In this article I provide a brief overview on Scalar Vector Graphics (SVG) defined using HTML tags, including circle, ellipse, Bezier curves and arcs. I then show how to create and modify SVG elements 'on the fly' using JavaScript based on user interactions with buttons and sliders.

Scalar Vector Graphics is a standard, distinct from HTML or anything else, for producing graphics. You can think of it as drawing instructions in the form of XML elements. Basic shapes include ellipses, rectangles and paths. Paths are made up of line segments and curves. Though SVG can be produced just using tags in the body of an HTML document, you also can create and/or change SVG, making it dynamic as opposed to static, with JavaScript. I will explain dynamic SVG through two applications. Figure 1 shows the opening screen of an application displaying the HTML5 logo created using SVG tags. The screen shot was taken with Chrome as the browser. Certain browsers, most notably Firefox, do not display a slider, but default (also termed “gracefully degrade”) to a text field for input elements of type range.

I built this program by converting an application I had done previously using the HTML5 canvas. Figure 2 shows the logo re-scaled to be a smaller size after I changed the slider.

To demonstrate a variety of SVG features, including dynamic creation as well as transformation of elements, I created another application featuring a cartoon drawing. Figure 3 shows the opening screen, with a blank head-like figure plus buttons and another slider input to add features.
I chose what I did to demonstrate dynamic SVG and scaling. I could have programmed this in any number of ways. The hierarchical structure of the SVG elements enables the moving mouth, the nose is a path consisting of a line followed by an arc. It is implemented as 2 SVG ellipses, and 3 SVG paths. More specifically, the nose and mouth for the cartoon. It also is important to define the graphics and the face is made up of two eyes, a nose, a mouth and one slider. The user interface consists of 4 buttons and a single hair.

Creating graphics using SVG is not a trivial task. You need to determine those pesky coordinate values. Putting it another way, you can produce quite complex graphics assuming you have the inclination to take the time. I acquired the data for the HTML5 Logo from another source (see my book cited in the Learn More section) and it took some time to produce the single hair, mouth and nose for the cartoon. It also is important to define the graphics so the structure captures the appropriate hierarchy. However, once those operations are complete, generating the code for the manipulation of the graphics is easily accomplished.

Overview

To motivate what follows, here is an overview of the two applications. The geometric terms, e.g., polygon, correspond to SVG primitives and will be explained in the later sections.

The HTML5 Logo application has the basic design in HTML markup within the body element. It consists of SVG polygons along with SVG text. The polygons are defined in a specific order so that the result is the official logo. That is, a polygon defined later covers up what was defined earlier. The markup includes an input element of type range that produces the slider.

The cartoon application also includes markup in the body element for 3 circles. One of the circles is defined, and consequently drawn, last and it has the effect of erasing some of the stroke (outline or border) lines of the first two. The user interface consists of 4 buttons and one slider. The face is made up of two eyes, a nose, a mouth and a single hair. These are created entirely by JavaScript code and attached as child nodes to the head SVG element. The face parts are implemented as 2 SVG ellipses, and 3 SVG paths. More specifically, the nose is a path consisting of a line followed by an arc. The mouth is a single Bezier curve and the hair is a compound Bezier curve. The hierarchical structure of the SVG elements enables the moving and scaling. I could have programmed this in any number of ways. I chose what I did to demonstrate dynamic SVG.

Background on SVG elements

SVG is a form of XML. As such, there is a namespace that specifies the possible parts. You do not have to appreciate the deep significance of this. The tags will be similar to HTML tags and include familiar HTML attributes such as id and height and width. The <svg> tag for the HTML5 Logo as shown in Listing 1 contains the indicated setting for the xmlns attribute. What is critical to understanding is that svg elements are structured and the structure matters. It also means that you cannot be careless about opening and closing of elements and omitting quotation marks for attribute values. In the HTML5 Logo example, the polygon elements are contained within a g (I think this stands for group) element which contains a setting for the transform attribute. This in turn is the second element contained in a g element. The first element is a text element. The transform="translate(0,80)" moves all the polygons down the screen to make room for the text. That is, the first operand for translate is the x or horizontal value and the second is the y or vertical value. Remember that vertical values increase moving down the screen. This is necessary because I copied the coordinate values from a site that just produced the shield for the HTML5 Logo, without the text.

A text SVG element has the text content as the content of the element, that is, between the opening and closing tags. Valid attributes include x and y positions to indicate the relative location within the parent element, font-family and font-size. The font-family designation can include back-up choices, which is what I do in this situation. There are other attributes as well. I wanted the default color black for both the stroke attribute (this is the outline) and the fill attribute, so I could omit them.

```
<svg id="wholesvg" height="600" width="800" xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <g id="logo">
    <text x="45" y="60" font-family="Gill Sans Ultra Bold", sans-serif" font-size="54">HTML</text>
    <g id="shield" transform="translate(0,80)">
      <polygon points="39 250, 17 0, 262 0, 239 250, 139 278" fill="#F06529"/>
      <polygon points="139 113, 103 144, 105 184, 72 159, 62 51, 70 144, 139 144" fill="#E34C26"/>
      <polygon points="139 193, 105 184, 103 159, 72 159, 62 51, 70 144, 139 144" fill="#EBEBEB"/>
      <polygon points="139 113, 98 113, 96 82, 139 82, 139 51, 62 51, 70 144, 139 144" fill="#E8E8E8"/>
      <polygon points="139 193, 105 184, 103 159, 72 159, 62 51, 70 144, 139 144" fill="#F06529"/>
      <polygon points="139 113, 103 144, 105 184, 72 159, 62 51, 70 144, 139 144" fill="#E34C26"/>
      <polygon points="139 193, 105 184, 103 159, 72 159, 62 51, 70 144, 139 144" fill="#EBEBEB"/>
      <polygon points="139 113, 103 144, 105 184, 72 159, 62 51, 70 144, 139 144" fill="#F06529"/>
    </g>
  </g>
</svg>
```

Listing 1: The svg element for the HTML5 logo

The polygon element has an attribute named points. The value is a sequence of x and y pairs. Notice that there is a comma after each pair and no comma in-between. Another attribute used here is fill.

I continue here explaining more SVG elements. I will return to the manipulation of the graphics in a later section.
Another common SVG element is circle. The cx and cy attributes define the center of the circle; the r attribute specifies the radius. Examine Listing 2 and you will see the specification of the peanut shaped head for the cartoon. Notice the definition of the fill and stroke attributes in the 3 circles. The circle with id="middle" effectively covers up the stroke lines in the interior of the peanut. By the way, the 3 circles are all still present and if we wrote JavaScript to move them "out from under" so to speak, each would appear as a complete item. This is in contrast to drawing on a canvas element. Once drawn, a figure ceases to exist. Think of it as paint on canvas.

You can see all the source code by taking the link indicated in the Learn More section. An init function includes the statement:

```javascript
myHead = document.getElementById("head");
```

Setting a variable to reference the head element means it can be referenced later in the program.

An ellipse SVG element has the x and y attributes and additional attributes governing the horizontal (rx) and vertical (ry) radii. I used the ellipse element for the eyes, for which I wrote code to create these elements dynamically as explained in the next section. You can look ahead to Listing 4.

The path SVG element is the most general. It has an attribute named d (for data) that accepts a string consists of a sequence of operators and operands. The M operator defines a move to an x y position. Once a position is defined, there are operators for drawing a line segment (L) and several types of curves, including an elliptical arc (A) and a cubic Bezier curve (C). Lower case letters (m, l, a, c, for example) treat the coordinates supplied as relative values. In what follows, "l" is a lower-case "L". To demonstrate absolute versus relative: "M 100 200 L 110 250" is the same as "M 100 200 l 10 50". The "L 10 50" indicates a position 10 over in x and 50 over in y from the previous position.

I found this website the most useful for understanding the various Paths. Many other sites exist, including special tools for creating Bezier curves. However, I suspect that any specific problem will require experimentation. Here is an overview on my use of curves.

The operands for the a (elliptical arc) operator are: the horizontal radius, vertical radius, x-axis rotation (in degrees, not radians, in contrast to what is expected for drawing on canvas and many other computer applications), the large arc flag, the sweep flag and the endpoint (x and y). The starting point of the arc is the current position, perhaps given by a m command. The two flags indicate which of 4 different possible ellipses you want.

I wrote JavaScript code to produce the d attribute values for the nose, hair and mouth. The last part of the data for the nose indicated a straight line and then a circular arc. I defined a variable

```javascript
var nosedataltail="l 15 22 a 20 20 0 0 1 -15 4";
```

This achieves the desired effect.

A cubic Bezier curve is defined using positions and control points. As with the arc, the starting point is the current position and the operands are the first control point (x and y), the second control point (x and y), and the ending point (x and y). After specifying one curve, you can specify another one that connects smoothly with the first using the s command. The first control point for the subsequent curve is calculated from the last control point. This means the s command just requires two operands: a control point (x and y) followed by the end position. The effect I wanted for the hair was two cubic curves so I used the c operator and then the s operator serves to tack on another Bezier curve. The last part of the data for the hair was defined

```javascript
var hairdatatail = "c -50 30, 0 25, 15 30 s 2 12 -20 30";
```

### Creating new SVG elements

It is always possible to create HTML elements dynamically and make them visible by appending them as child elements to something already in the document. There is a special method for creating SVG elements that is required because SVG elements are associated with a specific namespace. The method required is a method of document and is called `createElementNS`. It takes two arguments: the namespace and the type of element. Listing 3 shows the coding for creating the parts of the face and attaching them to the `<svg>` element. Doing this means that the mouth, eyes, nose and hair will move and scale with the head, exactly what I want.
<p>The null parameter used in my code refers to the default namespace.</p>

```javascript
function setFace() {
    var nosedata = "M" +String(cheadx)+" "+String(cheady)+" " +"nosedataTail; 
    var mouthx1 = cheekx -.25*rx; //starting x 
    var mouthy = cheeky + ry; // starting y 
    var mouthx2 = cheekx+.25*rx; //end point x 
    var mouthy2 = cheeky -.25*ry; //control point y 
    var mouthx3c = mouthx1+.15*rx; // left control point x 
    var mouthx2c = mouthx2-.15*rx; // right control point x 
    var mouthdata = "M"+String(mouthx1)=" " 
    + String(mouthy)+" C" 
    + String(mouthx1c)+" " 
    + String(mouthyc)+" " 
    +String(mouthx2c)+" " 
    +String(mouthyc)+", " 
    +String(mouthx2)+" " 
    +String(mouthy);

    myLeftEye.setAttributeNS(null,"rx",ex); 
    myLeftEye.setAttributeNS(null,"cy",eyey); 
    myLeftEye.setAttributeNS(null,"cx",leftx); 
    myRightEye.setAttributeNS(null,"ry",eyey); 
    myRightEye.setAttributeNS(null,"cx",rightx); 
    myRightEye.setAttributeNS(null,"cy",eyey); 
    myRightEye.setAttributeNS(null,"rx",ex); 
    myRightEye.setAttributeNS(null,"ry",ey);
}
```

Listing 4: Setting the attributes for the face elements

Creating the text node is similar, but with one difference. It is necessary to create the contents of the text node in addition to setting attributes and that requires use of the <code>.createTextNode</code> method. The code is shown in Listing 5.

```javascript
function addText() {
    if (textadded) {
        alert("text already added ");
        return;
    }
    myText = document.createTextNode("My Daddy Joe");
    myHead.appendChild(myText);
    myText.setAttributeNS(null,"font-family","serif");
    myText.setAttributeNS(null,"font-size",16);
    myText.setAttributeNS(null,"font-weight","bold");
    myText.setAttributeNS(null,"y",12);
    textadded = true;
}
```

Listing 5: Adding the text element to the document

### Changing SVG based on user input

You may not realize it, but you already know the critical method for changing SVG attributes. It is <code>setAttributeNS</code>, the same method to set an attribute to an initial value. To determine how to write the code to respond to the Move horizontally and Move vertically buttons and the slider/range input, the question is what attribute or attributes need to be changed? The code setting up the responses to user actions is shown in Listing 6. Notice that the <code>onChange</code> for the slider/range input field is set to call the function <code>changeScale</code> with parameter the current value of the input field.

```javascript
<body>
  <button onClick="moveOvalH();">Move horizontally </button>
  <button onClick="moveOvalV();">Move vertically</button>
  <button onClick="createFace();">Add face </button>
  <button onClick="addText();">Add Text </button>
  <button onClick="moveOvalH();">Move horizontally </button>
  <button onClick="moveOvalV();">Move vertically</button>
  <button onClick="createFace();">Add Face </button>
  <button onClick="addText();">Add Text </button>

  <input type="range" onChange="changeScale(this.

  Head scale <input type="range" onChange="changeScale(this.

  <p>Listing 6: HTML markup setting up buttons and slider</p>

The answer to the question of what attribute needs to be changed is the <code>transform</code> attribute of the <code>myHead</code> element. This is because all the elements making up the face plus the text are child nodes of the element pointed to by the variable <code>myHead</code>. The <code>transform</code> attribute takes as value a String consisting of the term <code>translate</code> followed by a position, given in the form (x, y), followed by the term scale followed by (value). The code for <code>createFace</code> and <code>addText</code> has been shown already (Listing 3 and Listing 5). The code for the <code>moveOvalH</code>, <code>moveOvalV</code> and <code>changeScale</code> is shown in Listing 7. All the face parts and the text will be transformed when the <code>myHead</code> element is transformed. This code makes use of variables, such as <code>headx</code> and <code>heady</code>. You can check out the whole source code by going to the site provided in the Learn More section.

```javascript
function moveOvalH() {
    if (headx>300) {headx = 0;}
    var trans = "translate("+String(headx)+","+String(heady)+") scale("+String(scalefactor)+")";
    myHead.setAttributeNS(null,"transform",trans);
}
```

Listing 7: Functions responding to buttons and slider
The HTML5 Logo example is similar. In this case, the code invoked by the change of the range/slider input is shown in Listing 8.

```javascript
function changeScale(val) {
    factorvalue = val/100;
    var factorvalues = String(factorvalue);
    var trans = "translate(0,0) scale("+factorvalues+")";
    logo.setAttributeNS(null,"transform",trans);
}
```

Listing 8: The changeScale function for the HTML5 Logo program

These are teaching examples demonstrating a hodge-podge of techniques. More exactly, the examples incorporate a variety of SVG elements, created in HTML markup or with JavaScript code, and include how to make changes to SVG element attributes, mainly to demonstrate different capabilities of HTML, JavaScript and SVG and also illustrate how techniques can be combined. Once you understand the possibilities, you can use these techniques to expand the power of your websites. You also will be able to compare the use of SVG with the use of the HTML5 canvas element.

Learn more

There are many sources, online and in-print and some sort of e-books, for learning HTML5 and JavaScript techniques along with SVG. Here are links to my recent books and the website for these examples.

- The Essential Guide to HTML5: Using Games to learn HTML5 and JavaScript, [http://www.friendsofed.com/book.html?isbn=9781430233831](http://www.friendsofed.com/book.html?isbn=9781430233831). This is a text for beginners at programming as well as more experienced programmers who want to learn about HTML and JavaScript, including the new features of HTML5, including canvas, an alternative to SVG.

- HTML5 and JavaScript Projects, [http://www.apress.com/9781430240327](http://www.apress.com/9781430240327). This book is more advanced than the first one. The first chapter describes producing essentially the same HTML5 Logo application using the HTML5 canvas element.

- To see the applications in action and to view the source code, go to [http://faculty.purchase.edu/jeanine.meyer/html5/html5logoscalesvg.html](http://faculty.purchase.edu/jeanine.meyer/html5/html5logoscalesvg.html) for the HTML5 Logo and to [http://faculty.purchase.edu/jeanine.meyer/html5/littlejoesvg2.html](http://faculty.purchase.edu/jeanine.meyer/html5/littlejoesvg2.html) for the cartoon.

Jeanine Meyer lives just north of New York City and currently teaches at Purchase College/SUNY after many years at IBM, doing research on robotics and manufacturing and consulting on educational grants. She likes providing programming examples for her Mathematics/Computer Science and New Media students and building applications involving media of her family. The cartoon application features a drawing her Daddy would make to represent himself.