

Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 5

Introduction to D3.js

Contents

- The basics of HTML, CSS, and the Document Object Model (DOM)
- The principles of Scalable Vector Graphics (SVG)
- Node and ES2015 functionality
- Data-binding and selections with D3
- Different data types and their data visualization methods

What is D3.js?

- D3 stands for data-driven documents
- Use D3 if you want to develop web pages that needs to be:
 - Fast
 - Interactive
 - Animated
 - Shareable
- If you really want to have a simple visualization like pie chart layout, probably you can use other charting libraries
- But if you want to change the pie chart as the data changes then d3.js is useful

How D3 works

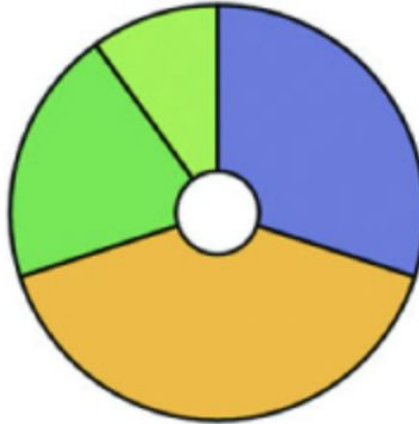
- It has several components
 - Data management
 - Data generation
 - Loading
 - Binding dataset
 - Processing
 - Charts
 - Basic charting
 - Design
 - Hierarchical layouts
 - Networks
 - Maps

How D3 works

- Interactivity
 - Mouse events
 - Brush filtering
 - Zoom
- MVC dashboard
- Optimization

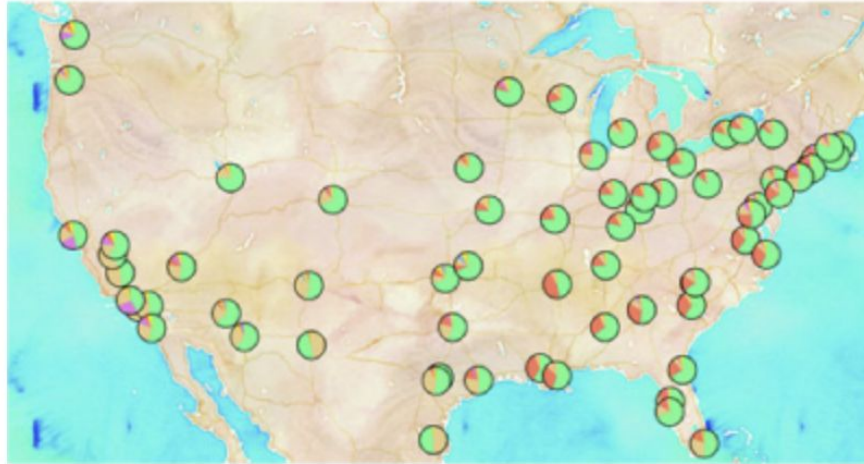
Data visualization is more than charts

- D3 can be used for simple charts, such as this donut chart



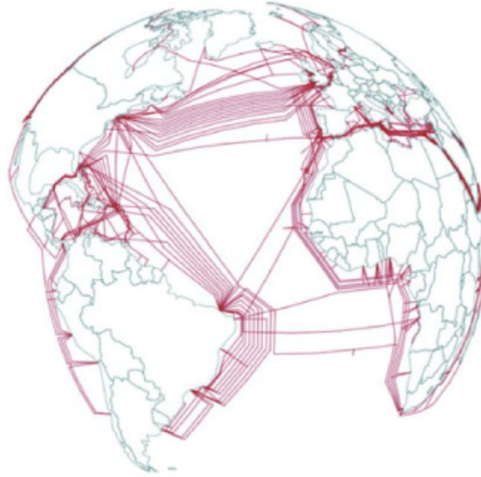
Data visualization is more than charts

- D3 can also be used to create web maps, such as this map showing the ethnic makeup of major metropolitan areas in the United States.



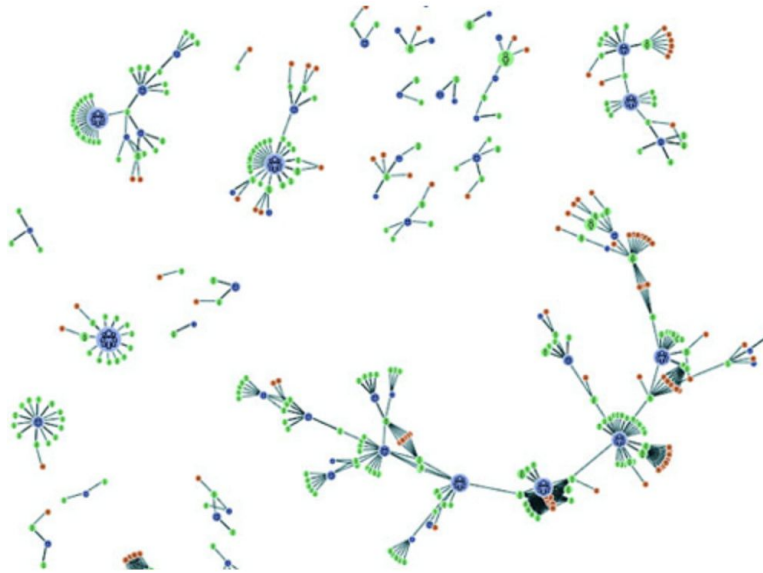
Data visualization is more than charts

- Maps in D3 aren't limited to traditional Mercator web maps. They can be interactive globes, like this map of undersea communication cables, or other, more



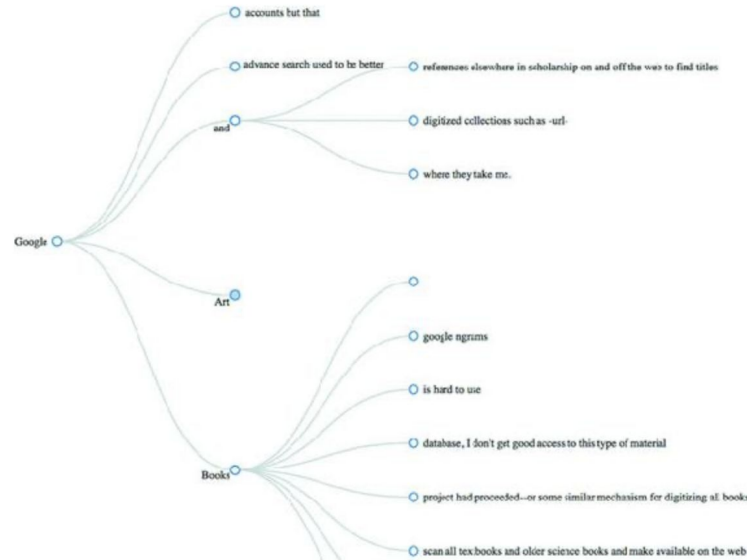
Data visualization is more than charts

- D3 also provides robust capacities to create interactive network visualizations



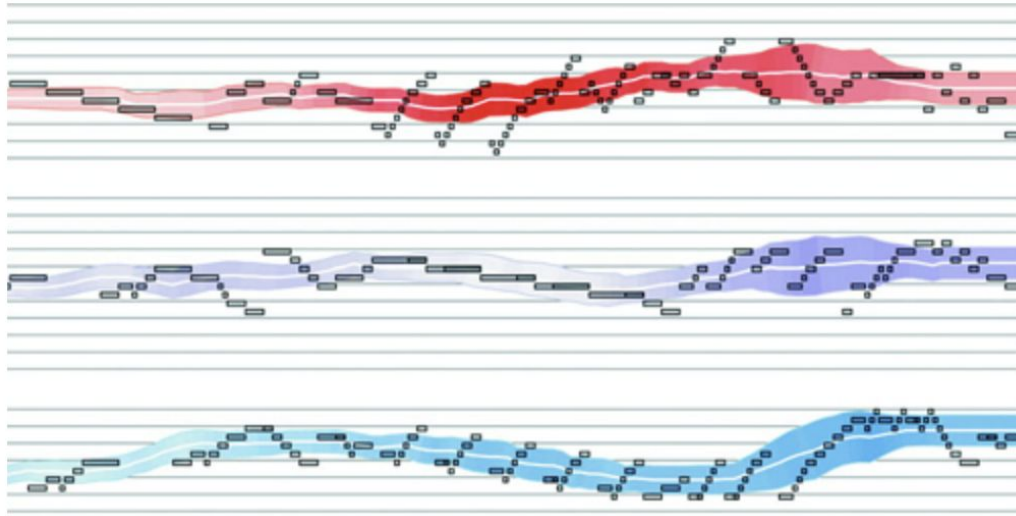
Data visualization is more than charts

- D3 includes a library of common data visualization layouts, such as the dendrogram



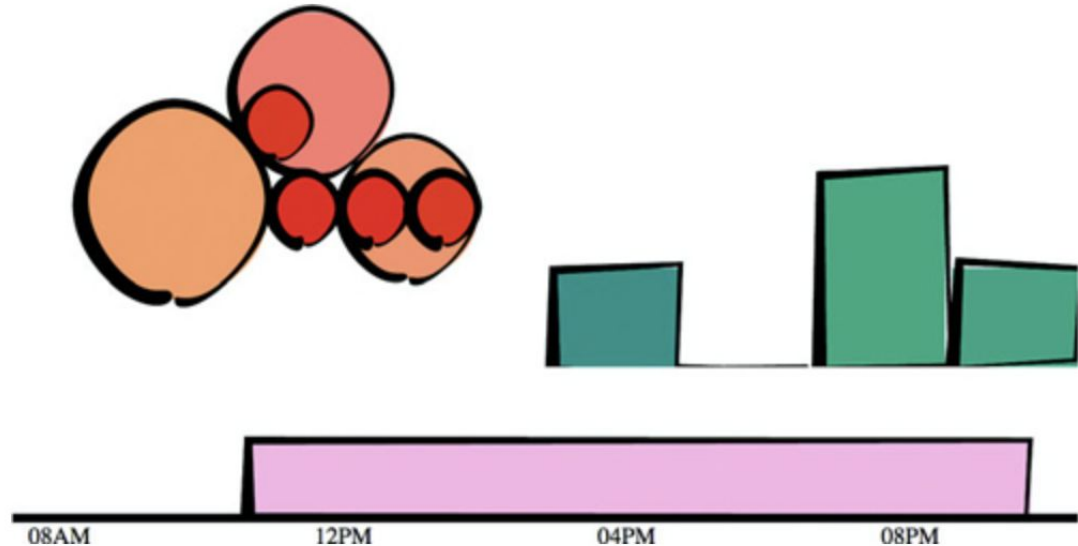
Data visualization is more than charts

- D3 has SVG and canvas drawing functions so you can create your own custom visualizations, such as this representation of musical scores.



Data visualization is more than charts

You can combine these layouts and functions to create a data dashboard. You can also use the drawing functions to make your bar charts look distinctive, such as this “sketchy” style.



D3.js

- The fundamental principles for all these visualizations are same!
- Not only the method is same, the data could be also same
- The only difference is the representation
- We will now go through some of the fundamental concepts

D3 is about selecting and binding

- Concentrate on two things
 - The selection of elements in your web page
 - The data that you want to bind with the elements
- Here's a selection without any data:

```
d3.selectAll("circle.a").style("fill", "red").attr("cx", 100);
```

- Likewise, this code turns every div on our web page red and changes its class to b:

```
d3.selectAll("div").style("background", "red").attr("class", "b");
```

D3 is about selecting and binding

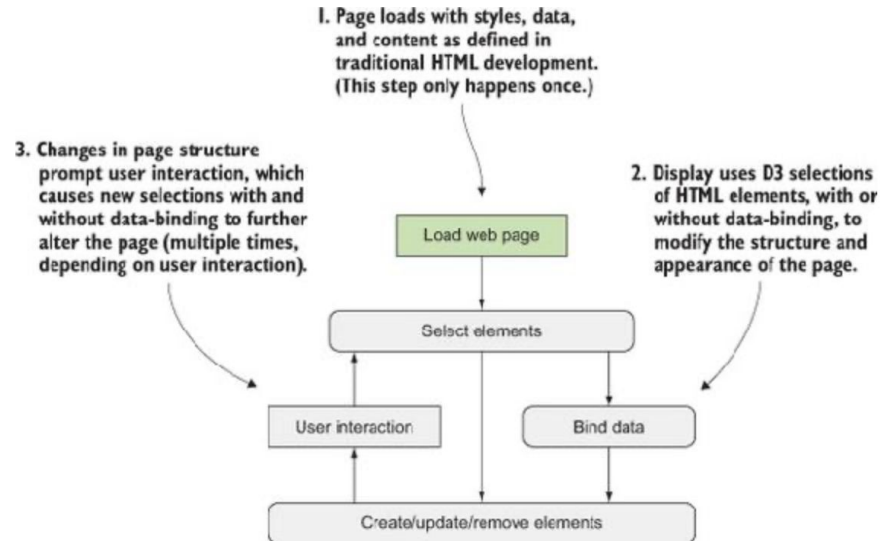
- So far we have seen selecting element, let's now bind some data:

```
d3.selectAll("div.market").data([1,5,11,3])
```

- What if the number of data is more or less than the number of elements?
- There are nice ways to tackle these situations
- We will see in future lectures

D3 is about selecting and binding

An application created with D3 can use selections and data binding over and over again, together and separately, to update the content of the data visualization based on interaction.



Web page elements can now be divs, countries, and flowcharts

- The fundamental principle is to selecting and binding data
- You may need to do it continuously because of user interaction or due to other events
- The elements will change, which will create different types of visualizations

The power of HTML5

- 20-30 years ago the only type of animation in web was GIF
- GIF is one type of animation but user do not have any control
- Modern HTML has different feature to control the contents:
 - The DOM
 - SVG
 - Canvas
 - CSS
 - Javascript
 - ES2015 and Node

The DOM

```
<!doctype html>
<html>
<head>
  <script src="d3.v4.min.js"></script>
</head>
<body>
  <div id="someDiv" style="width:200px;height:100px;border:black"
<input id="someCheckbox" type="checkbox" />
  </div>
</body>
</html>
```

The DOM

- From the HTML code the browser creates a tree
- For example the <html> is the root of the tree
- The <head> and <body> are children of the root
-
- Three categories of information about each element determine its behavior and appearance: styles, attributes, and properties.

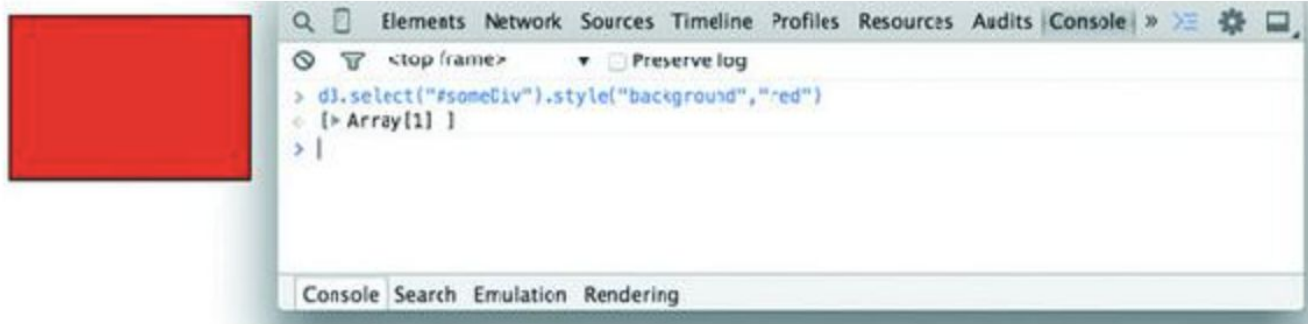
```
d3.select("#someDiv").style("border", "5px darkgray dashed");  
d3.select("#someDiv").attr("id", "newID");  
d3.select("#someCheckbox").property("checked", true);
```

The DOM

- One technical point: from the CSS lecture, we remember that there is z-index that determines the order of overlapping element
- In SVG version 1 this feature is not available
- So we want to append the elements carefully

Examining the DOM in the console

You can run JavaScript code in the console and call global variables or declare new ones as necessary. Any code you write in the console and changes made to the web page are lost as soon as you reload the page.

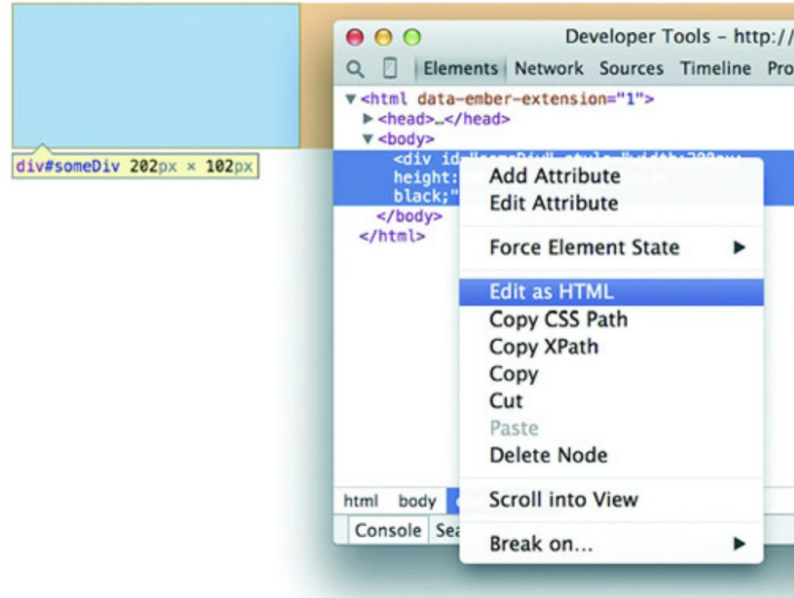


Examining the DOM in the console

- You can put breakpoints, which really helps for debugging
- You can modify elements in console
- You can delete elements
- But the elements can be reloaded
- You can manually change different elements

Examining the DOM in the console

Rather than adding or modifying individual styles and attributes, you can rewrite the HTML code as you would in a text editor. As with any changes, these only last until you reload the page.



Examining the DOM in the console

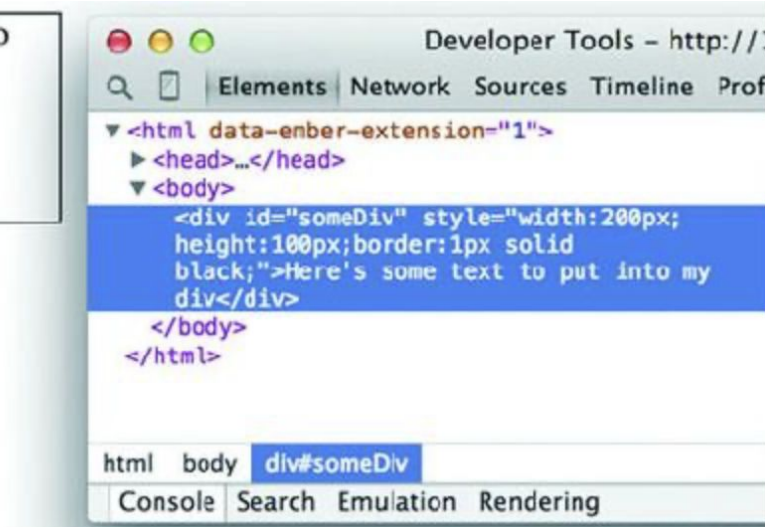
Changing the content of a DOM element is as simple as adding text between the opening and ending brackets of the element.

```
▶ <head>...</head>
▼ <body>
  <div id="someDiv"
    style="width:200px;height:100px;border
      :1px solid black;">Here's some text to
    put into my div</div>
  </body>
</html>
```

Examining the DOM in the console

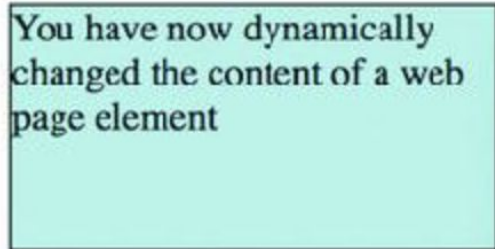
The page is updated as soon as you finish making your changes. Writing HTML manually in this way is only useful for planning how you might want to dynamically update the content.

Here's some text to put into my div



Coding in the console

```
d3.select("div").style("background", "lightblue").style("border", "solid black 1px").html("You have now dynamically changed the content of a web page element");
```

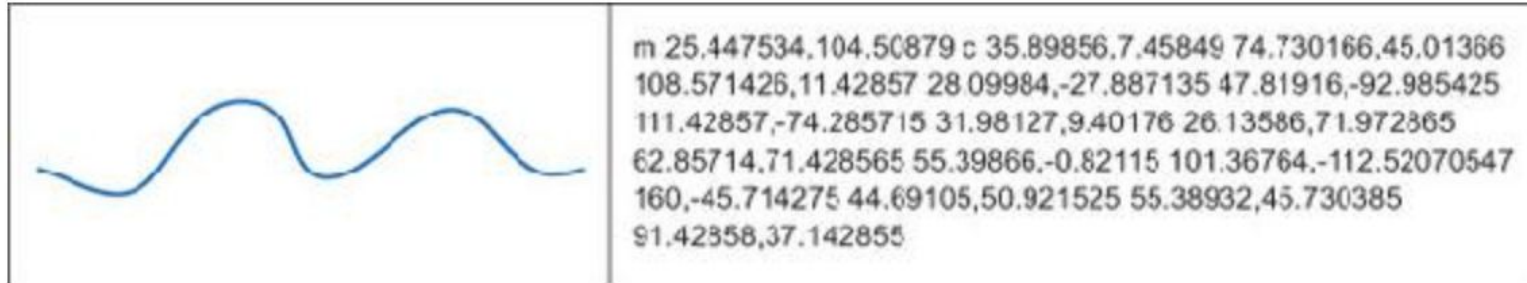


You have now dynamically changed the content of a web page element



SVG

The commands to draw an SVG path (right) and the resulting graphic (left)



SVG

A sample web page with SVG elements

```
<!doctype html>
<html>
  <script src="d3.v4.min.js">
</script>
<body>
  <div id="infovizDiv">
    <svg style="width:500px;height:500px;border:1px lightgray s
      <path d="M 10,60 40,30 50,50 60,30 70,80"
        style="fill:black;stroke:gray;stroke-width:4px;" />
      <polygon style="fill:gray;"
        points="80,400 120,400 160,440 120,480 60,460" />
    <g>
      <line x1="200" y1="100" x2="450" y2="225"
        style="stroke:black;stroke-width:2px;" />
      <circle cy="100" cx="200" r="30" />
      <rect x="410" y="200" width="100" height="50"
        style="fill:pink;stroke:black;stroke-width:1px;" />
    </g>
  </svg>
</div>
</body>
</html>
```

SVG

```
d3.select("circle").remove() 1  
d3.select("rect").style("fill", "purple") 2
```

- **1 Deletes the circle**
- **2 Changes the rectangle color to purple**

SVG

```
d3.select("svg").style("background", "darkgray"); 1
```

- ***1* Infoviz is always cooler on a dark background**

Canvas

- Creating static images
- Large amount of data
- WebGL for 3d

Geometric primitives

- Circle
- Rect
- Line
- Ellipse
- Polygon

Geometric primitives

Modifying the height and width attributes of a `<rect>` element changes the appearance of that element. Inspecting the element also shows how the stroke adds to the computed size of the element.



Text in SVG

- doesn't have the formatting support found in HTML elements
- primarily used for labels
- If you do want to do basic formatting, you can nest `<tspan>` elements in `<text>`

Grouping element

- Use the <g> tag for grouping, like a node and a label
- Does not has any graphical representation
- It's a logical grouping
- Is used extensively
- You can move a group together

```
<g>  
  <circle r="2"/>  
  <text>This circle's Label</text>  
</g>
```

Grouping

All SVG elements can be affected by the transform attribute, but this is particularly salient when working with `<g>` elements, which require this approach to adjust their position. The child elements are drawn by using the position of their parent `<g>` as their relative 0,0 position. The `scale()` setting in the transform attribute then affects the scale of any of the size and position attributes of the child elements.

No transform:

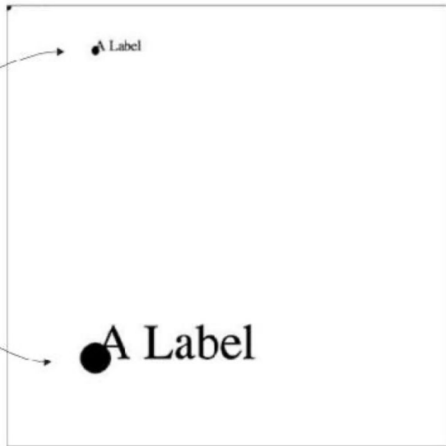
Like all SVG elements, the `<g>` is by default drawn at the 0,0 position. The result in this case is that the graphics are drawn such that you only see a tiny piece of the bottom right corner of the circle, and none of the label text.

Translate(100,50):

The entire `<g>` element is moved 100 pixels to the right (along the x-axis) and 50 pixels down (along the y-axis). The child elements are drawn from this position.

Translate(100,400)Scale(3.5):

The `<g>` element is moved to XY position 100,400, lining it up along the y-axis with the above `<g>`, and the child elements are drawn from there at 3.5 times their size.



```
<g>
  <circle r="2" />
  <text>This circle's Label</text>
</g>

<g transform="translate(100,50)">
  <circle r="2" />
  <text>This circle's Label</text>
</g>

<g transform="translate(100,400) scale(2.5)">
  <circle r="2" />
  <text>This circle's Label</text>
</g>
```


Path

Open – unfilled:

Path elements are by default filled with no stroke. You need to set the fill style to “none” and stroke and stroke-width style if you want to draw it as a line.



```
<path style="fill:none;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80"
  transform="translate(0,0)" />
```

Open – filled:

An open path can be filled just like a closed path, with the fill area defined by the same area that would be bounded if the path were closed.



```
<path style="fill:black;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80"
  transform="translate(0,100)" />
```

Closed – unfilled:

A path will always close by drawing a line from the end point to the start point.



```
<path style="fill:none;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80Z"
  transform="translate(0,200)" />
```

Closed – filled:

Notice the stroke overlaps the fill area slightly.



```
<path style="fill:black;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80Z"
  transform="translate(0,300)" />
```

```
<path style="fill:none;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80"
  transform="translate(0,0)" />
<path style="fill:black;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80"
  transform="translate(0,100)" />
<path style="fill:none;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80Z"
  transform="translate(0,200)" />
<path style="fill:black;stroke:gray;stroke-width:4px;"
  d="M 10,60 40,30 50,50 60,30 70,80Z"
  transform="translate(0,300)" />
```