1 Einleitung

D3.js (oder kurz D3 für *Data-Driven Documents*, also »datengestützte Dokumente«) ist eine JavaScript-Bibliothek, die dazu dient, den DOM-Baum (Document Object Model) zu bearbeiten, um Informationen grafisch darzustellen. Sie ist zu einem De-facto-Standard für Infografiken im Web geworden.

Trotz ihrer Beliebtheit wird D3 eine steile Lernkurve nachgesagt. Meiner Meinung nach liegt das nicht daran, dass D3 kompliziert wäre (was sie nicht ist), und auch nicht an ihrer umfangreichen API (die zwar groß, aber gut strukturiert und sehr gut gestaltet ist). Viele der Schwierigkeiten, mit denen neue Benutzer zu kämpfen haben, sind, so glaube ich, auf *falsche Vorstellungen* zurückzuführen. Da D3 verwendet wird, um eindrucksvolle Grafiken zu erstellen, liegt es nahe, sie als »Grafikbibliothek« anzusehen, die den Umgang mit grafischen Grundelementen erleichtert und es ermöglicht, gängige Arten von Plots zu erstellen, ohne sich um die Einzelheiten kümmern zu müssen. Neulinge, die sich D3 mit dieser Erwartung nähern, sind unangenehm überrascht von der Ausführlichkeit, mit der so grundlegende Dinge wie die die Farbe eines Elements festzulegen sind. Und was hat es

mit diesen ganzen »Selections« auf sich? Warum kann man nicht einfach ein leinwandartiges Element verwenden?

Das Missverständnis beruht darauf, dass D3 eben *keine* Grafikbibliothek ist, sondern eine JavaScript-Bibliothek zur Bearbeitung des DOM-Baums. Ihre Grundbausteine sind keine Kreise und Rechtecke, sondern Knoten und DOM-Elemente. Die typischen Vorgehensweisen bestehen nicht darin, grafische Formen auf eine »Leinwand« (Canvas) zu zeichnen, sondern Elemente durch Attribute zu formatieren. Die »aktuelle Position« wird nicht durch *x*/*y*-Koordinaten auf einer Leinwand angegeben, sondern durch die Auswahl von Knoten im DOM-Baum.

Das führt – aus meiner Sicht – zu dem zweiten Hauptproblem für viele neue Benutzer: Bei D3 handelt es sich um eine Webtechnologie, die sich auf andere Webtechnologien stützt, auf die DOM-API und das Ereignismodell, auf CSS-Selektoren und -Eigenschaften (Cascading Style Sheets), auf das JavaScript-Objektmodell und natürlich auf das SVG-Format (Scalable Vector Graphics). In vielen Aspekten stellt D3 nur eine relativ dünne Schicht über diesen Webtechnologien dar und ihr eigenes Design spiegelt oft die zugrunde liegenden APIs wider. Das führt zu einer sehr umfangreichen und uneinheitlichen Umgebung. Wenn Sie bereits mit dem kompletten Satz moderner Webtechnologien vertraut sind, der als HTML5 bekannt ist, werden Sie sich darin zu Hause fühlen, doch wenn *nicht*, dann kann das Fehlen einer ausgeprägten, einheitlichen Abstraktionsschicht ziemlich verwirrend sein.

Glücklicherweise müssen Sie nicht sämtliche dieser zugrunde liegenden Technologien ausgiebig studieren. D3 erleichtert ihre Nutzung und bietet eine erhebliche Vereinheitlichung und Abstraktion. Der einzige Bereich, in dem es definitiv nicht möglich ist, sich einfach durchzumogeln, ist SVG. Sie müssen auf jeden Fall ein ausreichendes Verständnis von SVG mitbringen, und zwar nicht nur der darstellenden Elemente, sondern auch der Strukturelemente, die steuern, wie die Informationen in einem Graphen gegliedert sind. Alles, was Sie wissen müssen, habe ich in Anhang B zusammengestellt. Wenn Sie mit SVG nicht vertraut sind, sollten Sie diesen Anhang durcharbeiten, bevor Sie sich an den Rest des Buches machen. Sie werden später dafür dankbar sein.

Zielgruppe

Dieses Buch ist für *Programmierer* und *Wissenschaftler* gedacht, die D3 zu ihrem Werkzeugkasten hinzufügen möchten. Ich gehe davon aus, dass Sie ausreichende Erfahrungen als Programmierer aufweisen und ohne Schwierigkeiten mit Daten und Grafiken arbeiten können. Allerdings erwarte ich nicht, dass Sie mehr als oberflächliche Kenntnisse in professioneller Webentwicklung haben.

Folgende Voraussetzungen sollten Sie mitbringen:

- Kenntnisse in einer oder zwei Programmiersprachen (nicht unbedingt JavaScript) und ausreichend Selbstvertrauen, um sich die Syntax einer neuen Sprache aus ihrer Referenz anzueignen.
- Vertrautheit mit modernen Programmierkonzepten, also nicht nur mit Schleifen, Bedingungen und gewöhnlichen Datenstrukturen, sondern auch mit Closures und Funktionen höherer Ordnung.
- Grundkenntnisse in XML und der hierarchischen Struktur von XML-Dokumenten. Ich erwarte, dass Sie das DOM kennen und wissen, dass es die Elemente einer Webseite als Knoten eines Baums behandelt. Allerdings setze ich nicht voraus, dass Sie mit der ursprünglichen DOM-API oder einem ihrer modernen Nachfolger (wie jQuery) vertraut sind.
- Einfache Kenntnisse in HTML und CSS (Sie sollten in der Lage sein, `<body>`- und `<p>`-Tags usw. zu erkennen und zu verwenden) sowie eine gewisse Vertrautheit mit der Syntax und den Mechanismen von CSS.

Insbesondere aber gehe ich davon aus, dass meine Leser *ungeduldig* sind: erfahren und fähig, aber frustriert von früheren Versuchen, mit D3 zurechtzukommen. Wenn Sie sich darin wiedererkennen, dann ist dieses Buch genau das richtige für Sie!

Warum D3?

Warum sollten sich Programmierer und Wissenschaftler – oder überhaupt irgendwelche Personen, die nicht vorrangig Webentwickler sind – mit D3 beschäftigen? Dafür gibt es vor allem die folgenden Gründe:

- D3 bietet eine bequeme Möglichkeit, um Grafiken im Web zu verbreiten. Wenn Sie mit Daten und Visualisierungen arbeiten, kennen Sie ja den Vorgang: Sie erstellen Diagramme in ihrem bevorzugten Plotprogramm, speichern die Ergebnisse als PNG oder PDF und erstellen dann eine Webseite mit `<img>`-Tags, sodass andere Ihre Arbeit einsehen können. Wäre es nicht schöner, wenn Sie Ihre Diagramme in einem Schritt erstellen und veröffentlichen könnten?
- Noch wichtiger ist jedoch der Umstand, dass es mit D3 auf einfache und komfortable Weise möglich ist, *animierte* und *interaktive* Grafiken zu erstellen. Dieser Punkt kann gar nicht deutlich genug herausgestellt werden: Die Visualisierung wissenschaftlicher Daten profitiert genauso von Animation und Interaktivität wie jeder Bereich, allerdings ließ sich dies früher nur schwer erreichen, es erforderte meistens komplizierte und unpassende Technologien (haben Sie sich jemals an Xlib-Programmierung versucht?) oder spezialisierte

und oftmals teure kommerzielle Lösungen. Mit D3 können Sie diese Hürden überwinden und Ihre zeitgemäßen Visualisierungsbedürfnisse erfüllen.

- Ganz abgesehen von Grafiken ist D3 ein gut zugängliches, leicht zu lernenden und leicht zu verwendendes Framework zur DOM-Bearbeitung für alle möglichen Zwecke. Wenn Sie hin und wieder mit dem DOM arbeiten müssen, kann D3 dazu völlig ausreichen, sodass Sie nicht auch noch das Beherrschen aller anderen Frameworks und APIs für die Webprogrammierung erlernen müssen. Der Aufbau der Bibliothek selbst ist außerdem ein bemerkenswertes Modell der Funktionalitäten für gängige Datenbearbeitungs- und Visualisierungsaufgaben, die sie im Lieferzustand mitbringt.

Vor allem aber bin ich der Meinung, dass D3 eine *emanzipierende* Technologie ist, die es ihren Benutzern erlaubt, ihren Bestand an verfügbaren Lösungsmöglichkeiten ganz allgemein zu erweitern. Die bemerkenswertesten Nutzanwendungen von D3 sind wahrscheinlich diejenigen, die noch nicht entdeckt wurden.

Was Sie in diesem Buch finden werden ...

Dieses Buch soll eine möglichst *umfassende* und gleichzeitig *knappe* Einführung in D3 sein, die die wichtigsten Aspekte in ausreichender Tiefe darstellt.

- Es soll als komfortables, *zentrales* Nachschlagewerk dienen, da es sowohl die API-Referenzdokumentation als auch Hintergrundinformationen zu verwandten Themen bietet, mit denen Sie nicht unbedingt vertraut sein müssen (wie SVG, JavaScript und das DOM, aber auch Farbräume und das `canvas`-Element von HTML).
- Der Schwerpunkt liegt auf *Mechanismen* und *Gestaltungsprinzipien* und nicht auf vorgefertigten »Kochrezepten«. Ich gehe davon aus, dass Sie D3 gründlich genug lernen wollen, um es für Ihre eigenen und möglicherweise neuartigen und unvorhergesehenen Zwecke einzusetzen.

Im Grunde genommen wünsche ich mir, dass dieses Buch Sie auf die Dinge vorbereitet, die Sie mit D3 tun können, an die ich aber selbst niemals gedacht hätte.

... und was nicht!

Dieses Buch ist bewusst auf D3 und die Möglichkeiten und Mechanismen dieser Bibliothek beschränkt. Daher fehlt eine ganze Reihe von Dingen:

- Ausführliche Fallstudien und Kochrezepte
- Einführungen in Datenanalyse, Statistik und Grafikdesign

- Andere JavaScript-Frameworks als D3
- Allgemeine Erörterung der modernen Webentwicklung

Ich möchte insbesondere die beiden letzten Punkte betonen. In diesem Buch geht es *ausschließlich* um D3, ohne Nutzung oder Abhängigkeiten von anderen JavaScript-Frameworks und Bibliotheken. Das ist volle Absicht: Ich möchte D3 auch solchen Lesern zugänglich machen, die mit dem reichhaltigen, aber uneinheitlichen Umfeld von JavaScript nicht vertraut sind oder sogar auf Kriegsfuß stehen. Aus demselben Grund werden in diesem Buch auch keine anderen Themen der modernen Webentwicklung besprochen. Insbesondere finden Sie hier keinerlei Diskussionen zur *Browserkompatibilität* und ähnlichen Themen. Ich setze voraus, dass Sie einen modernen, aktuellen JavaScript-fähigen Browser verwenden, der in der Lage ist, SVG darzustellen.[1]

Ein weiterer Aspekt, der nicht behandelt wird, betrifft die Unterstützung von D3 für geografische und raumbezogene Informationen. Diese Themen sind zwar wichtig, aber gut überschaubar, sodass es nicht allzu schwierig sein sollte, sie durch die Lektüre der D3-Referenzdokumentation (*https://github.com/d3/d3/blob/master/API.md*) zu erlernen, wenn Sie mit den Grundlagen von D3 vertraut sind.

Ein Leitfaden durch dieses Buch

Dieses Buch ist kontinuierlich aufgebaut. Von Kapitel zu Kapitel wird neuer Stoff eingeführt. Allerdings können Sie insbesondere die hinteren Kapitel in beliebiger Reihenfolge lesen, nachdem Sie die Grundlagen in der ersten Hälfte des Buches erlernt haben. Ich schlage die folgende Vorgehensweise vor:

1. Sofern Sie nicht bereits solide Kenntnisse in SVG haben, empfehle ich Ihnen dringend, mit Anhang B anzufangen. Ohne diese Kenntnisse ergibt alles andere nicht viel Sinn.
2. Lesen Sie auf jeden Fall Kapitel 2 zur Einführung und um Ihre Erwartungen für die kommenden Themen richtig einzuordnen.
3. Kapitel 3 ist Pflichtlektüre. Selections bilden *das* grundlegende Ordnungsprinzip von D3. Sie bieten nicht nur Zugriff auf den DOM-Baum, sondern kümmern sich auch um die Verknüpfung zwischen den DOM-Elementen und den Datenmengen. Praktisch jedes D3-Programm beginnt mit einer Selection, und das Verständnis, was eine Selection ist und was sie kann, ist für die Arbeit mit D3 unbedingt erforderlich.

1 Das ist auch im Einklang mit dem Geist von D3. So heißt es auf der D3-Website: »D3 ist keine Kompatibilitätsschicht. Wenn Ihr Browser die Standards nicht unterstützt, haben Sie Pech gehabt« (*https://github.com/d3/d3/wiki*).

4. Streng genommen ist Kapitel 4 zum Thema Ereignisbehandlung, Interaktivität und Animation optional. Da diese Dinge jedoch zu den faszinierenden Möglichkeiten gehören, die D3 bietet, wäre es schade, dieses Kapitel zu überspringen.
5. Kapitel 5 ist wichtig, da es die Grundprinzipien des Designs von D3 beschreibt (wie Komponenten und Layouts) und eine Einführung in allgemein nützliche Techniken gibt (wie SVG-Transformationen und benutzerdefinierte Komponenten).
6. Die restlichen Kapitel können Sie im Großen und Ganzen in beliebiger Reihenfolge lesen, wann immer Sie etwas über die jeweiligen Themen wissen müssen. Insbesondere möchte ich jedoch Ihre Aufmerksamkeit auf Kapitel 7 mit seiner ausführlichen Beschreibung der unscheinbaren, aber äußerst vielseitigen Skalierungsobjekte sowie auf die Vielzahl der Funktionen zur Handhabung von Arrays in Kapitel 9 lenken.

Konventionen

In diesem Abschnitt werden einige Vereinbarungen vorgestellt, die in diesem Buch gelten.

Konventionen in der D3-API

In der D3-API gelten einige Konventionen, die die Nützlichkeit der Bibliothek erhöhen. Bei einigen davon handelt es sich nicht um D3-spezifische Regeln, sondern um gängige JavaScript-Idiome, mit denen Sie aber möglicherweise nicht vertraut sind, wenn Sie nicht selbst in JavaScript programmieren. Ich gebe Sie hier einmal gesammelt an, um die nachfolgenden Erörterungen von überflüssigen Wiederholungen frei zu halten.

- D3 ist vor allem eine Schicht für den Zugriff auf den DOM-Baum. In der Regel versucht D3 nicht, die zugrunde liegenden Technologien zu kapseln, sondern bietet stattdessen komfortable, aber allgemein gehaltene Handles dafür. Beispielsweise stellt D3 keine eigenen Abstraktionen für Kreise und Rechtecke bereit, sondern gibt Programmierern unmittelbaren Zugriff auf die Elemente von SVG zum Erstellen von grafischen Formen. Diese Vorgehensweise bietet den Vorteil einer enormen Anpassungsfähigkeit, denn dadurch ist D3 nicht an eine bestimmte Technologie oder Version gebunden. Der Nachteil besteht darin, dass Programmierer neben D3 auch die zugrunde liegenden Technologien kennen müssen, da D3 selbst keine vollständige Abstraktionsschicht bildet.

- Da JavaScript keine formalen Funktionssignaturen erzwingt, sind technisch gesehen alle Funktionsargumente optional. Viele D3-Funktionen nutzen daher das folgende Idiom: Beim Aufruf *mit* geeigneten Argumenten dienen sie als *Set-Methoden* (sie setzen die entsprechende Eigenschaft auf den übergebenen Wert), beim Aufruf *ohne* Argumente dagegen als *Get-Methoden* (sie geben den aktuellen Wert der Eigenschaft zurück). Um eine Eigenschaft komplett zu *entfernen*, rufen Sie die entsprechende Set-Methode mit dem Argument `null` auf.
- Beim Aufruf als Set-Methoden geben Funktionen gewöhnlich einen Verweis auf das aktuelle Objekt zurück und ermöglichen damit eine Methodenverkettung. (Dieses Idiom wird so konsequent verwendet und lässt sich intuitiv nutzen, sodass ich nur noch selten ausdrücklich darauf hinweisen werde.)
- Anstelle eines Wertes können viele Set-Funktionen von D3 auch eine *Zugriffsfunktion* als Argument entgegennehmen, wobei der von ihr zurückgegebene Wert dazu verwendet wird, die betreffende Eigenschaft festzulegen. Nicht alle Zugriffsfunktionen in D3 erwarten die gleichen Parameter, aber miteinander verwandte D3-Funktionen rufen ihre Zugriffsfunktionen immer auf einheitliche Weise auf. Die Einzelheiten zu den Zugriffsfunktionen werden jeweils bei den entsprechenden D3-Funktionen angegeben.
- Einige wichtige Funktionalitäten von D3 sind als *Funktionsobjekte* implementiert. Sie führen ihre Hauptaufgabe aus, wenn sie als Funktionen aufgerufen werden. Gleichzeitig aber sind sie auch Objekte mit Memberfunktionen und einem internen Status. (Beispiele dafür sind die *Skalierungsobjekte* aus Kapitel 7 sowie die *Generatoren* und *Komponenten* aus Kapitel 5.) Ein häufig genutztes Muster besteht darin, ein solches Objekt zu instanziieren, es mithilfe seiner Memberfunktionen zu konfigurieren und es schließlich aufzurufen, damit es seinen Zweck erfüllen kann. Häufig wird für den endgültigen Aufruf nicht die explizite Funktionsaufrufsyntax verwendet, sondern eine der JavaScript-Vorgehensweisen für »synthetische« Funktionsaufrufe: Das Funktionsobjekt wird an eine andere Funktion übergeben (etwa `call()`), die die erforderlichen Argumente bereitstellt und das Funktionsobjekt auswertet.

Konventionen für die API-Referenztabellen

In diesem Buch finden Sie Tabellen mit Erklärungen zu einzelnen Teilen der D3-API. Die Einträge in diesen Tabellen sind nach Relevanz geordnet, wobei zusammengehörige Funktionen auch zusammen aufgeführt werden.

- D3-Funktionen werden entweder für das globale Objekt `d3` oder als Memberfunktionen von D3-Objekten aufgerufen. Bei einigen Funktionen ist beides

möglich. Wird eine Funktion durch ein Objekt aufgerufen, so wird dieses Objekt als *Empfänger* des Methodenaufrufs bezeichnet. Innerhalb der Memberfunktion zeigt die Variable `this` auf dieses Objekt.

- In allen API-Referenztabellen ist jeweils der Typ des Empfängers in der Tabellenunterschrift angegeben. Die Tabellen verweisen nicht ausdrücklich auf den Prototyp des Objekts.
- Bei den Funktionssignaturen wird *versucht*, die Typen der einzelnen Argumente anzugeben, aber da viele Funktionen eine breite Palette an Argumenttypen akzeptieren, lässt sich eine eindeutige Schreibweise nicht umsetzen. Um alle Einzelheiten zu erfassen, müssen Sie die Beschreibung lesen. *Eckige Klammern* stehen für Arrays. Optionale Funktionsargumente werden nicht ausdrücklich angegeben.

Konventionen für die Codebeispiele

Die Codebeispiele sollen die Merkmale und Mechanismen von D3 veranschaulichen. Um den Kernpunkt jeweils möglichst deutlich zu zeigen, wurden die Beispiele auf das Notwendige reduziert. Ich habe auf die meisten Feinheiten wie ansprechende Farben und interessante Datenmengen verzichtet. Meistens verwende ich Primärfarben und kleine und einfache Datenmengen.

Dafür ist jedes Beispiel in sich abgeschlossen, kann wie gezeigt ausgeführt werden und erstellt den zugehörigen Graphen. Bis auf wenige Ausnahmen gebe ich keine Codefragmente an. Meiner Erfahrung nach ist es besser, einfache Beispiele komplett darzustellen, anstatt nur die »interessanten Teile« längerer Beispiele zu zeigen. Auf diese Weise besteht keine Gefahr, dass der Gesamtzusammenhang verloren geht. Alle Beispiele sind ausführbar und können nach Belieben erweitert und ausgeschmückt werden.

Namenskonventionen

In den Beispielen gilt die folgende Namenskonvention für Variablen:

- Für einzelne Objekte wird der erste Buchstabe der englischen Bezeichnung verwendet: `c` für »circle« (Kreis), `p` für »point« (Punkt) usw. Bei Sammlungen wird ein Plural-s angehängt. So ist etwa `cs` ein Array aus Kreisen und `ps` ein Array aus Punkten.
- Häufig verwendete Größen haben davon abweichende Bezeichnungen. Pixel heißen `px` und Skalierungsobjekte `sc` (»scale«). Generationen und Komponenten sind Funktionsobjekte, die etwas erzeugen, und werden daher `mkr` (»maker«) genannt.

- Der Buchstabe `d` wird allgemein verwendet, um in anonymen Funktionen »das aktuelle Etwas« zu bezeichnen. Bei der Arbeit mit D3-Selections ist `d` gewöhnlich ein einzelner Datenpunkt, der an ein DOM-Element gebunden ist, bei der Arbeit mit Arrays dagegen ein Arrayelement (z. B. in `ds.map( d => +d )`).
- Datenmengen werden als `data` oder `ds` bezeichnet.
- Selections, die für ein `<svg>`- oder `<g>`-Element stehen, kommen häufig vor. Werden sie einer Variablen zugewiesen, so werden sie als `svg` bzw. `g` bezeichnet.

Aufbau der Quelldateien

Ab Kapitel 3 setze ich bei jedem Codelisting voraus, dass die Seite bereits ein `<svg>`-Element mit einem eindeutigen `id`-Attribut und ordnungsgemäß festgelegten `width`- und `height`-Attributen enthält. Der Beispielcode wählt dann dieses SVG-Element anhand seines `id`-Attributs aus und weist diese Selection häufig einer Variablen zu, um später darauf verweisen zu können:

```
var svg = d3.select( "#fig" );
```

Dadurch wird die Mehrdeutigkeit vermieden, die sich bei der Verwendung eines allgemeineren Selektors (wie `d3.select( "svg" )`) einstellen würde, und es erleichtert, mehrere Beispiele in eine einzige HTML-Seite aufzunehmen.

Zu jedem Graphen gibt es eine JavaScript-Funktion, die die SVG-Elemente des Diagramms dynamisch erstellt. Vereinbarungsgemäß beginnen die Funktionsnamen mit `make`, worauf der Wert des `id`-Attributs für das SVG-Zielelement folgt.

Bis auf Kapitel 2 gibt es in jedem Kapitel nur eine HTML-Seite und eine JavaScript-Datei. (Von wenigen Ausnahmen abgesehen nehme ich den JavaScript-Code nicht unmittelbar in die HTML-Seite auf.)

Plattform, JavaScript-Version und Browser

Um die Beispiele auszuführen, brauchen Sie einen lokalen oder gehosteten Webserver (siehe Anhang A). Die Beispiele sollten in jedem modernen Browser mit aktiviertem JavaScript laufen. Zurzeit sind verschiedene Versionen von JavaScript im Umlauf.[2] In den Codebeispielen wird fast ausschließlich »klassisches« JavaScript (ES5, freigegeben 2009/2011) ohne weitere Frameworks und Bibliotheken verwendet. Für die folgenden drei Merkmale ist jedoch eine jüngere Version von JavaScript (ES6, veröffentlicht 2015) erforderlich:

- Die knappe *Pfeilschreibweise* für anonyme Funktionen (siehe Anhang C), die in den Beispielen durchgehend verwendet wird.

2 Siehe *https://en.wikipedia.org/wiki/ECMAScript*.

- Destrukturierende Zuweisungen (`[a, b] = [b, a]`), die an einigen Stellen verwendet werden.
- In mehreren Beispielen wird mithilfe von D3-Wrappern für die Fetch-API (siehe Kapitel 6) auf Remoteressourcen zugegriffen. Dafür ist das JavaScript-Objekt `Promise` erforderlich.

2

Los geht's: erste Graphen mit D3

Zu Anfang wollen wir einige Beispiele durcharbeiten, die die Möglichkeiten von D3 demonstrieren und Sie in die Lage versetzen, selbst Aufgaben aus der Praxis zu lösen – und sei es auch nur, indem Sie diese Beispiele entsprechend anpassen. Die ersten beiden Beispiele in diesem Kapitel zeigen, wie Sie aus Datendateien die gebräuchlichen *Streu-* und *XY-Diagramme* erstellen, und zwar komplett mit Achsen. Die Diagramme sehen nicht sehr hübsch aus, erfüllen aber ihren Zweck, und es ist auf einfache Weise möglich, sie zu verschönern und die Prinzipien auf andere Datenmengen zu übertragen. Das dritte Beispiel ist nicht vollständig, sondern dient mehr der Veranschaulichung, um Ihnen zu zeigen, wie einfach es ist, Ereignisbehandlung und Animationen in Ihre Dokumente aufzunehmen.

Erstes Beispiel: eine einzige Datenmenge

Um D3 kennenzulernen, betrachten wir die kleine, einfache Datenmenge aus Beispiel 2–1. Bei der grafischen Ausgabe dieser Daten mithilfe von D3 kommen wir schon mit vielen der Grundprinzipien in Berührung.

Beispiel 2–1: Eine einfache Datenmenge (examples-simple.tsv)

```
x       y1
100     50
200     100
300     150
400     200
500     250
```

Wie bereits in Kapitel 1 erwähnt, ist D3 eine JavaScript-Bibliothek zur Bearbeitung des DOM-Baums, um Informationen grafisch darzustellen. Das deutete schon darauf hin, dass jeder D3-Graph über mindestens *zwei* oder *drei* »bewegliche Teile« verfügt:

- Eine HTML-Datei mit dem DOM-Baum
- Eine JavaScript-Datei oder ein Skript mit den Befehlen zur Bearbeitung des DOM-Baums
- Häufig eine Datei oder eine andere Ressource mit der Datenmenge

Beispiel 2–2 zeigt die gesamte HTML-Datei.

Beispiel 2–2: Eine HTML-Datei, die ein SVG-Element definiert

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>    ❶
  <script src="examples-demo1.js"></script>    ❷
</head>

<body onload="makeDemo1()">    ❸
  <svg id="demo1" width="600" height="300"
       style="background: lightgrey" />    ❹
</body>
</html>
```

Ja, Sie sehen richtig – die HMTL-Datei ist im Grunde genommen leer! Alle Aktionen finden im JavaScript-Code statt. Sehen wir uns kurz die wenigen Dinge an, die in dem HTML-Dokument geschehen:

❶ Als Erstes lädt das Dokument *d3.js*, also die Bibliothek D3.

❷ Danach lädt das Dokument unsere JavaScript-Datei. Sie enthält alle Befehle zur Definition des Graphen, den wir anzeigen wollen.

❸ Das `<body>`-Tag definiert einen `onload`-Ereignishandler, der ausgelöst wird, wenn der Browser das `<body>`-Element komplett geladen hat. Die Ereignishand-

lerfunktion `makeDemo1()` ist in unserer JavaScript-Datei *examples-demo1.js* definiert.[1]

❹ Am Ende enthält das Dokument ein SVG-Element mit einer Größe von 600 × 300 Pixeln. Es hat einen hellgrauen Hintergrund, damit Sie es erkennen können, ist sonst aber leer.

Der dritte und letzte Bestandteil ist die JavaScript-Datei, die Sie in Beispiel 2–3 sehen.

Beispiel 2–3: Befehle für Abbildung 2–1

```
function makeDemo1() {    ❶
    d3.tsv( "examples-simple.tsv" )    ❷
        .then( function( data ) {    ❸ ❹
            d3.select( "svg" )    ❺
                .selectAll( "circle" )    ❻
                .data( data )    ❼
                .enter()    ❽
                .append( "circle" )    ❾
                .attr( "r", 5 ).attr( "fill", "red" )    ❿
                .attr( "cx", function(d) { return d["x"] } )    ⓫
                .attr( "cy", function(d) { return d["y"] } );
        } );
}
```

Wenn Sie diese drei Dateien (die Datendatei, die HTML-Seite und die JavaScript-Datei) zusammen mit der Bibliotheksdatei *d3.js* in ein gemeinsames Verzeichnis stellen und die Seite im Browser laden, sollte der Browser den Graphen aus Abbildung 2–1[2] anzeigen.

Sehen wir uns nun nacheinander die einzelnen JavaScript-Befehle an:

❶ Das Skript definiert nur eine einzige Funktion, nämlich den Callback `makeDemo1()`, der aufgerufen wird, wenn die HTML-Seite vollständig geladen ist.

❷ Die Funktion lädt die Datendatei mithilfe der `Fetch`-Funktion `tsv()` (sie »ruft sie ab«, daher der Bezug zu »to fetch«). In D3 sind mehrere Funktionen definiert, um Dateiformate mit Trennzeichen zu lesen. `tsv()` ist für tabulatorgetrennte Dateien bestimmt.

---

1 Die Definition eines Ereignishandlers über das `onload`-Tag wird oft missbilligt, weil dadurch JavaScript-Code in HTML eingebettet wird. Moderne Alternativen finden Sie in Anhang A und C.

2 Sie sollten in der Lage sein, die Seite und die zugehörige JavaScript-Datei zu laden, indem Sie dem Browser das lokale Verzeichnis als Zieladresse angeben. Allerdings kann es sein, dass der Browser sich weigert, die Datendatei auf diese Weise zu laden. Daher ist es gewöhnlich notwendig, bei der Arbeit mit D3 einen Webserver auszuführen. Mehr darüber erfahren Sie in Anhang A.

❸ Wie alle Funktionen aus der JavaScript-API `Fetch` gibt auch `tsv()` ein `Promise`-Objekt zurück. Ein solches Objekt enthält ein Resultset und einen Callback und ruft Letzteren auf, wenn das Resultset vollständig und zur Verarbeitung bereit ist. Anschließend stellt das `Promise`-Objekt die Funktion `then()` bereit, um den gewünschten Callback zu registrieren. (Mehr über JavaScript-Promises erfahren Sie im Abschnitt »*Promises in JavaScript*« auf *S. 120*.)

❹ Der Callback, der nach dem Laden der Datei aufgerufen werden soll, wird als anonyme Funktion definiert, die den Inhalt der Datendatei als Argument erhält. Die Funktion `tsv()` gibt den Inhalt der tabulatorgetrennten Datei als Array von JavaScript-Objekten zurück. Jede Zeile in der Datei ergibt ein Objekt, wobei die Eigenschaftennamen in der Kopfzeile der Eingabedatei definiert sind.

❺ Wir wählen das `<svg>`-Element als die Position im DOM-Baum, die den Graphen enthalten soll. Die Funktionen `select()` und `selectAll()` nehmen einen CSS-Selektorstring entgegen (siehe »*CSS-Selektoren*« auf *S. 38* in Kapitel 3) und geben die übereinstimmenden Knoten zurück – `select()` nur die erste Übereinstimmung, `selectAll()` eine Sammlung aller Übereinstimmungen.

❻ Als Nächstes wählen wir *alle* `<circle>`-Elemente innerhalb des `<svg>`-Knotens aus. Das mag absurd erscheinen, denn schließlich gibt es darin gar keine `<circle>`-Elemente! Dieser merkwürdige Aufruf stellt jedoch kein Problem dar, da `selectAll("circle")` lediglich eine *leere* Sammlung (von `<circle>`-Elementen) zurückgibt, sondern erfüllt sogar eine wichtige Funktion, da er mit dieser leeren Sammlung einen *Platzhalter* erstellt, den wir anschließend füllen werden. Das ist ein gängiges D3-Idiom, um Graphen mit neuen Elementen zu versehen.

❼ Als Nächstes verknüpfen wir die Sammlung der `<circle>`-Elemente mit der Datenmenge. Dazu verwenden wir den Aufruf `data(data)`. Es ist wichtig, zu erkennen, dass die beiden Sammlungen (DOM-Elemente auf der einen Seite und Datenpunkte auf der anderen) nicht im Ganzen miteinander verbunden werden. Stattdessen versucht D3 eine 1:1-Beziehung zwischen den DOM-Elementen und den Datenpunkten herzustellen: *Jeder Datenpunkt wird durch ein eigenes DOM-Element* dargestellt, das wiederum seine Eigenschaften (wie Position, Farbe und Erscheinungsbild) aus den Informationen über den Datenpunkt bezieht. Der Aufbau und die Pflege dieser 1:1-Beziehungen zwischen einzelnen Datenpunkten und den zugehörigen DOM-Elementen ist ein grundlegendes Merkmal von D3. (In Kapitel 3 werden wir uns diesen Vorgang noch ausführlicher ansehen.)

❽ Die Funktion `data()` gibt die Sammlung der Elemente zurück, die mit den einzelnen Datenpunkten verknüpft wurden. Zurzeit kann D3 die einzelnen Datenpunkte nicht mit `<circle>`-Elementen verknüpfen, da es (noch) keine

gibt. Die zurückgegebene Sammlung ist daher leer. Allerdings bietet D3 mithilfe der (merkwürdig benannten) Funktion `enter()` auch Zugriff auf alle restlichen Datenpunkte, für die keine übereinstimmenden DOM-Elemente gefunden werden konnten. Die übrigen Befehle werden für die einzelnen Elemente in dieser »Restsammlung« aufgerufen.

❾ Als Erstes wird der Sammlung der `<circle>`-Elemente innerhalb des in Zeile 6 ausgewählten SVG-Elements ein `<circle>`-Element angehängt.

❿ Anschließend werden einige fixe (also nicht datenabhängige) Attribute und Formatierungen festgelegt, nämlich der Radius (das Attribut `r`) und die Füllfarbe.

⓫ Schließlich wird die Position der einzelnen Kreise aufgrund des Werts bestimmt, den der zugehörige Datenpunkt hat. Die Attribute `cx` und `cy` der einzelnen `<circle>`-Elemente werden jeweils auf der Grundlage der Einträge in der Datendatei festgelegt. Statt eines festen Werts stellen wir *Zugriffsfunktionen* bereit, die zu einem gegebenen Eintrag der Datendatei (also zu einem einzeiligen Datensatz) den entsprechenden Wert zurückgeben.

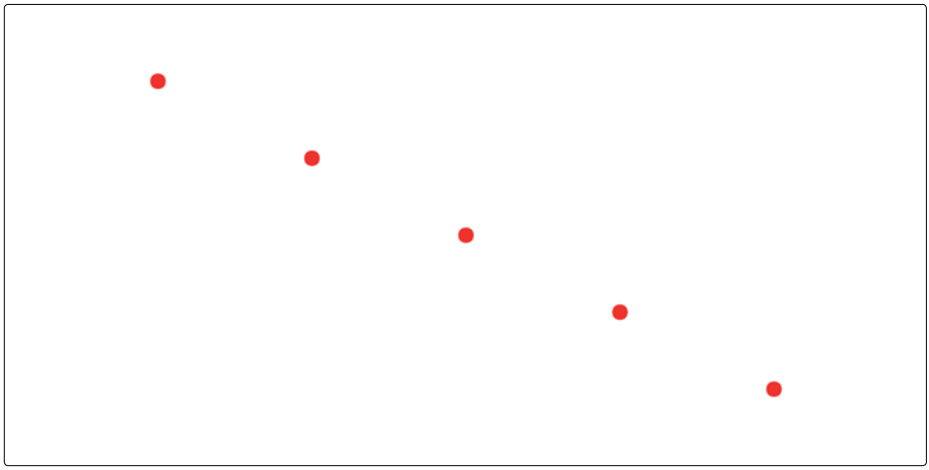

Abb. 2–1 *Grafische Darstellung einer einfachen Datenmenge (siehe Beispiel 2–3)*

Um ehrlich zu sein: Für einen so einfachen Graphen ist das ein erheblicher Aufwand! Hieran können Sie schon etwas erkennen, was im Laufe der Zeit noch deutlicher wird: D3 ist keine Grafikbibliothek und erst recht kein Werkzeug zur Diagrammerstellung, sondern eine Bibliothek, um den DOM-Baum zu bearbeiten und dadurch Informationen grafisch darzustellen. Sie werden die ganze Zeit damit beschäftigt sein, Operationen an Teilen des DOM-Baums vorzunehmen (die Sie mit Selections auswählen). Außerdem werden Sie feststellen, dass bei D3 nicht

gerade wenig Tipparbeit anfällt, da Sie Attributwerte bearbeiten und eine Zugriffsfunktion nach der anderen schreiben müssen. Andererseits ist der Code meiner Meinung nach zwar sehr wortreich, aber auch sauber und unkompliziert.

Wenn Sie ein wenig mit den Beispielen herumspielen, werden Sie womöglich noch einige weitere Überraschungen erleben. Beispielsweise ist die Funktion `tsv()` ziemlich wählerisch: Spalten *müssen* durch Tabulatoren getrennt sein, Weißraum wird *nicht* ignoriert, eine Kopfzeile *muss* vorhanden sein usw. Bei genauerer Untersuchung der Datenmenge und des Graphen werden Sie schließlich auch feststellen, dass das Diagramm nicht korrekt ist, sondern auf dem Kopf steht! Das liegt daran, dass SVG »grafische Koordinaten« verwendet, bei denen die horizontale Achse zwar wie üblich von links nach rechts, die vertikale aber von oben nach unten läuft.

Um diese ersten Eindrücke besser einordnen zu können, fahren wir mit einem zweiten Beispiel fort.

Zweites Beispiel: zwei Datenmengen

Für unser zweites Beispiel verwenden wir die Datenmenge aus Beispiel 2–4. Sie sieht zwar fast genauso harmlos aus wie die vorherige, aber bei genauerer Betrachtung werden einige zusätzliche Schwierigkeiten deutlich. Sie enthält nämlich nicht nur *zwei* Datenmengen (in den Spalten `y1` und `y2`), sondern umfasst auch Datenbereiche, die unserer Aufmerksamkeit bedürfen. Im vorherigen Beispiel konnten wir die Datenwerte unmittelbar als Pixelkoordinaten verwenden, doch die Werte der neuen Datenmenge erfordern eine Transformation, bevor sie als Bildschirmkoordinaten genutzt werden können. Wir müssen also etwas mehr Arbeit aufwenden.

Beispiel 2–4: Eine kompliziertere Datenmenge (examples-multiple.tsv)

```
x       y1      y2
1.0     0.001   0.63
3.0     0.003   0.84
4.0     0.024   0.56
4.5     0.054   0.22
4.6     0.062   0.15
5.0     0.100   0.08
6.0     0.176   0.20
8.0     0.198   0.71
9.0     0.199   0.65
```

Symbole und Linien darstellen

Die HTML-Seite für dieses Beispiel ist im Großen und Ganzen identisch mit derjenigen, die wir zuvor benutzt haben (Beispiel 2–2). Allerdings müssen wir die folgende Zeile ersetzen:

```
<script src="examples-demo1.js"></script>
```

Für dieses Beispiel muss sie wie folgt lauten, damit sie auf das neue Skript verweist:

```
<script src="examples-demo2.js"></script>
```

Auch der `onload`-Ereignishandler muss den neuen Funktionsnamen angeben:

```
<body onload="makeDemo2()">
```

Das Skript sehen Sie in Beispiel 2–5, das resultierende Diagramm in Abbildung 2–2.

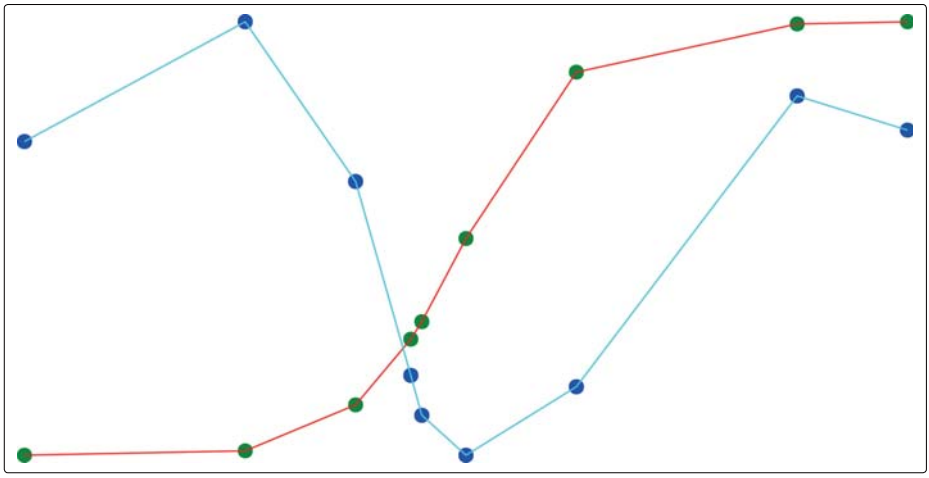

Abb. 2–2 *Einfaches Diagramm der Datenmenge aus Beispiel 2–4 (siehe auch Beispiel 2–5)*

Beispiel 2–5: Befehle für Abbildung 2–2

```
function makeDemo2() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var pxX = 600, pxY = 300;    ❶

            var scX = d3.scaleLinear()    ❷
                .domain( d3.extent(data, d => d["x"] ) )    ❸
                .range( [0, pxX] );
```

```
            var scY1 = d3.scaleLinear()   ❹
                .domain(d3.extent(data, d => d["y1"] ) )
                .range( [pxY, 0] );   ❺
            var scY2 = d3.scaleLinear()
                .domain( d3.extent(data, d => d["y2"] ) )
                .range( [pxY, 0] );

            d3.select( "svg" )   ❻
                .append( "g" ).attr( "id", "ds1" )   ❼
                .selectAll( "circle" )   ❽
                .data(data).enter().append("circle")
                .attr( "r", 5 ).attr( "fill", "green" )   ❾
                .attr( "cx", d => scX(d["x"]) )   ❿
                .attr( "cy", d => scY1(d["y1"]) );   ⓫

            d3.select( "svg" )   ⓬
                .append( "g" ).attr( "id", "ds2" )
                .attr( "fill", "blue" )   ⓭
                .selectAll( "circle" )   ⓮
                .data(data).enter().append("circle")
                .attr( "r", 5 )
                .attr( "cx", d => scX(d["x"]) )
                .attr( "cy", d => scY2(d["y2"]) );   ⓯

            var lineMaker = d3.line()   ⓰
                .x( d => scX( d["x"] ) )   ⓱
                .y( d => scY1( d["y1"] ) );

            d3.select( "#ds1" )   ⓲
                .append( "path" )   ⓳
                .attr( "fill", "none" ).attr( "stroke", "red" )
                .attr( "d", lineMaker(data) );   ⓴

            lineMaker.y( d => scY2( d["y2"] ) );   ㉑

            d3.select( "#ds2" )   ㉒
                .append( "path" )
                .attr( "fill", "none" ).attr( "stroke", "cyan" )
                .attr( "d", lineMaker(data) );

            // d3.select( "#ds2" ).attr( "fill", "red" );   ㉓
        } );
}
```

❶ Wir weisen die Abmessungen des eingebetteten SVG-Bereichs Variablen zu (`px` für Pixel), um später darauf verweisen zu können. Es ist natürlich auch möglich, die Größenangabe aus dem HTML-Dokument herauszunehmen und sie stattdessen über JavaScript festzulegen. (Probieren Sie es aus!)

❷ In D3 gibt es *Skalierungsobjekte*, die den eingegebenen *Definitionsbereich* auf den ausgegebenen *Wertebereich* abbilden. Hier verwenden wir lineare Skalierungen, um die Datenwerte aus ihrem natürlichen Definitionsbereich in den Wertebereich der Pixel im Graphen zu übertragen. Die Bibliothek enthält jedoch auch logarithmische und exponentielle Skalierungen und sogar solche, die die Zahlenbereiche auf Farben abbilden, um Falschfarbendiagramme und Heatmaps zu erstellen (siehe Kapitel 7 und 8). Skalierungen sind Funktionsobjekte: Sie werden mit einem Wert des Definitionsbereichs aufgerufen und geben den skalierten Wert zurück.

❸ Sowohl der Definitions- als auch der Wertebereich werden als zweielementiges Array angegeben. Bei `d3.extent()` handelt es sich um eine Komfortfunktion, die ein Array (von Objekten) entgegennimmt und den größten und den kleinsten Wert als zweielementiges Array zurückgibt (siehe Kapitel 10). Um den gewünschten Wert aus den Objekten im Eingabearray zu entnehmen, müssen wir eine Zugriffsfunktion bereitstellen (ähnlich wie im letzten Schritt des vorherigen Beispiels). Um uns Tipparbeit zu ersparen, nutzen wir hier (und in den meisten folgenden Beispielen!) die *Pfeilfunktionen* von JavaScript (siehe Anhang C).

❹ Da die drei Spalten in der Datenmenge unterschiedliche Wertebereiche aufweisen, brauchen wir drei Skalierungsobjekte, eines für jede Spalte.

❺ Um die auf dem Kopf stehende Ausrichtung des SVG-Koordinatensystems auszugleichen, kehren wir für die vertikale Achse den Ausgabebereich in der Definition des Skalierungsobjekts um.

❻ Wir wählen das `<svg>`-Element aus, um Symbole für die erste Datenmenge hinzuzufügen.

❼ Dieser Vorgang ist neu: Bevor wir irgendwelche Elemente des Graphen hinzufügen, hängen wir ein `<g>`-Element an und geben ihm einen eindeutigen Bezeichner mit. Das endgültige Element sieht wie folgt aus:

```
<g id="ds1">...</g>
```

Das SVG-Element `<g>` sorgt für eine logische Gruppierung, sodass wir auf *alle* Symbole für die erste Datenmenge gleichzeitig verweisen und sie von denen der zweiten Datenmenge unterscheiden können (siehe Anhang B und den Kasten »*Das praktische <g>-Element*« auf *S. 115*).

❽ Wie zuvor erstellen wir mit `selectAll( "circle" )` eine leere Platzhaltersammlung. Die `<circle>`-Elemente werden als Kinder des gerade hinzugefügten `<g>`-Elements erstellt.

❾ Feste Formatierungen werden unmittelbar auf die einzelnen `<circle>`-Elemente angewendet.

❿ Eine Zugriffsfunktion wählt die geeignete Spalte für die horizontale Achse aus. Beachten Sie, dass der Skalierungsoperator auf die Daten angewendet wird, bevor sie zurückgegeben werden.

⓫ Eine Zugriffsfunktion wählt die geeignete Spalte für die erste Datenmenge aus. Auch hier erfolgt wieder eine passende Skalierung.

⓬ Das schon bekannte Verfahren wird erneut angewandt, um Elemente für die zweite Datenmenge hinzuzufügen. Beachten Sie aber die Unterschiede!

⓭ Für die zweite Datenmenge wird die Füllfarbe mit dem Attribut `fill` des `<g>`-Elements angegeben. Dieses Erscheinungsbild werden die Kinder erben. Durch die Definition im Elternelement können wir das Aussehen aller Kinder später in einem Rutsch ändern.

⓮ Abermals wird eine leere Sammlung für die neu hinzugefügten `<circle>`-Elemente erstellt. Hier ist das `<g>`-Element zu mehr als nur zu unserer Bequemlichkeit da: Wenn wir an dieser Stelle `selectAll( "circle" )` für das `<svg>`-Element aufriefen, würden wir keine leere Sammlung, sondern die `<circle>`-Elemente der *ersten* Datenmenge erhalten. Anstatt neue Elemente hinzuzufügen, würden wir die vorhandenen bearbeiten und die erste Datenmenge mit der zweiten überschreiben. Das `<g>`-Element erlaubt es uns, klar zwischen den Elementen und ihrer Zuordnung zu Datenmengen zu unterscheiden. (Das wird noch deutlicher, wenn wir uns in Kapitel 3 eingehender mit der `Selection`-API von D3 beschäftigen.)

⓯ Die Zugriffsfunktion wählt jetzt eine geeignete Spalte für die zweite Datenmenge aus.

⓰ Um die beiden Datenmengen besser unterscheiden zu können, wollen wir die Symbole, die zu einer Menge gehören, jeweils durch gerade Linien verbinden. Diese Linien sind komplizierter als die Symbole, da jedes Liniensegment von *zwei* aufeinanderfolgenden Datenpunkten abhängt. D3 kommt uns dabei aber zu Hilfe: Die Factory-Funktion `d3.line()` gibt ein Funktionsobjekt zurück, das bei Übergabe einer Datenmenge einen für das Attribut `d` des SVG-Elements `<path>` geeigneten String produziert. (Mehr über das Element `<path>` und seine Syntax lernen Sie in Anhang B.)

⓱ Der Liniengenerator erfordert eine Zugriffsfunktion, um die horizontalen und vertikalen Koordinaten für jeden Datenpunkt zu entnehmen.

⓲ Das `<g>`-Element der ersten Datenmenge wird anhand des Werts seines `id`-Attributs ausgewählt. Ein ID-Selektorstring besteht aus dem Nummernzeichen (`#`), gefolgt von dem Attributwert.

⓳ Ein `<path>`-Element wird als Kind der Gruppe `<g>` der ersten Datenmenge hinzugefügt ...

⓴ ... und sein Attribut `d` wird durch den Aufruf des Liniengenerators für die Datenmenge festgelegt.

㉑ Anstatt einen komplett neuen Liniengenerator zu erstellen, verwenden wir den vorhandenen weiter. Dabei geben wir eine neue Zugriffsfunktion an, diesmal für die zweite Datenmenge.

㉒ Ein `<path>`-Element für die zweite Datenmenge wird an der passenden Stelle im SVG-Baum angehängt und gefüllt.

㉓ Da der Füllungsstil für die Symbole der zweiten Datenmenge im Elternelement definiert wurde (nicht in den einzelnen `<circle>`-Elementen), ist es möglich, ihn in einer einzigen Operation zu ändern. Wenn Sie die Auskommentierung dieser Zeile aufheben, werden alle Kreise für die zweite Datenmenge rot dargestellt. Nur Optionen für das Erscheinungsbild werden vom Elternelement geerbt. Dagegen ist es nicht möglich, etwa den Radius aller Kreise auf diese Weise zu ändern oder die Kreise in Rechtecke umzuwandeln. Für solche Operationen müssen alle betroffenen Elemente einzeln angefasst werden.

Inzwischen haben Sie wahrscheinlich schon einen Eindruck von der Arbeit mit D3 gewonnen. Der Code ist zwar sehr wortreich, aber der ganze Vorgang hat auch etwas von einem Baukastensystem, bei dem sie einfach vorgefertigte Elemente zusammenstecken. Insbesondere die Methodenkettung kann wie die Zusammenstellung einer Unix-Pipeline wirken (oder wie das Bauen mit LEGO-Steinen). Bei den Komponenten selbst liegt der Schwerpunkt gewöhnlich mehr auf den Mechanismen als auf der Strategie. Dadurch können Sie für viele verschiedene Zwecke wiederverwendet werden, allerdings werden Programmierer oder Designer zusätzlich mit der Verantwortung belastet, semantisch sinnvolle grafische Konstruktionen zusammenzustellen. Als letzten Gesichtspunkt möchte ich noch hervorheben, dass D3 Entscheidungen zugunsten einer »späten Bindung« verzögert. So werden beispielsweise gewöhnlich Zugriffsfunktionen als Argumente übergeben, anstatt zu verlangen, dass die passenden Spalten vor der Übergabe an das Rendering-Framework aus der ursprünglichen Datenmenge extrahiert werden.

Diagrammelemente mit wiederverwendbaren Komponenten hinzufügen

Der Graph aus Abbildung 2–2 ist ziemlich spartanisch: Er zeigt nur die Daten und nichts sonst. Insbesondere gibt er die Skalen nicht an, was hier besonders wichtig wäre, da für die beiden Datensätze unterschiedliche Wertebereiche gelten. Ohne Skalen oder Achsen ist es nicht möglich, quantitative Informationen aus diesem

Graph (oder überhaupt aus irgendeinem Diagramm) abzulesen. Daher müssen wir ihm Achsen hinzufügen.

Aufgrund der Teilstriche und Teilstrichbeschriftungen sind Achsen komplexe grafische Elemente. Zum Glück gibt es in D3 eine Funktionalität zum Anlegen von Achsen, die bei Übergabe eines Skalierungsobjekts (das den Definitions- und den Wertebereich sowie die Abbildung zwischen ihnen definiert) alle erforderlichen grafischen Elemente erstellt und zeichnet. Da die grafische Achsenkomponente aus vielen einzelnen SVG-Elementen besteht (für die Teilstriche und Beschriftungen), sollte sie immer in ihrem eigenen `<g>`-Container erstellt werden. Die auf dieses Elternelement angewandten Formatierungen und Transformationen werden von allen Teilen der Achse geerbt. Das ist wichtig, da sich alle Achsen zu Anfang im Ursprung befinden (in der oberen linken Ecke) und mithilfe des Attributs `transform` zu ihrer gewünschten Position verschoben werden müssen. (Genauere Erläuterungen zu Achsen folgen in Kapitel 7.)

Neben dem Hinzufügen von Achsen und der bis jetzt noch nicht gezeigten D3-Funktionalität zum Erstellen von Kurven demonstriert Beispiel 2–6 auch eine andere Arbeitsweise mit D3. Der Code in Beispiel 2–5 war geradlinig, aber umfangreich, und enthielt eine Menge Wiederholungen. Beispielsweise war der Code zum Erstellen der drei verschiedenen Skalierungsobjekte fast identisch. Auch der Code für die Symbole und Linien wurde für die zweite Datenmenge größtenteils dupliziert. Die Vorteile dieses Stils sind Einfachheit und ein linearer logischer Verlauf, was allerdings mit einer größeren Wortfülle erkauft wird.

In Beispiel 2–6 gibt es weniger redundanten Code, da die mehrfach verwendeten Anweisungen in lokale Funktionen gepackt wurden. Da diese Funktionen innerhalb von `makeDemo3()` definiert sind, haben sie Zugriff auf die Variablen in diesem Gültigkeitsbereich. Das hilft dabei, die Anzahl der erforderlichen Argumente für die lokalen Hilfsfunktionen gering zu halten. Dieses Beispiel stellt auch *Komponenten* als Grundbausteine der Kapselung und Wiederverwendung vor und zeigt »synthetische« Funktionsaufrufe – lauter wichtige Techniken für die Arbeit mit D3.

Beispiel 2–6: Befehle für Abbildung 2–3

```
function makeDemo3() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var svg = d3.select( "svg" );   ❶

            var pxX = svg.attr( "width" );   ❷
            var pxY = svg.attr( "height" );
```

```
        var makeScale = function( accessor, range ) {    ❸
            return d3.scaleLinear()
                .domain( d3.extent( data, accessor ) )
                .range( range ).nice();
        }
        var scX = makeScale( d => d["x"], [0, pxX] );
        var scY1 = makeScale( d => d["y1"], [pxY, 0] );
        var scY2 = makeScale( d => d["y2"], [pxY, 0] );

        var drawData = function( g, accessor, curve ) {    ❹
            // Zeichnet Kreise
            g.selectAll( "circle" ).data(data).enter()
                .append("circle")
                .attr( "r", 5 )
                .attr( "cx", d => scX(d["x"]) )
                .attr( "cy", accessor );

            // Zeichnet Linien
            var lnMkr = d3.line().curve( curve )    ❺
                .x( d=>scX(d["x"]) ).y( accessor );

            g.append( "path" ).attr( "fill", "none" )
                .attr( "d", lnMkr( data ) );
        }

        var g1 = svg.append( "g" );    ❻
        var g2 = svg.append( "g" );

        drawData( g1, d => scY1(d["y1"]), d3.curveStep );    ❼
        drawData( g2, d => scY2(d["y2"]), d3.curveNatural );

        g1.selectAll( "circle" ).attr( "fill", "green" );    ❽
        g1.selectAll( "path" ).attr( "stroke", "cyan" );

        g2.selectAll( "circle" ).attr( "fill", "blue" );
        g2.selectAll( "path" ).attr( "stroke", "red" );

        var axMkr = d3.axisRight( scY1 );    ❾
        axMkr( svg.append("g") );    ❿

        axMkr = d3.axisLeft( scY2 );

        svg.append( "g" )
            .attr( "transform", "translate(" + pxX + ",0)" )    ⓫
            .call( axMkr );    ⓬

        svg.append( "g" ).call( d3.axisTop( scX ) )
            .attr( "transform", "translate(0,"+pxY+")" );    ⓭
    } );
}
```

❶ Das <svg>-Element wird gezeichnet und einer Variablen zugewiesen, sodass es später verwendet werden kann, ohne `select()` aufrufen zu müssen.

❷ Als Nächstes wird die Größe des <svg>-Elements abgefragt. Viele D3-Funktionen können sowohl als Set- als auch als Get-Methoden dienen: Wird ein zweites Argument angegeben, so wird die genannte Eigenschaft auf diesen Wert gesetzt; anderenfalls wird der aktuelle Wert der Eigenschaft zurückgegeben. Hier nutzen wir diesen Umstand, um die Größe des <svg>-Elements zu ermitteln.

❸ Bei der Funktion `makeScale()` handelt es sich lediglich um einen komfortablen Wrapper, um D3-Funktionsaufrufe knapper formulieren zu können. Skalierungsobjekte kennen Sie bereits aus dem vorhergehenden Listing (Beispiel 2–5). Wird die Funktion `nice()` für ein Skalierungsobjekt aufgerufen, so erweitert sie den Wertebereich auf den nächsten »runden« Wert.

❹ Die Funktion `drawData()` kombiniert alle erforderlichen Befehle, um eine einzige Datenmenge auszugeben: Sie erstellt sowohl die Kreise für die einzelnen Datenpunkte als auch die verbindenden Linien. Beim ersten Argument von `drawData()` muss es sich um eine Selection handeln, gewöhnlich ist das ein <g>-Element als Behälter für alle grafischen Elemente zur Darstellung einer einzelnen Datenmenge. Funktionen, die eine Selection als erstes Argument übernehmen und dann Elemente zu ihr hinzufügen, werden als *Komponenten* bezeichnet und bilden einen wichtigen Mechanismus für Kapselung und Codewiederverwendung in D3. Dies ist das erste Beispiel dafür. Das Anlegen der Achsen weiter hinten in diesem Code ist ein weiteres. (Eine ausführliche Darstellung finden Sie in Kapitel 5.)

❺ Die Factory `d3.line()` kennen Sie bereits aus Beispiel 2–5. Sie kann einen Algorithmus entgegennehmen, der definiert, mit welcher Art von Kurve aufeinanderfolgende Punkte verbunden werden sollen. Als Standard werden gerade Linien verwendet, aber D3 definiert noch viele andere Algorithmen. Außerdem können Sie Ihre eigenen schreiben. (Wie das geht, erfahren Sie in Kapitel 5.)

❻ Hier werden die beiden <g>-Container für die beiden Datenmengen erstellt.

❼ Bei ihrem Aufruf werden der Funktion `drawData()` eines der Containerelemente, eine Zugriffsfunktion, die die Datenmenge angibt, und die gewünschte Kurvenform übergeben. Nur um zu zeigen, was alles möglich ist, verwenden wir für die beiden Datenmengen hier zwei unterschiedliche Arten von Kurven, nämlich eine mit Stufenfunktionen und die andere mit natürlichen kubischen Splines. Die Funktion `drawData()` fügt die erforderlichen <circle>- und <path>-Elemente zum <g>-Behälter hinzu.

❽ Für jeden Behälter werden die gewünschten grafischen Elemente ausgewählt, um ihre Farbe festzulegen. Obwohl es sehr einfach möglich gewesen wäre, wurde die Auswahl der Farbe absichtlich nicht in die Funktion `drawData()` aufgenommen. Das steht im Einklang mit einem üblichen D3-Idiom: Das Erstellen von DOM-Elementen wird von der Konfiguration ihres Erscheinungsbilds getrennt gehalten.

❾ Die Achse für die erste Datenmenge wird an der linken Seite des Graphen gezeichnet. Denken Sie daran, dass alle Achsen standardmäßig im Ursprung dargestellt werden. Das Objekt `axisRight` zeichnet Teilstrichbeschriftungen *rechts* neben die Achse, sodass sie sich außerhalb des Graphen befinden, wenn die Achse an dessen rechter Seite platziert wird. Hier dagegen geben wir die Achse links aus, sodass sich die Teilstrichbeschriftungen innerhalb des Diagramms befinden.

❿ Die Factory-Funktion `d3.axisRight( scale )` gibt ein Funktionsobjekt zurück, das die Achse mit all ihren Teilen generiert. Sie erfordert einen SVG-Container (gewöhnlich ein `<g>`-Element) als Argument und erstellt alle Elemente der Achse als Kinder dieses Containerelements. Mit anderen Worten, es handelt sich dabei um eine *Komponente* nach der weiter vorn gegebenen Definition. (Mehr darüber erfahren Sie in Kapitel 7.)

⓫ Für die Achse auf der rechten Seite des Graphen muss das Containerelement an die gewünschte Stelle verlegt werden. Dazu wird das SVG-Attribut `transform` verwendet.

⓬ Hier geschieht etwas Neues: Anstatt die Funktion `asMkr` explizit mit dem einschließenden `<g>`-Element als Argument aufzurufen, übergeben wir sie als Argument an die Funktion `call()`, die zur API `Selection` gehört (siehe Kapitel 3) und nach einer ähnlichen Funktionalität in der Sprache JavaScript gestaltet wurde. Sie ruft ihr Argument (bei dem es sich um eine Funktion handeln muss) auf und übergibt ihr die aktuelle Selection als Argument. Diese Art von »synthetischem« Funktionsaufruf ist in JavaScript sehr gebräuchlich, weshalb es sinnvoll ist, sich daran zu gewöhnen. Ein Vorteil dieses Programmierstils besteht darin, dass er eine Verkettung von Methoden ermöglicht, wie Sie in der nächsten Zeile sehen.

⓭ Hier fügen wir die horizontale Achse am unteren Rand des Graphen hinzu. Nur um zu demonstrieren, was alles möglich ist, wurde die Reihenfolge der Funktionsaufrufe umgekehrt: Die Achsenkomponente wird zuerst aufgerufen, und erst danach wird die Transformation angewendet. An dieser Stelle brau-

chen wir auch die Hilfsvariable `axMkr` nicht mehr. Dies ist wahrscheinlich die idiomatischste Weise, um diesen Code zu schreiben.[3]

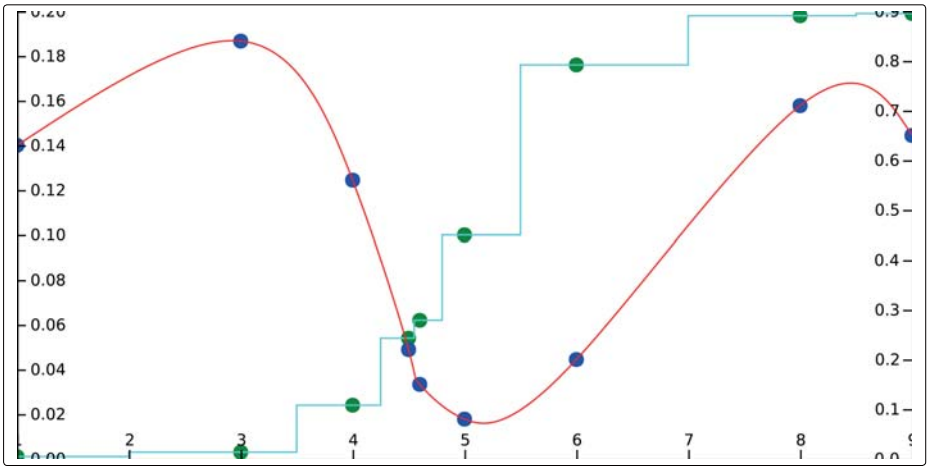

Abb. 2–3 *Ein verbessertes Diagramm mit Koordinatenachsen und verschiedenen Arten von Kurven (vgl. Abb. 2–2 und Beispiel 2–6)*

Den resultierenden Graphen sehen Sie in Abbildung 2–3. Er ist nicht schön, aber er erfüllt seinen Zweck und zeigt die beiden Datenmengen jeweils mit den zugehörigen Maßstäben. Um das Aussehen des Graphen zu verbessern, könnte in einem ersten Schritt um die eigentlichen Daten des SVG-Bereichs herum noch etwas Platz für die Achsen und Teilstrichbeschriftungen geschaffen werden. (Probieren Sie es aus!)

Drittes Beispiel: Listeneinträge animieren

Unser drittes und letztes Beispiel mag etwas seltsamer sein als die beiden ersten, veranschaulicht aber zwei wichtige Dinge:

- D3 ist nicht darauf beschränkt, SVG-Grafiken zu generieren, sondern kann *alle* Teile des DOM-Baums bearbeiten. In diesem Beispiel wollen wir damit herkömmliche HTML-Listenelemente gestalten.
- Mit D3 ist es einfach, reaktive und animierte Dokumente zu erstellen, die auf Benutzerereignisse (wie Mausklicks) reagieren bzw. ihr Erscheinungsbild mit der Zeit ändern. Hier zeige ich nur einen winzigen Ausschnitt der erstaunlichen Möglichkeiten. Ausführlicher werden wir dieses Thema noch in Kapitel 4 behandeln.

3 Auch die Funktion `drawData()` wird gewöhnlich auf diese Weise aufgerufen: `g1.call( drawData, d => scY1(d["y1"]), d3.curveStep ).`

HTML-Elemente mit D3 erstellen

Zur Abwechslung – und weil das Skript sehr kurz ist – sind die JavaScript-Befehle hier unmittelbar in der HTML-Seite und nicht in einer eigenen Datei enthalten. Die vollständige Seite für das aktuelle Beispiel einschließlich der D3-Befehle finden Sie in Beispiel 2–7 (siehe auch Abb. 2–4).

Beispiel 2–7: D3 für nicht grafische HTML-Bearbeitung verwenden (siehe Abb. 2–4)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>
  <script>
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];   ❶

    d3.select( "body" )   ❷
        .append( "ul" ).selectAll( "li" )   ❸
        .data(vs).enter()   ❹
        .append("li").text( d => d );   ❺
}
  </script>
</head>

<body onload="makeList()" />
</html>
```

Die Struktur dieses Beispiels ist fast identisch mit derjenigen des Beispiels am Anfang dieses Kapitels (siehe insbesondere Beispiel 2–3), allerdings gibt es auch einige Abweichungen im Detail:

❶ Die Datenmenge wird nicht aus einer externen Datei geladen, sondern im Code selbst definiert.

❷ Als äußerster Container wird das HTML-Element `<body>` ausgewählt.

❸ Der Code hängt ein `<ul>`-Element an und erstellt dann (mit `selectAll( "li" )`) einen leeren Platzhalter für die Listeneinträge.

❹ Wie zuvor wird die Datenmenge an die Selection gebunden und die Sammlung der Datenpunkte ohne übereinstimmende DOM-Elemente abgerufen.

❺ Schließlich wird für jeden Datenpunkt ein Listeneintrag angehängt, dessen Inhalt (in diesem Beispiel der Text des Listeneintrags) jeweils mit Werten aus der Datenmenge gefüllt wird.

All das ähnelt sehr stark dem, was wir schon zuvor getan haben. Der Unterschied besteht darin, dass das Ergebnis einfaches HTML in Textform ist. D3 ist also auch ein hervorragendes Werkzeug für die allgemeine (also nicht nur die grafische) DOM-Bearbeitung.

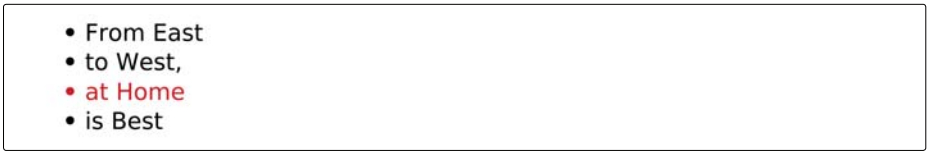

Abb. 2–4 *Eine Liste mit Aufzählpunkten in HTML (siehe Beispiele 2–7 und 2–8)*

Eine einfache Animation erstellen

Es kostet nicht viel Mühe, dafür zu sorgen, dass dieses Dokument auf Benutzerereignisse reagiert. Wenn Sie die Funktion `makeList()` in Beispiel 2–7 durch diejenige aus Beispiel 2–8 ersetzen, können Sie die Textfarbe von Schwarz in Rot und zurück ändern, indem Sie auf ein Listenelement klicken. Die Änderung erfolgt allerdings nicht sofort, sondern zieht sich über einige Sekunden hin.

Beispiel 2–8: Animation als Reaktion auf Benutzerereignisse

```
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];

    d3.select( "body" )
        .append( "ul" ).selectAll( "li" )
        .data(vs).enter()
        .append("li").text( d => d )    ❶
        .on( "click", function () {    ❷
            this.toggleState = !this.toggleState;    ❸ ❹
            d3.select( this )    ❺
                .transition().duration(2000)    ❻
                .style("color", this.toggleState?"red":"black");    ❼
        } );
}
```

❶ Bis zu diesem Punkt ist die Funktion identisch mit der aus Beispiel 2–7.

❷ Die Funktion `on()` registriert einen Callback für den genannten Ereignistyp (in diesem Fall `"click"`) mit dem aktuellen Element als DOM-Ereignisziel (`EventTarget`). Jeder Listeneintrag kann jetzt Klickereignisse empfangen und übergibt sie an den bereitgestellten Callback.

❸ Wir müssen uns den jeweils aktuellen Zustand der einzelnen Listeneinträge merken. Wo sonst können wir diese Information ablegen als in dem Element selbst? Die sehr liberale Haltung von JavaScript macht das extrem einfach: Wir fügen einfach einen neuen Member zu dem Element hinzu! Vor dem Aufruf des Callbacks weist D3 das aktive DOM-Element `this` zu und ermöglicht so den Zugriff auf dieses Element.

❹ Diese Zeile nutzt die liberale Haltung von JavaScript auch noch für einen anderen Zweck. Beim ersten Aufruf des Callbacks (für jeden Listeneintrag) ist `toggleState` noch nicht zugewiesen und hat daher den besonderen Wert `undefined`, der in einem booleschen Kontext zu `false` ausgewertet wird. Daher ist es nicht nötig, die Variable zu initialisieren.

❺ Um mithilfe der Methodenverkettung auf dem aktuellen Knoten arbeiten zu können, müssen wir ihn in eine Selection einschließen.

❻ Die Methode `transition()` interpoliert nahtlos zwischen dem aktuellen Status des ausgewählten Elements (in diesem Fall dem aktuellen Listeneintrag) und dessen gewünschtem endgültigem Erscheinungsbild. Für das Intervall für den Übergang sind 2000 ms festgelegt. D3 kann zwischen Farben (über deren numerische Darstellung) und vielen anderen Größen interpolieren. (Die Interpolation wird in Kapitel 7 behandelt.)

❼ Schließlich wird die neue Textfarbe auf der Grundlage des aktuellen Zustands der Statusvariablen ausgewählt.

8

Farben, Farbskalen und Heatmaps

Bei der Datenvisualisierung können Farben verschiedene Zwecke erfüllen: So können sie einfach dazu dienen, den Graphen optisch ansprechender zu gestalten, aber auch bestimmte Aspekte der Daten betonen oder sogar die Hauptträger der Informationen sein. In diesem Kapitel schauen wir uns an, wie Farben in D3 dargestellt werden, und besprechen dann verschiedene Farbschemas und ihre Verwendung zur Informationsvermittlung. Am Ende des Kapitels folgt eine Beschreibung von Falschfarbendiagrammen, in denen die Daten ausschließlich mithilfe von Farben dargestellt werden.

Farben und Farbräume

Eine einzelne Farbe lässt sich in D3 ganz einfach angeben. Dazu müssen Sie lediglich mit der CSS3-Syntax (siehe Anhang B) einen String mit einem vordefinierten Farbnamen, den RGB-Komponenten (Rot, Grün, Blau) oder den HSL-Komponenten (Hue, Saturation, Lightness, also Farbton, Sättigung, Helligkeit) übergeben. Wenn Sie aber Farben programmgesteuert bearbeiten wollen, ist das Stringformat dazu nicht sehr gut geeignet. Für diese Zwecke können Sie mit einer der Factory-

Funktionen aus Tabelle 8–1 ein `color`-Objekt erstellen und dann überall dort verwenden, wo eine Farbangabe erwartet wird. Seine Funktion `toString()` wird automatisch aufgerufen und gibt eine Darstellung der Farbe im CSS3-Format zurück.

Die API von `color`-Objekten ist allerdings stark eingeschränkt. Hauptsächlich dienen sie als Container für ihre Kanalinformationen. Jedes `color`-Objekt stellt seine drei Komponenten als entsprechend benannte Eigenschaften zur Verfügung, eine für jeden Kanal. Außerdem verfügt jedes Objekt über die Eigenschaft `opacity` für den Alphakanal.

Die Funktionen aus Tabelle 8–1 können die folgenden Argumente entgegennehmen und geben ein `color`-Objekt aus dem angeforderten Farbraum zurück:

- Ein CSS3-Farbstring
- Ein anderes `color`-Objekt (zur Umwandlung in einen anderen Farbraum)
- Eine Menge von drei Komponenten (bzw. vier bei Angabe eines Opazitätswerts)

Eine Ausnahme bildet die generische Factory `d3.color()`, die einen CSS3-String oder ein anderes `color`-Objekt entgegennimmt und je nach Eingabe ein RGB- oder HSL-`color`-Objekt zurückgibt.

Funktion	Komponenten[a]	Bemerkungen
`d3.rgb()`	`r`, `g`, `b`	$0 \le r, g, b \le 255$ (streng)
`d3.hsl()`	`h`, `s`, `l`	• h: Beliebiger (positiver oder negativer) Wert; der Farbton wird als modulo 360 aufgefasst. • $0 \le s, l \le 1$ (streng)
`d3.lab()`	`l`, `a`, `b`	• $0 \le l \le 100$ (streng) • $-100 \le a, b \le 100$ (genähert)
`d3.hcl()`	`h`, `c`, `l`	• h: Beliebiger (positiver oder negativer) Wert; der Farbton wird als modulo 360 aufgefasst. • $-125 \le c \le 125$ (genähert) • $0 \le l \le 100$ (genähert)
`d3.color()`	`r`, `g`, `b` oder `h`, `s`, `l`	Die Eingabe muss ein CSS3-String oder ein anderes Objekt sein. Die Ausgabe ist ein RGB-Objekt, es sei denn, die Eingabe liegt im HSL-Format vor.

[a] Alle `color`-Objekte verfügen auch über eine `opacity`-Komponente mit Werten von 0 (transparent) bis 1 (opak).

Tab. 8–1 *Factory-Funktionen zum Erstellen von* `color`*-Objekten in verschiedenen Farbräumen mit den Farbkomponenten des zurückgegebenen Objekts und den zulässigen Bereichen*

Die zulässigen Parameterbereiche unterscheiden sich im Allgemeinen je nach Farbraum. Bei offensichtlich unzulässigen Eingaben (etwa einer negativen RGB-Komponente) geben die Funktionen aus Tabelle 8–1 `null` zurück, aber auch wenn der Rückgabewert nicht `null` ist, ist nicht garantiert, dass die zurückgegebene Farbe auch *darstellbar* ist. Um das herauszufinden, können Sie die Funktion `displayable()` auf die zurückgegebene Farbinstanz anwenden. Kann eine Farbe nicht angezeigt werden, dann ersetzt ihre Methode `toString()` diese durch eine »passende« darstellbare Farbe (z. B. durch Weiß oder Schwarz, wenn die erforderliche Farbe zu hell oder zu dunkel ist).

Farbräume

Im Allgemeinen sind zur Angabe einer Farbe drei Koordinaten erforderlich (ohne Berücksichtigung der Transparenz). Im RGB-Modell wird der Farbraum als Würfel dargestellt, der von den drei Komponentenachsen im rechten Winkel aufgespannt wird. Der RGB-Farbraum kommt der technischen Umsetzung in der Hardware am nächsten, ist aber dafür berüchtigt, alles andere als intuitiv verständlich zu sein. (Oder können Sie auf Anhieb sagen, wie `#b8860b` aussieht?)

Im HSL-Modell wird der RGB-Würfel zu einem Doppelkegel verformt, dessen Achse von Schwarz nach Weiß entlang der Raumdiagonale des ursprünglichen Würfels verläuft. Die drei Komponenten werden jetzt als Zylinderkoordinaten aufgefasst. Der Farbton gibt den Winkel um die Zylinderachse an, die Helligkeit die Position entlang der Achse (von Schwarz am Boden bis zu Weiß an der Spitze) und die Sättigung den radialen Abstand von der Achse (wobei Grautöne die Achse bilden und völlig gesättigte Farben die Kegeloberfläche). Völlig gesättigte Farben mit maximaler Helligkeit sind auf halber Höhe am »Bauch« des Doppelkegels zu finden.

Im Vergleich zu RGB- lassen sich HSL-Koordinaten etwas intuitiver deuten, allerdings leiden beide Darstellungsarten unter demselben Mangel: Ihre Farbkomponenten geben die Intensitäten des Renderinggeräts wieder, ohne die ungleichmäßige menschliche *Farbwahrnehmung* zu berücksichtigen. Das führt zu einer Reihe von Unstimmigkeiten. Betrachten Sie beispielsweise die drei Farben mit den CSS3-Farbnamen DarkKhaki (`#bdb76b`), Gold (`#ffd700`) und Yellow (`#ffff00`). Von diesen dreien hat DarkKhaki im HSL-Raum die größte Helligkeit, obwohl sie dunkler erscheint als die anderen. Gelb und Gold haben gleiche Helligkeitswerte, wirken aber völlig unterschiedlich. Es lassen sich noch leicht weitere Beispiele finden, in denen die Helligkeitswerte nicht mit der Wahrnehmung übereinstimmen.

➥

Farbräume, die auf den *wahrgenommenen* Intensitäten basieren, sollen solche Unstimmigkeiten zu vermeiden helfen. Zwei davon sind CIELAB (oder LAB) und HCL (Hue, Chroma, Luminance). Im LAB-Farbraum spannen die Koordinatenachsen ein Parallelepiped auf, wobei `l` die Helligkeit misst und die Koordinaten `a` und `b` die Position entlang von zwei Achsen angeben, von denen die eine von Rot nach Grün und die andere von Blau nach Gelb verläuft. (Im Standardfarbkreis stehen diese beiden Achsen angenähert senkrecht aufeinander.) Der HCL-Farbraum wandelt dieselben Informationen in Zylinderkoordinaten ähnlich denen des HSL-Raums um, wobei der Chroma-Wert angibt, wie farbig eine Farbe erscheint, und die Luminanz ein Maß für die Leuchtkraft ist. Der HCL-Raum lässt sich ebenso intuitiv nutzen wie der vertraute HSL-Raum, allerdings funktionieren Farbvergleiche mit HSL-Komponenten besser.

Ein Problem bei der Verwendung von Farbkomponenten, die sich auf die menschliche Wahrnehmung statt auf die technischen Aspekte stützen, besteht darin, dass es damit ziemlich einfach ist, Farben zu konstruieren, die sich mit regulärer Hardware nicht anzeigen lassen. Anders als bei RGB und HSL sind die Bereiche der Komponenten nicht auf einen festen Satz von Werten beschränkt. Daher muss sorgfältig darauf geachtet werden, dass die resultierende Farbe auch tatsächlich darstellbar ist.

Farbschemas

Im Lieferumfang von D3 ist eine ziemlich große Menge von Farbschemas zur Verwendung mit Skalierungsobjekten enthalten. Da es schwierig sein kann, sich in diesen vielen Schemas zurechtzufinden, gebe ich Ihnen hier eine Übersicht über die verfügbaren Farbschemas.[1]

Kartografieschemas

Die folgenden Schemas basieren auf solchen, die ursprünglich für Landkarten entwickelt wurden, um politische und andere thematische Informationen darzustellen. Sie eignen sich am besten für *Flächen*, aber weniger gut für Linien oder für Formen mit winzigen Details. Da sie entworfen wurden, um ein angenehmes Erscheinungsbild hervorzurufen, sind sie auch oft eine gute Wahl, um die Wahrnehmung von Informationen, die bereits durch andere Mittel dargestellt werden, zu *verstärken*.

1 Farbige Darstellungen finden Sie auf *https://github.com/d3/d3-scale-chromatic*.

Kategorieschemas
: Bei diesen neun nicht semantischen Kategorieschemas mit jeweils acht bis zwölf einzelnen Farben weisen benachbarte Farben gewöhnlich einen starken Kontrast auf. Die einzige Ausnahme davon bildet das Schema `d3.schemePaired` aus sechs Paaren, die jeweils aus einer hellen und einer dunklen Nuance mit vergleichbarem Farbton bestehen (Beispiele siehe Abb. 5–7 und Abb. 9–3).

Divergierende Schemas
: Es gibt neun divergierende Schemas jeweils mit dunklen und gesättigten Farben unterschiedlichen Farbtons an den Enden und einem Verlauf über helle und blasse Versionen von Grau, Weiß oder Gelb in der Mitte. Mit diesen Farbschemas können Sie Abweichungen von einem Grundwert in beide Richtungen darstellen (siehe Abb. 8–1).

Sequenzielle Schemas mit einem Farbton
: Diese sechs Schemas verlaufen jeweils von einem hellen und blassen Weiß oder Grau zu einer dunklen und gesättigten Farbe, wobei nur ein Farbton verwendet wird. (Eine Beispielanwendung finden Sie in Abb. 9–2.)

Sequenzielle Schemas mit zwei oder mehr Farbtönen
: Diese zwölf Schemas verlaufen jeweils von einem hellen und blassen Weiß oder Grau über ein oder zwei einander »ähnliche« Zwischenfarben (wie Blau und Grün oder Gelb, Grün und Blau) zu einer dunklen und gesättigten Farbe.

Die sequenziellen Schemas gibt es jeweils in zwei Varianten:

- Eine kontinuierliche Variante (als Interpolierer) zur Verwendung mit `d3.scale.Sequential()`
- Eine diskrete Variante (als Array aus diskreten Farbwerten) zur Verwendung mit `d3.scaleOrdinal()` oder den Binning-Skalierungen (wie `d3.scaleThreshold()`)[2]

Von den meisten der diskreten Schemas gibt es mehrere Versionen mit jeweils einer anderen Anzahl (zwischen drei und zehn) gleichmäßig verteilter diskreter Elemente. Die genauen Zahlen für die einzelnen Elemente können Sie der D3-Dokumentation auf *https://github.com/d3/d3/blob/master/API.md* entnehmen.

2 Ein diskretes Farbschema zu verwenden, anstatt einfach den entsprechenden Interpolierer für diskrete Werte auszuwerten, ermöglicht es, auch *nicht numerische* Kategoriewerte auf Farben abzubilden.

Falschfarbenschemas

Die folgenden Farbschemas sind für Falschfarbendiagramme und Heatmaps gedacht. Sie sind nur als kontinuierliche Interpolierer verfügbar (also nicht in einer diskreten Version).

Sequenzielle Schemas mit mehreren Farbtönen
: Es gibt fünf Schemas mit mehreren Farbtönen, die alle von einer sehr dunklen (fast schwarzen) Farbe zu Gelb oder Weiß verlaufen. Bei diesen Schemas werden Sättigung und Helligkeit gleichzeitig geändert (was für die Praxis möglicherweise nicht so sinnvoll ist, wie es sich in der Theorie anhört).

Zyklische oder Regenbogenschemas
: Die beiden Reigenbogenschemas sind in der Wahrnehmung einheitlicher als der Standardfarbkreis im HSL-Raum. Bei einem von ihnen stehen die beiden Hälften auch getrennt zur Verfügung, wobei die eine über die warme Seite des Farbkreises verläuft (Rot) und die andere über die kalte (Blau, Grün).

Besonders bemerkenswert ist das »Sinusbogen-Schema« `d3.interpolateSinebow`. Im herkömmlichen Regenbogenschema variieren die RGB-Komponenten auf reguläre, stückweise lineare Weise im Farbtonbereich von 0 bis 360. Beim Sinusbogen wird dieses stückweise lineare Verhalten durch drei Sinusfunktionen ersetzt, deren Phasen jeweils um 120° gegeneinander verschoben sind. Das ergibt einen helleren, aber in der Wahrnehmung sehr viel einheitlicheren Regenbogen (siehe Abb. 8–1). Dieses Schema wird wegen seines Erscheinungsbilds und seiner konzeptionellen Einfachheit sehr empfohlen.[3]

Palettengestaltung

Die Gestaltung von Farbschemas oder Farbpaletten für die Datenvisualisierung ist ein schwieriges Thema. Beachten Sie dabei die folgenden Überlegungen und Empfehlungen:[4]

- Überlegen Sie, ob die Farbe nur zur Verstärkung der Wahrnehmung von Informationen dient, die auch auf andere Weise aus dem Graphen abgelesen werden können, oder ob sie der Hauptinformationsträger ist. Beispielsweise hebt die Tönung von topografischen Karten lediglich die Höheninformationen hervor, die auch schon durch die Höhenlinien zur Verfügung stehen, wohingegen alle wichtigen Informationen in einem echten Falschfarbendiagramm wie dem aus Abbildung 8–2 ausschließlich durch Farben angezeigt werden.

➞

3 Siehe auch *http://basecase.org/env/on-rainbows*.

- Überlegen Sie, ob die Daten eine inhärente Sortierung aufweisen. Wenn ja, müssen Sie dafür sorgen, dass diese Sortierung auch von dem Farbschema unterstützt wird. Beispielsweise verbinden wir vereinbarungsgemäß und intuitiv Blau mit niedrigeren und Rot mit höheren Werten, was in einem sorgfältig ausgearbeiteten Graphen berücksichtigt werden sollte. (In dieser Hinsicht sind die mitgelieferten Farbschemas inkonsistent!) Manchmal kann die wahrgenommene Sortierreihenfolge von Farben unerwünscht sein (etwa bei bestimmten Arten von Kategoriedaten).
- Beachten Sie die Bedeutungen, die wir gewöhnlich mit bestimmten Farben verknüpfen (z. B. bei den Ampelfarben Rot, Gelb und Grün).
- Vermeiden Sie sehr dunkle und sehr helle Farben, da sie Einzelheiten unterdrücken können (»Weiß auf Weiß«, »Schwarz auf Schwarz«). Vermeiden Sie auf der anderen Seite aber auch schreiende Farben. Finden Sie einen Ausgleich zwischen grell und nicht wahrnehmbar. (Farbräume, die in der Wahrnehmung einheitlich sind, wie HCL, können Ihnen dabei behilflich sein.)
- Verwenden Sie starke Farbverläufe, wo es sinnvoll ist. Oft sind relativ kleine Änderungen in einigen Teilen des Definitionsbereichs viel interessanter als anderswo. Wenn Sie starke optische Wechsel im Farbschema dort platzieren, wo die Änderungen in den Daten von größter Bedeutung sind, können Sie die Wahrnehmung der wichtigen Einzelheiten verstärken.
- Im gleichen Sinne können Sie auch nicht kontinuierliche Änderungen verwenden, um eine Schwelle in den Daten anzuzeigen. Beispielsweise werden in topografischen Karten kontinuierliche Farbverläufe eingesetzt, Küstenlinien aber durch einen scharfen Übergang gekennzeichnet.
- Immer wenn Farbe verwendet wird, um Informationen zu vermitteln (also nicht nur aus rein ästhetischen Gründen), sollte das Diagramm über eine Farblegende verfügen, die die Zuordnung der Farben zu Werten aufschlüsselt. Wie offensichtlich Ihr Farbschema auch erscheinen mag, so ist es doch unmöglich, die genaue Zuordnung allein aus dem Graphen zu entnehmen.
- Denken Sie daran, dass etwa 8 % der Männer und 0,5 % der Frauen zumindest teilweise von Farbenfehlsichtigkeit betroffen sind, wobei die Rot-Grün-Sehschwäche am weitesten verbreitet ist.

Andere Farbschemas

Die schiere Anzahl und Vielseitigkeit der im Lieferumfang enthaltenen Farbschemas scheint alle Bedürfnisse abdecken zu können, aber je nach Zusammenhang und Zweck kann es sein, dass andere Schemas besser geeignet sind (siehe »*Palet-*

4 In Anhang D meines Buches *Gnuplot in Action* (Manning Publications, 2. Auflage) erfahren Sie noch mehr zu diesem Thema.

tengestaltung« auf *S. 166*). Die vorhandenen Farbschemas können Sie auch als Anregungen nutzen.[5] Es gibt keinen Grund dafür, die mitgelieferten Schemas zu verwenden, wenn sie für Ihre Zwecke unpassend erscheinen oder sich nur schwer an eine bestimmte Anwendung anpassen lassen.

Farbskalierungen

Um Farben zur Darstellung von Informationen zu nutzen, ist es oft praktisch, sie mit einem Skalierungsobjekt zu kombinieren (siehe Kapitel 7), insbesondere, um den Wertebereich des Skalierungsobjekts mit Farben zu definieren. In diesem Abschnitt lernen Sie dazu einige einfache Anwendungen und Techniken kennen.

Diskrete Farben

Diskrete Farbskalierungen können mit Binning- und mit diskreten Skalierungen realisiert werden. Die Binning-Skalierungen, die von `d3.scaleQuantize()` und `d3.scaleThreshold()` erstellt werden, eignen sich für die Wiedergabe eines kontinuierlichen Zahlenbereichs durch eine diskrete Menge von Farben. Die erste dieser beiden Skalierungen teilt den Definitionsbereich in gleich große Bins auf, während die zweite erwartet, dass der Benutzer Schwellenwerte zur Definition der Bins angibt (siehe Beispiele 4–7 und 7–3). Denken Sie daran, dass die Funktion `domain()` für Binning-Skalierungen unterschiedliche Bedeutungen hat (Einzelheiten siehe Kapitel 7):

```
var sc1 = d3.scaleQuantize().domain( [0, 1] )
    .range( [ "black", "red", "yellow", "white"] );
var sc2 = d3.scaleThreshold().domain( [ -1, 1] )
    .range( [ "blue", "white", "red" ] );
```

Diskrete Skalierungen, wie sie von `d3.scaleOrdinal()` erstellt werden, eignen sich, wenn diskrete Eingabewerte durch verschiedene Farben dargestellt werden sollen. Eine diskrete Skalierung kann eine explizite Beziehung zwischen einzelnen Werten des Definitionsbereichs und den Farben zu ihrer Darstellung herstellen:

```
var sc = d3.scaleOrdinal( [ "green", "yellow", "red"] )
    .domain( [ "good", "medium", "bad" ] );
```

5 Eine sehr große und vielseitige Sammlung von Paletten, die allerdings nicht alle für die Visualisierung im wissenschaftlichen Bereich gedacht sind, finden Sie auf *http://soliton.vm.bytemark.co.uk/pub/cpt-city/*. Besonders interessant für unsere Zwecke sind die Arbeiten von Kenneth Moreland und des Projekts Generic Mapping Tools, insbesondere die Palette GMT Haxby.

Wurde kein Definitionsbereich angegeben, weist eine diskrete Skalierung jedem neuen Symbol bei dessen Auftreten die nächste noch nicht verwendete Farbe zu (siehe Beispiele 5–6 und 9–5).

Ganz unabhängig von ihrer Implementierung durch Skalierungsobjekte müssen die Farben für diskrete Farbschemas ihrem Zweck entsprechend ausgewählt werden. Manchmal besteht dieser Zweck einfach darin, verschiedene grafische Elemente besser unterscheiden zu können, ohne dass dadurch irgendeine Bedeutung oder Reihenfolge angedeutet werden soll. Die mitgelieferten Kategorieschemas sind von dieser Art (siehe Beispiele 5–7 und 9–3). In anderen Fällen sollen die Farben tatsächlich eine Bedeutung vermitteln, wenn auch nicht unbedingt eine monotone Sortierung oder strenge Anordnung. Ein Beispiel dafür ist das Ampelschema in Rot, Gelb und Grün aus Abbildung 7–5. Schließlich werden diskrete Farbschemas manchmal auch verwendet, um eine monoton geordnete Folge von Schritten darzustellen (siehe den rechten Teil von Abb. 9–2). In letzterem Fall können Binning-Skalierungen zusammen mit den mitgelieferten sequenziellen oder divergierenden Schemas eingesetzt werden, um einen kontinuierlichen Bereich von Werten auf eine feste Menge einzelner Farben mit einer deutlichen Sortierung abzubilden.

Farbverläufe

Die Möglichkeiten von D3 zur Interpolation von Farben machen es besonders einfach, Farbverläufe zu erstellen. Die Beispiele in diesem Abschnitt zeigen Ihnen die Syntax für einige einfache, aber typische Fälle (siehe Beispiel 8–1 und Abb. 8–1).

Beispiel 8–1: Farbverläufe erstellen (siehe Abb. 8–1)

```
sc1 = d3.scaleLinear().domain( [0, 3, 10] )   ❶
    .range( ["blue", "white", "red"] );

sc2 = d3.scaleLinear().domain( [0, 5, 5, 10] )   ❷
    .range( ["white", "blue", "red", "white"] );

sc3 = d3.scaleSequential( t => "" + d3.hsl( 360*t, 1, 0.5 ) )   ❸
    .domain( [0, 10] );

sc4 = d3.scaleSequential( t => d3.interpolateSinebow(2/3-3*t/4) )   ❹
    .domain( [0, 10] );

sc5 = d3.scaleDiverging( t => d3.interpolateRdYlGn(1-t) )   ❺
    .domain( [0, 2, 10] );

sc6 = d3.scaleSequential( d3.interpolateRgbBasis(   ❻
    ["#b2d899","#ffffbf","#bf9966","#ffffff"] ) ).domain( [0,10] );
```

❶ Ein einfacher Farbverlauf von Blau über Weiß nach Rot mithilfe der Standardinterpolation des Farbraums. Beachten Sie aber die asymmetrische Lage des weißen Bandes!

❷ Ein Verlauf mit einem scharfen Übergang in der Mitte.

❸ Der viel gescholtene »HSL-Standardregenbogen«. Das Beispiel ist hier vor allem angegeben, um die Syntax zu demonstrieren (und nicht etwa, weil dies ein besonders gut geeigneter Farbverlauf wäre).

❹ Ein weit besserer Regenbogen, erstellt mit dem mitgelieferten Sinusbogen-Interpolierer. Hier wird er rückwärts durchlaufen (also von Blau zu Rot, um die übliche Bedeutung von niedrigen zu hohen Werten zu vermitteln). Der Wertebereich ist beschränkt, um zu verhindern, dass der Verlauf umlaufend weitergeführt und Farben wiederverwendet werden.

❺ Ein Beispiel für einen divergierenden Farbverlauf. Hier wird der mitgelieferte Rot/Gelb/Grün-Interpolierer verwendet, aber die Richtung geändert, sodass höhere Werte rot dargestellt werden. Der Definitionsbereich eines divergierenden Verlaufs muss *drei* numerische Werte enthalten, wobei der mittlere Wert auf die Interpoliererposition t = `0,5` abgebildet wird. Hier wird die gelbe Farbe, die die Mitte des zurückgegebenen Farbbereichs darstellt, verschoben und mit dem Wert 2 des Definitionsbereichs verknüpft.

❻ Ein Beispiel für die Art von Farbschema, die gewöhnlich in topografischen Karten verwendet wird. Im Gegensatz zu den anderen Schemas wächst die Helligkeit hier nicht monoton, sondern unterliegt einer zusätzlichen, systematischen Schwankung (Grün und Braun sind relativ dunkel, Gelb und Weiß relativ hell), was als optische Orientierungshilfe dienen kann. Dieses Schema verwendet den Interpolierer `d3.interpolateRgbBasis`, der ein Spline durch alle bereitgestellten Farben konstruiert (die als gleichmäßig verteilt angenommen werden). Der Vorteil des Spline-Interpolierers liegt nicht so sehr im glatteren Farbverlauf, sondern darin, dass es nicht erforderlich ist, die Position der Zwischenfarben mit `domain()` anzugeben.

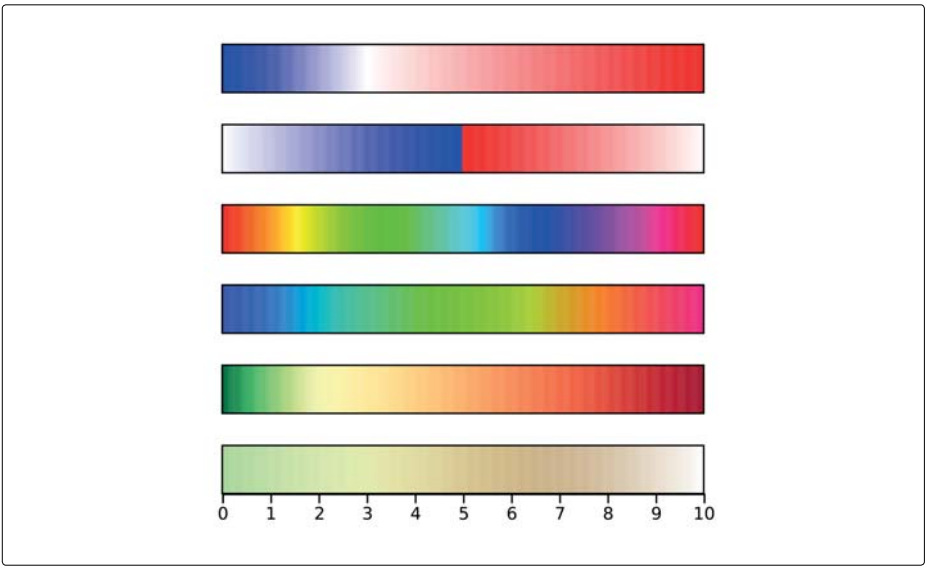

Abb. 8–1 *Farbverläufe (siehe Beispiel 8–1)*

Farblegenden

Ein Diagramm, das Farben als Informationsträger nutzt (also nicht nur, um das Erscheinungsbild und die Wahrnehmung zu verbessern), muss über eine *Legende* verfügen, die deutlich anzeigt, welche Werte welchen Farben entsprechen. Manchmal kann auch eine einfache Beschreibung in Textform ausreichen, aber eine Farblegende ist gewöhnlich viel deutlicher. D3 bringt keine Komponenten für diesen Zweck mit, aber es ist ziemlich einfach, sie selbst zu entwickeln. Die Legende aus Abbildung 8–1 wurde mit der Komponente aus Beispiel 8–2 gestaltet. Sie können sie als Vorlage auch für andere Anwendungen nutzen.

Beispiel 8–2: Befehle zum Erstellen einer Komponente für Farblegenden (siehe Abb. 8–1)

```
function colorbox( sel, size, colors, ticks ) {    ❶
    var [x0, x1] = d3.extent( colors.domain() );    ❷
    var bars = d3.range( x0, x1, (x1-x0)/size[0] );

    var sc = d3.scaleLinear()    ❸
        .domain( [x0, x1] ).range( [0, size[0] ] );
    sel.selectAll( "line" ).data( bars ).enter().append( "line" )    ❹
        .attr( "x1", sc ).attr( "x2", sc )
        .attr( "y1", 0 ).attr( "y2", size[1] )
        .attr( "stroke", colors );
```

```
    sel.append( "rect" )    ❺
        .attr( "width", size[0] ).attr( "height", size[1] )
        .attr( "fill", "none" ).attr( "stroke", "black" )

    if( ticks ) {    ❻
        sel.append( "g" ).call( d3.axisBottom( ticks ) )
            .attr( "transform", "translate( 0," + size[1] + ")" );
    }
}
```

❶ Neben der Ziel-Selection nimmt die Komponente noch die folgenden Argumente entgegen: die Größe der Farblegende (in Pixeln) als zweielementiges Array, die Farbskalierung und optional eine reguläre Skalierung für Teilstriche.

❷ Erstellt ein Array, das für jedes Pixel in der gewünschten Breite des Farbbalkens den zugehörigen Wert aus dem Definitionsbereich enthält.

❸ Diese Skalierung bildet die Werte des ursprünglichen Definitionsbereichs auf die Pixelkoordinaten der Farblegende ab.

❹ Erstellt eine ein Pixel breite, farbige Linie für jeden Eintrag in `bars`.

❺ Zeichnet einen Rahmen um die Farben ...

❻ ... und fügt Teilstriche hinzu, falls ein geeignetes Skalierungsobjekt übergeben wurde.

Falschfarbendiagramme und verwandte Techniken

Wenn sich Daten naturgemäß auf eine zweidimensionale Ebene abbilden lassen, bieten sich Falschfarbendiagramme oder Heatmaps als eine attraktive Darstellungsmöglichkeit an, möglicherweise kombiniert mit (oder als Alternative zu) Höhenlinien.

Heatmaps

Eine Form der Visualisierung, die sich sehr stark auf Farben als Informationsträger stützt, ist das *Falschfarbendiagramm* oder die *Heatmap*. Der Wert eines Datenpunkts in einem zweidimensionalen Raster wird dabei durch eine Farbe dargestellt. Beispielsweise werden in topografischen Karten Geländehöhen auf diese Weise angegeben.

Die Anzahl der einzelnen Graphenelemente (Datenpunkte oder Pixel) in einem Falschfarbendiagramm kann selbst für Raster bescheidener Größe rasch sehr groß werden. Aus Leistungsgründen kann es daher beim Erstellen solcher Diagramme in D3 ratsam (wenn nicht sogar notwendig) sein, das HTML5-Element `<canvas>` zu nutzen (siehe »*Das HTML5-Element <canvas>*« auf *S. 175*). Beispiel 8–3 führt

eine einfache Anwendung dieses Elements vor. Das Ergebnis sehen Sie in Abbildung 8–2.

Beispiel 8–3: Falschfarbendiagramm von Teilen der Mandelbrotmenge. Dieser Graph wurde mit dem `canvas`*-Element von HTML5 erstellt (siehe Abb. 8–2).*

```
function makeMandelbrot() {
    var cnv = d3.select( "#canvas" );    ❶
    var ctx = cnv.node().getContext( "2d" );

    var pxX = 465, pxY = 250, maxIter = 2000;    ❷
    var x0 = -1.31, x1 = -0.845, y0 = 0.2, y1 = 0.45;

    var scX = d3.scaleLinear().domain([0, pxX]).range([x0, x1]);    ❸
    var scY = d3.scaleLinear().domain([0, pxY]).range([y1, y0]);

    var scC = d3.scaleLinear().domain([0,10,23,35,55,1999,2000])    ❹
        .range( ["white","red","orange","yellow","lightyellow",
                 "white","darkgrey"] );

    function mandelbrot( x, y ) {    ❺
        var u=0.0, v=0.0, k=0;
        for( k=0; k<maxIter && (u*u + v*v)<4; k++ ) {
            var t = u*u - v*v + x;
            v = 2*u*v + y;
            u = t;
        }
        return k;
    }

    for( var j=0; j<pxY; j++ ) {    ❻
        for( var i=0; i<pxX; i++ ) {
            var d = mandelbrot( scX(i), scY(j) );
            ctx.fillStyle = scC( d );
            ctx.fillRect( i, j, 1, 1 );
        }
    }
}
```

❶ Wählt das `<canvas>`-Element auf der Seite aus und ruft damit einen Zeichenkontext für einfache, zweidimensionale Grafiken ab. Beachten Sie, dass `getContext()` nicht zu D3, sondern zur DOM-API gehört. Daher müssen Sie zunächst mit der Methode `node()` den zugrunde liegenden DOM-Knoten aus der D3-Selection gewinnen.

❷ Legt einige Konfigurationsparameter fest: die Größe der Leinwand in Pixeln, die maximale Anzahl der Schritte in der Mandelbrotiteration und den Ausschnitt der komplexen Ebene, an dem wir interessiert sind.

❸ Erstellt zwei Skalierungen, die Pixelkoordinaten auf Örter in der komplexen Ebene abbilden.

❹ Erstellt eine Skalierung, um die Anzahl der Iterationsschritte auf eine Farbe abzubilden. Das Aussehen des Graphen hängt sehr stark von der Platzierung der Zwischenwerte ab.

❺ Diese Funktion implementiert die eigentliche Mandelbrotiteration: Für einen gegebenen Punkt x + `i`y der komplexen Ebene führt sie die Iteration durch, bis entweder das Quadrat des Abstands vom Ursprung 2^2 überschreitet oder bis die Höchstzahl an Schritten überschritten ist. Der Rückgabewert ist die Anzahl der durchgeführten Schritte. (Mehr über Mandelbrotmengen erfahren Sie auf *https://de.wikipedia.org/wiki/Mandelbrot-Menge*.)

❻ Die doppelte Schleife durchläuft alle Pixel der Leinwand. Die Pixelkoordinaten werden in Örter in der komplexen Ebene transformiert, die dann an die Funktion `mandelbrot()` übergeben werden. Deren Rückgabewert wird in eine Farbe umgewandelt, mit der dann ein gefülltes Rechteck der Größe 1 × 1 (also ein Pixel) auf die Leinwand gezeichnet wird.

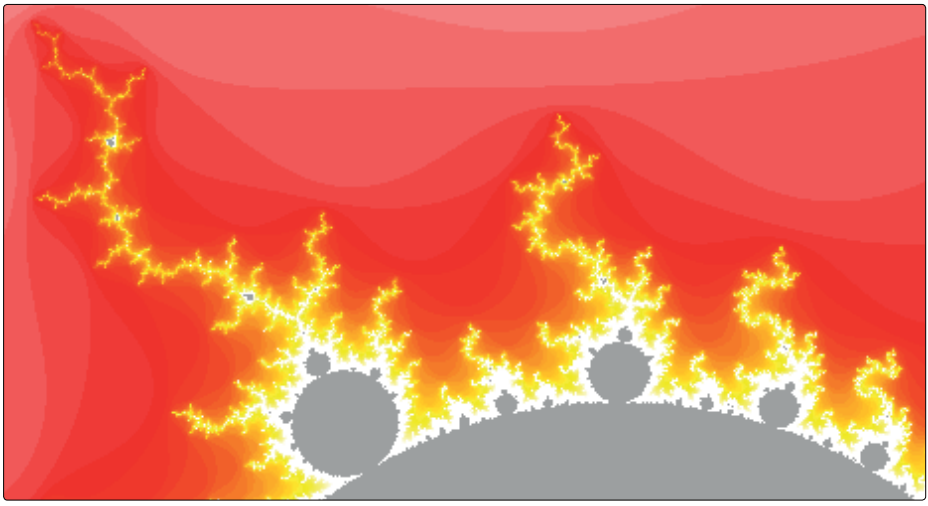

Abb. 8–2 *Verwendung des HTML5-Elements* `<canvas>` *(siehe Beispiel 8–3)*

Das HTML5-Element <canvas>

Das HTML5-Element `<canvas>` dient zum Erstellen von *Bitmapbildern*. Ebenso wie SVG kann es vom Browser aus mit Skripten gesteuert werden, allerdings bietet es im Gegensatz zu SVG nur elementare Bearbeitungsmöglichkeiten. Insbesondere stellt es keinen Zugriff auf einzelne Bestandteile des Diagramms bereit, um sie einzeln zu bearbeiten, wie es im DOM-Baum der Fall ist. Ein Bild auf dieser Leinwand ist eine »flache« Bitmap.

Sie müssen sich darüber im Klaren sein, dass es sich bei `<canvas>` in erster Linie um einen Platzhalter handelt. Um irgendetwas zeichnen zu können, müssen Sie zunächst mit dem API-Aufruf `getContext()` einen *Zeichenkontext* beschaffen. Dabei gibt es verschiedene Arten von Zeichenkontext mit unterschiedlichen APIs für unterschiedliche Arten der Programmierung (z. B. beschleunigte 3D-Grafik).

Wir beschäftigen uns hier ausschließlich mit dem einfachen Renderingkontext `CanvasRenderingContext2D`, der nur wenige grafische Grundelemente bietet. Darunter befinden sich Möglichkeiten, um gefüllte und nicht gefüllte Rechtecke und Text zu zeichnen, sowie eine Funktionalität zur Pfaderstellung, die ähnlich wie das SVG-Element `<path>` eine Art von Turtle-Grafik verwendet. Bevor Sie irgendetwas zeichnen können, müssen Sie die Strich- oder Füllfarbe als Kontexteigenschaft festlegen.

Der grundlegende Arbeitsablauf zum Zeichnen auf einer HTML5-Leinwand in D3 sieht wie folgt aus:

```
var cnv = d3.select( "#canvas" );
var ctx = cnv.node().getContext( '2d' );

ctx.fillStyle = color;
ctx.fillRect( x, y, w, h );
```

Der Wert von `color` muss eine CSS3-Farbspezifikation sein. Bei `x` und `y` handelt es sich um die Koordinaten der oberen linken Ecke des Rechtecks und bei `w` und `h` um seine Breite und Höhe.

Das Element `<canvas>` lässt sich im Großen und Ganzen recht einfach verwenden. Ein Tutorial finden Sie auf *https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial*.

Höhenlinien

Eine Alternative (oder Ergänzung) zu Falschfarbendiagrammen – insbesondere für Daten mit fließenden Übergängen – bilden *Höhenlinien*, also Kurven, die Punkte gleicher Höhe verbinden (wie in topografischen Karten). Zur Berechnung solcher Linien stellt D3 ein *Layout* bereit (siehe Tabelle 8–2).

Funktion	Beschreibung
`d3.contours()`	Gibt ein neues Höhenlinienlayout mit Standardeinstellungen zurück.
`conMkr( [data] )`	Berechnet die Höhenlinien für die übergebene Datenmenge. Die Daten müssen als eindimensionales Array so formatiert sein, dass das Arrayelement mit dem Index `[i + j*cols]` dem Element an der Position `[i, j]` im Raster entspricht. Gibt ein Array aus GeoJSON-Objekten zurück, die nach den Schwellenwerten sortiert sind, für die sie stehen.
`conMkr.size( [cols, rows] )`	Legt die Anzahl der Zeilen und Spalten als zweidimensionales Array fest.
`conMrk.thresholds( args )`	Das Argument ist gewöhnlich ein Array der Werte, für die Höhenlinien berechnet werden sollen. Ist es dagegen ein einzelner Integer *n*, dann werden etwa *n* Höhenlinien in geeignetem Abstand gezeichnet.
`conMkr.contour( [data], threshold )`	Erstellt am angegebenen Schwellenwert eine einzelne Höhenlinie für die übergebene Datenmenge und gibt ein einzelnes GeoJSON-Objekt zurück.

Tab. 8–2 *Funktionen zur Berechnung von Höhenlinien (wobei* `conMkr` *ein Höhenlinienlayout ist)*

Die erforderliche Datendarstellung ist etwas merkwürdig. Es wird vorausgesetzt, dass sich die Datenpunkte in einem gleichmäßigen, rechteckigen Raster aus Zeilen (`rows`) und Spalten (`cols`) befinden, das als *eindimensionales Array* aus Zahlen gespeichert ist, wobei die Zeilen eine kontinuierliche Folge bilden. Das Element am Arrayindex `i + j*cols` steht daher für den Punkt an der Position `[i, j]`. Es gibt keine Möglichkeit, um die einzelnen Datenpunkte mit den eigentlichen (Definitionsbereichs-)Koordinaten zu verknüpfen. Das hat unter anderem zur Folge, dass das resultierende Diagramm `cols` x `rows` Pixel groß ist. Wenn Sie eine andere Größe haben wollen, müssen Sie eine Skalierungstransformation anwenden.

Wenn das Höhenlinienlayout für die Daten aufgerufen wird, gibt es ein Array aus je einem GeoJSON-Objekt[6] für jede einzelne Höhenlinie zurück. Anschließend können Sie mit dem Generator `d3.geoPath()` einen geeigneten Befehlsstring für das Attribut `d` eines regulären `<path>`-Elements erstellen. (Dieser Vorgang ähnelt demjenigen aus Kapitel 5.) Jedes Höhenlinienobjekt stellt die Eigenschaft `value` mit dem Schwellenwert bereit, für den die Linie steht.

6 GeoJSON ist ein Standard zur Darstellung von geografischen Daten. Definiert ist er in RFC 7946.

Wenn Sie die Höhenlinien mit Farben füllen, ergibt sich wiederum ein Falschfarbendiagramm oder eine Heatmap. In Beispiel 8–4 werden beide Darstellungen gemeinsam verwendet: Erst wird eine große Menge von gefüllten Höhenlinien gezeichnet, um einen farbigen Hintergrund mit fließenden Verläufen zu bilden. Anschließend werden auf diesen Hintergrund einige Höhenlinien an ausgewählten Schwellenwerten gezeichnet (siehe Abb. 8–3).

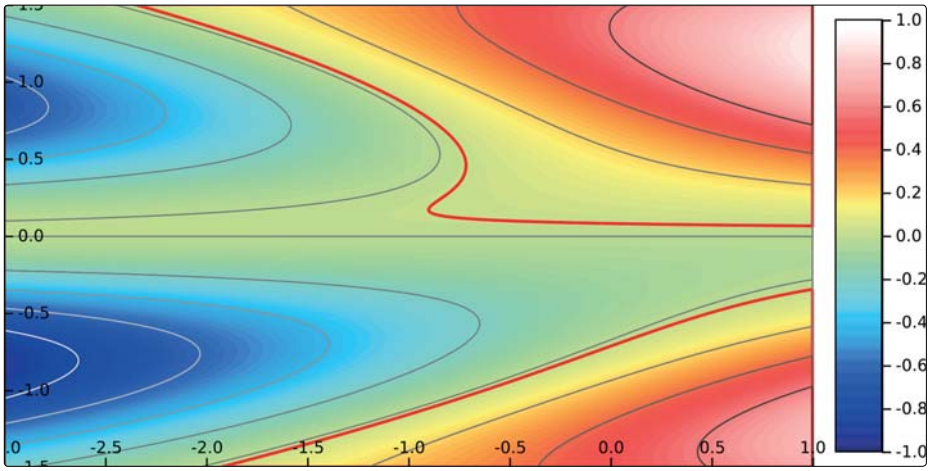

Abb. 8–3 *Ein Falschfarbendiagramm einer glatten Funktion. Dieser Graph wurde mit dem D3-Mechanismus der Höhenlinienlayouts erstellt (siehe Beispiel 8–4).*

Beispiel 8–4: Ein Falschfarbendiagramm mit Höhenlinienlayouts erstellen (siehe Abb. 8–3)

```
function makeContours() {
    d3.json( "haxby.json" ).then( drawContours );   ❶
}

function drawContours( scheme ) {
    // Richtet die Skalierungen einschließlich Farbskalierungen ein
    var pxX = 525, pxY = 300;   ❷
    var scX = d3.scaleLinear().domain([-3, 1]).range([0, pxX]);
    var scY = d3.scaleLinear().domain([-1.5, 1.5]).range([pxY, 0]);
    var scC = d3.scaleSequential(
        d3.interpolateRgbBasis(scheme["colors"]) ).domain([-1,1])   ❸
    var scZ = d3.scaleLinear().domain( [-1, -0.25, 0.25, 1] )
        .range( [ "white", "grey", "grey", "black" ] );

    // Generiert die Daten
    var data = [];   ❹
    var f = (x, y, b) => (y**4 + x*y**2 + b*y)*Math.exp(-(y**2))
```

```
    for( var j=0; j<pxY; j++ ) {
        for( var i=0; i<pxX; i++ ) {
            data.push( f( scX.invert(i), scY.invert(j), 0.3 ) );
        }
    }

    var svg = d3.select( "#contours" ), g = svg.append( "g" );   ❺
    var pathMkr = d3.geoPath();   ❻

    // Erstellt und zeichnet gefüllte Höhenlinien (Tönung)
    var conMkr = d3.contours().size([pxX, pxY]).thresholds(100);   ❼
    g.append("g").selectAll( "path" ).data( conMkr(data) ).enter()
        .append( "path" ).attr( "d", pathMkr )   ❽
        .attr( "fill", d=>scC(d.value) ).attr( "stroke", "none" )

    // Erstellt und zeichnet Höhenlinien
    conMkr = d3.contours().size( [pxX,pxY] ).thresholds( 10 );   ❾
    g.append("g").selectAll( "path" ).data( conMkr(data) ).enter()
        .append( "path" ).attr( "d", pathMkr )
        .attr( "fill", "none" ).attr( "stroke", d=>scZ(d.value) );

    // Erstellt eine einzelne Höhenlinie
    g.select( "g" ).append( "path" )   ❿
        .attr( "d", pathMkr( conMkr.contour( data, 0.025 ) ) )
        .attr( "fill", "none" ).attr( "stroke", "red" )
        .attr( "stroke-width", 2 );

    // Erstellt die Achsen
    svg.append( "g" ).call( d3.axisTop(scX).ticks(10) )   ⓫
        .attr( "transform", "translate(0," + pxY + ")" );
    svg.append( "g" ).call( d3.axisRight(scY).ticks(5) );

    // Erstellt die Farblegende
    svg.append( "g" ).call( colorbox, [280,30], scC )   ⓬
        .attr( "transform", "translate( 540,290 ) rotate(-90)" )
        .selectAll( "text" ).attr( "transform", "rotate(90)" );
    svg.append( "g" ).attr( "transform", "translate( 570,10 )" )
        .call( d3.axisRight( d3.scaleLinear()
                            .domain( [-1,1] ).range( [280,0] ) ) );
}
```

❶ Lädt die Farben des Farbschemas aus einer Datei und übergibt sie an die Funktion `drawContours()`, die die eigentliche Arbeit ausführt.[7]

❷ Legt die Größe des Diagramms in Pixeln fest. Beachten Sie, dass es einen Datenpunkt pro Pixel geben muss.

7 Dieses Farbschema basiert auf der Palette GMT Haxby des Projekts Generic Mapping Tools (siehe *http://soliton.vm.bytemark.co.uk/pub/cpt-city/gmt/index.html*).

❸ Die aus der Datei geladenen Farben werden zusammen mit dem Interpolierer `d3.interpolateRgbSpline` eingesetzt, um ein Farbskalierungsobjekt zu erstellen (siehe das letzte Element in Beispiel 8–1). Ein separates Skalierungsobjekt namens `scZ` bestimmt die Farbe der *Höhenlinien*, damit sie gegen den Hintergrund mit seinen wechselnden Farben gut sichtbar sind.

❹ Generiert die Daten durch Auswertung der Funktion `f(x, y)` für jede Pixelkoordinate. Mit der Funktion `invert()` des Skalierungsobjekts werden die Definitionsbereichskoordinaten für jede Pixelkoordinate abgerufen.

❺ Erstellt ein Handle für den DOM-Baum und hängt ein `<g>`-Element als Container für den Hauptteil des Diagramms an.

❻ Erstellt eine Instanz des Generators `d3.geoPath`. Hierbei handelt es sich um ein Funktionsobjekt. Wenn es für ein GeoJSON-Objekt aufgerufen wird, gibt es einen für das Attribut `d` eines `<path>`-Elements geeigneten String zurück.

❼ Erstellt ein Höhenlinienlayout und legt die Anzahl der Pixel im Graphen und die Daten fest. Die Anzahl der Höhenlinien ist sehr groß. Wenn jede davon mit Farbe gefüllt wird, ergeben sie insgesamt eine Heatmap mit einem fließenden Farbverlauf.

❽ Hängt für jede Höhenlinie ein `<path>`-Element an und füllt es entsprechend der Eigenschaft `value` des Höhenlinienobjekts mit einer Farbe. Für jede Höhenlinie wird der Generator `pathMkr` aufgerufen und gibt einen geeigneten String für das Attribut `d` eines `<path>`-Elements zurück.

❾ Konfiguriert das Höhenlinienlayout für etwa zehn Höhenlinien und ruft es dann auf, um ungefüllte Höhenlinien zu erstellen.

❿ Nur um Ihnen zu zeigen, wie es gemacht wird: Mit der Funktion `contour()` können Sie eine *einzelne* Höhenlinie für den angegebenen Schwellenwert erstellen.

⓫ Fügt mithilfe des ursprünglichen Skalierungsobjekts Koordinatenachsen zu dem Graphen hinzu.

⓬ Fügt eine Farblegende hinzu, die zeigt, welche numerischen Werte den Farben in der Heatmap entsprechen. Dieser Code nutzt die Komponente `colorbox` aus Beispiel 8–2 und wendet dann eine SVG-Transformation an, um sie vertikal auszurichten.