CHAPTER 1

Introduction

Welcome to Constructing Visualizations! This book is for anyone who wants to learn how to create custom interactive data visualizations for the Web using D3 (D3.js). No prior experience with D3 is required, although some familiarity JavaScript and Web Standards including HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and SVG (Scalable Vector Graphics), will be helpful. This book can be used as a textbook for an undergraduate or graduate level data visualization course, or as a reference for professionals.

WHY USE D3?

D3, short for "Data Driven Documents" is a popular JavaScript library used primarily for developing interactive data visualizations. The "Documents" of "Data Driven Documents" refers to the Document Object Model (DOM), which is the tree data structure built into Web browsers that defines the content of Web pages. The core idea behind D3 is to provide the building blocks required to build interactive data visualizations. These building blocks include utilities for loading data, transforming data in various ways, and ultimately rendering data as graphics on the screen.

D3 is designed to be modular, so you can use as much or as little of it as you need. For example, you could use D3 only for its utilities such as scales, but use your favorite framework or graphics library for rendering. Even though "Documents" is in the name, you don't necessarily need to use D3 for any DOM manipulation at all. This modularity makes D3 a powerful and lightweight toolbox for creating data visualizations with no limits as far as how custom and interactive you can make them. D3 is also free and open source, which makes it a great choice for creating custom visualizations that you can share with others, with no need to pay any licensing fees.

Other tools for visualization are also great. For example, BI tools such as Tableau and PowerBI are powerful tools for creating visualizations. Excel and Google Sheets are great for data visualization as well. These tools have user-friendly interfaces that make them easy to use for quickly trying out various ways of visualizing the data. The R and Python ecosystems also have amazing visualization tools, specifically for exploratory data analysis tasks and tasks that require statisti-

cal analysis or machine learning. There are many cases where a tool other than D3 is the best tool for the job. However, if you need something highly custom (bespoke) and interactive that works on the Web, D3 is where it's at.

Most tools other than D3 suffer from what I like to call the "barrier of abstraction". The barrier of abstraction is like a brick wall that abstracts away the details of how the visualization is created. This barrier makes it easy to create visualizations quickly based on limited parameters, but it also makes it difficult to customize the visualization beyond what the tool author imagined people would need. Visualizations built with D3 do not have this barrier of abstraction because the are assemblies of neatly isolated and decoupled building blocks that you can compose and string together in any which way you darn well please. When construcing visualizations with D3, you have full internal control over how the visualization is created, and nothing is hidden behind a barrier of abstraction.

In many workflows, it is useful to use a combination of tools. For example, you might use Tableau, Excel, or Google Sheets to create a quick prototype visualization to explore your data initially, and then use D3 to create a more polished and customized version that you can embed in a Web page or incorporate into a Web-based product. You might use R or Python to perform exploratory data analysis to create a series of initial static visualizations, and then use D3 to create an interactive version of the one the team likes best.

Another advantage of D3, for teaching purposes in particular, is that it exposes each computational step. This is excellent from a pedagogical perspective because it forces us to examine and understand the fundamentals of data visualization algorithms in gruesome detail. There is no black box where you pass an extra configuration parameter and by some magic something new happens. All the logic is bare, exposed, and ready to be modified and extended. This is especially useful when you are learning how to create visualizations, because you can see exactly how each step works and how it impacts the final result.

Regardless of your background, the tools you already know, or your particlar needs for using D3, the goal of this book is to provide you with the knowledge and skills you need to construct compelling and useful interactive data visualizations.

CONSTRAINTS FOSTER CREATIVITY

Constraints foster creativity. When we have constraints, we are forced to think creatively to work within those constraints. This is especially true in the world of coding and data visualization. When we have constraints on the data, the tools, the language or the platform, we are forced to think creatively. I choose to adopt a mindset of creativity that enables me to not only survive but thrive and excel within these constraints. This mindset favors working within constraints over circumventing them. The code content of this book adheres to the following constraints, which are designed to foster creativity:

- **D3 and Web Standards**: We limit ourselves primarily to D3 and Web Standards. This constraint excludes tools other than D3 from our scope of interest. We adopt Web Standards including JavaScript, CSS, and SVG. We

will not be using TypeScript, SASS, or frameworks such as React, Vue, or Svelte. However, we *will* be developing code that can be easily invoked by components within such frameworks. This constraint allows a clearer focus on the essence of data visualization algorithms. It also makes the code more accessible to a wider audience, as Web Standards are widely adopted and understood.

- **Unidirectional Data Flow**: For handling interaction and state management, we use the "unidirectional data flow" pattern. While there are many ways to manage "state" (dynamic configuration that changes over time based on user interactions) in JavaScript applications, the unidirectional data flow pattern centralizes state. This centralization of state requires the definition of a single monolithic function that serves to re-render the entire application every time state changes.

- **Immutable Update Patterns**: We will update state using "immutable update patterns." An immutable update pattern is a way to update state without modifying the original state. For example, instead of mutating a property value on a JavaScript object, we create a brand new object and copy over all the properties, overriding the value for the property we want to update. This constraint allows for easier debugging and testing of code. It also makes it easier to reason about the code, as the state is never mutated. This constraint also supports the unidirectional data flow pattern by ensuring that state changes are always explicit and predictable. Immutable update patterns also work well with performance optimizations based on "memoization", a form of caching that avoids repeating expensive computations unnecessarily.

- **Idempotent Rendering**: We will use only "idempotent rendering". The term "rendering", in our usage of it, refers to the phenomenon of computer code causing changes in the graphics that you see on the computer screen. The so-called "rendering logic" is the code that ultimately manipulates the pixels on the computer screen. The term "idempotent" refers to a kind of function that can be executed multiple times and always result in the same output. Idempotent rendering refers to a style of rendering logic that is designed to be executed multiple times. This idempotent rendering approach allows us to use the unidirectional data flow pattern for interactive graphics. This constraint encourages us to write rendering logic that can execute again and again whenever state changes, which is much more portable and useful than rendering logic that can only execute once.

- **Hot Reloadable Code**: We will produce code that supports "hot reloading". Hot reloading, also called "hot module replacement" (HMR), is when a development environment injects code changes into the running application, retaining state, instead of performing a full refresh and resetting state. This constraint lets us iterate more quickly on visualizations by tweaking various parameters and constants without clobbering the state. Fast iteration is a key

component of the creative process. The faster you can iterate, the more you can do in a certain amount of time. This constraint serves to accelerate our development cadence by saving time and effort. It also makes the development process more joyful.

- **Open Source Reference Implementations**: All code snippets showcased in the book draw from a collection of fully working open source reference implementations. You can easily access, run locally, and modify these implementations to suit your needs. This constraint lets you learn by doing, which is often the best way to learn. Many assignments throughout this book are based on the idea of "Fork and Modify", wherein a reference implementation is provided that students can fork (make a copy of) and modify to complete the assignment. Students can then submit a link to their modified version as the assignment submission. In the professional world, that exact same "Fork and Modify" process can enable rapid prototyping and development of real-world digital products.

These constraints were designed to provide the maximum value to the maximum amount of people. By using only the most widely adopted technologies, the reference implementations will be directly usable to the widest possible audience. Taken together, these constraint support a consistent and coherent mental model for developing interactive graphics, while also supporting maximum interoperability with various frameworks. I hope you find these constraints helpful and that they foster your creativity. These constraints are not meant to be restrictive, but rather to provide a solid foundation on which to build a comprehensive guide to constructing visualizations.

1.1 CIRCLES WITH D3

As a first introduction to rendering graphics with D3 and SVG, let's display some circles of various sizes and colors. This program will serve as a starting point for understanding the fundamental concepts of D3 including selections, data joins, method chaining, DOM manipulation, and idempotent rendering patterns.

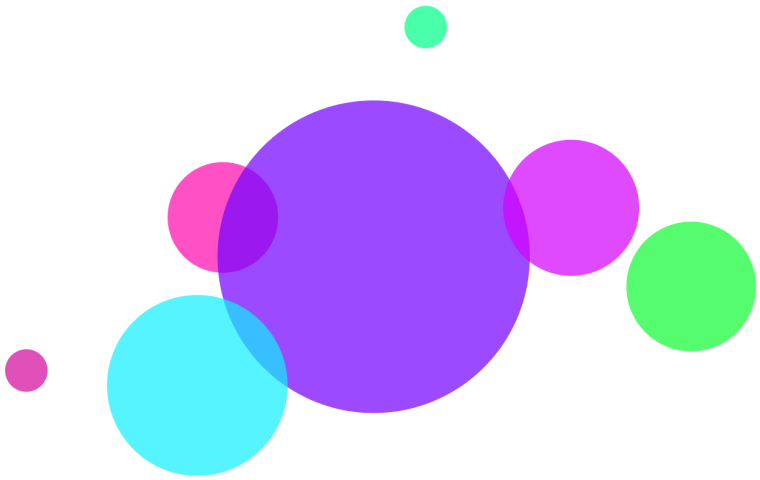

Figure 1.1 Circles with D3: Rendered Graphic

This code is divided into three parts:

- Main Program (`index.js`)
- Defining Data (`data.js`)
- Rendering the SVG Container (`renderSVG.js`)
- Rendering Circles (`renderCircles.js`)

Main Program

The `main` function is the entry point for this program. Rather than put the entire program into a single file, we break it down into smaller parts using JavaScript Modules (also called ECMAScript Modules or ESM). This modular approach allows us to split the code into multiple files and discuss each individually. These modules can be imported into the main program, which defines the flow of the entire program by connecting and orchestrating the imported modules.

circles-with-d3/index.js

```
1 import { data } from './data';
2 import { renderSVG } from './renderSVG';
3 import { renderCircles } from './renderCircles';
4 export const main = (container) => {
5   const svg = renderSVG(container);
6   renderCircles(svg, { data });
7 };
```

Figure 1.2 Circles with D3: Main Program

The first 3 lines of the `index.js` file import the `data`, `renderSVG`, and `renderCircles` functions from modules defined in other files (the `./` prefix indicates a relative path). The body of the `main` function is then defined on line 6. The `main` function takes a single argument, `container`, which is a DOM element that will contain the visualization. The `main` function calls the `renderSVG` function to set up the SVG container, assigning the resulting D3 selection to the variable `svg`. Once the SVG container element is in hand, the next step on line 7 invokes the `renderCircles` function to render the circles, passing in an "options object" containing the imported `data`.

Defining Data

The module `data.js` defines and exports the "data" that will inform the position, size, and color of our SVG circles. This `data` array contains many objects, each of which defines properties of a circle. The `x` and `y` properties represent the center (x,y) coordinates of each circle (in pixel coordinates). The `r` property represents the radius of the circle (in pixels). The `fill` property represents the fill color of the circle using a hexadecimal representation. This pattern of representing data as an array of objects is a common pattern when working with data in JavaScript.

circles-with-d3/data.js

```
1 export const data = [
2   { x: 155, y: 382, r: 20, fill: '#D4089D' },
3   { x: 340, y: 238, r: 52, fill: '#FF0AAE' },
4   { x: 531, y: 59, r: 20, fill: '#00FF88' },
5   { x: 482, y: 275, r: 147, fill: '#7300FF' },
6   { x: 781, y: 303, r: 61, fill: '#0FFB33' },
7   { x: 668, y: 229, r: 64, fill: '#D400FF' },
8   { x: 316, y: 396, r: 85, fill: '#0FF0FF' },
9 ];
```

Figure 1.3 Circles with D3: Data Definition

Rendering the SVG Container

In order to have a place to put our circles, we must first set up an SVG container element. The module `renderSVG.js` file defines a module that exports a function called `renderSVG`. This function expects a single argument, `container`, which is a DOM element that will contain the visualization. This function sets up an SVG element, defines the dimensions, and sets a background color. This function is idempotent, meaning it can be executed multiple times without side effects.

circles-with-d3/renderSVG.js

```
1 import { select } from 'd3';
2 export const renderSVG = (container) =>
3   select(container)
4     .selectAll('svg')
5     .data([null])
6     .join('svg')
7     .attr('width', container.clientWidth)
8     .attr('height', container.clientHeight)
9     .style('background', '#F0FFF4');
```

Figure 1.4 Circles with D3: Rendering the SVG Container

Line 1 imports `select` from the D3 package. This imported `select` function is responsible for creating "D3 Selections", a construct that aids in manipulating the DOM. The expression `select(container)` creates a D3 selection of the container DOM element. Once we have a D3 selection, we can use all sorts of chainable methods on it. The pattern seen here with multiple methods being called in succession on a D3 selection is called "method chaining". This pattern is common in D3 code because it allows you to write concise and expressive code that sets multiple attributes and styles on an element in a single statement.

The `.selectAll('svg')` expression selects all `<svg>` elements that are children of the `container` element (resulting in an empty selection on first invocation). The `.data([null])` expression binds an array containing a single `null` value to the selection. The `.join('svg')` expression has the effect of creating a new `<svg>` element the first time that `main` executes, but for subsequent executions it will re-use the existing `<svg>` element. This pattern of using `[null]` as the data is a common pattern for managing a singular DOM element. Because this rendering logic always results in the same output regardless of how many times it runs, it is considered idempotent.

Typically you'll see `.selectAll`, `.data`, `join` in sequence. This is a special incantation that creates a "data join". A data join in D3 is a powerful pattern that allows you to declaratively specify how data should be bound to DOM elements. This pattern is the foundation of D3's rendering logic and is used to create and update visualizations based on data. The last of these methods, `join`, is the most important because it creates new selection with DOM elements reconciled with data elements that you can then set attributes and styles on. In later chapters we'll dig deeper into the data join and its constituent parts, namely the Enter, Update, and Exit selections.

The `.attr` method is used to set attributes on elements in a selection. Calling this method has a side effect of setting the DOM attributes, but it *does not* create a new selection. Rather, it returns the same selection that it was called on. This is why you can chain multiple `.attr` expressions together in a single statement.

The `.style` method is similar to `.attr` in that it returns the same selection it was called on, but it has a side effect of setting inline styles rather than setting DOM attributes. This is why we can return the returned value from the last chained method as the returned value of the function.

Note that our `main` function expects a `container` div element that has *measurable dimensions* using its `clientWidth` and `clientHeight` properties. This approach allows us to use CSS to style the `container` element, including its dimensions, then use JavaScript to measure those dimensions, as we do here when setting the dimensions of the `<svg>` element. While this implementation is not responsive to changes in the size of the container over time, it does use the measured dimensions on page load and is a good starting point for understanding how to set up an SVG container for a visualization.

The `.attr('width', container.clientWidth)` expression sets the width of the `<svg>` element to the width of the `container` element. The same is done for height. Setting the dimensions of the `<svg>` element in this way makes it fill up the container `div`. If we don't set the width and height attributes, the SVG will get a default size which is not what we want. The `.style('background', '#F0FFF4')` method sets the background color of the `<svg>` element to a light green color. This string of chained methods demonstrates a common pattern in D3 for setting multiple attributes and styles on an element.

The `renderSVG` function omits parentheses and uses the fat arrow syntax. This means that the return value from the last expression (`.style`) is implicitly returned from the function. This is a shorthand syntax for defining functions that return values in JavaScript that is commonly used in modern JavaScript code. It is equivalent to writing `return select(container)` at the beginning of the function and enclosing the function body in parentheses.

Rendering Circles

Now that we have the SVG element set up and the data defined, we can render the circles. The code below shows how to render one circle for each element of the data array using D3. The `renderCircles.js` file defines a module that exports a function called `renderCircles`. This function expects two arguments as follows:

- `svg` - A D3 selection representing a singular SVG element
- `{ data }` - An object that contains the data array

This pattern is one we'll use frequently throughout this book: encapsulating reusable logic by defining a function that takes a D3 selection as a first argument and an options object as a second. This maps roughly onto the mental model for components in React and other frameworks. The function `renderCircles` is essentially a "component" that renders circles in an SVG element.

The syntax used for unpacking the options object here is called "destructuring assignment", a feature of JavaScript that allows you to extract values from objects and arrays and assign them to variables. In this case, the `{ data }` syntax extracts

the `data` property from the object passed as the second argument to the function and makes it available as a variable in the function body.

circles-with-d3/renderCircles.js

```
export const renderCircles = (svg, { data }) =>
  svg
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', (d) => d.x)
    .attr('cy', (d) => d.y)
    .attr('r', (d) => d.r)
    .attr('fill', (d) => d.fill);
```

Figure 1.5 Circles with D3: Rendering SVG Circles

Within the function body itself, we use the D3 selection `svg` to scaffold the DOM structure for the circles. The expression `svg.selectAll('circle').data(data).join('circle')` selects all `<circle>` elements that are children of the `<svg>` element, binds the `data` array to the selection, and creates a new `<circle>` element for each data point in the array. This is the data join pattern in D3, also called the "General Update Pattern". This pattern results in idempotent rendering logic because it can execute multiple times and always result in the same DOM configuration.

The chained calls to `.attr` set the `cx`, `cy`, `r`, and `fill` attributes on each circle. Here, the values are defined as functions rather than constants. When we pass a function as the second argument to `.attr`, that function is invoked multiple times, once for each element in the selection. The function is passed the data bound to the element, traditionally named `d` for datum, and it returns the value that should be set for that attribute. This is a common pattern in D3 for setting attributes based on data, thus creating "Data Driven Documents".

That concludes our first code example! We've set up an SVG element, defined some data, and rendered some circles. This example demonstrates the fundamental concepts of D3, such as D3 Selections, method chaining, data joins, and idempotent rendering patterns. These concepts are the foundational building blocks for constructing custom interactive data visualizations with D3. In the next chapters, we'll dive deeper into these concepts and explore how they relate to interactions and changes in state. We'll also learn how to load and parse data from external sources, such as CSV files, to construct more complex and meaningful interactive data visualizations.

ASSIGNMENT: CREATIVE CODING WITH CIRCLES

Instructions

1. Fork this example: `https://vizhub.com/curran/circles-with-d3`

2. Modify it:
    - Maybe make it Halloween or Christmas themed
    - Change the colors to a color scheme you like
    - Be creative
    - Maybe try different shapes like `<rect>` or `<line>` elements
3. Submit the link to your forked viz

Grading Criteria

- If you make any changes at all and it runs, you'll get full marks
- This is just a warm up and a way to get everyone started coding fast
- You can either use VizHub to start coding without any setup, or use your own local development environment
- Don't stress about it - relax and experience how fun coding can be!