Accessible D3 and SVG

Group 2

Alexander Grass, Lea Novak, and Danica Radulovic

Institute of Interactive Systems and Data Science (ISDS),
Graz University of Technology
A-8010 Graz, Austria

30 Jun 2019

Contents

List of Figures

List of Listings

Chapter 1

ARIA Attributes

ARIA, Accessible Rich Internet Applications, defines rules for creating a proper accessibility to web content for assistive technologies, such as screen readers. All ARIA enhanced controls must be keyboard focusable. ARIA tags were primarily introduced for HTML [W3C 2017] and were afterwards extended to SVG [W3C 2018].

1.1 ARIA Attributes for HTML

There are many different ARIA attributes, however in this report one can find only a few of them. These are the ones which are most frequently used and that have been implemented into this project.

- `aria-label` - this is the simplest way to label an element by giving it a short name.

  ```
  <button aria-label="menu" class="hambuger">&#9776;</button>
  ```

- `aria-labelledby` - gives an alternative way for labeling an element by using the ID of another element. If an example below is considered, one would have the following: there are three controls with corresponding IDs: myDeliveryId, myNameId, myAddressId, and two controls that are labeled using those IDs. If one has a look at the first control inside of `<input>` tag, one can see that the aria-labelledby attribute takes myDeliveryId and myNameId for describing the input element. Therefore, when focusing that element the screen reader would read the content of the other two elements, which IDs were used for the description. This would result in reading: “Delivery Name”. The similar case is with the second input control - the screen reader would read: “Delivery Address”.

  ```
  <div id="myDeliveryId">Delivery</div>
  <div>
      <div id="myNameId">Name</div>
      <input type="text" aria-labelledby="myDeliveryId myNameId"/>
  </div>
  <div>
      <div id="myAddressId">Address</div>
      <input type="text" aria-labelledby="myDeliveryId myAddressId"/>
  </div>
  ```

- `aria-describedby` - this attribute is very similar to the `aria-labelledby`. Moreover, it gives a developer a possibility to provide a longer description of an element, instead of giving it only a short label. When a button for closing a dialog window is focused, screen reader would read “Closing dialog returns to the previous page”.

```
<button aria-describedby="dialogClose" onclick="myDialog.close()">X</
    button>
<div id="dialogClose">Closing dialog returns to the previous page</div>
```

- `aria-valuemin / aria-valuemax` - used for any controls that represent some range, such as slider, axis, etc. In the example below shifting a focus to the slider, screen reader would give the minimal (1), maximal (100) and the current (50) value of the slider.

```
<div class="slidecontainer">
  <input type="range" aria-valuemin="1" aria-valuemax="100" aria-
      valuenow="50">
</div>
```

1.2 ARIA Attributes for SVG

The same ARIA Attributes are used with both SVG and HTML, with a small extension for SVG, which relies on the `role` attribute. Role is the main indicator of the type of an object, and is usually used to provide the semantics within SVG or HTML. There are different values the `role` attribute can take, depending on what kind of role it refers to. Two main classes of roles are:

- Widget roles (takes values such as - `button, checkbox, slider, switch`, etc.),
- Document structure roles (take values: `document, img, list, presentation`, etc.).

The `role` attribute has been extended for SVG by three additional values this attribute can take:

- `graphics-document` - should be used for the root element. In the case of a SVG file, it should be assigned to role in the line of code where the SVG is defined, since it refers to the whole SVG chart, graph, etc.

```
<svg xmlns="https://www.w3.org/2000/svg" viewBox="0 0 200 100"
        role="graphics-document" >
```

- `graphics-object` is usually a section of a `graphics-document` that represents a distinct object or sub-component with semantic meaning. Graphics-object can have more graphic-objects as sub-components. It is mostly assigned as the value of a role attribute inside of SVG `g` grouping element. If one considers as an example a SVG file representing an image of a house with windows and doors, then each of those should be considered as a separate graphic-object.

```
<g role="graphics-object" aria-label="house">
        <g role="graphics-object" aria-label="door">
        ...
        </g>

        <g role="graphics-object" aria-label="window">
         ...
        </g>
...
</g>
```

- `graphics-symbol` represents a graphical object used to reveal a simple meaning or category. It can be a component of a larger structured graphic such as a chart or map. However, it is frequently used for representing a repeated symbol (for example: an icon).

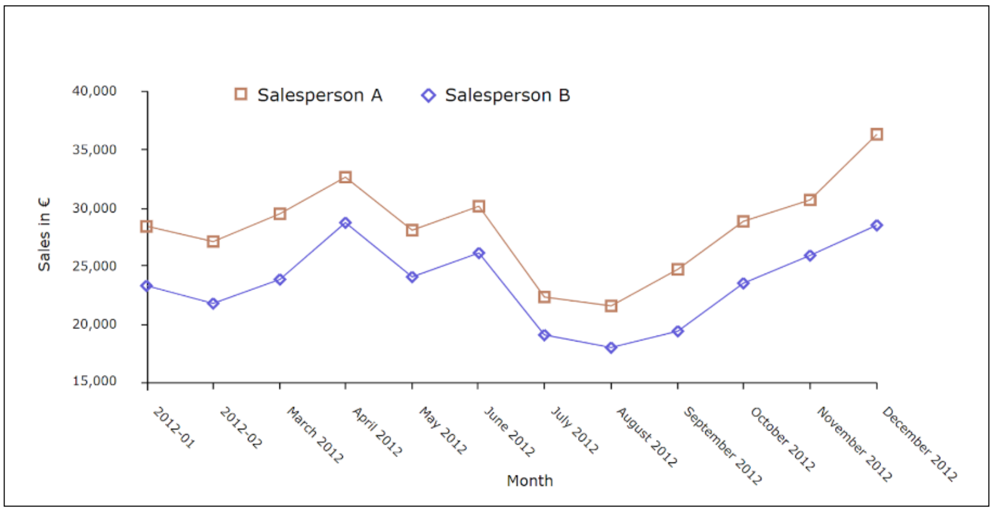


Figure 1.1: Hand-edited SVG line chart. [The image was provided to the authors of the paper by professor Keith Andrews.]

1.3 Hand-Edited Accessible SVG File

Considering the hand-edited SVG example (see Figure 1.1) here are just some of the lines of code that provide accessibility. Firstly, the whole chart is represented as one root object, therefore `role="graphics -document"` .

```
<svg xmlns="https://www.w3.org/2000/svg" viewBox="0 0 200 100"
        role="graphics-document" >
```

If y-axis and the code below is observed, one can see that `aria-valuemin` gets the smallest value of the y-axis, and the `aria-valuemax` the maximal one. The attribute `tabindex = "0"` allows keyboard focusing of the y-axis.

```
<g role="yaxis" tabindex="0" aria-valuemin="15000" aria-valuemax
    ="40000" >
```

The datapoint in the following code snippet includes the attribute `aria-labelledby = "x-2012-01"` , which means that `x-2012-01` is the id, defined in another part of the code. Therefore when this datapoint is focused, screen reader would read "2012 -01", which is the content of the element that has `id = "x-2012-01"`.

```
<text role="axislabel" id="x-2012-01" class="chart-label">2012-01</
    text>
<line class="chart-line" x1="150" y1="443" x2="150" y2="448"/>
```

```
<g role="datapoint" tabindex="0">
  title role="datavalue" aria-labelledby="x-2012-01">28366</title>
  <rect x="93" y="315" class="chart-data-rect" />
</g>
```

Chapter 2

Generating SVG Files Programmatically with D3

Currently, there is no obvious way to generate a SVG file programmatically with D3. Even extensive research leads to the conclusion that a tool which could accomplish this needs to be written by oneself.

2.1 Introduction

To generate SVG files programmatically, there are some workarounds out there. One possibility is for example to write HTML and D3 code, open the resulting file in a browser, right click the desired picture and click "save to file". However, this comes with a huge overhead and often the user wants to generate multiple SVG files programmatically. This would not be possible in means of the described way above. To solve this issue, a simple command line tool is provided. The user simply writes D3 code in a TypeScript file and executes a command in the command line after to generate the SVG.

2.2 Toolset

For the process of generating SVG programmatically, the following tool chain is used:

- NPM as package manager.
- TypeScript with D3v5 as source.
- CSV as dataset.
- Gulp as task runner.

2.3 Folder Structure

```
svggenerator
├─ build
│  ├─ js
│  │  ├─ bar.js
│  │  ├─ line.js
│  │  └─ pie.js
│  └─ svg
│     ├─ bar.svg
│     ├─ line.svg
│     └─ pie.svg
├─ help
│  └─ template.ts
├─ packagelock.json
├─ package.json
├─ source
│  ├─ assets
│  │  └─ asset.txt
│  ├─ data
│  │  └─ csv
│  │     ├─ fruitsbar.csv
│  │     ├─ fruitspie.csv
│  │     ├─ linedata.csv
│  │     └─ programmlanguages.csv
│  └─ ts
│     ├─ bar.ts
│     ├─ line.ts
│     └─ pie.ts
└─ gulpfile.js
```

2.4 Background

An explanation of the architecture and design decisions is now given. Since D3 assumes it is running in the browser it needs a DOM to work. However, for the scenario of generating SVG programmatically with D3 this is not possible. So the best approach is to simulate a DOM in the back end. This is accomplished by a JavaScript library called "jsdom". This makes it possible for the user to write D3 code, manipulating a standalone virtual DOM. After the D3 code is written, the SVG DOM node is written to a local file. Gulp handles the transpiling of TS to JS.

2.5 Requirements

Since the tool builds upon node.js, there are many dependencies used throughout. Fortunately, the only dependency the user needs to install manually is the node package manager (NPM) [Bogensberger 2019]. After that the user just needs to run the command "npm install" in the root folder of the tool and all dependencies will be installed automatically.

2.6 Process

On a high level, the process of generating the SVG is simple. There is a "template.ts" file in the "help" folder. The desired D3 code needs to be added where the comments hint at. Listing 2.1 shows the template file. After that, the user simply opens the command line and runs a gulp command.

2.6.1 File Flow

In the initial setting, right after the desired D3 code is written, there should be at least a TypeScript file in the "ts" folder and a corresponding dataset in the "csv" folder. After executing the gulp command, all TS files are transpiled to JS files and are stored in the "js" folder. The next step gulp takes is executing the JS files. After that the user finds the SVG files in the "svg" folder.

2.6.2 Gulp commands

There are two gulp commands implemented:
The first is "`gulp compile --file <ts-file> --dataset <csv-file>`". The command requires the typescript file name and the corresponding csv file name as parameters. It doesn't matter if the ".ts" and ".csv" extensions are added to the name. It works either way. The files should be located in the corresponding folders. After execution the user will find the resulting SVGs in the "svg" folder.
The second command is "`gulp clean`", which gets rid of all files which were generated during the run of the compile command, namely everything in the "js" and "svg" folder.

```
{
  const d3 = require("d3")
  const { JSDOM } = require("jsdom");
  const fs = require('fs');
  const csv = require('csvtojson')

  const dom = new JSDOM(`<!DOCTYPE html>
  <meta charset="utf-8">

  <html><body></body></html>`);

  const doc = dom.window.document;

  const fileName = process.argv[2];
  const csvName = process.argv[3];
  const csvPath = 'source/data/csv/' + csvName;


  csv()
  .fromFile(csvPath)
  .then((data) => {
      //---------------------------------------------------- insert d3
         code ...

      var svg = d3.select(doc).select("body")
          .append("svg")
                .attr("xmlns", "http://www.w3.org/2000/svg")
                .attr("xmlns:xlink", "http://www.w3.org/1999/xlink");


      //---------------------------------------------------- ... until
         here

      // PRINT OUT:
      fs.writeFileSync('build/svg/' + fileName + '.svg', d3.select(doc).
         select("body").html());
  });
}
```

Listing 2.1: The Template File for the Creation of a D3 Chart.

Chapter 3

Chart Structure

This project features three types of charts: line, bar, and pie chart. Each of them is a separate TypeScript file which in the end is transpiled into a SVG file. This section provides an overview of the chart structure which is written in D3 and when the ARIA attributes need to be added.

3.1 General

To create an accessible SVG chart the HTML elements need to be structured and grouped together and the corresponding ARIA and role attributes added. D3 allows the user to create a chart by dynamically inserting SVG nodes into the Document Object Model. The DOM is the internal data structure of the web browser which stores/models a HTML page and is constructed as a tree consisting of HTML elements as shown in Figure 3.1.

As for the data set an external CSV file is passed via the command line and handled through D3 to extract the data. The first row is extracted to aquire the header of the CSV with the IDs in it. Since this project works only with 2D charts there are only two IDs: a key and a value. Figure 3.2 shows the dataset and Listing 3.1 displays the process.

Listing 3.2 shows the added "svg" element using D3. Since the "svg" element is the root element a "role" attribute has to be added with the "graphics-document" value. This structure applies to all three charts. In the next step the title, description and chart area are created and grouped together. The "aria-charttype" and "role" are defined and the title and description id are assigned to the "aria-labelledby"

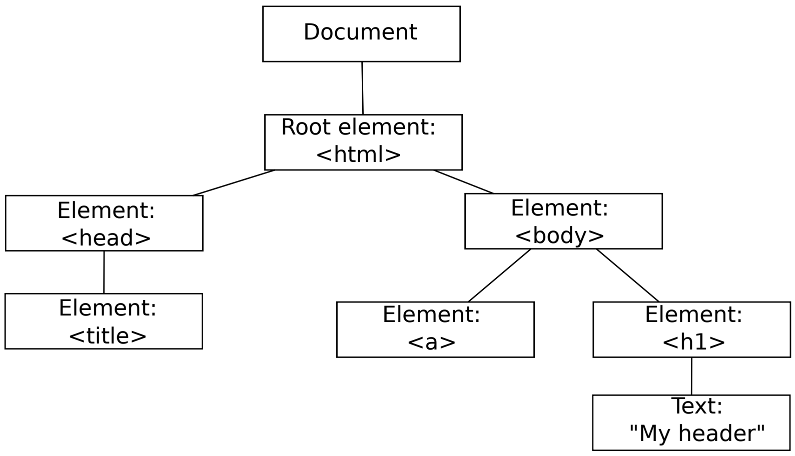


Figure 3.1: DOM Example. [Redrawn by the authors of this survey. Original by: [w3schools 2019]]

```
csv()
.fromFile(csvPath)
.then((data)=> {

    var headers = d3.keys(data[0]);
    var key = headers[0];
    var value = headers[1];

}
```

Listing 3.1: Extract the Header of the CSV File.

```
var svg = d3.select(doc).select("body")
    .append("svg")
    .attr("viewBox", "0 0 " + width + " " + height)
    .attr("version", "1.1")
    .attr("xmlns", "http://www.w3.org/2000/svg")
    .attr("xmlns:xlink", "http://www.w3.org/1999/xlink")
    .attr("role", "graphics-document");
```

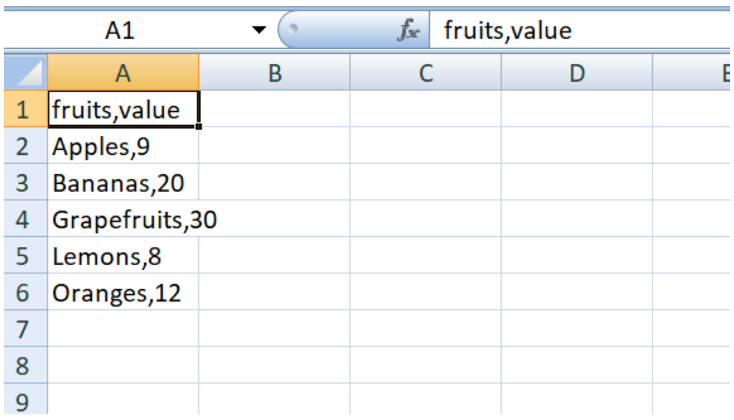
Listing 3.2: SVG element attributes in D3.

```
var rootNode = svg.append("g")
    .attr("id", "ChartRoot")
    .attr("role", "chart")
    .attr("aria-charttype", "line")
    .attr("tabindex", "0")
    .attr("aria-labelledby", "title desc");

rootNode.append("title")
    .attr("id", "title")
    .attr("role", "heading")
    .text("Monthly Fees of Student Dorms");

rootNode.append("desc")
    .attr("id", "desc")
    .text("Monthly Fees of Student Dorms in Graz since 2011");
```

Listing 3.3: SVG Chart Type.



	A	B	C	D	E
1	fruits,value				
2	Apples,9				
3	Bananas,20				
4	Grapefruits,30				
5	Lemons,8				
6	Oranges,12				
7					
8					
9					

A1 fruits,value

Figure 3.2: CSV Dataset. [The image was created by the authors of this paper.]

```
xAxisGroup.attr("id", "xScale")
    .attr("role", "xaxis")
    .attr("aria-axistype", "category")
    .attr("tabindex", "0")
    .attr("aria-valuemin", newYMin)
    .attr("aria-valuemax", newYMax)
    .attr("transform", "translate("+ 0 +"," + height  + ")")
    .call(d3.axisBottom(xScale).ticks(10))
        .selectAll("text")
        .data(data)
        .attr("role", "axislabel")
        .attr("id", function(d: any) { return "x-" + d[key]; })
        .style("text-anchor", "middle");
```

Listing 3.4: SVG X-Axis.

attribute. Listing 3.3 highlights the steps.

For the x- and y-axis almost the same attributes need to be added. Depending on the data set "aria-axistype" can be defined as "category" if the input is not a number and with the "tabindex" attribute set to "0" the values are keyboard focusable. With the help of D3 the "aria-valuemin" and "aria-valuemax" can easily be assigned as shown in Listing 3.4 and Listing 3.5.

3.2 Line Chart

The line chart has a "circle" element for each data entry as shown in Figure 3.3. Each circle has its own group element consisting of a title with the the corresponding value to it. The group element gets for the "role" attribute the value "datapoint" assigned. There are various elements which are added to the "title" element. First, for the "role" attribute "datavalue" is assigned. For the "aria-labelledby" attribute are two IDs given. The first one is the referred x-axis value and the second one is the current value or the y-axis value. The current value is assigned as separate ID in the next line, in this case as "point- + d[value];". The final step is to connect all circles with a line and add the related "role" attribute to the line. Listing 3.6 shows the essential steps for the line chart.

Figure 3.4 shows a section of the DOM of the line chart. The "text" element is from the x-axis and shows the first entry value which is 2011. The "circle" element is from the data area group and contains

```
yAxisGroup
    .attr("id", "yScale")
    .attr("role", "yaxis")
    .attr("tabindex", "0")
    .attr("aria-valuemin", d3.min(data, function(d:  any)  return d[value]; ))
    .attr("aria-valuemax", d3.max(data, function(d:  any)  return d[value]; ))
    .call(d3.axisLeft(yScale).tickFormat(function(d: any){
         return d;
    }).ticks(10))
    .selectAll("text")
        .attr("role", "axislabel")
        .attr("id", function(d: any) { return "y-" + d});
```

Listing 3.5: SVG Y-Axis.

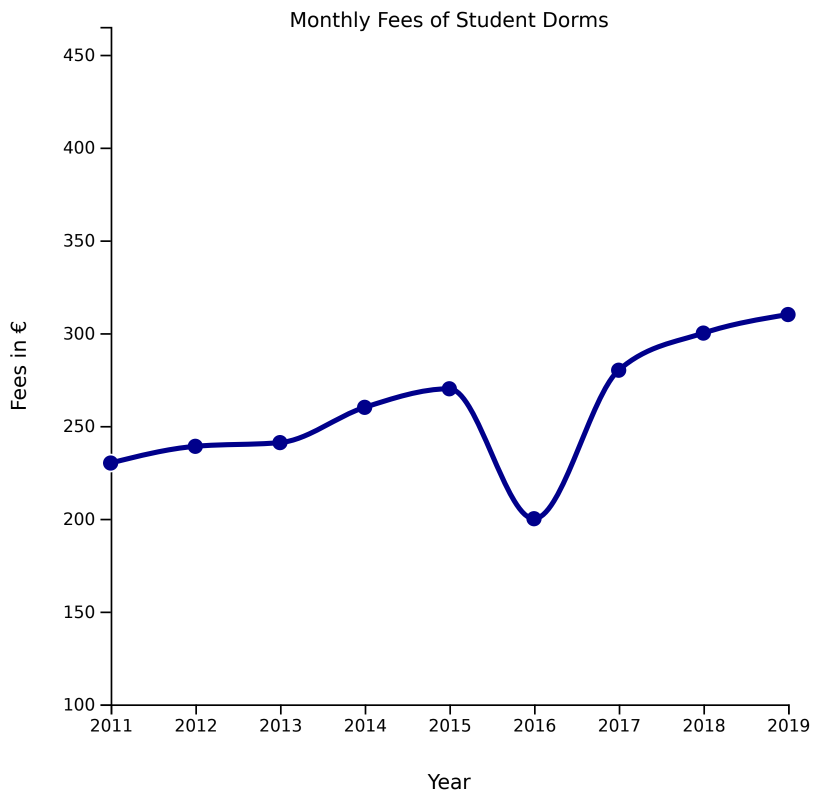
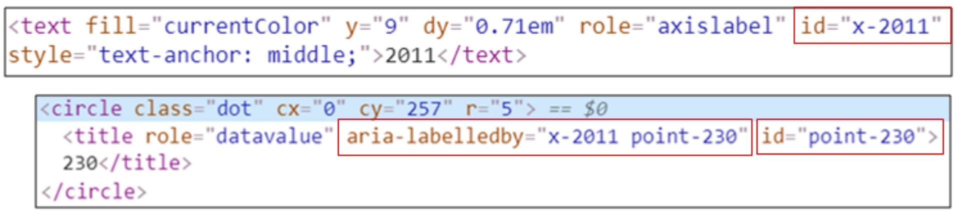


Figure 3.3: Line Chart. [The image was created by the authors of this paper.]

```
lineData.selectAll(".dot")
    .data(data)
    .enter().append("g")
        .attr("tabindex", "0")
        .attr("role", "datapoint")
    .append("circle") // Uses the enter().append() method
    .attr("class", "dot") // Assign a class for styling
    .attr("cx", function(d: any) { return xScale(d[key])})
    .attr("cy", function(d: any) { return yScale(d[value]) })
    .attr("r", 5)
    .append("title")
        .attr("role", "datavalue")
        .attr("aria-labelledby",
            function(d:  any)  return "x-" + d[key]+ " point-" + d[value]; )
        .text(function(d: any) { return d[value]; })
        .attr("id", function(d: any) { return "point-" + d[value]; });

lineData.append("path")
        .attr("class", "line") // Assign a class for styling
        .attr("d", line(data)) // Calls the line generator
        .attr("role", "line");
```

Listing 3.6: Appends a Circle Element for each Data Point.



```
<text fill="currentColor" y="9" dy="0.71em" role="axislabel" id="x-2011"
style="text-anchor: middle;">2011</text>
```

```
<circle class="dot" cx="0" cy="257" r="5"> == $0
  <title role="datavalue" aria-labelledby="x-2011 point-230" id="point-230">
  230</title>
</circle>
```

Figure 3.4: Line Chart Aria-Labelledby Attribute. [The image was created by the authors of this paper.]

the "title" element which has the "aria-labelledby" attribute with the two IDs. The screen reader will first read the year 2011 and then the value which is 230.

3.3 Bar Chart

The bar chart has a "rect" element for each data entry as shown in Figure 3.5. Each "rect" consists of a "title" element with the the corresponding value to it and is keyboard focusable because of "tabindex" set to "0". The width and the height are scaled according the dataset with D3. The "title" element gets for the "role" attribute "datavalue" assigned, an ID which is the current value and an "aria-labelledby" attribute with two IDs. It follows the same principle as the line chart as shown in Listing 3.7. The first one is the referred x-axis value and the second one is the current value or the y-axis value. Additionally, the value of each bar is displayed within in the bar to get a better overview of the whole chart.

Figure 3.6 shows a section of the DOM of the bar chart. The first group element is the root node of the x-axis and contains all entries of the x-axis. The first entry is Apples and the "text" element has the associated ID to it. The "rect" element is from the data area group and contains the "title" element which has the "aria-labelledby" attribute with two IDs. The screen reader will read Apples and then 67.8.

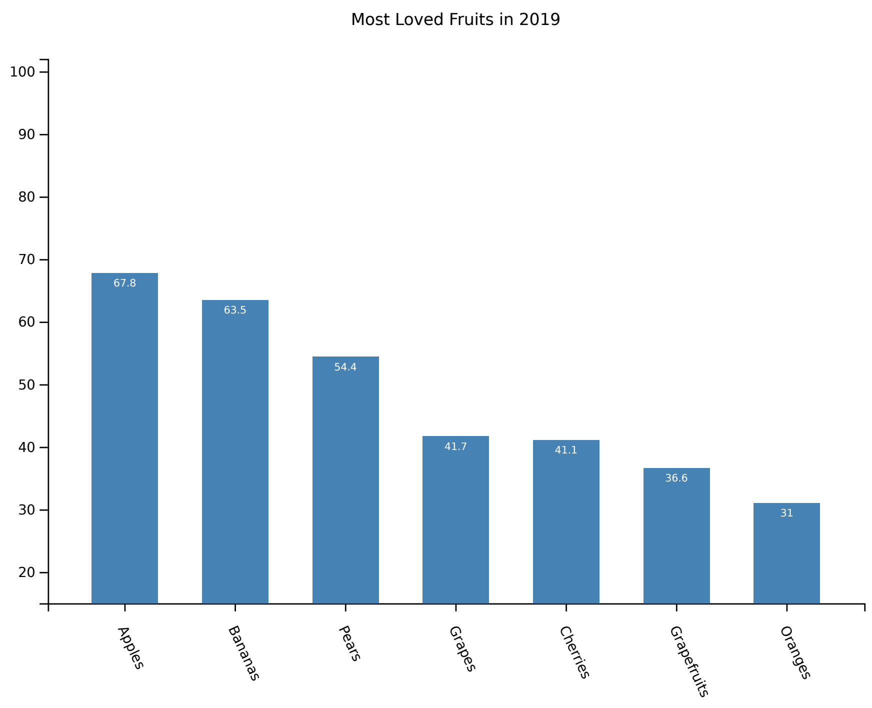
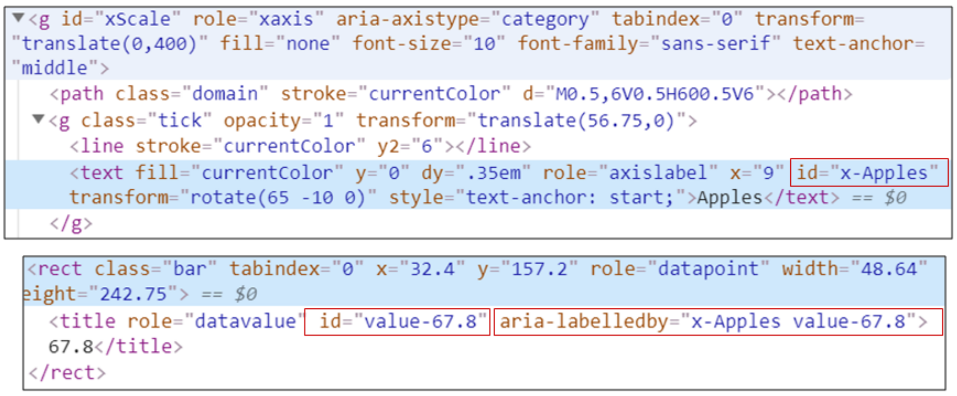


Figure 3.5: Bar Chart. [The image was created by the authors of this paper.]

```
bar.selectAll(".bar")
    .data(data)
    .enter().append("rect")
        .attr("class", "bar")
    .attr("tabindex", "0")
        .attr("x", function(d: any) { return xScale(d[key]); })
        .attr("y", function(d: any) { return yScale(d[value]); })
        .attr("role", "datapoint")
        .attr("width", xScale.bandwidth())
        .attr("height", function(d: any) { return height - yScale(d[value])
           ; })
        .append("title").text(function(d: any) { return d[value]; })
              .attr("role", "datavalue")
              .attr("id", function(d: any) { return "value-" + d[value]; })
              .attr("aria-labelledby",
        function(d:  any)  return "x-" + d[key] + " value-" + d[value]; );
```

Listing 3.7: Appends a Rect Element for each Data Point.



```
▼<g id="xScale" role="xaxis" aria-axistype="category" tabindex="0" transform=
"translate(0,400)" fill="none" font-size="10" font-family="sans-serif" text-anchor=
"middle">
   <path class="domain" stroke="currentColor" d="M0.5,6V0.5H600.5V6"></path>
 ▼<g class="tick" opacity="1" transform="translate(56.75,0)">
     <line stroke="currentColor" y2="6"></line>
     <text fill="currentColor" y="0" dy=".35em" role="axislabel" x="9" id="x-Apples"
     transform="rotate(65 -10 0)" style="text-anchor: start;">Apples</text> == $0
   </g>
```

```
<rect class="bar" tabindex="0" x="32.4" y="157.2" role="datapoint" width="48.64"
eight="242.75"> == $0
  <title role="datavalue" id="value-67.8" aria-labelledby="x-Apples value-67.8">
  67.8</title>
</rect>
```

Figure 3.6: Bar Chart Aria-Labelledby Attribute. [The image was created by the authors of this paper.]

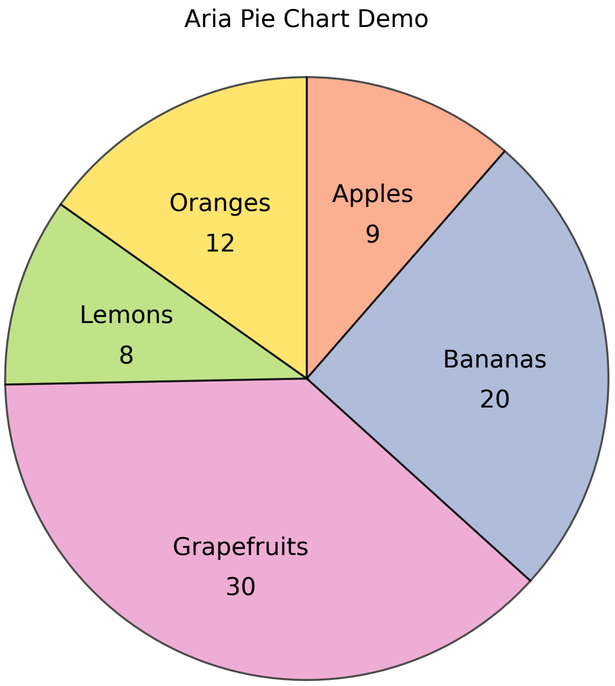


Figure 3.7: Pie Chart. [The image was created by the authors of this paper.]

3.4 Pie Chart

The pie chart as shown in Listing 3.7 is a series of paths and every path is built with the D3 arc()-function. Each "path" element gets the role "datapoint" assigned and is keyboard focusable. To each path a "title" element is appended and the role "datavalue" is assigned. For the "aria-labelledby" attribute two IDs are given, one for the label and one for the value. An ID for the current path is added with the current value. Listing 3.8 shows the procedure for the path elements. The value inside each slice is a "text" element with an assigned ID and all text elements for the labels are grouped together. We have two groups for the labels. The first one is the key entry which in this case are: Apples, Bananas, Grapefruits, Lemons, and Oranges. The second one is the value entry and are: 9, 20, 30, 8 , and 12. To get the best coordinates for both groups of labels the D3 arc()-function lets the user define a custom inner radius and outer radius and offers a centroid method as shown in Listing 3.9.

Figure 3.8 shows a section of the DOM of the pie chart. The first element is the label of the first entry Apples. The "text" element has the associated ID. The next element is the group element of the data

```
rootNode
    .selectAll("g")
    .selectAll("path")
    .data(data_ready)
    .enter()
    .append('path')
        .attr("role", "datapoint")
        .attr("tabindex", "0")
        .attr('d', d3.arc()
        .innerRadius(0)
        .outerRadius(radius))
        .attr('fill', function (d: any) { return (color(d.data[key])); })
        .attr("stroke", "black")
        .style("opacity", 0.7)
    .append("title")
        .attr("role", "datavalue")
        .attr("aria-labelledby",
        function(d:  any)  return "l-" + d.data[key] + " value-" + d.data[value]; )
        .attr("id", function (d: any) { return "value-" + d.data[value]; })
        .text(function (d: any) { return  d.data[value]; });
```

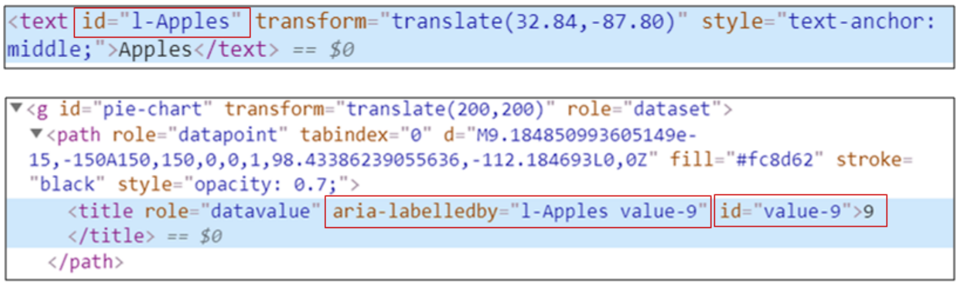
Listing 3.8: Appends a Path Element for each Data Value.

```
// Another arc that won't be drawn. Just for labels positioning
var outerArc = d3.arc()
  .innerRadius(0)
  .outerRadius(radius * 1.25);

chart
    .selectAll("g")
    .data(data_ready)
    .enter()
    .append("text")
        .text(function(d: any) { return d.data[key]; })
        .attr("id", function (d: any) { return "l-" + d.data[key]; })
        .attr("transform", function(d: any) { return "translate(" +
           outerArc.centroid(d) + ")"; })
        .style("text-anchor", "middle");

chart
  .selectAll('g')
  .data(data_ready)
  .enter()
    .append("text")
    .text(function(d: any) { return d.data[value]; })
    .attr("transform", function(d: any) { return "translate(" + outerArc.
       centroid(d) + ")"; })
    .attr("dy", "20")
    .style("text-anchor", "middle");
```

Listing 3.9: Pie Chart Labels.



```
<text id="l-Apples" transform="translate(32.84,-87.80)" style="text-anchor:
middle;">Apples</text> == $0
```

```
▼<g id="pie-chart" transform="translate(200,200)" role="dataset">
  ▼<path role="datapoint" tabindex="0" d="M9.184850993605149e-
  15,-150A150,150,0,0,1,98.43386239055636,-112.184693L0,0Z" fill="#fc8d62" stroke=
  "black" style="opacity: 0.7;">
      <title role="datavalue" aria-labelledby="l-Apples value-9" id="value-9">9
      </title> == $0
    </path>
```

Figure 3.8: Pie Chart Aria-Labelledby Attribute. [The image was created by the authors of this paper.]

area and contains all the paths of the chart. The first path element contains the first value in this case 9. The "title" element has the "aria-labelledby" attribute with the two IDs. The screen reader will first read Apples and then the value which is 9.

Chapter 4

Concluding Remarks

A general description of the ARIA standard and its usage with SVG was given. In addition, a tool for easily generating SVG programmatically with D3 was implemented. The next step is to take this concept further into Vega and let Vega generate accessible graphs by implementing our work in the underlying renderer. It was shown in this work, that is possible to modify D3 in a way to generate accessible graphs and since Vega builds upon D3 it is possible for Vega too. What is remaining is to figure out in more detail how to add valuable meta data to Vegas renderer's output and then generate the right annotations. After this it will be possible to take it another step further and integrate this concept into Vega Lite too. If future people are willing to take on this task, it will be beneficial to ask questions in the official Vega Slack channel [Vega Community 2019].

Bibliography

Bogensberger, Bryan [2019]. *NPM*. `https://npmjs.com/` (cited on page 7).

Vega Community [2019]. *Vega Slack*. `https://communityinviter.com/apps/vega-js/join` (cited on page 19).

W3C [2017]. *Accessible Rich Internet Applications (WAI-ARIA) 1.1*. 14 Dec 2017. `https://w3.org/TR/2017/REC-wai-aria-1.1-20171214` (cited on page 1).

W3C [2018]. *WAI-ARIA Graphics Module*. 02 Oct 2018. `https://w3.org/TR/2018/REC-graphics-aria-1.0-20181002` (cited on page 1).

w3schools [2019]. *JavaScript HTML DOM*. `https://w3schools.com/js/js_htmldom.asp` (cited on page 9).