Tutorial 4
Interactivity

DATA VISUALISATION

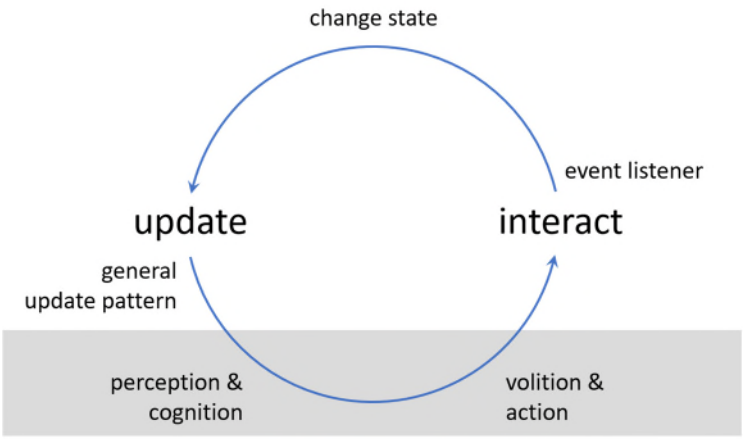
Alfonso Bueno Orovio
DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF OXFORD | HILARY TERM 2023

Tutorial 4: Interactivity

In this tutorial, we introduce examples on mouse events and interactions with web components, in order to make your D3 visualisations fully interactive.

1. The Unidirectional Data Flow pattern

An important *design pattern* when handing user input in visualisations is that of "Unidirectional Data Flow", illustrated in the figure below. Any user interaction (captured by its corresponding event listener) should incur into a change of internal state, and trigger an update of the visualisation in order to make it fully consistent with its current state (we will use for this the D3's general update pattern). An important consideration about this design pattern is that it establishes an architecture where the user is in the picture, using their perception and cognition to interpret the visualisation under any updates, and their volition and action to interact with it as required (shaded region of the figure).



2. D3 event listeners

Click events

Let's visualise a shopping ticket (see full codes in `ex4.1_click_events/`). Each item in the receipt will have fields `type`, `price`, and `id` (unique in-store identifier). Building on our previous tutorials, the function `showItems` below implements the general update pattern for a simple chart where each item is represented as a circle, with its radius encoding the price of the item:

```
export const showItems = (parent, props) => {
  // Unpack my properties
  const data = props;
  const height = +parent.attr('height');

  const circles = parent.selectAll('circle').data(data, d => d.id);
  const circlesEnter = circles.enter().append('circle')
    .attr('cx', (d,i) => (i * 100) + 80);
  circlesEnter.merge(circles)
    .attr('cy', height/2)
    .transition().duration(500)
      .attr('cx', (d,i) => (i * 100) + 80)
      .attr('r', d => d.price);
  circles.exit().remove();
}
```

Our main `index.js` only needs to select the SVG element and call the function on the data to initialise the chart (here using the wrapper `updateVis`, to make clearer the use of the unidirectional data flow pattern):

```
import {showItems} from './showItems.js';

const svg = d3.select('svg');

const data = [
  { 'type': 'meat',       'price': 40, 'id': 233 },
  { 'type': 'fruit',      'price': 35, 'id':  58 },
  { 'type': 'meat',       'price': 20, 'id': 112 },
  { 'type': 'diary',      'price': 10, 'id': 779 },
  { 'type': 'vegetables', 'price': 35, 'id': 287 },
  { 'type': 'vegetables', 'price': 10, 'id':  11 },
  { 'type': 'fruit',      'price': 50, 'id':  82 }
];

const updateVis = () => {
  showItems(svg, data);
};

// Initialise visualisation
updateVis();
```

Initial output:

We now want to add some interactivity, highlighting an element with a thick black border when we click on it. In general, it is recommended that the main JS code holds the interactivity. We can create there a state variable `selectedItem` (initialised to `null`) to keep track of the `id` of the selected item. This will need to be passed to `showItems` as part of its `props`, in order to update the items upon any changes in the state[1]. We can do this easily as:

```
  circlesEnter.merge(circles)
    .attr('cy', height/2)
    .attr('stroke-width', 5)
    .attr('stroke', d => d.id === selectedItem ? 'black' : 'none')
    .transition().duration(500)
      .attr('cx', (d,i) => (i * 100) + 80)
      .attr('r', d => d.price);
```

We now need to define a function that enables us to interactively modify our state when clicking on a circle. Functions for mouse event handlers in D3 v7 have 2 inputs (the calling event and the specific datum), as:

```
const setSelectedItem = (event, d) => {
  selectedItem = d.id;
  updateVis();
}
```

[1] Inside the `merge` section of the general update pattern, as this constitutes a change of already existing elements.

In the definition of `setSelectedItem` above, note the visualisation is updated immediately upon changes in the state to make it consistent with the internal state representation, following the unidirectional data flow design pattern. As clicking an element implies a change on its current properties, we can capture it as part of the `merge` section of its general update pattern (adding a click listener event by means of D3's Event Listeners). Obviously, `setSelectedItem` also becomes part of the `props` of `showItems`, so the latter can invoke it:

```
  circlesEnter.merge(circles)
    .attr('cy', height/2)
    .attr('stroke-width', 5)
    .attr('stroke', d => d.id === selectedItem ? 'black' : 'none')
    .on('click', setSelectedItem)
    .transition().duration(500)
      .attr('cx', (d,i) => (i * 100) + 80)
      .attr('r', d => d.price);
```

By implementing such simple changes, we have enabled on-click interaction in our visualisation, making sure our chart always reflects its current internal state[2].

Output (on click):

Hovering events

Our example above can be easily adapted to accommodate hovering events[3] (i.e., when the cursor moves over or out a specific element). In the D3's Event Listeners API, the highlighting action becomes as easy as replacing the type of event from `'click'` to `'mouseover'`. However, this would have the drawback of the element remaining highlighted when the cursor leaves the selection until it moves over a new circle. Nevertheless, we can easily address this issue by adding a second event listener for `'mouseout'` events:

```
  circlesEnter.merge(circles)
    .attr('cy', height/2)
    .attr('stroke-width', 5)[4]
    .attr('stroke', d => d.id === selectedItem ? 'black' : 'none')
    .on('mouseover',  setSelectedItem)
    .on('mouseout', unsetSelectedItem)
    .transition().duration(500)
      .attr('cx', (d,i) => (i * 100) + 80)
      .attr('r', d => d.price);
```

with the function `unsetSelectedItem` simply resetting the state variable when the cursor leaves:

```
const unsetSelectedItem = (event, d) => {
  selectedItem = null;
  updateVis();
}
```

[2] Please go through the code to make sure all individual changes are correctly understood.
[3] See corresponding codes in folder `ex4.2_hovering_events/`
[4] If no colour is specified, the stroke remains not visible.

3. Interactivity using CSS

Basic interactions such as hovering can be more easily handled using CSS. In this case, it suffices to add the `stroke-width` attribute to the circles and the rule below to our CSS file (see `'ex4.3_hovering_CSS/'`):

```
circle:hover {
  stroke: black;
}
```

to yield the desired effect. In addition, we can add a summary information of the item that pops in if the cursor stays on the element, by appending a `'title'` to the `enter` selection of the general update pattern:

```
  const circlesEnter = circles.enter().append('circle')
    .attr('cx', (d,i) => (i * 100) + 80);
  circlesEnter.append('title')
    .text(d => `${d.type}\nPrice: £${d.price}`);
```

These are 'cheap interactivity tricks' that can be implemented with no effort using CSS in many visualisations. Nevertheless, for more sophisticated interactions, it is necessary to exploit D3's event listeners.

4. Interaction with HTML components

Implementing a select menu

For this part of the tutorial (see `'ex4.4_html_interaction/'`), our aim is to interact in D3 with an HTML select menu. If we do a web search for this type of menus, we can easily find documentation describing that these are implemented in HTML as:

```
<select name="cars" id="cars">
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

From here, if we want to implement in D3 this HTML component, our requirements seem to be to implement a single element of class `select`, to which we will append as many elements of class `option` as required, with the name of the option within the attribute `value` as well as in its `text` field. The component will be changing as we pick different options, so let's implement it using a general update pattern (with a singular element for the class `select`). In addition, if we want to make our component reusable, it seems reasonable that at least we will need to pass it an array containing the options to be shown, and the function to be executed when we select one of them. As event listener, we can use in this case the `.on('change')` listener. Putting it together in `dropdownMenu.js`:

```
export const dropdownMenu = (parent, props) => {
  const {
    options,
    onOptionSelected
  } = props;

  const select = parent.selectAll('select').data([null]);
  const selectEnter = select.enter().append('select')
    .merge(select)
      .on('change', onOptionSelected);
```

```
  const option = selectEnter.selectAll('option').data(options);
  option.enter().append('option')
    .merge(option)
      .attr('value', d => d)
      .text(d => d);
};
```

Interacting with the select menu

We are now in the position to interact with the HTML component. First, in the body of `index.html`, we can place a `div` container with an appropriate `id` in order to place our dropdown menu:

```
  <body>
    <div id="menus"></div>
    <svg width="960" height="500"></svg>
    <script src="index.js" type="module"></script>
  </body>
```

The only work left to be done in `index.js` is to define our state variable (`selectedOption`) and the function triggered by the event listener (`onOptionSelected`)[5]. Note that, upon any call to the latter, it automatically updates the chart to match its state representation, thus complying with the unidirectional data flow design pattern. A simple text element has been also added to the SVG, whose content is updated upon choosing a new option. Please explore the codes provided in detail, including the `styles.css` to see how to style some of the properties of these menus.

```
import { dropdownMenu } from './dropdownMenu.js';

const options = ['A', 'B', 'C'];
let selectedOption = options[0];  // state (consistent with first item shown)

const onOptionSelected = event => {
  //console.log(event)
  selectedOption = event.target.value;
  updateVis();
}

const updateVis = () => {
  // Dropdown menu
  dropdownMenu(d3.select('#menus'), { options, onOptionSelected });

  // Update text message on screen upon choosing an option
  const message = d3.select('svg').selectAll('text').data([selectedOption]);
  const messageEnter = message.enter().append('text')
    .attr('x',100).attr('y',50);
  messageEnter.merge(message)
    .text('My selection is: ' + selectedOption);
};
updateVis(); // Initialise visualisation
```

[5] Note that the function onOptionSelected only takes one argument, compared to those for cursor events which had two. This is because there is now no need of an additional DOM element from where we may read information (such as the clicked circles). Once the event is triggered, the selected option resides within its `target.value` field. In order to know that, the best option is to simply log to the console the triggered event, and explore the fields of the returned object.

5. Tooltips

When you create interactive visualisations, you often want to show tooltips to reveal more details about your data to your audience. Different approaches to achieve this exist, but a recommended option is to create a global tooltip container outside of the SVG that you can show/hide and position whenever users hover over a mark. This approach allows you to create more complex tooltip objects that can be styled with CSS and contain images or even small visualisations. Example implementation workflow:

- Add tooltip placeholder to the HTML file:

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

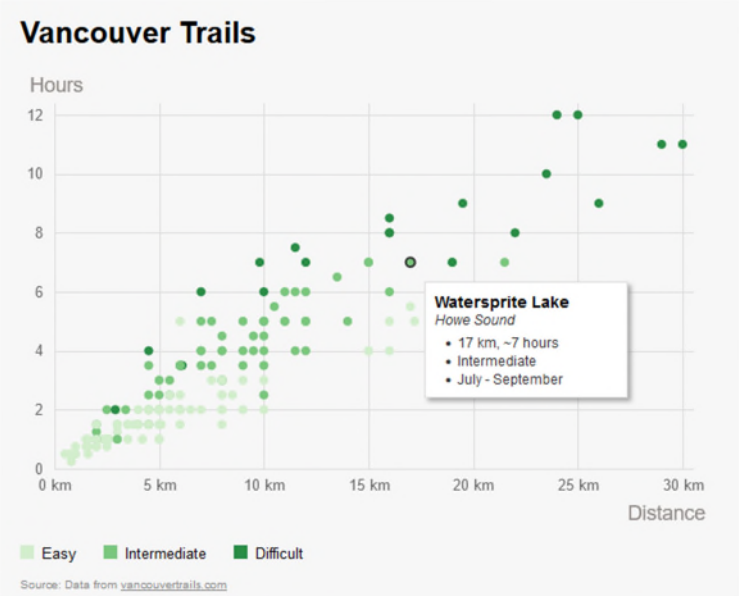
- Set absolute position, hide tooltip by default, and define additional optional styles in CSS:

```
#tooltip {
    position: absolute;
    display: none;
    /* ... other tooltip styles ... */
}
```

- In JS (D3), update tooltip content, position, and visibility when the user hovers over a mark. We distinguish between three different states: `mouseover`, `mousemove`, and `mouseleave` (in case of small marks, we add the positioning to the `mouseover` function and leave out `mousemove`).

```
myMarks
    .on('mouseover', (event,d) => {
      d3.select('#tooltip')
        .style('display', 'block')
        .html(`<div class="tooltip-label">Population</div>
              ${d3.format(',')(d.population)}`);
    })
    .on('mousemove', (event) => {
      d3.select('#tooltip')
        .style('left', (event.pageX + tooltipPadding) + 'px')
        .style('top',  (event.pageY + tooltipPadding) + 'px')
    })
    .on('mouseleave', () => {
      d3.select('#tooltip').style('display', 'none');
    });
```

Please explore `'ex4.5_tooltips/'` to see it in action![6]



[6] For clarity, I've removed in this example the extra complexity of the general update pattern in the scatterplot generator. However, remember to make use of it to avoid plotting and overlapping your SVG elements multiple times!