Graphics with SVG

Overview

In this lab, you'll see how to create SVG vector graphics dynamically. You'll create elements such as `<rect>`, `<circle>`, and `<polygon>` by using the HTML Document Object Model (DOM) API, and add them to an `<svg>` element on the web page.

You'll also see how to define styles, handle events, and transform SVG content dynamically using a combination of traditional JavaScript code and jQuery.

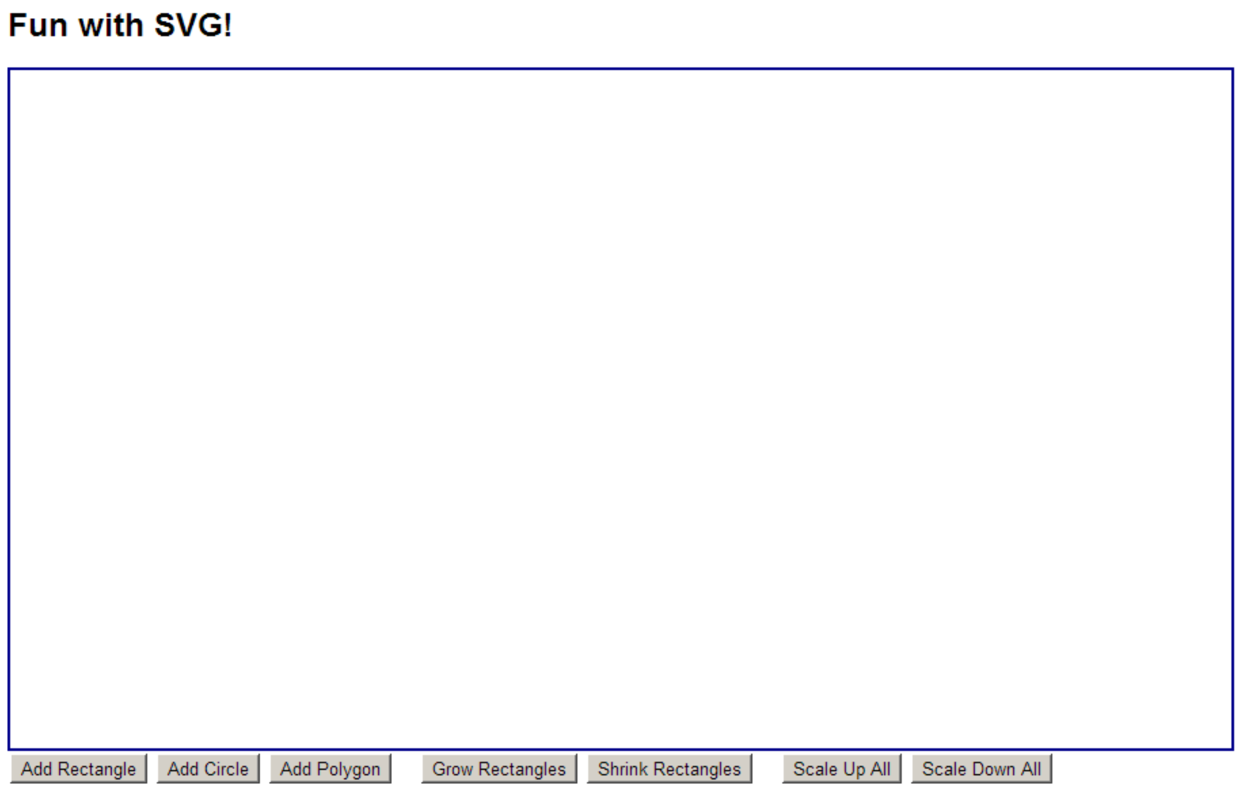
Roadmap

There are 6 exercises in this lab, of which the two last exercises are "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Adding `<rect>` elements by using DOM
2. Adding `<circle>` elements by using DOM
3. Defining styles and handling events
4. Locating elements in the DOM tree
5. Transforming SVG content (if time permits)
6. Adding `<polygon>` elements by using DOM (if time permits)

Familiarization

Open a browser and browse to `FunWithSvg.html` in the *Solutions* folder. The Web page appears as follows:



The Web page has a large `<svg>` element (enclosed in a border), with various buttons beneath it that allow the user to create and manipulate content in the `<svg>` element:

- When you click *Add Rectangle*, *Add Circle*, or *Add Polygon*, it creates a new `<rect>`, `<circle>`, or `<polygon>` element and adds it as a child of the `<svg>` element in the DOM tree. Each shape has a random location, size, colour, and opacity. Also, the `<polygon>` elements have a random number of points.
- When you click *Grow Rectangles* or *Shrink Rectangles*, it doubles or halves the size of all the `<rect>` elements in the web page. This illustrates how you can locate specific SVG elements in the DOM tree.
- When you click *Scale All Up* or *Scale All Down*, it transforms all child elements inside the `<svg>` element by applying a scaling factor of `2.0` or `0.5` respectively.

Also notice the following general features:

- When you hover over a shape in the `<svg>` element, it sets the `fill-opacity` of the shape to `1.0`. When you move the mouse away, the `fill-opacity` returns to its original value. This illustrates how to define CSS styles for SVG elements.
- When you mouse-click any shape in the `<svg>` element, it adds a thick red border around the shape. If you click the shape again, it removes the border. This illustrates how you can handle events on SVG elements.
- If you hold the `SHIFT` and `CTRL` buttons down while you mouse-click a shape in the `<svg>` element, it removes the element from the DOM tree. This illustrates that you have complete control over the DOM tree in SVG.

Now switch to the *Start* folder and open `FunWithSvg.html` in the text editor. We've already written the HTML mark-up and some of the JavaScript code. There are several TODO comments indicating where you will add your code in each exercise. Note the following points:

- At the bottom of the page, we've already defined an `<svg>` element, plus all the buttons. Each button is associated with a click-handler function – all of these functions exist, but their implementations are mostly empty at the moment.
- At the top the top of the page, we’ve defined a CSS style for `<svg>` elements.
- We've already imported the jQuery script file. You'll use various jQuery APIs in this lab, to simplify manipulation of the DOM tree.
- In our own `<script>` tag, we've defined a few global variables:
    - `svgNS`
      Contains the XML namespace for SVG elements. When you create SVG elements (e.g. `<rect>`), the elements must have this XML namespace.
    - `svgPanelWidth` and `svgPanelHeight`
      These variables are assigned in the `init()` function, to hold the width and height of the `<svg>` element. These variables are used when it comes to generating random locations and sizes for SVG elements.
- Our `<script>` tag also contains various stubbed-out functions, plus two helper functions near the bottom that are already complete:
    - `getRandomAttributes()`
      Returns an object containing random values for a shape's location, size, colour, and opacity.
    - `getRandomInt()`
      Returns a random integer in a specified range.

Exercise 1: Adding <rect> elements by using DOM

Now that you understand how the Web page is structured, it's time for you to start implement the empty functions.

You'll start with the `addRect()` function, to create a new `<rect>` element and add it as a child of the `<svg>` element in the DOM tree Follow these steps (each of these steps corresponds to a TODO step in the code):

a) Create a new `<rect>` element (with the correct SVG namespace), by using the standard DOM function `document.createElementNS()`. Pass in two parameters:
   - The namespace URI for the element – use the `svgNS` variable here.
   - The tag name for the element, i.e. `"rect"`.

b) Set the `x`, `y`, `width`, `height`, `fill`, and `fill-opacity` attributes on the `<rect>` element. We've already called `getRandomAttributes()` to get random property values, so use the standard DOM function `setAttribute()`to set each attribute on the `<rect>` element. For each call to `setAttribute()`, pass in two parameters:
   - The name of the attribute you want to set, e.g. `"x"`.
   - The value you want to set it to, e.g. `attrs.x`.

c) Append the `<rect>` element as a child of the `<svg>` element, so that it becomes linked-in to the DOM tree. You can use the `append()`jQuery function to achieve this effect, e.g. `$("svg").append(rect)`.

That completes the implementation of the `addRect()` function, so save your work and open your Web page in the browser. Click *Add Rectangle* several times – you should see a new rectangle appear each time, with a random location, size, colour, and opacity.

Exercise 2: Adding <circle> elements by using DOM

The next step is to implement the `addCircle()` function, to create a new `<circle>` element and add it as a child of the `<svg>` element in the DOM tree.

This is similar to the code you wrote in `addRect()`, but note the following points:

- The `<circle>` element requires attributes named `cx` and `cy` that define the centre-point of the circle. Set these attributes using the values in the `attrs.x` and `attrs.y` randomised properties.
- The `<circle>` element also requires an attribute named `r` that defines the radius of the circle. Set this attribute to half of the value in the `attrs.w` randomised property.
- Also remember to set the `fill` and `fill-opacity` properties for the `<circle>` element.

Test your new functionality. You should be able to generate circles at random locations, radii, colours, and opacity.

Exercise 3: Defining styles and handling events

As mentioned in the chapter, one of the key differences between SVG graphics and Canvas graphics is that with SVG, the shapes are actually resident in the DOM tree. This means you can interact programmatically with SVG elements, in exactly the same way as you would with normal HTML elements.

To illustrate these capabilities, this exercise asks you to define CSS style rules for SVG elements, and to handle events on SVG elements.

Let's deal with CSS styles first. At the top of the Web page in the text editor, locate the TODO 3a comment. Define a CSS style rule such that whenever you hover over any element inside an `<svg>` element, it sets its `fill-opacity` property to `1.0`.

Now let's deal with event handling. In the `init()` function, locate the TODO 3b comment. At this location, handle the `mouseup` event for any elements inside an `<svg>` element – use the `live()` jQuery function as follows (remember, this ensures the event handler works for all SVG elements that will ever be created, not just the ones that happen to exist at this moment in time):

```
$("svg *").live("mouseup", function(e) {
    …
});
```

Implement the body of the event handler function as follows:

- If the `SHIFT` and `CTRL` keys are depressed, delete the current element from the DOM tree. Here are some hints:
    - To test if the `SHIFT` and `CTRL` keys are depressed, use the `e.shiftKey` and `e.ctrlKey` properties on the event argument.
    - To delete the current element, use the `$(this).remove()` jQuery function.
- Otherwise, toggle the `stroke` and `stroke-width` attributes (i.e. turn the shape's border on or off). Hints:
    - First, determine if the `stroke` attribute is currently set on the element, by using the `$(this).attr("stroke")` jQuery function.
    - If the `stroke` attribute *is* set, remove this attribute by using the `$(this).removeAttr("stroke")` jQuery function. Do likewise to remove the `stroke-width` attribute.
    - If the `stroke` attribute *isn't* set, set this attribute now by using the `$(this).attr("stroke", "red")` jQuery function (for example). Do likewise to set the `stroke-width` attribute (e.g. set it to `5`).

After you've done all this, test your new functionality fully in the browser.

Exercise 4: Locating elements in the DOM tree

When you're adding and removing content dynamically in a Web page, you often find yourself needing to locate particular elements in the DOM tree. For example, you might want to find all `<li>` elements in a list, find all text boxes in a form, etc.

You can do likewise with SVG elements. To illustrate this capability, this exercise asks you to implement code to locate all `<rect>` elements and then double or halve their dimensions.

Locate the `growRects()` function. The purpose of this function is to locate all the `<rect>` elements and to double the width and height of each one. Use the following jQuery code to locate all the `<rect>` elements and perform processing on each one.

```
$("rect").each(function() {

  … within this function, $(this) represents the current rect …

});
```

For each `<rect>` element, double its `width` and `height` as follows:

- Get the current value of its `width` and `height` attributes (see the previous exercise for a reminder of how to get an attribute value using jQuery).
- Double the values of the `width` and `height`.
- Set the new values for the `width` and `height` attributes (again, see the previous exercise for a reminder).

Now locate the `shrinkRects()` function. Implement this function in a similar way to `growRects()`, but this time, halve the `width` and `height` of each `<rect>` element.

After you've done all this, test your new functionality in the browser. Verify that you can double and halve the dimensions of every rectangle (whilst the circles remain unaffected).

Exercise 5 (If Time Permits): Transforming SVG content

As we described in the chapter, SVG supports transformations such as scaling, translation, rotation, and skewing. We showed how to define static transformations using fixed mark-up such as the following:

```
<g transform="translate(320, 0), scale(2), skewX(-30)">
    <rect … … … />
</g>
```

In the static mark-up above, we've defined a `<g>` element to wrap the content that we want to transform. The `<g>` element has a `transform` attribute that specifies the transformation(s) that we want to apply to that content.

It's also possible to apply transformations dynamically, by creating the `<g>` element programmatically and adding it to the DOM tree where needed. The *Solution* application illustrated this behaviour via the *Scale All Up* and *Scale All Down* buttons, which scaled all shapes by a factor of `2.0` (to double their scale) and `0.5` (to halve their scale) respectively.

Try to implement this behaviour for yourself in the *Start* application, in the `scaleAllUp()` and `scaleAllDown()` stub functions. Here are some hints:

- First, you need to locate all the elements inside the `<svg>` element…
- For each child element, you need to wrap it in a separate `<g>` element...
- When you create `<g>` elements, remember to specify the SVG XML namespace. Also, set the `transform` attribute to an appropriate transformation string.
- Use the `$(this).wrap()` jQuery function to wrap each element in a `<g>` element.

Exercise 6 (If Time Permits): Adding <polygon> elements by using DOM

Implement the `addPolygon()` function so that it adds a regular polygon to the DOM tree. Randomise the number of points in the polygon, and do some maths to calculate the coordinates of each point in the polygon. Here are some hints:

- You're aiming to create a `<polygon>` element (!).
- You'll need to use SOH/CAH/TOA to calculate the x and y coordinate of each point. The `Math.sin()` and `Math.cos()` functions will come in handy here.
- Set the `points` property on the `<polygon>` element, as a series of x and y coordinates. Make sure you get the syntax correct for this property!