SVG

- **Scalable Vector Graphics (SVG)** is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C) since 1999.

- SVG images and their behaviors are **defined with XML**. This means that they can be **searched, indexed, scripted, and compressed**. As XML files, SVG images can be created and edited with any text editor, as well as with drawing software.

- All major modern web browsers—including Mozilla Firefox, Internet Explorer, Google Chrome, Opera, Safari, and Microsoft Edge—have SVG rendering support. **SVG XML code** is either embedded in HTML pages wrapped with specific root tag (in the same way we did it with MathML) or maintained as separate XML files.

- Since **SVG is also a subset of XML** it follows the same syntax rules as we defined for XML or MathML.
Vector vs Raster Graphics

- Although you can embed bitmap/raster images in SVG code, **SVG is a vector graphics format in the first place**.
- **Vector graphics** is the use of polygons to represent images in computer graphics. Vector graphics are based on vectors, which lead through locations called control points or nodes. Each of these points has a definite position on the x- and y-axes of the work plane and determines the direction of the path; further, each path may be assigned various attributes, including such values as stroke color, shape, curve, thickness, and fill.
- In turn, in **raster graphics** all objects must be drawn pixel by pixel, so that even creating a line from one point to another might require you to write a function in a programming language.
- In vector graphics many of those details are taken care of by the browser or other software client: the user has only to worry about **higher-level graphical objects such as lines, circles, polygons, and so on**.
SVG Basics

• SVG code (either embedded in HTML code or placed in a separate XML file) must be enclosed in `<svg>...</svg>` root tag. You can specify width (e.g. `width="100"`) and height (e.g. `height="150%"`) attributes using standard HTML units to define size of your image. Sometimes you may also see version (e.g. `version="1.1"`) and xmlns (xmlns="http://www.w3.org/2000/svg") attributes. They tell the web browser what exact version of SVG specification should be applied while rendering the image. Here is an example:

```xml
<svg width="300" height="200">
  <rect width="100%" height="100%" fill="red" />
  <circle cx="150" cy="100" r="80" fill="green" />
  <text x="150" y="125" font-size="60" text-anchor="middle" fill="white">SVG</text>
</svg>
```

• In the above example the background is set to red by drawing a rectangle that covers the complete image area with `<rect/>` tag.

• A green circle `<circle/>` with a radius of 80px is drawn atop the center of the red.

• The text "SVG" is drawn `<text/>` tag. The interior of each letter is filled in with white. The text is positioned by setting an anchor at where we want the midpoint to be: in this case, the midpoint should correspond to the center of the green circle. Fine adjustments can be made to the font size and vertical position to ensure the final result is aesthetically pleasing.
Positions

• For all elements, SVG uses a coordinate system or **grid system**. That is, the **top left corner** of the document is considered to be the point \((0,0)\). Positions are then measured in **pixels from the top left corner**, with the positive x direction being to the right, and the positive y direction being to the bottom. For example, the following code defines a rectangle from the upper left corner, that spans from there 100px to the right and to the bottom: `<rect x="0" y="0" width="100" height="100" />

• In contrast to **HTML**, **SVG** elements are not rendered sequentially, i.e. one after another. Instead **later** elements are rendered atop **previous** elements (unless an absolute positions are specified in the way that elements don’t overlap). The further down an element is the more will be visible.
Basic Shapes

• **Rectangle:** `<rect/>` tag (attributes: `x` - the x position of the top left corner of the rectangle, `y` - the y position of the top left corner of the rectangle, `width` - the width of the rectangle, `height` - the height of the rectangle, `rx` - the x radius of the corners of the rectangle, `ry` - the y radius of the corners of the rectangle)

• **Circle:** `<circle/>` tag (attributes: `r` - the radius of the circle, `cx` - the x position of the center of the circle, `cy` - the y position of the center of the circle).

• **Ellipse:** `<ellipse/>` tag (attributes: `rx` - the x radius of the ellipse; `ry` - the y radius of the ellipse, `cx`, `cy` – the same as for circle).

• **Straight Line:** `<line/>` tag (attributes: `x1`, `y1` - the x and the y positions of the starting point, `x2`, `y2` - the x and the y positions of the starting point).
Basic Shapes

- **Polyline** (group of connected straight lines): `<polyline/>` tag (attributes: `points` - a list of point coordinates separated comma. Each point coordinate must contain two numbers, an x coordinate and a y coordinate. So the list (0,0), (1,1) and (2,2) could be written: "0 0, 1 1, 2 2").

- **Polygon** (the same as polyline, by the path automatically returns to the first point at the end): `<polygon/>` tag (attributes: `points` - the same as for polyline).

- **Text**: `<text/>` tag (attributes: `x, y` - the x and the y positions of the starting point, `font-family, font-style, font-weight, font-variant, font-stretch, font-size, letter-spacing, word-spacing, text-decoration` - may be used to customize font directly as attributes or as CSS properties defined by `style` or `class` attribute).
Paths

• The `<path/>` element is the most powerful element in the SVG library of basic shapes. You can use it to create lines, curves, arcs and more. Paths create complex shapes by combining multiple straight lines or curved lines.

• The shape of a path element is defined by one attribute: `d`. The "d" attribute contains a series of commands and parameters used by those commands.

• Each of the commands is defined by a specific letter. For instance, if we want to move to the x and y coordinates (10, 10). The "Move to" command is called with the letter `M`. When the parser runs into this letter, it knows you want to move to a point. So, to move to (10,10) you would use the command "M 10 10". After that, the parser begins reading for the next command.

• All of the commands also come in two variants. An uppercase letter specifies absolute coordinates on the page, and a lowercase letter specifies relative coordinates (e.g. move from the last point 10px up and 7px to the left).
Paths

- Commands may be also divided into two types as **line commands** and **curve commands**.

- There are five **line commands** available. **M/m** – moves current position to specified coordinates \((x, y)\) attributes without actually drawing anything; **L/l** – draws a line from the current position to a new position \((x, y)\) attributes; **H/h** and **V/v** – draws a horizontal or vertical line (respectively) from the current position to a new position (only one relevant attribute, \(x\) or \(y\), should be used since we are moving in a single dimension); **Z/z** – simply draws a straight line from the current position back to the first point of the path (no attributes required).

- There are three different commands that you can use to **create smooth curves**.

  - **A Cubic Beziers Curve** (see [https://en.wikipedia.org/wiki/B%C3%A9zier_curve](https://en.wikipedia.org/wiki/B%C3%A9zier_curve)) can be created with **C/c** command. Cubic Beziers take in two control points for each point. Therefore, to create a cubic Bezier, you need to specify three sets of coordinates: \((x_1, y_1)\), \((x_2, y_2)\), \((x, y)\). The last set of coordinates here \((x, y)\) are where you want the line to end. The other two are control points. The control point for the start of your curve is \((x_1, y_1)\), and \((x_2, y_2)\) is the end point of your curve. The control points essentially describe the slope of your line starting at each point. The Bezier function then creates a smooth curve that transfers you from the slope you established at the beginning of your line, to the slope at the other end.

  - You can **string together several Bezier curves** to create extended, smooth shapes. Often, the control point on one side of a point will be a reflection of the control point used on the other side to keep the slope constant. In this case, you can use a shortcut version of the cubic Bezier, designated by the command **S/s** which takes following arguments: \((x_2, y_2)\), \((x, y)\).

  - **A Quadratic Beziers Curve** could be produced with **Q/q** command and to join two curves **T/t** command may be used. An arc could be created with **A/a** command.

There are several ways to color shapes. The most basic is specifying attributes directly on the object. The other way may be CSS (either inline or, an embedded CSS section, or an external CSS file.

Basic coloring can be done by setting two attributes on the node: fill and stroke. Fill sets the color inside the object and stroke sets the color of the line drawn around the object. You can use the same CSS color naming schemes that you use in HTML, whether that’s color names (that is red), rgb values (that is rgb(255,0,0)), hex values, etc.

In addition to its color properties, there are a few other attributes available to control the way a stroke is drawn on a line. The stroke-width property defines the width of this stroke. The stroke-linecap property defines how line endings should look like (possible values are butt, square, round). The stroke-linejoin is used to control how the joint between two line segments is drawn (possible values are miter, round, bevel). Finally, you can also use dashed line types on a stroke by specifying the stroke-dasharray attribute. The stroke-dasharray attribute takes a series of comma-separated numbers as its argument. The numbers go in pairs: the first number specifies a distance for the filled area, and the second a distance for the unfilled area. You can specify more numbers if you would like a more complicated dash pattern.

To change opacity of both stroke and fill use stroke-opacity and fill-opacity respectively with possible values ranging from 0 to 1.
Gradients

• Frequently SVG files will require customized gradients or other such graphics constructs. These things are not visible objects, but rather values for attributes that other vector objects will possess. They are defined in a <def> section near the top of the SVG file. For example:

```xml
<defs>
  <linearGradient id="orange_red" x1="0%" y1="0%" x2="100%" y2="0%">
    <stop offset="0%" style="stop-color:rgb(255,255,0); stop-opacity :1"/>
    <stop offset="100%" style="stop-color:rgb(255,0,0); stop-opacity :1"/>
  </linearGradient>
</defs>
```

• The first line defines a linear gradient color scheme that will be oriented from left to right. We know that it is oriented left to right because its beginning is at x=0%, y=0%, while its end is at x=100% of the width of the graphics object where it is used, but y=0% still. In other words, the variation in the gradient happens only over the width of the graphics object, with no variation over the height. The first <stop> tag indicates the color at the start (x1,y1) of the gradient; in this case it is red+green = yellow. The second <stop> gives the color at the end of the gradient (x2,y2); here it is red. We can refer to the gradient in our graphics objects using the id attribute: here we will call it orange_red.
ICE #6

• Write an SVG page to create the following graphic:

Sources

• Math 300 Notes by Prof Cooper ([http://www.math.wsu.edu/math/kcooper/M300/text.php](http://www.math.wsu.edu/math/kcooper/M300/text.php))
• SVG Cheat Sheet by Prof Cooper ([http://www.math.wsu.edu/math/kcooper/M300/svgcheat.php](http://www.math.wsu.edu/math/kcooper/M300/svgcheat.php))
• SVG Specification by W3C ([https://www.w3.org/TR/SVG/](https://www.w3.org/TR/SVG/))