O'REILLY®

SVG Colors, Patterns & Gradients

PAINTING VECTOR GRAPHICS

Amelia Bellamy-Royds

SVG Colors, Gradients, & Patterns

Painting Vector Graphics

Amelia Bellamy-Royds
and Kurt Cagle

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

SVG Colors, Gradients, & Patterns
by Author Name

If your book has an ISBN or a book ID number, add it here as well.

Table of Contents

Preface

This book takes a deep dive into a specific aspect of SVG, painting. Painting not with oils or watercolor, but with graphical instructions that a computer can transform into colored pixels. The book explores the creative possibilities, and also the potential pitfalls. It describes the basics, but also suggests how you can mix and match the tools at your disposal to generate complex effects.

This book was born from another project, an introduction to using SVG on the web. In order to keep that book a manageable length—and keep it suitable for introductory audiences—many details and complexities had to be skimmed over. But those details and complexities add up to the full, wonderful potential of SVG as a graphics format. Once you understand the basics of SVG, you can start thinking about creating more intricate drawings and more nuanced effects.

What We'll Cover

If you're reading this, hopefully you're already familiar with the basics of SVG—how to define a graphic as a set of shapes, and how to use that graphic either as a stand-alone image file or as markup in an HTML page. If you're not sure if you're ready, Chapter 1 reviews the basic concepts we'll expect you to know.

The rest of the book focuses on the *Colors, Patterns, and Gradients* described in the title:

- Chapter 2 discusses the rendering model used to convert SVG code into visual graphics, and introduces the basic properties

you can set on your shapes and text to control how they are painted to the screen.

- Chapter 3 discusses color: how it works in nature, how it works on the computer, and the many different ways it can be specified within your SVG code.
- Chapter 4 discusses transparency, or more specifically, opacity; it introduces the many ways you can control the opacity of your graphics, and how these affect the end result.
- Chapter 5 introduces the concept of a paint server: complex graphics content that defines how other SVG shapes and text should be painted to the screen. It also introduces the solid color paint server, which is actually more useful than it first sounds.
- Chapter 6 looks at gradients with a particular focus on the different color transition effects you can achieve by adjusting color stop positions and properties.
- Chapter 7 explores the ways in which you can manipulate a linear gradient to move it within the shape being painted.
- Chapter 8 covers repeating linear gradients and some of the effects you can create with them.
- Chapter 9 looks at radial gradients, including repeated radial gradients, and concludes with some examples of creating complex effects with multiple gradients.
- Chapter 11 introduces the `<pattern>` element, using it to define a single block of graphics that can be used to fill or stroke shapes or text.
- Chapter 10 explores the more conventional usage of patterns, to create repeating tiles and textures.
- Chapter 13 looks at some of the issues that come into play when using paint servers to paint strokes instead of fill regions.
- Chapter 14 gives some examples of animated paint servers and discusses the benefits and limitations of the different animation methods available in SVG.

At the end of the book, two appendices provide a quick reference for the basic syntax you'll need to put this all to use:

- Appendix A recaps all the ways you can define colors, including all the pre-defined color keywords.
- Appendix B summarizes all the paint server elements, their attributes, and the related style properties.

About This Book

Whether you're casually flipping through the book, or reading it meticulously cover-to-cover, you can get more from it by understanding the following little extras used to provide additional information.

Conventions Used in This Book

(O'Reilly boilerplate on code & term formatting)

Tips like this will be used to highlight particularly tricky aspects of SVG, or simple shortcuts that might not be obvious at first glance.

Notes like this will be used for more general asides and interesting background information.

Warnings like this will highlight combatibility problems between different web browsers (or other software), or between SVG as an XML file versus SVG in HTML pages.

About the Examples

(Where to download sample files or view online, compatibility info)

(O'Reilly boilerplate on copyright & permissions)

How to Contact Us

(O'Reilly boilerplate)

Acknowledgements

(Thank you, thank you very much)

CHAPTER 1
Things You Should Already Know

The rest of this book is written with the assumption that you already know something about SVG, web design in general, and maybe even a little JavaScript programming.

However, there are always little quirks of a language that some people think are straightforward and other, equally talented, developers have never heard of. So this chapter gives a quick review of topics that you might want to brush up on—if you don't already know them.

SVG is Drawing with Code

An SVG is an image file. It is perfectly possible to only use it as an image file, the same way you would use other image formats such as PNG or JPEG. You can create and edit an SVG in a visual editor. You can embed it in web pages as an image.

But SVG is more than an image. It is a structured document containing markup elements, text, and style instructions. While other image formats tell the computer which color to draw at which point on the screen, SVG tells the computer how to rebuild the graphic from its component parts. That has two main consequences:

- The final appearance of an SVG depends on how well the computer displaying it follows the SVG instructions. Cross-browser compatibility is often a concern.
- It is easy to edit an SVG, to add, remove, or modify parts of it without changing the rest. You can do this in your editor,

but you can also do it dynamically in your web browser to create animated or interactive graphics.

SVG is Always Open Source
: Not only is an SVG a set of coded instructions for a computer, it is also a human-readable text file. You can edit your SVG in a text editor. Even better, you can edit SVG in a code editor with syntax highlighting and autocomplete!

  The examples in this book all focus on the basic SVG code. You can, of course, use a visual editor to draw shapes, select colors, and otherwise fuss with the appearance of your graphic. But for full control, you will need to take a look at the actual code that editor creates.

SVG is XML (and sometimes HTML)
: The SVG code you view in your text editor looks an awful lot like HTML code—full of angle brackets and attributes—but a stand-alone SVG file is parsed as an XML document. This means that your SVG can be parsed and manipulated by tools meant for XML in general. It also means that your web browser won't display anything if you forget to include the XML namespaces or mixed up an important detail of XML syntax.

  Nonetheless, when you insert SVG code directly in HTML 5 markup, it is processed by the HTML parser. The HTML parser forgives errors (like missing closing tags or unquoted attributes) that would halt the XML parser. But it also ignores any custom namespaces, downcases any unrecognized attribute or tag names, and otherwise changes things up in ways you might not expect.

SVG is Squishable
: The syntax for SVG was designed to make it easy to read and understand, not to make it compact. This can make certain SVG files seem rather verbose and redundant. However, it also makes SVG very suitable for gzip compression, which should always be used when serving SVG on the web. It will usually reduce file sizes by more than half, sometimes much more. If storing a gzipped SVG on a regular file server, it is typical to use the *.svgz* extension.

SVG is also bloatable, which makes it squishable in another way. SVG editors can add their own elements and attributes to an SVG file by giving them unique XML namespaces. A class of SVG optimizing tools has developed that will strip out code that does not affect the final result. Just be careful about the settings you use—optimizers can remove attributes you might want later if you're manipulating the code yourself!

Pictures are a Collection of Shapes
: So what does all that code represent? Shapes, of course! (And text and embedded images, but we'll get to that in a moment…) SVG has only a few different shape elements: `<rect>`, `<circle>`, `<ellipse>`, `<line>`, `<polyline>`, `<polygon>`, and `<path>`. However, those last three can be extensively customized to represent any shape you can imagine, to a certain degree of precision. The `<path>` in particular contains its own coded language for describing the curves and lines that create that shape.

Images Can Have Images Inside Them
: An SVG is an image, but it is also a document, and that document can contain other images, using the `<image>` element. The embedded images could be other SVG files, or they could be raster images such as PNG or JPEG. However, for security and performance reasons, some uses of SVG prevent those external images (and other external resources such as stylesheets or fonts) from being downloaded.

Text is Art
: The one other building block in SVG is text. But text isn't an alternative to graphics—the letters that make up that text are treated like another type of vector shape. Importantly for this book, text can be painted using the exact same style properties as vector shapes.

Art is Math
: The core of all vector graphics (shapes or text) is that the end result can be defined using mathematical parameters (the XML attributes) to the browser's SVG rendering functions for each element. The most pervasive mathematical concept in SVG is the coordinate system, used to define the position of every point in the graphic. You can control the initial coordinate system by setting a `viewBox` attribute, and you can use coordinate system

transformations to shift, stretch, rotate and skew the grid for certain elements.

An SVG is a Limited View of an Infinite Canvas
: There are no limits on the coordinates you can give for your vector shapes (except for the practical limits of computer number precision). The only shapes displayed, however, are those that fit within the particular range of coordinates established by the `viewBox` attribute. This range of coordinates is scaled to fit the available area (the "viewport"), with accomodations for mismatched aspect ratios controlled by the `preserveAspectRatio` value. You can create nested viewports with nested `<svg>` elements or re-used `<symbol>` elements; in addition to providing regions of aspect ratio control, these re-define how percentage lengths are interpretted for child content.

  The distinction between `viewBox` and "viewport" is one of the more confusing aspects of the SVG specs. As we'll see when we get to Chapter 11, the `<pattern>` element (and also the `<marker>` element) can have a `viewBox` attribute, but does not establish a viewport for the purposes of percentage lengths.

SVG has Structure
: The structure of an SVG includes the basic shapes, text, and images that are drawn to the screen, and the attributes that define their geometry. But SVG can have more structure than that, with elements grouped into logical clusters. Those groups can be styled and their coordinate systems transformed. But they can also be given accessible names and descriptions to help explain exactly what the graphics represent.

SVG has Style
: SVG graphics can consist solely of XML, with all style information indicated by presentation attributes. However, these presentation styles can also be specified with CSS rules, allowing styles to be grouped by class or element type. Using CSS also allows conditional styles to depend on media features or transient states such as `:hover` or `:focus`.

  The strict separation between geometric structure (XML attributes) and presentation style (presentation attributes or CSS style rules) has always been a little arbitrary. As SVG moves forward, expect it to collapse even more. The SVG 2 draft specifi-

cations promote many layout attributes to become presentation attributes. This opens these properties to all the syntactic flexibility CSS offers: classes of similar elements can be given matching sizes with a single style rule, and those sizes or layout can be modified with CSS pseudoclasses or media queries.

Behind All Good Markup is a Great DOM

The SVG markup and styles are translated into a document object model (DOM) within a web browser. This DOM can then be manipulated using JavaScript. All the core DOM methods defined for all XML content apply, so you can create and reorder elements, get and set attributes, and query the computed style values.

The SVG specifications define many unique properties and methods for SVG DOM elements. These make it easier to manipulate the geometry of a graphic mathematically. Support for SVG DOM in web browsers is not as good as one might hope, but certain methods—such as determining the length of a curved path—are indispensable for SVG designs.

SVG Can Move

In a dynamic SVG viewer (e.g., web browser) with scripting support, you can use those scripts to create animated and interactive graphics. However, SVG also supports declarative means of interaction, whereby you define the scope of an entire interaction and the browser applies it with its own optimizations. There are two means of doing this:

- using animation elements in the markup, with a syntax borrowed from the Synchronized Multimedia Integration Language (SMIL)
- using CSS animations and transitions of presentation styles

At the time of writing, scripted animation is supported in all web browsers, but may be blocked for certain uses of SVG. Declarative animation (SMIL and CSS) is supported in most browsers, but not all (particularly, not Internet Explorer).

SVG Can Change

Not only can individual SVG graphics change as you use them, but the definition of SVG can change too. The established standard (at the time this book was written) is SVG 1.1, but work is ongoing to develop a level 2 SVG specification with new fea-

tures and clearer definitions of some existing features. Furthermore, because SVG uses CSS and JavaScript, and because it is heavily integrated in HTML, it inherits changes to those languages, as well.

CHAPTER 2
The Painter's Model

If I asked you to draw a yellow circle with a blue outline, would it look the same as if I asked you to draw a blue circle and fill it in with yellow?

If I asked you to draw a red pentagon and a green square centered on the same spot on a page, would most of the image be red or green?

There are no hard and fast rules when you're drawing things by hand. If someone gives you ambiguous instructions, you can always ask for clarification. But when you're giving instructions to a computer, it only has one way to follow them. So you need to make sure you're saying exactly what you mean.

Even if you use SVG a lot (and we're going to assume you use it at least a little), you probably haven't given much thought to how the computer converts your SVG code into colored patterns on the screen. If you're going to really make the most of those colored patterns, however, you need to know how your instructions will be interpretted.

This chapter discusses the basics of the SVG *rendering model*, the process by which the computer generates a drawing from SVG markup and styles. It reviews the basic `fill` and `stroke` properties that define how you want shapes or text to be painted. The entire rest of the book can really summed up as different ways you can specify fill or stroke values.

The SVG rendering model is known as a *painter's model*. Like layers of paint on a wall, content on top obscures content below. The SVG specifications define which content gets put on top of which other content. However, this chapter also introduces two properties, `z-index` and `paint-order`, which allow you to change up the rendering rules. These properties are newly introduced in SVG 2, and are only just starting to be supported in web browsers. We therefore also show how you can achieve the same effect with SVG 1.1 code.

Fill 'er Up with the `fill` Property

The basic elements and attributes in you SVG code define precise geometric shapes. For example, a one-inch square, positioned with its top left corner at the coordinate system origin, looks like this:

```
<rect width="1in" height="1in" />
```

A circle ten centimeters in diameter, centered on the middle of the coordinate system, is created with code like the following:

```
<circle cx="50%" cy="50%" r="5cm" />
```

If you used either of those elements in an SVG, without any style information, it would be displayed as a solid black region exactly matching the dimensions you specify. This is the default `fill` value: solid black.

The `fill` property tells the SVG-rendering software what to do with that geometric shape. For every pixel on the screen—or ink spot on the paper—the software determines if that point is inside or outside of the shape. If it is inside, the software turns to the `fill` value to find out what to do next.

In the simple case (like the default black), the fill value is a color and all the points inside the shape get replaced by that color. In other cases, the fill value is a direction to look up more complicated painting code. Where to look it up is indicated by a URL referencing the `id` of an SVG element representing the instructions (a *paint server*, which we'll talk more about starting in Chapter 5)

If you *don't* want the software to fill in the shape, all you have to do is say so. The `fill` property also takes a value of `none`.

The fill is by default painted solid and opaque (unless there are different instructions in the paint server). The `fill-opacity` property can adjust this. It takes a decimal number as a value: values between 0 and 1 cause the shape's paint to be blended with the colors of the background. A value of 1 (the default) is opaque, while a value of 0 has much the same effect as `fill: none`. We'll discuss opacity in detail in Chapter 4.

When it is not clear which sections of the shape are inside versus outside, the `fill-rule` property gives the computer exact instruction. It affects `<path>` elements with donut holes inside them, as well as paths, polygons, and polylines with criss-crossing edges.

The `fill-rule` property has two options:

- `evenodd` switches between inside and outside every time you cross an edge.
- `nonzero` (the default) gets "more inside" when you cross an edge that is drawn in the same direction as the last one, and only gets back to outside again when you have cancelled them all out by crossing edges in the opposite direction.

Example 2-1 draws a criss-crossed `<polygon>`, first with the default `nonzero` fill rule and then with an `evenodd` fill rule; Figure 2-1 shows the result. The shapes have a thin dark stroke around the edges so you can see them even when the shape is filled on both sides of the edge.

Example 2-1. Modifying the fill region with the `fill-rule` property

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     viewBox="0 0 400 200" width="4.3in" height="2.15in">
    <title>Fill-rule comparison</title>
    <rect fill="lightSkyBlue" height="100%" width="100%" />          ❶

    <polygon id="p"
             fill="blueViolet" stroke="navy"
             points="20,180 20,20 180,20 180,180 60,60 140,60" /> ❷
    <use xlink:href="#p" x="50%" fill-rule="evenodd" />              ❸
</svg>
```

❶ The opening `<svg>` element establishes the coordinate system and sets the default size of the printed figure. A `<rect>` element

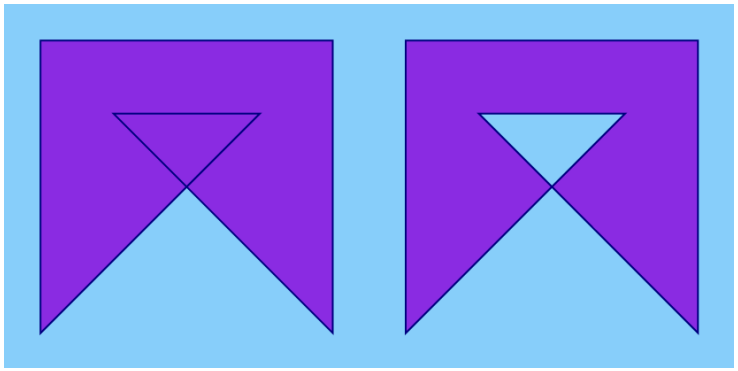

Figure 2-1. A polygon with `nonzero` fill rule (left) and with `evenodd` fill rule (right)

adds a solid-color backdrop. For this simple SVG code, styles are set with presentation attributes.

❷ The basic polygon has `fill` and `stroke` styles, but the `fill-rule` property will inherit the default `nonzero` value.

❸ A duplicated copy of the same polygon is offset horizontally by half the width of the SVG. The copied polygon will inherit the `fill-rule="evenodd"` value set on the `<use>` element.

No matter how many times the edges or sub-paths cross over each other, each point is either inside or outside the shape. Areas are not painted twice just because they are inside two different sub-paths. That may not seem like a relevant distinction when the fill is a solid color, but it becomes important when fill is partially transparent.

Future Focus
Filling in the Future

The discussion of the `fill` property in this section has focused on the way it is currently defined in the stable SVG 1.1 specifications. The in-progress SVG 2 specifications will offer more flexibility to the way shapes are filled, most notably by allowing a single shape to have multiple fill lay-

ers. These proposed features will be discussed in more detail elsewhere in the book, in future-focused asides such as this.

Every shape in SVG, as well as text, can be filled—and will be filled by default. This includes open-ended `<path>` elements and `<polyline>` elements, which define a shape where the end of the line does not connect with the beginning. The fill region of these shapes is created by connecting the final point back to the beginning in a straight line. If it ends up crossing other edges as it does so, the `fill-rule` calculations apply.

Open segments within a `<path>` are closed by connecting them back to the initial point on that sub-path: the last point created by a move-to command.

Even a straight `<line>` element is technically filled in by default: however, because the return line that connects the end point to the beginning exactly overlaps the original line, the resulting shape does not enclose any area. No points are inside the shape, and so no points are affected by the fill value. You need to *stroke* the line if you want to see it.

Stroke It with the `stroke` Property

In computer graphics, stroking a shape means drawing a line along its edge. Different programs have different interpretations of what that can mean.

In SVG (currently, anyway), stroking is implemented by generating a secondary shape extending outwards and inwards from the edges of the main shape. That stroke region is then painted using the same approach as for filling the main shape: the software scans across, and determines whether a point is inside or outside the stroke. If the point is inside, the software uses the painting instructions from the `stroke` property to assign a color.

Each section of the stroke-shape is only painted once, regardless of whether the strokes from different edges of the shape overlap or cross each other.

The default for `stroke` is `none`, meaning don't paint a stroke region at all. Just like for `fill`, the other options are a color value or a reference to a paint server element.

Just as with fill, there is a `stroke-opacity` property to modify the stroke paint. Just as with `fill-opacity`, we'll discuss `stroke-opacity` in more detail in Chapter 4.

There are many other stroke-related properties. We're not going to talk about them much in this book, but they control the geometry of the stroke region. As a quick reference, they are as follows:

`stroke-width`
: The thickness of the stroke, as a length, number of user units, or percentage of the weighted width and height of the coordinate system. In SVG 1.1, the stroke region is always centered on the edge of the shape, so half the stroke width extends outside it.

`stroke-linecap`
: The approach to use for stroking around open ends of a path or line; the default `butt` trims the stroke tight and perpendicular to the endpoint. The other options, `round` and `square`, extend the stroke by half the stroke width, in the specified shape.

`stroke-linejoin`
: The approach to use for stroking around corners in the shape; the default `miter` extends the strokes in straight lines until they meet in a point. The other options are `round` (use a circular arc to connect the two strokes) and `bevel` (connect the two strokes with an additional straight line).

`stroke-miterlimit`
: The maximum distance to extend a mitered line join beyond the official edge of the shape, as a multiple of the stroke width (default 4 times the width). If the stroke edges don't meet in a point within that distance, a bevelled line join is used instead.