CMPT 165, Spring 2020

JavaScript and Graphics

jQuery $

- When jQuery is loaded, the variable $ is created and is a synonym for jQuery.
- That is, these are exactly equivalent:

```
jQuery('p').click(say_hello)
$('p').click(say_hello)

jQuery('#changeme').html('Somebody clicked me.')
$('#changeme').html('Somebody clicked me.')
```

jQuery Styles

- We have seen how to change an element's CSS by modifying its class:

```
$('#element').attr('class', 'highlight')
```

- jQuery can also be used to set CSS properties directory. e.g. I want to change the colour and font size of an element:

```
newstyle = {
    'color': '#f60',
    'font-size': '1.5em'
}
$('#element').css(newstyle)
```

- Or equivalently:

```
$('#element').css({
    'color': '#f60',
    'font-size': '1.5em'
})
```

jQuery Styles

```
newstyle = {
    'color': '#f60',
    'font-size': '1.5em'
}
$('#element').css(newstyle)
```

- In the first implementation, newstyle is a variable containing a JavaScript object. Like we said before: an object is a value containing other stuff.
- In this case, newstyle contains something called color which is a variable holding the string '#f60'.

jQuery Styles

```
newstyle = {
    'color': '#f60',
    'font-size': '1.5em'
}
```

- To specify style information for jQuery, we create an object like this: it contains variables named after CSS properties, which hold the CSS values.

- The syntax is similar to CSS, but not the same: be careful. Here, both the property and value are JavaScript strings, and commas separate the pairs.

jQuery Animations

- jQuery can animate any numeric CSS property. e.g. we can animate font-size from 10pt to 20pt, but not font-weight from normal to bold.
- The animation is always from the current value to the new value(s) you give.
- e.g. animate the font size and left margin to new values:

```
newstyle = {
    'font-size': '2.5em',
    'margin-left': '2em'
}
$('#element').animate(newstyle, 2000)
```

- This will transition from old margin & font-size value to the ones given in 2 seconds (2000 milliseconds).

jQuery Animations

- We can use CSS animations to make elements appear/disappear in interesting ways.

animate(*{styles},speed,easing,callback*)

```
newstyle = {
    'opacity': '0',
    'font-size': '0pt',
    'margin-left': '100%'
}
$('#element').animate(newstyle, 4000)
```

opacity property

- … and a million other combinations of user events, styles, and animations.

JavaScript + SVG: Raphaël

- We know a little JavaScript, and a little about vector graphics. Let's put the two together.
- SVG: a vector graphics format. It contains circles, lines, text, etc. It contains them very much like web pages contain headings, paragraphs, etc.
- jQuery lets us manipulate headings, paragraphs, etc. within HTML pages.
- Raphaël is a JavaScript library that lets us manipulate SVG images within our pages in a similar way.
- First: we need the Raphaël JavaScript library loaded as part of the page:

```
<script src="https://cmpt165.csil.sfu.ca/js/jquery-3.4.1.js"></script>
<script src="https://cmpt165.csil.sfu.ca/js/raphael-2.3.0.js"></script>
<script src="raphael-code.js"></script>
```

JavaScript + SVG: Raphaël

- We will create the SVG entirely in JavaScript, inside an (initially-empty) element on the page (not load an existing SVG image and modify it).
  -
- Now we can actually create an image. In our setup() function:
  - paper = Raphael('container', 200, 100)
- This creates an SVG image in the that is 200 pixels wide and 100 high.

JavaScript + SVG: Raphaël

- This will draw a circle on the paper (which is what Raphaël calls the SVG image you draw on):
  - circ = paper.circle(60, 40, 20)
  - with centre at x = 60 and y = 40, with radius 20 (pixels).
- For Raphaël, x = 0, y = 0 is the top-left.
- We can modify the appearance of SVG elements almost like we did with HTML elements, using the .attr() function:

```
circ.attr({
    'fill': '#f00',
    'stroke': '#000',
    'stroke-width': '2'
})
```

- This sets the fill color, and line (stroke) colour and width for that circle.

JavaScript + SVG: Raphaël

- SVG images contain elements (like circles). Those elements have CSS-like style properties (like fill) that can be set on them to change their appearance.

- Raphaël is giving us a way to create (and modify) these things so we can create an image with code.

About SVG

- I said before: SVG contains circles in the same way HTML contains paragraphs. It's actually a very close analogy.
- SVG files are actually text files, containing code very much like HTML.
- For example, this is an (almost perfectly correct) SVG file:

```
<svg xmlns="http://www.w3.org/2000/svg"
    width="200" height="100">
<circle fill="#0f0" r="10" cy="20" cx="50" />
</svg>
```

- It looks a lot like HTML: tags, attributes, etc. It's an XML document: like HTML, but without HTML's rules for which tags exist (but in this case, with the SVG rules about which tags exist)

About SVG

- When creating SVG with Raphaël, the <svg> element is inserted inline into the page.
- You can explore it with the development tools: if you find the that was originally empty, it will now contain an <svg> element.
- Raphaël is giving us a convenient way to create these elements.
- … like jQuery gave us a convenient way to explore/modify the HTML.
- The .attr() function on SVG elements is literally setting attributes on the tags. e.g.

```
r = paper.rect(10, 20, 30, 40)
r.attr({
    'fill': '#f00',
    'stroke': '#000',
})
```

- … creates this in the SVG on our page:
  - <rect x="10" y="20" width="30" height="40“
  - fill="#f00" stroke="#000" />

About SVG

- You may see descriptions of how to do things in SVG phrased with SVG code examples. It's usually straightforward to translate that to Raphaël, if you know the basic ideas.
- Sometimes Raphaël makes things easier than the obvious translation though. e.g.

```
r = paper.rect(10, 20, 30, 40)
r.rotate(30)
```

- … creates:
  - <rect x="10" y="20" width="30" height="40“
  - fill="none" stroke="#000“
  - transform="matrix(0.866,0.5,-0.5,0.866,23.3494,-7.141)" />
- And we didn't have to calculate that transformation matrix.

About SVG

- So far, we have seen two things we can draw with Raphaël: a circle by specifying the x and y position of the centre, and the radius; and a rectangle by specifying the x and y of the top-left corner, and the width and height.

```
c = paper.circle(100, 60, 16)
r = paper.rect(110, 80, 15, 19)
```

Interacting with SVG

- We can use all the things we have learned before, and let the user interact with the images we create.
- In the above examples, we were imagining just putting Raphaël code in the setup function, but it could go anywhere JavaScript code runs.
- For example, have a button that, when clicked, adds something to the image. (Complete .js file.)

```
radius = 50
more = function() {
  circ = paper.circle(50, 50, radius)
  radius = radius - 4
}
$( function() {
  paper = Raphael('container', 100, 100)
 $('#more').click(more)
})
```

Interacting with SVG

- We can modify an SVG image with Raphaël as the result of any JavaScript event we like: page load, click, mouse-over, etc.

Paths: Lines

- Something we haven't drawn yet: lines. They are a little tricky, because there's a lot of flexibility.
- The paper.path function is used to draw a path. Its argument is an SVG path string, which is a very compact way to say a lot about the line to draw.
- For example, draw a line from (20, 40) to (50, 80):

```
paper = Raphael('container', 400, 400)
p = paper.path('M20,40 L50,80')
p.attr({
    'stroke-width': '4',
    'stroke': 'red'
})
```

- That is, move to (20, 40), and draw a line to (50, 80). Then we can change the line colour and thickness of the path (or do any other attribute changes we want).

Paths: Lines

- p = paper.path('M20,40 L50,80')
- The path string consists of a series of commands (and points to go with them).
  - M: move to the given coordinates.
  - L: draw a straight line to the given coordinates.
- Both of these commands are given one x, y point as a destination.
- We can draw a bunch of line segments attached together:
  - p = paper.path('M20,40 L50,80 L120,60 L50,60')

Paths: Lines

- We can create a closed path, one that finishes at the same place it starts, with the Z command:
  - triangle = paper.path('M10,10 L10,50 L50,50 Z')
- i.e. a triangle with corners (10, 10), (10, 50), (50, 50).
- The Z doesn't need a point given: it goes pack to wherever this path started.

Paths: Curves

- The same path() function can also be used to draw curves. We just need to specify the right details in the path string.
- The T command can be used to draw curves through a point (or series of points). e.g. curve from (10, 10) to (50, 50) to (100, 20):

```
curve1 = paper.path('M10,10 T50,50 T100,20')
```

- It's probably more clear if we also draw those points and have a look:

```
curve1 = paper.path('M10,10 T50,50 T100,20')
p1 = paper.circle(10, 10, 3).attr({'fill': 'red'})
p2 = paper.circle(50, 50, 3).attr({'fill': 'green'})
p3 = paper.circle(100, 20, 3).attr({'fill': 'blue'})
```

Paths: Curves

- I find T hard to work with: difficult to get a sense for the direction the curve is going to go.
- The Q command needs two arguments: a control point and the destination. For example,

```
curve2 = paper.path('M10,10 Q50,50 100,20')
```

- Here, the curve starts at (10, 10) and goes to (100, 20) with control point (50, 50).

Paths: Curves

- The control point determines the direction the curve starts/ends. Again, we can draw all of that:

```
curve2 = paper.path('M10,10 Q50,50 100,20')
curve2.attr({'stroke-width': '2', 'stroke': '#f00'})
p1 = paper.circle(10, 10, 3).attr({'fill': '#0f0'})
p2 = paper.circle(50, 50, 3).attr({'fill': '#00f'})
p3 = paper.circle(100, 20, 3).attr({'fill': '#0f0'})
l1 = paper.path('M10,10 L50,50 L100,20')
```

Paths: Curves

- The farther the control point is away from the start and end points, the sharper the curve will be.

Wikipedia: Bézier curve

- This is a quadratic Bézier curve, in case you're interested in the math, or why it's abbreviated Q.

Paths: Curves

- The lines and curves can be combined in any way you want.

```
curve3 = paper.path(
    'M110,10 Q80,75 60,40 L60,20 Q60,0 20,60'
)
```

- As-written, and with dots added on the points used
- There are a few other path commands, including.
  - A: elliptical/circular arc
  - C: cubic Bézier curve
- … but I generally only use L, Q, and Z when creating paths by hand.

Paths: Curves

- The SVG path strings are case-sensitive. The lowercase version of each command lets you give a position relative to where you started: L50,50 draws a line to the point (50, 50), but l50,50 draws a line 50 units right and 50 units down.

- You don't need to worry about them, but it could be a cause of problems when working with paths.

Animation

- We can animate SVG elements using Raphaël almost exactly like we did with HTML elements and jQuery.
- With jQuery, we had the .css() function that could set CSS properties, and .animate() took the same kind of style info and transitioned to the new appearance over some time.
- With Raphaël, we can set element appearance with .attr(), and .animate() takes an argument in the same format, animating to the new look.
- For example, we can create a rectangle, and then (immediately) start an animation to a different size:

```
rect1 = paper.rect(10, 10, 50, 80)
new_size = {
    'width': '80',
    'height': '50'
}
rect1.animate(new_size, 1000)
```

Animation

- Or we can apply a transformation to move/rotate/scale:

```
original = {
    'transform': 'r0'
}
turned = {
    'transform': 'r360'
}
rect2.attr(original)
rect2.animate(turned, 2000)
```

- See the Raphaël reference, but r for rotation, t for translation (move), s for scaling.

Animation

- There are two more options we can give .animate():
  - An easing type: should the movement be linear, accelerating, bouncy, etc. Default is 'linear'.
  - A callback function: something to do when the animation is finished.
- You probably don't need the easing option, but if we want to give the callback, it needs to be the fourth argument, so setting 'linear' might be necessary to get there.

Animation

- We can use this to create an animation…

```
rect3 = paper.rect(100, 20, 50, 80)
slide = {
    'transform': 't50,50'
}
rect3.animate(slide, 1000, 'linear', recolour)
```

- … and do something else when it completes.

```
recolour = function() {
    blush = {
        'fill': '#f99'
    }
    rect3.animate(blush, 1000)
}
```

Creative Commons

Animation

- We can also use the callback function to create a repeating animation.
- The idea: after one animation, call a function that starts another, then another, and another, …
- In the setup, we'll create the image, and call the function that starts the animation.

```
setup = function() {
  paper = Raphael('container', 200, 200)
  c = paper.circle(100, 100, 40)
  grow()
}
$(document).ready(setup)
```

Animation

- … and definte functions to animate (scaling 1x and 2x):

```
grow = function() {
  bigger = {
    'transform': 's2'
  }
  c.animate(bigger, 1000, 'linear', shrink)
}
shrink = function() {
  smaller = {
    'transform': 's1'
  }
  c.animate(smaller, 1000, 'linear', grow)
}
```

Animation

- What's happening here:
  1. When the page is ready, draw and call grow() to start the animation.
  2. When grow runs, animate to a larger circle, and start shrink after it.
  3. When shrink runs, animate to a smaller circle, and start grow after it. (i.e. back to #2.)

SVG Events

- We have used events on HTML elements:
  - jQuery('#changing').click(change_it)
- i.e. when the HTML element with id="changing" is clicked, call change_it().
- SVG elements have similar events that we can connect functions to…

SVG Events

- Let's start by drawing some shapes (in the setup function):

```
paper = Raphael('event-container', 400, 400)
tri = paper.path("M10,53 L60,53 L35,10 Z")
tri.attr({ 'fill': '#b00' })
square = paper.rect(70, 40, 40, 40)
square.attr({ 'fill': '#00a' })
```

- Then we can attach to the click (or double-click or mouse-over or other) events just like in HTML + jQuery (except we don't have to select them because we already have a variable referring to them):

```
tri.click(tri_click)
square.hover(square_mouse)
```

SVG Events

- The functions can do anything any other JavaScript can do.

```
tri_click = function() {
    tri.attr({
        'fill': '#faa'
    })
}
square_mouse = function() {
    tri.animate({
        'transform': 's0'
    }, 1000)
}
```

SVG Events

- Summary of what we can do with JavaScript code:
  - Create/modify HTML elements.
  - Animate numeric CSS style changes.
  - Attach logic to events on HTML elements.
  - Create/modify SVG elements.
  - Animate SVG attribute changes.
  - Attach logic to events on SVG elements.

In Class Activity

- Download the code posted on 12/Feb/2020 or see the one presented on the screen.
- We need to change the color of the circle. When it is small the color is orange and when it big the color is red.
- We need the circle to disappear when some click on it and when you click at any part of the svg area it will show up again.

- You need to write the steps to do this modification. You can refer to the code by line number or by function name.