UNIT – 3

HTML5 – SVG

Introduction : Scalable Vector Graphics

SVG stands for Scalable Vector Graphics. It is a widely used XML-based vector graphics format for displaying two-dimensional graphics on the web. SVG is supported by all major web browsers and provides a flexible and resolution-independent way to create and display graphics.

Unlike **raster graphics** formats like JPEG or PNG, which are composed of a fixed grid of pixels, **SVG graphics** are defined using mathematical equations and geometric shapes. This means that SVG images can be scaled to any size without losing quality. Whether you view an SVG image on a small screen or a large monitor, it will retain its sharpness and clarity.

SVG graphics are created using XML tags and attributes to define shapes, paths, colors, gradients, text, and other graphical elements. These elements can be manipulated and styled using CSS (Cascading Style Sheets) or animated using JavaScript.

SVG came about in **1999** after several competing formats had been submitted to the W3C (World Wide Web Consortium) and failed to be fully ratified. SVG is supported by all major browsers.

HTML provides elements for defining headers, paragraphs, tables, and so on. In much the same way, SVG provides elements for circles, rectangles, and simple and complex curves. A simple SVG document consists of nothing more than the **<svg>** root element and several basic shapes that build a graphic together. In addition, there is the **<g>** element, which is used to group several basic shapes together.

SVG supports gradients, rotations, filter effects, animations, interactivity with JavaScript, and so on. A downside is loading SVG can be slow. SVG does offer benefits too.

Major advantages of SVG images:

1. Scalability : An SVG file is used for any type of graphics that you might need to scale. Usually, if you try to modify different image sizes, you'll run into pixelation issues. SVG files, on the other hand, it can be scaled to any size without loss of quality, because you're working with vectors instead of pixels.

2. They're indexed by Google : SVG files appear in Google image searches, which mean they don't pose any downsides from a Search Engine Optimization (SEO) perspective.

3. Interactivity: SVG supports interactivity through scripting languages like JavaScript. Elements within an SVG image can be made responsive to user actions, such as clicks or hover events, allowing for interactive and dynamic graphics.

4. Animation: SVG supports animation, allowing elements within the image to be animated and transformed over time. This enables the creation of visually engaging and interactive graphics, such as animated icons or data visualizations.

5. Accessibility: SVG images can be made accessible by adding textual descriptions or alternative text, making them usable for people with disabilities who rely on screen readers or assistive technologies.

6. Compact file size: SVG files are generally compact in size compared to other image formats. They can be compressed effectively using tools like GZIP, resulting in faster loading times for web pages.

SVG is widely used for a variety of purposes on the web, including icons, logos, illustrations, data visualizations, and more. It provides a powerful and versatile way to create visually appealing and interactive graphics that can adapt to different screen sizes and devices.

Viewing SVG Files:

There are several ways to open SVG files depending on what you want to do with them.

1. Web Browsers : Unlike other graphics files, when you open an SVG, it'll launch in your default browser. That's because your computer recognizes SVGs not as graphics, but as XML files.Most modern web browsers, such as Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge, have built-in support for SVG

2. Text Editors and IDE's : Since SVG files are XML-based images, you can open and modify them using a text editor. IDEs like Visual Studio Code, Sublime Text, or Atom offer plugins or extensions that enable live preview and editing of SVG files within the development environment.

3. Image Editors : Most modern photo editing software such as Adobe Illustrator, Adobe Photoshop, Inkscape, or GIMP have the capability to open and edit SVG files. That means you can open, edit them, and then save the changes or export the image to other formats

Remember that SVG files are essentially text files, so you can also open and view them using a simple text editor.

Embedding SVG in HTML5:

There are several ways to embed SVG in HTML5 , which depends on user requirement

1. Using <img> element: To embed an SVG via an <img> element t, you just need to reference it in the **src** attribute as you'd expect. You will need a height or a width attribute (or both if your SVG has no inherent aspect ratio). For example :

```
<img    src="Triangle.svg"
        alt="triangle with all three sides equal"
        height="87"      width="100"  />
```

Advantage of using <img> tag for svg :

- Quick, familiar image syntax with built-in text equivalent available in the alt attribute
- You can make the image into a hyperlink easily by nesting the **<img>** inside an **<a>** element.
- The SVG file can be cached by the browser, resulting in faster loading times for any page that uses the same image in the future

Disadvantage of using <img> tag for svg :

- You cannot manipulate the image with JavaScript.
- If you want to control the SVG content with CSS, you must include inline CSS styles in your SVG code. (External stylesheets invoked from the SVG file take no effect.)
- You cannot restyle the image with CSS pseudo-classes (like :focus).

2. Using CSS background-image : The older browser will give support for .png,.jpg files as background whereas the newer browser will load SVG files as background. For example : **background-image : url("image.svg") ;**
Like the method described above, inserting SVGs using CSS background images means that the SVG can't be manipulated with JavaScript, and is also subject to the same CSS limitations.

3. Using <SVG> element: This is sometimes called putting your SVG inline, or inlining SVG. Make sure your SVG code snippet begins with an **<svg>** start tag and ends with an **</svg>** end tag. And the svg code is directly placed within the **<svg>** element. The height and width attributes specify the dimensions of the SVG viewport.

For Example :

```
<svg  width="300"  height="200" >
        <rect  width="100%"   height="100%"   fill="green"  />
</svg>
```

Advantage of using <svg> element:

- Putting your SVG inline saves an HTTP request, and therefore can reduce your loading time a bit.
- You can assign classes and ids to SVG elements and style them with CSS, either within the SVG or wherever you put the CSS style rules for your HTML document. In fact, you can use any SVG presentation attribute as a CSS property.
- Inlining SVG is the only approach that lets you use CSS interactions (like :focus) and CSS animations on your SVG image (even in your regular stylesheet.)
- You can make SVG markup into a hyperlink by wrapping it in an **<a>** element.

Disadvantage of using <svg> element:

- This method is only suitable if you're using the SVG in only one place. Duplication makes for resource-intensive maintenance.
- Extra SVG code increases the size of your HTML file. If the SVG content is complex or contains a large number of elements. This can impact the overall page load time

- The browser cannot cache inline SVG as it would cache regular image assets, So each time you load the page it takes time due to inline svg.
- You may include fallback image in a element, but browsers that support SVG still download any fallback images.

4. Embedding an SVG with an iframe : You can open SVG images in your browser just like webpages. So embedding an SVG document with an **<iframe>** is done just by using **src** attribute of iframe. For example :

```
<iframe  src="triangle.svg"  width="500"  height="500">
   <img  src=”triangle.jpg”  alt=”Triangle image” />
</iframe>
```

Disadvantage :

- iframes do have a fallback mechanism, as you can see, but browsers only display the fallback if they lack support for iframes altogether.
- Moreover, unless the SVG and your current webpage have the same origin, you cannot use JavaScript on your main webpage to manipulate the SVG.

5. Embedding SVG as an <object> : You can also use an HTML **<object>** element to add SVG images to a webpage using the code syntax below:

<object data="happy.svg" width="300" height="300"> </object>

You use the **data** attribute to specify the URL of the resource that you'll use by the object, which is the SVG image in our case. You use the width and height to specify the size of the SVG image. Which is similar to the **<img>** element.

6. Embedding SVG as an <embed> : The HTML **<embed>** element is another way to use an SVG image in HTML5 using the syntax : **<embed src="happy.svg" />**

However, that this method has limitations, too. According to MDN, most modern browsers have deprecated and removed support for browser plug-ins. This means that relying upon **<embed>** is generally not wise if you want your site to be operable on the average user's browser

SVG Shapes :

SVG (Scalable Vector Graphics) supports various shape elements that allow you to create geometric shapes and paths. SVG has some predefined shape elements that can be used by developers:

- Rectangle **<rect>**
- Circle **<circle>**
- Ellipse **<ellipse>**
- Line **<line>**
- Polyline **<polyline>**
- Polygon **<polygon>**
- Path **<path>**

These shape elements can be customized using various SVG attributes, such as `fill` (to set the fill color), `stroke` (to set the outline color), `stroke-width` (to specify the outline thickness), and more. Additionally, you can apply transformations, gradients, patterns, and other styling options to enhance the appearance of SVG shapes.

By combining these shape elements and utilizing SVG's capabilities, you can create complex and visually appealing graphics and illustrations that scale without losing quality across different devices and screen resolutions.

SVG Rectangle :

The **<rect>** element is applied to create an SVG rectangle and variations of rectangle figures. There are six attributes determine the rectangle's shape and position on the screen:

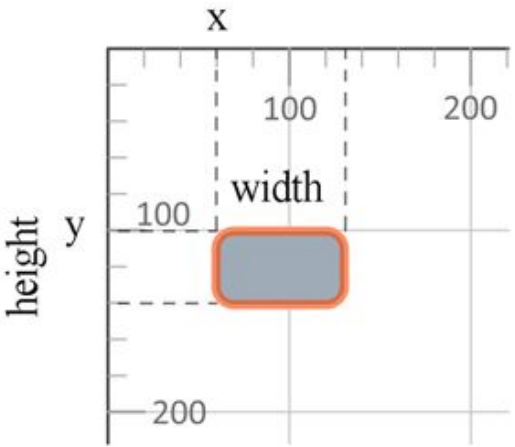
x	The x position of the top left corner of the rectangle.
y	The y position of the top left corner of the rectangle.
width	The width of the rectangle.
height	The height of the rectangle.
rx	The x radius of the corners of the rectangle.
ry	The y radius of the corners of the rectangle.

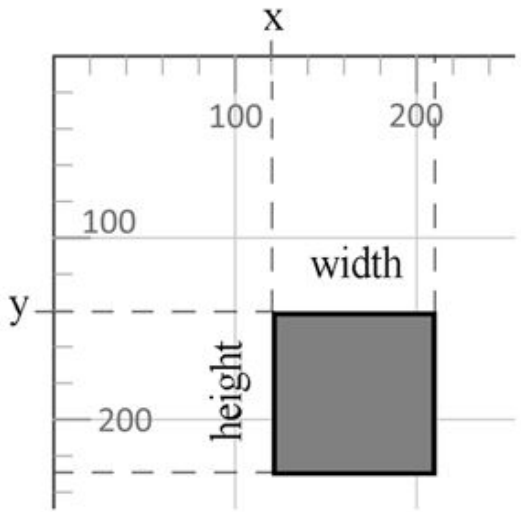
If **x and y attributes are not set**, the top-left corner of the rectangle is placed at the point **(0,0).** If rx and ry radii are not specified, they default to 0. You can fill the rectangle with a color, make the filling transparent, and style the stroke using the **style** attribute properties

The SVG code to generate the SVG rectangle looks like (**rectangle1.svg**):

```
<svg width="500" height="550" xmlns="http://www.w3.org/2000/svg">
    <rect x="60" y="100" width="70" height="40" rx="10" ry="10"
    style="fill:#778899; stroke:#FF4500; stroke-width:5; fill-opacity:0.7; stroke-opacity:0.6" />
</svg>
```

The above code example shows that you create an SVG rectangle with the top-left point at coordinates (60,100), the width="70", and the height="40". It has rounded edges and the stroke-width:5. All units are in pixels.





Example2: Without rounded borders

```
<svg width="500" height="550" >
  <rect x="120" y="140" width="90"
    height="90" style="fill:grey;
    stroke-width:5; stroke:rgb(0,0,0)"/>
</svg>
```

Style Properties in Shape Elements :

fill – defines the fill color of the rectangle

fill-opacity – determines the transparent of the rectangle

stroke-width – determines the width of the stroke

stroke – determines the color of the stroke

stroke-opacity – determines the transparent of the stroke

In the CSS fill and stroke properties the color can be set in several ways:

1. **fill: blue** - color is taken from CSS color names. All modern browsers support the 140+ color names.
2. **fill: rgb(0,0,255)** - color is written in RGB color model (rgb values).
3. **fill: #0000ff** - color is written in RGB color model (hex rgb values).

rgb (red, green, blue) is an additive color model that describes how any color is encoded using three basic ones. The values r, g and b are the intensity (in the range from 0 to 255), respectively, of the red, green and blue components of the determined color. That is, a bright blue color can be defined as (0,0,255), red as (255,0,0), bright green - (0,255,0), black - (0,0,0), and white - (255,255,255)

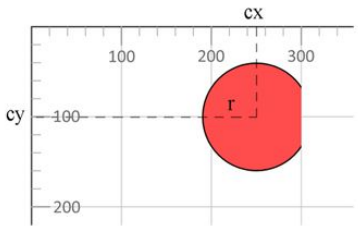
SVG Circle :

The SVG **<circle>** element is used to draw a circle on the screen. You need to set the position of the SVG circle's center and radius. These are cx, cy, and r attributes, respectively. You can set the stroke and fill properties for an SVG circle.

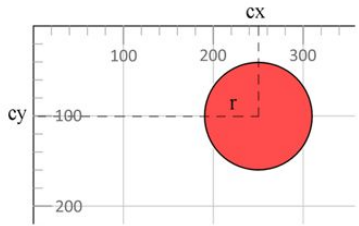
r	The radius of the circle.
cx	The x position of the center of the circle.
cy	The y position of the center of the circle.

Example1 :

```
<svg width="300" height="550" >
  <circle cx="250" cy="100" r="60"
    style="fill:red; stroke-width:3;
    stroke:rgb(0,0,0); fill-opacity:0.7" />
</svg>
```



the circle is not fully visible in the viewport



the circle is fully visible

Example2 :

```
<svg width="320" height="550" >
  <circle cx="250" cy="100" r="60"
  style="fill:red; stroke-width:3;
  stroke:rgb(0,0,0); fill-opacity:0.7" />
</svg>
```

In the Example1, the circle image does not entirely fit into the viewport. The viewport's width is 300, i.e. on the x-axis, it is cropped by 300 pixels distance. And for the full SVG circle viewing, 310-pixel wide window is needed (cx+r=250+60=310). To make the circle fully visible, you need to increase the width of the viewport until 310 or more as in Example 2.

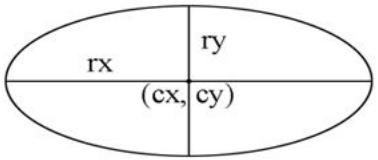
SVG Ellipse :

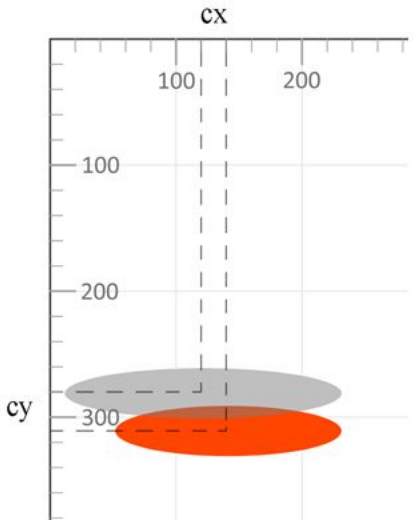
An ellipse is a more general figure than a circle. In the context of an **<ellipse>** element creating, the values of a semi-major axis rx, a semi-minor axis ry, and the coordinates of their intercrossing point (cx, cy) are indicated.

rx , ry	x and y radii of the ellipse (semi-major axis and semi-minor axis)
cx , cy	x and y coordinates of the center of the ellipse

Eg:
```
<svg  width="500" xmlns="http://www.w3.org/2000/svg">
    <ellipse cx="140"  cy="310"  rx="90"  ry="20"  style="fill:OrangeRed"
    <ellipse cx="120"  cy="280"  rx="110"  ry="20"  style="fill:grey; fill-op
</svg>
```

The second SVG ellipse in the code has transparency 50% and will be displayed over the first. A rule about the order of the SVG elements showing is: the later elements in the code are displayed on top of the previous ones.





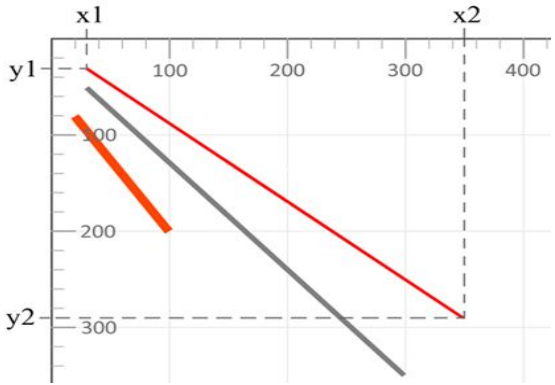
SVG Line :

The `<line>` element takes the positions of two points as parameters and draws a straight line between them. Which can have the following attributes:

x1 , y1	the x, y coordinates of the origin point
x2, y2	the x, y coordinates of the end point

Example:

```
<svg width="500" height="550" xmlns="http://www.w3.org/2000/svg">
    <line x1="30" y1="30" x2="350" y2="290" style="stroke:rgb(255,0,0); stroke-width:3" />
    <line x1="30" y1="50" x2="300" y2="350" style="stroke:grey; stroke-width:5" />
    <line x1="20" y1="80" x2="100" y2="200" style="stroke:orangered; stroke-width:8" />
</svg>
```

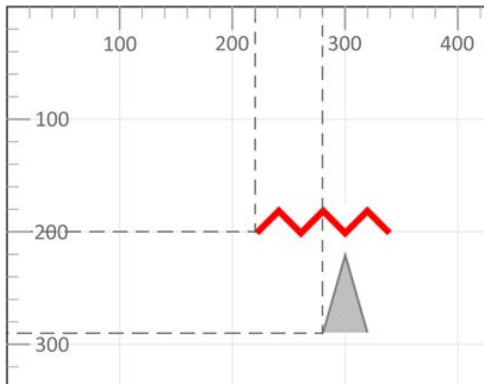


SVG Polyline :

The SVG **<polyline>** element is used to draw multiple connected straight lines. As a rule, polylines are open forms, the beginning of the first and the end of the last line do not match. The **points** attribute specifies the x, y coordinates of the points at which the polyline bends. **The first group of two numbers in the points defines the coordinates of the beginning of the first line, the second group defines the end of the first line and at the same time the beginning of the second line**, etc.

Example:

```
<svg width="500" height="550" xmlns="http://www.w3.org/2000/svg">
  <polyline points="280,290 300,220 320,290" style="fill:grey; stroke:grey; stroke-width:2; fill-opacity:0.5" />
  <polyline points="220,200 240,180 260,200 280,180 300,200 320,180 340,200" style="fill:none;
   stroke:red; stroke-width:6" />
</svg>
```



In the first SVG polyline example, there are 3 points that define a triangle. The space between the points will be filled with the **fill** property. In the example, the fill color is grey: style="fill:grey". **The default fill color is black.** In the second example, seven points are connected by the SVG polyline with the stroke-width:6 and the fill property **none** .

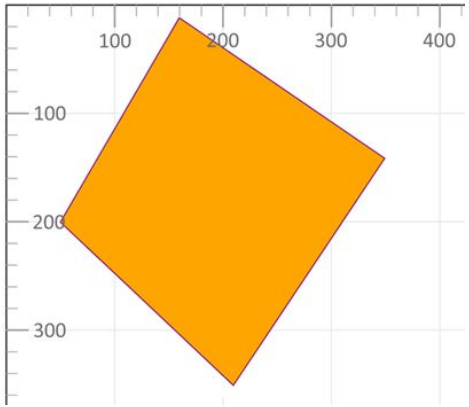
SVG Polygon

A polygon is a plane geometric shape formed by a closed polyline. If the polyline does not have self-crossing points, the polygon is simple. For example, triangles and squares are simple polygons, but a five-pointed star is not.

The **<polygon>** element is used to create a shape that contains at least three sides. The attribute points defines the coordinates (x, y) for each corner of the SVG polygon.

Example :

```
<svg width="500"  height="550" xmlns="http://www.w3.org/2000/svg">
   <polygon points="160,10 350,140 210,350 50,199"
   style="fill:orange;stroke: purple;stroke-width:1" />
</svg>
```



SVG Path Data

A **<path>** is the most general shape that can be used in SVG. Using a `path` element, you can draw rectangles (with or without rounded corners), circles, ellipses, polylines, and polygons. Basically any of the other types of shapes, bezier curves, quadratic curves, and many more.

The shape of a **<path>** element is defined by one parameter : **d**. The **d** attribute contains a series of commands and parameters used by those commands. All of these 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 10px up and 7px to the left from the last point*).

The following groups of commands in here to d attribute:

There are five line commands for **<path>** nodes used with d attributes

- **moveto (M, m)**
- **lineto (L, l, H, h, V, v)**
- **closepath (Z, z)**

The **moveTo(M, m)** command sets the origin point for SVG path drawing. The commands group, that draw straight line segments includes the **lineto (L, l, H, h, V and v)** and the **closepath (Z and z)** commands

The following three groups of commands draw curves:

- **cubic Bézier curve (C, c, S, s)**
- **quadratic Bézier curve (Q, q, T, t)**
- **elliptical Arc (A, a)**

Coordinates are always written without units specifying and refer to the user's coordinate system. Usually, they are in pixels. The path is described by the position of the current point - a —**virtual pen**. The "**pen**" moves along the path sections from the starting to the endpoint. These points are the key parameters for all drawing commands. **Each command has parameters; they are indicated in brackets**.

Lines and Paths :

Any SVG path begins with the **moveto M (x,y)** command. x and y coordinates indicate the current point where the path should start.

Three lineto commands draw straight lines from the current point to the new one:
L (x, y) - the command takes two parameters - x and y coordinates of a point, and draws a line from the current position to this point (x, y).
H (x) - draws a horizontal line from the current position with x coordinate. The y coordinate does not change the value.
V (y) - draws a vertical line from the current position to point with y coordinate. The x coordinate does not change the value.

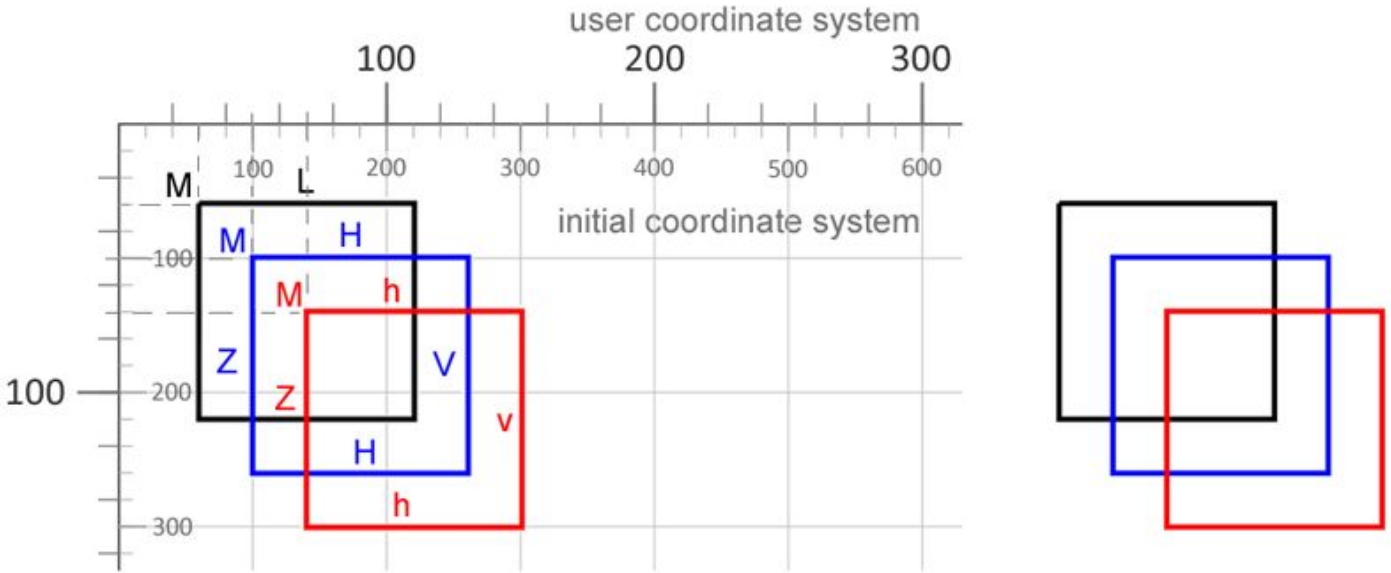
*The **H and V** commands only use **one** argument since they only move in one direction.*

After doing any command, the —virtual pen point will be located at the endpoint of that drawing command. The next drawing command will start from this point.

Closepath Z ends the current SVG path, returning it to the starting point. The Z command draws a straight line from the current position back to the first point in the path. *The command has no parameters.*

Let’s draw a square using the lineto commands :

```
<svg   height="400"  width="400"   viewBox="0 0 200 200" >
        <path d="M 30 30   L 110 30   L 110 110   L 30 110   L 30 30"  fill="transparent"  stroke-width="2"  stroke="black" />
        <path d="M 50 50   H 130   V 130   H 50   Z"  fill="transparent"  stroke-width="2"  stroke="blue" />
        <path d="M 70 70   h 80     v 80     h -80   Z"  fill="transparent"  stroke-width="2"  stroke="red" />
</svg>
```



SVG Arc :

SVG (Scalable Vector Graphics) Arc refers to the arc element in SVG, which is used to create elliptical or circular arcs. An arc is ***a portion of an ellipse or a circle*****.** In SVG, arcs are defined by their start and end points, along with additional parameters that control the shape and orientation of the arc.

The basic syntax for creating an SVG arc is as follows:

<path d="M x1 y1 A rx ry x-axis-rotation large-arc-flag sweep-flag x2 y2" />

`M x1 y1` **:** This represents the starting point of the arc. The arc will begin at coordinates `x1` and `y1`.

Let's break down the attributes used in the Arc **A** command:

- **`rx`** **and** **`ry`** represent the radii of the ellipse. If `rx` and `ry` are equal, the arc will be circular. Otherwise, it will be elliptical.
- **`x-axis-rotation`** is an optional attribute that allows you to rotate the arc along the x-axis. It is measured in degrees.
- **`large-arc-flag`** determines which of the two possible arcs between the start and end points should be drawn. If set to 1, the longer arc will be used; if set to 0, the shorter arc will be used.

- **`sweep-flag`** determines the direction in which the arc is drawn. If set to 1, the arc will be drawn in a positive angle direction (clockwise); if set to 0, it will be drawn in a negative angle direction (counterclockwise).
- **`x2` and `y2`** represent the ending point of the arc. The arc will end at coordinates `x2` and `y2`.

The a command is the same as A but interprets the coordinates relative to current —pen point.

```
<svg height="500" width="700" viewBox="0 0 100 100">
    <path d="M10,20  A 30,30  0 0,0  40,70"  style="stroke:#FFA500;  stroke-width:1;  fill:none" />
    <path d="M10,20  A 30,30  0 1 0  40,70"  style="stroke: #FF0000;  stroke-width:1;  fill:none" />
    <path d="M10,20  A 30,30  0 0 0  40,70  A 30,30  0 1 1  10,20"  style="stroke: #FFA500; stroke-width:1; fill:#FFD700"
     transform="translate(70,0)" />
</svg>
```

Bézier Curves :

Bezier curves are an important tool for computer graphics programs. Bezier curves provide flexibility in creating smooth and precise curves, making them a fundamental tool in digital design and illustration. There are different types of Bezier curves, including **Quadratic and Cubic Bezier curves.**

The most commonly used type is the **Cubic Bezier curve**, which is defined by : a starting point and ***two additional control points,*** an ending point.

Syntax : `C x1 y1, x2 y2, x y`

Quadratic Bezier curves are similar to cubic Bezier curves, but they have *:* a starting point and ***one control point,*** an ending point. The formula for a quadratic Bezier curve is simpler, and it follows a similar structure as the cubic Bezier curve.

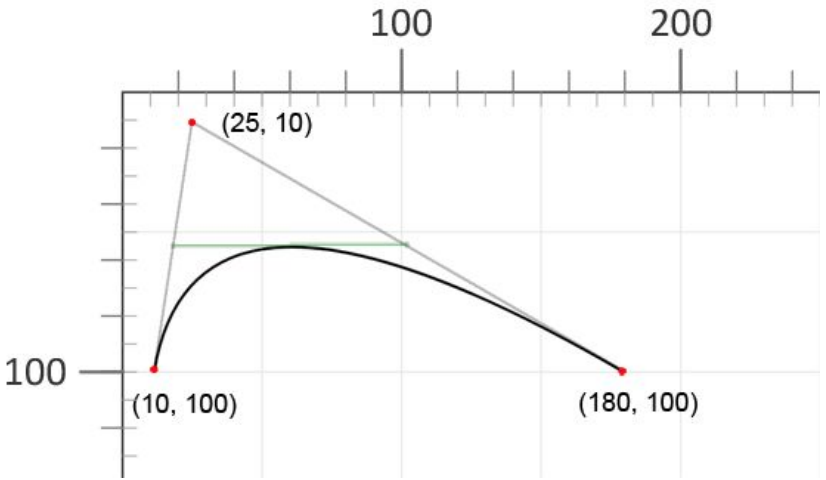
Syntax : `Q x1 y1 , x y`

Quadratic Bézier Curves (Q,T):

Any Bezier curve as the current (starting) point takes the pen's location after the executing of the previous command. The **Q** command of the quadratic curve is specified by only two points: the control point **(x1, y1)** and the endpoint of the curve **(x, y).** The **q** command is also given by two points, the coordinates of which are relative to the current point.

Let's consider an example :

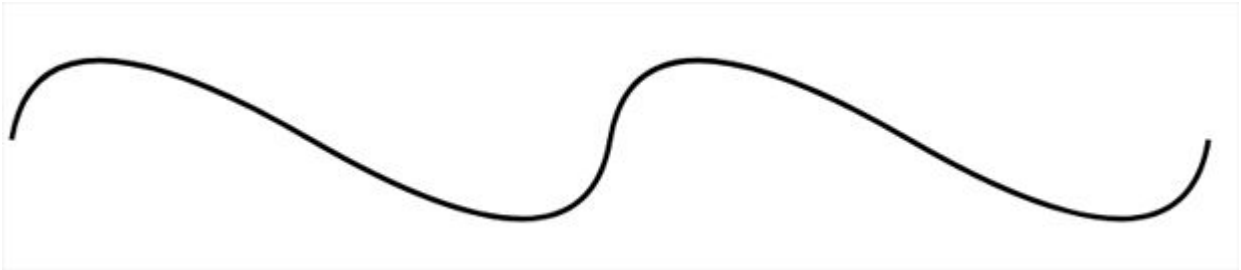
```
<svg width="600" height="600" viewBox="0 0 200 200">
    <path d="M 10 100 Q 25 10 180 100" stroke="black" stroke-width="1" fill="transparent" />
</svg>
```



Several **Q** commands can be used sequentially for extended curves stringing, but the resulting *curve may not be smooth.* You can combine multiple quadratic Béziers ***without losing smoothness*** *with the help of* **T** command.

The **T** command draws smooth quadratic Bézier curve from the current point to **endpoint (x,y).** The command's parameters are only the endpoint coordinates **(x,y).** The **T** command indicates that the control point should be **mirrored from the previous control point**. This works if the last command was a **Q or a T.** At the end of the T command, the final **(x,y)** coordinate pair becomes the new current point used in the polybézier. Below is an example of a curve created using the **T** command. Here, the **x** coordinates of the curve segments are equidistant, **y** coordinate does not change.

```
<svg width="700" height="600">
    <path d="M 10 100  Q 25 10 180 100  T 350 100  T 520 100  T 690 100"  stroke="black"  stroke-width="3"  fill="none" />
</svg>
```



Cubic Bezier Curve :

Cubic Bézier curve is more complex than a quadratic one. Two control points describe the appearance of the curve at its beginning and the end. To create a cubic Bezier curve, you need to specify **three sets of coordinates in the C command:** the coordinates of two control points **(x1 y1, x2 y2)** and the endpoint of the curve **(x y)**:

The syntax for creating a Cubic Bezier curve in SVG is as follows:

```
C x1 y1, x2 y2, x y
```

Let's break down the attributes used in the cubic Bezier curve:

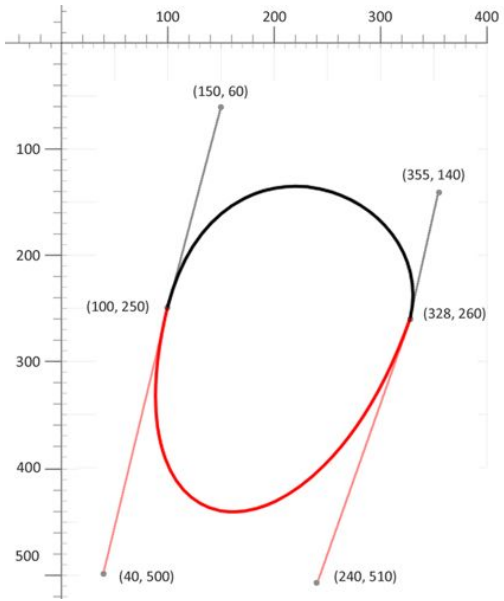
- `(x1, y1)` is the first control point, which determines the direction of the curve near the starting point.
- `(x2, y2)` is the second control point, which determines the direction of the curve near the ending point.
- `(x, y)` is the ending point of the curve.

The control points position determines the slope of the tangent line at the start and endpoint. The Bézier function creates a smooth curve that paths from the slope set at the beginning of the line to the slope at the curve end. You can specify several C commands in the one <path> element; they will be realized sequentially. The endpoint of the first C command becomes the starting point for the new C command.

The following code example makes a shape using two paths

```
<svg height="700" width="750">
  <path d="M 100 250 C 150 60 355 140 328 260 " stroke="black" stroke-width="3" fill="none" />
  <path d="M 100 250 C 40 500 240 510 328 260" stroke="red" stroke-width="3" fill="none" />
</svg>
```

On the figure, different paths are shown in black and red



In the sample we have connected in the one path two curves with the help of C commands. The curves binding in this way may result in a loss of smoothness at the connection points.

For smooth long curves creating you may use a shortcut version of the cubic Bézier, designated by the **S x2 y2, x y** command. It allows to string together multiple cubic Béziers similar to the T command for the quadratic Béziers.For the S command, the first control point is considered a reflection of the previous one, that is necessary for a constant slope and smooth connection of the curves. The second control point (x2 y2) and the endpoint (x y) coordinates must be specified.

To summarize basic SVG path commands are:

Command	Meaning	Parameters
M	Establish origin at point specified	2 Parameters giving absolute(x,y) current positions
L	Straight line path from current position to point specified	2 Parameters giving absolute(x,y) position of the line endpoint which becomes the current position.
H	Horizontal line path from current position to point specified	Single Parameter giving absolute X-coordinate of the endpoint. The Y-coordinate is same as the previous current position. The point becomes the current position.
V	Vertical line path from current position to point specified	Single Parameter giving absolute Y-coordinate of the endpoint. The X-coordinate is same as the previous current position. The point becomes the current position.
Z	Straight line back to original Move origin	No Parameters.

SVG Text Content Elements :

A text content element is an SVG element that causes text to be rendered on the canvas.

The SVG text content elements are **: <text>, <tspan> and <textPath>.**

i) SVG text element <text> :

The **<text>** element is used to define a text. **x** and **y** are the main attributes responsible for the text position. The **baseline** for the text begins from the bottom-left corner of the first text symbol. ***It is essential to set y value larger than the font size.*** Otherwise, the text does not get into the viewport.

The following example illustrates how to specify a start of baseline correctly. The **x and y** set the coordinates of the baseline beginning :

```
<svg height="100" width="200" xmlns="http://www.w3.org/2000/svg">
        <text  x="10"  y="6"  fill="red"> The text is not fully visible </text>
        <text  x="10"  y="30" fill="green">The text is fully visible </text>
 </svg>
```

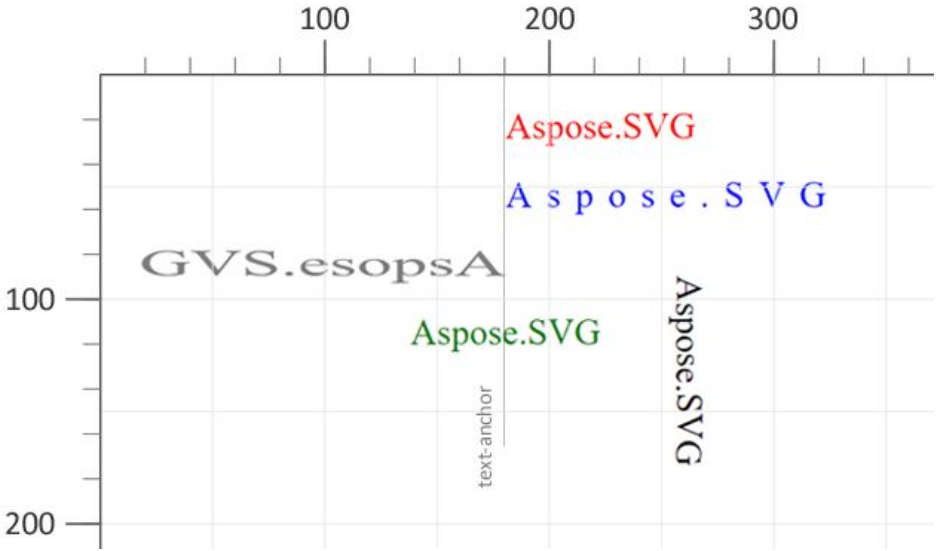
The attributes of the **<text> and <tspan>** elements indicate ***writing direction, alignment, font,*** and other specifying properties and features that precisely describe how to render characters. The main attributes are:

x, y − the absolute x and y coordinates of characters
dx, dy − shift along the x-axis or y-axis (relative coordinates)
rotate − rotation applied to all characters
textlength − rendering length of the text
lengthAdjust − type of adjustment with the rendered length of the text

The **x, y, dx, dy, and rotate** attributes in the **<text> and <tspan>** elements are often used for **individual characters** that require minor position adjustments to achieve a visual effect.

In the example, the **x** and **y** set the start coordinates of the **baseline**. Using the **textLength** attribute, you can set the length of the text. Herewith the text length is then adjusted to match the specified length by adjusting the spacing and the size of the glyphs (graphical representation of font). With the **lengthAdjust** attribute, you can specify whether to adjust both the letter spacing and the glyph size (stretching effect).

```
<svg height="300" width="400"  xmlns="http://www.w3.org/2000/svg">
        <text x="180"  y="30"  fill="red">Aspose.SVG</text>
        <text x="180"  y="60"  fill="blue"  textLength="140" >Aspose.SVG</text>
        <text x="180"  y="90"  fill="grey"  textLength="160" lengthAdjust="spacingAndGlyphs"
        style="direction: rtl; unicode-bidi: bidi-override">Aspose.SVG</text>
        <text x="180"  y="120"  fill="green"  style="text-anchor:middle" >Aspose.SVG</text>
        <text x="260"  y="90"  style="writing-mode:tb">Aspose.SVG</text>
</svg>
```



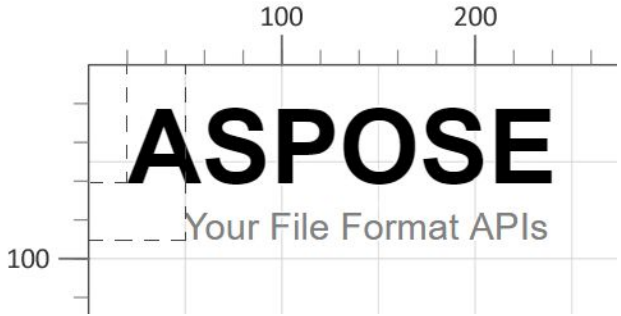
ii) SVG tspan element <tspan> :

The **<tspan>** element is within the **<text>** element or other **<tspan>** element. Being a child element, **<tspan>** serves several important functions in text displaying and formatting:

1. In SVG, the text is rendered as a single line inside a viewport; it does not automatically wrap to another string. You may break it into multiple lines using the **<tspan>** element. Each **<tspan>** element can contain different formatting and position.
2. For styles (or other attributes) setting and applying only to a specific part of the text, you need to use the **<tspan>** element. It allows you to switch the style or position of the displayed text within the **<tspan>** element relative to the parent element.

Consider a simple <tspan> example :

```
<svg height="300" width="600" xmlns="http://www.w3.org/2000/svg">
  <text x="20" y="60" style="font-family:arial">
    <tspan style="font-weight:bold;  font-size:55px">ASPOSE</tspan>
    <tspan x="50" y="90" style="font-size:20px; fill:grey">Your File Format APIs </tspan>
  </text>
</svg>
```



iii) SVG textPath element :

In SVG, text can be displayed not only horizontally or vertically but along any vector curve.
SVG can place text along a path defined by a **<path>** element. This is making by a **<textPath>** element in a few ways:

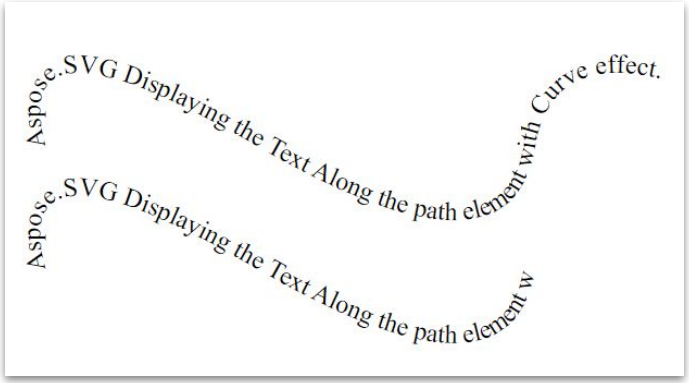
- Attribute **href** (**xlink:href**) references to an URL pointing to the <path> element.
- Attribute **path** specifies the SVG path data directly.

Both the path attribute and the href attribute specify a path along which the characters will be rendered. The text displaying along the curve mostly takes attribute href with reference to the <path>element.

The SVG text can be styled using CSS properties like **font-weight, font-style, text-decoration, text-transform**, etc. It can be decorated with an **underline, overline, and strike-through**. Like other SVG shapes, text can have both a **stroke** and **fill** set on it.

Here is an example :

```
<svg height="300" width="800" xmlns="http://www.w3.org/2000/svg">
   <path id="path1" d="M 50 100 Q 25 10 180 100 T 350 100 T 520 100 T 690 100"  fill="transparent" />
   <path id="path2" d="M 50 100 Q 25 10 180 100 T 350 100"  transform="translate(0,75)" fill="transparent" />
   <text>
        <textPath href="#path1"> Aspose.SVG Displaying the Text Along the path element with Curve effect. </textPath>
        <textPath href="#path2"> Aspose.SVG Displaying the Text Along the path element with Curve effect. </textPath>
   </text>
</svg>
```



If the length of the path is shorter than the text size, then only the text part that is within the extent of the path is drawn. In the figure, the second curve is shorter than the text length, so text breaks off at the path end.

SVG Transformations

SVG allows modifying graphic elements using translation, rotation, scaling, and skewing. All these SVG transformations refer to the geometric kind. SVG objects can be altering using the **transform attribute's** properties:

translate, scale, rotate, skewX, skewY, and matrix.

Translate :

The translation moves all the object points at the same distance along parallel lines. This can be interpreted as shifting the origin of the element's system of coordinates. There are three transform functions: **translateX (tx), translateY (ty) and translate (tx, ty).** The translate (tx, ty) function moves an element by a tx value along the x-axis and by ty along the y-axis. If one of the values is not specified, it defaults to zero.

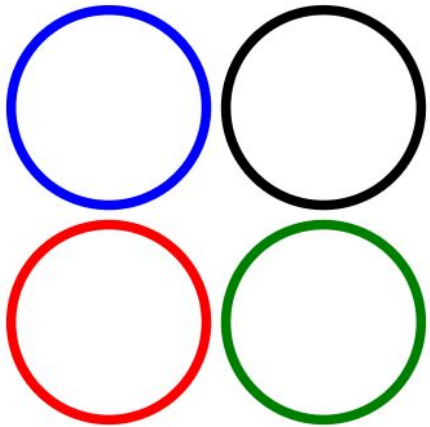
The action of attribute transform="translate(tx, ty)" means the changing of object coordinates according to the formula:

x(new) = x(old) + tx

y(new) = y(old) + ty

Here is a simple example:

```
<svg viewBox="0 0 100 100" xmlns="https://www.w3.org/2000/svg">
    <g fill="none">
        <circle  cx="15"  cy="15"  r="10"  stroke="blue" />
        <circle  cx="15"  cy="15"  r="10"  stroke="black"  transform="translate(22)" />
        <circle  cx="15"  cy="15"  r="10"  stroke="red"     transform="translate(0,22)" />
        <circle  cx="15"  cy="15"  r="10"  stroke="green"  transform="translate(22,22)" />
    </g>
</svg>
```

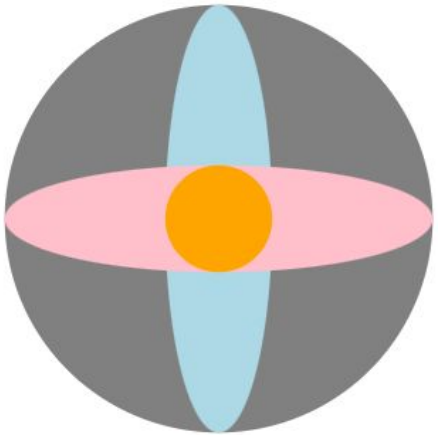


Scale :

Scaling is an SVG transformation that enlarges or reduces an object using a scaling factor. You have to distinguish the uniform and directional scaling. The **scale (sx, sy)** transform function allows scaling image along the **x and y axis**. This function takes one or two values which specify horizontal and vertical scaling: **transform="scale(sx, sy)"**. The **sy** scaling factor value is optional and if omitted it is assumed to be equal to **sx**.

Here is a simple example:

```
<svg viewBox="-50 -50 200 200" xmlns="https://www.w3.org/2000/svg">
  <g transform="translate(100)">
        <circle  cx="0"  cy="0"  r="10"  fill="grey"       transform="scale(4)" />
        <circle  cx="0"  cy="0"  r="10"  fill="lightblue"  transform="scale(1,4)" />
        <circle  cx="0"  cy="0"  r="10"  fill="pink"       transform="scale(4,1)" />
        <circle  cx="0"  cy="0"  r="10"  fill="orange" />
  </g>
</svg>
```

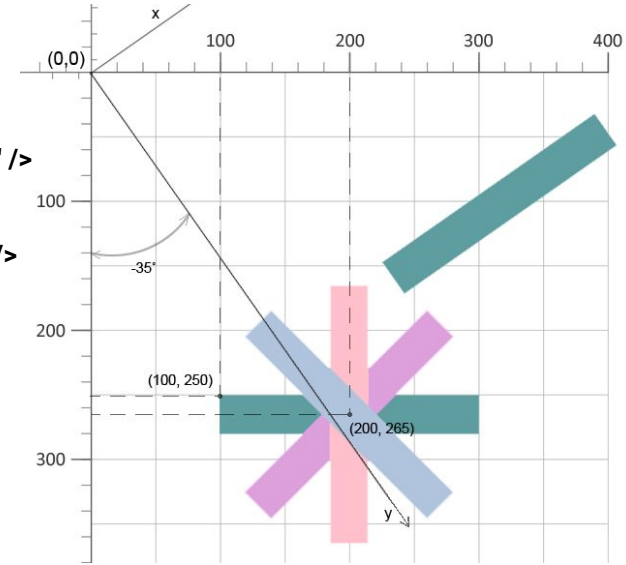


Rotate :

The **rotate(angle, cx, cy)** function rotates the element by angle around the point with coordinates **(cx, cy)**. If coordinates are not specified, then the rotation will be performed around the point **(0, 0) of the initial coordinate system.** All rotate or skew angle values should be specified in **degrees**, you cannot use the other units we have available in CSS. If we use a **positive** angle value, then the rotation will be **clockwise**, and conversely, a **negative** angle value gives us **counterclockwise** spin.

Here is a simple example:

```
<svg width="450" height="450" xmlns="https://www.w3.org/2000/svg">
  <rect x="100" y="250" width="200" height="30" fill="CadetBlue" />
  <rect x="100" y="250" width="200" height="30" fill="#DDA0DD" transform ="rotate(-45 200 265)" />
  <rect x="100" y="250" width="200" height="30" fill="Pink"    transform ="rotate(-90 200 265)" />
  <rect x="100" y="250" width="200" height="30" fill="#B0C4DE" transform ="rotate(45 200 265)" />
  <rect x="100" y="250" width="200" height="30" fill="CadetBlue" transform ="rotate(-35)" />
</svg>
```



Skew :

Skewing is a transformation that rotates one of the axes of the element's coordinate system by a certain angle **clockwise** or **counterclockwise**. SVG elements can be skewed through the use of the **skewX(angle) and skewY(angle)** functions. The angle value included within these functions represents a skew SVG transformation in **degrees** along the appropriate axis. The using **skewX(angle)**, only the x coordinate of the points of the shape changes, but the y coordinate remains unchanged.

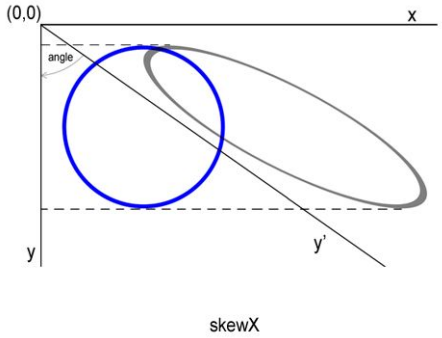
The skewX(angle) function makes the vertical lines look like they have been rotated by a given angle. The x coordinate of each point changes on a value proportional to the specified angle and distance to the origin.

Here is shown an example of the circle with skewX(55) value :

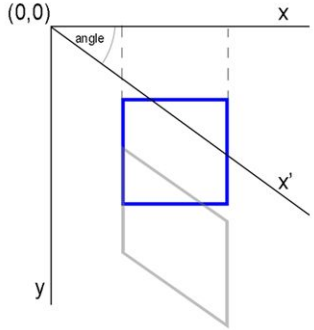
```
<svg viewBox="0 0 100 100" >
  <circle cx="20" cy="20" r="15" stroke="blue" fill="none" />
  <circle cx="20" cy="20" r="15" stroke="grey"
    stroke-opacity="0.7" fill="none" transform="skewX(55)" />
</svg>
```

A simple example of the rectangle skewed by skewY(35) :

```
<svg viewBox="0 0 200 200" >
  <rect x="20" y="20" width="30" height="30" stroke="blue" fill="none" />
  <rect x="20" y="20" width="30" height="30" stroke="grey"
    stroke-opacity="0.5" fill="none" transform="skewY(35)" />
</svg>
```



skewX



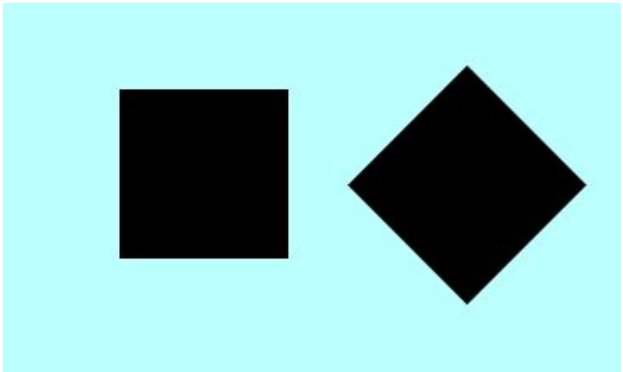
skewY

Multiple transformations :

Transformations can be concatenated easily just by separating them with spaces. For example, translate() and rotate() are common used transformations.

```
<svg width="40" height="50" style="background-color:#bff;">
        <rect  x="100"  y="100"  width="100"  height="100"  />
        <rect  x="100"  y="100"  width="100"  height="100"    transform="translate(200) rotate(45 120,100)" />
</svg>
```

This example shows Square which is translated in X axis with 200px and rotated 45^0 ate coordinate(120,100)



To summarize the transformation:

The possible transformations are:

Transformation	Meaning	Parameters
translate	Defines a translation of the coordinates	x and y defining the x and y translation
scale	Defines a scaling of the X and Y coordinates	sx and sy defining the scaling in the X and Y directions s defining the same scaling in the X and Y directions
rotate	Defines a rotation about a point	angle, x and y defining a clock-wise rotation of angle degrees about the point (x,y) angle defining a clock-wise rotation of angle degrees about the origin
skewX	Defines a skew along the X axis	angle degrees defining a skew of the X position by Y*tan(angle)
skewY	Defines a skew along the Y axis	angle degrees defining a skew of the Y position by X*tan(angle)

Fills and Strokes in SVG :

Filling and stroking are both painting operations. All graphical elements such as shapes, paths and text – are rendered by being filled. *The fill is painting the interior of the object, and the stroke is painting along its outline*.

The SVG stroke and SVG fill are some of the main CSS properties that can be set for any lines, text and shapes. In the SVG text file, they can be specified both in the style attribute and using **presentation attributes**.

SVG Fill and SVG Stroke Properties :

Colorization or painting means the operation of adding color, gradients or patterns to graphics using fill and stroke.

A set of **presentation attributes** define their properties:

fill, fill-opacity, fill-rule

stroke, stroke-dasharray, stroke-dashoffset, stroke-linecap, stroke-linejoin, stroke-miterlimit, stroke-opacity, and stroke-width.

SVG fills and SVG strokes features can be set in the **style attribute**.

The syntax for properties specifying is: **style="stroke:#00ff00; stroke-width:2; fill:#ff0000"**

But the same style properties can be given in the **presentation attributes** with such the syntax:

stroke="green" stroke-width="2" fill="#ff0000"

Fill attribute :

For the SVG color specifying, you can take **color names, rgb values, hex values, etc.**

The fill attribute colors the interior of a graphic element. When you fill an SVG shape or curve, the fill colourizes open paths too as if the last its point was connected to the first, even though the stroke color in that part of the path will not appear.

If the fill attribute property is not specified, t**he default is black.** So that there is no filling, you need to specify the attribute value **fill="none" or fill= "transparent".**

SVG Lines and Stroke Caps :

The more often used stroke properties are the following: **stroke, stroke-width, stroke-linecap, and stroke-linejoin.**

They define the color, thickness, types of line endings to an open path, and kind of join of the two meet lines.

For any line, it is possible to set the shape of its ends. This makes sense if the line has the **stroke-width** property.

For Example :

```
<svg height="200" width="800" xmlns="http://www.w3.org/2000/svg">
    <g stroke="grey" stroke-width="30">
        <path stroke-linecap="butt"   d="M 300 60  l 215 0" />
        <path stroke-linecap="round"  d="M 300 100 l 215 0" />
        <path stroke-linecap="square" d="M 300 140 l 215 0" />
    </g>
</svg>
```

The **stroke-linecap** CSS attribute defines how the ends of an SVG line are rendered, and has three possible values: **butt, square and round.**

- As a result of **butt** using, the stroke cap is cut off with a straight edge that is normal exactly where the line ends.
- The value **square** results in a stroke cap that looks like a cut off, but it extends slightly beyond where the line ends. The distance that the stroke goes beyond the path is half of the stroke-width value.
- The value **round** means the stroke cap has the round ends, which radius depends by the stroke-width.

One more sample illustrates a **stroke-linejoin** property :

```
<svg width="400" height="400" xmlns="http://www.w3.org/2000/svg">
    <g stroke-width="20" fill="none" stroke="black">
        <polyline points="40 60   80  20  120  60  160  10 "   stroke-linecap="butt"     stroke-linejoin="miter" />
        <polyline points="40 140 80 100 120 140 160 100 " stroke-linecap="square" stroke-linejoin="round" />
        <polyline points="40 220 80 180 120 220 160 180"  stroke-linecap="round"   stroke-linejoin="bevel" />
  </g>
</svg>
```

The **stroke-linejoin** attribute can take a three value: **miter, round, and bevel.**

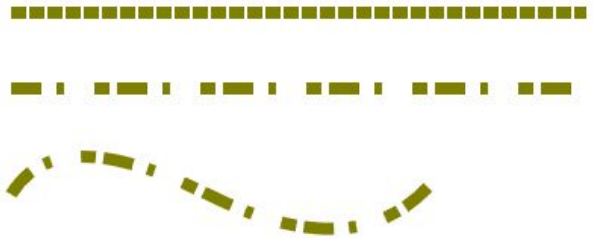
- **miter** extends the stroke to create a square corner in the lines joinpoint;
- **round** builds a rounded stroke in the joining end;
- **bevel** makes a newangle to aid in the transition between the two lines.

Dots and Dashes, Stroke-dasharray:

All the SVG stroke properties can be applied to any kind of lines, text and outlines of elements like a circle, rectangle, etc. The **stroke-dasharray** property in CSS is for creating dashes in the stroke of SVG shapes. The stroke-dasharray attribute converts paths into dashes and makes the stroke of an SVG shape rendered with dashed lines.

The values that stroke- dasharray attribute takes are an array of commas or whitespace-separated numbers. These values define the **length of dashes and spaces.** Eg:

```
<svg width="400" height="300">
        <g style="stroke:olive; fill:none; stroke-width:8;" >
        <line x1="20" y1="30" x2="400" y2="30" stroke-dasharray="10 2" />
        <line x1="20" y1="80" x2="400" y2="80" stroke-dasharray="20 10 5" />
        <path d="M 20 150 Q 50 100 150 150  T 300 140" stroke-dasharray="20 10 5" />
        </g>
</svg>
```



In Stroke-dasharray, we specify even number of values: each of number couples means **filled-unfilled** areas. The default values are in pixels. If an odd number of values is specified, the list is then repeated to produce an even number of values. For example, 20 10 5 becomes 20 10 5 20 10 5.

```
<svg height="600" width="600" xmlns="http://www.w3.org/2000/svg">
   <g fill="none" stroke-width="50" stroke="olive">
        <circle cx="100" cy="100" r="25"  stroke-dasharray="4,4" />
        <circle cx="100" cy="100" r="25"  stroke-dasharray="4"     transform="translate(150,0)" />
        <circle cx="100" cy="100" r="25"  stroke-dasharray="4,1,3" transform="translate(300,0)" />
        <rect x="80" y="200" width="80" height="80" stroke-dasharray="2 2" />
        <text x="200" y="250" font-family="arial" font-size="50"  stroke-width="3" stroke-dasharray="2 2">I love SVG!</text>
   </g>
</svg>
```

SVG Color :

Using color is an important part of creating SVG. You can colorize SVG shapes, lines, paths, text. SVG graphics can be colorized, textured, shaded, or built from partially-transparent overlapping layers.

SVG Color Specifying

Colorization or painting means the operation of adding color, gradients or patterns to SVG graphics using **fill and stroke**. Filling and stroking are both painting operations. The SVG stroke and SVG fill are some of the main CSS properties that can be set for any lines, text and shapes.

In an SVG file, **fill and stroke** can be specified both in the style attribute (fill and stroke properties) and using fill and stroke attributes as presentation attributes.

So you can set color for SVG elements in two ways:

- using fill and stroke properties of the *style attribute*.
- using presentation attributes *fill and stroke*.

Fill and Stroke Properties of the Style Attribute :

SVG fill and SVG stroke features can be set in the style attribute. The syntax for properties specifying is:

style=" stroke-width:2; stroke:green; fill:#ff0000 "

Fill and Stroke attributes (presentation attributes) :

SVG fills and SVG strokes can be given presentation fill and stroke attributes with such the syntax:

stroke-width="2" stroke="green" fill="#ff0000”

SVG Color Codes :

To specify an SVG color, you can take *color names, RGB or RGBA values, HEX values, HSL or HSLA values*. The following examples will use different ways to set fill and stroke characteristics :

1. SVG Color Names. There are the 147 color names defined by the Scalable Vector Graphics (SVG) Specification. You may set named colors like this: stroke="Green" or fill="Red".

2. HEX Color Codes. The code is expressed as follows: #RRGGBB, where each of the two-digit values is a range of each of the three colors (red, green, blue), Each two-digit hex pair can have a value from 00 to FF. By the combination of these there colors we can finalize a color. For example, #00FF00 is displayed as green, because the green component is set to its maximum value (FF) and the others are set to 00.

3. RGB(Red, Blue, Green) Color Codes. The values R, G and B are the intensity (in the range from 0 to 255), respectively, of the red, green and blue components of the determined color. You can set the green and red RGB colors like this: stroke="rgb(0,255,0)" or fill="rgb(255,0,0)".

4. RGBA(Red, Blue, Green, Alpha) Color Codes. RGBA color values are an extension of **RGB** color values with an alpha channel that determines the opacity of the color. The alpha parameter is a number between 0.0 and 1.0 that specifies transparency. You may determine the green and red RGB colors like this:
stroke="rgba(0,255,0,1.0)" or fill="rgba(255,0,0,0.5)".

5. HSL Color Codes. HSL stands for Hue, Saturation and Lightness. Each color has an angle on the RGB color wheel and a percentage value for the saturation and lightness values. HSL codes for green and red colors you can set like this: stroke="hsl(120, 100%, 50%)" or fill="hsl(0, 100%,50%)"

6. HSLA(Hue, Saturation, Lightness, Alpha) Color Codes. HSLA color values are an extension of **HSL** color values with an alpha channel that determines the opacity of the color. HSL codes for green and red colors you can set like this: stroke="hsla(120, 100%, 50%, 1.0)" and fill="hsla(0,100%, 50%, 0.5)"

Color Specifying Rules :

- If the **fill** attribute (or fill property of the **style** attribute) **is not specified, the default is black**.
- If the **fill** attribute (or fill property of the style attribute) has **none or transparent** value, the shapes filling is transparent.
- If the **stroke** attribute (or stroke property of the style attribute) is not specified, **the stroke is invisible, is absent.** This remains true even if the stroke-width attribute is specified.
- To specify fill color or **stroke** color, you can use color names, RGB or RGBA values, HEX values, HSL or HSLA values. Also, you can take gradients and patterns (see the Text Color section or the SVG Filters and Gradients article).

SVG Filters :

The advantage of SVG filters is the ability to combine multiple types.The results obtained after applying one filter can be the source of the image for another filter. Required attributes for filter primitive are **x, y, width, and height.**

- Each filter primitive can take **one or two inputs** and **output only one result.**
- The input data of the filter primitive is specified in the **in** attribute.
- The default is **in=”SourceGraphic”.** The output of the operation is defined in the result attribute.

i) Gaussian Blur : (<feGaussianBlur>)

The Gaussian blur function is obtained by blurring & smoothing an image using the Gaussian function. It can be considered as a filter that reduces image noise and minor details.

Designers and photographers often apply Gaussian blur functions for different purposes:

- Gaussian blur can **soften the image,** it is useful for reducing chromatic aberration,
- It can hide a license plate or brand logo you don’t have permission to use, etc.

The **<feGaussianBlur>** filter creates a soft blur effect. It is a frequently used effect. The **stdDeviation attribute** specifies the number that characterizes the standard deviation for the blur operation.

If two numbers are provided, the **first number** represents a *standard deviation value along the x-axis* of the coordinate system, the second one - on *the y-axis*.

Example :

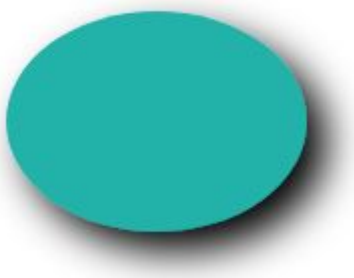
```
<svg height="400" width="600" xmlns="http://www.w3.org/2000/svg">
  <defs>
        <filter id="f2" x="-20" y="-20" height="100" width="100">
            <feGaussianBlur in="SourceGraphic" stdDeviation="10, 0" />
        </filter>
  </defs>
  <g stroke="none" fill="#20B2AA">
        <ellipse cx="60"  cy="80" rx="55" ry="35" />
        <ellipse cx="200" cy="80" rx="55" ry="35" filter="url(#f2)" />
   </g>
</svg>
```

ii) Drop Shadow Effect : (<feOffset>)

A typical example of the drop shadow effect is to add a 3D look to a shape or text. Also, a drop shadow effect can be used to make an image stand out. The **<feOffset>** filter primitive is used to offset a layer in SVG. In addition to the **in and result** attributes, this primitive accepts two main attributes - **dx and dy,** which define the distance the layer is offset along the x and y axes, respectively. The **<feBlend>** filter blends two objects; its mode attribute defines the blending mode.

Let's see how to create a drop shadow effect :

```
<svg height="200" width="200" xmlns="http://www.w3.org/2000/svg">
  <defs>
        <filter id="shadow" x="-20" y="-20" height="150" width="150">
           <feOffset result="offset" in="SourceAlpha" dx="10" dy="10" />
           <feGaussianBlur result="blur" in="offset" stdDeviation="10" />
           <feBlend in="SourceGraphic" in2="blur" mode="normal" />
        </filter>
   </defs>
   <ellipse cx="95" cy="90" rx="75" ry="55" fill="#20B2AA" filter="url(#shadow)"/>
</svg>
```



Thee filters are used to create drop shadow effect:

- **<feOffset>** takes in="SourceAlpha", and shifts the result 10 px to the right and 10px to the bottom, and stores the result in the buffer as "offset". Note, the alpha channel of the shape is used as input. The value SourceAlpha leads to a black-color result.
- **<feGaussianBlur>** takes in="offset", applies a blur of 10, and stores the result in a temporary buffer named "blur".
- **<feBlend>** takes two inputs in="SourceGraphic" and in2="blur" then blends the SourceGraphic on top of the offset black blurred image.

iii) <u>SVG Light Effects :</u>

The lighting effect is making in SVG using a set of filters. Consider some of them:

<feDiffuseLighting>, **<feSpecularLighting>**, and **<fePointLight>.**

The **<fePointLight>** filter defines a light source which sets a point light effect. It can be used within the <feDiffuseLighting> or <feSpecularLighting> primitive as a child.

Specific attributes x, y, and z indicate the position of the point light source. Both <feDiffuseLighting> and the <feSpecularLighting> filter light an image using its alpha channel as a bump map. The difference between them is the various calculations of the lighting model components

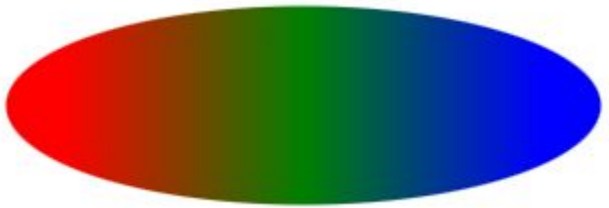
iv) Linear Gradient in SVG (<linearGradient>):

The linear gradient vector connects starting and ending points onto which the gradient stops are mapped. The attributes x1, y1, x2 and y2 set the linear gradient vector. Their values can be either numbers or percentages.

- The **<linearGradient>** has nested children **<stop>** elements that control the colors used in the gradient.
- Each color is specified with the <stop> tag.
- An **offset** attribute of the **<stop>** element indicates where the gradient stop is placed. For linear gradients, it represents a location along the gradient vector.

Example :

```
<svg height="250" width="700" xmlns="http://www.w3.org/2000/svg">
  <defs>
      <linearGradient id="grad1" x1="0%" y1="0%" x2="100%" y2="0%">
        <stop  offset="10%" style="stop-color:red" />
         <stop offset="50%" style="stop-color:green" />
         <stop offset="90%" style="stop-color:blue" />
      </linearGradient>
  </defs>
  <ellipse cx="300" cy="170" rx="165" ry="55"  fill="url(#grad1)" />
</svg>
```



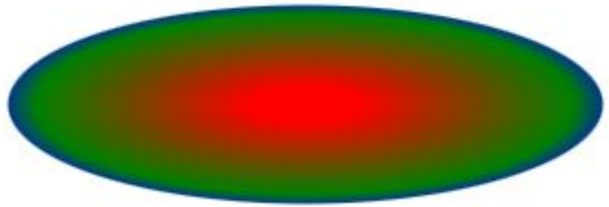
v) Radial Gradient in SVG (<radialGradient>) :

A radial gradient is more difficult than a linear one. The colors change **circularly** rather than linearly in it.

The attributes **cx, cy and r** define the outermost circle for the radial gradient : **cx** and **cy** mark the center of this circle. Their values specify as percentages of width and height of shape to fill. Both defaults to 50% if omitted. The **fx and fy** attributes define the innermost circle for the radial gradient. This is the point at which the SVG gradient «radiates».

Example :

```
<svg height="250" width="700" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <radialGradient id="grad1" cx="0.5" cy="0.5" r="0.9" fx="0.5" fy="0.5" spreadMethod="pad">
      <stop offset="10%" style="stop-color:red" />
      <stop offset="50%" style="stop-color:green" />
      <stop offset="60%" style="stop-color:blue" />
    </radialGradient>
  </defs>
  <ellipse cx="300" cy="170" rx="165" ry="55" fill="url(#grad1)" />
</svg>
```



SVG –Grouping : <g>

Frequently there is a need to group drawing elements together for one reason or another. Grouping in SVG is achieved by the **g** element. A set of elements can be defined as a group by enclosing them within a **<g>** element.

Benefits of grouping are:

1. One reason is if a set of elements share the same attribute.
2. To define a new coordinate system for a set of elements by applying a transformation to each coordinate specified in a set of elements.
3. Grouping is also useful as the source or destination of a reference.

Example :

```
<g style="fill:red;stroke:black">
        <circle cx="70" cy="100" r="50" />
        <rect x="150" y="50" rx="20" ry="20" width="135" height="100" />
        <line x1="325" y1="150" x2="375" y2="50" />
        <polyline points="50, 250 75, 350 100, 250 175, 350" />
        <polygon points=" 250, 250 297, 284 279, 340 202, 284" />
        <ellipse cx="400" cy="300" rx="72" ry="50" />
</g>
```

The g element can have any of the attributes or style properties defined for it that are generally applicable to individual drawing elements. In the example above, all the basic shapes will be rendered with the interior red and the border black.

SVG viewBox :

The viewBox attribute defines the **position and dimension, in user space,** of an SVG viewport.

- The value of the viewBox attribute is a list of four numbers: **min-x, min-y, width and height.**
- The numbers **min-x and min-y** represent the top left coordinates of the viewport.
- The numbers width and height represent its dimensions.

These numbers, which are separated by **whitespace and/or a comma,** specify a rectangle in user space which is mapped to the bounds of the viewport established for the associated SVG element

Example :

```
<svg width="400" height="300" viewBox="0 0 800 600">
    <rect x="100" y="100" width="600" height="400" fill="blue" />
</svg>
```

In this example, the initial size of the SVG element is 400x300 pixels, but the actual content within the SVG is **scaled based on the viewBox values.** The rectangle's position and size are defined in relation to the viewBox coordinate system, allowing it to be displayed properly regardless of the initial SVG dimensions.