Community Experience Distilled

Learning Raphaël JS Vector Graphics

Over 70 code examples to create vector graphics and data visualizations!

Damian Dawber

[PACKT] PUBLISHING open source* community experience distilled

Learning Raphaël JS Vector Graphics

Over 70 code examples to create vector graphics and data visualizations!

Damian Dawber

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Learning Raphaël JS Vector Graphics

First published: May 2013

Production Reference: 1100513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-916-1

www.packtpub.com

Cover Image by Sujay Gawand (sujaygawand@gmail.com)

Credits

Author
Damian Dawber

Reviewers
Charles Thomas
Lee Turver

Acquisition Editor
Mary Jasmine Nadar

Commissioning Editor
Harsha Bharwani

Technical Editor
Jalasha D'costa

Project Coordinator
Sneha Modi

Proofreader
Mario Cecere

Indexer
Rekha Nair

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Damian Dawber is a web developer working on medium- to large-scale e-commerce websites and bespoke web and mobile applications. He works with both frontend and server-side technologies having had exposure to a wide range of projects working on behalf of small and medium enterprises through to FTSE 100 companies.

He started his career after being exposed to programming as it used to solve problems in physics and decided he wanted to write code on a daily basis thereafter.

Naturally I would like to thank everyone who has in some way shaped my personal and career development up to now. In particular, I would like to thank my family, my colleagues, and my current and former employers for giving me the opportunity to develop in this industry.

Special thanks must go to this book's technical reviewers, Charles Thomas and Lee Turver. Their input has been most valuable.

I would also like to thank Tomás Alabés for providing me with content pertaining to the future of the Raphaël library.

About the Reviewers

After leaving school and having spent significant periods in occupations as diverse as a trainee accountant and soldier, **Charles Thomas** supplied a software solution for a company he worked for as a laboratory supervisor in 1990. When he was made unemployed by his company in 1992, he completed further software training (HNC in Computer Studies in 1994).

Since 1994, he has always been engaged in projects that are on the "bleeding edge of technology" and he enjoys doing these very much. During his career he has worked globally for different companies in England, Europe, Canada, and USA as a software engineer (Some of these companies include the European Bank, GE (Amsterdam), Associated Newspapers / Daily Mail, and HSBC). He now lives and works in the UK and enjoys working as a graphical frontend developer on contract as required by interested companies.

He additionally did a smaller review for *Instant RaphaelJS Starter* by Packt Publishing.

I would like to thank Tomás Alabés (the main collaborator of the latest Raphaël release) for his contribution that provided information on the future of Raphaël, which appears in the last chapter of the book.

Lee Turver is a PHP developer based in Manchester with close to 10 years' experience working with various web-related technologies. Now working for one of the leading digital agencies in the North West, building high-quality solutions for high-profile clients.

You can follow Lee on Twitter `@LeeTurver`.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Learning Raphaël JS Vector Graphics takes you on a tour of the Raphaël JavaScript library and its potential applicability in today's online landscape. While it is a "mini", it is by no means limited. Coverage is given to all of the library's main facets and applications thereof and on completion of this book you will be able to create vector drawings and interactive applications in the browser.

What this book covers

Chapter 1, *The Graphical Web*, gives an introduction to Raphaël: what it is, its applications, and its relevance in the context of the Web.

Chapter 2, *Basic Drawing with Raphaël*, looks at how we can draw and embellish the fundamental graphics elements.

Chapter 3, *Drawing Paths*, covers drawing and manipulating more complex shapes using paths.

Chapter 4, *Transformations and Event Handling*, covers transforming graphics elements, basic event handling, the Events library, and the drag-and-drop functionality.

Chapter 5, *Vector Animation*, looks at bringing our elements to life by way of animation. We look at different ways in which we can animate graphics and lead by example.

Chapter 6, *Working with Existing SVGs*, explains how we can use Raphaël to work with the existing SVGs with an overview of Inkscape and a discussion on choropleth maps.

Chapter 7, *Creating a Suite of Social Media Visualizations*, looks at building four social media data visualization demos based on everything we have covered in the previous chapters.

Who this book is for

This book is for any and all developers with an interest in frontend technologies. It is perfectly suited to those with a desire to create interactive applications or vector drawings in the web browser.

It is assumed that the reader has a basic understanding of or familiarity with JavaScript.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The equivalent path using the lowercase variant of the command would be `"M 50,150 q 175,-130 350,0"` where the `(x, y)` and `(x1, y1)` parameters are the relative distances…"

A block of code is set as follows:

```
var bBox = path.getBBox(); // width/height of icon
var xTranslation = (bBox.width + 4) * i;
var yTranslation = (bBox.height + 4) * j;
path.transform(['T', xTranslation, yTranslation]);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
paths[i].animate({
    subtend: [360, cx, cy, r]
}, interval,  function() {
    paths[i].remove();
    setTimeout(function() {
    update(i);
});
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The demos for each of the code samples in this book can be found at `http://raphaeljsvectorgraphics.com`. Readers are strongly encouraged to open the demos alongside reading the book in order to best relate to the content covered.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output/result.

You can download this file from `http://www.packtpub.com/sites/default/files/downloads/9161OS_Images.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
The Graphical Web

The ability to draw graphics in the web browser is important in today's online landscape. While the need to effectively represent information has long existed in the empirical sciences, mathematics, and print media, it has taken on a new lease of life since the advent of the Web. With the colossal volumes of data that we now generate and the existence of a plethora of devices by which we can communicate with users, there has been a raise in both the desire and ability to visualize data online. In addition, the browser has grown from being a convenient entry point into a world of information to a truly interactive media outlet.

There are a number of technologies that have emerged to enable the process of browser-based drawing including, but not limited to VML and SVG, HTML5 Canvas, and WebGL. Each is generally suited to a particular purpose—2D vector drawing in the case of VML and SVG, bitmap drawing in the case of HTML5 Canvas, and 3D graphics rendering in the case of WebGL.

This book covers 2D vector drawing using **Raphaël**, a lightweight, open source, JavaScript library that uses SVG and VML to draw vector graphics in the browser. It provides a number of convenient methods for path drawing, vector transformations, interactivity via the DOM and path animation, and supports bitmap image embedding and text drawing. The library greatly simplifies drawing through these methods, acting as an *adapter* to the underlying (SVG and VML) specifications.

While an appreciation of the SVG specification will be sought throughout this book insofar as it will help us harness the full power of the library, an intimate understanding of the workings of SVG is not required and VML is not covered at all. Raphaël takes care of outputting graphics at the SVG/VML specification level, which means that we only need to concern ourselves with utilizing the library itself.

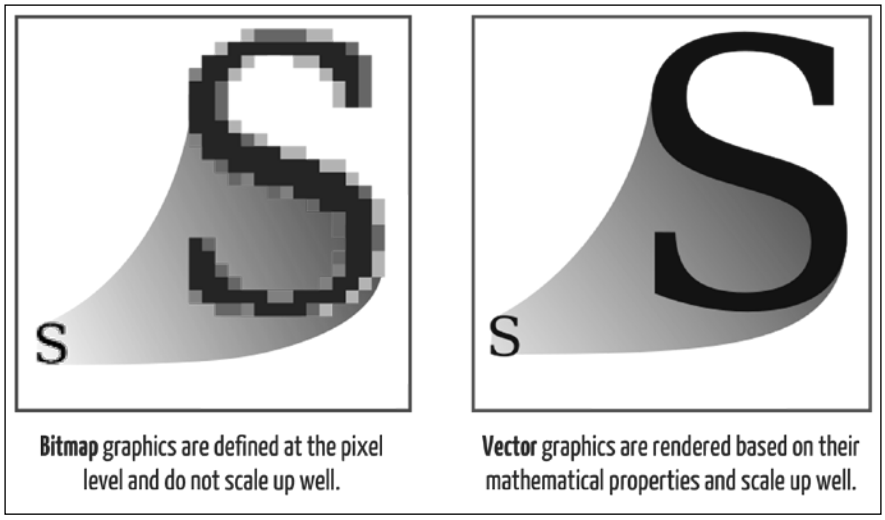
The library's creator and JavaScript virtuoso, **Dmitry Baranovskiy**, has done a great job documenting Raphaël at `http://raphaeljs.com/` and we will make extensive reference to the documentation throughout the course of this book in order to concoct quality bespoke vector graphics in the browser.

In this chapter we will:

- Discuss vector drawing on the Web
- Talk more about Raphaël and the SVG specification
- Look at some of the potential applications of Raphaël
- Download the Raphaël library

Vector drawing on the Web

Vector graphics are composed of a number of primitive geometric objects such as points, lines, shapes, and polygons. Each individual object is represented by a mathematical expression rather than some fixed physical point. Scaling vector graphics means changing the mathematical properties of the constituent objects and not the objects themselves and quality is not lost as a result. Fixed point, or *bitmap*, graphics are constrained to a particular resolution meaning that when they are scaled up the quality is lost:



Source: Wikimedia Commons
(`http://en.wikipedia.org/wiki/File:Bitmap_VS_SVG.svg`)

All graphics are ultimately rendered at the fixed point, or *pixel*, level. Your display is, after all, the sum of many millions of these points. The difference between bitmap and vector graphics is that in the case of vector graphics, the rendering is done *at the very last moment*. Scaling up a bitmap means creating new pixels based on the existing ones whereas with vector graphics, pixels are created at the resolution at which the final graphic is rendered based on the abstract mathematical definition of the object.

Vector drawing on the Web has predominantly taken shape in the following two forms:

- **Vector Markup Language** (**VML**) devised by Microsoft and submitted to the W3C in 1998, describes a file format based on XML by which vector graphics can be created in Internet Explorer Versions 5 to 9
- **Scalable Vector Graphics** (**SVG**) put forward by the W3C themselves in 2001, describes an XML-based file format that has been adopted by all major modern browsers including Firefox, Google Chrome, Safari, and Opera

Indeed, Versions 9 and 10 of Internet Explorer offer native support for SVG and VML has been deprecated in favor of SVG.

Vector drawing libraries

There are a number of JavaScript libraries available for drawing vector graphics. In addition to Raphaël, two popular libraries are **Paper.js** (`http://paperjs.org`) and **D3.js** (`http://d3js.org`). Each library is best suited to different tasks and the one that you use will depend upon what you're trying to achieve.

The following tables summarizes the Raphaël, Paper.js, and D3.js libraries:

Library	IE 6, 7, 8 support	Out-of-the-box (generic) visualizations	Low-level (bespoke) visualizations
Raphaël	Yes	No	Yes
D3.js	No	Yes	No
Paper.js	No	No	No

Raphaël is particularly well suited to creating bespoke vector graphics since it provides a low-level API to the SVG specification. It is very adept at working with primitive vector graphics and offers scope for working with existing SVG images. D3.js makes creating data visualizations straightforward owing to its API being very much oriented to mathematical operations and data manipulation but does not offer the same level of flexibility as Raphaël when dealing with vector graphics at the lower levels. Paper.js uses the HTML5 Canvas for vector graphics drawing and so offers less in the way of manipulating vectors but does, however, have very good facilities for manipulating images and working with paths.

Raphaël is the only framework that supports the VML specification and falls back on using VML in browsers (IE 6, 7, and 8) that do not have native SVG support (technically Paper.js can work on older browsers but the project does not support them). Despite the best efforts of the developer community at large, supporting older browsers is still a requirement for many companies.

The SVG specification

SVG is the most widespread form of vector graphics specification implemented on the Web today. All modern browsers boast comprehensive support for the format and the latest mobile browsers offer at least basic support for SVG (including Safari 3.2 and higher, the Blackberry browser, and Chrome for Android).

SVG specifies an XML-based file format for creating vector graphics. For example, a basic path defining a black, solid line of width 1 from a point at *x=50*, *y=50* in some arbitrary coordinate system to *x=10*, *y=10* is described by the following code:

```
<path d="M50 50 L10 10" stroke-width="1" stroke="#000" />
```

In addition to paths, SVG supports shape drawing, text, fills and gradients, animation, and interaction whereby SVG images can respond to events triggered by a user. You can see some examples of SVG at `http://raphaeljsvectorgraphics.com/the-graphical-web/raw-svg/`.

Raphaël uses the XML-based language extensively and throughout this book you will become familiar with some of the syntax of SVG, especially in relation to path drawing.

Working with Raphaël rather than SVG directly

Browser support aside, one good reason for using Raphaël over writing raw SVG is that it makes vector drawing *easier*. Take for example, drawing a rectangle and animating its width from 50 pixels to 100 pixels using raw SVG:

```
<rect x="10" y="10" width="50" height="30">
    <animate attributeType="XML"
        attributeName="width"
        to="100"
        fill="freeze"
        dur="1s" />
</rect>
```

Here we draw a rectangle at (10, 10) with a width of 50 and a height of 30 using the `<rect>` element. The nested `<animate>` element then defines an animation over one second on the rectangle's width to a total width of 100, where `fill="freeze"` is used to retain the state of the rectangle after the animation is complete (otherwise its width would be reset to 50). This is fairly verbose given that what we are looking to achieve is quite straightforward. The equivalent animation in Raphaël is achieved by the following code:

```
var rect = paper.rect(10, 10, 50, 30);
rect.animate({
    width: 100
}, 1000);
```

Here, the `animate` and `rect` methods are more concise.

Raphaël also integrates well with other libraries, for example, jQuery, owing to being compact and appropriately namespaced (it creates just one global variable named `Raphael`).

Applications of Raphaël

Raphaël has been used in a variety of projects ranging from map visualization during the *Town Hall @ the White House* event (`http://askobama.twitter.com/`) to online games such as **Ebocs** (`http://www.dejapong.com/ebocs/`). The former usage demonstrates a popular application of the library, namely in cartography, while the latter is a less common but a viable application of the library for creating games with a low sprite (bitmap image) count.

Raphaël is particularly well suited to cartography as the library affords us the ability to add interactivity to DOM elements and to scale graphics up and down without the loss of quality. A nice example of this can be found at `http://raphaeljs.com/australia.html`, where an interactive choropleth map of Australia is demoed (we will look at creating choropleth maps in *Chapter 6*, *Working with Existing SVGs*, of this book).

There are other, more subtle, applications of the library. While CSS3 is becoming increasingly prevalent, many properties have not yet been unified across all browsers and there is little or no support in older browsers. In some situations, utilizing Raphaël to achieve effects to the same end might be more suitable for the project that you are working on. A demo that looks at transforming images can be found at `http://raphaeljsvectorgraphics.com/the-graphical-web/css3-raphael-transforms/`.

Downloading Raphaël

Throughout this book we will work with Version 2.1.0 of Raphaël, which is the latest version of the library at the time of writing. The latest version of library can be downloaded from `http://raphaeljs.com`. The project is hosted on GitHub at `https://github.com/DmitryBaranovskiy/raphael/` and older versions of the library are tagged at `https://github.com/DmitryBaranovskiy/raphael/tags` should you ever require them.

It is recommended that you download both the compressed and uncompressed versions of the library when beginning a new project. During development you should use the uncompressed version of Raphaël: you may want to refer to specific methods in the source code, and debug messages will make more sense in the uncompressed version. For production purposes, you should always use the compressed version. The compressed version is significantly smaller in file size and using it will reduce the load on your server and the response time of your pages.

You can reduce the size of Raphaël further by gzipping it. Your web server is usually responsible for gzipping static content. Apache's **mod_gzip**, for example, can compress HTML, text, CSS, and JavaScript files (for more information you should consult your web server's documentation).

Creating Raphaël JavaScript applications

The most important aspect of learning how to utilize a new technology is understanding the concepts driving it. Code samples will be used where appropriate throughout this book but reams of unnecessary code will be omitted in favor of concentrating on the core aspects of the task at hand. Where appropriate, you should refer to the source code and website (`http://raphaeljsvectorgraphics.com`) accompanying this book.

Project structure and optimization

The projects in this book will be fairly self-contained and so the project structure will be kept quite simple. As your own projects grow, you should pay careful attention to project structure.

You should also consider merging and compressing your CSS and JavaScript files when pushing your work into production. This will have the effect of increasing the responsiveness of your site or application and reducing load on your web server. Useful tools for achieving this are **minify** (`http://code.google.com/p/minify/`) and Google's **Closure Tools** (`https://developers.google.com/closure/`).

Summary

It is often remarked that vector drawing in the browser is an under-utilized technology. Developers are often impressed by demonstrations of the technology but are not always sure to what extend it may be used. Hopefully after reading this chapter you will have a better understanding of what Raphaël does and the types of problems it is able to solve.

In the next chapter, we will look at basic drawing with Raphaël and creating our first vector drawings in the browser.

2
Basic Drawing with Raphaël

Raphaël supports three fundamental types of graphics elements: shapes, images, and text. Shapes can be predefined shapes such as a rectangle, circle, or ellipse or they can be a combination of lines, curves, and paths. Vector graphics are by their very definition the sum of such shapes, and throughout the next two chapters we will create a number of interesting drawings based on them. Images are either bitmap images (such as `.png` or `.jpeg`) or existing SVG images, and can be manipulated by Raphaël in many of the ways that shapes can be.

Shapes and text can be painted, that is, filled and stroked (given an outline). Fills can be either a single color or a linear or radial gradient. Strokes can only be filled by a single color but the way in which a shape is stroked is modifiable.

Some examples of painted shapes can be seen at `http://raphaeljsvectorgraphics.com/basic-drawing/shapes-and-attributes`.

In this chapter we will:

- Look at the browser as the context in which we draw
- Draw basic shapes
- Apply fills, strokes, and other attributes to elements
- Apply gradient fills to elements
- Work with custom text and look at embedding custom fonts

The drawing context

In order to be able to draw graphics we need to define a space into which our content can be rendered. Like an artist, we need a canvas onto which we can draw. The visible region of the browser window, or **viewport**, defines a region in which we can draw our **canvas**.

The SVG specification refers to the drawing area itself as the viewport. Strictly speaking, a viewport is any rectangular viewing region. We will always refer to the browser window as the *viewport* and our drawing region as the *canvas* (not to be confused with the HTML5 Canvas).

We can create a canvas using the `Raphael` constructor as follows:

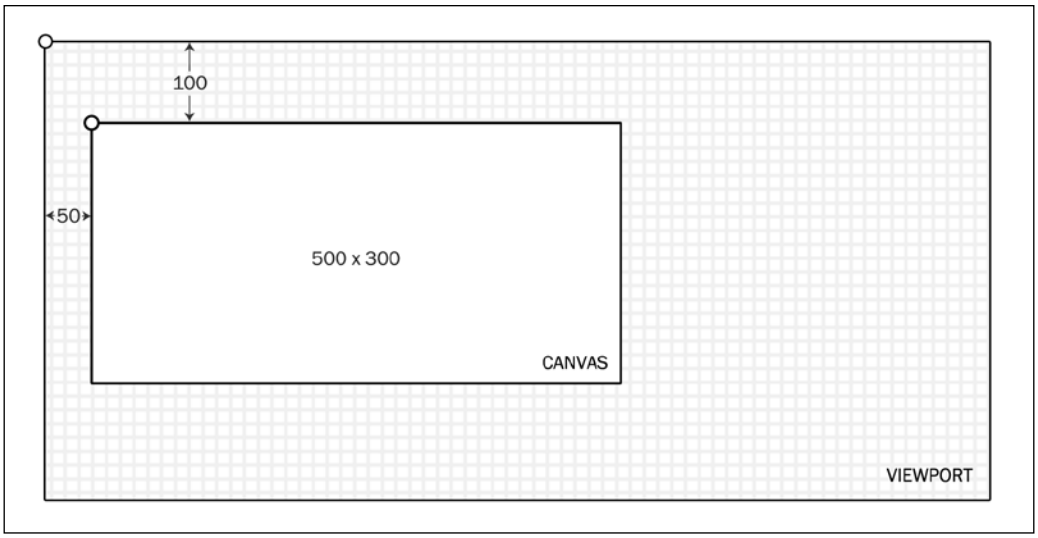
```
var paper = Raphael(50, 100, 500, 300);
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The demos for each of the code samples in this book can be found at `http://raphaeljsvectorgraphics.com`. Readers are strongly encouraged to open the demos alongside reading the book in order to best relate to the content covered.

This creates a canvas of width 500px and height 300px at 50px to the left of the viewport and 100px from the top of the viewport as shown in the following diagram. The `paper` variable holds a reference to a `Paper` object which is our canvas and we will use it for all future drawing:



More commonly we will want to use an existing DOM element as a container for our canvas rather than the viewport. In this case, the left-/right- (x-/y-) offsets are not required as the canvas will always be rendered at the top-left corner of the element. Consider the element:

```
<div id="my-canvas"></div>
```

We would create our 500px wide, 300px high canvas using the following code:

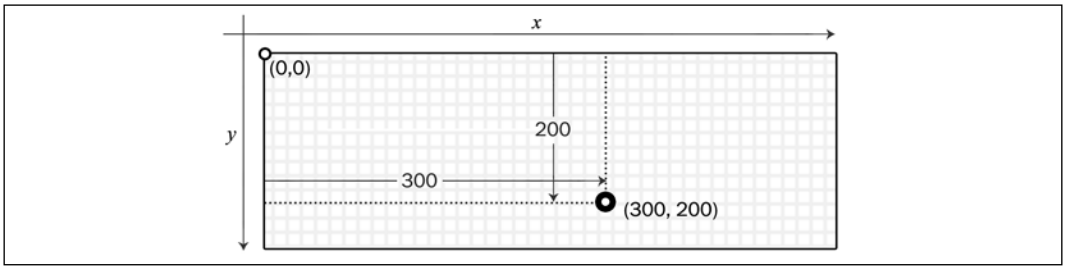
```
var paper = Raphael('my-canvas', 500, 300);
```

When using this form of the constructor, we always pass an element ID when passing a string as the first parameter. Alternatively, we can pass a DOM element as the first parameter like so:

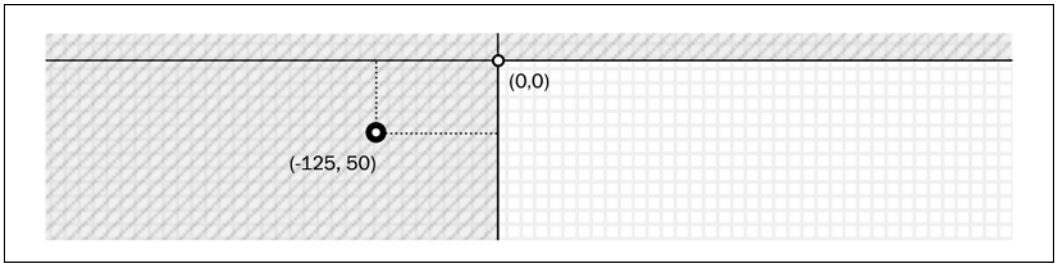
```
Raphael(document.getElementById('my-canvas'), 500, 300);
```

Canvas coordinates

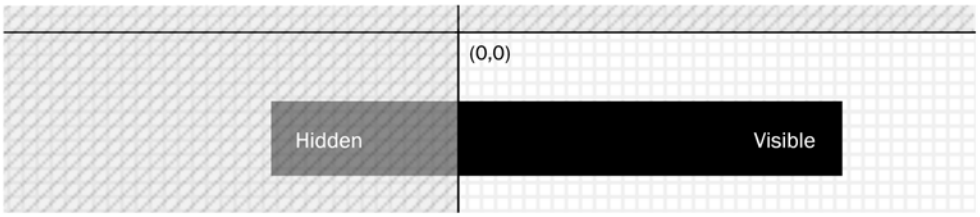
The following diagram illustrates the default coordinate system of the canvas created using the `Raphael` constructor. The **origin** of the canvas is defined as the point at which *x* and *y* are both equal to zero. A point on the canvas at *x = 300* and *y = 200*, or (300, 200) in vector notation, is the point 300px to the right of the origin *x*-point and 200px down from the origin *y*-point as shown:



Likewise, a point at (-125, 50) is that relative to the origin in *x* and *y* as shown in this diagram:



Note that graphics drawn at negative *x* or *y* points will be drawn starting from outside of the visible region of the canvas meaning that they will be partially or fully hidden from the viewer as illustrated in the following diagram:



Drawing basic shapes

Raphaël provides the `rect`, `circle`, and `ellipse` methods for drawing basic shapes. These are methods of the `Paper` object that is returned when we create our canvas. The `rect` method for example has the following syntax (taken from the Raphaël documentation at `http://raphaeljs.com/reference.html#Paper.rect`):

```
Paper.rect(x, y, width, height, [r]);
```

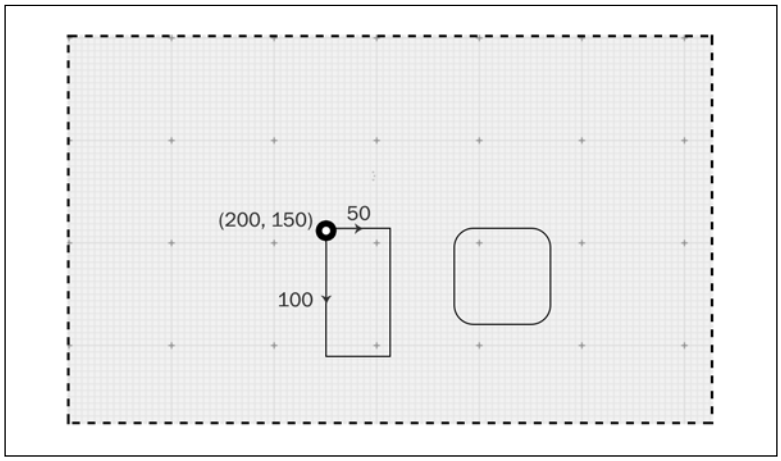
The *x* and *y* coordinates describe the top-left point of the rectangle and the fifth parameter, *r*, is the radius of rounded corners where the square brackets indicate that it is optional (by default the corners have a radius of 0). A rectangle of width 50px and height 100px at x = 200px, y = 150px is created as follows:

```
var paper = Raphael('my-canvas', 500, 300);
var rectangle = paper.rect(200, 150, 50, 100);
```

Similarly, we create a square of width 75px with a corner radius of 15px by:

```
var square = paper.rect(300, 150, 75, 75, 15);
```

Both rectangles are shown in the following diagram. Note that drawing is from the top-left corner of the rectangle in the positive *x* and *y* directions:



The syntax for a circle (`http://raphaeljs.com/reference.html#Paper.circle`) is:

```
Paper.circle(x, y, r);
```

While the *x* and *y* coordinates of a rectangle describe its top-left point, for a circle they define its **center** point. The third parameter, *r*, is the radius of the circle. A circle at *x = 100, y = 100* with a radius of 50px is drawn by:

```
var paper = Raphael('my-canvas', 500, 300);
var circle = paper.circle(100, 100, 50);
```

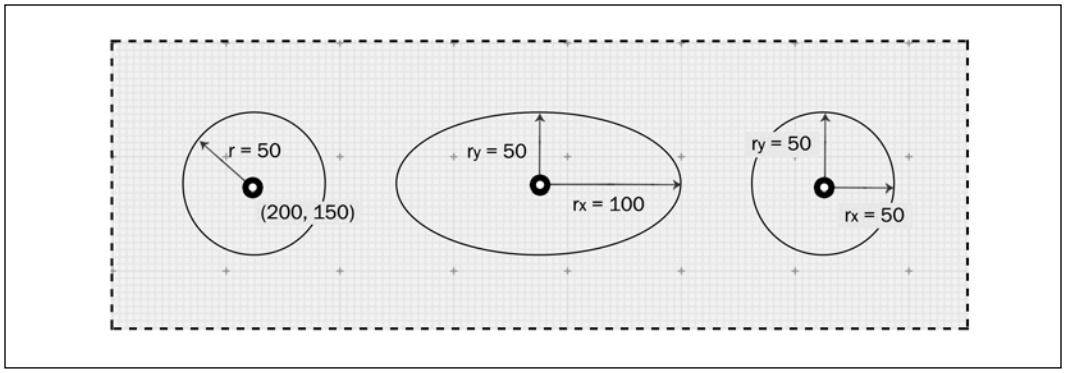
The syntax for an ellipse (`http://raphaeljs.com/reference.html#Paper.ellipse`) is very similar to a circle except that we specify the individual x- and y-radii:

```
Paper.ellipse(x, y, x-radius, y-radius);
```

An ellipse whose `x-radius` is twice its `y-radius` is given by:

```
var ellipse = paper.ellipse(300, 100, 100, 50)
```

It should be clear that the circle is in fact just a special case of the ellipse where `x-radius` is equal to `y-radius`. The resultant circles and ellipses are shown in the following diagram:



Raphaël is well documented at `http://raphaeljs.com/reference.html`. For complete coverage of the methods used throughout this book, you are strongly advised to make extensive reference to this documentation.

Embedding images

Raphaël allows us to embed bitmap (`.jpg` or `.png`) images into our canvas. It does so using the `image` method of the the `Paper` object. The following code demonstrates embedding a `.jpg` image:

```
var paper = Raphael('my-canvas', 600, 252);
paper.image('stella.jpg', 15, 15, 144, 192);
```

The `.jpg` image is included at (15, 15) and its width and height are scaled to 144px and 192px respectively.

Element attributes

The shapes that we have drawn can have fills, strokes, and a number of other attributes applied to them. When we create a shape, an `Element` object is returned. This object has an `attr` method that accepts a number of key-value pair attributes. In this section we will look at the various attributes that can be applied to our drawings using this method (see `http://raphaeljs.com/reference.html#Element.attr`).

Basic fills

In order give an element a background, we fill it using the `fill` attribute. For single-color fills, the format is that specified by the CSS specification (that is, `#rrggbb` or the `#rgb` shorthand, the `rgb(r, g, b)` string or a color keyword, for example, `navy`). To fill rectangles black, for example, we can write:

```
var rect = paper.rect(200, 50, 200, 100);
rect.attr({fill: '#000'});
var square = paper.rect(450, 50, 50, 50);
square.attr({fill: 'rgb(0, 0, 0)'});
```

Similarly, to fill a circle pink we could write:

```
var circle = paper.circle(100, 100, 50);
circle.attr({fill: 'pink'});
```

The following is what you will find as a result:

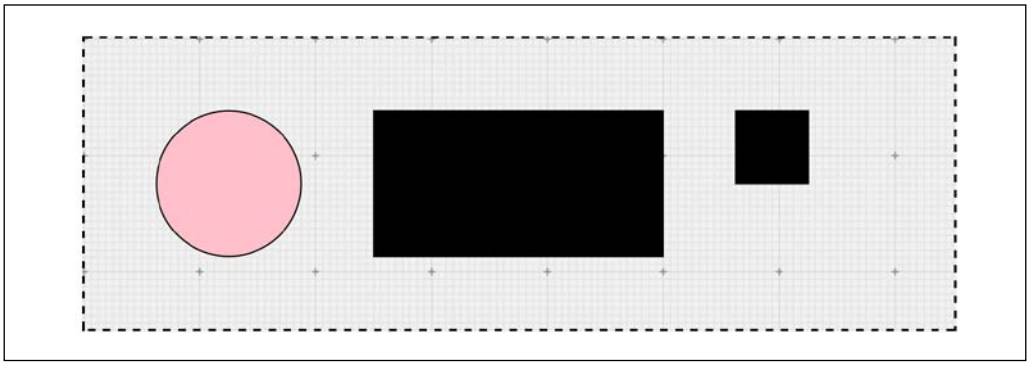
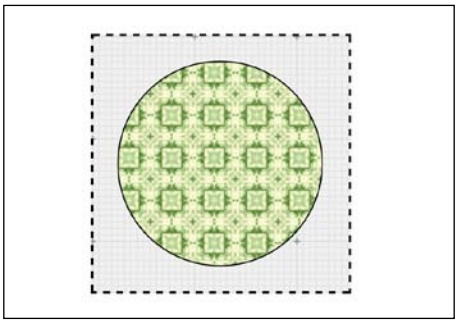


Image fills

Raphaël also allows you to fill an element with an image. Consider the following code:

```
var circle = paper.circle(100, 100, 80);
circle.attr({fill: 'url(bg_pattern.png)'});
```

This creates a circle whose fill is a pattern image named `bg_pattern.png` as shown in the following screenshot. Image fills are always repeated in *x* and *y*:

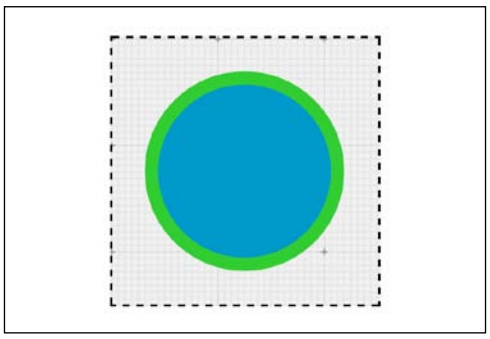


Applying strokes

Elements can have a number of different stroke attributes. The most common are the `stroke` and `stroke-width` attributes. The `stroke` attribute takes a string value for color in line with the CSS specification while `stroke-width` is a number in pixels. The following code has the effect of applying a 10px-wide lime-green stroke to a circle:

```
var circle = paper.circle(100, 100, 70);
circle.attr({
    fill: '#09c',
    stroke: 'limegreen',
    'stroke-width': 10
});
```

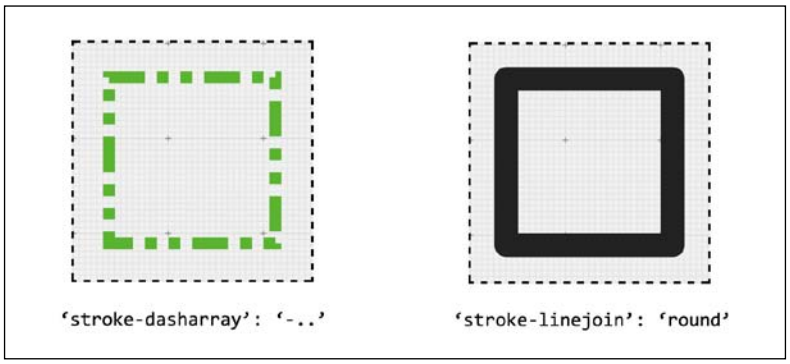
The resulting circle is shown as follows:



Strokes are drawn centrally meaning that half of the visible stroke width is inside the element and half is outside of it.

You will notice in some of the earlier examples in this chapter that even though we did not apply attributes to them they were still stroked. This is because the default `stroke-width` value is 1px. To remove the stroke when drawing shapes, we must always specify a `stroke-width` value of `'none'`.

Some other useful stroke attributes are `stroke-dasharray` and `stroke-linejoin` as demonstrated. The former allows us to specify a pattern of dashes, dots, and spaces by which the resultant stroke will be drawn using them while the latter dictates how path lines should be joined together:



You can find out more about the different stroke attributes at `http://raphaeljs.com/reference.html#Element.attr` and `http://www.w3.org/TR/SVG/painting.html#StrokeProperties`.

Other attributes

Besides strokes and fills, there are a number of other attributes that you may want to use when creating elements. We will demonstrate some of them in this section.

href

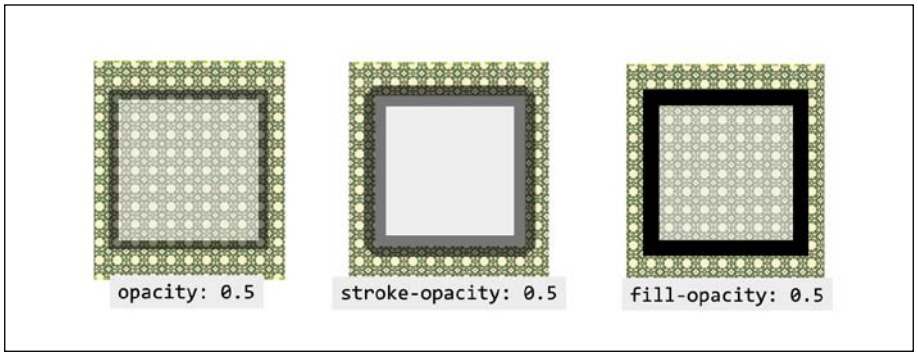
The elements that we create are registered in the DOM and specifying a `href` attribute allows them to behave as hyperlinks. For example:

```
var rect = paper.rect(30, 30, 140, 140);
rect.attr({
    href: 'http://www.packtpub.com'
});
```

This will create a rectangle that links to the Packt Publishing website when clicked on.

opacity

The opacity of elements can be set to a value between 0 (complete transparency) and 1 (complete opacity). We can also set the opacity on the fill and stroke separately by using the `stroke-opacity` and `fill-opacity` attributes:



An unfortunate side effect of strokes always being drawn centrally is that half of the stroke overlaps the fill meaning that `stroke-opacity` has limited applicability.

Specifying `'stroke-opacity': 0.5` in combination with a `stroke` color that matches the `fill` color gives a very smooth edge to your shape, effectively anti-aliasing it.

clip-rect

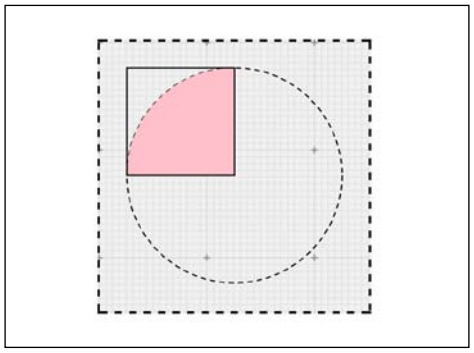
Raphaël supports rectangular clipping of elements using the `clip-rect` attribute. Clipping allows us to show only part of an element. We define a rectangular clipping region as follows:

```
var circle = paper.circle(100, 100, 80, 80);
circle.attr({
    fill: 'pink',
    'stroke-width': 0,
    'clip-rect': '20 20 80 80'
});
```

This creates a circle of radius 80px centered on the point (100, 100). The `clip-rect` attribute defines a rectangular region from a space-separated string of the form:

```
"x y width height"
```

This has the effect of only showing the part of the circle that is overlapped by the rectangular clipping region `'20 20 80 80'`:



Applying gradients

Raphaël supports applying linear and gradient fills to elements. To achieve this, rather than specifying a color string on the `fill` attribute, we specify a string of the following form in order to create linear gradients:

```
<angle>-<color>[-<color>[:<offset>]]*-<color>
```

The following syntax is in order to create radial gradients:

```
r[(<fx>, <fy>)]<color>[-<color>[:<offset>]]*-<color>
```

This syntax is best illustrated by an example.

Linear gradients

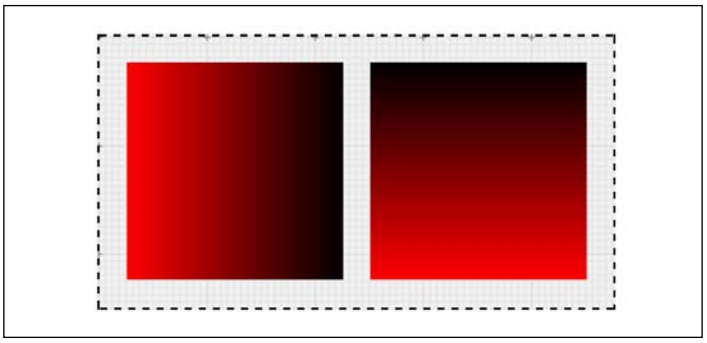
The most basic form of a linear gradient fill is of the form:

```
<angle>-<color1>-<color2>
```

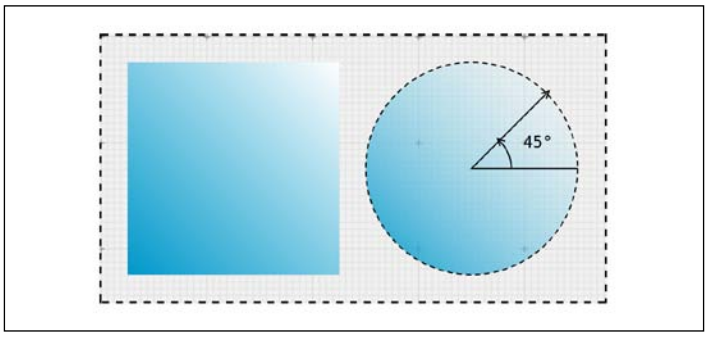
This specifies a gradient drawn at an angle `angle` degrees from `color1` to `color2`. For example:

```
var rect = paper.rect(20, 20, 160, 160);
rect.attr({
    'stroke-width': 0,
    fill: '0-#f00-#000'
});
```

This draws a rectangle with a linear gradient from red to black at an angle of 0 degrees as shown on the left in the following figure. The rectangle on the right is created with a fill value of `'90-#f00-#000'`, where this time the gradient is drawn at an angle of 90 degrees:



Note that the angle of drawing is that defined from the positive *x* axis going counterclockwise as shown in the following figure for a 45 degrees gradient running blue to white:



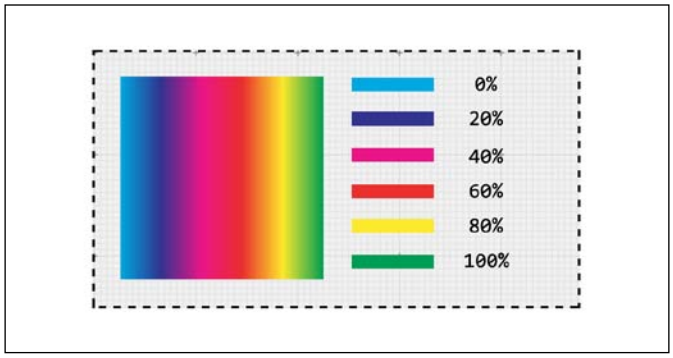
Linear gradients can be composed of any number of colors and the points at which the colors make up the gradient defined as **offsets**. This is what is meant by the following syntax:

```
[-<color>[:<offset>]]*
```

Here any number of `-color:offset` combinations can be created. For example, consider a gradient created with the following string:

```
fill: "0-#00a9e0-#323490:20-#ea1688:40-#eb2e2e:60-#fde92d:80-#009e54"
```

This has the effect of creating a rainbow-style pattern as shown in the following figure. Each color is offset by an additional 20 percent from the previous color. The first and last colors have implied offsets of 0 percent and 100 percent respectively:



Technically, since the difference between each offset is equal for all colors (20 percent), we could have omitted the offset values from the string in the previous example. The gradient described by the following expression yields the same result:

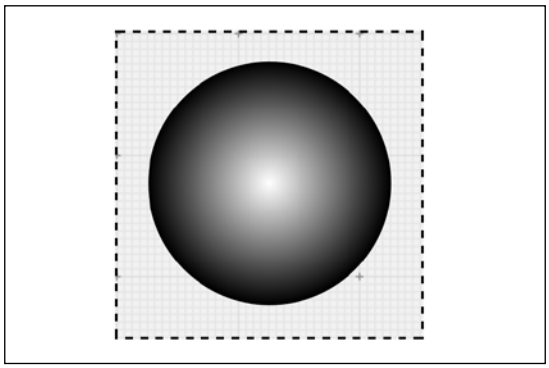
```
"0-#00a9e0-#323490-#ea1688-#eb2e2e-#fde92d-#009e54"
```

Radial gradients

Radial gradients are similar to linear gradients except that they are drawn *radially outwards* from a fixed point on an element (the center point of an element by default). The most basic form a radial gradient is given by the syntax `r<color>-<color>`. For example:

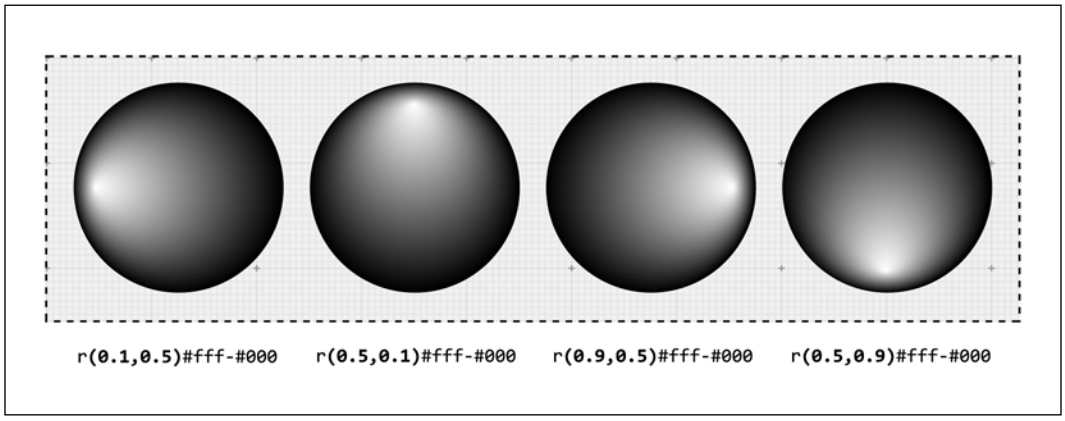
```
var circle = paper.circle(100, 100, 80);
circle.attr({
    'stroke-width': 0,
    fill: 'r#fff-000'
});
```

This describes a circle with a gradient from white at the center to black at its edges:



As with linear gradients, we can define multiple colors with an optional offset for each. The string `'r#fff-#f00-#00f:90-#000'`, for example, creates a radial gradient from white to red to blue at an offset of 90 percent to black at the edges.

The point from which a radial gradient is drawn, known as the **focal point**, can be specified using the `(<fx>, <fy>)` syntax where `fx` and `fy` have values between 0 and 1. The center point of an element has the focal point definition (0.5, 0.5). Reducing or increasing the `fx` value has the effect of shifting the focal point to the left and right in *x*, while increasing and decreasing the value of `fy` has the effect of moving the focal point up and down in *y*. The following diagram shows how changing the value of `fx` and `fy` affects the position of the focal point:



Note that radial gradients cannot be applied to a `Path` element (see *Chapter 3, Drawing Paths*) owing to a bug in VML and so only really applies to the basic shapes we cover in this chapter.

Grouping elements

There are times when we wish to apply the same attributes, transformations, or animations to *multiple* elements. We can group elements in Raphaël by using the `set` method. Consider the following shapes:

```
var paper = Raphael('my-canvas', 540, 200);

var circle = paper.circle(100, 100, 80);
var rect = paper.rect(205, 40, 120, 120);
var ellipse = paper.ellipse(430, 100, 80, 60);
```

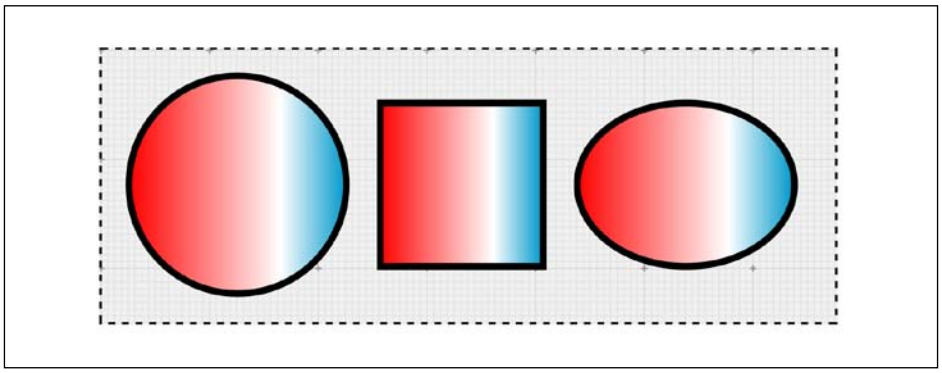
By creating a **set** that contains references to all of these shapes, we can apply the same attributes to all of them by invoking the `attr` method *on the set* rather than each individual element. We create a set containing all elements and hold a reference to it in a variable named `group` as follows:

```
var group = paper.set();
group.push(circle, rect, ellipse);
```

We then apply the following attributes to the set we have just created:

```
group.attr({
    fill: '180-#09c-#fff:30-#f00',
    stroke: '#000',
    'stroke-width': 5
});
```

The result is shown as follows:



Working with text

Drawing text in the canvas rather than as HTML markup with CSS styling allows us to animate and transform it in the same way as we would for other shapes. Text is created using the `text` method and its properties (such as `size` and `font-family`) are modifiable as attributes. The `text` method has the following definition, where the `text` parameter is our text string and accepts the standard escape sequence `'\n'` as input, which places the proceeding text onto a new line:

```
Paper.text(x, y, text)
```

The following code creates two pieces of text using this method and then groups them into a set in order to apply `font-size` and `font-family` attributes to both elements:

```
var text1 = paper.text(0, 15, 'I am text anchored start.');
text1.attr({
    'text-anchor': 'start',
});
var text2 = paper.text(270, 100, 'I am text\nanchored middle');
text2.attr({
    'text-anchor': 'middle'
});
var group = paper.set(text1, text2);
group.attr({
    'font-size': 20,
    'font-family': 'serif'
});
```

You should notice that we have defined an individual attribute, `text-anchor`, on each `text` element. The `text-anchor` attribute has the effect of determining whether the origin *x* point on the `text` method is defined at the center of the text or at its leftmost edge. By default it is defined at its center.

Embedding custom fonts

The font converter and renderer **Cufón** has been superseded by `@font-face` for general web font rendering. However, fonts generated by Cufón have fully defined paths and integrate seamlessly with Raphaël meaning that we can manipulate such fonts in many of the same ways that we would for shapes. A demo of this can be seen at `http://raphaeljsvectorgraphics.com/the-graphical-web/custom-fonts`.

Cufón supports generating custom fonts from **TrueType** (**TTF**), **OpenType** (**OTF**), **Printer Font Binary** (**PFB**) and **PostScript** font files. To generate a custom font using Cufón, go to `http://cufon.shoqolate.com/generate/` and fill in the font generator form. Ensure that you enter `Raphael.registerFont` in the textbox (or click on the small Raphaël logo next to it) under the heading **Customization (for 3rd-party scripts only)** as shown in the following screenshot:

Your generated font will be a JavaScript file. You should include this in your HTML document just after the Raphaël library, that is, as follows:

```
<script type="text/javascript" src="raphael.min.js"></script>
<script type="text/javascript" src="Myfont.font.js"></script>
```

In order to use the font in your canvas, you will use the `print` method (`http://raphaeljs.com/reference.html#Paper.print`). Consider, for example, embedding a font named `Pacifico.font.js`:

```
<script type="text/javascript" src="Pacifico.font.js"></script>
```

We print text using our custom font onto our canvas as follows:

```
var text = paper.print(
    50,
    50,
   'Vi veri veniversum vivus vici',
    paper.getFont('Pacifico'),
    38
);
```

This draws custom text with a font size of 38px onto our canvas at the point (50, 50). The `getFont` method simply references the name of the family of the font that you have embedded, whereas Cufón should have taken care of registering the font with Raphaël.

Attributes can be applied to custom font text in the same way that we have already applied them to shapes and text. For example:

```
text.attr({
    fill: '315-#000-#666',
    'stroke-width': 0
});
```

This produces the text shown as follows:

Summary

In this chapter we have covered many examples of basic drawing and painting in Raphaël in the context of a particular drawing area or canvas. You should now be comfortable drawing basic vector graphics using the library and exploring the documentation at `http://raphaeljs.com`.

In the next chapter, we will look at drawing more complex elements using paths.

3
Drawing Paths

The ability to draw and manipulate paths is an extremely powerful feature of Raphaël. In the previous chapter we drew basic shapes based on the paths' geometric properties such as width, height, and radius. Paths allow us to draw all manner of shapes by defining points connected by lines, arcs, and curves representing the outline of the shape itself.

Paths, such as basic shapes, are *elements* meaning that user interaction, painting, transformations, and animation are all possible with paths. Raphaël provides a number of methods for drawing and manipulating paths to this end so it will be the focus of the rest of this book.

The syntax for drawing paths is defined by the SVG specification. Understanding this syntax is essential for creating complex shapes and over the course of this chapter we will become intimately familiar with it.

To give you an idea of what you can achieve with paths, a demo at `http://raphaeljs.com/chart.html` will show you how smooth curves are used to join data points as is common in charting applications. Indeed, complex SVG illustrations (for example, those created using a graphics package such as Adobe Illustrator or Inkscape) are composed of many paths, an interesting demo of which can be seen at `http://raphaeljs.com/tiger.html`.

This chapter is quite comprehensive in its coverage of paths and so it is suggested that you break off at the end of each section to review and experiment with the concepts covered. In this chapter we will look at the following topics:

- The concepts behind path drawing
- The moveto, lineto, and closepath commands
- Drawing quadratic and cubic Bezier curves
- The arc drawing commands
- Utility methods in Raphaël for working with paths
- The Catmull-Rom curve as exemplified by data interpolation

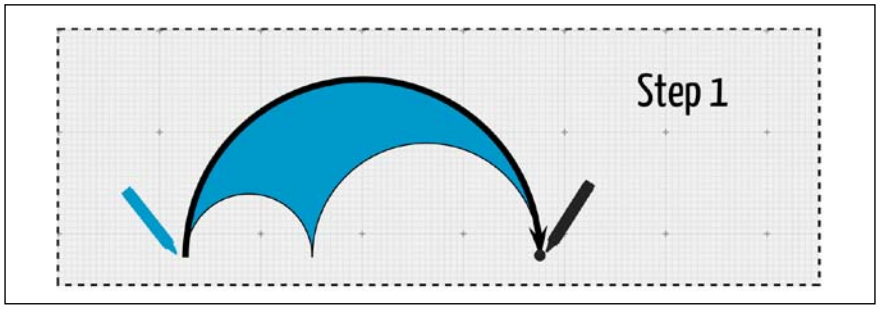
Path drawing concepts

The process of drawing with a pen on paper can be broken down into the following steps:

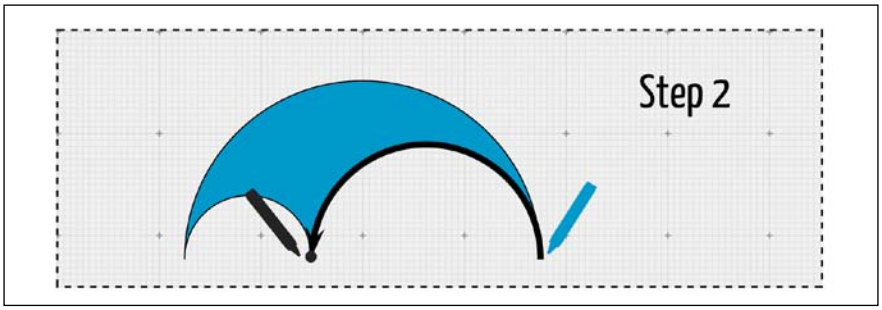
1. You place your pen at a particular point on a piece of paper.
2. You press and move the pen freely from this point to another point.
3. You keep your pen at this point or lift up the pen and place it at another point on the paper.
4. The process is repeated until you have finished drawing.

Path drawing works in much the same way. The point at which you place your pen, known as the **current point**, defines the start of a path while the free movement of the pen describes the path itself.

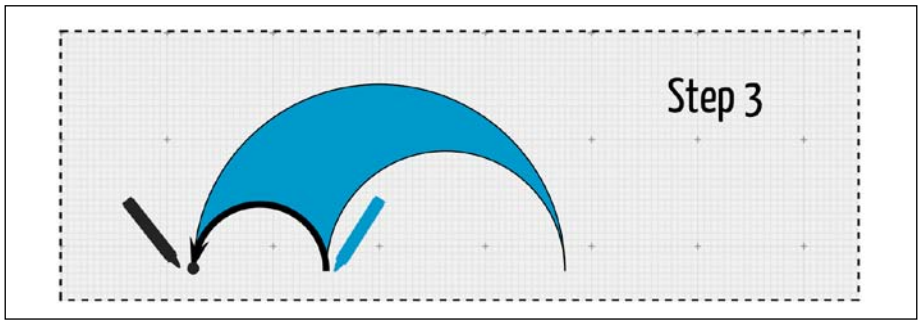
Consider drawing an arbelos shape. We first place our pen at a point (100, 180) on our canvas and draw an arc to the point (380, 180) as shown:



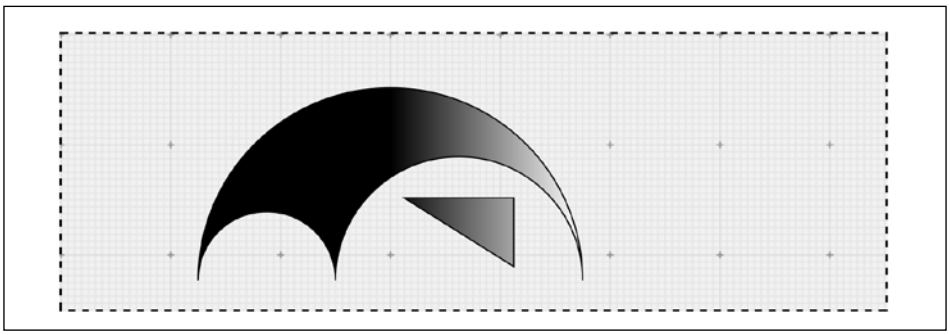
We then create an arc from the point (380, 180) to the point (200, 180):



Finally, we create an arc back to the point (100, 180) to complete the path:



The example of the arbelos demonstrates the drawing of a single path, where we did not lift up the pen at any point during the drawing process. Were we to lift up the pen, the subsequent drawing would technically create **subpaths** on the main path element. In the example, creating a triangle as a subpath has the effect of *adding* to our single path element. Notice also that the fill attribute is applied to the *overall* path rather than individual subpaths:



Path drawing commands

Paths are based on a number of drawing commands executed in the order in which they're defined. Commands are expressed as a single letter that is either uppercase, meaning that a subsequent drawing uses absolute coordinates in the drawing context, or lowercase, meaning that a subsequent drawing uses coordinates relative to the current point. Most commands take parameters as arguments that determine the behavior of the command. Consider, for example, the following path string:

```
"M100,180 a140,140,0,0,1,280,0"
```

This path string is used in drawing the arbelos of the previous section. Drawing is started using the `M` command and subsequent arcs are then drawn using the `a` command.

We draw paths in Raphaël using the `path` method, which accepts either a path string or an array as a parameter. Arrays are usually a more convenient and more readable way of defining paths. Consider the following code:

```
var xStart = 100;
var yStart = 180;
var path = paper.path([
    'M', xStart, yStart,
    'a', 140, 140, 0, 0, 1, 280, 0
]);
```

This draws the path described by the previous path string. The array format is more readable and is a much cleaner way of dealing with parameter values assigned to the `xStart` and `yStart` variables.

In the interests of brevity, we will use both formats for defining a path interchangeably throughout this book.

The moveto command

The **moveto** command establishes a new current point. In the pen analogy detailed in the previous section, issuing a moveto command has the effect of lifting the pen off the page and placing it at a given point. All paths start with a moveto command.

The moveto command has the following syntax:

Command	Parameters	Example
`M` or `m`	`(x, y)+`	`M 50,50 100,100`

In the example of the arbelos, `M 100,180` has the effect of placing the pen at the point x = 100, y = 180. Subsequent commands then use these coordinates as the starting point of the drawing.

You may have noticed that the moveto syntax allows us to repeat the `x` and `y` parameters (indicated by the `+` symbol). Consider, for example, the following path:

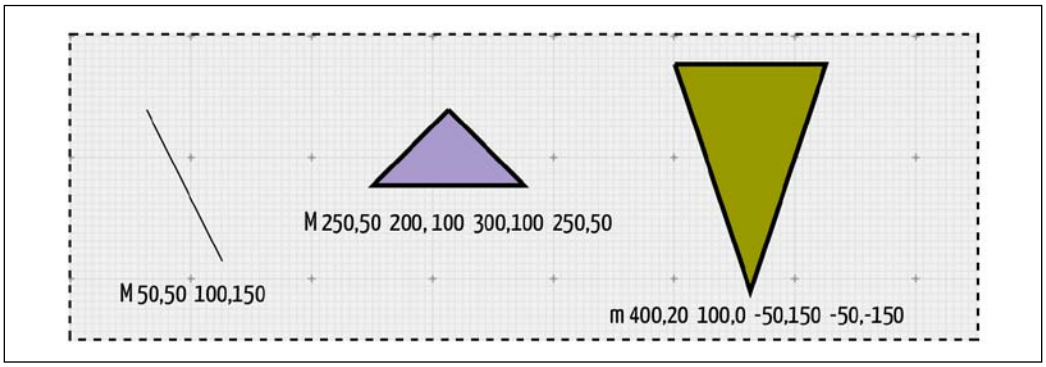
```
"M 50,50 100,150"
```

The first *(x, y)* pair moves the pen to the point (50, 50) on our canvas as we would expect. However, the second *(x, y)* pair has the effect of drawing a line to the point (100, 150), as well as moving the current point. This behavior means that the moveto command can be used to draw polygons. The following code draws a triangle with vertices at (250, 50), (200, 100), and (300, 100) using only the moveto command:

```
var path = paper.path([
    'M', 250, 50
    200, 100,
    300, 100,
    250, 50
]);
```

You should notice that all the *(x, y)* points mentioned are *absolute*, that is, *defined relative to the origin point of our canvas*. Were we to specify the lowercase variant of the command, `m`, every successive *(x, y)* pair would be relative to the previous one (but note that the first *(x, y)* pair is always positioned using absolute coordinates regardless of the `M` command's case). The same triangle defined using relative coordinates is described by the path string, `m 250,50 -50,50 100,0 -50,-50`.

A number of examples of the moveto command are shown here:



The lineto commands

The **lineto** commands draw straight lines from the current point to a point specified. There are three lineto commands:

- `L` for drawing lines to any *(x, y)* point
- `H` for drawing horizontal lines in *x*
- `V` for drawing vertical lines in *y*

Command	Parameters	Example
`L` or `l`	`(x, y)+`	`L 100 100`
`H` or `h`	`x+`	`H 75`
`V` or `v`	`y+`	`V -75`

By way of example, a line from the point (10, 10) to the point (75, 100) would be drawn as follows:

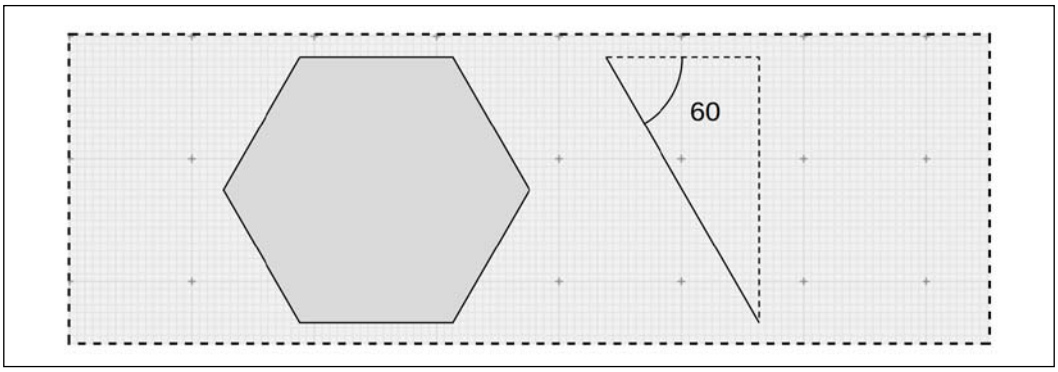
```
var line = paper.path(['M', 10, 10, 'L', 75, 100]);
```

Note that because we use the uppercase command, `L`, the line is drawn from the point (0, 0) and not relative to our start point (10, 10). Had we used the lowercase variant, `l`, the line would have been drawn from (10, 10) to the point (10 + 75, 10 + 100) = (85, 110).

By repeating parameters, we can draw polygons (like we did in the previous section with moveto, except that lineto can be specified from any current point, not just moveto points). For example, a regular hexagon with side length 100 starting at the point (200, 15) could be drawn as follows:

```
var sideLength = 100;
var x = sideLength * Math.cos(Raphael.rad(60));
var y = sideLength * Math.sin(Raphael.rad(60));

var hexagon = paper.path(
    ['M', 250, 15,
    'l', sideLength, 0,
    x, y,
    -x, y,
    -sideLength, 0,
    -x, -y,
    x,-y,
    ]
);
```

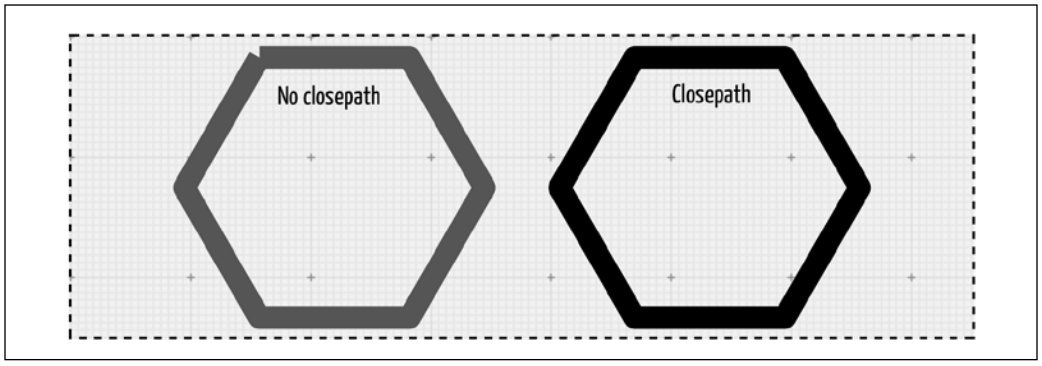
We use the fact that an exterior angle of a hexagon is 60 degrees (as shown in the following diagram) in conjunction with the trigonometric functions `sin` and `cos` to calculate the *x* and *y* coordinates of the point to which we want to draw. Starting from the point (250, 15), we draw a line 250 pixels relative in *x* and 0 pixels in *y*; we then draw a line that is `x` pixels in *x* and `y` pixels in *y* relative to this point. Since the hexagon is regular, the magnitudes of the relative *x*- and *y*-points are equal for sides for which *y* is greater than zero (all edges except the top and bottom edges), hence we are able to draw in factors of *x* and *y* back to the start point.



Note that in JavaScript, the trigonometric functions expect a value of angle measured in radians to be passed so we have used the `rad` utility method on the `Raphael` object to convert 60 degrees into radians.

The closepath command

The **closepath** command draws a straight line from the current point to the starting point of the current subpath. In the example of the hexagon, even though we drew back to the start point, the path's start and end points are not joined. The effect is pronounced when applying a large `stroke-width` value to the path as shown:



The closepath command character is `z` and does not accept any parameters. Since in the previous example we drew a straight line back to the end point, we could instead close the path using the closepath command as follows:

```
var hexagon = paper.path(
    ['M', 250, 15,
    'l', sideLength, 0,
    x, y,
    -x, y,
    -sideLength, 0,
    -x, -y,
    'z'    // in place of the line to (x, -y)
    ]
);
```

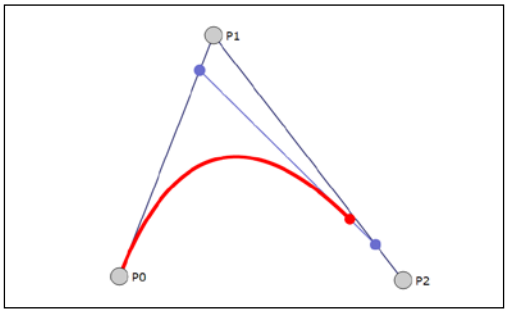
In this instance, the path's start and end points *are* joined. Note that the closepath command can be either uppercase or lowercase but this has no effect on its behavior.

Drawing curves

There are three types of curve path: quadratic Bézier curves, cubic Bézier curves, and arcs. Bézier curves are curves defined between a start and end point but whose direction we can determine by using **control points**, while arcs are a portion of the circumference of an ellipse.

Quadratic Bézier curves

A **quadratic Bézier curve** is a curve between two points with a single control point. To appreciate how quadratic Bézier curves are drawn, you should experiment with the animated demo at `http://www.jasondavies.com/animated-bezier/`. As shown here, the curve is drawn from a point **P0** to a point **P2** with a control point **P1**. The control point **P1** relative to **P0** and **P2** determines the extent to which the curve bends: from start to finish, the curve starts off heading in the direction of the control point **P1** and then bends towards the end point **P2** from the direction of **P1**.



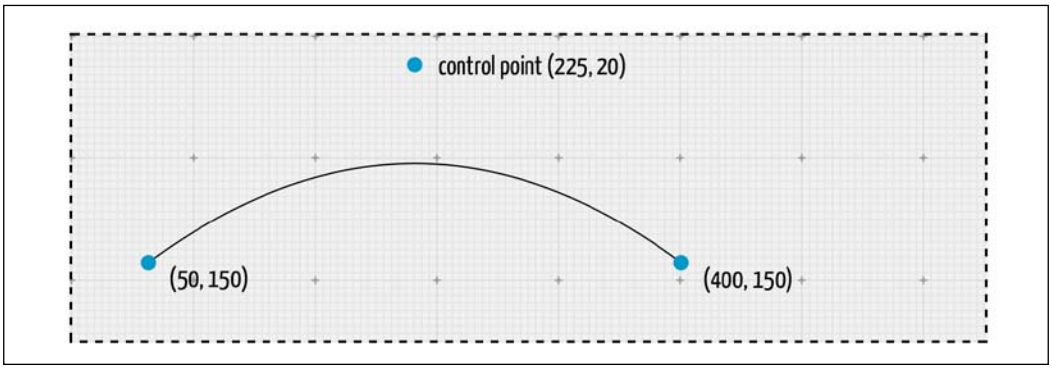
There are two quadratic Bézier curve commands, the syntax for which is given here:

Command	Parameters	Example
`Q` or `q`	`(x1, y1, x, y)+`	`Q 100 50 200 250`
`T` or `t`	`(x y)+`	`T 400 250`

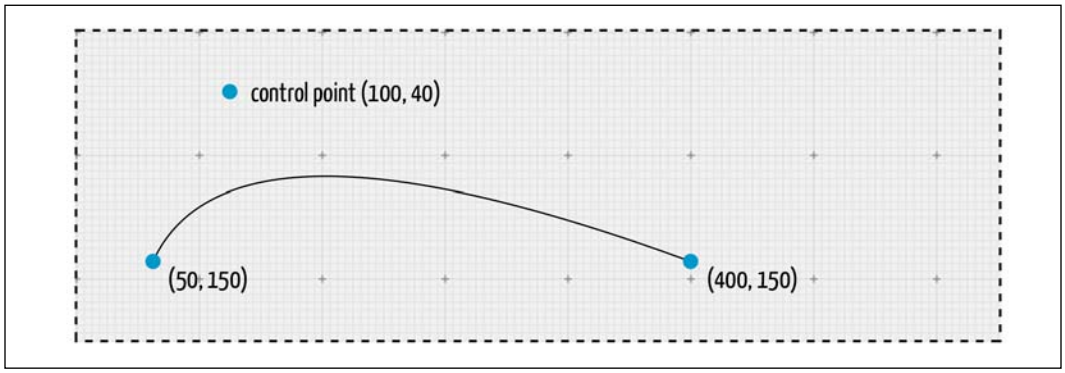
The `Q` command (or `q` for relative points) describes a curve drawn from the current point on a path to the point *(x, y)* using *(x1, y1)* as a control point. For example, consider the following code:

```
paper.path(['M', 50, 150, 'Q', 225, 20, 400, 150]);
```

This draws the quadratic Bézier curve shown. The equivalent path using the lowercase variant of the command would be `"M 50,150 q 175,-130 350,0"`, where the `(x, y)` and `(x1, y1)` parameters are the relative distances from the start point (50, 100):



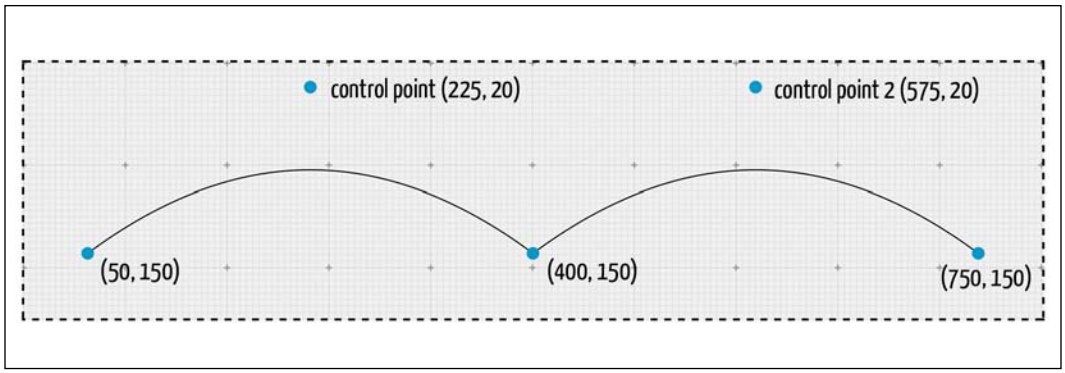
Moving the control point affects the way that the path is drawn. For example, the path `"M 50, 150 Q 100,40 400,150"` is shown as:



As with the other commands we have encountered so far, parameters can be repeatable, which allows us to draw multiple connected quadratic Bézier curves. Consider the following code:

```
paper.path([
    'M', 50, 150
    'Q', 225, 20, 400, 150,
    575, 20, 750, 150
]);
```

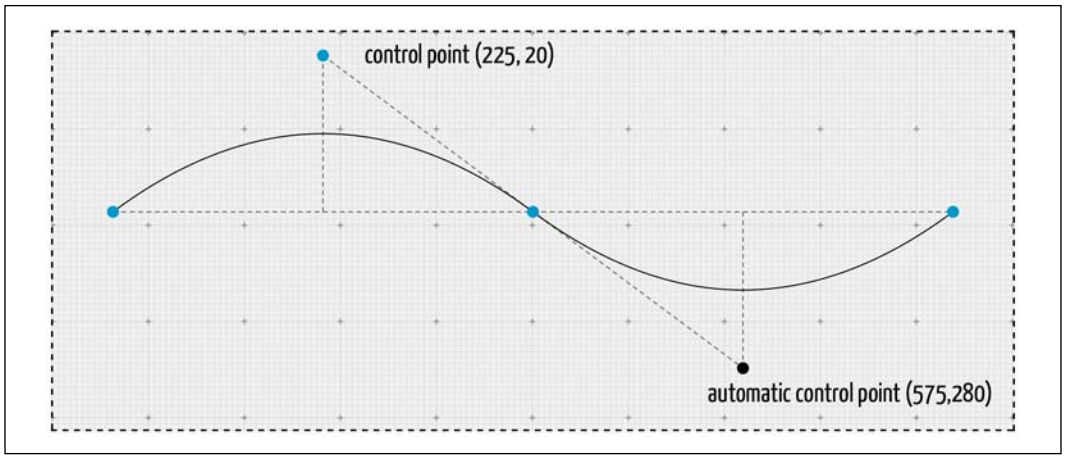
This has the effect of drawing a second curve from (400, 150) to the point (750, 150) with a control point at (575, 20):



The `T` or `t` command is shorthand whereby the control point coordinates are not specified. Instead, the control point is determined automatically as a reflection of the previous control point. Consider the path drawn by the following code:

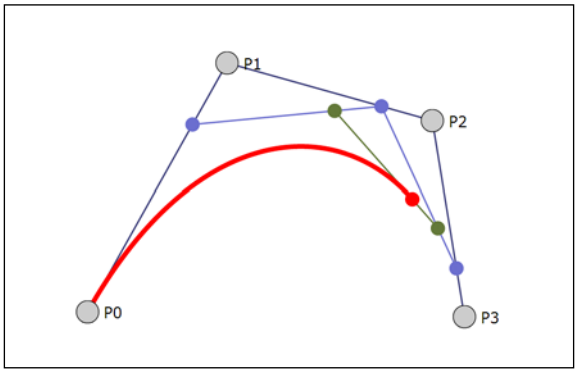
```
paper.path([
    'M', 50, 150,
    'Q', 225, 20, 400, 150,
    'T', 750, 150
]);
```

This creates two curves as shown in the following screenshot. The current point at the start of the path drawn by `T` is (400, 150). Relative to this point, a reflection of the previous control point (225, 20) is (575, 280):



Cubic Bézier curves

The principles behind drawing cubic Bézier curves are similar to those for quadratic Bézier curves except that cubic Bézier curves have two control points. Again, in order to appreciate how cubic Bézier curves are drawn, you should take a look at the animated demo at `http://www.jasondavies.com/animated-bezier/`. The cubic Bézier curve's path (highlighted in red in the screenshot) starts from **P0** and heads in the direction of **P1** before arriving at the end point **P3** from the direction of **P2**:



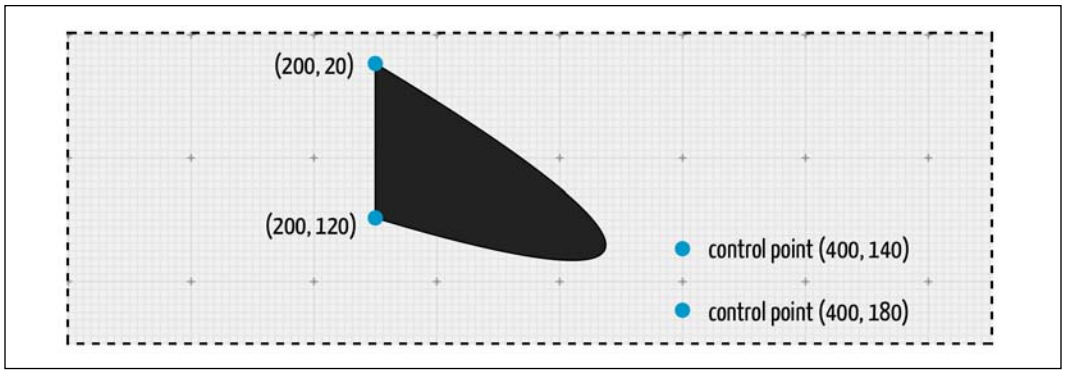
There are two cubic Bézier curve commands, the syntax for which is shown in the table given:

Command	Parameters	Example
`C` or `c`	`(x1, y1, x2, y2, x, y)+`	`C 150 25 350 50 500 300`
`S` or `s`	`(x2, y2, x y)+`	`t 200 -100 200 50`

The `C` command requires that we specify both control points. The following code has the effect of drawing a curve from the point (200, 20) to (200, 120) via the control points (400, 140) and (400, 180). Note also that we have closed the path and applied an attribute fill:

```
paper.path([
    'M', 200, 20,
    'C', 400, 140, 400, 180, 200, 120,
    'z'
]).attr({fill: '#222'});
```

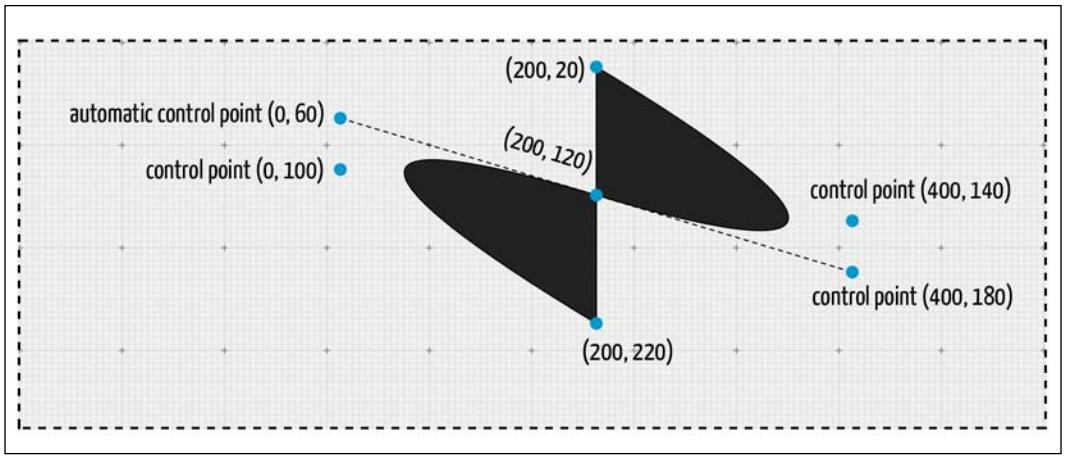
The result is shown as follows:



The `S` command has a similar effect to the `T` command in automatically creating a control point reflected about the current point. What it actually does is to reflect the second control point about the current point and this becomes the first control point for our new subpath, for which we now have to specify our second control point. The code given here extends the previous example with a subpath drawn using the `S` command:

```
paper.path([
    'M', 200, 20,
    'C', 400, 140, 400, 180, 200, 120,
    'S', 0, 100, 200, 220,
    'z'
]);
```

The `S` path starts from the current point (200, 120) and finishes at (200, 220) as shown. The second control point from our first curve, defined at (400, 180), is reflected about the current point, which creates a control point at (0, 60). We have then specified a second control point at (0, 100):

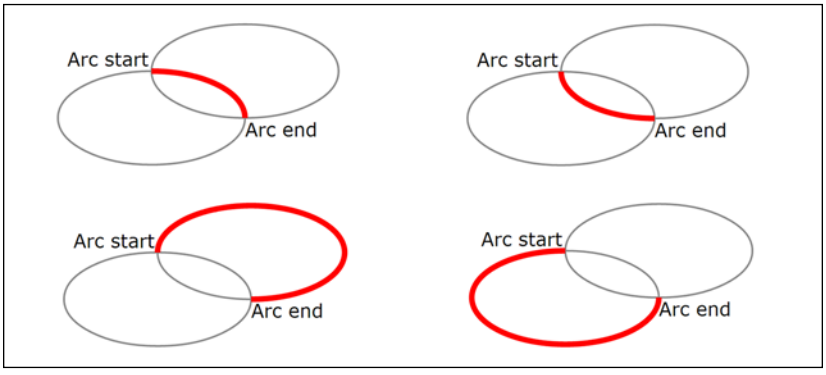


The hardest thing about working with Bézier curves is deciding where to define your control points. Figuring these out is usually achieved by one of the following:

- Trial and error
- Mathematical computation
- Using a graphics package such as Inkscape to aid you in drawing

Drawing arcs

The syntax for drawing arcs is the most involved of all the drawing commands. This is because when we draw an arc from one point to another, there are four possible arc paths available to us as shown here—there are two ellipses on whose circumference the start and end points can lie. The first two arcs are known as **minor** arcs while the latter two arcs are known as **major** arcs. The resultant arc is also affected by the angular direction taken in going from the start point to the end point:



The syntax for arc drawing is given in the following table:

Command	Parameters	Example
`A` or `a`	`(rx, ry, x-rotation, large-arc-flag, sweep-flag, x, y)+`	`a 25 50 0 1 0 100 200`

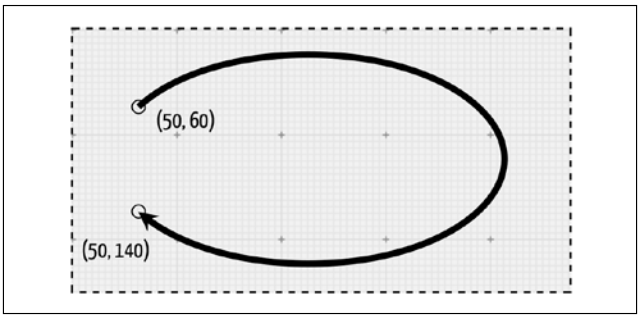
The following table describes each of the individual parameters:

Parameters	Description
`rx` and `ry`	Since an arc is a portion of an ellipse, it has a radius in *x* and *y*
`x-rotation`	The counterclockwise rotation of the ellipse along which the arc is drawn (a value in degrees)
`large-arc-flag`	Whether our arc is drawn as a major (1) or minor (0) arc
`sweep-flag`	Whether the arc is drawn in a positive (1), that is, clockwise or negative (0), that is, counterclockwise, angular direction
`x` and `y`	The end point of our arc

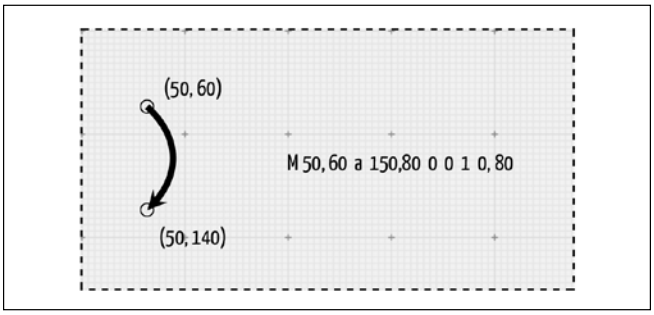
These parameters are perhaps best illustrated by example. Consider drawing a major arc path with a `sweep-flag` equal to 1:

```
paper.path([
    'M', 50, 60,
    'a', 150, 80, 0, 1, 1, 0, 80
]);
```

We first move to the current point (50, 60) and then draw an arc with an *x*-radius of 150 and a *y*-radius of 80. We set the `x-rotation` value equal to 0 and identify that we're drawing the large arc by setting `large-arc-flag` equal to 1. The `sweep-flag` value is equal to `1` and we use the lowercase, that is, relative, variant of the `A` command meaning our final `x` and `y` parameters (0, 80) are units relative to the start point, that is: 0 additional pixels in `x` and 60 additional pixels in `y` giving end point coordinates of (50, 140). The resultant arc is shown as follows.

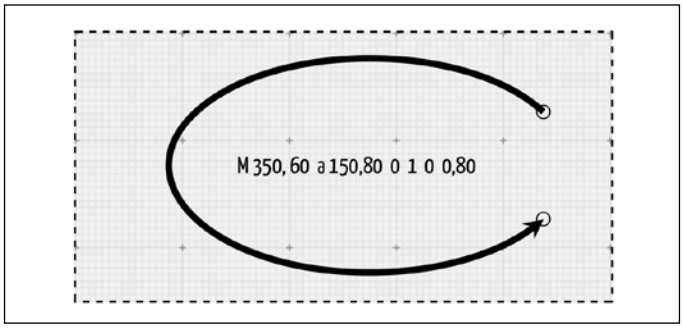


Were we to specify a `large-arc-flag` value equal to 0, the minor arc of the ellipse passing through (50, 60) and (50, 140) would be drawn as shown:

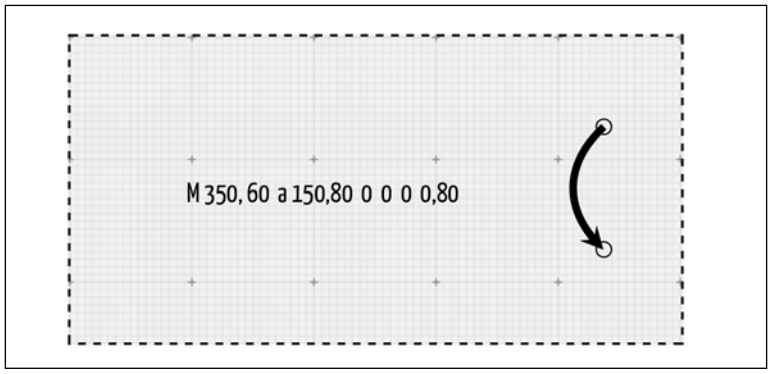


Were we to specify `sweep-flag` equal to 0, for example, for the path described by `M 350,60 a 150,80 0 1 0 0,80` in the case of the major arc or `M 350,60 a 150,80 0 0 0 0,80` in the case of the minor arc, the arc would be drawn in a counterclockwise direction.

The following image describes a major arc (`large-arc-flag = 1`) drawn in a counterclockwise direction:



The following diagram describes a minor arc (`large-arc-flag = 0`) drawn in a counterclockwise direction:



Utility methods for working with paths

Raphaël provides a number of convenient utility methods for working with paths. We will demonstrate these methods by example.

Element.getTotalLength()

By combining a major arc and a cubic Bézier curve, we can draw a bowling pin shape as follows:

```
var pin = paper.path({
    'M', 130, 350,
    'A', 100, 180, 0, 1, 0, 210, 350,
    'C', 290, 100, 50, 100, 130, 350,
    'z'
});
```

This path starts at (130, 350), arcs to (210, 350), and then curves back to the start point with Bézier control points (290,100) and (50, 100). We get the total length of the path using the `getTotalLength` method. We retrieve this value and then display it to two decimal places on our canvas as follows:

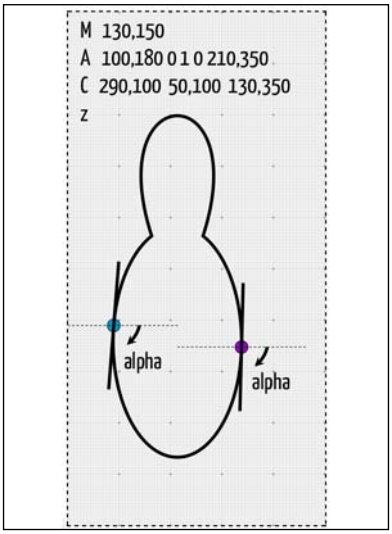
```
var totalLength = pin.getTotalLength();
paper.text(
    180, 50,
    "Total Length: "  + totalLength.toPrecision(2)
);
```

Element.getPointAtLength(length)

We can get the *x* and *y* coordinates of any point along the length of a path using the `getPointAtLength` method. This method returns an object with `x` and `y` attributes. Using the example of a bowling pin from the previous section, we plot two circular points at the one-eighth and one-half of the total length of the path as follows:

```
var totalLength = pin.getTotalLength();
p1 = pin.getPointAtLength(totalLength / 8);
p2 = pin.getPointAtLength(totalLength / 2);
paper.circle(p1.x, p1.y, 10);
paper.circle(p2.x, p2.y, 10);
```

Another attribute of the object returned by `getPointAtLength` is the `alpha` attribute. This returns the positive angle relative to the *x*-axis subtended by the gradient or derivative line at the point specified as illustrated:



Knowing the derivative at a point is useful as it informs us of the extent to which the arc is changing in *x* and *y* at that particular point.

Element.getSubpath(from, to)

Given a path element, the `getSubpath` method calculates the path string for a subsection of its path. Given two points on a path, it returns the path string that describes the original path between just those two points. Consider the cubic Bézier curve between the points (100, 100) and (300, 100) with anchor points at (100, 200) and (300, 0):

```
var path = paper.path([
    'M', 100, 100, 'c', 0, 100, 200, -100, 200, 0
]);
path.attr('stroke-width', 5);
```

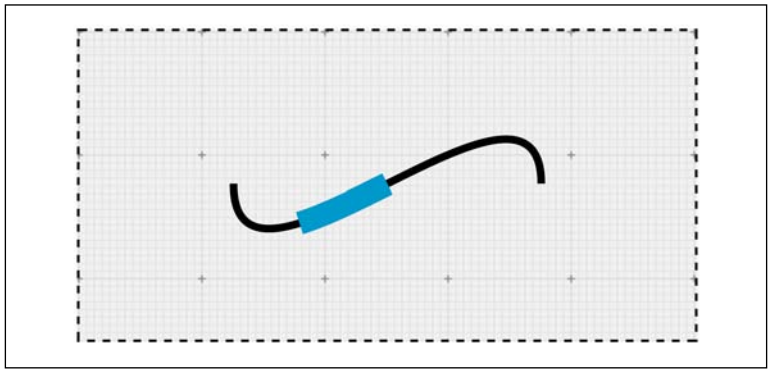
We retrieve the subpath between the point one-quarter along its length to the point at one half of its total length as follows:

```
var totalLength = path.getTotalLength();
var subPath = path.getSubpath(
    totalLength /4, totalLength  / 2
);
```

The `getSubpath` method returns a path string that can be re-used. For example:

```
paper.path(subPath).attr('stroke-width', 15);
```

This code draws a shorter, thicker section of our original path as shown here:



There are a number of other utility methods on the `Raphael` object that are particularly useful when working with Bézier curves in the context of transformation and animation. We will look in more detail at some of these methods in the remainder of this book.

Catmull-Rom curves

In those cases where we need to draw a curve that passes through a prescribed set of points, a Catmull-Rom curve is often appropriate. Catmull-Rom curves are commonly used in charting (that is, in data plotting) and gaming (for example, to move a character along a predefined trajectory).

Raphaël provides a path command to facilitate the drawing of Catmull-Rom curves. The syntax for drawing Catmull-Rom curves is given:

Command	Parameters	Example
`R` or `r`	`x1,y1 (xi, yi)+`	`a 25 50 0 1 0 100 200`

The point `(x1, y1)` is the second point the curve should pass through while the `(xi, yi)` points are every subsequent point. The first point should always be our current point, from which the curve starts.

To illustrate Catmull-Rom curves, consider the following data points stored in an array:

```
var data = [
    {x: 50, y: 250},    {x: 100, y: 100},    {x: 150, y: 150},
    {x: 200, y: 140},   {x: 250, y: 250},    {x: 300, y: 200},
    {x: 350, y: 180},   {x: 400, y: 230}
];
```

To plot these data points, we iterate over the array and draw a circle at each *x* and *y* position (note that the points are grouped in to a set which makes applying attributes to them easier):

```
var plottedPoints = paper.set();
for(var i = 0, num = data.length; i < num; i+=1) {
    var point = data[i];
    plottedPoints.push(paper.circle(point.x, point.y, 5));
};
plottedPoints.attr({fill: '#09c', 'stroke-width':0});
```

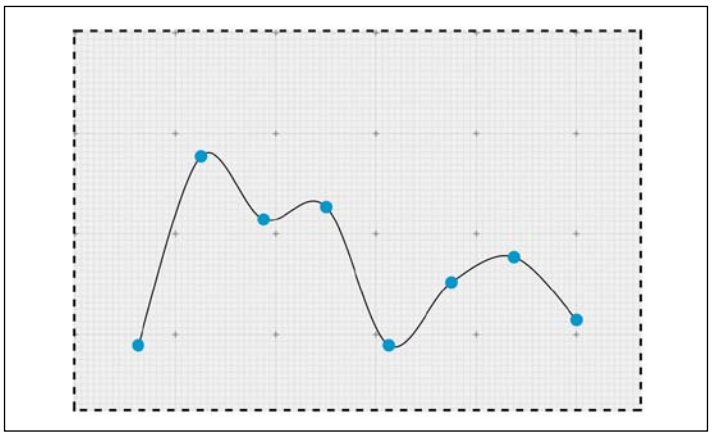
When constructing our path string or path array, we make the current point our first data point using the `M` command and append the `R` command to begin drawing our Catmull-Rom curve:

```
var path = ['M', data[0].x, data[0].y];
path.push('R');
```

We then iterate over all remaining points (all points except the first) and append their values to our path array:

```
for(var i = 1, num = data.length; i < num; i+=1) {
    path.push(data[i].x);
    path.push(data[i].y);
}
var curve = paper.path( path );
```

The resultant curve drawn using the `path` command is shown here:



Summary

We have now covered all the main drawing methods in Raphaël and you should be familiar with the syntax used to draw paths. At this point, it is definitely worth experimenting with the path commands and utility methods we have covered in order to create your own unique graphics.

We are nicely poised to begin looking at interacting with and manipulating our drawings. Up to now, the graphics we have created are drawn once and do not change. In the next chapter we will look at responding to events triggered by a user and transforming the elements that we create.

4
Transformations and Event Handling

Having spent the past two chapters drawing static graphics, we will now start to look at how we can transform them and how we can interact with them in the browser.

When you create an element in Raphaël, you are effectively creating a **Document Object Model** (**DOM**) object. Like the DOM objects you are already familiar with elements drawn on to our canvas will trigger events when interacted with by a user. Raphaël provides a number of convenient, cross-browser methods for interacting with elements and also provides methods that facilitate the drag-and-drop functionality.

Transformations are equally versatile: Raphaël gives us the ability to transform (translate, scale, and rotate) elements with ease using the concept of a **transformation string**.

Over the course of this chapter, we will look at event handling and transformations alongside each other, combining both facets of the library to yield interactive graphics. Overall, we will cover:

- The transform method and basic transformations
- Matrix transformations
- Registering and unregistering basic event handlers
- Achieving the drag-and-drop functionality

Basic transformations and event handling

We perform transformations in Raphaël using the `transform` method on elements, which accepts as an argument a transformation string. Transformation strings define a sequence of transformations to take place on a particular element.

You can also transform elements and sets using the `transform` attribute. This is often convenient when we are changing multiple attributes at the same time. You will see examples of this in *Chapter 5*, *Vector Animation*.

Basic transformations

Like a path string, a transformation string is composed of a number of commands defined in the order in which they are to be interpreted. The syntax for the basic commands is as follows, where brackets indicate optional parameters:

Command	Parameters	Example
`T` or `t`	`x,y`	`t 50,100`
`R` or `r`	angle of clockwise rotation, `(rotation_point_x, rotation_point_y)`	`R 45,0,0`
`S` or `s`	`scale_x, scale_y, (scale_point_x, scale_point_y)` or `scale_factor, (scale_point_x, scale_point_y)`	`S 2,4.5,75,125`

As with a path string, transformation strings have uppercase and lowercase variants. The uppercase variant means that we transform, irrespective of the previous transformation, while the lowercase variant takes previous transformations into account. The following examples demonstrate the basic transformations.

Translation

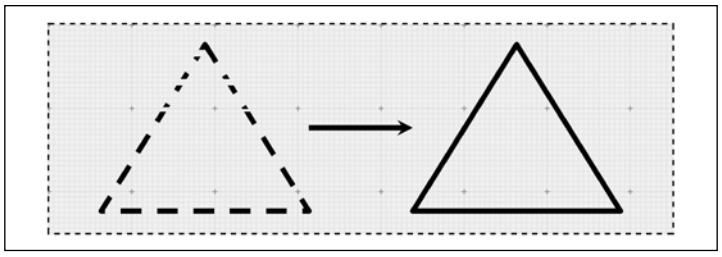
First off, consider drawing a triangle and creating a copy of it using the `clone` method:

```
var triangle = paper.path("M 150 20 l 100 160 l -200 0 z").attr({
    'stroke-dasharray': '- ',
    'stroke-width': 5
});

var triangleCopy = triangle.clone(); // Clones our object
triangleCopy.attr({
    'stroke-dasharray': ''
});
```

When cloned, the copied triangle has all the same attributes as the original triangle so we unset `stroke-dasharray` to give it a solid outline. We then perform a translation of 300px on the *x*-axis and 0px on the *y*-axis as follows:

```
triangleCopy.transform("T 300 0");
```

This has the effect of moving our cloned triangle in *x* as shown in the following diagram:



Note that while the `clone` method is convenient, it is known to affect performance adversely when cloning multiple elements. When cloning multiple elements, creating a new element using the original path is advised.

Rotation

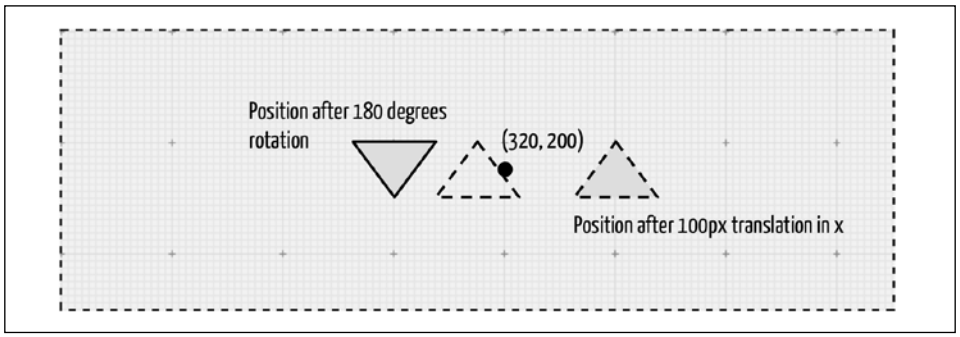
Creating another cloned triangle now and transforming it with the transformation string below gives us a triangle rotated 45 degrees clockwise about its own center point (the default when we do not specify a rotation point) *after* being translated 350px in *x* and -35px in *y*.

```
triangleCopy.transform("t 350 -35 r 45");
```

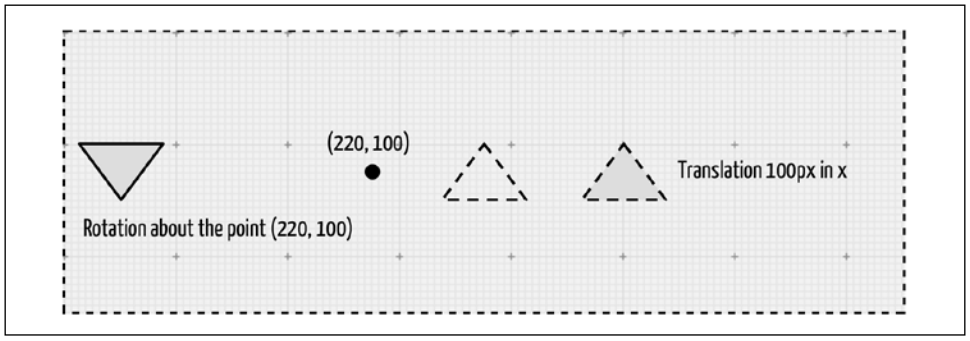
Here we have combined the commands. In this instance, the transformations are mutually independent meaning that the case of each command is unimportant. However, consider rotation about a point:

```
triangleCopy.transform("t 100 0 r 180 220 100");
```

This transformation first translates our triangle by 100px in *x* and then rotates it by 180 degrees clockwise. Our rotation point (220, 100) is not the point that the triangle is rotated about because the rotation point is effectively translated 100px in *x* to the new point (320, 100). The important point to note here is that *the lowercase transformation command takes the previous transformation into account*:

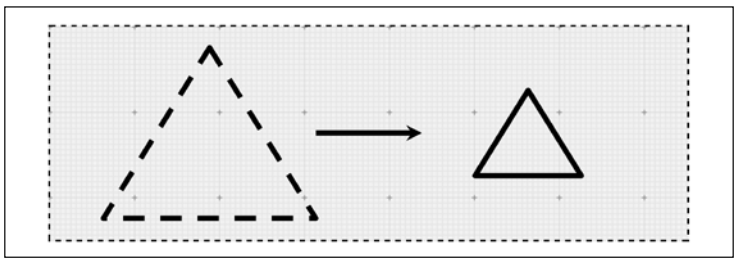


In the case of the absolute rotation command, `R`, rotation about the point (220, 100) likewise occurs after translation but the results are markedly different as shown in the following diagram given here. The `t 100,0 R 180,220,100` transformation string always rotates an element about the point (220, 100) irrespective of the previous transformations:

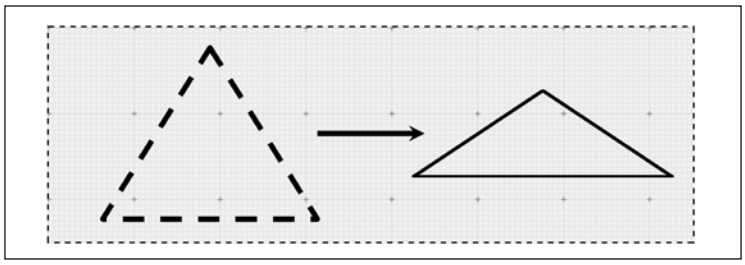


Scaling

Scaling can be performed either equally on all points about a scaling point or a factor in *x* and *y*. The scaling for the `t 300 0 s 0.5` transformation scales all points by 0.5 times their original about the center point of the element as shown:



Scaling in *x* and *y* by the `t 310 0 s 1.2 0.5` transformation string scales the element 1.2 times in *x* and 0.5 times in *y* as shown:



Basic event handling

Raphaël supports registering and unregistering a whole bunch of event handlers on elements. You are encouraged to experiment with the available event handlers, a number of which we will demonstrate by example.

Registering basic event handlers

We will demonstrate registering the double-click, mouse over, and mouse out event handling methods on a salinon shape. Firstly, we can draw a salinon as follows:

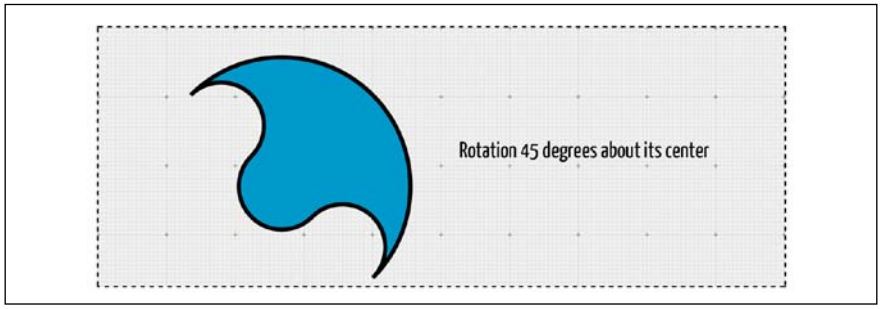
```
var salinon = paper.path([
    'M', 100, 200,
    'a', 150, 150, 0, 0, 1, 300, 0,
    50, 50, 0, 0, 0, -100, 0,
    50, 50, 0, 0, 1, -100, 0,
    50, 50, 0, 0, 0, -100, 0
]);
```

We then define the aforementioned event handlers as follows, where each creates a transformation on an element:

```
salinon.mouseover(function() {
    this.transform('R 180');
});
salinon.dblclick(function() {
    this.transform('r 45');
});
salinon.mouseout(function() {
    this.transform('');
});
```

This has the effect of:

- When moving your mouse over the element, it is rotated by 180 degrees
- When double-clicking the element, it is rotated by 45 degrees
- When moving your mouse outside of the element, the transformation is reset



Concerning the resetting of transformations, passing an empty string to the `transform` method has the effect of returning the shape to its original position.

You will notice by running the previous code that the 45 degree rotation is performed relative to its initial position, not that defined by the 180 degrees rotation. This is because every invocation of the `transform` method is relative to the original state of the element. If, however, we want to transform an element from its current state—that resulting from the previous invocation of `transform`—we can prefix our transformation commands with an ellipsis in order to append a new transformation. Consider the following transformations:

```
salinon.click(function() {
    this.transform('t 300 0 R 180');
});
salinon.dblclick(function() {
    this.transform('...t 200 0 r 45');
});
```

The first transformation, invoked on a single click, translates the element 300px in *x* and also rotates it by 180 degrees clockwise. The second transformation, invoked on a double-click, where the start command is prefixed by an ellipsis, is translated by an additional 200px and rotated 45 degrees clockwise relative to the current rotation. Note that the additional 100px translation in *x* is in the negative x-direction, this is because we take the previous 180 degrees rotation into account. This means that our salinon now "faces" the opposite way.

Note that the `this` keyword refers to the element to which the event handler is bound.

Unregistering basic event handlers

In order to unregister event handlers, Raphaël provides us with methods whose names are the names of the registering event handler prefixed with "un-". The click event handler, for example, is unregistered using `unclick`.

The unregistering event handlers are methods of the element to which you have registered event handlers and accepted as a parameter the event handling function. Consider a square that is rotated by 15 degrees clockwise relative to its current position when clicked:

```
var square = paper.rect(50, 50, 100, 100);
var clickHandler = function() {
    this.transform('...r 15');
};
square.click(clickHandler);
```

We have defined an event handler, `clickHandler`, in our current scope and passed this as a parameter to our `click` method in order to register it. We unregister the click event handler on the square as follows:

```
circle.dblclick(function() {
    square.unclick(clickHandler);
}
```

This has the effect of `clickHandler` no longer being called when the circle is double-clicked.

You can register and unregister multiple event handlers for an event on a single element. The handlers are run in the order in which they're defined.

Working with matrices

A **matrix** is a rectangular array of elements arranged in rows and columns. It is convention to specify the number of rows first when referring to the size of a matrix, so a 3-by-2 matrix, for example, would be:

$$\begin{bmatrix} 3 & 4 \\ 1 & 2 \\ 4 & 7 \end{bmatrix}$$

Each element in a matrix can be thought of as a component that maps some point to another point. The point *(x, y)*, for example, can be mapped to *(p, q)* as follows:

$$\begin{pmatrix} p \\ q \end{pmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x + 0y \\ 1x + 2y \end{pmatrix} = \begin{pmatrix} 2x \\ x + 2y \end{pmatrix}$$

This matrix defines a transformation from one point in 2D space to a new point in 2D space. The new *x*-point, `p`, is scaled by 2 while the new *y*-point, `q`, is scaled by 2 and translated by the original x (this effect is known as **shear**).

Transformation matrices

Transformations in Raphaël are known as **affine transformations**. The idea is that we treat every point in 2D space as a point in 3D space, which allows us to represent a transformation using matrices. The 3D point is fixed at 1 so that an *(x, y)* point is represented as *(x, y, 1)*, that is, it does not vary in the third dimension.

In this notation, a point resulting from a transformation, *(p, q)*, is derived as follows:

$$\begin{pmatrix} p \\ q \\ 1 \end{pmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The elements `a`, `b`, `c`, `d`, `e`, and `f` represent a variety of transformations depending on the values assigned to them.

Using transformation matrices

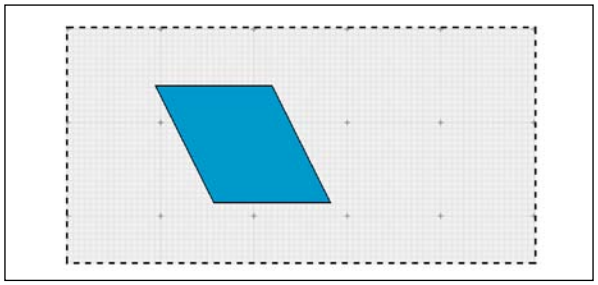
When creating transformations, you can pass matrix parameters directly to a transformation string using the syntax `"M  a, b, c, d, e, f"`. This has the effect of applying a transformation matrix to every point on an element.

Using transformation matrices to perform shear

The matrix for shear transformations is as follows:

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & Sx & 0 \\ Sy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`Sx` and `Sy` represent the factor by which *x* and *y* points are sheared respectively. For example, applying the transformation string `"m 1 0 0.5 1 0 0"` (`Sx` = 0.5, `Sy` = 0) to a square has the effect of transforming it into a rhombus in *x*:



Usually we want to apply a shear at angles relative to the *x* and *y* axis respectively. This is demonstrated at `http://raphaeljsvectorgraphics.com/the-graphical-web/css3-raphael-transforms`. In order to achieve this, we use the following transformation matrix:

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & \tan(\alpha) & 0 \\ \tan(\beta) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In this form, the angle β is the angle of shear clockwise relative to the *x*-axis (giving a resultant shear in *y*) and α is the angle of shear counterclockwise relative to the *y*-axis (giving a resultant shear in *x*). Consider the following transformation on a square:

```
Square.transform(
    ['m', 1, 0, Math.tan(Raphael.rad(25)), 1, 0, 0]
);
```

A 25 degrees transformation is applied relative to the *y*-axis giving a shear in *x*.

The drag-and-drop functionality

The drag-and-drop functionality is a powerful feature of Raphaël. In particular, we are offered a good deal of control over `drag` events, which affords us the ability to create more usable applications.

The Element.drag() method

The `drag` method of an element has the following syntax:

```
Element.drag(
    onmove, onstart, onend, [mcontext], [scontext], [econtext]
)
```

The `onmove`, `onstart`, and `onend` functions are callback functions invoked during the life cycle of a dragged element, while `mcontext`, `scontext`, and `econtext` define the `this` variable in the scope of these functions (defaulting to the element that is being dragged).

The onstart event handler

The `onstart` event handler is called when element dragging is first initiated. It is usually used as an initializer to mark the start *x* and *y* points of the element that we are modifying.

The onend event handler

The `onend` event handler is called when element dragging finishes, for example, when the mouse is released after moving an element from one point to another. It is commonly used to:

- Reinitialize an element after dragging
- Check the current position of an element in our drawing context
- Define any other actions that should occur subsequent to a drag-and-drop

The onmove event handler

The `onmove` event handler is called every time an element is moved during dragging. It has the parameters `dx` and `dy` that describe the amount by which the element has been dragged in *x* and *y* respectively.

Note that `dx` and `dy` are commonly used in mathematics to refer to small changes in *x* and *y*. Every time that the `onmove` event handler is called, its `dx` and `dy` parameters will have values representing a small change in each direction.

Dragging by example

Consider for example dragging a square. We first define our element and keep a reference to its start *x* and *y* positions in the `onstart` event handler. To highlight that our element is in a "drag" state, we also paint it pink. In our `onend` event handler, we reset the fill to be black to indicate that dragging has finished:

```
var startX, startY;
function onstart() {
    startX = this.attr('x');
    startY = this.attr('y');
    this.attr('fill', 'pink');
}

function onend() {
    this.attr('fill', '#000');
}
```

Our `onmove` event handler is responsible for setting the *x* and *y* positions of the element based on the original *x* and *y* positions plus `dx` and `dy` as follows:

```
function onmove(dx, dy) {
    this.attr({
        x: startX + dx,
        y: startY + dy
    });
}
```

Finally, we invoke drag on our element, passing in the individual event handlers:

```
square.drag(onmove, onstart, onend);
```

The resultant transformation is demoed in the sample code accompanying this chapter.

Dropping elements

At the end of dragging an element, we can perform checks to see where it has ended up relative to other elements. A **drop** is essentially a check to see whether our element falls within some particular set of bounds.

Bounding box overlapping

Every element has a bounding box that defines the smallest rectangular region containing all points of an element. By checking to see whether two bounding boxes overlap we can determine whether or not an element was "dropped" on to another.

Consider defining a target rectangle alongside our square:

```
var target = paper.rect(400, 100, 200, 200);
var square = paper.rect(100, 100, 100, 100);
```

Our `onstart` and `onmove` event handlers are defined as per the example in the previous section and are responsible for initialization and movement respectively.

This time, in our `onend` event handler, we get the bounding boxes of each of our square and target element using the `getBBox` method and then check whether or not the elements overlap using `isBBoxIntersect`:

```
function onend() {
    var bBox = this.getBBox(),
        targetBBox = target.getBBox();

    if(Raphael.isBBoxIntersect(bBox, targetBBox)) {
        this.undrag();
        this.attr({
            x: 450,
            y: 150
        });
    }
}
```

When the elements do overlap, we remove all `drag` event handlers on the element by invoking `undrag` and setting its *x* and *y* positions to some arbitrary values that fall within the bounding box of our target.

Bounding box inside bounding box

While the `isBBoxIntersect` method is convenient, we sometimes need to know whether or not all points on an element are inside another. Going back to the previous example, we could check to see whether the top-left vertex and bottom-right vertex of our square's bounding box fall within the target bounding box using the `isPointInsideBBox` method like so:

```
function onend() {
    var targetBBox = target.getBBox();
    var topLeftVertex = {
        x: this.attr('x'),
        y: this.attr('y')
    };
    var bottomRightVertex = {
        x: this.attr('x') + this.attr('width'),
        y: this.attr('y') + this.attr('height')
    };

    var point1InsideBox = Raphael.isPointInsideBBox(
        targetBBox, topLeftVertex.x, topLeftVertex.y
    );
    var point2InsideBox = Raphael.isPointInsideBBox(
        targetBBox, bottomRightVertex.x, bottomRightVertex.y
    );
    if(point1InsideBox && point2InsideBox) {
        this.undrag();
        this.attr({ x: 450, y: 150 });
    }
}
```

The top-left vertex of the square is defined by its `x` and `y` attribute while the bottom-right vertex is described by the *(x, y)* point plus the width and height of the element. Only when both points lie within the bounding box of our target do we uninitialize the `drag` event handling.

This approach works really well for target elements whose overall shape is defined accurately by a rectangular bounding box. In those cases where a shape is not defined particularly well by a rectangular bounding box, a more suitable mathematical approach would be sought.

Summary

In this chapter we have covered a lot of ground relating to both event handling and transformations. You should be familiar with some of the ways that we are able to interact with our drawings and transform them at will.

We are in a prime position now to look at truly bringing our drawings to life by animating them and will do so in the next chapter.

5
Vector Animation

Till now we have looked at drawing static graphics and interacting with and transforming them instantaneously. Animation affords us the ability to change graphics with regard to time, that is, we can modify the attributes of a vector graphic over some non-zero time interval.

Prior to the emergence of the JavaScript frameworks of the early 2000s, animation was largely achieved by Flash and to a lesser extent by Java applets. With the advent of these frameworks, animation of the DOM lead to slicker and more intuitive user interfaces and more semantic websites. In more recent years, HTML5, CSS3, WebGL, and SVG afford developers the ability to create more complex, fully fledged animations and interactive user interfaces.

Raphaël makes animating vector graphics straightforward and in the course of this chapter we look at some of the different ways you can animate graphics using the library. We will be covering the following topics in particular:

- Animating method and basic attribute animation
- Animating paths
- Easing using the cubic Bézier syntax
- Animating transformations
- Animation using custom attributes and animation along a path

Basic animation

Animation is accomplished using the `animate` method of an element. The method takes the following parameters in the order that they're defined:

Parameter	Type	Description
attributes	`Object`	Key-value pair attributes as per the `Element.attr()` method
duration	`Number`	The number of milliseconds that the duration should last for
easing (optional)	`String`	A string describing the rate of change of an attribute with regard to time
callback (optional)	`Function`	A function that is executed at the point at which the animation ends

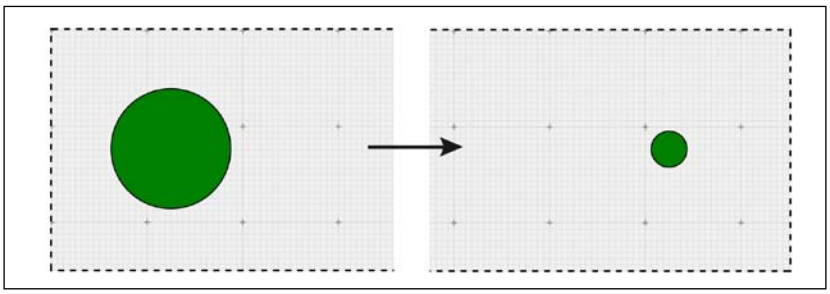
To demonstrate animation, consider a circle whose center point is at (100, 100) with radius 50px:

```
var circle = paper.circle(100, 100, 50).attr({fill: 'green'});
```

We could animate the circle to a point (500, 100) and a reduced radius of 15px as follows:

```
circle.animate({
    cx: 500,
    r: 15
}, 1000);
```

The circle is animated over a period of a second and its initial and final states are shown as follows:

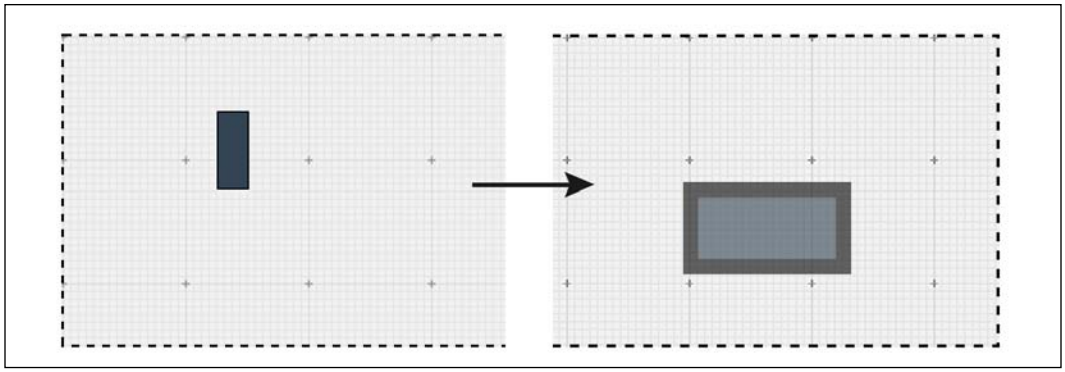


To really appreciate the examples given in this chapter you should make recourse to the source code demos or site accompanying this book.

Likewise, a rectangle at (100, 50) is animated to a point (400, 100) in *x* and *y* with increased width, reduced opacity, and a larger stroke-width as follows:

```
var rectangle = paper.rect(100, 50, 20, 50).attr({
    fill: '#345'
});
rectangle.animate({
    opacity: 0.6,
    x: 400,
    y: 100,
    width: 100,
    'stroke-width': 10
}, 1000);
```

The result is as follows:



Animating paths

The `path` attribute of an element allows us to animate paths in the same way that we animated attributes in the previous section. While the basic shapes have intrinsic *x* and *y* or `cx` and `cy` attributes and width and height, paths do not and so we will animate custom paths from one path to another.

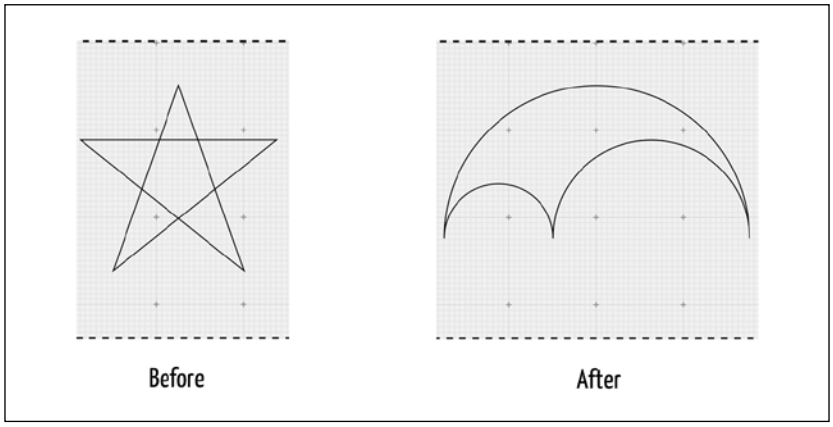
Consider drawing a star as follows:

```
var star = paper.path(
    'M 100,40 40,210 190,90 10,90 160,210 z'
);
```

The star is a polygon whose lines overlap. This makes for an interesting effect when animating the star to an arbelos shape. To demonstrate this, we define a square that when clicked initializes a path animation as follows:

```
var square = paper.rect(500, 100, 50, 50).attr({
    'fill': '#09c', cursor: 'pointer'
});
square.click(function() {
    star.animate({
        path: [
            'M', 100, 180,
            'a', 140, 140, 0, 0, 1, 280, 0,
            90, 90, 0, 0, 0, -180, 0,
            50, 50, 0, 0, 0, -100, 0
        ]
    }, 1000);
});
```

When running this code, you will see that the triangle is smoothly translated into an arbelos as shown:



There is a particularly good example of path animation on the Raphaël website at `http://raphaeljs.com/chart.html`. The example demonstrates the cubic Bézier curve animation when toggling through months. We can achieve something similar using the Catmull-Rom curves we encountered in *Chapter 3, Drawing Paths*. We begin by defining initial and end Catmull-Rom paths and drawing the first one on to our canvas:

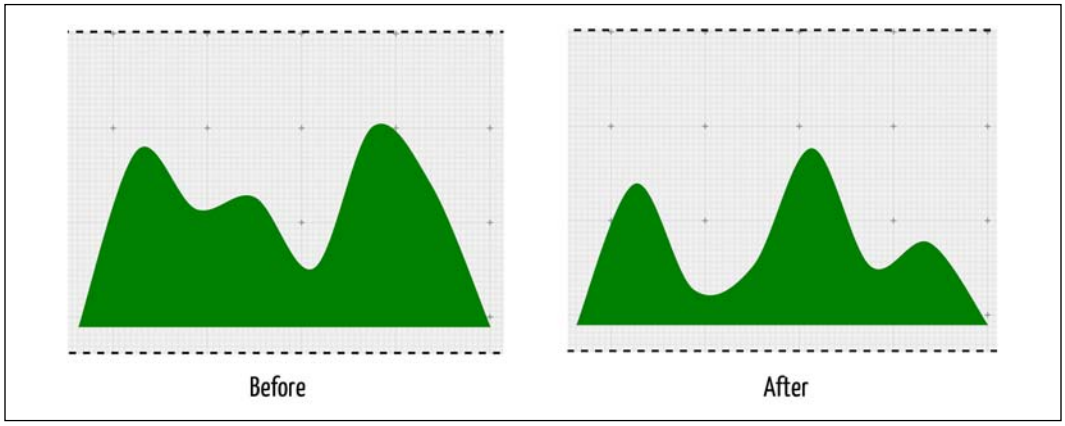
```
var pathStart = [
    'M', 50, 250,
    'R', 100, 100, 150, 150, 200, 140, 250, 200, 300, 80,
```

```
            350, 130, 400, 250,
        'H', 50,
        'z'
    ];
    var pathEnd = [
        'M', 50, 250,
        'R', 100, 130, 150, 220, 200, 200, 250, 100, 300, 200,
            350, 180, 400, 250,
        'H', 50,
        'z'
    ];
    var curve = paper.path(pathStart).attr({
        'stroke-width': 0,
        'stroke-linejoin': 'round'
        'fill': 'green'
    });
```

You will notice that we have joined a straight line to the start point on each path. This is because simply closing a Catmull-Rom curve will continue with the curvature of the path and not give us a straight line. On clicking on a square, we animate the curve to the end path as follows:

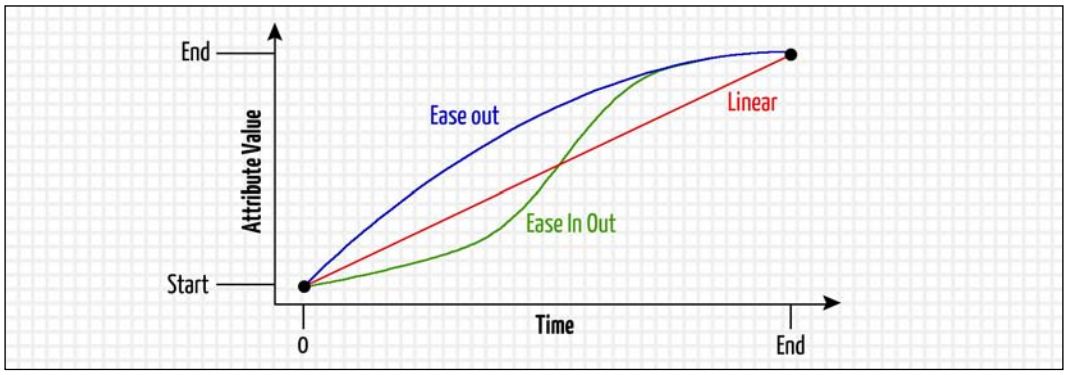
```
square.click(function() {
    curve.animate({
        path: pathEnd
    }, 500);
});
```

The result is shown here:



Animation easing

Easing describes how the value of an attribute varies with regard to time. By default, the value of an attribute changes consistently—that is, *linearly*—over the course of an animation but by specifying a particular easing type, we can change the way in which the attribute is animated. Consider the following graph that demonstrates three of the available easing types:



For each easing type, the rate at which the attribute changes varies along the graph in the time axis. Each easing type can be described as such:

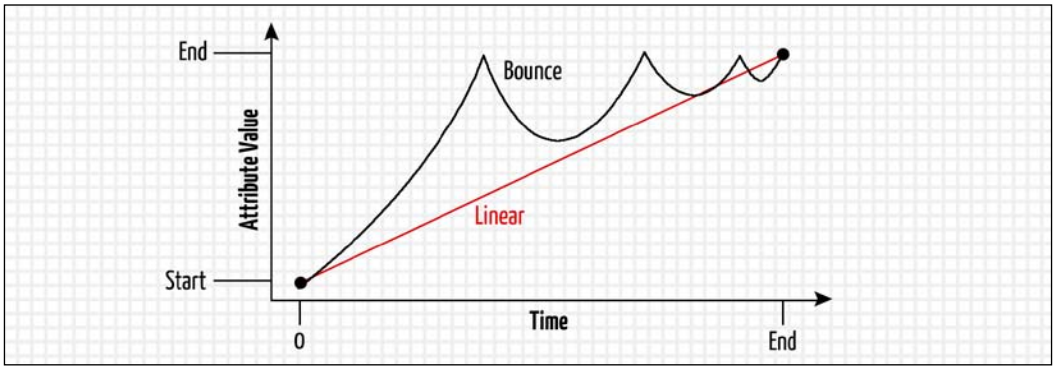
- **Linear**: The value varies consistently from its start value to its end value over the course of the animation
- **Ease Out**: The value increases quickly towards its end point before slowing down towards the end of the animation
- **Ease In Out**: The value decreases slowly at first and then increases quickly before finally slowing down to its end point towards the end of the animation

Built-in easing formulas

Raphaël has a number of built-in easing types including those detailed in the previous section and others such as `'elastic'` or `'bounce'`. The effect of each is well-demonstrated at `http://raphaeljs.com/easing.html` wherein the `cx` and `opacity` of circles are animated with different easing types. As an example, consider applying an easing type of `'bounce'` to our Catmull-Rom path animation curve:

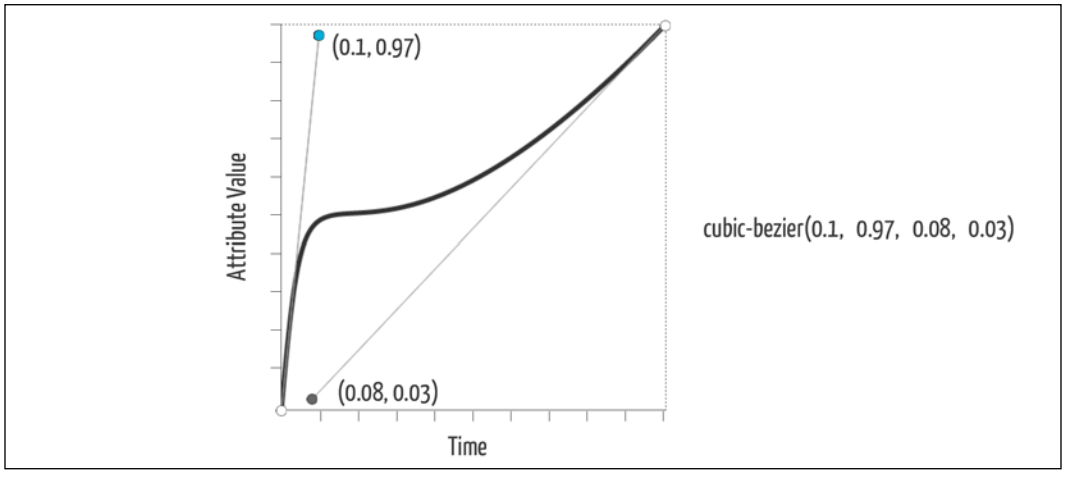
```
square.click(function() {
    curve.animate({
        path: pathEnd
    }, 500, 'bounce');
});
```

In this instance, the attribute value is `path`, which is quickly animated to its end point and then back again until finally coming to rest. The change in the path with regard to time is illustrated here:



Custom easing using the cubic Bézier format

The format for CSS3 transitions describing animation easing can be used to define custom easing paths in Raphaël. The syntax is `'cubic-bezier(point1_x, point1_y, point2_x, point2_y)'`, where each parameter defines the *x* and *y* positions of anchor points describing a cubic Bézier curve from (0, 0) to (1,1) in attribute value and time. Consider the easing path that describes an attribute value that increases very sharply and then almost comes to a stop before finally easing in to its end point:



The CSS3 syntax that describes this path is `cubic-bezier(0.1, 0.97, 0.08, 0.03)`. This was generated using a useful tool at `http://www.roblaplaca.com/examples/bezierBuilder/` that allows you to visually construct your custom easing path and get the `cubic-bezier` values for the resultant easing.

By way of example consider two circles, one of which is animated in `cx` with this easing while the other has linear easing:

```
circle.animate({
    cx: 450
}, 1000, "cubic-bezier(0.10,0.97,0.08,0.03)");

circle2.animate({
    cx: 450
}, 1000, 'linear');
```

When this code is run you will see that the first circle sets off much quicker than the second but by the end of the animation the `cx` value varies approximately linearly for both circles.

Animating transformations

As with paths, an element's transform is an attribute on the element meaning we can animate in the same manner. Consider, for instance, a square that we can rotate on a click:

```
square.click(function() {
    this.animate({
        transform: ['...r 45']
    }, 300);
});
```

The ellipsis ensures that every successive click will rotate the square from its current position by 45 degrees in the clockwise direction.

Similarly, we can animate more complex transformations. Consider, for instance, successively animating a square with a 5-degree shear in *x*:

```
var anim = Raphael.animation({
    transform: [
        '...', 'M', 1, 0, Math.tan(Raphael.rad(5)), 1, 0, 0
    ]
}, 300);

square.click(function() {
    this.animate(anim);
});
```

Successive clicks of the square give rise to an additional shear of five degrees relative to the *y*-axis. You will notice that in this example we have defined an `animation` object and then passed this to the `animate` method on the square when clicked. This is an alternative way of defining animations and is useful in that we can re-use them.

Animation using custom attributes

Raphaël facilitates advanced animation via **custom attributes**. Custom attributes can be used to apply, or animate, multiple attributes on a particular element based on custom-defined logic.

Custom attributes

A custom attribute is a custom-defined function that returns a set of attributes to be applied to an element. It can be thought of as a helper function for which existing attributes are derived based on calculation. A custom attribute is defined as an attribute of the `customAttributes` namespace as follows:

```
Paper.customAttributes.yourAttribute = function(a1, a2, ...){};
```

Here, the `a1`, `a2`, ... arguments are numeric and `yourAttribute` is the name of the custom attribute. A custom attribute can then be used in the same ways that we have used other element attributes so far.

As an example, consider that we need to represent population data using the following:

- The radius of a circle to indicate relative population size
- The color of a circle to indicate how densely populated a country is

The following data represents the relative populations and population densities of five countries:

Country	Population (normalized between 0 and 1)	Population density (normalized between 0 and 1)
Taiwan	0.10	1.00
South Korea	0.32	0.50
Netherlands	0.05	0.22
Belgium	0.00	0.07
Japan	1.00	0.00

We define a custom attribute, `popDensity`, for a visual representation of this data as per the aforementioned rules:

```
paper.customAttributes.popDensity = function(population, density) {
    var radius = 25 + (25 * population),
        fillColor = 'hsb(0, 0,' + (1 - density) + ')';

    return {
        fill: fillColor,
        r: radius
    };
};
```

This method defines a radius that is 25px plus 25 multiplied by the relative population of the country meaning that the most populous country has a radius of 50px, the least populous has a radius of 25px and all other countries have a radius somewhere in between these two values. The fill color is determined as a proportion of the brightness where hue and saturation are fixed. The formula, `(1 - density)`, will return a value of 1 (a saturation giving a white color) for the least densely populated countries and a value of 0 (a saturation giving a black color) for the most densely populated countries. All values of saturation in between 0 and 1 give rise to shades of gray.

Given that we define our dataset as an array of objects like so:

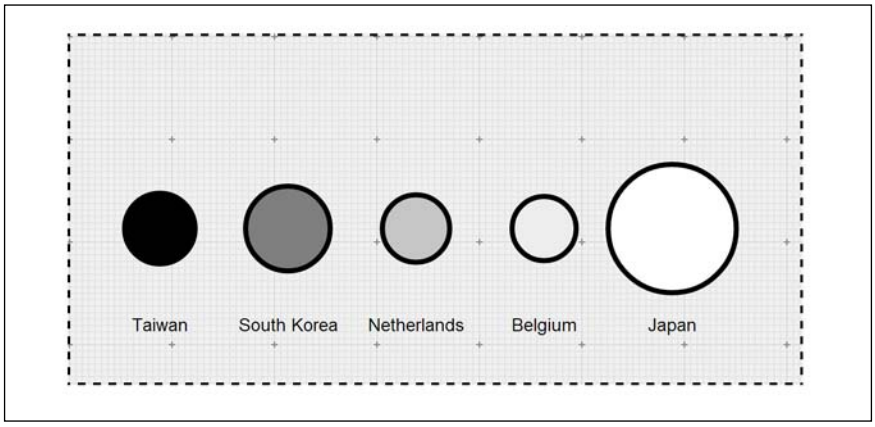
```
var data = [
    {
        name: 'Taiwan',
        population: 0.1,
        density: 1
    },
    {
        name: 'South Korea',
        population: 0.32,
        density: 0.5
    },
   // ...
];
```

We can iterate over each country and draw a circle with a `popDensity` attribute as follows:

```
var currentX = 70;
for(var i = 0, ii = data.length; i < ii; i+=1) {
    var country = data[i];
    paper.circle(currentX, 150, 50).attr({
```

```
            popDensity: [country.population, country.density],
            'stroke-width': 4
        });
        paper.text(currentX, 225, country.name).attr({
            'font-size': 14
        });

        currentX += 100;
    }
```

During each iteration of the loop, the population density attributes are determined from our data objects and are drawn to our canvas. We have labeled each circle with the country's name as text and incremented the *x*-point so that successive circles are drawn to the right of the previous one. The result is shown here:



As you can see, the most populous countries do not necessarily have the largest population densities. While Taiwan and the Netherlands have roughly equal populations, Taiwan has significantly more people per square kilometer than the Netherlands.

As mentioned, we can animate custom attributes from one set of values to another set of values. Consider what happens if we set the population and density values equal to zero for all of our elements when they are initialized in the loop and then animate to the real values of population and density:

```
var currentX = 70;
for(var i = 0, ii = data.length; i < ii; i+=1) {
    var country = data[i];
    var circle = paper.circle(currentX, 150, 50).attr({
        popDensity: [0, 0],
        'stroke-width': 4
```

```
        });

        circle.animate({
            popDensity: [country.population, country.density]
        }, 1000, 'easeOut');

        paper.text(currentX, 225, country.name).attr({
            'font-size': 14
        });

        currentX += 100;
    }
```

You will notice when running this code that both the radius and fill attributes are animated to their final states.

Animation along a path

Over the course of an animation, attribute values vary according to the easing specified and the time that has elapsed. Custom attributes work in the same way. In that, the values passed to a custom attribute function are dependent on the easing and the time elapsed. We can utilize this feature to animate elements along a given path.

To begin, we draw a cubic Bézier curve along which a circle will be animated:

```
var path = paper.path([
    'M', 100, 100,
    'C', 100, 0, 400, 200, 400, 100,
    'S', 100, 200, 100, 100,
    'z'
]).attr({'stroke-width': 2});

var circle = paper.circle(0, 0, 13).attr({
    fill: '#09c', cursor: 'pointer'
});
```

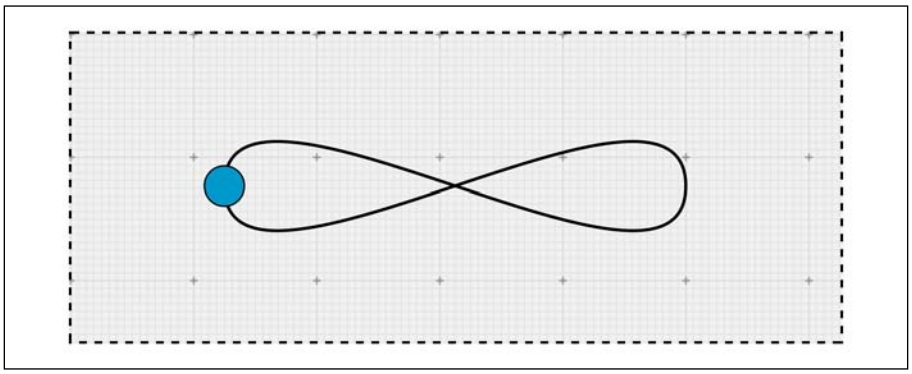
Our circle is initialized to the (0, 0) point since we are going to apply a custom attribute to it to determine its initial position.

Our custom attribute, which we'll name `pathFactor`, determines the point on the path at which our circle should lie given how much of the animation has elapsed. If we define *distance = 0* as the start point of our animation and *distance = 1* as the end point, we can multiply distance by the total length of the path to determine how much of its length our circle should traverse:

```
paper.customAttributes.pathFactor = function(distance) {
    var point = path.getPointAtLength(distance * pathLength);
    var dx = point.x,
        dy = point.y;
    return {
        transform: ['T', dx, dy]
    };
};
```

We always return an absolute transformation that uses the *(x, y)* point on the path to determine where on the path our circle lies.

In order to animate our circle, we first initialize its `pathFactor` attribute to zero. This has the effect of placing it at the zero-length point of our path, which is the point at which we started drawing it, which is, (100, 100):



To animate our circle along the path, we define and execute a `runloop` function that initializes an animation of our `pathFactor` attribute to a value of 1. Our end of the animation callback resets `pathFactor` to zero and then invokes `runloop` recursively. You will notice that our recursive function call is done as a callback to `setTimeout`; this is to ensure that the minimum time required for the browser to progress to the next iteration in the cycle elapses before the animation is repeated:

```
function runloop() {
    circle.animate({pathFactor: 1}, 4000, function() {
        this.attr({pathFactor: 0}); // Reset
            setTimeout(runloop);
        });
    });
};
runloop();
```

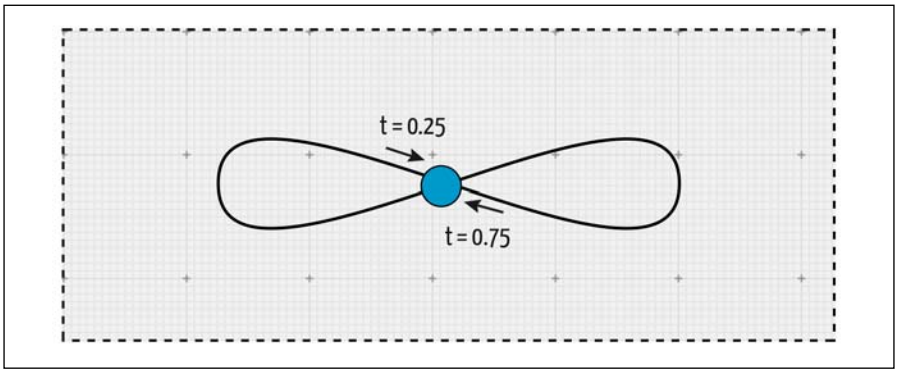
When this code is executed, you will see that our circle is animated along our path forever, taking four seconds to traverse the path from the start point and back again.

Pausing and resuming animation

We have not yet looked at the ability to pause and resume animations. Consider, in the example of animating along a path, if we define click and double-click event handlers on our circle as part of our run loop like so:

```
function runloop() {
    circle.animate({pathFactor: 1}, 4000, function() {
        this.attr({pathFactor: 0}); // Reset
            setTimeout(runloop);
        });
        circle.unclick().click(function() {
            this.pause();
        });
        circle.undblclick().dblclick(function() {
            this.resume();
        });
    });
};
runloop();
```

We unregister and then re-register the click and double-click event handlers on each animation run since a new animation is started each time. When running this code, you will see that the pause and resume methods have the effect of pausing the current animation that is running on the circle when clicked and double-clicked respectively. The result of pausing the animation at a quarter of the way through its cycle (t = 0.25) or three quarters of the way through its cycle (t = 0.75) is shown:



Summary

Over the course of this chapter we have looked at the concepts driving animation and the different ways in which we can attribute elements to attributes. You should now be in a position to comfortably create, transform, and animate vector graphics using the Raphaël library and should have no trouble utilizing the Raphaël documentation to build on what we have covered and manipulate drawings to your own ends.

In the next chapter we will look at how we can work with existing SVGs in Raphaël.

6
Working with Existing SVGs

Most designers use a mixture of raster and vector graphics in their projects and so are familiar with vector graphics editors such as Adobe Illustrator and Inkscape. The usefulness of these tools has grown considerably in recent years inline with advances in digital media—in particular, designers need to be able to create resolution-independent graphics that work in different contexts.

The mobile and tablet market, for example, encourages designers to create images that scale appropriately at really low resolutions (around 320 x 240) and really high resolutions (2,048 x 1,536). Desktop browsing is pretty much the same nowadays where user experience is tailored to satisfy a range of viewport dimensions. Satisfying a range of viewport dimensions is often referred to as **responsive design**.

While SVG itself has limited support in older browsers, we can still provide vector graphics solutions. We now see that vector graphics are utilized more readily in websites and applications. By this token, we're witnessing a shift in design from rigid, albeit more polished artwork to flexible vector-based designs.

With this said, there is now greater importance placed on being able to work with the vector graphics produced by graphics editing software. While there is no silver bullet approach to working with existing SVGs in Raphaël, with a little determination we can utilize the existing SVGs in our projects.

In this chapter we will look at the following:

- The Inkscape vector graphics editor
- The manual method of inspecting SVG paths
- SVG to Raphaël conversion tools
- Choropleth maps as an example of an existing SVG

Inkscape

Inkscape is an open source SVG editor whose feature set is similar to that of Adobe Illustrator. It is used by many designers given that:

- It is open source and actively maintained / developed
- There are a good number of tutorials available online
- It has a comprehensive feature set
- It conforms to the W3C standards SVG

Inkscape also has the advantage of being cross-platform with releases available for Windows, Mac, and Linux.

Downloading Inkscape

Inkscape is available for download at `http://inkscape.org/download/`. It is recommended that you download the latest stable release, which is 0.48.4 at the time of writing.

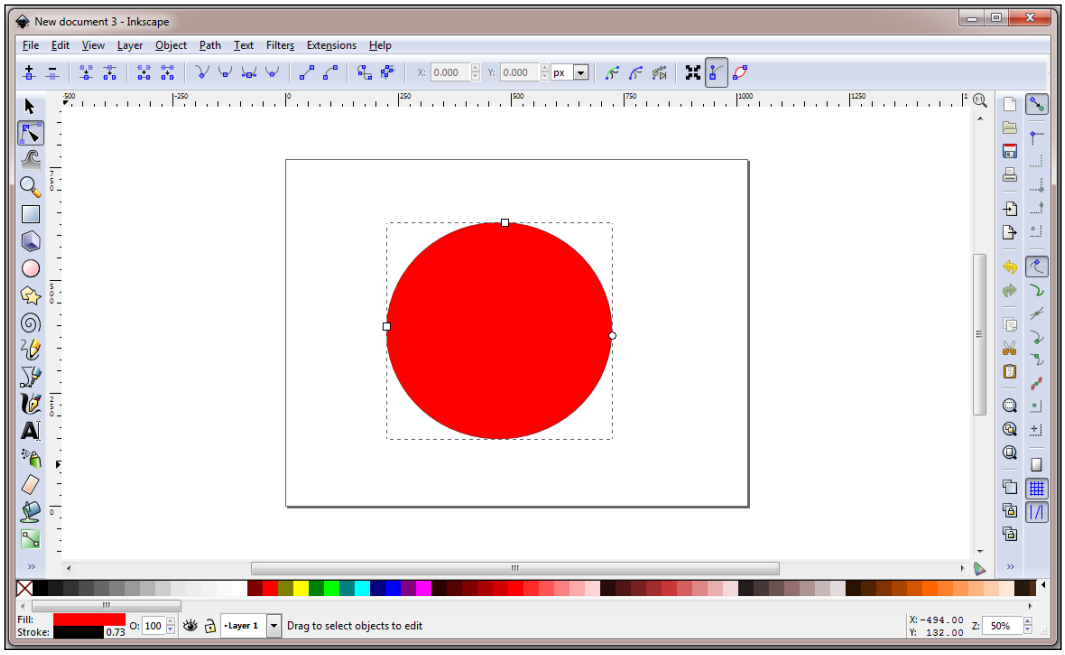
Getting started with Inkscape

While an in-depth coverage of Inkscape is beyond the scope of this book, we will look at creating some basic SVGs using Inkscape using the following steps:

1. To begin, open up Inkscape and choose **File** | **New** from the menu. Inkscape comes with a number of predefined document sizes out of the box and you will want to select the one best suited to the graphic that you're creating. For now, choose **default**.

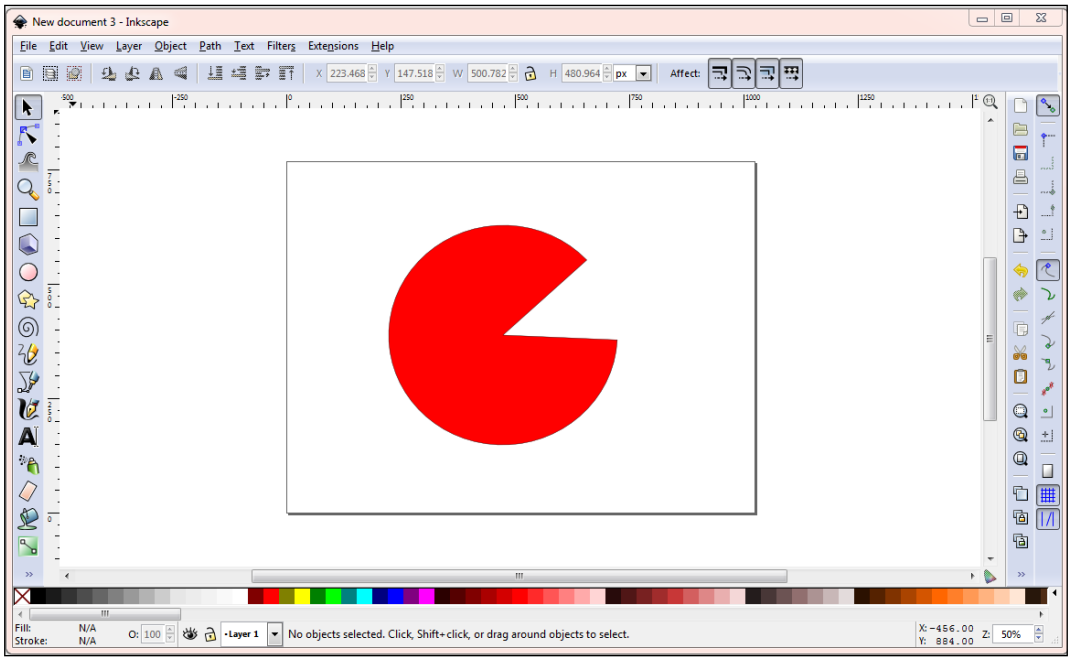
   On the left-hand side, you will see that Inkscape has a number of shape drawing tools for drawing basic shapes, text, freehand, and Bézier paths.

2. Choose the circle.
3. Click-and-drag anywhere on your canvas.
4. When releasing your mouse, your circle will look as follows:



Notice the anchor points on the circle. The leftmost point is used to control the width of the circle, the topmost point controls the height while the rightmost point is used to change the end point. Click-and-drag on the rightmost breakpoint: dragging inside the circle draws a straight line from the end point to the start point while dragging outside creates a segment.

The result of dragging outside of the circle by an angle of a few degrees is shown in the following screenshot:



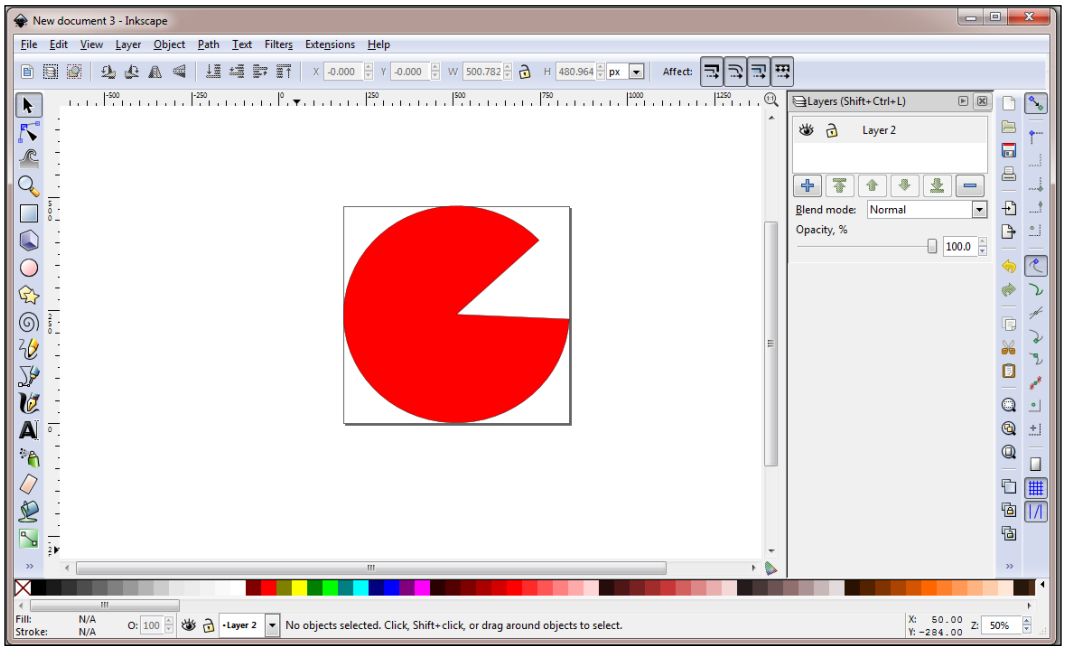
For our web-based projects, the document size is usually going to be that defined by the constituent objects. Click on **File** | **Document Properties** | **Page** and then expand the **Resize** page to a content drop-down menu. Here you can define custom widths and heights for the document. Clicking on **Resize page to drawing or selection** will resize the page and the dimensions of the selected element (or all drawn elements if none is selected).

What this actually does is it applies a translation to all elements on the current layers, that is, the paths are drawn and then a transformation is applied. If we want all paths to be drawn relative to the "new" origin point (which is usually the case) the following steps should be performed:

1. Resize the document to the desired size.
2. Go to **Layer** | **Layers...**.

3. Create a new layer by clicking on the **+** symbol on the layers pane.
4. Select all objects in the previous layer.
5. Choose **Layer | Move Selection to Layer Above**.
6. Delete the previous layer.

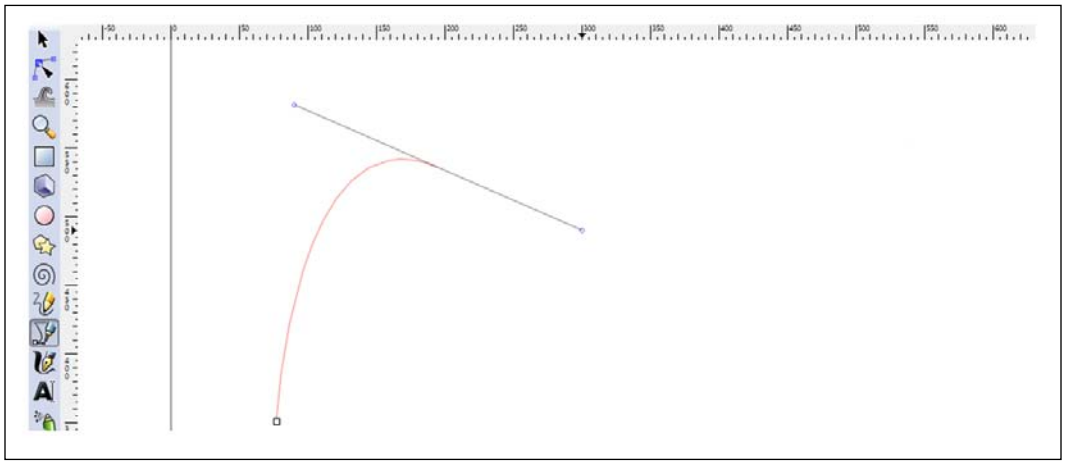
The result is as follows:



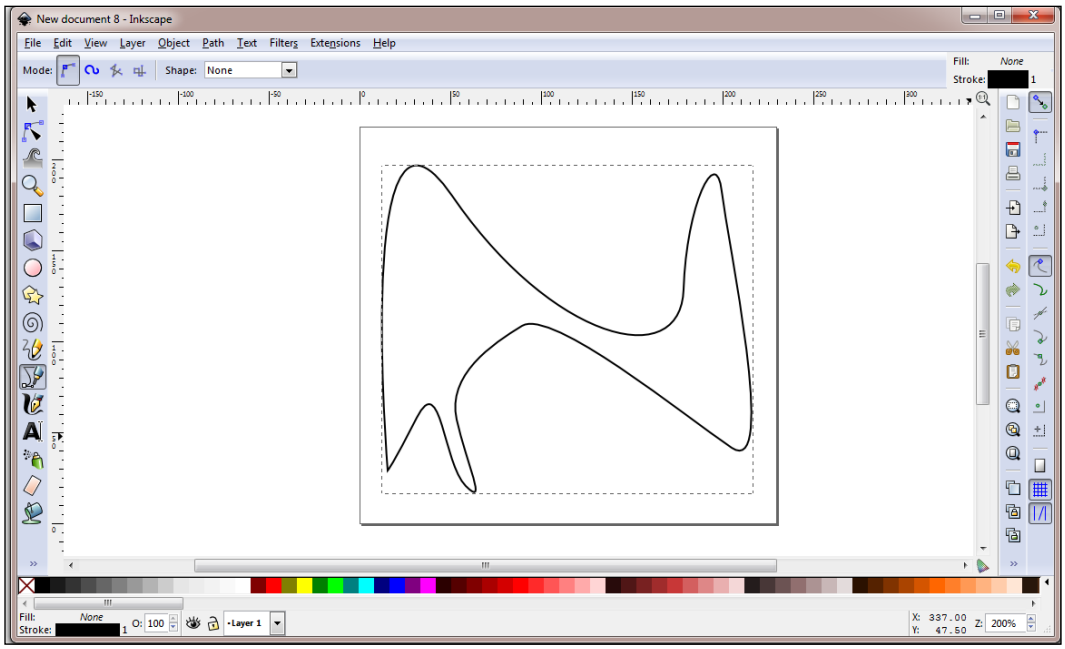
Arguably the best tool we have at our disposal is the pen tool, which is used to draw Bézier curve paths:

1. Select the pen tool (*Shift* + *F6*).
2. Single click anywhere in the document.
3. Click-and-drag at another point in the document.

You will notice that each click defines an anchor point that defines the extent to which the curve is curved as shown in the following screenshot:



Each successive click defines a new anchor point and the path is closed by joining it to the start point anchor point:



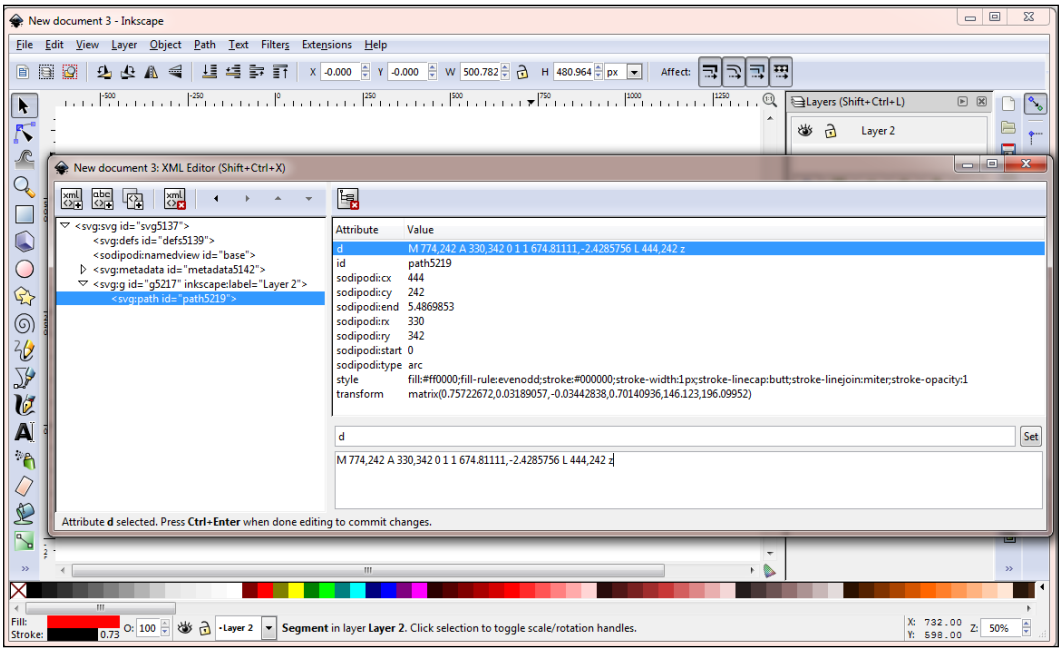
Inkscape comes with a number of tutorials at **Help** | **Tutorials**.

Inspecting paths

Since we're familiar with supplying raw path string data to the `path` method in Raphaël, it goes without saying that we can copy a path string from an SVG and draw paths this way. For complex graphics, the raw XML of an SVG is not always easy to work with and manipulating multiple paths can be cumbersome but this approach of working with existing SVGs does give us complete control over the graphics that we are manipulating.

Inkscape's XML Editor

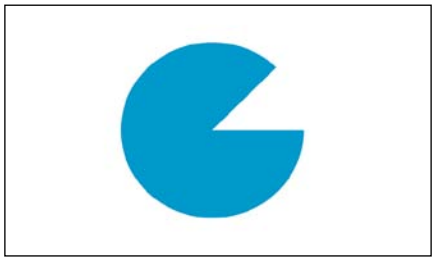
One of the great features of Inkscape is the ability to view and edit an SVG's XML markup directly. Consider the example of the circle in the previous section. If you click on **Edit** | **XML Editor**, you will be presented with the SVG's document structure. Clicking on the element you have created on the left—the `svg:path` element—will list all of the object's attributes including the `d` attribute, which is the element's path prior to the transformations being applied. Clicking on the `d` attribute reveals the path for the element:



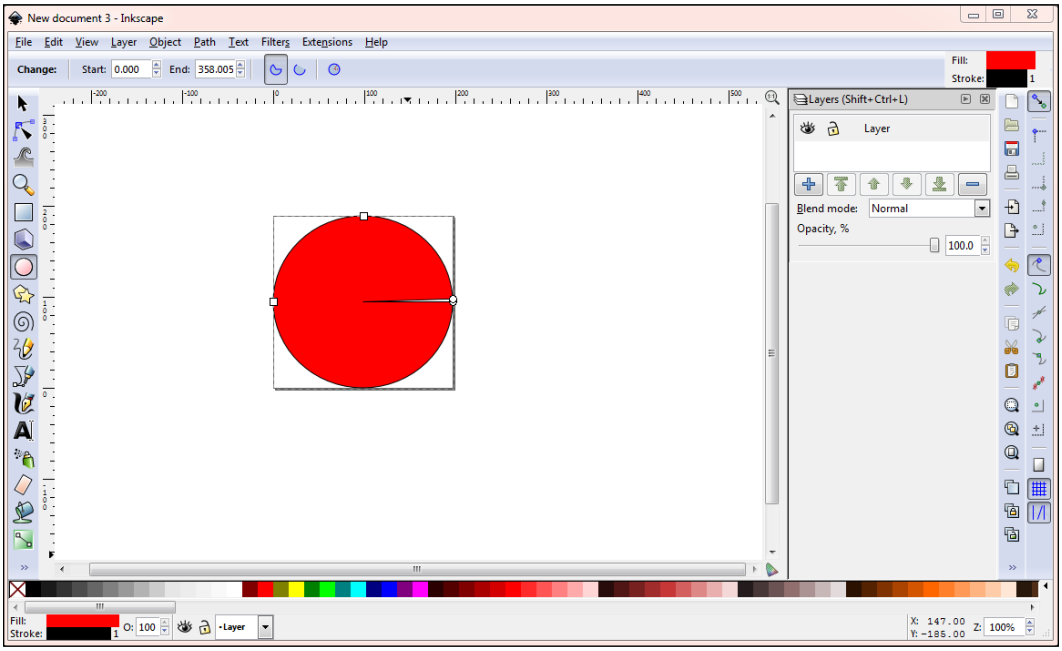
If we take this path and use it in Raphaël as follows:

```
var initialPath = 'M 197,94.46 '
    + 'A 98.5,94.5 0 1 1 167.39,26.92 L 98.5,94.46 z';
var pac = paper.path(initialPath);
pac.attr({fill: '#09c', 'stroke-width': 0});
```

Then it is drawn onto our canvas as shown:



Going back into Inkscape, move the arc anchor point so as to close the "mouth" of the segment that we created:



As before, we get the path for this element using the XML Editor.

We can achieve an interesting effect here with relatively little effort using path animation, animating from the former "open mouth" path to the latter "closed mouth" path. To begin, we define a function named `move` that determines the *x* and *y* coordinates of our `pac` element based on rotation (which is initialized to zero to begin with):

```
var rotation = 0;
var move = function() {
    var x = Math.cos(Raphael.rad(rotation)) * 800;
    var y = Math.sin(Raphael.rad(rotation)) * 800;
```

Next we animate the element to our new path or "closed mouth" path:

```
pac.animate({
    path: 'M 197,94.46 '
      + 'a 98.5,94.5 0 1 1 -2.1e-4,-0.194727, L 98.5,94.46 z',
    transform: ['...T', x, y]
}, 500, function() {
    // Rotate our element and reset the path
    rotation += 90;
    pac.attr({
        path: initialPath
     }).transform(['...R', 90]);
     setTimeout(move);
});
```

We apply a translation based on the *x* and *y* coordinates of our element over half a second. Once the animation has finished, we rotate our element. Note that the transformation is prefixed with an ellipsis that ensures that any translation takes the rotation into account after a 90-degree rotation, translations will be along the *y*-axis. This is repeated ad infinitum by calling `setTimeout` on our `move` function.

When this code is run, the "mouth" goes from being open to closed during the length of one animation and is then repeated with an applied rotation. The rotation has the effect of moving our element along the edge of our container (this demo is available in the source code and website accompanying this book).

Taking paths from an existing SVG image

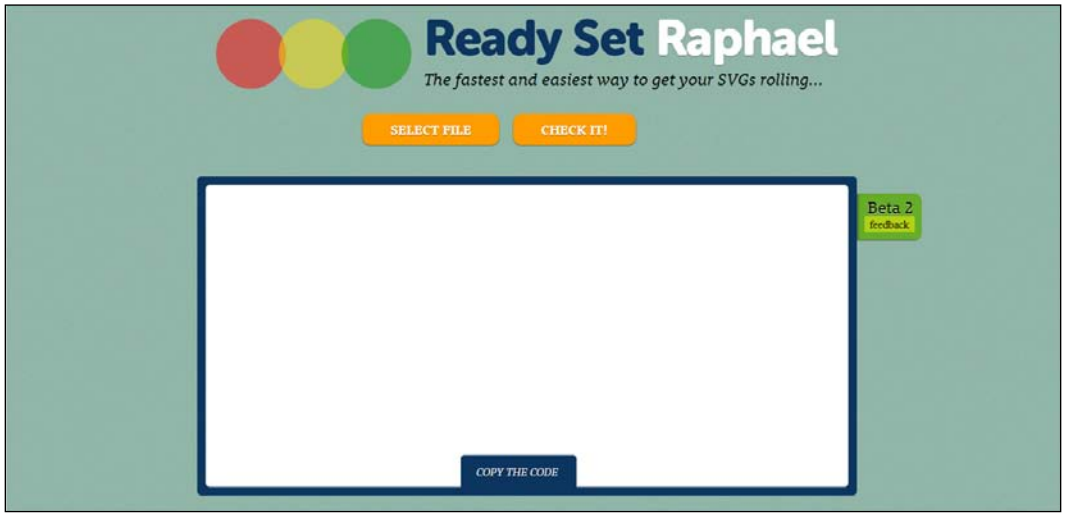
Since SVG images are just XML files, you can open them in your text editor in order to obtain the path information. Often it is just as simple as copying and pasting path information from individual `path` elements in the XML. While this is, as mentioned, cumbersome, there are other problems with this approach. The main issue is whether or not elements, or groups of elements, have had transformations applied to them. In the case where they have transformation applied to them, you will need to apply the same transformation to the `path` elements that you create.

SVG to Raphaël conversion tools

There are a number of good SVGs to Raphaël JS converters out there that will convert an SVG image into Raphaël automatically and without too much pain.

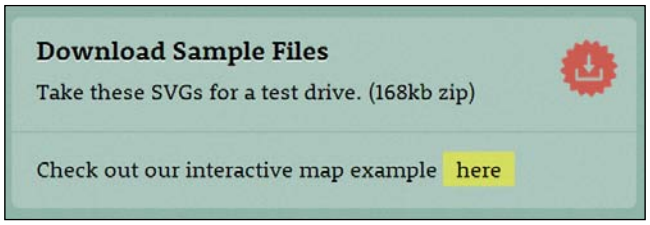
Ready Set Raphaël

Ready Set Raphaël (`http://www.readysetraphael.com`) is a web-based SVG converter that allows you to upload an image and grab the resultant Raphaël code. It supports SVG groups, ellipses, polygons, text, circles, paths, and rectangles and parses all associated styling information into element attributes:

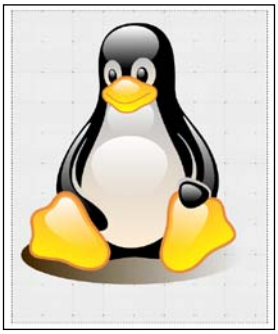


To use it, you simply upload an existing SVG image and click on the **Copy the Code** button.

To test this out, download one of the sample files from `http://www.readysetraphael.com` at the bottom of the page and copy and paste the resultant code into an HTML document between the `<script></script>` tags. Note that you will need to create an HTML element with an ID `rsr`:



The result of rendering the sample penguin file is shown here:



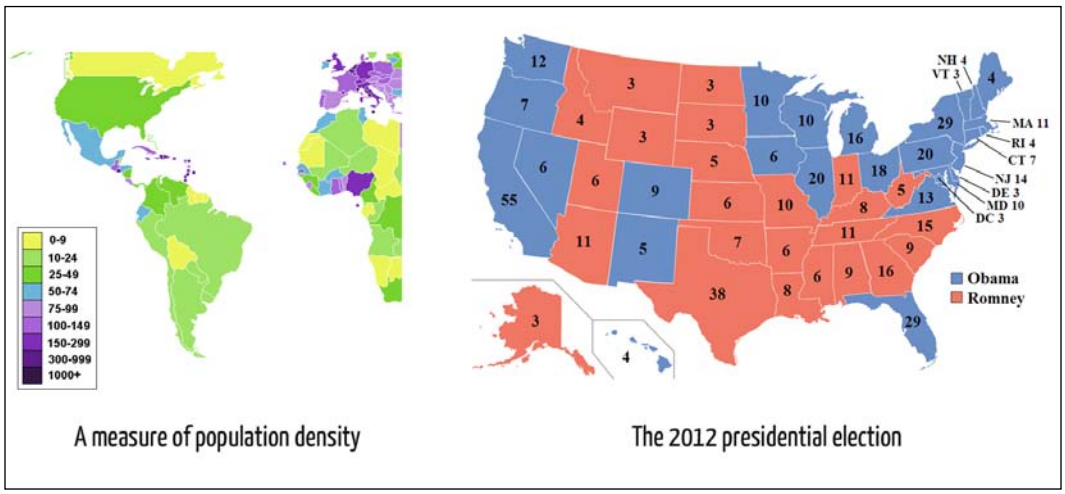
Other converters

It is worth mentioning that there are a number of other good SVG to Raphaël converters available online. You may find that each has a different result when converting your native SVG and so you should experiment to find out which works best for you.

Charles Thomas's SVG to HTML converter (`http://www.irunmywebsite.com/raphael/SVGTOHTML_LIVE.php`) converts all primitive shapes to paths and ensures that all paths use absolute coordinates while the converter at `http://toki-woki.net/p/SVG2RaphaelJS/` is fast and lightweight.

Choropleth maps

Choropleth maps are maps that show some particular area or region with an associated measurement or statistic. They are commonly used in cartography to depict population densities and similar metrics and are used regularly to visualize data with geographical significance. The following image, compiled using the `https://commons.wikimedia.org/wiki/File:World_population_density_map.PNG` and `https://en.wikipedia.org/wiki/File:ElectoralCollege2012.svg` Wikimedia Commons images, demonstrates applications in cartography (left) and in data visualization (right):



Creating choropleth maps

Often when you need to associate data with some geographical region or area, the best approach is to utilize an existing map in SVG format that you can then tailor to your needs (manually by modifying the raw XML or using a graphics editor such as Inkscape). For the purposes of demonstration, we will work with an existing map of the United Kingdom taken from `http://en.wikipedia.org/wiki/File:England_Regions_within_UK.svg`.

If you download this map and open it up in your XML editor you will see that the districts of England are defined as individual `<path>` elements with IDs on them (for example, `<path id="London">` represents the London region while `<path id="north-west">` depicts the entire North West England). Ready Set Raphael handles this really well in that it defines variables for each of the paths with IDs:

```
var london = rsr.path(...);
var northwest = rsr.path(...);
```

You can convert the aforementioned image using the Ready Set Raphael converter and import the resultant JavaScript into your HTML document. When running this code, you will see that the converter does a really good job of maintaining the paths and associated attributes.

There are a total of nine official regions (administrative divisions) of England that are all drawn on the resultant map. These regions have the following populations where the resultant variable names created by Ready Set Raphael are shown in the table:

Region	Variable name	Population
North West England	`northwest`	7,052,000
North East England	`northeast`	2,597,000
Yorkshire and the Humber	`yorkshireandhumber`	5,284,000
West Midlands	`westmidlands`	5,602,000
East Midlands	`eastmidlands`	4,533,000
East of England	`eastengland`	5,847,000
London	`london`	8,174,000
South East England	`southeast`	8,635,000
South West England	`southwest`	5,289,000

The sum of all populations is 53,013,000.

At the end of the code created by Ready Set Raphael, we define an array of objects that relate a `path` property to the path variable listed in the table with the associated populations. This rationalization is useful as it allows us to associate a region's population with the defined path:

```
var data = [
    {
        path: northwest,
        population: 7052000
     },
    {
        path: northeast,
        population: 2597000
    },
    // ... and so on ...
];
```

Now, in order to visualize the varying populations, we define a custom attribute named `shading` that returns a fill based on 0 hue and 0 saturation where only brightness varies. This will have the effect of shading the countries according to their population sizes:

```
rsr.customAttributes.shading = function(population, min, max, total) {
    var brightness =  1 - ((population - min) / (max - min));
    return {
        fill: 'hsb(0, 0, ' + brightness + ')'
    };
}
```

The brightness is defined as follows:

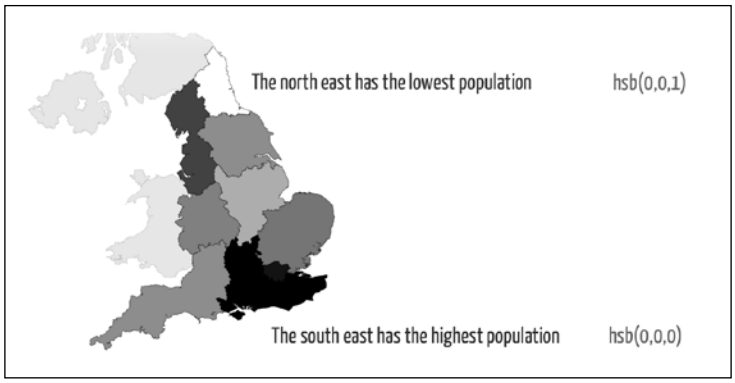
$$brightness = 1 - \frac{population - smallest\ population}{largest\ population - smallest\ population}$$

This equation normalizes the brightness based on the largest and smallest population sizes. When the population size is equal to the largest population, the brightness will be zero (black) and when the population size is equal to that of the smallest population, the brightness will be equal to 1 (white). All other values of the population return a value between 0 and 1 (a gray shade).

We utilize this attribute by iterating through our data array that defines the drawn paths and associated populations:

```
var totalPopulations = 53013000;
var minPopulation = 2597000;
var maxPopulation = 8635000;
for(var i = 0, ii = data.length; i < ii; i+=1) {
    var path = data[i].path;
    var population = data[i].population;
    path.attr({
        shading: [population, minPopulation, maxPopulation,
totalPopulations]
    });
}
```

Each region's population, as well as the total, maximum, and minimum populations, are passed to our custom attribute which in turn fills the region accordingly. The result of this operation is shown in the following image:



You can see that the north-east region—which has the lowest population—is colored white while the south-east region—which has the largest population—is colored black.

One unfortunate effect of London having a population size that is nearly as large as that of the south-east region of the country is that it is impossible to distinguish between the London region and the south-east. This can be addressed in this instance by applying a lighter-colored stroke to each region when looping through our data array:

```
path.attr({
    shading: [population, minPopulation, maxPopulation,
totalPopulations],
    'stroke': '#bbb'
});
```

The result is as shown:

Open source SVGs

There are a number of SVGs available via the Open Clip Art Library at `http://openclipart.org`. The site features an excellent SVG editor that allows you to edit hosted SVGs on the fly and view their underlying source code or download them.

Summary

You should now be comfortable working with the existing SVGs in the context of your Raphaël-driven projects. You should have a basic understanding of Inkscape and be able to utilize it to create basic SVGs and inspect the XML makeup of graphics. Online SVG to Raphaël converters are invaluable as our coverage of chloropeth maps demonstrates, and you should now be able to work with more complex graphics whose characteristics give us additional scope in what we're able to offer to the end user.

7
Creating a Suite of Social Media Visualizations

We have now covered all of the main facets of the Raphaël JavaScript library and over the course of this chapter we will look at creating data visualization demos based on social media statistics.

Data visualization has really boomed in recent years. The number of infographics and data-related expositions online continue to grow as we try to make sense of and convey the wealth of data available to us. That said, many of the fine examples in existence are *static* and there is plenty of scope for utilizing vector drawing libraries to create dynamic and interactive data visualizations, making this a very real and valid application of the Raphaël library.

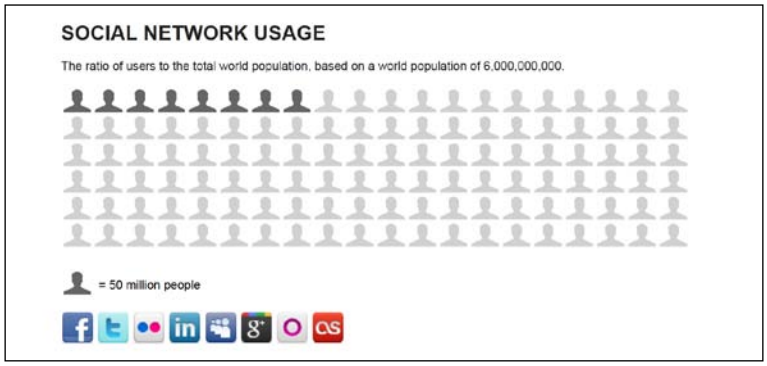
Over the course of this chapter we will look at:

- Creating a social network usage statistics data visualization
- Creating an animated donut chart showing number of tweets with regard to time
- Discussing the future of the Raphaël library

Social network usage

The idea that people could communicate and interact over a computer network goes back to, well, before the days of the World Wide Web (the first network e-mail was sent in 1971), but it is really in the past ten or so that we've started to see a widespread and indeed colossal usage of social networks. Facebook is currently the world's largest social network and boasts of one billion registered users while the likes of Twitter and Google+ have close to half a billion users each.

In this section we will create an interactive visualization similar to a histogram / area chart that demonstrates just how significant these figures are by comparing the number of social network users to the total world population (six billion). It will use a common technique of highlighting icons to indicate the proportion of users relative to all world citizens when the relevant social network is selected. The end result is shown as follows:



Getting started

To get started, we first create a basic HTML file with markup containing social media icon images and include all relevant JavaScript files just before the closing `body` tag. We will write our JavaScript in the `social-network-usage.js` file:

```
<body>
    <h1>Social Network Usage</h1>
    <p>The ratio of users to the total world population,
        based on a world population of 6,000,000,000.</h1>
    <div id="world-population-usage"></div>
    <div class="networks">
        <img class="facebook" src="icons/facebook_32.png" />
        <img class="twitter" src="icons/twitter_32.png" />
        <img class="flickr" src="icons/flickr_32.png" />
        <img class="linkedin" src="icons/linkedin_32.png" />
        <img class="myspace" src="icons/myspace_32.png" />
        <img class="googleplus" src="icons/google_plus_32.png" />
        <img class="orkut" src="icons/orkut_32.png" />
        <img class="lastfm" src="icons/lastfm_32.png" />
    </div>
    <script type="text/javascript"
        src="raphael-2.1.0-min.js"></script>
    <script type="text/javascript"
        src="jquery-1.9.1.min.js"></script>
    <script type="text/javascript"
        src="social-network-usage.js"></script>
</body>
```

When a social media icon is clicked on, the number of highlighted persons will be updated to reflect the total number of users of that network. The icons are provided courtesy of `komodomedia.com` and are distributed along with the source code accompanying this book.

Using jQuery

You will notice in the markup of the previous section that we have included the jQuery JavaScript library. This will be used to facilitate DOM interaction throughout this chapter.

You can download jQuery from `http://jquery.com/`. The latest version at the time of writing was 1.9.1 and it is the version we will use throughout this chapter.

For those of you that are not familiar with jQuery, there are many resources available online. Your first port of call should be `http://docs.jquery.com/`.

Social network usage data

The data used for this visualization is taken from `http://en.wikipedia.org/wiki/List_of_social_networking_websites` where usage is measured based on the number of registered users and this figure is rounded to the nearest 50 million users:

Social network	Number of registered users (rounded to nearest 50,000,000)
Facebook	1,000,000,000
Twitter	500,000,000
Flickr	50,000,000
LinkedIn	150,000,000
Myspace	50,000,000
Google+	400,000,000
Orkut	100,000,000
last.fm	50,000,000

We define this data as an object where the keys correspond to the class names defined in the preceding markup and the values are the number of registered users:

```
var usage = {
    facebook: 1000000000,
    twitter: 500000000,
    flickr: 50000000,
    linkedin: 150000000,
    myspace: 50000000,
    googleplus: 400000000,
    orkut: 100000000,
    lastfm: 50000000
};
```

Drawing people icons

As usual, we define our canvas and store a reference to it in the `paper` variable. Based on the default width and height of the icons we're using, a 770px wide and 220px high canvas suits our purposes nicely:

```
var paper = Raphael('world-population-usage', 770, 220);
```

The person icon that we are using comes courtesy of Dmitry Baranovskiy who has created no less than 266 icons as paths at `http://raphaeljs.com/icons/`. Each icon on this page comes with an associated path string that we can use directly in our code. We hold a reference to one of the path strings in a variable called `personPath` so that we may re-use it (note that the path shown here is truncated for the sake of brevity):

```
var personPath = "M20.771,12.364c0,0,0.849-3.51 ..."
```

Given this, we now draw 120 of these icons in six rows of 20 each. Each icon represents 50 million people meaning that 120 icons represent the world population of 6 billion people:

```
var people = paper.set();
for(var j = 0; j < 6; j+=1) {
    for(var i = 0; i < 20; i+=1) {
        var path = paper.path(personPath).attr({
            fill: '#666',
            'stroke-width': 0,
            opacity: 0.3
        });
        var bBox = path.getBBox(); // width/height of icon
        var xTranslation = (bBox.width + 4) * i;
```

```
            var yTranslation = (bBox.height + 4) * j;
            path.transform(['T', xTranslation, yTranslation]);

            people.push(path);
        }
    }
```

In the given code, we iterate over `i` columns and `j` rows. We draw the icon using the `path` method and then get its bounding box in order to ascertain the width and height of the space it occupies. We then translate the icon by this width and height plus a small padding of 4px for each iteration. The number of widths that we wish to translate the icon in x is determined by the column index `i` while the number of heights by which we translate the icon in y is determined by the column index `j`:

```
var xTranslation = (bBox.width + 4) * i;
var yTranslation = (bBox.height + 4) * j;
```

Finally, we store our created people paths in a **set** named `people`. This will allow us to operate on all people based on which social media icon is clicked:

```
people.push(path);
```

Note that each icon is drawn at an opacity of 0.3, which will allow us to differentiate between users of a social network and non-users that make up the world's population.

Responding to icon clicks

We register a click event handler on each of the social media icons defined in our markup. When an icon is clicked, we will operate on the `people` set that we created in the previous section:

```
$('.networks img').click(function() {
    // Operate on people Set
});
```

On click, we end any existing animations on all of the paths stored in our `people` set and reset the opacity to 0.3. The `stop` method prevents an animation from continuing to its target attributes and is necessary to prevent future animations conflicting with ongoing ones:

```
people.stop().attr({opacity: 0.3});
```

We then get the class name of the element that was clicked and use it to get the number of users for this particular social network:

```
var className = $(this).attr('class');
var numPeople = Math.ceil(
    people.length * (usage[className] / 6000000000)
);
```

Since the class name maps to a usage value stored in our `usage` object, we calculate the usage as a proportion of the total population and then multiply this figure by the total number of people in our `people` set. This will give us a value between 0 and 120, that is, the number of icons that should be highlighted based on 50 million users per icon. The `ceil` method rounds this value up to the nearest one person.

Having determined the number of icons we need to highlight, we then iterate over them using the `forEach` method of the set object:

```
people.forEach(function(o) {
    if(o.id < numPeople) {
        o.animate({opacity: 1}, 500);
        return true;
    }
    return false;
});
```

The `forEach` method passes the object in the set to the function it accepts as the first parameter. This object has an `id` attribute corresponding to its indexed position in the set. It is a zero-based ID meaning that we can compare its value during looping to see if it is less than the `numPeople` variable that should be highlighted. If it is, we animate it to an opacity of 1 that differentiates between all other defined icons whose opacity is 0.3. The time taken to iterate over all paths in the `people` set is negligible meaning that all animations appear to take place at the same time. When the Facebook icon is clicked, for example, the first 20 icons in the `people` set are animated to an opacity of 1.

Drawing a key

Finally, to identify that one icon is equivalent to 50 million people, we redraw our person path and add some accompanying text:

```
var keyPerson = paper.path(personPath).attr({
    fill: '#666',
    'stroke-width': 0
});
var bBox = keyPerson.getBBox();
```

```
keyPerson.transform(['T', 0, bBox.height * 8]);
paper.text(
     bBox.width + 10,
     (bBox.height * 8) + 18,
     '= 50 million people'
).attr('text-anchor', 'start');
```

The position of this person path on our canvas is arbitrary. We place it at eight times the height of the paths bounding box so that it appears just below our six rows of people.

Tweets by time

There are approximately 3,000 tweets every second on average, meaning that over the space of 5 minutes there are about one million tweets created. In this section, we will create an interesting animated donut chart visualization that utilizes path arcs to illustrate the time taken for a particular number of tweets to be created:

Getting started

Our HTML markup defines a canvas into which we will draw four animated arc paths on page load. We will write our JavaScript in `tweets-by-time.js`:

```
<h1>Tweets with respect to time</h1>
<p>There are approximately 3,000 tweets every ...</p>
<div id="tweets-timed"></div>
<script type="text/javascript"
    src="raphael-2.1.0-min.js"></script>
<script type="text/javascript"
    src="jquery-1.9.1.min.js"></script>
<script type="text/javascript"
    src="tweets-by-time.js"></script>
```

We define a drawing canvas whose width is 770px and height is 450px to accommodate these timers:

```
var paper = Raphael('tweets-timed', 770, 450);
```

We also define two arrays to hold references to the paths that we will be drawing and the number of tweet counts with regard to time:

```
var paths = Array(), counts = Array();
```

Tweets by time data

Based on there being about 3,000 tweets per second, we define timers with the following values:

Time interval	Number of tweets
1 second	3,000
3.333 seconds	10,000
30 seconds	100,000
5 minutes	1,000,000

In our JavaScript file we create an array object with these values along with some associated text and the center-*x*, center-*y*, and radii of the arc paths:

```
var timers = [
    {
        cx: 225,
        cy: 100,
        r: 50,
```

```
            interval: 1000,
            total: 3000,
            text: '~ 3,000 tweets per second'
        },
       // ...
```

This code is truncated for the sake of brevity. For the full code you should download the source code accompanying this book.

The subtend custom attribute

The extent to which an arc is curved, or *subtended*, is dependent upon the time elapsed during an animation. Initially, the arc subtends a sector of zero length, gradually increasing in length until the angle subtended by the arc reaches 360 degrees.

To achieve this, we define a custom attribute named `subtend` that returns a path based on the angle of the arc at a particular time and its innate center-*x*, center-*y*, and radius attributes. The *x* and *y* coordinates of a curve's end point vary sinusoidally with regard to the angle subtended by it, meaning we can define a path that arcs from our start point (the center point in *x* and the center point in *y* minus the radius) as follows:

```
paper.customAttributes.subtend = function(angle, cx, cy, r) {
    var x = cx + r * Math.cos(Raphael.rad(90 - angle));
    var y = cy - r * Math.sin(Raphael.rad(90 - angle));
    var path = [
        'M', cx, cy - r,
        'A', r, r, 0, +(angle > 180), 1, x, y
    ];
    return {
        path: path
    };
};
```

Note the subpath definition `'A', r, r, 0, +(angle > 180), 1, x, y` describes an arc whose radius is fixed at `r` and whose large arc flag is 0 for when the subtended angle is less than or equal to 180 degrees (where we want to draw the smaller of the available arcs) and 1 when the angle is greater than 180 degrees (when we want to draw the larger arc).

The counts custom attribute

Our visualization will display the number of tweets as text based on the proportion (expressed as a factor between 0 and 1) of the animation that has passed and the total tweets for that timer. We define the `counts` custom attribute to return this ratio of time elapsed to total tweets as text as follows:

```
paper.customAttributes.counts = function(ratio, total) {
    return {
        text: Math.round(ratio * total)
    }
};
```

Updating the timer

When the animation of a timer initially begins, or is reset after traversing a full 360 degrees (the length of one animation), we run an update method that is responsible for setting the "start" values of our arc. The arc starts at 0 degrees and is given a 20px stroke that gives it the arcing clock effect:

```
function update(i) {
    // Get properties from our timers data array
    var cx = timers[i].cx, cy = timers[i].cy,
        radius = timers[i].r, interval = timers[i].interval;

    // Draw a new path for this particular timer
    paths[i] = paper.path().attr({
        subtend: [0, cx, cy, radius],
        'stroke-width': 20,
        stroke: '#009ec4'
    });

    animate(i, cx, cy, radius, timers[i].interval);
}
```

The helper method `animate` is called at the end of this function in order to initialize our animation from 0 degrees to 360 degrees. Note that our paths are stored in the paths array so that we can refer to them by their index later.

The animate helper method

The purpose of the `animate` function is to animate:

- The arc from its initial arc path to the final arc path
- The number of tweets counted over the time interval elapsed

In the code given here, we animate our arc to subtend 360 degrees and our text counts as a proportion of the total number of tweets:

```
function animate(i, cx, cy, r, interval) {
    // Animate the path
    paths[i].animate({
        subtend: [360, cx, cy, r]
    }, interval, function() {
        paths[i].remove();
        setTimeout(function() {
            update(i);
        });
    });

    // Animate the counts
    counts[i].animate({
        counts: [1, timers[i].total]
    }, interval, function() {
        counts[i].attr({
            'counts': [0, timers[i].total]
        });
    });
}
```

The `animate` method is run over the duration specified in our `timers` array based on the index of that timer passed as the first parameter. Note that when the animation of the arc is complete, we run the `update` function again to restart the whole process. Likewise, the `counts` text is reset to 0.

Iterating over our timers and starting the animation

To initialize the animation, we iterate over our timers, defining our `counts` text with an initial value of `0` positioned at the center-*x* and center-*y* point of our arc. We then run our `update` function on each timer, which has the effect of animating the arcs in a recursive manner as we have discussed.

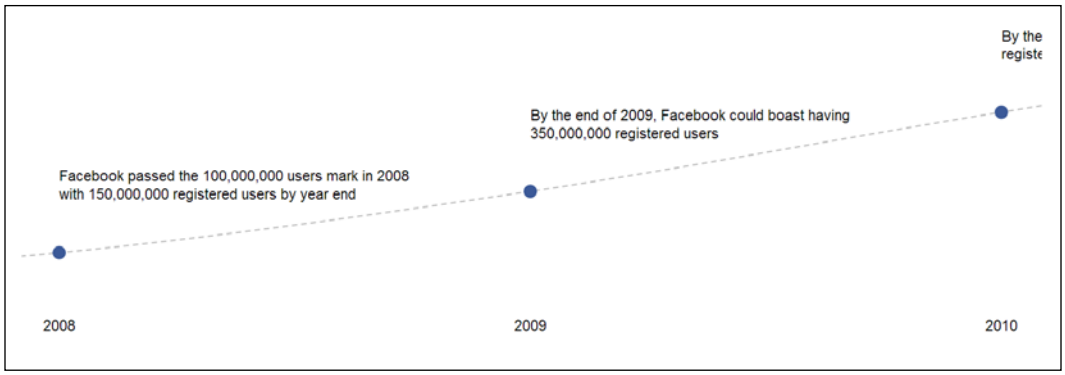
```
for(var i = 0, ii = timers.length; i < ii; i+=1) {
    (function(i) {
        // Draw descriptive text beneath the timer
        paper.text(timers[i].cx, timers[i].cy + timers[i].r + 30,
timers[i].text);
        // Define counts text
        counts[i] = paper.text(
            timers[i].cx, timers[i].cy, '0'
        ).attr({
            'counts': [0, timers[i].total]
        });

        update(i);
    })(i);
}
```

Supplementary material

Two additional data visualization demos have been written up at `http://raphaeljsvectorgraphics.com` and are available to you as supplementary material.

Facebook usage by year

A simple timeline visualization has been created to demonstrate the rise in the number of users by year. The timeline is a Catmull-Rom curve that can be panned in the *x*-direction as shown in the following screenshot:

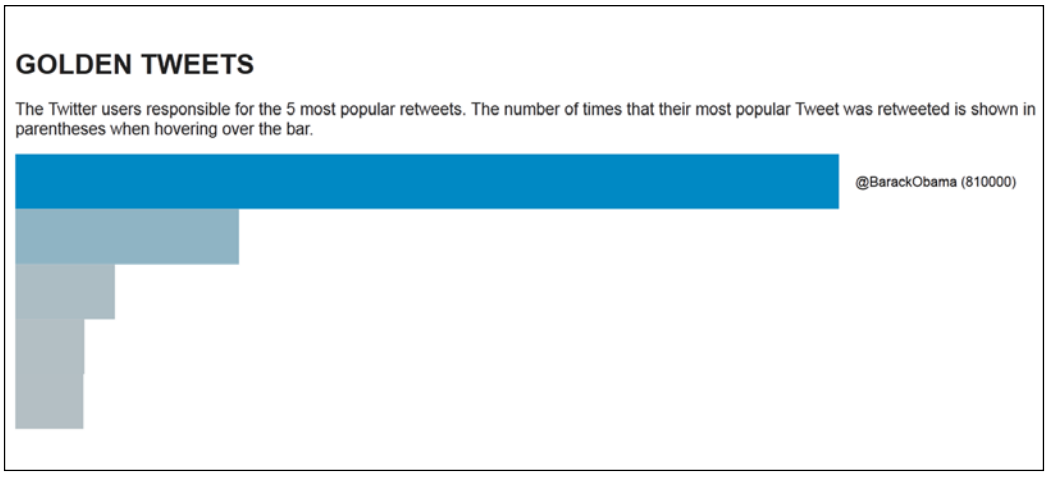


The supplementary material for this visualization is available at `http://raphaeljsvectorgraphics.com/book/supplementary/facebook-usage-by-year`.

Golden tweets

Twitter published an article at the end of 2012 discussing some of the tweets that were re-tweeted the highest number of times. These have been marketed as *golden tweets* given the high levels of participation that they encouraged.

A bar chart visualization has been created to represent the number of times that these tweets were re-tweeted:



The supplementary material for this visualization is available at `http://raphaeljsvectorgraphics.com/book/supplementary/golden-tweets`.

The future of Raphaël

The current maintainer of Raphaël, Tomás Alabés, kindly contacted me with information pertaining to the future of the library. Going forward, we expect increased adoption of the library and changes are being put in place to enable more open source collaboration of Raphaël.

Milestones

The Raphaël project currently has four milestones as shown in the following table:

Version	Milestone
2.1.1	Addressed bug fixes contributed by the community
2.2.0	Brought in enhancements from contributors' pull requests
2.2.1	Addressed bug fixes from issues created by the community
2.3.0	Developed enhancements suggested in issues created by the community

As you can see, the forthcoming versions of Raphaël will address a number of existing bug fixes but perhaps more interestingly, start to implement features that have either been contributed or suggested by the community. This makes for a solid library that continues to evolve.

Long term goals

Tomás is especially eager to encourage open source collaboration on Raphaël. He has already begun making commits to the source (`https://github.com/DmitryBaranovskiy/raphael/`) that will help facilitate wider collaboration.

Additionally, when VML-compatible browsers start to lose market share and supporting them is no longer a requirement, the library will begin to focus exclusively on SVG which will bring with it a number of features and improvements that are inhibited by VML.

Summary

Having come to the end of this book, you will hopefully have a firm grip on the capabilities of the Raphaël library and have gained an understanding of some of the ways in which it can be used and the approaches taken to drawing vector graphics and visualizing data. You are reminded to explore the library's documentation and experiment with all of its available features and invest your time in researching the concepts driving your future applications, be they in data visualization or game development or cartography. The true power of any library is at the behest of the one utilizing it.

It is hoped that you have enjoyed reading this book as much as I have enjoyed writing it.

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

O

P

Q

R

S

T

U

V

W

X

Thank you for buying

Learning Raphaël JS Vector Graphics

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

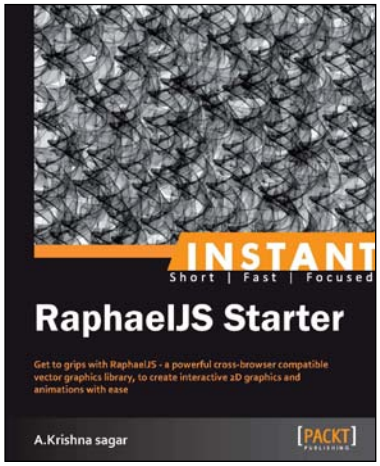
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

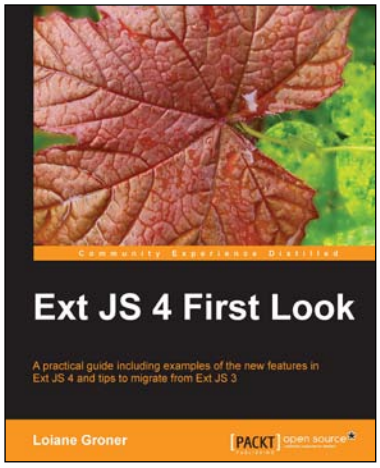


Instant RaphaelJS Starter

ISBN: 978-1-78216-985-7 Paperback: 62 pages

Get to grips with RaphaelJS - a powerful cross-browser compatible vector graphics library, to create interactive 2D graphics and animations with ease

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Create cross-browser imageless drawings and animations, with DOM access
3. Create your own shape, almost any shape, with simple and illustrated techniques
4. Write once and run in almost any browsers including IE6



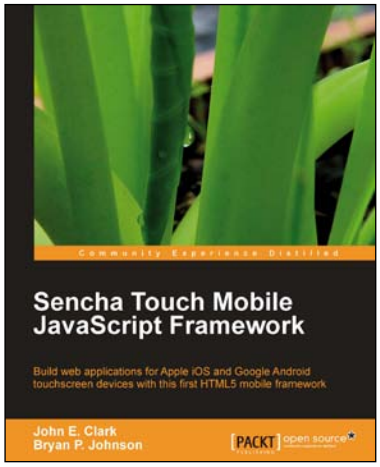
Ext JS 4 First Look

ISBN: 978-1-84951-666-2 Paperback: 340 pages

A practical guide including examples of the new features in Ext JS 4 and tips to migrate from Ext JS 3

1. Migrate your Ext JS 3 applications easily to Ext JS 4 based on the examples presented in this guide
2. Full of diagrams, illustrations, and step-by-step instructions to develop real word applications
3. Driven by examples and explanations of how things work

[PACKT] open source
community experience distilled
PUBLISHING



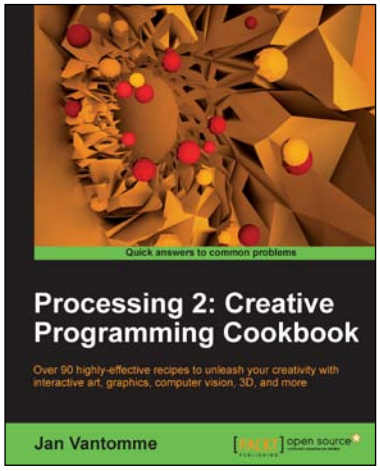
Sencha Touch Mobile JavaScript Framework

ISBN: 978-1-84951-510-8 Paperback: 316 pages

Build web applications for Apple iOS and Google Android touchscreen devices with this first HTML5 mobile framework

1. Learn to develop web applications that look and feel native on Apple iOS and Google Android touchscreen devices using Sencha Touch through examples
2. Design resolution-independent and graphical representations like buttons, icons, and tabs of unparalleled flexibility
3. Add custom events like tap, double tap, swipe, tap and hold, pinch, and rotate

---



Processing 2: Creative Programming Cookbook

ISBN: 978-1-84951-794-2 Paperback: 306 pages

Over 90 highly-effective recipes to unleash your creativity with interactive art, graphics, computer vision, 3D, and more

1. Explore the Processing language with a broad range of practical recipes for computational art and graphics
2. Wide coverage of topics including interactive art, computer vision, visualization, drawing in 3D, and much more with Processing
4. Create interactive art installations and learn to export your artwork for print, screen, Internet, and mobile devices