Data Visualization

Nathan Warner

Northern Illinois
University

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.

    M x y

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

    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.**

    Z
  - **Quadratic Curve (Q)**: (cx, cy) = control point (x, y) = end point

    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)**:

    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;
```

- **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(...)**:

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

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

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

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

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

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

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

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

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

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

```
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