pueda actualizar el contenido de la presentación.
5. A partir de la segunda vez que aplicamos la función selection.data sobre un elemento ocurre lo siguiente:
   - La selección `exit` almacenará los elementos cuyos datos definimos la primera vez pero ahora no se encuentran entre los nuevos datos. Es muy común aplicar sobre ellos la función selection.remove para eliminarlos.
   - La selección `enter` alamacenará los elementos cuyos datos han entrado nuevos, los cuales no existían antes en el DOM.

CAPÍTULO 3

Escalas — d3-scale

Las escalas son abstracciones que nos permiten representar una dimensión de datos abstractos en una representación visual. Pueden representar cualquier codificación visual, tales como divergencia de colores, anchos de trazo o tamaños de símbolos. Dependendiendo del tipo de datos que queramos representar debemos elegir un tipo de escala adecuado.

Las escalas se componen de dos elementos principales: **dominios** y **rangos**.

3.1 Dominios y rangos

Las escalas mapean un dominio de entrada a un rango de salida. Por lo tanto, las funciones de escala toman un intervalo y lo transforman en otro.

```
var x = d3.scaleLinear()
    .domain([10, 130])
    .range([0, 960]);

// Obtener un valor del rango de su equivalente del dominio
console.log(x(20));   // 80
console.log(x(5));    // -40

// Obtener un valor del dominio de su equivalente del rango
console.log(x.invert(960));  // 130
```

Como puedes ver en el ejemplo anterior, el dominio se ha establecido entre `10` y `130` y el rango entre `0` y `960`. Para obtener el valor del rango que equivale a un cierto valor del dominio llamamos a la escala como función pasándole el valor del dominio del cual queremos consultar su equivalencia en el rango. Para el proceso contrario usamos la función `invert()`.

Nota: Para calcular cuantas unidades de rango equivale cada una de dominio: $(range_y - range_x)/(domain_y - domain_x)$.

Las escalas no sólo representan datos matemáticos, si no cualquier rango de matices, como por ejemplo gamas de colores:

```
var color = d3.scaleLinear()
    .domain([0, 100])
    .range(["brown", "steelblue"])
console.log(color(30));  // rgb(137, 68, 83)
```

Ver también:

- Tipos de funciones (Jupyter Notebook)

3.2 Interpoladores

Otra noción básica de las escalas son los interpoladores. En matemáticas la interpolación es la obtención de nuevos puntos partiendo del conocimiento de un conjunto discreto de puntos.

Nota: Aquí puedes ver ejemplos de interpoladores usados en D3JS para mapear escalas de colores y una implementación básica en Python .

- Interpolar (RAE): **Calcular el valor aproximado de una magnitud en un intervalo cuando se conocen algunos de los valores que toma a uno y otro lado de dicho intervalo**.

3.3 Escalas continuas

Las escalas continuas mapean un dominio cuantitativo de entrada a un rango continuo de salida.

3.3.1 Clamping

El clamping o «represión» (en español) es un atributo que está desactivado por defecto en la mayoría de escalas. Al activarlo, no podremos acceder a los valores fuera de rango y dominio: al intentarlo devolverá los valores del extremo.

```
var x = d3.scaleLinear()
    .domain([10, 130])
    .range([0, 960]);

// Clamping desactivado, acceso a los valores externos al mapeo
x(-10);          // -160
x.invert(-160);  // -10

// Activación del clamping
x.clamp(true);
// Ahora no se permite acceder a los valores externos
x(-10);          // 0
x.invert(-160);  // 10
```

3.3.2 Ticks

La función escala_continua.ticks([count]) devuelve aproximadamente `count` valores del dominio de la escala (por defecto `10` si el parametro `count` no es especificado).

```
var x = d3.scaleLinear()
    .domain([0, 100])
    .range([3000, 5000])
x.ticks(5);     // Array [ 0, 20, 40, 60, 80, 100 ]
```

Los valores devueltos están uniformemente espaciados, tienen valores legibles por humanos (como múltiplos de potencias de 10) y se garantiza que estarán dentro de la extensión del dominio. Los ticks son usados a menudo para mostrar líneas de referencia o marcas, en conjunción con los datos visualizados.

3.3.3 Escalas lineales - d3.scaleLinear()

Esta función contruye una nueva escala con dominio y rango `[0, 1]`, el interpolador por defecto y el clamping desactivado. Este tipo de escalas son una buena elección para datos cuantitativos continuos porque estos preservan diferencias proporcionales.

Nota: Cada valor del rango `y` puede ser expresado como una función del valor del dominio `x`: $y = mx + b$.

3.3.4 Escalas exponenciales - d3.scalePow()

Construye una escala continua con dominio y rango `[0, 1]`, exponente `1`, el interpolador por defecto y el clamping desactivado. Esta escala será igual que una escala lineal si mantenemos el exponente a 1. Para cambiarlo podemos usar el método `exponent()`:

```
var x = d3.scalePow()
    .domain([0, 10])
    .range([0, 100])
console.log(x(4)); // 40

x.exponent(2);
console.log(x(4)); // 16
```

Nota: Cada valor del rango `y` puede ser expresado como una función del valor de dominio `x`: $y = mx^k + b$, donde `k` es el valor del exponente.

3.3.5 Escalas logarítmicas - d3.scaleLog()

Las escalas logarítmicas son similares a las escalas lineales, excepto en que aplica una transformación logarítmica es aplicada a los valores dominio de entrada antes de que el los valores del rango de salida sean calculados.

Nota: El mapeo al valor del rango `y` puede ser expresado com una función del valor de dominio `x`: $y = mlog(x) + b$.

3.3.6 Escalas de tiempo - d3.scaleTime()

Las escalas de tiempo son una variante de las escalas lineales que tienen un dominio temporal: los valores de dominio son coercidos a fechas en lugar de números y la función `invert()` devuelve una fecha asimismo. Estas escalas implementan ticks basados en intervalos de calendarios, eliminando el dolor de generar ejes para dominios temporales.

```
var x = d3.scaleTime()      // Year, month, day
    .domain([new Date(2010, 8, 12), new Date(2011, 8, 12)])
    .range([0, 100]);

x(new Date(2010, 11, 12));  // 24.942922374429223
x(new Date(2011, 2, 2));    // 46.86073059360731
x.invert(200);        // Date 2012-09-10T22:00:00.000Z
x.invert(640);        // 2017-02-02T22:00:00.000Z
```

```
var x = d3.scaleTime()
    .domain([new Date(1900, 1, 1), new Date(2000, 1, 1)])
    .range([0, 36500]);
x.ticks(3);  /* Array [ Date 1949-12-31T23:00:00.000Z,
                       Date 1999-12-31T23:00:00.000Z ] */
```

3.4 Escalas secuenciales

Este tipo de escalas son similares a las escalas *Escalas continuas* en que mapean un dominio de entrada numérico a un rango de salida. Sin embargo, a diferencia de las continuas, el rango de salida de una escala secuencial es fijado por su interpolador y no es configurable.

```
var interpolator = function(t){ return t*2 };
var secuencial = d3.scaleSequential(interpolator)
      .domain([1, 100]);
console.log(secuencial(99)); // 1.97979797979798
```

CAPÍTULO 4

Ejes — d3-axis

El componente de D3 `axis` renderiza marcas de referencia para escalas (ver *Escalas — d3-scale*).

4.1 Funciones

D3 provee 4 métodos, con sus nombres indicando su alineamiento, para crear un generador de eje: d3.axisTop([scale]), d3.axisRight([scale]), d3.axisBottom([scale]) and d3.axisLeft([scale]). Un eje alineado arriba (`axisTop`) tiene los ticks dibujados debajo del eje. Un eje alineado abajo (`axisBottom`) es horizontal y tiene sus ticks dibujados debajo del eje. Un eje alineado a la izquierda (`axisLeft`) es vertical y tiene sus ticks alineados a la izquierda del eje, y el eje alineado a la derecha (`axisRight`) en el lado opuesto con los ejes dibujados en el lado exterior.

Todas admiten una escala como primer parámetro, pero esta también puede ser añadida mediante la función axis.scale([scale]).

Para añadir el eje a una selección usamos la función selection.call([axis]).

4.1.1 Paso a paso

Input

```
<!-- Creamos una figura SVG con dimensiones -->
<svg width="100%" height="40">
  <g class="eje">  <!-- Dentro ubicamos un grupo -->
</svg>

<style>
  .eje {
    fill: none;
    stroke: #aaa;
  }
</style>
```

(continué en la próxima página)

(proviene de la página anterior)

```
<script>
  // Creamos una escala
  var scale = d3.scaleLinear()
      .domain([0, 1000])
      .range([0, 600]);

  // Creamos un axis pasándole la escala
  var axis = d3.axisBottom(scale)

  // Seleccionamos el grupo dentro del svg
  d3.select('.eje')  // Lo movemos a la derecha
      .attr("transform", "translate(40, 0)")
      .call(axis);  // Llamamos al eje para insertarlo
</script>
```

Output

Si observamos el código HTML renderizado por D3, veremos que cada tick en el eje es un grupo en SVG con el siguiente código.

```
<g class="tick" opacity="1" transform="translate(240.5,0)">
  <line stroke="#000" y2="6"></line>
  <text fill="#000" y="9" dy="0.71em">400</text>
</g>
```

Nota: El elemento g es un contenedor usado para agrupar objetos. Las transformaciones aplicadas al elemento g son realizadas sobre todos los elementos hijos del mismo. Los atributos aplicados son heredados por los elementos hijos. Además, puede ser usado para definir objetos complejos que pueden luego ser referenciados con el elemento `<use>`.

4.2 Cambiar el color de un eje

Veamos un ejemplo en el que cambiamos el color de un eje, el cual nos servirá para observar más de cerca los elementos HTML renderizados como ejes por D3js.

Input

```
<style>
  .ejeVerde line{
    stroke: green;
  }

  .ejeVerde path{
    stroke: green;
  }

  .ejeVerde text{
    fill: green;
  }
```

(continué en la próxima página)

(proviene de la página anterior)

```
</style>

<div id="container"></div>

<script>
  var scale = d3.scaleLinear()
      .domain([0, 1000])
      .range([0, 600]);

  var axis = d3.axisTop(scale);

  var svg = d3.select("#container")
      .append("svg")
      .attr("width", "100%")
      .attr("height", 40)
    .append("g")
      .attr("class", "ejeVerde")
      .attr("transform", "translate(40, 20)")
      .call(axis)
</script>
```

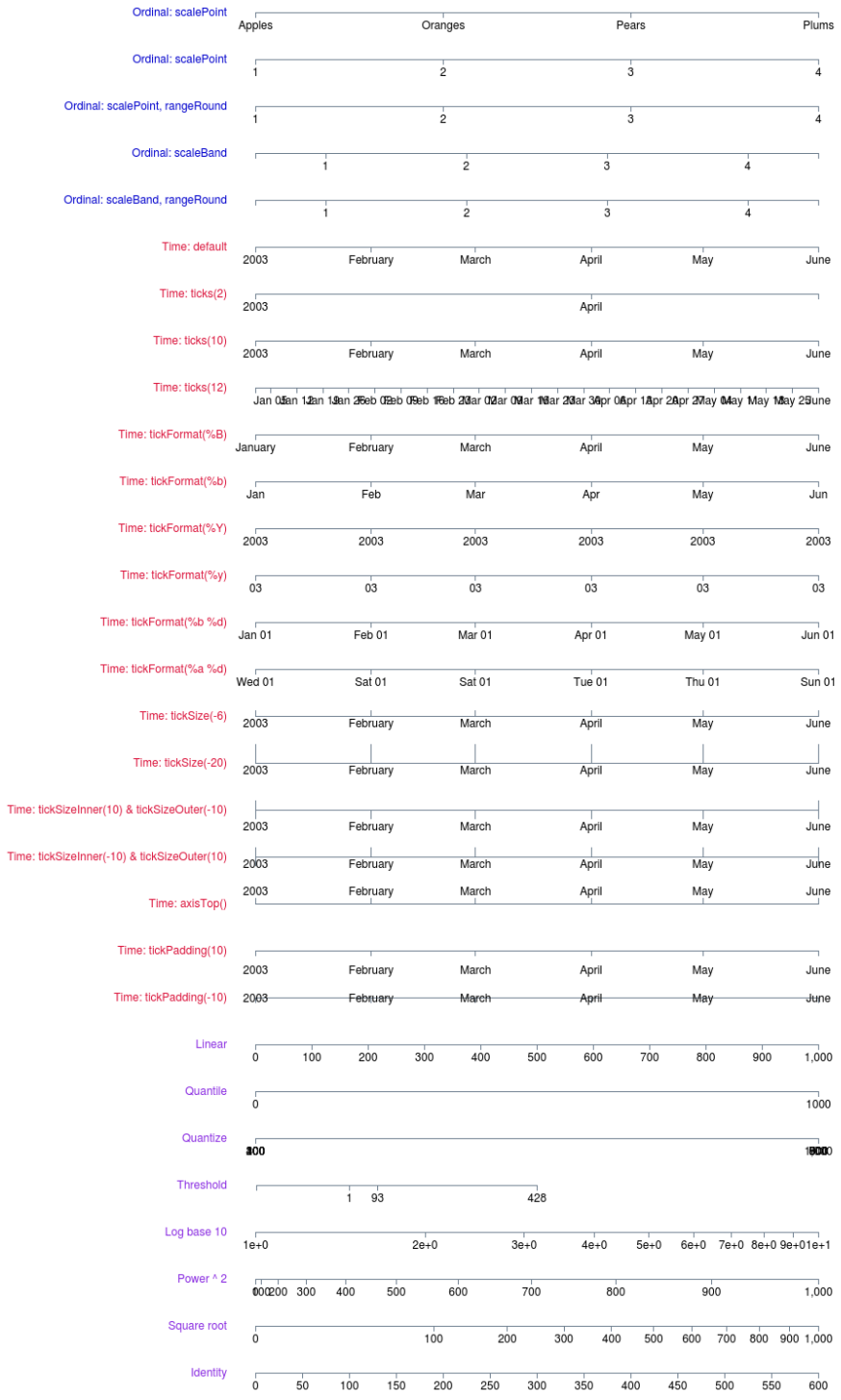
Output

Como puedes observar en el código anterior, debemos establecer 3 propiedades CSS;

- La propiedad `stroke` del elemento `path`. Este elemento se encarga de dibujar la línea horiontal a lo largo de todo el eje.
- La propiedad `stroke` del elemento `line`. Este se encarga de las líneas verticales que van desde el `path` hasta el número.
- La propiedad `fill` del elemento `text`. Este se encarga de los números.

4.3 Ejemplos de escalas

Puedes ampliar la imagen y ver el código fuente que la renderiza en este enlace.


Ordinal: scalePoint
Apples Oranges Pears Plums
Ordinal: scalePoint
1 2 3 4
Ordinal: scalePoint, rangeRound
1 2 3 4
Ordinal: scaleBand
1 2 3 4
Ordinal: scaleBand, rangeRound
1 2 3 4
Time: default
2003 February March April May June
Time: ticks(2)
2003 April
Time: ticks(10)
2003 February March April May June
Time: ticks(12)
Time: tickFormat(%B)
January February March April May June
Time: tickFormat(%b)
Jan Feb Mar Apr May Jun
Time: tickFormat(%Y)
2003 2003 2003 2003 2003 2003
Time: tickFormat(%y)
03 03 03 03 03 03
Time: tickFormat(%b %d)
Jan 01 Feb 01 Mar 01 Apr 01 May 01 Jun 01
Time: tickFormat(%a %d)
Wed 01 Sat 01 Sat 01 Tue 01 Thu 01 Sun 01
Time: tickSize(-6)
2003 February March April May June
Time: tickSize(-20)
2003 February March April May June
Time: tickSizeInner(10) & tickSizeOuter(-10)
2003 February March April May June
Time: tickSizeInner(-10) & tickSizeOuter(10)
2003 February March April May June
Time: axisTop()
2003 February March April May June
Time: tickPadding(10)
2003 February March April May June
Time: tickPadding(-10)
2003 February March April May June
Linear
0 100 200 300 400 500 600 700 800 900 1,000
Quantile
0 1000
Quantize
Threshold
1 93 428
Log base 10
1e+0 2e+0 3e+0 4e+0 5e+0 6e+0 7e+0 8e+0 9e+0 1e+1
Power ^ 2
0 100 200 300 400 500 600 700 800 900 1,000
Square root
0 100 200 300 400 500 600 700 800 900 1,000
Identity
0 50 100 150 200 250 300 350 400 450 500 550 600

CAPÍTULO 5

Escalas de colores

El código común a todas las escalas es el siguiente:

```
var width = 700,
    height = 175,
    lenght = 20;  // Cantidad de colores en cada barra
var unit = width/lenght;  // Ancho de cada color

/**
 * Renderiza una barra de colores en un contenedor SVG
 *   usando un interpolador de D3.
 *
 * @param {string} svgId - Identificador del elemento SVG
 *   donde será renderizada la barra de colores.
 * @param {object} interpolator - Interpolador D3 usado
 *   para construir la escala.
 **/
var renderColorsBar = function(svgId, interpolator) {
    var colorScale = d3.scaleLinear()
        .domain([1, lenght])
        .interpolate(interpolator)
        .range([d3.rgb(color1), d3.rgb(color2)]);

    for (var i = 0; i < lenght; i++) {
       d3.select(svgId)
          .attr("height", height)
          .attr("width", width)
        .append("rect")
          .attr("x",  i*unit )
          .attr("y", 0)
          .attr("width", unit)
          .attr("height", 200)
          .style("fill", colorScale(i));
    }
}
```

5.1 Interpoladores desde d3-interpolate

5.1.1 d3.interpolateRgb(a, b)

Devuelve un interpolador en el espacio de color RGB entre los colores `a` y `b` con un parámetro `gamma` configurable (`1` si no es especificado).

Podemos cambiar el parámetro `gamma` de un interpolador con la función interpolator.gamma(x).

Input

```
<svg id="colors-interpolate-rgb-gamma"></div>

<script>
  var interpolator = d3.interpolateRgb.gamma(2);
  renderColorsBar("#colors-interpolate-rgb-gamma", interpolator);
</script>
```

Output

5.1.2 d3.interpolateHsl(a, b)

Devuelve un interpolador en el espacio de color HSL entre los colores `a` y `b`.

5.1.3 d3.interpolateLab(a, b)

Devuelve un interpolador en el espacio de color Lab entre los colores `a` y `b`.

5.1.4 d3.interpolateHcl(a, b)

Devuelve un interpolador en el espacio de color HCL entre los colores `a` y `b`.

5.1.5 d3.interpolateCubehelix(a, b)

Devuelve un interpolador en el espacio de color Cubehelix entre los colores `a` y `b`.

5.2 Un interpolador de color RGB básico en Python

Para entender claramente lo que hacen los interpoladores, pongamos este sencillo interpolador RGB de ejemplo con Python (el código fuente está sacado de aquí):

```
import string

def make_color_tuple(color):
    """Convierte algo como "#000000" en "0,0,0"
    ó "#FFFFFF" en "255,255,255".
    """
```

(continué en la próxima página)

(proviene de la página anterior)

```
    R = color[1:3]
    G = color[3:5]
    B = color[5:7]

    R = int(R, 16)
    G = int(G, 16)
    B = int(B, 16)

    return R,G,B

def interpolate_tuple( startcolor, goalcolor, steps ):
    """Toma dos colores RGB o los mezcla en
    un número específico de pasos. Devuelve
    la lista de todos los colores generados.
    """
    R = startcolor[0]
    G = startcolor[1]
    B = startcolor[2]

    targetR = goalcolor[0]
    targetG = goalcolor[1]
    targetB = goalcolor[2]

    DiffR = targetR - R
    DiffG = targetG - G
    DiffB = targetB - B

    buffer = []

    for i in range(0, steps +1):
        iR = R + (DiffR * i / steps)
        iG = G + (DiffG * i / steps)
        iB = B + (DiffB * i / steps)

        hR = string.replace(hex(iR), "0x", "")
        hG = string.replace(hex(iG), "0x", "")
        hB = string.replace(hex(iB), "0x", "")

        if len(hR) == 1:
            hR = "0" + hR
        if len(hB) == 1:
            hB = "0" + hB

        if len(hG) == 1:
            hG = "0" + hG

        color = string.upper("#"+hR+hG+hB)
        buffer.append(color)

    return buffer

def interpolate(startcolor, goalcolor, steps):
    """Envoltura para la función ``interpolate_tuple``
    que acepta colores como "#CCCCCC".
    """
    start_tuple = make_color_tuple(startcolor)
    goal_tuple = make_color_tuple(goalcolor)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    return interpolate_tuple(start_tuple, goal_tuple, steps)


def printchart(startcolor, endcolor, steps):
    """Imprime los colores que forman la escala
    en formato exadecimal.

    :param startcolor: Color de comienzo.
    :type startcolor: str

    :param endcolor: Color final.
    :type endcolor: str

    :param steps: Número de pasos de la escala.
    :type steps: int
    """
    colors = interpolate(startcolor, endcolor, steps)

    for color in colors:
        print(color)


# Muestra 16 valores de gradiente entre esos dos colores
printchart("#999933", "#6666FF", 16)
```

CAPÍTULO 6

Timer — d3-timer

6.1 Funciones útiles

6.1.1 d3.now()

Devuelve el tiempo actual en unix timestamps. Esta función asegura consistenca temporal durante el manejo de eventos.

6.2 Temporizadores

6.2.1 d3.timer()

Ejecuta un temporizador, invocando un callback específico repetidamente hasta que el temporizador es apagado. Se puede pasa un numero al parámetro opcional `delay` en milisegundos para invocar al callback después de este tiempo.

Input

```
t = d3.timer(function(elapsed) {
    console.log(elapsed);
    if (elapsed > 300) { t.stop() };
}, 250)
```

Output

```
12
72
94
```

(continué en la próxima página)

(proviene de la página anterior)

```
168
236
252
272
308
```

6.2.2 timer.stop()

Detiene un temporizador, previniendo la invocación de callbacks subsecuentes. No tiene efecto si el temporizador ya ha sido detenido.

6.2.3 timer.restart()

Reinicia un temporizador con un `callback` especificado y con parametros opcionales `delay` y `time`. Esta función es equivalente a para el temporizador y crear uno nuevo con los argumentos provistos, sin embargo este temporizador retiene la prioridad de la invocación anterior.

Input

```
var restarted = false;
var callback = function(elapsed){
    console.log(elapsed);
    if (elapsed > 100) {
        if (restarted == false) {
            restarted = true;
            console.log("");
            t.restart(callback, 200);
        } else {
            t.stop();
        };
    };
};

var t = d3.timer(callback, 200);
```

```
36
84
94
118

52
58
172
```

6.2.4 d3.timeout()

Como la función d3.timer(), con la diferencia de que este temporizador se detiene en el primer callback.

Input

```
var callback_2 = function(elapsed){
    console.log("¡Callback ejecutado!")
}
var t = d3.timeout(callback_2, 200);
```

Output

```
¡Callback ejecutado!
```

6.2.5 d3.interval()

Igual que la función d3.timer() excepto en que el callback es inovocado cada `x` milisegundos definidos en el parámetro `delay`.

Input

```
var callback = function(elapsed){
    console.log(elapsed);
    if (elapsed > 2000) {
        t.stop();
    }
}

var t = d3.interval(callback, 500)
```

Output

```
562
1000
1502
2000
2500
```

CAPÍTULO 7

Números aleatorios — d3-random

7.1 Generadores de números aleatorios

7.1.1 d3.randomUniform([min,][max])

Devuelve una función generadora de números aleatorios con una distribución continua. El valor mínimo del número devuelto será `min` (`0` si no se especifica) y el máximo `max` (`1` si no se especifica).

```
// Compilamos la función generadora
var uniform = d3.randomUniform(1, 10);
// y la ejecutamos.
uniform();  // 5.2935108488665366

// Compilamos y ejecutamos la función generadora
var num = d3.randomUniform(1, 10)();
```

7.1.2 d3.randomNormal([mu][, sigma])

Devuelve una función generadora de números aleatorios con una distribución normal o gaussiana. El valor esperado de los números generados es `mu` (`0` si no se especifica), con la desviación estandar dada por el parámetro `sigma` (`1` si no se especifica).

```
d3.randomNormal(4, 2)();  // 3.2048540025245473
```

CAPÍTULO 8

Formas — d3-shape

8.1 Arcos

8.1.1 d3.arc()

El generador de arcos produce una sección circular o en anillo, como en un gráfico *pie* o *donut*. Si la diferencia entre el ángulo de comienzo y el de final es mayor a 360 grados, el generador de arco producirá un círculo o anillo completo, si no, los arcos pueden tener las esquinas redondeadas y espaciado entre seccionares (padding angular).

Los arcos siempre están centrados en la posición *(0,0)*, por lo que hay que usar el attributo `transform translate(x,y)` para moverlos a una posición diferente.

Los diferentes generadores producen formas que pueden ser pasadas al atributo `d` de un elemento `path` en `svg`. Por ejemplo:

```
<div id="arc-1"></div>
<script>
  var arco_d = d3.arc()
      .innerRadius(0)
      .outerRadius(100)
      .startAngle(0)
      .endAngle(Math.PI);

  arco_d();  // M6.123233995736766e-15,-100A100,100,0,1,1,6.123233995736766e-15,100L0,
↪0Z

  d3.select("#arc-1")
      .append("svg")
        .attr('width', 250)
        .attr('height', 250)
      .append("g")
        .attr("transform", "translate(125,125)")
      .append("path")
        .attr("d", arco_d)
```

(continué en la próxima página)

(proviene de la página anterior)

```
        .attr("fill", "red");
</script>
```

8.2 Areas

Para generar un area en un chart hemos de escribir algo como:

Input

```
<div id="area-1"></div>
<script>
  var margin = {top: 25, bottom: 25}
      width = 680,
      height = 340 - margin.top - margin.bottom;

  var svg = d3.select("#area-1").append("svg")
      .attr("width", width)
      .attr("height", height + margin.top + margin.bottom);

  var data = [1, 3, 2, 3, 5, 8, 4, 9];

  yScale = d3.scaleLinear()
      .domain(d3.extent(data))
      .range([0, height]);

  var area = d3.area()
      .x(function(d, i){ return (width/data.length)*i; })
      .y0(height)
      .y1(function(d){ return height-yScale(d); })
      .curve(d3.curveMonotoneX);

  var path = svg.append("path")
      .attr("d", area(data))
      .style("fill", "lightsteelblue");
</script>
```

Output

La función d3.area() crea un generador de areas. La línea de arriba se define por los métodos area.x1([x]) y area.y1([y]), y es renderizada primero. La línea de abajo se define por los métodos area.x0([x]) y area.y0([y]). Las funciones area.x([x]) y area.y([y]) establecen los parámetros `0` y `1` de cada coodenada a un mismo valor, es decir:

```
// Esto es lo mismo...
.x(function(d, i){ return (width/data.length)*i; })

// ...que esto
.x0(function(d, i){ return (width/data.length)*i; })
.x1(function(d, i){ return (width/data.length)*i; })
```

8.3 Links

La forma de **link** genera una curva Bézier cúbica suave desde un punto hasta otro.

Para incluir un link con un punto inicial y destino indicados manualmente, primero inicializamos un generador `link` que puede ser vertical, horizontal o radial y luego pasamos a la función un objeto con los atributos `source` y `target` donde especificamos las coordenadas de comienzo y final de la línea por medio de arrays numéricos de dos elementos:

```
var link = d3.linkVertical();
var path = svg.append("path")
    .attr("d", link({
        source: [50, 50],
        target: [150, 150]
    }));
```

8.3.1 d3.linkVertical()

Genera una curva Bézier cúbica suave con tangentes verticales.

Input

```
<div id="link-vertical-1"></div>
<script>
var width = 300,
    height = 200;

var svg = d3.select("#link-vertical-1").append("svg")
    .attr("width", width)
    .attr("height", height);

var link = d3.linkVertical();
var path = svg.append("path")
    .attr("d", link({
        source: [50, 50],
        target: [150, 150]
    }))
    .style("stroke", "navy")
    .style("stroke-width", "3px")
    .style("fill", "none");
</script>
```

Output

8.3.2 d3.linkHorizontal()

Genera una curva Bézier cúbica suave con tangentes horizontales.

Input

```
<div id="link-horizontal-1"></div>
<script>
  var width = 300,
      height = 200;

  var svg = d3.select("#link-horizontal-1").append("svg")
      .attr("width", width)
      .attr("height", height);

  var link = d3.linkHorizontal();
  var path = svg.append("path")
      .attr("d", link({
          source: [50, 50],
          target: [150, 150]
      }))
      .style("stroke", "royalblue")
      .style("stroke-width", "3px")
      .style("fill", "none");
</script>
```

Output

CAPÍTULO 9

Mapas — d3-geo

Para generar mapas en D3 seguimos los siguientes pasos:

Generamos un objeto `path` con la función d3.geoPath() añadiéndole una proyección de las tantas existentes en d3-geo. Un ejemplo sería:

```
var path = d3.geoPath().projection(geoAzimuthalEquidistant());
```

O también:

```
var projection = geoAzimuthalEquidistant(),
    path = d3.geoPath(projection);
```

Podemos optar por renderizar el mapa con `svg` o con `canvas`, vamos a explorar ambas opciones.

9.1 SVG

Obtenemos el elemento `svg` donde queremos representar el mapa. A este le añadimos los datos con data() o datum() y mediante el atributo `"d"` insertamos el `path` que hemos creado anteriormente:

```
<svg width="100%" height="500" id="example_map"></svg>
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.13.0/d3.min.js"></script>
<script src="https://unpkg.com/topojson-client@3"></script>

<script>
  var url = "https://unpkg.com/world-atlas@1.1.4/world/110m.json";
  d3.json(url, function(error, world){
      if (error) throw error;

      var svg = d3.select("#example_map")
        .datum(topojson.feature(world, world.objects.land))
        .attr("d", path);
  })
</script>
```

Nota: Lee este artículo para entender la diferencia entre data() o datum().

Input

```
<style>
#map-1 {
  background-color: skyblue;
}
.province {
  fill: royalblue;
  stroke: #FFF;
  stroke-width: .5px;
  stroke-dasharray: 3 3;
}

.province:hover {
  fill: crimson;
  stroke: orange;
  stroke-width: 2px;
  stroke-dasharray: none;

}

.graticule {
  fill: none;
  stroke: #FFF;
  stroke-width: .6px;
  stroke-opacity: 0.5;
}
</style>

<svg id="map-1" width="748" height="500"></svg>

<script>
  var svg = d3.select("#map-1"),
      width = +svg.attr("width"),
      height = +svg.attr("height");

  // Creamos la proyección  (ver Proyecciones abajo)
  var projection = d3.geoMercator()
    .scale(2200)
    .center([0, 40])
    .translate([width / 1.7, height / 2]);
    //.translate([350, 200]);          // Otros atributos
    //.rotate([122.4194, -37.7749])
    //.clipAngle(180 - 1e-3)
    //.precision(0.1);

  // Creamos el path añadiendo la proyección
  var path = d3.geoPath(projection),

  // Creamos una rejilla que se repita cada 2 grados tanto
  //   en direcciones norte-sur como este-oeste
  var graticule = d3.geoGraticule().step([2, 2]);
```

(continué en la próxima página)

(proviene de la página anterior)

```
  // Añadimos la rejilla
  svg.append("path")
      .datum(graticule)
      .attr("class", "graticule")
      .attr("d", path);

  // Obtenemos las provincias de España en formato geojson
  var url = "https://raw.githubusercontent.com/codeforamerica/click_that_hood/master/
↪public/data/spain-provinces.geojson";
  d3.json(url, function(error, spain){
      if (error) throw error;  // Manejamos cualquier posible error

      var group = svg.selectAll("g")  // Creamos un grupo para cada provincia
          .data(spain.features)
          .enter()
          .append("g");

      // Para cada grupo añadimos el path correspondiente
      var areas = group.append("path")
          .attr("d", path)
          .attr("class", "province");

  });
</script>
```

Output

Como puedes observar en este ejemplo, básicamente son 6 pasos.

1. Crear una proyección (ver *Proyecciones*).
2. Crear un `path` con la función d3.geoPath() y añadirle la proyección.
3. [Opcional] Crear una rejilla con la función d3.geoGraticule(). Añadir un elemento `path` al svg, enlazar los datos de la cuadrícula con la función datum() y añadir el `path` del paso anterior al atributo `d`.
4. Obtener los datos geográficos del mapa.
5. Enlazar los datos al contenedor `svg` por medio de grupos.
6. Añadir a cada grupo un elemento `path`, cuyo atributo `d` será el path que hemos creado en el paso 2.

9.2 Canvas

Para crear un mapa usando elementos `canvas` habría que escribir algo como esto:

Input

```
<canvas width="680" width="480"></canvas>

<script src="https://unpkg.com/topojson-client@3"></script>
<script>
```

(continué en la próxima página)

(proviene de la página anterior)

```
  var context = d3.select("canvas").node().getContext("2d");

  var projection = d3.geoMercator()
    .scale(2200)
    .center([0, 40])
    .translate([width / 1.7, height / 2]);

  var path = d3.geoPath(projection).context(context);

  var url = "https://raw.githubusercontent.com/codeforamerica/click_that_hood/master/
↪public/data/spain-provinces.geojson";
  d3.json(url, function(error, spain) {
      if (error) throw error;

      // Comenzamos a dibujar en el lienzo
      context.beginPath();

      // Añadimos el path con los datos del archivo .json
      path(topojson.mesh(spain));

      // Dibujamos el contenido
      context.stroke();
  });

</script>
```

Output

En este ejemplo, mucho más simple, hemos usado topojson para cargar los datos del archivo en json e insetarlos en el canvas.

Ver también:

- context.beginPath()
- context.stroke()
- topojson.mesh(topology [, object[, filter]])

9.3 Proyecciones

D3 provee muchas proyecciones dentro de los módulos d3-geo y d3-geo-projection. Cada proyección puede ser controlada mediante métodos.

9.3.1 Métodos

- projection.scale([scale]): El factor de escalado corresponde linealmente a la distancia entre los puntos projectados; sin embargo, mismos factores de escalado no son equivalentes entre diferentes proyecciones.
- projection.center([center]): Array de dos elementos con las coordenadas longitud y latitud en grados (por defecto `[0, 0]`).

- projection.translate([translate]): si se especifica el parámetro `translate`, el la proyección será trasladada en los ejes `[x, y]` según los valores introducidos (por defecto `[480, 250]`).
- projection.rotate([angles]): Si él parámetro `angles` es introducido, establece la rotación esférica sobre los tres ejes a los ángulos especificado, el cual debe ser un array de dos ó tres números `[lambda, phi, gamma]` , especificando los ángulos de rotación en grados sobre cada eje esférico (por defecto `[0, 0, 0]`.

9.4 Rejillas

9.4.1 d3.geoGraticule()

Con esta función creamos una rejilla. Para añadir una rejilla simplemente hemos de insertarla en el contenedor `svg` con la función datum():

```
var graticule = d3.geoGraticule().step([2, 2]);

svg.append("path")
    .datum(graticule)
    .attr("d", path);
```

9.4.2 Métodos

- graticule.step([step]): Acepta un array de dos números que indican los grados de distancia entre cada filamento de la rejilla. El primero indica diferencia entre longitudes y el segundo latitudes.

CAPÍTULO 10

Layouts

10.1 De jerarquía - d3-hierarchy

10.1.1 Arbol

Los conjuntos de datos en árbol (anidados unos datos dentro de otros) son perfecto para el tipo de layout de la función d3.tree().

Input

```
<svg width="600" height="600"></svg>

 <script>
   var data = {
     "name": "Max",
     "children": [
       {
         "name": "Carlos",
         "children": [
           {"name": "Carla"},
           {"name": "Eusebio"},
           {"name": "Alicia"}
        ]
       },
       {
         "name": "Joana",
         "children": [
           {"name": "Cristina"},
           {"name": "Julián"},
           {"name": "Recaredo"}
         ]
       }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ]
};

var svg = d3.select("svg"),
    width = +svg.attr("width"),     // La anotación con + extrae
    height = +svg.attr("height"),  // el valor del atributo
    g = svg.append("g").attr("transform", "translate(0,40)");

// Parseamos los diccionarios anidados y extraemos el nodo padre
var root = d3.hierarchy(data)

// Creamos un layout de árbol
var tree = d3.tree().size([width-30, height-160]);
tree(root);    // Insertamos la jerarquía en el árbol

// Creamos los enlaces entre los elementos
var link = g.selectAll(".link")
    .data(root.links())      // Pasamos como dato los enlaces pade-hijo
    .enter().append("line")  // Para cada enlace una linea...
        .attr("class", "link")
        .attr("stroke-width", "2px")
        .attr("stroke", "#ddd")
        .attr("x1", function(d) { return d.source.x; })
        .attr("y1", function(d) { return d.source.y; })
        .attr("x2", function(d) { return d.target.x; })
        .attr("y2", function(d) { return d.target.y; });

// Creamos nodos para los elementos del árbol
var node = g.selectAll(".node")
    .data(root.descendants())  // Accedemos a los descendientes del árbol
    .enter()
      .append("g")      // Para cada nodo creamos un grupo
        .attr("transform", function(d) {
            return "translate(" + d.x + "," + d.y + ")";
        })

// En cada nodo añadimos lo que queramos
node.append("circle")
  .attr("r", 2.5);

node.append("text")
  .text(function(d) { return d.data.name });
</script>
```

Output

La función d3.hierarchy(data[, children]) extrae el nodo padre de un conjunto de datos anidados.

CAPÍTULO 11

Animaciones

11.1 Transiciones – d3-transition

Para crear transiciones usamos la función selection.transition() sobre una selección.

Input

```
<style>
  #contenedor1 {
    height: 100px;
    width: 600px;
    margin: 12px auto;
    background-color: gold;
  }
</style>

<div id="contenedor1"></div>

<script>
  d3.select("#contenedor1")
    .transition()      // Creamos una transición
    .duration(3000)    // - Tiempo que tardará
    .delay(2000)       // - Tiempo en espera antes de activar la transición
      .style("background-color", "navy");
</script>
```

Output

11.2 Desenvoltura de transiciones – d3-ease

Podemos definir como se comportan las transiciones mediante diferentes funciones que flexibilizan el movimiento de transición. Simplemente la aplicamos mediante la función transition.ease() a una transición (por defecto easeCubicInOut()).

Los nombres de las diferentes funciones se pueden observar en el explorador de funciones, además de en la documentación de d3-ease.

Input

```
<style>
  #contenedor2 {
    height: 100px;
    width: 600px;
    margin: 12px auto;
    background-color: dodgerblue;
  }
</style>
<div id="contenedor2"></div>

<script>
  d3.select("#contenedor2")
      .transition()
      .duration(4000)
      .delay(4000)
      .ease(d3.easeElasticOut)   // Indicamos la función de transición flexible
        .style("width", "200px");
</script>
```

Output

CAPÍTULO 12

Polígonos — d3-polygon

El módulo d3-polygon ofrece algunas operaciones que se pueden efectuar sobre polígonos de dos dimensiones. Cada polígono está representado por un array de arrays de dos elementos y pueden ser cerrados (si el primer y el último punto son el mismo) o abiertos. Típicamente, el orden de los puntos están dados en dirección a las agujas del reloj donde el origen `(0, 0)` se sitúa en la esquina izquierda-arriba de la pantalla.

```
.. raw:: html

<script>
  var puntos = [
      [10, 10],
      [30, 10],
      [50, 40],
      [45, 60],
      [15, 35],
      [10, 10]
  ];

  // Definimos un polígono a partir de sus puntos ("hull" == cáscara)
  var poligono = d3.polygonHull(puntos);
  console.log(poligono);   // [Array(2), Array(2), Array(2), Array(2), Array(2)]

  // Obtenemos el área del polígono
  var area = d3.polygonArea(poligono);
  console.log(area);       // 1087.5

  // Obtenemos el perímetro del polígono
  var perimetro = d3.polygonLength(poligono);
  console.log(perimetro);  // 141.2173868302254

  // Obtenemos el centro del polígono
  var centro = d3.polygonCentroid(poligono);
  console.log(centro);     // [30.268199233716476, 31.590038314176244]

  // Comprobamos si un punto está dentro del polígono
```

(continué en la próxima página)

(proviene de la página anterior)

```
  var contiene = d3.polygonContains(poligono, (35, 10));
  console.log(contiene);   // false
</script>
```

CAPÍTULO 13

Gráficos

13.1 Gráficos de lineas

Veamos un ejemplo sacado de aquí, en el cual creamos paso a paso un gráfico simple de una línea. Observa como varios componentes de la biblioteca D3js trabajan en conjunto.

Primero creamos un contenedor donde alojar el gráfico y el estilo de la línea.

```
<div id="container"></div>

<style>
  .line {
    fill: none;
    stroke: steelblue;
    stroke-width: 2px;
  }
</style>
<script src="path/to/create_line_graph.js">
```

Siempre, al realizar figuras, es útil establecer los márgenes, ancho y alto al principio.

create_line_graph.js

```
// Establecemos las dimensiones y los márgenes del gráfico
var margin = {top: 30, right: 20, bottom: 20, left: 20},
    width = 680 - margin.left - margin.right,
    height = 315 - margin.top - margin.bottom;

// Establecemos las escalas y sus rangos a lo largo de los ejes x y
var x = d3.scaleTime().range([0, width]);
var y = d3.scaleLinear().range([height, 0]);

// Creamos la figura svg
```

(continué en la próxima página)

(proviene de la página anterior)

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

// Creamos una línea que más tarde cargaremos en el path
var valueline = d3.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

// Compilamos un parser para las fechas (por ejemplo: 1-May-12)
var parseTime = d3.timeParse("%d-%b-%y");

// Creamos el eje X, formateando las fechas
var xAxis = d3.axisBottom(x)
    .tickFormat(d3.timeFormat("%Y-%m-%d"));

// Obtenemos los datos
var url = "https://gist.githubusercontent.com/d3noob/402dd382a51a4f6eea487f9a35566de0/
↪raw/6369502941b44261f381399a24fb455cb4290be8/data.csv";
d3.csv(url, function(error, data) {
    if (error) throw error;

    // Los formateamos
    data.forEach(function(d) {
        console.log(d);
        d.date = parseTime(d.date);
        d.close = +d.close;
    });

    // Escalamos el rango de los datos
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    // Añadimos y estilizamos la línea
    svg.append("path")
        .data([data])
        .attr("class", "line")
        .attr("d", valueline);

    // Añadimos los ejes
    svg.append("g")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis)
        .selectAll("text")
          .style("text-anchor", "end")
          .attr("dx", "-.8em")
          .attr("dy", ".15em")
          .attr("transform", function(d) {
              return "rotate(-40)"
              });

    svg.append("g")
        .call(d3.axisLeft(y));
  });
```

Output

13.2 Gráficos de barras

13.2.1 Diferencias entre un gráfico mediante `div` y `svg`

En los siguientes dos ejemplos vamos a realizar el mismo gráfico de dos formas diferentes: la primera añadiendo dinámicamente elementos `div` a un contenedor padre y la segunda mediante elementos `svg`.

El código común a ambos ejemplos es:

```
var margin = {top: 20, right: 20, bottom: 20, left: 20},
    width = 680 - margin.right - margin.left,
    height = 680 - margin.top - margin.bottom;

var data = [
    {coin: "BTC", value: 10000},
    {coin: "LTC", value: 5000},
    {coin: "ETH", value: 3000},
    {coin: "XMR", value: 1500},
    {coin: "BCH", value: 1000},
    {coin: "BTZ", value: 400}
]

scaleX = d3.scaleLinear()
  .domain([0, d3.max(data, function(d){ return d.value })])
  .range([0, width - margin.right - margin.left])
```

Mediante elementos `div`

A continuación puedes observar un ejemplo simple de creación de un gráfico de barras añadiendo contenedores dentro de otro contenedor.

Input

```
<style>
  #chart-1 div {
    font: 10px sans-serif;
    background-color: steelblue;
    text-align: right;
    padding: 3px;
    margin: 1px;
    color: white;
  }
</style>
<div id="chart-1" style="margin-bottom:50px;"></div>
<script>
  d3.select("#chart-1")
    .selectAll("div")
      .data(data)
    .enter().append("div")
      .style("width", 0)
      .text(function(d) { return d.value })
```

(continué en la próxima página)

(proviene de la página anterior)

```
    .transition().duration(3000)
      .style("width", function(d){ return scaleX(d.value) + "px" });

</script>
```

Output

Mediante elementos SVG

El movimiento de las barras es más fluido y el gráfico tarda menos en cargar si lo realizamos mediante SVG.

Input

```
<style>
  #chart-2 rect {
    fill: steelblue;
  }

  #chart-2 text {
    fill: white;
    font: 10px sans-serif;
    text-anchor: end;
  }
</style>

<svg id="chart-2" style="margin-bottom:30px;"></div>

<script>
  var barHeight = 20;

  var chart = d3.select("#chart-2")
    .attr("width", width)
    .attr("height", barHeight * data.lenght);

  var bar = chart.selectAll("g")
    .data(data)
    .enter().append("g")
    .attr("transform", function(d, i) { return "translate(0," + i * barHeight + ")"; }
↪);

  bar.append("rect")
      .attr("width", 0)
    .transition().duration(2000)
      .attr("width", function(d){ return scaleX(d.value) + "px"; })
      .attr("height", barHeight-1);

  bar.append("text")
      .attr("x", 0)
    .transition().duration(2000)
      .attr("x", function(d) { return scaleX(d.value) - 3; })
      .attr("y", barHeight / 2)
      .attr("dy", ".35em")
      .text(function(d) { return d.value; });
</script>
```

Output

Gráfico de barras con datos reales

Input

```
<style>
  #chart-3 rect {
    fill: teal;
  }

  #chart-3 text {
    fill: teal;
    font: 10px sans-serif;
    text-anchor: end;
  }
</style>

<svg id="chart-3"></svg>

<script>
  var limit = 20;
  var url = "https://api.coinmarketcap.com/v2/ticker/?limit=" + limit;

  var barHeight = 20;

  d3.json(url, function(error, res) {
    if (error) { throw error }

    var data = [];
    for (var id in res["data"]) {
      data.push({
        symbol: res["data"][id]["symbol"],
        value: res["data"][id]["quotes"]["USD"]["price"]
      })
    }

    scaleX = d3.scaleLinear()
      .domain([0, d3.max(data, function(d){ return d.value + 2000; })])
      .range([0, width])

    var chart = d3.select("#chart-3")
      .attr("width", width)
      .attr("height", barHeight * data.length);

    var bar = chart.selectAll("g")
        .data(data)
      .enter().append("g")
        .attr("transform", function(d, i) { return "translate(0," + i * barHeight + ")
↪"; });

    bar.append("rect")
        .attr("width", function(d){ return scaleX(d.value) + "px"; })
        .attr("height", barHeight-1);

    bar.append("text")
      .attr("x", function(d) { return (scaleX(d.value) + 105) + "px"; })
```

(continué en la próxima página)

(proviene de la página anterior)

```
      .attr("y", barHeight / 2)
      .attr("dy", ".35em")
      .text(function(d) { return  d.symbol + " --- " + d.value + " $"; });

  });
</script>
```

Output

13.3 Gráficos de torta (pie)

Si quisieramos realizar un gráfico de torta usando la función d3.arc() deberiamos crear varios arcos con valores de comienzo y final de ángulos diferentes, calculando el porcentaje en grados de cada dato con respecto al total, redefiniendo los lugares de aparición de cada trozo... una aproximación nada práctica.

Por ello d3-shape ofrece la función d3.pie() la cual ayuda mucho en este cometido.

13.3.1 d3.pie()

Esta función no produce una forma directamente, si no que calcula los ángulos necesarios para representar un conjunto de datos tabulado como un gráfico de torta o de donut. Esos ángulos pueden ser pasados a un generador de arco (d3.arc()).

Input

```
var data = [1, 1, 2, 3, 5, 8, 13, 21];
var arcs = d3.pie()(data);
console.log(arcs);
```

Output

```
[
  {"data":  1, "value":  1, "index": 6, "startAngle": 6.050474740247008, "endAngle":␣
↪6.166830023713296, "padAngle": 0},
  {"data":  1, "value":  1, "index": 7, "startAngle": 6.166830023713296, "endAngle":␣
↪6.283185307179584, "padAngle": 0},
  {"data":  2, "value":  2, "index": 5, "startAngle": 5.817764173314431, "endAngle":␣
↪6.050474740247008, "padAngle": 0},
  {"data":  3, "value":  3, "index": 4, "startAngle": 5.468698322915565, "endAngle":␣
↪5.817764173314431, "padAngle": 0},
  {"data":  5, "value":  5, "index": 3, "startAngle": 4.886921905584122, "endAngle":␣
↪5.468698322915565, "padAngle": 0},
  {"data":  8, "value":  8, "index": 2, "startAngle": 3.956079637853813, "endAngle":␣
↪4.886921905584122, "padAngle": 0},
  {"data": 13, "value": 13, "index": 1, "startAngle": 2.443460952792061, "endAngle":␣
↪3.956079637853813, "padAngle": 0},
  {"data": 21, "value": 21, "index": 0, "startAngle": 0.000000000000000, "endAngle":␣
↪2.443460952792061, "padAngle": 0}
]
```

Ejemplo

Input

```
<style>
  .arc text {
    font: 10px sans-serif;
    text-anchor: middle;
  }
  .arc path {
    stroke: teal;
    stroke-width: 3.5px;
  }
</style>

<div id="pie-chart-1"></div>

<script>
  var data = [10, 20, 100];

  var width = 748,
      height = 530,
      radius = Math.min(width, height) / 2;

  var color = d3.scaleOrdinal()
      .range(["#98abc5", "#8a89a6", "#7b6888"]);

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

  var labelArc = d3.arc()
      .outerRadius(radius - 50)
      .innerRadius(radius - 50);

  var pie = d3.pie()
      .sort(null)
      .value(function(d) { return d; });

  var svg = d3.select("#pie-chart-1").append("svg")
      .attr("width", width)
      .attr("height", height)
    .append("g")
      .attr("transform", "translate(" + width / 2 + "," + height / 2 + ")");

    var g = svg.selectAll(".arc")
        .data(pie(data))
      .enter().append("g")
        .attr("class", "arc");

    g.append("path")
        .attr("d", arc)
        .style("fill", function(d) { return color(d.data); });

    g.append("text")
        .attr("transform", function(d) { return "translate(" + labelArc.centroid(d) +
↪")";   })
```

(continué en la próxima página)

(proviene de la página anterior)

```
        .attr("dy", ".35em")
        .text(function(d) { return d.data; });
</script>
```

Output

Para realizar un gráfico de torta en D3 seguimos los siguientes pasos:

1. Realizamos un mapeo de colores a los datos. En el ejemplo anterior puedes ver un mapeo de 1 a 1, ya que hay el mismo número de datos que de colores, pero el mapeo puede ser un dominio numérico, por ejemplo, en una escala de colores más claros a oscuros.
2. Creamos las formas circulares tanto del gráfico en sí como de la posición de las marcas de texto con la función d3.arc(), estableciendo las propiedades innerRadius y outerRadius.
3. Creamos un generador de gráfico de torta con la función d3.pie(), al cual podemos pasarle un ordenamiento con el método sort(). Además, con value() establecemos como se distribuirán las cantidades para generar los ángulos correspondientes a cada porción de los datos.
4. Renderizamos las porciones en el contenedor `svg` pasando los datos a través del generador `pie` con `.data(pie(data))`. Cada porción la incluimos en un grupo.
5. Para cada grupo insertamos los paths correspondientes a las etiquetas y a la propia porción definidos en el paso 2 mediante `.attr("d", arc)`.

13.4 Gráficos de dispersión

13.4.1 Iris dataset

En este ejemplo hemos usado la escala de colores d3.schemeDark2, por lo que hemos de cargar el módulo d3-scale-chromatic con:

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

Input

```
<div id="scatterplot-graph-1"></div>
<script>
  var margin = {top: 30, right: 50, bottom: 40, left: 40};
  var width = 680 - margin.left - margin.right;
  var height = 450 - margin.top - margin.bottom;

  var svg = d3.select("#scatterplot-graph-1").append("svg")
      .attr("width", width + margin.left + margin.right)
      .attr("height", height + margin.top + margin.bottom)
    .append("g")
      .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

  var xScale = d3.scaleLinear().range([0, width]);
  var yScale = d3.scaleLinear().range([height, 0]);

  var xAxis = d3.axisBottom().scale(xScale);
```

(continué en la próxima página)

(proviene de la página anterior)

```
  var yAxis = d3.axisLeft().scale(yScale);

  var color = d3.scaleOrdinal(d3.schemeDark2);

  var url = "https://gist.githubusercontent.com/Jverma/
↪076377dd0125b1a508621441752735fc/raw/b3c1e1bdafd135d6cd01f5a4b53f1bd347dacd03/iris.
↪csv";

  d3.csv(url, function(error, data){
      xScale.domain(d3.extent(data, function(d){
          return d.SepalLength;
      })).nice();

      yScale.domain(d3.extent(data, function(d){
          return d.SepalWidth;
      })).nice();

      svg.append("g")
          .attr("transform", "translate(0," + height + ")")
          .call(xAxis);

      svg.append("g")
          .attr("transform", "translate(0,0)")
          .call(yAxis);

      var bubble = svg.selectAll(".bubble")
          .data(data)
          .enter().append("circle")
          .attr("cx", function(d){ return xScale(d.SepalLength); })
          .attr("cy", function(d){ return yScale(d.SepalWidth); })
          .attr("r", "3px")
          .style("fill", function(d){ return color(d.Name); });
  });
</script>
```

Output

CAPÍTULO 14

SVG

14.1 Filtros SVG

14.1.1 Básico

Para crear un filtro con SVG debemos crear un elemento `defs` donde definimos todos los filtros que más tarde podemos aplicar a los elementos referenciándolos. Dentro de estas etiquetas, creamos filtros con la etiqueta `filter` a la cual es importante otorgar un identificador para luego poder referenciarlos.

El siguiente código es común a todos los ejemplos:

```
var margin = {top: 30, right: 20, bottom: 20, left: 20},
    width = 680 - margin.left - margin.right,
    height = 400 - margin.top - margin.bottom;

function createSVG(id){
    return d3.select(id)
      .attr("height", height + margin.left + margin.right)
      .attr("width", width + margin.bottom + margin.top);
}

function appendCircleToSVG(svg, cx, cy, r, fill){
    return svg.append("circle")
      .attr("cx", cx)
      .attr("cy", cy)
      .attr("r", r)
      .attr("fill", fill);
}
```

14.1.2 Filtros de movimiento

feOffset - Desplazar elementos

El efecto feOffset nos permite desplazar el elemento a través de la imagen, lo que lo hace perfecto, combinado con un filtro de distorsión como el gaussiano, para crear sombras.

Ejemplo básico

En el siguiente ejemplo ambos círculos se encuentran originalmente en la misma posición pero el azul ha sido movido `200px` hacia la derecha por un filtro feOffset. Observa que el filtro sólo actuará en la zona de incluencia definida por sus atributos `width` and `height`.

Input

```
<svg width="100%" height="200">
  <defs>
    <filter id="uno" width="300%" height="120%">
      <feOffset dx="100" dy="0">
    </filter>
  </defs>

  <circle cx="100" cy="100" r="75" fill="navy" filter="url(#uno)"></circle>
  <circle cx="100" cy="100" r="75" fill="teal"></circle>
</svg>
```

Output

Input- Versión D3

```
<div id="container-offset-1"></div>
<script>
  var svg = d3.select("#container-offset-1")
      .append("svg")
    .attr("width", "100%")
    .attr("height", 200);

  var defs = svg.append("defs");

  var filter_id = "offset-filter-2"
  var filter = defs.append("filter")
    .attr("width", "300%")
    .attr("height", "120%")
    .attr("id", filter_id)
      .append("feOffset")
    .attr("dx", 200)
    .attr("dy", 0)
    .attr("in", "SourceGraphic")

 var circleNavy = svg.append("circle")
   .attr("cx", 100)
   .attr("cy", 100)
```

(continué en la próxima página)

(proviene de la página anterior)

```
   .attr("r", 75)
   .attr("fill", "navy")
   .attr("filter", "url(#" + filter_id + ")");

 var circleTeal = svg.append("circle")
   .attr("cx", 100)
   .attr("cy", 100)
   .attr("r", 75)
   .attr("fill", "teal");
</script>
```

Nota: No tiene mucho sentido usar este filtro en solitario ya que podríamos conseguir el mismo efecto cambiando la posición, más adelante hay otros ejemplos en conjunción con otros filtros.

14.1.3 Filtros de distorsión

feGaussianBlur - Desenfoque gaussiano

Este filtro desenfoca el elemento al que se aplica dependiendo del valor especificado en el atributo `stdDeviation`.

Input

```
<svg width="100%" height="200">
  <defs>
    <filter id="gaussian-blur-filter-1" width="200%" height="120%">
      <feGaussianBlur stdDeviation="5">
    </filter>
  </defs>

  <rect x="200" y="50" width="100" height="100" fill="indianred" filter="url(
↪#gaussian-blur-filter-1)"></rect>
  <rect x="50" y="50" width="100" height="100" fill="indianred"></rect>
</svg>
```

Output

Input - Version D3

```
<div id="container-gaussian-blur-1"></div>
<script>
  var svg = d3.select("#container-gaussian-blur-1")
      .append("svg")
    .attr("width", "100%")
    .attr("height", 200);

  var defs = svg.append("defs");

  var filter_id = "gaussian-blur-filter-1";
  var filter = svg.append("filter")
```

(continué en la próxima página)

(proviene de la página anterior)

```
    .attr("id", filter_id)
    .attr("width", "200%")
    .attr("height", "120%");

  var rectBlurred = svg.append("rect")
    .attr("x", 200)
    .attr("y", 50)
    .attr("width", 100)
    .attr("height", 100)
    .attr("fill", "indianred")
    .attr("filter", "url(#" + filter_id + ")");

  var rectNotBlurred = svg.append("rect")
    .attr("x", 50)
    .attr("y", 50)
    .attr("width", 100)
    .attr("height", 100)
    .attr("fill", "indianred");
</script>
```

14.1.4 Filtros de transformación

Dilatación y adelgazamiento

```
<style>
  #erode-dilate-container-1 p {
    margin: 0;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 3em;
    height: 60px;
  }

  #thin {
    filter: url(#erode);
  }

  #thick {
    filter: url(#dilate);
  }
</style>

<svg xmlns="http://www.w3.org/2000/svg" width="0" height="0">
  <filter id="erode">
    <feMorphology operator="erode" radius="1"/>
  </filter>
  <filter id="dilate">
    <feMorphology operator="dilate" radius="2"/>
  </filter>
</svg>

<div id="erode-dilate-container-1">
  <p>Texto normal</p>
  <p id="thin">Texto adelgazado</p>
  <p id="thick">Texto engordado</p>
</div>
```

14.1.5 Clip paths

«Clip» significa recortar por lo que son los elementos `clipPath` de SVG podemos recortar elementos. La estructura básica para añadir este elemento es la siguiente:

```
<svg>
  <defs>
    <clipPath id="myClippingPath">
      <!-- ... -->
    </clipPath>
  </defs>

 <!-- El elemento al que quieras aplicarlo, puede ser cualquiera -->
  <g id="my-graphic" clip-path="url(#myClippingPath)">
     <!-- ... -->
  </g>
</svg>
```

Ejemplo con D3

Input

```
<div id="container-clippath-1"></div>
<script>

  var width = 680,
      height = 200;

  var svg = d3.select("#container-clippath-1").append("svg")
      .attr("width", width)
      .attr("height", height);

  var circle_clip = svg.append("clipPath")
      .attr("id", "ellipse-clip")
    .append("circle")
      .attr("cx", width/2)
      .attr("cy", height/2)
      .attr("r", 100);

  var rect = svg.append("rect")
      .attr("x", width/3)
      .attr("y", height/2.9)
    .attr("clip-path", "url(#ellipse-clip)")
      .attr("fill","crimson")
      .attr("height",100)
      .attr("width",200)
      .style("stroke", "royalblue")
      .style("stroke-width", "3.5px");
</script>
```

Output

Paso a paso

1. Creamos un path mediante figuras, generadores de paths, rutas escritas manualmente... lo que sea, añadiendo un elemento `clipPath` a un `svg`. Le establecemos un identificador.
2. Añadimos a otro elemento que queremos recortar el atributo `clip-path` apuntando al identificador del `clipPath` (en D3 `.attr("clip-path", "url(#identificador-del-clip)")`).

14.2 Gradientes SVG

14.2.1 Lineal

Para crear un gradiente lineal definimos el filtro `linearGradient`.

Input

```
<svg width="100%" height="400">
  <defs>
    <linearGradient id="grad1" x1="0%" y1="0%" x2="100%" y2="10%">
      <stop offset="10%" style="stop-color:rgb(255,255,0);stop-opacity:1" />
      <stop offset="30%" style="stop-color:rgb(255,0,0);stop-opacity:1" />
      <stop offset="70%" style="stop-color:rgb(0, 50, 255);stop-opacity:1" />
    </linearGradient>
  </defs>

   <rect x="0" y="0" width="300" height="100" fill="url(#grad1)" />
</svg>
```

Output

CAPÍTULO 15

Canvas

15.1 Introducción a `canvas`

15.1.1 Insertando un dibujo `canvas`

```
<canvas id="micanvas" width="200" height="100">
  Este texto se muestra para los navegadores no compatibles con canvas.
  <br>
  Por favor, utiliza Firefox, Chrome, Safari u Opera.
</canvas>
```

Inicialmente el canvas está en blanco y cuando queremos pintar sobre él tenemos que acceder al contexto de renderizado del canvas, sobre el que podremos invocar distintos métodos para acceder a las funciones de dibujo. El proceso simplificado sería el siguiente:

```
var canvas = document.getElementById('micanvas');

// Accedo al contexto de '2d' de este canvas, necesario para dibujar
var contexto = canvas.getContext('2d');

if (contexto) {
    // Dibujo en el contexto del canvas
    contexto.fillRect(50, 0, 10, 150);
} else {
    console.log("Tu navegador no soporta la etiqueta 'canvas'.")
}
```

Ahora sólo falta una última cosa, que es ejecutar estas acciones sólo cuando la página esté cargada por completo y lista para recibirlas. Esto lo conseguimos con la el evento onload del body de la página:

```
<body onload="funcionDeDibujo()">
  ...
</body>
```

15.1.2 Ejemplo básico en canvas

Input

```
<canvas id="lienzo" width="100%" height="200">
  Tu navegador no soporta canvas.
</canvas>

<script>
  // Al cargar el documento
  window.onload = function(){

      // Obtenemos el contexto del canvas
      var elem = document.getElementById("lienzo");
      if (elem && elem.getContext) {
          var context = elem.getContext("2d");
          if (!context) {
              throw new Error("Tu navegador no soporta canvas.");
          } else {

              // Cambiamos el color a azul
              context.fillStyle = "navy";
              // Dibujamos un rectángulo
              context.fillRect(10, 10, 50, 50);

              // Cambiamos el color a amarillo con transparencia
              context.fillStyle = "rgba(255,255,0,0.7)";
              // Dibujamos un rectángulo amarillo semitransparente
              context.fillRect(60, 60, 120, 100);
          }
      }
  };
</script>
```

Output

CAPÍTULO 16

CSS

16.1 Enmascaramiento con CSS

16.1.1 Desvanecimiento de opacidad radial

Se trata de generar una imagen circular con los bordes difuminados a partir de una imagen normal.

Código

```
<style>
  .circle {
    border-radius: 50%;
    display: inline-block;
    position: relative;
  }

  .circle img {
    border-radius: 50%;
    display: block;
    border:1px solid rgba(255,255,255,0);
  }

  .circle:after {
   content: "";
   display: block;
   width: 100%;
   height: 100%;
   background: radial-gradient(ellipse at center, rgba(255,255,255,0) 0%,rgba(255,255,
↪255,1) 69%,rgba(255,255,255,1) 100%);
   border-radius: 50%;
   position: absolute;
   top: 0;
   left: 0;
  }
</style>
<div class="circle">
  <image src="./../../_static/img/camaron.jpg">
</div>
```

Paso a paso

1. Creamos un contenedor `div` para la imagen, lo redondeamos y añadimos la imagen dentro.
2. Redondeamos la imagen y eliminamos la opacidad del borde con `border: 1px solid rgba(255, 255, 255, 0)`.
3. Añadimos un contenido al `div` con una pseudopropiedad `after`. El punto importante aquí es el atributo `background` el cual, establecido a `radial-gradient(ellipse at center, ...` crea una elipse desde el exterior hacia el centro. Para más información sobre esta propiedad consulta la documentación de Mozilla.

16.1.2 Aplicar textura a una imagen

Coniste en superponer dos o más imágenes del mismo tamaño, disminuyendo la opacidad de la imagen de textura.

Mezcla de imagen y textura

Con textura distorsionada

Input

```
<style>
  .image-layers img {
    position: absolute;
  }

  #image-original {
    position: absolute;
    top: 10px;
    width: 335px;
    height: 342px;
  }

  #image-1 img {
    left: 350px;
    top: 10px;
  }

  #image-2 img {
    left: 750px;
    top: 10px;
  }

  #image-2 .image-texture {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    filter: blur(1.5px) brightness(0.88);
  }

  .image-layer {
    border-radius: 2%;
    width: 396px;
    height: 400px;
  }

  .image-texture {
    opacity: 0.22;
    z-index: 2;
  }
</style>

<img id ="image-original" src="../../_static/img/camaron.jpg">

<div class="image-layers" id="image-1">
  <img class="image-layer" src="../../_static/img/camaron.jpg">
  <img class="image-texture image-layer" src="../../_static/img/textures/001.jpg">
</div>

<div class="image-layers" id="image-2">
  <img class="image-layer" src="../../_static/img/camaron.jpg">
  <img class="image-texture image-layer" src="../../_static/img/textures/001.jpg">
</div>
```

Paso a paso

1. Para cada imagen creamos un contenedor `div` con una clase que engloba todas las capas de la imagen (`class="image-layers"`) y establecemos un identificador para poder controlarla individualmente.
2. Creamos una capa para las texturas y otra para la imagen de salida (engloba a todas las capas) mediante las clases `image-layer` e `image-texture`. Añadimos dentro de los contenedores cada imagen con su estilo.
3. Bajamos la opacidad de la capa de textura y establecemos las mismas dimensiones para todas las imágenes con la capa principal `image-layer`.
4. Controlando individualmente las imagenes podemos acceder a los elementos anidados que queremos mediante selección de elementos como `#image-2 .image-texture`. En el ejemplo se ha aplicado un filtro de CSS3.

CAPÍTULO 17

Generador de datos aleatorio

17.1 Faker.js

Esta biblioteca nos permite generar datos aleatorios de diferentes tipos. Puede ser usada mediante su descarga, mediante CDN o a través de un microservicio con llamadas HTTP.

17.1.1 Uso básico

Esta biblioteca tiene muchos tipos de generadores, los cuales podemos consultar en la documentación. Cada tipo de generador (en este ejemplo `name`) sirve para generar un tipo de datos distinto.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/Faker/3.1.0/faker.min.js"></
↪script>

<script>
  window.onload = function(){
      faker.locale = "es";
      var nombreAleatorio = faker.name.findName();
      console.log(nombreAleatorio);
  }
</script>
```

La lista de locales soportadas se encuentra aquí.

CAPÍTULO 18

Indices and tables

- genindex
- modindex
- search