1
Drawing Shapes in Canvas

In this chapter we will cover:

- Graphics with 2D canvas
- Starting from basic shapes
- Layering rectangles to create the flag of Greece
- Creating shapes using paths
- Creating complex shapes
- Adding more vertices
- Overlapping shapes to create other shapes

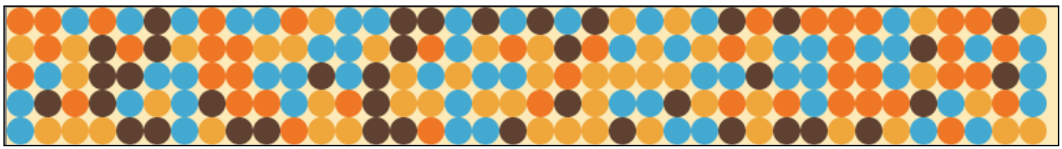
Introduction

This chapter's main focus is to make a breakthrough into working in canvas. We will spend the majority of our time working with canvas when creating our charts.

In this chapter, we will master the basic shapes and styles of drawing with the canvas API. This chapter will be the graphic's backbone to the rest of the book, so if at any stage you feel you need a refresher you could come back to this chapter. Drawing lines can be... well not very thrilling. What better way to make it more dramatic than to integrate a theme into this chapter as a subtopic: creating flags!

Graphics with 2D canvas

Canvas is the primary and most thrilling addition to HTML. It's the buzz of the industry, so let's start there. We will revisit canvas again in the later chapters. In this recipe, we will learn how to draw dynamically using canvas, and create an array of colorful circles that will update once every second.



How to do it...

We will be creating two files (an HTML5 file and a JS file). Let's start by creating a new HTML5 document:

1. The first step is to create an empty HTML5 document:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Canvas Example</title>
  </head>
  <body>
  </body>
</html>
```

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 code files are also available at `http://02geek.com/books/html5-graphics-and-data-visualization-cookbook.html`.

2. Create a new canvas element. We give our canvas element an ID of `myCanvas`:

```
  <body>
<canvas id="myCanvas"> </canvas>
  </body>
```

3. Import the JavaScript file `01.01.canvas.js` into the HTML document (we will create this file in step 5):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
 <script src="01.01.canvas.js"></script>
    <title>Canvas Example</title>
  </head>
```

4. Add an `onLoad` listener and trigger the function `init` when the document loads:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <script src="01.01.canvas.js"></script>
    <title>Canvas Example</title>
  </head>
  <body onLoad="init();" style="margin:0px">
    <canvas id="myCanvas" />
  </body>
</html>
```

5. Create the `01.01.canvas.js` file.
6. In the JavaScript file, create the function `init` and call the function `updateCanvas` within it:

```
function init(){
  updateCanvas();
}
```

7. Create the function `updateCanvas`:

```
function  updateCanvas(){
  //rest of the code in the next steps will go in here
}
```

8. In the `updateCanvas` function (for the rest of the steps all the code will be added in this function) create two variables that will store your desired width and height. In our case we will grab the width of our window:

```
function  updateCanvas(){
  var width = window.innerWidth;
  var height = 100;
...
```

9. Access the canvas layer in the HTML document and change its width and height:

```
var myCanvas = document.getElementById("myCanvas");
    myCanvas.width = width;
    myCanvas.height = height;
```

10. Get the 2D context of the canvas:

```
var context = myCanvas.getContext("2d");
```

11. Create a rectangle to fill the full visible area of the canvas:

```
context.fillStyle = "#FCEAB8";
context.fillRect(0,0,width,height);
```

12. Let's create a few helper variables to help us establish the color, size, and count of elements to be drawn:

```
var circleSize=10;
var gaps= circleSize+10;
var widthCount = parseInt(width/gaps);
var heightCount = parseInt(height/gaps);
var aColors=["#43A9D1","#EFA63B","#EF7625","#5E4130"];
var aColorsLength = aColors.length;
```

13. Create a nested loop and create a grid of circles in random colors:

```
for(var x=0; x<widthCount;x++){
  for(var y=0; y<heightCount;y++){
    context.fillStyle = aColors[parseInt
      (Math.random()*aColorsLength)];
    context.beginPath();
    context.arc(circleSize+gaps*x,circleSize+ gaps*y,
      circleSize, 0, Math.PI*2, true);
    context.closePath();
    context.fill();
  }
}
}
```

Woah! That was a lot of steps! If you followed all the steps, you will find a lot of circles in your browser when you run the application.

How it works...

Before we jump right into the JavaScript portion of this application, we need to trigger the `onLoad` event to call our `init` function. We do that by adding the `onLoad` property into our HTML body tag:

```
<body onLoad="init();">
```

Let's break down the JavaScript portion and understand the reason behind doing this. The first step is to create the `init` function:

```
function init(){
  updateCanvas();
}
```

Our `init` function immediately calls the `updateCanvas` function. This is done so that later we can refresh and call `updateCanvas` again.

In the `updateCanvas` function, we start by getting the current width of the browser and set a hardcoded value for the height of our drawing area:

```
var width = window.innerWidth;
var height = 100;
```

Our next step is to get our canvas using its ID, and then set its new width and height based on the previous variables:

```
var myCanvas = document.getElementById("myCanvas");
    myCanvas.width = width;
    myCanvas.height = height;
```

It's time for us to start drawing. To do that, we need to ask our canvas to return its context. There are a few types of contexts such as 2D and 3D. In our case we will focus on the 2D context as follows:

```
var context = myCanvas.getContext("2d");
```

Now that we have the context, we have all that we need to start exploring and manipulating our canvas. In the next few steps, we define the canvas background color by setting the `fillStyle` color using a hex value and by drawing a rectangle that would fit within the entire area of our canvas:

```
var context = myCanvas.getContext("2d");
    context.fillStyle = "#FCEAB8";
    context.fillRect(0,0,width,height);
```

The `fillRect` method takes four parameters. The first two are the (x,y) locations of the rectangle, in our case we wanted to start from (0,0), and the following parameters are the width and height of our new rectangle.

Let's draw our circles. To do so we will need to define the radius of our circle and the space between circles. Let's not space out the circles at all, and create circles with a radius of 10 px.

```
var rad=10;
var gaps= rad*2;
```

The first line assigns the radius for our circles, while the second line captures the gap between the centres of each circle we create, or in our case the diameter of our circle. By setting it up as two times the radius we space out our circles exactly one after the other.

```
var widthCount = parseInt(width/gaps);
var heightCount = parseInt(height/gaps);
var aColors=["#43A9D1","#EFA63B","#EF7625","#5E4130"];
var aColorsLength = aColors.length;
```

Using our new `gaps` variable, we discover how many circles we can create in the width and height of our canvas component. We create an array that stores a few color options for our circles and set a variable `aColorsLength` as the length of `aColors`. We do this to cut down the processing time, as variables are easier to fetch than properties as we are about to call this element many times in our `for` loop:

```
for(var x=0; x<widthCount;x++){
  for(var y=0; y<heightCount;y++){
    context.fillStyle = aColors[parseInt
      (Math.random()*aColorsLength)];
    context.beginPath();
    context.arc(rad+gaps*x,rad+ gaps*y, rad, 0, Math.PI*2, true);
    context.closePath();
    context.fill();
  }
}
```

Our nested `for` loops enable us to create our circles to the width and height of our canvas. The first `for` loop focuses on upgrading the width value while the second `for` loop is in charge of running through every column.

```
context.fillStyle =
aColors[parseInt(Math.random()*aColorsLength)];
```

Using `Math.random`, we randomly select a color from `aColors` to be used as the color of our new circle.

```
context.beginPath();
context.arc(rad+gaps*x,rad+ gaps*y, rad, 0, Math.PI*2, true);
context.closePath();
```

The first and last lines in the previous block of code declare the creation of a new shape. The `beginPath` method defines the start of the shape and the `closePath` method defines the end of it, while `context.arc` creates the actual circle. The `arc` property takes the following format of values:

```
context.arc(x,y,radius,startPoint,endPoint, isCounterClock);
```

The `x` and `y` properties define the center point of the arc (in our case a full circle). In our `for` loops we need to add a buffer of an extra radius to push our content into the screen. We need to do this as only one fourth of our first circle would be visible if we didn't push it to the left and to the bottom by an extra radius.

```
context.fill();
```

Last but not least, we need to call the `fill()` method to fill our newly-created shape with its color.

There's more...

Let's make our element refresh once a second; to do that all we need to do is add two more lines. The first one will trigger a new call to the `updateCanvas` function once every second using `setInterval`.

```
function init(){
  setInterval(updateCanvas,1000);
  updateCanvas();
}
```

If you refresh your browser you will find that our sample is working. If you try really hard to find issues with it you will not, but we have a problem with it. It's not a major problem but a great opportunity for us to learn about another useful functionality of the canvas. At any stage we can clear the canvas or parts of it. Instead of drawing on top of the current canvas, let's clear it before we redraw. In the `updateCanvas` function, we will add the following highlighted code:

```
var context = myCanvas.getContext("2d");
context.clearRect(0,0,width,height);
```

As soon as we get the context we can clear the data that was already present by using the `clearRect` method.

See also

- The *Starting from basic shapes* recipe

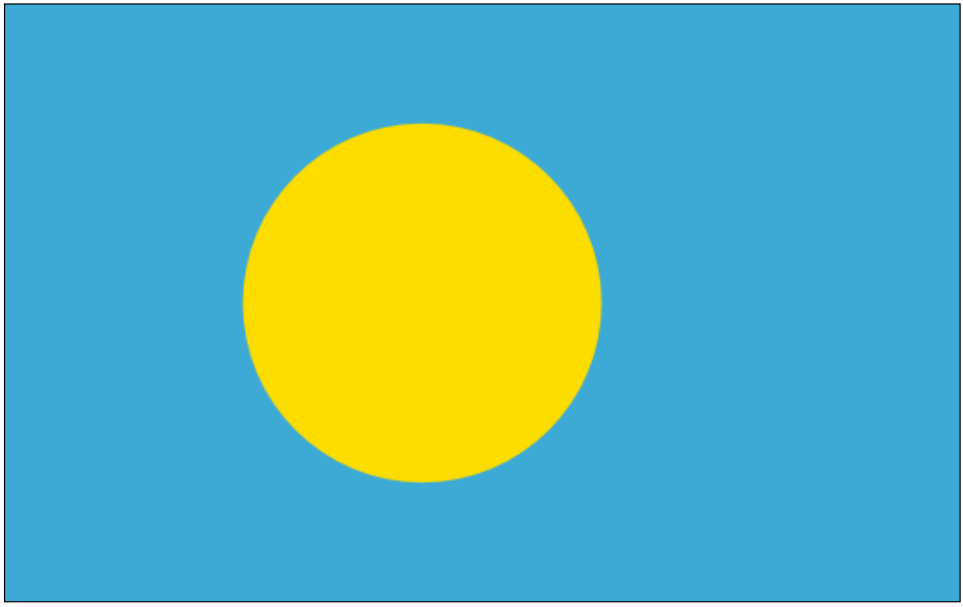
Starting from basic shapes

At this stage you know how to create a new canvas area and even create a few basic shapes. Let's expand our skill and start creating flags.

Getting ready

Well, we won't start from the most basic flag as that would just be a green rectangle. If you wanted to learn how to create a green flag you wouldn't need me, so let's move up to a tad more complex flag.

If you followed the *Graphics with 2D canvas* recipe you already know how to do it. This one is dedicated to our Palauan readers and to the perfect arc (also known as circle).



In this recipe we will ignore the HTML part, so if you need a refresher on how to create a canvas with an ID, please go back to the first recipe in this chapter and set up your HTML document. Don't forget to create the canvas with the right ID. You could also download our sample HTML files.

How to do it...

Add the following code block:

```
var cnvPalau = document.getElementById("palau");
  var wid = cnvPalau.width;
  var hei = cnvPalau.height;

  var context = cnvPalau.getContext("2d");
      context.fillStyle = "#4AADD6";
      context.fillRect(0,0,wid,hei);

      context.fillStyle = "#FFDE00";
      context.arc(wid/2.3, hei/2, 40, 0, 2 * Math.PI, false);
      context.fill();
```

That's it, you've just created a perfect arc, and with it your first flag that has a shape within it.

How it works...

A big chunk of this code should look very familiar at this stage. So I'll focus on the new lines compared to the ones used in the first recipe in this chapter.

```
var wid = cnvPalau.width;
var hei = cnvPalau.height;
```

In these lines, we extract the width and height of our canvas. We have two goals here: to shorten our lines of code and to reduce the number of times we make an API call when not needed. As we are using it more than one time, we first fetch the values and store them in `wid` and `hei`.

Now that we know our canvas width and height, it's time for us to draw our circle. Before we start drawing, we will call the `fillStyle` method to define a background color to be used in the canvas, and then we will create the arc followed by triggering the `fill` method when complete.

```
      context.fillStyle = "#FFDE00";
      context.arc(wid/2.3, hei/2, 40, 0, 2 * Math.PI, false);
      context.fill();
```

We then create our first perfect circle using the `arc` method. It's important to note that we can change the colors at any point, such as in this case, where we change our color just before we create a new circle.

Let's take a deeper look at how the `arc` method works. We start by defining the center of our circle with the `x` and `y` positions. The canvas tag follows the standard Cartesian coordinates: (0, 0) is at the top-left (`x` grows to the right and `y` grows towards the bottom).

```
context.arc(x, y, radius, startingAngle, endingAngle, ccw);
```

In our example, we decided to position the circle slightly to the left of the center by dividing the width of the canvas by `2.3`, and we positioned the `y` in the exact center of the canvas. The next parameter is the radius of our circle, It is followed by two parameters that define the starting and ending position of our stroke. As we want to create a full circle we start from `0` and end at two times `Math.PI`, a complete circle (`Math.PI` is equivalent to 180 degrees). The last parameter is the direction of our arc. In our case as we are creating a full circle, it doesn't matter what we set here (ccw = counterclockwise).

```
context.fill();
```

Last but not least, we call the `fill` function to fill and color the shape we created earlier. Contrary to the `fillRect` function that both creates and fills the shape, the `arc` method doesn't. The `arc` method only defines the bounds of a shape to be filled. You can use this method (and others) to create more complex shapes before actually drawing them onto the stage. We will explore this more deeply in the following recipes of this chapter.

Layering rectangles to create the flag of Greece

We learned as we created the flag for Palau that when we create a circle using the `arc` method, we have to trigger a request separately to fill the shape. This is true for all shapes that we create from scratch, and it is true for creating lines as well. Let's move to a slightly more complex flag: the flag of Greece.

Getting ready

As in the previous recipe, we will be skipping the HTML part and will jump right into the JavaScript portion of drawing in the canvas. For a detailed explanation of the steps involved in the creation of the canvas element, please refer to the first recipe of this chapter.

Before you start coding, look at the flag closely and try to come up with an attack plan on the steps you would need to perform to create this flag.

How to do it...

If we look at the flag, it's easy to figure out how to plan this out. There are many ways to do this but the following is our attempt:

1. We will first start our app and create a blank blue canvas:

```
var canvas = document.getElementById("greece");
var wid = canvas.width;
var hei = canvas.height;


var context = canvas.getContext("2d");
    context.fillStyle = "#000080";
    context.fillRect(0,0,wid,hei);
```

2. If you take a look at the previous figure, there are four white strips and five blue strips that will be part of the background. Let's divide the total height of our canvas by `9`, so we can find out the right size for our lines:

```
var lineHeight = hei/9;
```

3. So far we created shapes using built-in tools, such as `arc` and `fillRect`. Now we are going to draw our lines manually, and to do so we will set the `lineWidth` and `strokeStyle` values, so we can draw lines on the canvas:

```
context.lineWidth = lineHeight;
context.strokeStyle = "#ffffff";
```

4. Now, let's loop through and create four times a line that goes from the right-hand side to the left-hand side, as follows:

```
var offset = lineHeight/2;
for(var i=1; i<8; i+=2){
  context.moveTo(0,i*lineHeight + offset);
  context.lineTo(wid,i*lineHeight+offset);

}
```

That's it, we got it. Reload your HTML page and you will find the flag of Greece in all its glory. Well not in all its glory yet, but just about enough to guess it's the flag of Greece. Before we move on let's look deeper into how this works.

How it works...

Notice the addition of an offset. This is done because `lineWidth` grows in both directions from the actual point in the center of the line. In other words, a line with the width of 20 pixels that is drawn from (0, 0) to (0, height) would only have 10 pixels visible as the range of the thickness of the line would be between (-10 to 10). As such, we need to take into account that our first line needs to be pushed down by half its width so that it's in the right location.

The `moveTo` function takes in two parameters `moveTo(x,y)`. The `lineTo` function also takes two parameters. I believe you must have guessed the difference between them. One will shift the virtual point without drawing anything while the other will create a line between the points.

There's more...

If you run your HTML file, you will find that our lines were not revealed. Don't worry, you didn't make any mistake (At least, that's what I think ;)). For the lines to become visible, we need to tell the browser that we are ready, just like we called the `fill()` method when we used `arc`. In this case, as we are creating lines we will call the `stroke()` method right after we are done defining our lines, as follows:

```
var offset = lineHeight/2;
  for(var i=1; i<8; i+=2){
    context.moveTo(0,i*lineHeight + offset);
    context.lineTo(wid,i*lineHeight+offset);

  }
  context.stroke();
```

If you refresh the screen now you will see we are getting much closer. It's time for us to add that rectangle on the top-left area of the screen. To do that, we will reuse our `lineHeight` variable. The size of our rectangle is five times the length of `lineHeight`:

```
context.fillRect(0,0,lineHeight*5,lineHeight*5);
```

It is now time to create the cross in the flag:

```
context.moveTo(0, lineHeight*2.5);
context.lineTo(lineHeight*5,lineHeight*2.5);
context.moveTo(lineHeight*2.5,0);
context.lineTo(lineHeight*2.5,lineHeight*5+1);
context.stroke();
```

If you run the application now you will be really disappointed. We did exactly what we learned previously but it's not working as expected.

The lines are all mixed up! OK fear not, it means it's time for us to learn something new.

BeginPath method and closePath method

Our flag didn't pan out that well because it got confused by all the lines we created earlier. To avoid this, we should tell the canvas when we are starting a new drawing and when we are ending it. To do so we can call the `beginPath` and `closePath` methods to let the canvas know that we are done with something or are starting with something new. In our case by adding the method `beginPath` we can fix our flag issue.

```
context.fillRect(0,0,lineHeight*5,lineHeight*5);
context.beginPath();
context.moveTo(0, lineHeight*2.5);
context.lineTo(lineHeight*5,lineHeight*2.5);
context.moveTo(lineHeight*2.5,0);
context.lineTo(lineHeight*2.5,lineHeight*5+1);
context.stroke();
```

Congratulations! You just created your first two flags, and in the process learned a lot about how the canvas API works. This is enough to be able to create 53 country flags out of the 196 flags out there. That's a great start already; 25 percent of the world is in your hands.

The most complex flag you should be able to do right now is the flag of the United Kingdom. If you feel like exploring, give it a go. If you're really proud of it drop me a line at `ben@02geek.com`, I would love to see it.

Creating shapes using paths

We ended the last recipe learning how to create one fourth of the flags of the world, but that can't be the end of it, can it? This recipe will be dedicated to using paths to create more complex shapes. We will start by creating a triangle and progress from there to more complicated shapes.

Getting ready

Let's start from the simplest shape that isn't included in the basic shapes library: a triangle. So if you're ready let's get started...

How to do it...

Let's start with creating our first shape, a triangle:

```
context.fillStyle = color;
context.beginPath();
context.moveTo(x1,y1);
context.lineTo(x2,y2);
context.lineTo(x3,y3);
context.lineTo(x1,y1);
context.closePath();
context.fill();
```

The code here with points `x1,y1` through `x3,y3` is pseudocode. You would need to pick your own points to create a triangle.

How it works...

Most of the elements here aren't new. The most important change here is that we are creating the shape from scratch using the elements we worked with before. When we create a shape we always start by declaring it using the `beginPath()` method. We then create the shape and end the creation with the `closePath()` method. We will still not have anything visible on the screen until we decide what we want to do with the shape we created, such as show its fill or show its strokes. In this case as we are trying to create a triangle we will call the `fill` function.

Let's see this in action in a live flag sample. This time we will visit Mount Roraima in Guyana.

OK, so you get the idea of the triangle. Let's see this in action. I've extracted this code and put it into a function. To create this flag, we will need to create four triangles.

```
var canvas = document.getElementById("guyana");
var wid = canvas.width;
var hei = canvas.height;

var context = canvas.getContext("2d");
    context.fillStyle = "#009E49";
    context.fillRect(0,0,wid,hei);

fillTriangle(context,  0,0,
             wid,hei/2,
             0,hei, "#ffffff");
fillTriangle(context,0,10,
             wid-25,hei/2,
             0,hei-10, "#FCD116");
fillTriangle(context,0,0,
             wid/2,hei/2,
             0,hei, "#000000");
fillTriangle(context,0,10,
             wid/2-16,hei/2,
             0,hei-10, "#CE1126");

function fillTriangle(context,x1,y1,x2,y2,x3,y3,color){
  context.fillStyle = color;
  context.beginPath();
  context.moveTo(x1,y1);
  context.lineTo(x2,y2);
  context.lineTo(x3,y3);
  context.lineTo(x1,y1);
  context.closePath();
  context.fill();
}
```

By creating the `fillTriangle()` function we can now quickly and effectively create triangles just as we created rectangles. This function makes it a breeze to create a flag with such a rich numbers of triangles. Now, with the help of the `fillTriangle` method we can create any flag in the world that has triangles in it.

There's more...

Don't let triangles be your most complex shape, as you can create any number of pointed shapes. Let's create a more complex zigzag pattern. To do so, we will fly over to the Kingdom of Bahrain.

Try to locate the new logic before we break it down and explain it.

```
var canvas = document.getElementById("bahrain");
var wid = canvas.width;
var hei = canvas.height;

var context = canvas.getContext("2d");
    context.fillStyle = "#CE1126";
    context.fillRect(0,0,wid,hei);
var baseX = wid*.25;
    context.fillStyle = "#ffffff";
    context.beginPath();
    context.lineTo(baseX,0);

var zagHeight = hei/5;
for(var i=0; i<5; i++){
  context.lineTo(baseX +25 , (i+.5)*zagHeight);
  context.lineTo(baseX  , (i+1)*zagHeight);

}
context.lineTo(0,hei);
context.lineTo(0,0);
context.closePath();
context.fill();

addBoarder(context,wid,hei);
```

Let's break down this zigzag and understand what's going on here. After starting up with our normal setting up of a canvas element, we jump right into creating our shape. We start by drawing a red background, leaving us to create a shape that will have the white area. It's very much like a rectangle except that it has zigzags in it.

In this code, we start by creating a rectangle but our goal will be to change the highlighted code line with zigzags:

```
var baseX = wid*.25;
context.fillStyle = "#ffffff";
context.beginPath();
context.lineTo(baseX,0);
context.lineTo(wid*.25,hei);
context.lineTo(0,hei);
context.lineTo(0,0);
context.closePath();
context.fill();
```

In this code we set the fill color to white, we set our `beginPath` and then `lineTo` (starting at the point `(0,0)`, the default starting point) and create a rectangle that fills 25 percent of the width of the canvas. I've highlighted the horizontal line as this is the one we want to make zigzags with. By looking at the flag we can see that we are going to create five triangles going across the screen, so let's switch this line with a `for` loop:

```
...
context.lineTo(baseX,0);

var zagHeight = hei/5;
for(var i=0; i<5; i++){
  context.lineTo(baseX +25 , (i+.5)*zagHeight);
  context.lineTo(baseX  , (i+1)*zagHeight);

}

context.lineTo(0,hei);
  ...
```

So our first step before we can run through the loop is to decide how tall each triangle will be:

```
var zagHeight = hei/5;
```

We take the total height of the canvas and divide it by five to give us the height for each triangle.

We draw the zigzags in the `for` loop itself. To do so we need to use the following two lines of code in each round:

```
context.lineTo(baseX +25 , (i+.5)*zagHeight);
context.lineTo(baseX  , (i+1)*zagHeight);
```

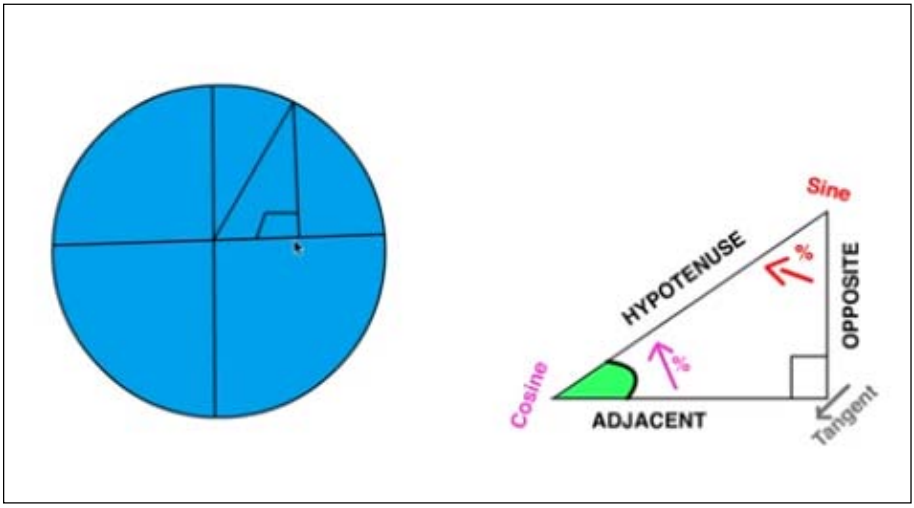
In the first line we step away from the current position and expand the line out half way through the height of the triangle, and to the extreme point on the right; and then on the second line we return back to the starting `x` point and update our `y` to the starting point of the next line segment. By the way, the addition of `baseX +25` is totally arbitrary. I just played with it until it looked good, but if you want you could use ratios instead (that way if you expand the canvas it would still look good).

The most amazing part of all of this is just knowing how to create some zigzags, triangles, rectangles, and circles. You can create an even larger number of flags but we are not done yet. Our quest to know how to create all the flags of the world continues.

If you are new to drawing via code or feel you can use some extra practice, just look at the map of the world and challenge yourself to create flags based on the skills we built already.

Creating complex shapes

It's time to take everything we learned and integrate it into the most complex shape we have seen so far, the Star of David. This star is part of the flag of Israel (one of my favorite flags in the world ;)). We need to take a roundabout before we can create it by visiting the magical world of sine and cosine.



You got to love it, right? I know many people fear cosines and sines but actually they are really easy and fun to work with. Let's break them down here in a more programming-for-drawing type of way. The most basic idea is that you have a triangle that has a 90 degree angle. You have some information about this triangle, and that's all you need to be able to start working with sine and cosine. Once you know that you have a 90 degree angle and know the sine/cosine, you have all the information you need and with it you can discover any missing information. In our case we know all the angles and we know the length of the hypotenuse (it's our radius; take a look at the image with the circle to see it in action). In JavaScript, the methods `Math.cos()` and `Math.sin()` are both representing a circle with a radius of one located on the (0,0) point on the screen. If we input the angle we are looking for into the `sin` function, it would return the `x` value (in this case the length of the adjacent) and the `cos` function would return the length of the opposite, in our case the required value `y`.

I've made a nice video which goes deeper into the logic. You can check it out at `http://02geek.com/courses/video/58/467/Using-Cos-and-Sin-to-animate.html`.

Getting ready

The easiest way to understand how sine/cosine work is by a live example, and in our case we will use it to help us figure out how to create the Star of David in the flag of Israel. We will take a step back and learn how we figured out the points on the screen to create the shapes. Again we will be skipping the creation of the HTML file and will go right into the JavaScript code. For an overview of how to get your HTML set, please review the *Graphics with 2D canvas* recipe.

How to do it...

After creating the JavaScript file add the following code in your `init` function.

1. Create our basic canvas variables:

```
var canvas = document.getElementById("israel");
var wid = canvas.width;
var hei = canvas.height;
var context = canvas.getContext("2d");
```

2. Define one degree in radians. We do that since `Math.cos` and `Math.sin` expect a radian value and not a degree value (`radian` is one degree measured in radians):

```
var radian = Math.PI/180;
```

3. Create a `tilt` variable. This variable will define the tilt of the triangle that will be created. Imagine the triangle is in a circle and we are rotating the circle with this `tilt` variable:

```
var tilt = radian*180;
```

4. Define the center point of the canvas:

```
var baseX = wid / 2;
var baseY = hei / 2;
```

5. Set the radius of the invisible bounding circle of the Star of David:

```
var radius = 24;
```

6. Define the height of the strips in the flag:

```
var stripHeight = 14;
```

7. Define a line width:

```
context.lineWidth=5;
```

8. Create two triangles (one tilted and one not):

```
createTrinagle(context,
  baseX+ Math.sin(0) * radius,
  baseY + Math.cos(0) * radius,
  baseX+ Math.sin(radian*120) * radius,
  baseY + Math.cos(radian*120) * radius,
  baseX+ Math.sin(radian*240) * radius,
  baseY + Math.cos(radian*240) * radius,
  null,"#0040C0");

createTrinagle(context,
  baseX+ Math.sin(tilt) * radius,
  baseY + Math.cos(tilt) * radius,
  baseX+ Math.sin(radian*120+tilt) * radius,
  baseY + Math.cos(radian*120+tilt) * radius,
```

```
  baseX+ Math.sin(radian*240+tilt) * radius,
  baseY + Math.cos(radian*240+tilt) * radius,
  null,"#0040C0");
```

9. Draw flag strips:

```
context.lineWidth=stripHeight;
context.beginPath();
context.moveTo(0,stripHeight);
context.lineTo(wid,stripHeight);
context.moveTo(0,hei- stripHeight);
context.lineTo(wid,hei- stripHeight);
context.closePath();
context.stroke();
```

10. Create the `createTriangle` function:

```
function createTriangle(context,x1,y1,x2,y2,x3,y3,fillColor,stroke
Color){
  context.beginPath();
  context.moveTo(x1,y1);
  context.lineTo(x2,y2);
  context.lineTo(x3,y3);
  context.lineTo(x1,y1);
  context.closePath();
  if(fillColor) {
    context.fillStyle = fillColor;
    context.fill();
  }
  if(strokeColor){
  context.strokeStyle = strokeColor;
  context.stroke();

  }
}
```

You are done. Run your application and you will find the flag of Israel with the Star of David in the center of the flag.

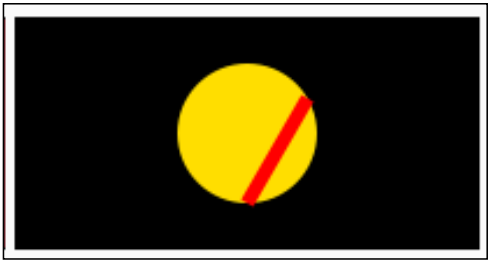
How it works...

Before we dig into the creation of the flag and how it was done, we need to understand how we locate points in a circle. To do so let's look at a simpler example:

```
var rad = Math.PI/180;
context.fillStyle = "#FFDE00";
context.arc(wid / 2, hei / 2, 30, 0, 2 * Math.PI, false);
context.fill();
```

```
context.beginPath();
context.strokeStyle = "#ff0000";
context.lineWidth=6;
context.moveTo(Math.sin(0) * 30 + wid / 2,
  Math.cos(0) * 30 + hei/2);
context.lineTo(Math.sin(rad*120) * 30 + wid / 2,
  Math.cos(rad*120) * 30 + hei/2);
context.stroke();
```

The following is the output the code will generate:



Although a circle, in our human-friendly head, is a shape that has 360 degrees, it's actually best represented in most programming languages in radians.

Radians are just like degrees, only instead of being human-friendly numbers between 0 and 360 these are numbers between 0 and two times Pi. You might be wondering what Pi is, so a bit more on Pi. Pi is in essence the value that is created when you take the circumference of any circle and divide it by the diameter of the same circle. The result that would come back would be Pi or about 3.14159. It is a magical number and the good news is you don't need to know much more about it if you don't want to. All you need to know is that 3.142 is equal to half of a circle. With that fact we can now divide Pi by `180` to get a value in radian that equals one degree:

```
var rad = Math.PI/180;
```

We then create a circle with a radius of `30` in the center of the screen, to help us visualize this, and move on to start creating a line that will start at angle `0` of our circle and end at angle `120` (as we want to create a triangle 360/3).

```
context.strokeStyle = "#ff0000";
context.lineWidth=6;
context.moveTo(Math.sin(0) * 30 + wid / 2,
  Math.cos(0) * 30 + hei/2);
context.lineTo(Math.sin(rad*120) * 30 + wid / 2,
  Math.cos(rad*120) * 30 + hei/2);
context.stroke();
```

Let's break down the most complex line:

```
context.lineTo(Math.sin(rad*120) * 30 + wid / 2,
  Math.cos(rad*120) * 30 + hei/2);
```

As `Math.sin` and `Math.cos` return a value for a radius of `1`, we will multiply any value returned by the radius of our circle (in this case `30`). In the parameters of `Math.sin` and `Math.cos`, we will provide the exact same values; in this example `120` radians. As our circle would be centered at the top left-hand side of the canvas we want to shift the circle to start from the center of the screen by adding to our values `wid/2` and `hei/2`.

At this stage, you should know how to find points on a circle, and with that how to draw lines between two points. Let's go back to our flag of Israel and take a deeper look into the new function `createTriangle`. It was based on the function `fillTriangle` created in the *Creating shapes using paths* recipe.

```
function
createTriangle(context,x1,y1,x2,y2,x3,y3,fillColor,strokeColor){

...

  if(fillColor) {
    context.fillStyle = fillColor;
    context.fill();
  }

if(stokeColor){
    context.strokeStyle = fillColor;
    context.stroke();

  }

}
```

I've highlighted the new components of this function compared to the function `fillTriangle`. The two new parameters `fillColor` and `strokeColor` define if we should fill or stroke the triangle. Notice that we moved the `strokeStyle` and `fillStyle` methods to the bottom of our function to reduce our code footprint. Great! We now have a modern triangle creator that could deal with the Star of David.

There's more...

OK, time to connect the dots (literally speaking) and create the flag of Israel. Looking back at our original code we find ourselves using the `createTriangle` function twice to create the full Star of David shape. Let's take a deeper look at the logic here by looking at the second triangle (the one that is turned upside down):

```
createTriangle(context,
  baseX+ Math.sin(tilt) * radius,
  baseY + Math.cos(tilt) * radius,
  baseX+ Math.sin(radian*120+tilt) * radius,
  baseY + Math.cos(radian*120+tilt) * radius,
baseX+ Math.sin(radian*240+tilt) * radius,
  baseY + Math.cos(radian*240+tilt) * radius,
  null,"#0040C0");
```

We are sending in three points on the virtual circle to create a triangle. We split our virtual circle to three equal parts and find the point values at the `0`, `120`, and `240` degrees. This way if we drew a line between these points we would get a perfect triangle in which all of the sides were equal.

Let's take a deeper look at one of the points sent to the `createTriangle` function:

```
baseX + Math.sin(radian*120+tilt) * radius,
baseY + Math.cos(radian*120+tilt) * radius
```

We start from `baseX` and `baseY` (the center of the screen) as the center point of our circle before we figure out the actual point gap from that base starting point. We then add to it the value that we get from `Math.sin` and `Math.cos` respectively. In this example, we are trying to get `120` degrees plus the tilt value. In other words, `120` degrees plus `180` degrees (or `300` degrees).

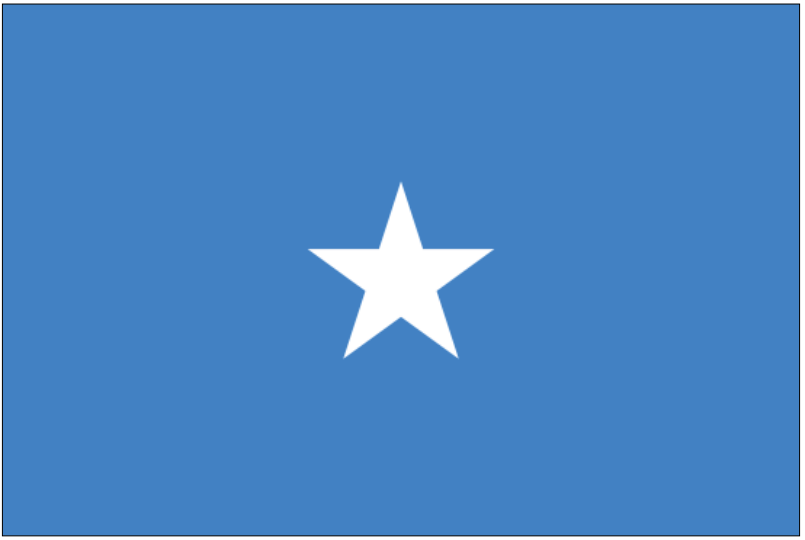
To make it easier to comprehend, in pseudocode it would look similar to the following code snippet:

```
startingPositionX + Math.sin(wantedDegree) * Radius
startingPositionY + Math.cin(wantedDegree) * Radius
```

Not much more to say besides congrats. We just finished creating another flag and in the process, learned how to create complex shapes, use math to help us figure out points on the screen, and mix together different shapes to create more advanced shapes.

Adding more vertices

There are many flags that contain stars that just cannot be created by overlapping triangles. In this recipe, we will figure out how to create a star that contains an arbitrary number of points We will use the same key concept we discovered in the previous recipe by taking advantage of a virtual circle to calculate positions, this time with only two virtual circles. In this recipe, we will create the flag of Somalia and in the process figure out how to create a function that will enable us to create stars.



Getting ready

Please continue working on the sample from the previous recipe. If you haven't worked on it yet, I strongly encourage you to do so as this recipe is the next logical step of the previous recipe. As in the previous recipe, we will be skipping the HTML portion of this sample. Please review the first recipe in the book to refresh on the required HTML code.

How to do it...

Let's jump right in and create the flag of Somalia.

1. Create the canvas standard logic:

```
var canvas = document.getElementById("somalia");
var wid = canvas.width;
var hei = canvas.height;

var context = canvas.getContext("2d");
```

2. Fill the background color of canvas:

```
context.fillStyle = "#4189DD";
context.fillRect(0,0,wid,hei);
```

3. Draw the star by calling the `createStar` function:

```
createStar(context,wid/2,hei/2,7,20,5,"#ffffff",null,0);
```

4. Create the `createStart` function:

```
function createStar(context,baseX,baseY,
                    innerRadius,outerRadius,
                    points,fillColor,
                    strokeColor,tilt){
// all the rest of the code in here
}
```

5. From this point on we will be working within the `createStart` function. Add a few helper variables:

```
function createStar(context,baseX,baseY,innerRadius,outerRadius,
  points,fillColor,strokeColor,tilt){
  var radian = Math.PI/180;
  var radianStepper = radian * ( 360/points) /2;
  var currentRadian =0;
  var radianTilt = tilt*radian;
```

6. Call the `beginPath` method before starting to draw any shape:

```
  context.beginPath();
```

7. Move the drawing pointer to the angle `0` in the internal circle:

```
  context.moveTo(baseX+ Math.sin(currentRadian +
    radianTilt) * innerRadius,baseY+
    Math.cos(currentRadian + radianTilt) * innerRadius);
```

8. Loop through the total points of the star and draw a line back and forth between the outer circle and inner circle to create a star:

```
for(var i=0; i<points; i++){
  currentRadian +=  radianStepper;
  context.lineTo(baseX+ Math.sin(currentRadian +
    radianTilt) * outerRadius,baseY+
    Math.cos(currentRadian + radianTilt) * outerRadius);
  currentRadian +=  radianStepper;
  context.lineTo(baseX+ Math.sin(currentRadian +
    radianTilt) * innerRadius,baseY+
    Math.cos(currentRadian + radianTilt) * innerRadius);
}
```

9. Close the path of the drawing and fill or stroke according to the function parameters:

```
context.closePath();

  if(fillColor){
    context.fillStyle = fillColor;
    context.fill();
  }

  if(strokeColor){
    context.strokeStyle = strokeColor;
    context.stroke();

  }

}
```

When you run your HTML wrapper, you will find your first star and with it another flag will be under your belt.

How it works...

Let's start by understanding what the function we are going to create expects. The idea is simple, to create a star we want to have a virtual inner circle and a virtual outer circle. We can then draw lines between the circles back and forth to create the star. To do so, we need some basic parameters.

```
function createStar(context,baseX,baseY,
     innerRadius,outerRaduis,points,fillColor,
                              strokeColor,tilt){
```

Our regular context, `baseX` and `baseY` don't need further introductions. The virtual `innerRadius` and `outerRadius` are there to help define the length of the line segments that create a star and their positions. We want to know how many points our star will have. We do so by adding in the `points` parameters. We want to know the `fillColor` and/or `strokeColor` so we can define the actual colors of the star. We top it with a `tilt` value (it can be useful as we've seen when creating the Star of David for the flag of Israel).

```
var radian = Math.PI/180;
var radianStepper = radian * ( 360/points) / 2;
var currentRadian =0;
var radianTilt = tilt*radian;
```

We then move on to configure our facilitator variables for our star. It's not the first time we see the radian variable, but it is our first `radianStepper`. The goal of the radian stepper is to simplify calculations in our loop. We divided 360 degrees by the number of points our triangle will have. We divided the value by `2`, as we will have two times the number of points as lines. Last but not least, we want to convert this value into radians so we are duplicating the full results by our radian variable. We then create a simple `currentRadian` variable to store the current step we are in and finish off by converting the `tilt` value to be a radian value, so we can add it into all our lines without extra calculations within the loop.

As always, we start and complete our shapes with the `beginPath` and `closePath` methods. Let's take a deeper look at the starting position for our soon-to-be shape:

```
context.moveTo(baseX+ Math.sin(currentRadian + radianTilt) *
  innerRadius,baseY+ Math.cos(currentRadian + radianTilt) *
  innerRadius);
```

Although at first glance this probably looks a bit scary, it's actually very similar to how we created the Star of David. We are starting at `currentRadian` (that is currently `0`) using `innerRadius` as our start point.

In our loop, our goal will be to weave back and forth between the inner and external circles. To do so we will need to progress the `currentRadian` value each time the loop cycles by a `radianStepper`:

```
for(var i=0; i<points; i++){
  currentRadian +=  radianStepper;
  context.lineTo(baseX+ Math.sin(currentRadian + radianTilt) *
    outerRadius,baseY+ Math.cos(currentRadian + radianTilt) *
    outerRadius);
  currentRadian +=  radianStepper;
  context.lineTo(baseX+ Math.sin(currentRadian + radianTilt) *
    innerRadius,baseY+ Math.cos(currentRadian + radianTilt) *
    innerRadius);
}
```

We start a loop based on the number of points in our parameter. In this loop, we go back and forth between the external radius and the internal one each time we draw two lines between the inner circle and the external one. Our step size is defined by the number of points (the value we configured with the `radianStepper` variable).

We covered the rest of the functions when we created the `createTriangle` function in an earlier recipe. There you have it! You can now run the app and find our seventh flag. With this new complex function, we can create all solid stars and all non-solid stars that are hollow within.

OK I hope you are sitting down... with the newly-acquired star powers, you can now create at least 109 flags including the United States of America and all the other countries that have stars in their flag (57 percent of the countries in the world and counting!).

Overlapping shapes to create other shapes

There are many flags and many shapes in general that can be created by combining the shapes we created so far. One of the most popular shapes in 82 flags we don't know how to create is the crescent shape like the one in the flag of Turkey. With it we learn a new skill of using subtraction to create more in-depth shapes.

Getting ready

The previous recipe is our starting point in this recipe. From here, we will continue working to create more advanced shapes that are built out of two shapes when combined. As such, we will be using the code created in the last recipe located in `01.02.flags.js`.

How to do it...

Let's jump right into our code and see it in action.

1. Gain access to the context and save the width and height of the canvas into variables:

```
var canvas = document.getElementById("turkey");
var wid = canvas.width;
var hei = canvas.height;

var context = canvas.getContext("2d");
```

2. Fill the rectangle canvas area:

```
context.fillStyle = "#E30A17";
context.fillRect(0,0,wid,hei);
```

3. Create a full circle:

```
context.fillStyle = "#ffffff";
context.beginPath();
context.arc(wid / 2 - 23, hei / 2, 23, 0, 2 *
  Math.PI, false);
context.closePath();
context.fill();
```

4. Change the color of canvas fill. Fill a circle within its bound with another circle that hides part of the last circle that was created. This effect creates a shape that looks like a crescent moon:

```
context.fillStyle = "#E30A17";
context.beginPath();
context.arc(wid / 2 - 18, hei / 2, 19, 0, 2 *
  Math.PI, false);
context.closePath();
context.fill();
```

5. Reuse `createStart` from the previous recipe to add the Turkish star:

```
createStar(context,wid/2 +
  13,hei/2,5,16,5,"#ffffff",null,15);
```

There you go! You've just created a shape that is not possible without masking one shape with another.

How it works...

The catch here is we are using two circles, one overlaps the other to create a crescent shape. By the way, notice how we are tilting the star as well so that one of its points will point to the middle of the circle.

We've gone through a lot in the last few examples and at this stage you should be very comfortable creating many shapes and elements in the canvas. There is still much to explore before we can say we have mastered canvas, but we can definitely say we have mastered creating most of the flags of the world and that's very cool. I would love to see your flags. Drop me a line when you create one not in the book! :)