Scalable Vector Graphics

**SVG**

Scalable Vector Graphics (SVGs) is a format for Vector Graphics that uses XML to describe the image. The format will not be covered in a large amount of detail since it uses XML. Specifics to SVG will be discussed. SVG images are commonly used on the web and can be tested with HTML, JS.

General Format

The general format follows XML and looks like this:

<type element=” ” element2=” ” element3=” ” />

Or

<type element=” ” element2=” ” element3=” ” >

</type>

The second format is an encapsulating format. Elements inside it inherit attributes.

* The numbers in an SVG file can be of different formats
  + Integer values
  + Floating points
  + Hexadecimal values
* The numbers may also have different extensions
  + pt, px
* Extensions at the end of numbers will be ignored for our purposes.
* The color can be of 3 formats
  + rgb(float,float,float)
  + Hexadecimal
  + Name
    - Example: black, blue, green

SVG Tag

* The SVG tag is the first and most important tag. Every document will begin with it. The width and height must be specified. A viewbox may not be specified. xmlns does not have to be specified and may be skipped for our purposes.
* Only elements encapsulated in the SVG tag are apart of this vector graphic.
  + This allows a file to have multiple vector graphics stored in one file.
* Format:
  + <svg width=”0” height=”0” background-color=”color” viewbox=”0 0 0 0”>
  + </svg>
    - background-color can be treated as the clear color for the image.

SHAPE TAGS

* The tags for the general shapes are as follows:
  + rect, circle, ellipse, polygon, polyline, path, text, image
* There are encapsulating tags such as
  + g
    - Used for transforms
* Some attributes for these tags are shared
  + transform=”translate(x,y) scale(x,y) scale(both) rotate(x) skewX(x) skewY(y) matrix(a,b,c,d,e,f)”
    - Scale does not have to have both x and y values
  + fill=”color”
  + fill-opacity=”number”
  + stroke=”color”
  + stroke-opacity=”number”
  + stroke-width=”number”
  + linecap=”type”
    - type can be butt, square, round
  + linejoin=”type”
    - type can be miter, round, bevel
  + fill-rule=”type”
    - type can be evenodd, nonzero

RECT Tag

* The rect tag specifies a rectangle. The rectangle can have round corners specified by an x radius and y radius. Since all objects and be scaled, skew, and rotated, special care must be taken so that it drawn correctly in all cases.
* Affected by linejoin attribute
* Not affected by linecap and fill-rule
* Format:
  + <rect x=”0” y=”0” rx=”0” ry=”0” width=”0” height=”0” />

CIRCLE Tag

* The circle tag specifies a circle. The circle can be affected by all transforms.
* Not affected by linejoin, linecap, and fill-rule
* Format:
  + <circle cx=”0” cy=”0” r=”0” />

ELLIPSE Tag

* The ellipse tag specifies a ellipse. The ellipse can be affected by all transforms. An ellipse can have a different x and y radius than a circle.
* Not affected by linejoin, linecap, and fill-rule
* Format:
  + <ellipse x=”0” y=”0” rx=”0” ry=”0” />

POLYGON Tag

* The polygon tag specifies a general polygon. The polygon can be affected by all transforms. A polygon always connects the first and last points together to form a completely closed shape.
* Affected by linejoin, linecap, and fill-rule
* Format:
  + <polygon points=”x1,y1 x2,y2 x3,y3” />
  + Points come in pairs and are separated by spaces.

POLYLINE Tag

* The polyline tag specifies a polyline. The polyline can be affected by all transforms. A polyline does not connect the first and last points together unlike a polygon.
* Affected by linejoin, linecap, and fill-rule
* Format:
  + <polyline points=”x1,y1 x2,y2 x3,y3” />
  + Points come in pairs and are separated by spaces.

PATH Tag

* The path tag specifies a shape defined by lines, arcs, and Bezier curves. A path can be affected by all transforms.
* Affected by linejoin, linecap, and fill-rule
* Format:
  + <path d=”stuff” />
  + (also note that lowercase means relative to the previous location)
  + M X,Y = move to x, y. Ex: M20,30
  + L X,Y = line to x, y. Ex: L50,50
  + H x = horizontal line to x. Ex: H0
  + V y = vertical line to y. Ex: V0
  + Z = close path. (Z and z are the same in this case)
  + C x1 y1, x2 y2, x y = Cubic bezier curve.
    - The start of the curve is the previous point, x1 y1 is the control for first point
    - x y is the end point. x2 y2 is the control for the end point
  + S x2 y2, x y = Cubic bezier curve (Shorthand)
    - Assumes that x1 y1 are the reflection of x2 y2 across the start point
    - x1 = startX + (startX - previousX2)
    - y1 = startY + (startY - previousY2)
  + Q x1 y1, x y = Quadratic bezier curve
    - start of the curve is the previous point, x1 y1 is the control point
    - x y is the end point
  + T x y = Quadratic bezier curve (Shorthand)
    - similar to S, assumes x1 y1 are the reflection of the previous control point across the start point.
    - x1 = startX + (startX - previousX1)
    - y1 = startY + (startY - previousY1)
  + A rx ry rot largeArcFlag sweepFlag x y = Arc
    - rx = x radius
    - ry = y radius
    - Rot = the rotation of the ellipse
    - largeArcFlag = should angle be greater than 180 flag
    - sweepFlag = which direction. (positive or negative/clockwise or counter clockwise)
    - x = x endPoint
    - y = y endPoint
      * The combination of both flag determines which circle to use
      * There are 2 circles which are reflected over the start point and end point
      * The start point is the previous end point
      * The end point is specified by x,y
      * May need to expand the ellipse so that they both make contact with the start and end points
      * Solve some system of equations to do that.

An Approach to Drawing the objects

Many of the objects (minus the circle, ellipse, arc) can be drawn in a similar way. Finding the valid x values for each valid y and filling in-between. This general approach allows for us to apply all transforms to the object and render them the same way. In terms of performance, it involves a matrix multiplication for each defined point in the description of the object. Many of the other operations do not change. Every shape would change into a path except circle and ellipse. Though those could be changed into a path as well.

The other approach would be to draw the object like normal but rotate the pixels location separately. The amount of matrix multiplications would be based on how many pixels are being drawn. This method would trump in performance, however, due to the large amount of matrix multiplications.

The main challenge becomes how to draw arcs, ellipses, and circles efficiently while still applying the appropriate transforms.

A method that could work well for circles, arcs, and ellipses is to find the interval of x values valid on a scanline and then apply the transform to that. You have positions that basically represent a line and can therefore treat them as separate objects. This reduces the amount of work needed since the amount of matrix multiplications needed is 2Y where Y is distance from the top of the object to the bottom. Essentially 2\*yRadius.

The other approach to circles, arcs, and ellipses is to use a series of cubic Bezier curves to approximate the curve. The amount of matrix multiplications needed will be lower however, it is just an approximation. The approximation fails at high resolutions and radius values or with large amounts of scaling. It would be 16 matrix multiplications in total which is much better. Increasing the number of cubic curves changes how accurate the approximation is. 32 or 64 matrix multiplications. Interestingly for small radii, the amount of operations needed can be smaller than the cubic curves if the radius is small enough.

Drawing Cubic Bezier Curves, unlike Quadratic Bezier Curves, require complex number calculations which involve trigonometric functions. This in turn makes it quite slow when solving for Y values. For Quadratic Bezier Curves, you could use the quadratic equation to find Y values. Both however are not the fastest, but they are more accurate.

Approximating the curves is a valid and common approach. Determining how to sub divide the curve to get a more accurate drawing and evaluating how accurate the drawing is complex. In the worst case, it is O(N) where N is the total number of steps. Should subdivide where the slope is 0 since the slope will be in the opposite direction on the other side. This involves solving a quadratic equation however, it must be solved only once per curve creation. For each subdivision, we use N steps to approximate that area. The number of steps is based on the distance traveled to that point from the previous point. The distance rounded up to nearest point will be how many steps you need.

Due to the nature of approximations, we will test the performance of both cases. With cubic, we either have 3 solutions or 1 solution. The formula works with one solution. With 3 solutions, we must use trig along with some additional square roots. Then we can reduce to find the other solutions without using trigonometry. The total amount of calculations is less than the approximations when curves don’t have a large number of Y values. If it does, the approximation can be better. The approximation does have to subdivide and approximate each subdivision which can be up to 3 subdivisions.

If (4P^3 + 27Q^2 < 0)

Use Trig then reduce

else

Only one solution, use formula