Multiple Views and Advanced Interactivity

Linked Interactions

Visualizations are often not just single charts. More complex analysis tasks may require to show different perspectives, such as *overview* and *detail*, or to facet data across multiple views. Linked highlighting and interaction techniques are essential tools to allow users to trace data points across views.

Basic linkage

We will first walk through a very basic mechanism to link two charts. This is good enough for some cases but more complex visualizations necessitate an event handling controller, that we will introduce later.

We want to visualize hiking trails near Vancouver, as illustrated in the figure below. The chart on the left shows a scatter plot and the difficulty level (*easy*, *intermediate*, *difficult*) is color-coded. The number of hikes in these three categories is not immediately obvious without counting each dot. For this purpose, we add a bar chart that can simultaneously serve as a filter. When users click on one of the bars, the data in the scatter plot is filtered accordingly.

In the following, we describe one possible implementation workflow:

1. **Setup**

   We create three JS files, `main.js`, `scatterplot.js`, and `barchart.js`, in addition to the HTML and CSS files.

   - In `main.js`, we load the data and initialize the two vis classes:

```
let data, scatterplot, barchart;
d3.csv('data/vancouver_trails.csv')
   .then(_data => {
     data = _data;

     // ... data preprocessing etc. ...

     scatterplot = new Scatterplot(config, data);
     scatterplot.updateVis();

     barchart = new Barchart(config, data);
     barchart.updateVis();
   });
```

- We implement two classes `Scatterplot` and `Barchart` as learned earlier. For the scatter plot, it is important that we use D3's enter-update-exit pattern because we don't want to remove and redraw the entire chart whenever the data changes.

2. **Filtering mechanism** in `main.js`
   We add a global array to store active filter options.

```
let difficultyFilter = [];
```

We create a new function to filter the data that is shown in the scatterplot. All the data is shown when no filters have been selected.

```
function filterData() {
  if (difficultyFilter.length == 0) {
    scatterplot.data = data;
  } else {
    scatterplot.data = data.filter(d =>
difficultyFilter.includes(d.difficulty));
  }
  scatterplot.updateVis();
}
```

3. **Add event listener** in `barchart.js`
   Whenever users click on a *bar*, we update the selection and call `filterData()` to trigger a change in the scatter plot.

```
const bars = svg.selectAll('.bar')
    .data(aggregatedData, xValue)
  .join('rect')
    .attr('class', 'bar')
    .attr('x', d => xScale(xValue(d)))
    // ... other attributes ...
    .on('click', function(event, d) {
     // Check if filter is already active
      const isActive = difficultyFilter.includes(d.key);
      if (isActive) {
        // Remove filter
        difficultyFilter = difficultyFilter.filter(f => f !== d.key);
```

```
          } else {
            // Add filter
            difficultyFilter.push(d.key);
          }
          // Call global function to update scatter plot
          filterData();

          // Add class to style active filters with CSS
          d3.select(this).classed('active', !isActive);
        });
```

4. Style bar chart filters in style.css

```
    .bar:hover {
      stroke: #777;
    }
    .bar.active {
      stroke: #333;
    }
```

Multi-View Event Handler (d3.dispatch)

In the previous example, we showed how to use a global array and a `filterData()` function to link two views. An alternative approach would be to update the scatter plot directly within the bar chart class whenever the selection changes.

However, to ensure good programming practice, especially with more complex visualizations, the components should remain independent and avoid the use of too many global variables. We will introduce `d3.dispatch()` to create a centralized event handling mechanism that scales well for many views. For larger web-based visualization projects, you might also want to consider React, AngularJS, or other frameworks that provide event handling functionalities but this is beyond the scope of this course.

In the following, we describe how to modify the previous example in order to use an event handler:

1. **Initialize dispatcher**

   We initialize a dispatcher that is used to orchestrate events in `main.js`.

   ```
   const dispatcher = d3.dispatch('filteredCategories');
   ```

   In our example, we just register a single event (`filteredCategories`) but we could expand this easily to a set of events:

   ```
   d3.dispatch('filteredCategories', 'selectedPoints', 'reset');
   ```

2. **Pass the dispatcher to the vis component during the instantiation**

```
barchart = new Barchart({
        parentElement: '#barchart'
}, dispatcher, data);
```

3. **Listen for mouse events**

   In the `barchart.js`, we bind a click listener to each bar/rectangle similar to the example earlier. Whenever a bar is selected, we add the class "active", get an array with the names of all active categories, and call the dispatcher. A second click makes a category inactive.

```
// Previous D3 code / attributes of the SVG rectangle ...
.attr('class', 'bar')
.on('click', function(event, d) {
  // Check if current category is active and toggle class
  const isActive = d3.select(this).classed('active');
  d3.select(this).classed('active', !isActive);

  // Get the names of all active/filtered categories
  const selectedCategories = vis.chart.selectAll('.bar.active').data().map(k => k.key);

  // Call dispatcher and pass the event name, D3 event object,
  // and our custom event data (selected category names)
  vis.dispatcher.call('filterCategories', event, selectedCategories);
});
```

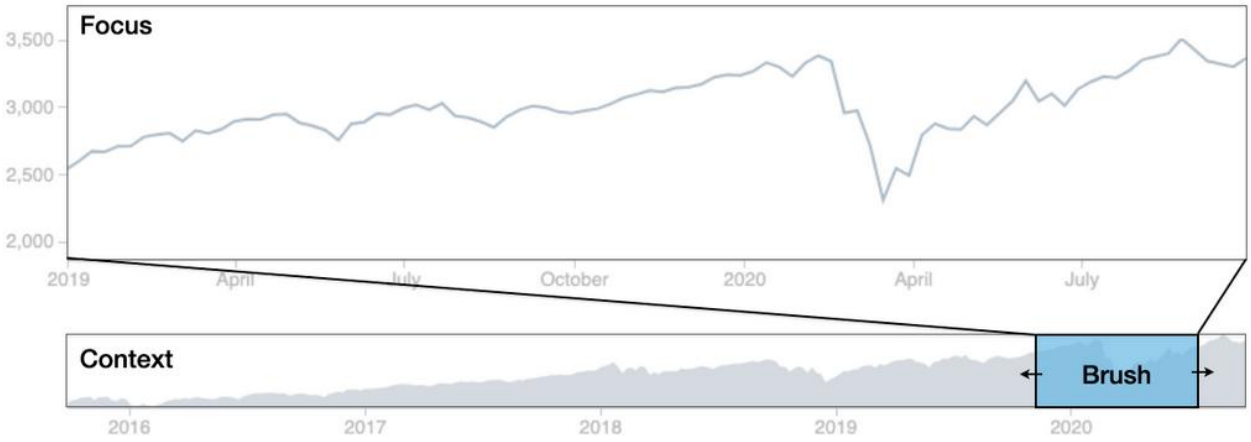
4. **Orchestrate events using the dispatcher**

   In `main.js`, we just need to wait until the `filterCategories` event gets triggered. We filter the data based on the selected categories and update the scatter plot. Here, it is important to not override the original data (`data`) with the filtered data.

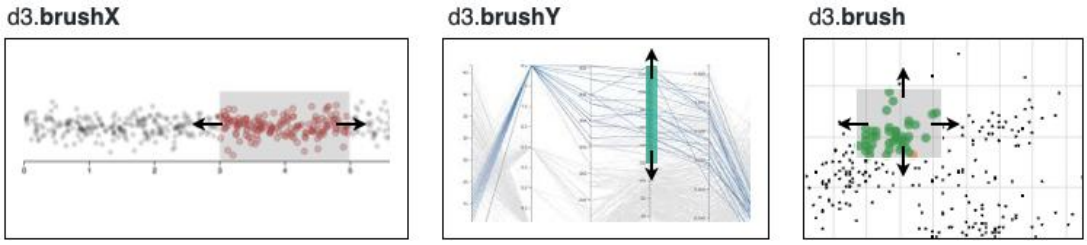
```
dispatcher.on('filterCategories', selectedCategories => {
        if (selectedCategories.length == 0) {
                scatterplot.data = data;
        } else {
                scatterplot.data = data.filter(d => selectedCategories.includes(d.difficulty));
        }
        scatterplot.updateVis();
});
```

Brushing & Linking

Brushing is a technique to interactively select a region or a set of data points in a visualization. In combination with linking, where changes are automatically dispatched to linked visualizations, we can create powerful multi-view visualizations. A popular use case is the *focus + context* visualization shown below. The *context view* provides a global perspective at reduced detail and allows users to brush. The *focus view* shows the selected data points, for example, a specific time period, in greater detail.



There are three types of brushes in D3 for brushing along the x, y dimensions, or both: `d3.brushX` (e.g., select time period), `d3.brushY` (e.g., select range in parallel coordinates plot), and `d3.brush` (e.g., select points in a scatter plot matrix). Each brush defines a selection in screen coordinates.



In the following code snippets we show you how to use `d3.brushX` but it is straightforward to adopt the workflow for other brush types. First, we need to define a scale function:

```
// Initialize time scale (x-axis)
const xScale = d3.scaleTime()
        .range([0, width])
        .domain(d3.extent(data, d => d.timestamp));
```

Then we initialize the brush. The brushable area is specified via `extent()`. In most cases, we only need to specify the *width* and *height*, and the top coordinates are just [0,0]. In addition, we listen to two types of events (`brush` and `end`) and call the functions `brushed` and `brushended` accordingly.

```
const brush = d3.brushX()
   .extent([[0, 0], [width, height]])
   .on('brush', brushed)
   .on('end', brushended);
```

We also need to append the brush component to the SVG area to make it accessible to the user:

```
const brushG = svg.append('g')
        .attr('class', 'brush x-brush')
        .call(brush);
```

Finally, we wait for the brush events. When a brush event listener is invoked, it receives the current brush event which contains the current `selection` in pixel coordinates (`[x0, x1]`). After both events, we check if the selection is empty to know if the brush is still active or if it has been removed, for example, when the user just makes a single click to reset it.

```
function brushed({selection}) {
    if (selection) {
        const selectedDomain = selection.map(xScale.invert, xScale);
        // Do something with the new selection
        // ...
    }
}
```

And

```
function brushended({selection}) {
    if (!selection) {
        // Brush has been removed
        // Probably we want to reset other views afterwards
        // ...
    }
}
```

D3 Shape Generators and Layouts

Visualizations typically consist of discrete graphical marks, such as circles, rectangles, symbols, arcs, lines and areas. While the rectangles of a bar chart or the points in a scatter plot may be easy enough to generate directly using SVG, other shapes are more complex.

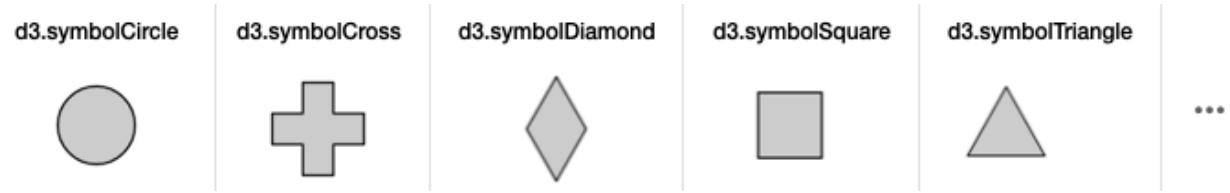
D3 provides functions — so-called *shape generators* — to help us with the creation of more complex shapes.

The D3 shape generators have no direct visual output but instead take the data you provide and transform it, thereby generating new data that is more convenient to draw.

In the following, we introduce two more shape generators: *symbols* and *stacks*. There are many more functions, such as arcs, pies, and links, that you can look up in the D3 documentation.

Symbols

Symbols are commonly used in scatter plots as a channel to encode categorical attributes and can be created in D3 using the shape generator function `d3.symbol()`. For example, we can generate the SVG path of a diamond symbol with `d3.symbol().type(d3.symbolDiamond)()`.



Example usage in a scatter plot that uses three different symbols for the categories *"Easy"*, *"Intermediate"*, and *"Difficult"*.

1. Initialize ordinal scale

   The output range are the three symbols: circle, square, and diamond.

```
const symbolScale = d3.scaleOrdinal()
  .range([
        d3.symbol().type(d3.symbolCircle)(),
        d3.symbol().type(d3.symbolSquare)(),
        d3.symbol().type(d3.symbolDiamond)()
  ])
  .domain(['Easy', 'Intermediate', 'Difficult']);
```

2. Append symbols to SVG

```
3.  const symbols = svg.selectAll('.symbol')
4.    .data(data)
5.    .enter()
6.   .append('path')
7.    .attr('class', 'symbol')
8.    .attr('transform', d => `translate(${xScale(d.time)}, ${yScale(d.distance)})`)
9.    .attr('d', d => symbolScale(d.difficulty));
```

Stacks

We have previously shown how to draw rectangles and how to position them within an SVG area. If we want to draw a stacked bar chart, we could manually calculate the various x- and y-positions of each rectangle but this can become complicated quickly, in particular if we want to draw areas (using the `path` element) for a stacked area chart or a streamgraph.

Conveniently, D3 provides a *stack generator* that is doing all the calculations for us and we just need to draw the generated coordinates. `d3.stack()` computes a baseline value for each datum, so we can *stack* layers of data on top of each other, where each layer corresponds to one SVG element.

Example data:

```
const data = [
```

```
        { 'year': 2015, 'milk': 10, 'water': 4 },
        { 'year': 2016, 'milk': 12, 'water': 6 },
        { 'year': 2017, 'milk': 6, 'water': 7 }
];
```

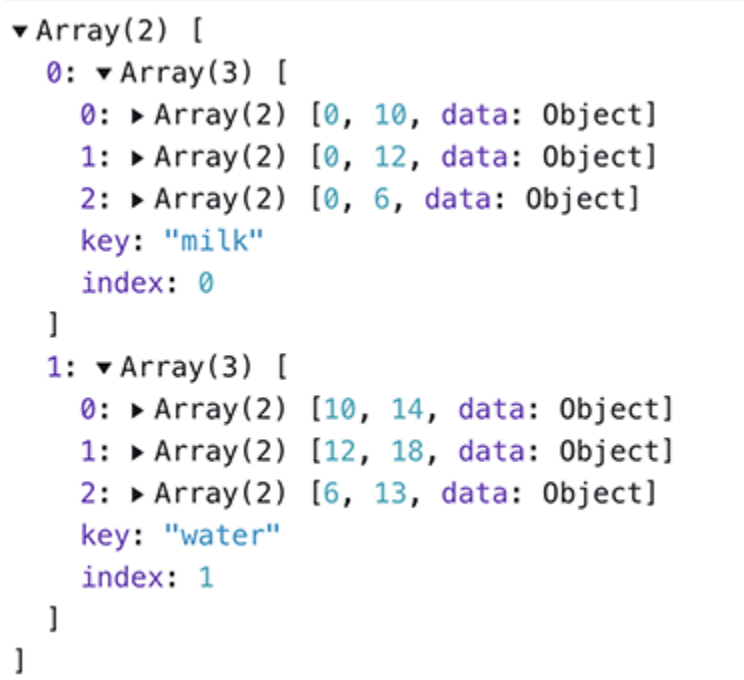
We initialize a stack generator and specify the categories or layers that we want to show in our chart:

```
const stack = d3.stack().keys(['milk', 'water']);
```

Compute stacked data:

```
const stackedData = stack(data);
console.log(stackedData)
```

When we print the data, we can see the computed values: milk (0 to 10, 0-12, 0-6) and water (10-14, 12-18, 6-13).



We can now use this data to draw, for example, a stacked bar chart. Similar to an ordinary bar chart, you need to create x- and y-scales first. The x-position and width is based on the year (`d.data.year`), and the y-position and height of each rectangle is based on the computed layers (stored in `d[0]` and `d[1]`). We add a CSS class to each SVG group to adjust the colour. This could be also done in SVG using the `fill` attribute.

```
const rectangles = svg.selectAll('category')
                .data(stackedData)
        .join('g')
                .attr('class', d => `category cat-${d.key}`)
        .selectAll('rect')
```

```
                .data(d => d)
        .join('rect')
                .attr('x', d => xScale(d.data.year))
                .attr('y', d => yScale(d[1]))
                .attr('height', d => yScale(d[0]) - yScale(d[1]))
                .attr('width', xScale.bandwidth());
```