Data Visualization

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

Web programming: Html, css, and JS

1.1 HTML and CSS

- **SVGS with html**: An SVG (Scalable Vector Graphics) file is an XML-based image format used to display vector graphics. Unlike PNG or JPG images, SVGs scale infinitely without losing quality.

  Key properties:

  - Resolution-independent
  - Small file size for simple graphics
  - Fully stylable with CSS
  - Scriptable with JavaScript
  - Ideal for icons, diagrams, charts, and UI graphics

- **Embedding SVG directly into HTML (inline SVG)**: This is the most powerful and flexible method.

```
<svg width="200" height="100" viewBox="0 0 200 100">
    <rect x="10" y="10" width="180" height="80"
    ↪  fill="steelblue" />
    <circle cx="100" cy="50" r="30" fill="orange" />
</svg>
```

  - <svg> defines the canvas
  - width / height define display size
  - viewBox defines the internal coordinate system
  - Shapes (rect, circle, line, path) are drawn inside

  - Fully stylable with CSS
  - Can be animated
  - JavaScript access to elements
  - Best for interactive graphics

- **SVG elements**:

Element	**Purpose**
<rect>	Rectangle
<circle>	Circle
<ellipse>	Ellipse
<line>	Line
<polyline>	Connected lines
<polygon>	Closed shape
<path>	Complex shapes
<text>	Text

- **Rect**:
  - **x**: x-coordinate (top-left)
  - **y**: y-coordinate (top-left)
  - **width**: rectangle width
  - **height**: rectangle height
  - **rx**: x-axis corner radius (rounded corners)
  - **ry**: y-axis corner radius
  - **fill**
  - **stroke**
  - **stroke-width**
  - **opacity**
- **Circle**
  - **cx** center x-coordinate
  - **cy** center y-coordinate
  - **r** radius
  - **fill**
  - **stroke**
  - **stroke-width**
- **Ellipse**
  - **cx** center x-coordinate
  - **cy** center y-coordinate
  - **rx** x-radius
  - **ry** y-radius
  - **fill**
  - **stroke**
  - **stroke-width**
- **Line**
  - **x1, y1**: start point
  - **x2, y2**: end point
  - **stroke**: (required)
  - **stroke-width**:
  - **stroke-linecap** (butt, round, square)
  - **stroke-dasharray**:
- **Polyline**
  - **points**: list of coordinate pairs ”x1,y1 x2,y2 x3,y3 ...”
  - **fill**: (usually none)
  - **stroke:**
  - **stroke-width:**
  - **stroke-linejoin**:

- **Polygon**
  - **points**: list of coordinate pairs ”x1,y1 x2,y2 x3,y3 ...”
  - **fill**:
  - **stroke**:
  - **stroke-width**:
  - **fill-rule**: (nonzero, evenodd)
- **Text**
  - **x**:
  - **y**:
  - **dx**:
  - **dy**:
  - **text-anchor**: (start, middle, end)
  - **font-family**:
  - **font-size:**
  - **font-weight:**
  - **letter-spacing**:
  - **fill**:
  - **stroke**:
  - **opacity**:
- **SVG viewbox**:

  ```
  <svg viewBox="0 0 200 100">
  ```

  Means:
  - Coordinate system starts at (0, 0)
  - Width = 200 units
  - Height = 100 units

  This allows scaling without distortion.
- **CSS for <svg>**:
  - fill
  - fill-opacity
  - fill-rule
  - stroke
  - stroke-width
  - stroke-opacity
  - stroke-linecap
  - stroke-linejoin
  - stroke-dasharray
  - stroke-dashoffset

- stroke-miterlimit
- color
- opacity
- x
- y
- cx
- cy
- r
- rx
- ry
- width
- height
- transform
- transform-origin
- transform-box
- font-family
- font-size
- font-style
- font-weight
- letter-spacing
- word-spacing
- text-anchor
- dominant-baseline
- alignment-baseline
- direction
- writing-mode
- display
- visibility
- overflow
- clip-path
- mask
- filter
- cursor
- pointer-events
- animation
- animation-name
- animation-duration
- animation-delay
- animation-iteration-count
- animation-timing-function
- transition

  - transition-property
  - transition-duration

- **Paths**: The <path> element is the most powerful and flexible shape in SVG. Unlike <rect> or <circle>, a path can describe:
  - Straight lines
  - Curves
  - Arcs
  - Complex shapes
  - Icons, symbols, letters
  - Entire illustrations

  It works by following a series of drawing commands stored in the *d* attribute. The *d* string is a mini drawing language, it reads left to right.

  - **Move to (M)**: Moves the "pen" without drawing.

    $$\text{M x y}$$

    Moves to (x, y)
  - **Line to (L)**: Draws a straight line.

    $$\text{L x y}$$

    Draws a line from the current point to (x, y)
  - **Close path (Z): Closes the shape by connecting back to the start.**

    $$\text{Z}$$
  - **Quadratic Curve (Q)**: (cx, cy) = control point (x, y) = end point

    $$\text{Q cx cy x y}$$

    The control point:
    - pulls the curve
    - bends its direction
    - determines how steep or shallow the curve is

    The curve only passes through:
    - The start point
    - The end point
  - **Cubic Bézier (C)**:

    $$\text{C x1 y1, x2 y2, x y}$$

    Two control points, more control.
  - **Arc Command (Rounded Shapes) (A)**:

A rx ry x-axis-rotation large-arc sweep x y

Note the difference between lowercase and uppercase control characters

  - **Uppercase**: Absolute position
  - **Lowercase**: Relative movement

- **Triangle with path**:

```
<path d="M 50 10 L 30 80 L 70 80 Z" />
```

- **Fill and stroke**:

```
<path d="M 50 10 L 30 80 L 70 80 Z"
stroke="black"
stroke-width="6"
fill="red"
/>
```

- **Grouping**: The <g> element is used to group multiple SVG elements together so that transformations, styles, or attributes can be applied to them collectively.

```
<svg width="200" height="200">
    <g>
        <!-- SVG elements go here -->
    </g>
</svg>
```

Instead of transforming each element individually, you apply the transformation once to the group.

```
<g transform="translate(50, 50)">
    <circle cx="0" cy="0" r="20" />
    <rect x="30" y="-10" width="40" height="20" />
</g>
```

Both shapes move together. You can apply styles such as fill, stroke, opacity, etc., to all child elements.

```
<g fill="blue" stroke="black" stroke-width="2">
    <circle cx="50" cy="50" r="20" />
    <rect x="90" y="30" width="40" height="40" />
</g>
```

Events applied to a <g> affect all child elements.

```
<g onclick="alert('Clicked group!')">
    <circle cx="50" cy="50" r="20" />
    <rect x="80" y="40" width="30" height="30" />
</g>
```

1.2 Observable

- **Window**: Global variables can be defined with the global window object

```
window.data = ...
```

Then, this property can be accessed in other cells.

- **Blocks**: In observable, multi-line javascript cells must be placed in a curly brace block
- **HTML templating**: Observable provides an html tag for writing HTML declaratively:

```
html`<h1>Hello</h1>`
```

  - Returns a live DOM node, not a string.
  - HTML is parsed immediately.
  - Values inside ${...} are safely interpolated.

In a multi statement (curly brace block), this node must be returned to be rendered.

To make a table dynamically with templating,

```
html`
  <table>
    <thead>
      <tr>
        <th>Year</th>
        <th>Weekday</th>
      </tr>
    </thead>
    <tbody>
      ${data.map(d => html`
        <tr>
          <td>${d.Year}</td>
          <td>${d.Weekday}</td>
        </tr>
      `)}
    </tbody>
  </table>
```

Notice that html must be placed inside ticks.

1.3 JS

- **Var, let, and const**:
  - **Var**: Function-scoped, not block-scoped. Ignores {} blocks such as if, for, and while. Hoisted to the top of the function. Initialized as **undefined**. Can be reassigned ,can be redeclared
  - **Let**: Block-scoped, exists only inside {} where it is defined. Hoisted, but not initialized. Exists in the Temporal Dead Zone (TDZ) until declared. Can be reassigned ,cannot be redeclared in the same scope

    ```
    let x = 3;
    x = 4;      // OK
    let x = 5; // Error
    ```

  - **Const**: Block-scoped, same as let. Hoisted but in the TDZ. Must be initialized at declaration

    ```
    const z  = 10; // Good
    const y;       // Error
    ```

    Cannot be reassigned. Const prevents reassignment, not mutation.

- **Immutable types**: These cannot be changed after creation. Any “modification” creates a new value. Primitive Types are all immutable
  - number
  - string
  - boolean
  - null
  - undefined
  - symbol
  - bigint

  ```
  let s = "hello";
  s[0] = "H";   // No effect
  console.log(s); // "hello"
  ```

- **Mutable types**: These are objects and collections, whose contents can change without changing the reference.
  - Object
  - Array
  - Function
  - Date
  - Map / Set

- **Pass by value and pass by reference**: JavaScript does not technically have pass-by-reference.

  - Primitive values are passed by value
  - Objects are passed by value of their reference

  All primitives are passed by value.

  Objects are somewhat passed by reference, technically by value of reference. The reference (memory address) is copied, not the object itself.

```
function modify(obj) {
    obj.x = 10;
}

const data = { x: 1 };
modify(data);

console.log(data.x); // 10
```

  - data holds a reference to the object
  - That reference is copied into obj
  - Both point to the same object
  - Mutating the object affects both

- **Objects**: Key value pairs

```
var obj = {x: 2, y: 4}; obj.x = 3; obj.y = 5;
```

  Prototypes for instance functions. We can access properties via dot notation or [] notation. Objects may also contain functions

```
var student = {firstName: "John",
    lastName: "Smith",
    fullName: function() { return this.firstName + " " +
     ↪  this.lastName; }};
student.fullName()
```

  Note: Dot-notation only works with certain identifiers, bracket notation works with more identifiers, like if the key was a string.

- **JSON**: Data interchange format, subset of JS. Uses nested objects and arrays. Data only, no functions.
- **Functional programming in JS**
- **Unary plus operator**: The unary plus (+) operator precedes its operand and evaluates to its operand but attempts to convert it into a number, if it isn't already.
- **Map, filter, and reduce**:
  - **Map**: map transforms each element of an array using a callback function and returns a new array of the same length. It is a pure transformation; the original array is not modified.

```
arr.map((element, index, array) => newElement) -> Array
```

  - **Filter**: filter selects a subset of elements based on a predicate function and returns a new array containing only those elements for which the predicate evaluates to true.

```
arr.filter((element, index, array) => boolean) -> Array
```

  - **Reduce**: reduce aggregates an array into a single value by repeatedly combining elements using an accumulator function. It is the most general of the three operations.

```
arr.reduce((accumulator, element, index, array) =>
↪  newAccumulator, initialValue) -> any
```

- **forEach**: forEach executes a provided callback function once for each element of an array. It is intended for side effects, not for producing values.

```
arr.forEach((element, index, array) => void) -> void
```

- **Object manipulation**
  - **Deleting properties**: We use the delete operator

```
delete person.age;
```

- **JS functions are objects**: We can assign properties to functions

```
function f() {}

f.x = 10;
f.y = 15;
```

- **Creating html elements manually with JS (SVGs)**:
  - **document.createElementNS**: We use create element to create new elements, to create a new svg element,

```
const new_element = document.createElementNS("http://ww⌟
↪  w.w3.org/2000/svg", "svg");
```

  - **setAttribute**: We can set attributes on our element with setAttribute

```
new_element.setAttribute(name: string, value);
```

  - **appendChild**: We use append child to append child to a root node in our document tree

```
root_element.appendChild(new_element);
```

- **createTextNode(value)**: Creates a plain text node

```
root_element.appendChild(document.createTextNode("Hello
↪ world"));
```

- **General JS to create HTML dynamically**:
  - **document.createElement(tagName)**: Creates a new DOM element but does not place it in the document.

    ```
    const div = document.createElement("div");
    ```

  - **document.createTextNode(text)**:

    ```
    const text = document.createTextNode("Hello world");
    div.appendChild(text);
    ```

  - **element.textContent**: Assigns or retrieves text content (preferred in most cases).

    ```
    div.textContent = "Hello world";
    ```

  - **div.setAttribute("class", "container")**:

    ```
    div.setAttribute("class", "container");
    ```

  - **getAttribute(name)**:
  - **removeAttribute(name)**:
  - **hasAttribute(name)**:
  - **parent.appendChild(child)**: Appends as the last child.

    ```
    document.body.appendChild(div);
    ```

  - **parent.insertBefore(newNode, referenceNode)**: Inserts at a specific position.

    ```
    parent.insertBefore(div, parent.firstChild);
    ```

  - **element.append(...)**:

```javascript
document.body.append(div);
```

  - **element.prepend(...):**
  - **element.before(...):**
  - **element.after(...)**:
  - **element.innerHTML**: Parses an HTML string and replaces the element's contents.

```javascript
div.innerHTML = "<strong>Hello</strong>";
```

  - **document.getElementById(id)**:
  - **document.querySelector(cssSelector)**:

```javascript
const container = document.querySelector(".container");
container.append(div);
```

  - **document.querySelectorAll(cssSelector)**:
  - **element.cloneNode(deep)**: Copies an element.

```javascript
const copy = div.cloneNode(true);
```

  - **element.addEventListener(type, handler)**:

```javascript
button.addEventListener("click", handleClick);
```

- **Direct property assignment (often cleaner and safer):**

```javascript
div.id = "main";
div.className = "container";
```

- **Element properties**:
  - **id**: Unique element identifier
  - **className**: Space-separated CSS classes
  - **classList**: Token-based class API (add, remove, toggle)
  - **tagName**: Uppercase tag name (read-only)
  - **nodeName**: Same as tagName for elements
  - **nodeType**: Numeric node type (1 = Element)
  - **style**: Inline style object
  - **hidden**: Shortcut for display: none

D3

- **Data-driven documents**: D3.js (Data-Driven Documents) is a JavaScript library for producing dynamic, interactive data visualizations in web browsers using standard web technologies — primarily HTML, SVG, and CSS. Unlike charting libraries that provide predefined plots, D3 is a low-level visualization toolkit that gives fine-grained control over how data maps to visual elements.

  D3's central abstraction is that data drives document structure. You associate data with elements on a page, and D3 computes what should be created, updated, or removed.

  Conceptually,

  $$\text{Data} \rightarrow \text{Visual elements}$$

- **Key features**:
  - Supports data as a core piece of Web elements
    - Loading data
    - Dealing with changing data (joins, enter/update/exit)
    - Correspondence between data and DOM elements
  - Selections (similar to CSS) that allow greater manipulation
  - Method chaining
  - Integrated layout algorithms
  - Focus on interaction support

- **Data Binding (Selections)**: D3 can bind arrays or objects directly to DOM elements.

  ```
  d3.selectAll("p")
  .data([10, 20, 30])
  .text(d => d);
  ```

  This replaces each paragraph's text with the corresponding data value.

- **Select and selectAll**:
  - **d3.select(selector)**: Selects the first element that matches the specified selector string. If no elements match the selector, returns an empty selection. If multiple elements match the selector, only the first matching element (in document order) will be selected.

    If the selector is not a string, instead selects the specified node; this is useful if you already have a reference to a node
  - **selection.select(selector)**: For each selected element, selects the first descendant element that matches the specified selector string

  If the selector is a function, it is evaluated for each selected element, in order, being passed the current datum (d), the current index (i), and the current group (nodes),

- **d3.selectAll(selector)**: Selects all elements that match the specified selector string. The elements will be selected in document order (top-to-bottom). If no elements in the document match the selector, or if the selector is null or undefined, returns an empty selection.

  If the selector is not a string, instead selects the specified array of nodes
- **selection.selectAll(selector)**: For each selected element, selects the descendant elements that match the specified selector string. The elements in the returned selection are grouped by their corresponding parent node in this selection.

- **selection.selectAll with function example**

```
function run5b(svg) {
    // arrow functions
    svg.selectAll("rect")
        .attr("x", 0)
        .attr("y", (d,i) => i*90+50)
        .attr("width", (d,i) => i*150+100)
        .attr("height", 20)
        .style("fill", "steelblue")
}
```

- **Enter-update-exit pattern**: When data size changes, D3 determines
  - **Enter:** New data → create elements
  - **Update:** Existing data → modify elements
  - **Exit:** Removed data → delete elements
- **Changing attributes**:

```
var r1 = svg.select("rect").attr("width",
↪  100).attr("height", 100);
```

- **Changing styles**:

```
var r1 = svg.select("rect").style("fill", "blue");
```

- **Binding data**: We use

```
selection.data(data: arr, key?: fn)
```

  Binds the specified array of data with the selected elements, returning a new selection that represents the update selection: the elements successfully bound to data. Also defines the enter and exit selections on the returned selection, which can be used to add or remove elements to correspond to the new data.

the key parameter in .data(data, key) specifies how data items are matched to existing DOM elements during a data join.

If we write

```
selection.data(data)
```

D3 matches data to elements by index (position):

– First data item → first element

– Second → second

etc... With a key,

```
selection.data(data, key)
```

D3 matches elements by identity, not position. The key function returns a unique identifier for each datum.

```
(d, i) => keyValue
```

Suppose the data is

```
const data = [
    { id: 101, name: "Alice" },
    { id: 102, name: "Bob" }
];
```

Then, the key could be

```
svg.selectAll("circle")
.data(data, d => d.id);
```

D3 now tracks circles by id, not position.

- **The enter selection**: In D3, the enter selection represents data items that do not yet have corresponding DOM elements. It is the mechanism by which new visual elements are created when the dataset grows.

  After a data join,

```
const sel = selction.data(data);
```

D3 partitions the result into three disjoint subsets

- Update — data matched to existing elements
- Enter — data with no element yet
- Exit — elements with no data anymore

.enter() returns a special selection containing placeholder nodes — not actual DOM elements. Each placeholder corresponds to one new data item. You cannot style or position them until you append real elements.

```
selection.data(data)
    .enter()
    .append("circle");
```

"For each new datum, create a new <circle> element."

```
function run8(svg) {
    var selection = svg.selectAll("rect")
    .data([127, 61, 256, 71])

    selection
        .attr("x", 0)
        .attr("y", (d,i) => i*90+50)
        .attr("width", d => d)
        .attr("height", 20)
        .style("fill", "steelblue")

    selection.enter().append("rect")
        .attr("x", 10) // let's just put it somewhere
        .attr("y", 10)
        .attr("width", 30)
        .attr("height", 30)
        .style("fill", "green")
}
```

- **Appending**:

```
selection.append(type)
```

If the specified type is a string, appends a new element of this type (tag name) as the last child of each selected element, or before the next following sibling in the update selection if this is an enter selection. The latter behavior for enter selections allows you to insert elements into the DOM in an order consistent with the new bound data

If the specified type is a function, it is evaluated for each selected element, in order, being passed the current datum (d), the current index (i), and the current group (nodes), with *this* as the current DOM element (nodes[i]). This function should return an element to be appended. (The function typically creates a new element, but it may instead return an existing element.)

- **Inserting**:

```
selection.insert(type, before?)
```

If the specified type is a string, inserts a new element of this type (tag name) before the first element matching the specified before selector for each selected element. For example, a before selector :first-child will prepend nodes before the first child. If before is not specified, it defaults to null.

- **Merge**: .merge() combines two selections into one — most commonly the enter selection and the update selection — so that you can apply the same operations to both newly created and already existing elements.

  ```
  function run10(svg) {
      var selection = svg.selectAll("rect")
      .data([127, 61, 256, 71]);

      selection.enter().append("rect")
          .merge(selection)
          .attr("x", 0)
          .attr("y", (d,i) => i*90+50)
          .attr("width", d => d)
          .attr("height", 20)
          .style("fill", "steelblue");
  }
  ```

  Here, the enter section is merged with the update section.

- **Join**: .join() is a high-level method that performs the complete data join lifecycle — handling enter, update, and exit selections in a single operation. It is the modern replacement for the classic pattern using .enter(), .merge(), and .exit().

  Formally, .join() orchestrates:

  – Creation of elements for new data (enter)
  – Updating of existing elements (update)
  – Removal of obsolete elements (exit)

  ```
  selection.data(data).join("tag")
  ```

  For example,

  ```
  svg.selectAll("circle")
      .data(data)
      .join("circle");
  ```

  For each data item, ensure there is exactly one <circle> element.

Internally:

- Enter → append <circle>
- Update → keep existing circles
- Exit → remove extra circles

After .join(), you typically chain attribute setters:

```
svg.selectAll("circle")
    .data(data)
    .join("circle")
    .attr("r", d => d);
```

The selector in selectAll() determines which existing elements participate in the join; the tag in .join() determines what new elements are created.

This affects both newly created and existing elements. This simplifies the classic no-join pattern

```
const update = svg.selectAll("circle") .data(data);

update.enter()
    .append("circle")
    .merge(update)
    .attr("r", d => d);

update.exit().remove();
```

For

```
svg.selectAll("rect")
    .data(data)
    .join("circle");
```

- **Update selection**: Existing <rect> elements are matched to data

  They remain <rect> elements (they are not converted)
- **Enter selection**: For new data items, D3 creates <circle> elements
- **Exit selection**: Extra <rect> elements (without data) are removed

- **.join with function parameters**: .join also has two optional parameters `update` and `exit`, specifying functions to call on update and exit. Note that the enter parameter can also be a function, and as is called for enter.

```
svg.selectAll("rect")
.data([127, 61, 256, 71])
.join(enter => enter.append("rect")
       .attr("x", 200)
       .attr("y", 200)
       .attr("width", 10)
       .attr("height", 10)
       .style("fill", "red")
       .call(update_bars),
     update => update.call(update_bars),
     exit => exit.attr("opacity", 1)
       .transition()
       .duration(3000)
       .attr("opacity", 0)
       .remove());
```

- **Remove**:

```
selection.remove()
```

  Removes the selected elements from the document. Returns this selection (the removed elements) which are now detached from the DOM. There is not currently a dedicated API to add removed elements back to the document; however, you can pass a function to selection.append or selection.insert to re-add elements.

- **Exit**: The .exit() selection represents DOM elements that no longer have corresponding data after a data join.

```
sel.exit()
```

  returns a selection of actual DOM nodes (unlike enter placeholders) that are now obsolete. These nodes still exist in the DOM until you remove them.

```
sel.exit().remove();
```

  Is most commonly used, removes all elements that no longer represent any data.

- **Transition**:

```
selection.transition(name?)
```

  Returns a new transition on the given selection with the specified name. If a name is not specified, null is used. The new transition is only exclusive with other transitions of the same name.

selection.transition() initiates an animated transition on the selected elements, allowing their attributes, styles, or properties to change smoothly over time rather than instantaneously.

```
d3.select("circle")
    .transition()
    .attr("r", 50);
```

Calling .transition():

1. Captures current state of each element
2. Schedules animation frames
3. Interpolates values over time
4. Applies intermediate values continuously

If nothing is specified,

- **Duration:** 250 ms (approximate default)
- **Delay:** 0
- **Easing:** cubic in-out

We have the functions

```
.duration(ms), .delay(ms), .ease(fn)
```

.ease() takes an ease function, for example d3.easeSin().

- **What can be animated**: Transitions interpolate numeric or color values. Common targets are
  - SVG attributes (cx, r, x, y, etc.)
  - CSS styles (opacity, fill, etc.)
  - Text content (with special handling)
  - Transforms
- **Chaining Transitions**: You can sequence animations

```
.transition()
.duration(500)
.attr("r", 20)
.transition()
.duration(500)
.attr("r", 5);
```

Second transition begins after the first completes.

```
function run15(svg) {
    var selection = svg.selectAll("rect")
        .data([127, 61,256,128])

    selection.enter().append("rect")
            .attr("x", 200)
            .attr("y", 200)
            .attr("width", 10)
            .attr("height", 10)
            .style("fill", "red")
        .merge(selection)
        .transition()
        .duration(3000)
            .attr("x", 0)
            .attr("y", (d,i) => i*10+50)
            .attr("width", d => d)
            .attr("height", 8)
            .style("fill", "steelblue")
            .transition()
            .duration(3000)
                .style("fill", "green")
                .attr("width", d => d*1.5)

    selection.exit()
        .attr("opacity", 1)
        .transition()
        .duration(3000)
            .attr("opacity", 0)
            .remove()
}
```

- **.call()**: .call() invokes a function on a selection (or transition) and passes that selection as the function's argument.

```
selection.call(function, ...args?)
```

```
selection.call(f);
```

  Is essentially the same as

```
f(selection)
```

  but .call() returns the original selection, enabling chaining.

- **Data as the key**: Consider

```
.data(data, d => d);
```

So, the key is simply the datum value itself. The key function computes a unique key for each datum. It is appropriate when your data consists of primitive unique values, such as numbers or strings.

Consider the initial data

```
[1,2,3]
```

Then, we update the data to

```
[3,2,1]
```

Without the key, D3 assumes

- Element 1 now represents 3
- Element 2 still represents 2
- Element 3 now represents 1

With $d \implies d$,

- Element with key 3 already exists → move/update
- Element with key 2 → unchanged
- Element with key 1 → move/update

- **Nested selections, grouping**:

```
function run19(svg) {
  var myData = [
    [15, 20],
    [40, 10],
    [30, 27]
  ]

  // First selection (within svg)
  var selA = svg.selectAll("g")
    .data(myData)
    .join("g")
    // .attr("y", (d,i) => i*100 + 50)
    .attr("transform", (d, i) => `translate(70,${i*100+50})`)
    // backticks (``) denote template literal
    // normal strings except that text inside ...
    // is evaluated (can be js expression)!

  // Second selection (within first selection)
  var selB = selA.selectAll('circle')
    .data(d => d)
    .join("circle")
    .attr("cx", (d, i) => i*80)
    .attr("r", (d,i) => d)
};
```

- **Scaling**: In D3, scaling refers to mapping values from a data domain to a visual range — typically pixel positions, lengths, colors, or sizes. Raw data values usually do not correspond directly to screen coordinates.

  Every D3 scale has two fundamental parts:

  1. **Domain**:

     ```
     0  [min data, max data]
     ```

  2. **Range**

     ```
     0  [min pixel, max pixel]
     ```

  For example,

  ```
  0  const x = d3.scaleLinear().domain([0,100]).range([0,500]);
  ```

  A scale is simply a function. Now, we can use it when defining x, y attributes.

  ```
  0  .attr("x", d => x(d));
  ```

- **scaleLog and scaleBand**: A log scale maps data using a logarithmic function, instead of using a linear mapping

  $$y = ax + b,$$

  it uses a logarithmic one

  $$y = a \log x + b.$$

  We use logarithmic scaling when data spans multiple orders of magnitude. A linear scale would compress small values and stretch large ones excessively.

  Note: Be careful with the domain,

  $$D(\log(x)) = (0, \infty).$$

  We can also define the base of the logarithm

  ```
  0  x.base(2);
  ```

  Log scales typically produce ticks at powers

  d3.scaleBand() maps discrete categories to evenly spaced rectangular bands across a continuous range. It is the standard tool for bar charts. Categorical data has no numeric spacing.

```
const x = d3.scaleBand().domain(["A", "B",
↪  "C"]).range([0,300]);
```

D3 divides the range into equal bands

$$
\begin{aligned}
A &\rightarrow 0 - 100 \\
B &\rightarrow 100 - 200 \\
C &\rightarrow 200 - 300.
\end{aligned}
$$

We get the bandwidth with `bandwidth()`

```
.attr("x", d=>x(d.category)).attr("width", x.bandwidth());
```

We can add spacing between bars with

```
x.padding(p)
```

- **d3.max(), d3.min()**: Retrieves the max or min value from an iterablea. Unlike Math.max() and Math.min(), these methods ignores undefined values, which is useful for missing data.
- **d3.maxIndex(), d3.minIndex()**: Retrieves the index of the max or min value from an iterable.
- **Axes**:
  - **axis(context)**: Render the axis to the given context, which may be either a selection of SVG containers (either SVG or G elements) or a corresponding transition.
  - **d3.axisTop(scale)**: Constructs a new top-oriented axis generator for the given scale, with empty tick arguments, a tick size of 6 and padding of 3. In this orientation, ticks are drawn above the horizontal domain path.
  - **d3.axisBottom(scale)**: Constructs a new bottom-oriented axis generator for the given scale, with empty tick arguments, a tick size of 6 and padding of 3. In this orientation, ticks are drawn below the horizontal domain path.
  - **d3.axisLeft(scale)**: Constructs a new left-oriented axis generator for the given scale, with empty tick arguments, a tick size of 6 and padding of 3. In this orientation, ticks are drawn to the left of the vertical domain path.
  - **d3.axisRight(scale)**: Constructs a new right-oriented axis generator for the given scale, with empty tick arguments, a tick size of 6 and padding of 3. In this orientation, ticks are drawn to the right of the vertical domain path.
  - **axis.scale(scale?)**: If scale is specified, sets the scale and returns the axis. If scale is not specified, returns the current scale.
  - **axis.ticks(args...)**: Sets the arguments that will be passed to scale.ticks and scale.tickFormat when the axis is rendered, and returns the axis generator. The meaning of the arguments depends on the axis' scale type: most commonly, the arguments are a suggested count for the number of ticks (or a time interval for time scales), and an optional format specifier to customize how the tick values are formatted.

This method has no effect if the scale does not implement scale.ticks, as with band and point scales. To set the tick values explicitly, use axis.tickValues. To set the tick format explicitly, use axis.tickFormat.

- **axis.offset(offset?)**: If offset is specified, sets the offset to the specified value in pixels and returns the axis. If offset is not specified, returns the current offset which defaults to 0 on devices with a devicePixelRatio greater than 1, and 0.5px otherwise. This default offset ensures crisp edges on low-resolution devices.
- **axis.tickSize(size?)**: If size is specified, sets the inner and outer tick size to the specified value and returns the axis. If size is not specified, returns the current inner tick size, which defaults to 6.
- **exis.tickSizeInner(size?)**:
- **axis.tickSizeOuter(size?)**:
- **axis.tickFormat(format?)**: If format is specified, sets the tick format function and returns the axis. If format is not specified, returns the current format function, which defaults to null. A null format indicates that the scale's default formatter should be used, which is generated by calling scale.tickFormat. In this case, the arguments specified by axis.tickArguments are likewise passed to scale.tickFormat.

  ```
  0  axis.tickFormat(d3.format(",.0f"));
  ```

  Formats integers with comma-grouping for thousands

  More commonly, a format specifier is passed to axis.ticks

  ```
  0  axis.ticks(10, ",f")
  ```

- **axis.tickValues(values?)**: If a values iterable is specified, the specified values are used for ticks rather than using the scale's automatic tick generator. If values is null, clears any previously-set explicit tick values and reverts back to the scale's tick generator. If values is not specified, returns the current tick values, which defaults to null.
- **axis.tickPadding(padding?)**: If padding is specified, sets the padding to the specified value in pixels and returns the axis. If padding is not specified, returns the current padding which defaults to 3 pixels.
- **axis.tickArguments([args])**: If arguments is specified, sets the arguments that will be passed to scale.ticks and scale.tickFormat when the axis is rendered, and returns the axis generator. The meaning of the arguments depends on the axis' scale type: most commonly, the arguments are a suggested count for the number of ticks (or a time interval for time scales), and an optional format specifier to customize how the tick values are formatted.

• **The return value of axisTop() and others**: The return value is the generator function that builds the axis.

  We can then call this generator passing in selectors so that an axis is built as a child of that selector.

• **Axis example**:

```
// the scale function
var xScale = d3.scaleLinear()
  .domain([0,d3.max(data)])
  .range([0,300]);

var xAxis = d3.axisTop()
xAxis.scale(xScale);
svg.append("g").call(xAxis);
```

Where

```
svg.append("g").call(xAxis);
```

means "Apply the axis generator to this <g> element, causing it to create ticks, labels, and lines inside it."

- **Adding text**:

```
selection.text(value?)
```

If a value is specified, sets the text content to the specified value on all selected elements, replacing any existing child elements. If the value is a constant, then all elements are given the same text content; otherwise, if the value is a function, it is evaluated for each selected element, in order, being passed the current datum (d), the current index (i), and the current group (nodes), with this as the current DOM element (nodes[i]). The function's return value is then used to set each element's text content. A null value will clear the content.

If a value is not specified, returns the text content for the first (non-null) element in the selection. This is generally useful only if you know the selection contains exactly one element.

- **Inner html**:

```
selection.html(value?)
```

If a value is specified, sets the inner HTML to the specified value on all selected elements, replacing any existing child elements. If the value is a constant, then all elements are given the same inner HTML; otherwise, if the value is a function, it is evaluated for each selected element, in order, being passed the current datum (d), the current index (i), and the current group (nodes), with this as the current DOM element (nodes[i]). The function's return value is then used to set each element's inner HTML. A null value will clear the content.

If a value is not specified, returns the inner HTML for the first (non-null) element in the selection. This is generally useful only if you know the selection contains exactly one element.

The what why and how

3.1 How

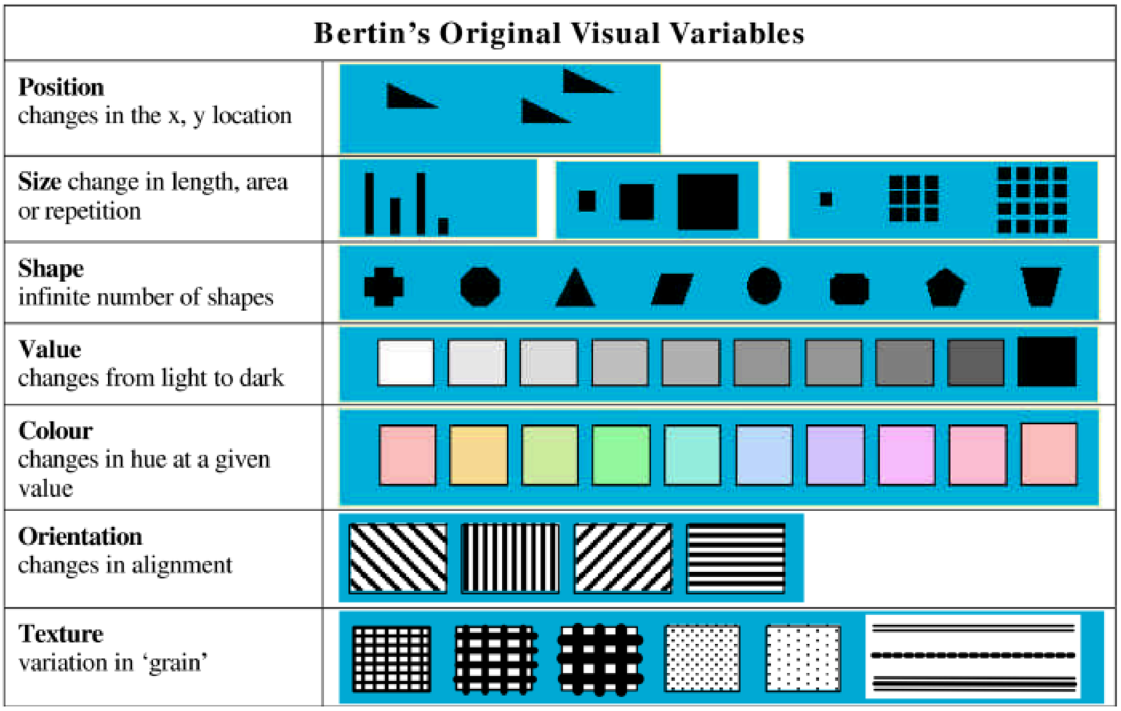
- **Marks and channels**:
  - Marks are the basic graphical elements in a visualization
  - Channels are ways to control the appearance of the marks

  Marks classified by dimensionality such as points, lines, area, etc. Also can surfaces, volumes.

  Usually map an attribute to a single channel, could use multiple channels but limited number of channels

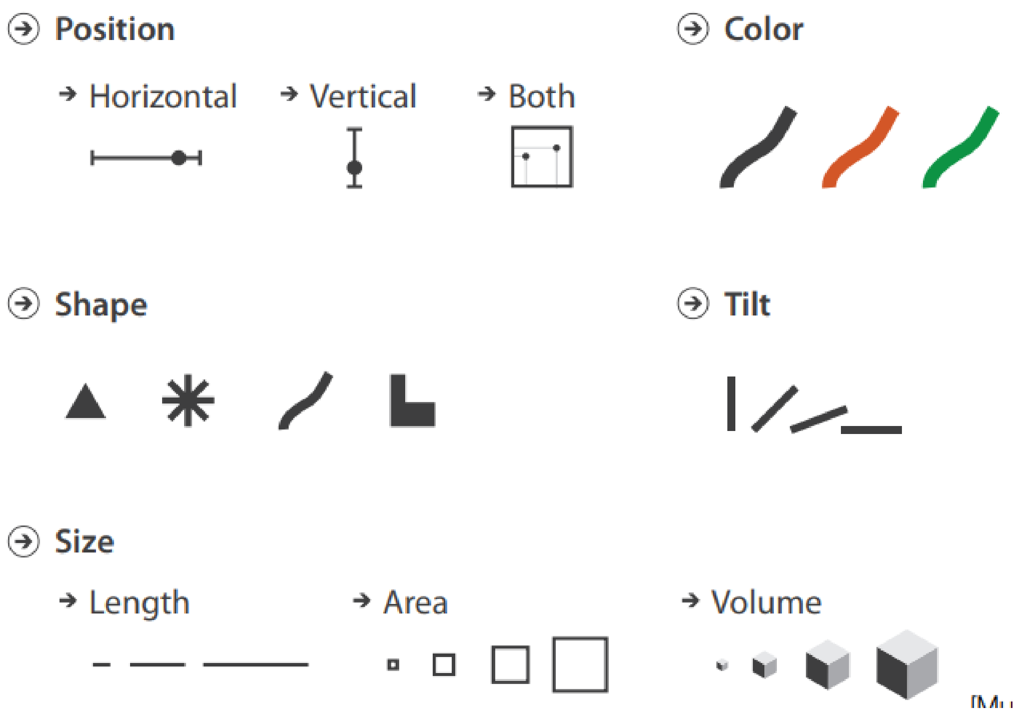
  Restrictions on size and shape
  - Points are nothing but location so size and shape are ok
  - Lines have a length, cannot easily encode attribute as length
  - Maps with boundaries have area, changing size can be problematic

- **Bertin - Visual Variables**:



Bertin's Original Visual Variables
Position changes in the x, y location
Size change in length, area or repetition
Shape infinite number of shapes
Value changes from light to dark
Colour changes in hue at a given value
Orientation changes in alignment
Texture variation in 'grain'

- **More visual channels**:


Position
Horizontal
Vertical
Both
Color
Shape
Tilt
Size
Length
Area
Volume
[Mu