2
Advanced Drawing in Canvas

- Drawing arcs
- Drawing curves with a control point
- Creating a Bezier curve
- Integrating images into our art
- Drawing with text
- Understanding pixel manipulation

Introduction

This is the last chapter where we will dig deep into canvas as the remaining chapters will focus on building charts and interactivity.

In this chapter, we will continue to master our skills with canvas by adding curves, images, text, and even pixel manipulation to our tool belt.

Drawing arcs

There are three types of curves we can create in canvas—using the arc, quadratic curves, and Bezier curves. Let's get started.

Getting ready

If you recall in *Chapter 1*, *Drawing Shapes in Canvas*, in our first recipe we used the arc method to create perfect circles. The arc method is much more than just that. We can actually create any partial curve in a circular shape. If you don't recall drawing circles, I strongly encourage you to scan through *Chapter 1*, *Drawing Shapes in Canvas* again, and while you are there, you will find the template for creating the HTML documents as well. We are exclusively going to focus on the JavaScript code in this recipe.

How to do it...

Let's jump into it and create our first noncircle that has curves:

1. Access the `pacman` canvas element and fetch its width and height by using the following code snippet:

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

2. Create a `radian` variable (one degree in radians):

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

3. Get the canvas context and fill its background in black by using the following code snippet:

```
var context = canvas.getContext("2d");
  context.fillStyle = "#000000";
  context.fillRect(0,0,wid,hei);
```

4. Begin a new path before starting to draw:

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

5. Change fill style color:

```
  context.fillStyle = "#F3F100";
```

6. Move the pointer to the center of the screen:

```
  context.moveTo(wid/2,hei/2);
```

7. Draw a curve that starts at 40 degrees and ends at 320 degrees (with a radius of 40) in the center of the screen:

```
  context.arc(wid / 2, hei / 2, 40, 40*radian, 320*radian,
  false);
```

8. Close the shape by drawing a line back to the starting point of our shape:

```
context.lineTo(wid/2,hei/2);
```

9. Close the path and fill the shape:

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

You have just created a PacMan.

How to do it...

For the first time, we take advantage and create a pie-type shape, known as PacMan (you can see how this can be very useful when we get into creating the pie graph). Very simple—again we connect to that idea of radians:

```
context.arc(wid / 2, hei / 2, 40, 40*radian, 320*radian, false);
```

Notice how our 4th and 5th parameters—instead of being a complete circle by starting from 0 and ending at `2*Math.PI`—are setting the angle to start the arc at radian 40 and end at radian 320 (leaving 80 degrees open to create the mouth of a PacMan). All that is left is to start drawing from the center of the circle:

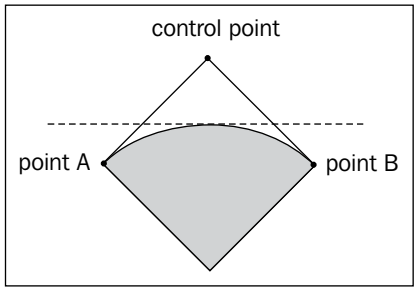
```
context.moveTo(wid/2,hei/2);
context.arc(wid / 2, hei / 2, 40, 40*radian, 320*radian, false);
context.lineTo(wid/2,hei/2);
```

We start by moving our pointer to the center of our circle. We then create the arc. As our arc isn't a complete shape it's continuing where we left off—drawing a line from the center of the arc to the starting point (40 degrees). We complete the action by drawing a line back to the center of the arc to complete the shape. Now we are ready to fill it and complete our work.

Now that we have got arcs out of the way, you can see how useful this will be for creating pie charts.

Drawing curves with a control point

If the world was just two points with a perfect arc this would be the end of the book, but alas or lucky for us, there are still many more complex shapes to learn and explore. There are many curves that are not perfectly aligned curves. Till now all the curves that we created were part of a perfect circle, but not any more. In this recipe, we will explore quadratic curves. The quadratic curves enable us to create curves that are not circular, by adding a third point—a controller to control the curve. You can easily understand this by looking at the following diagram:



A **quadratic curve** is a curve that has one control point. Consider the case when creating a line, we draw it between two points (A and B in this illustration). When we want to create a quadratic curve, we use an external gravity controller that defines the direction of the curve while the middle line (the dotted line) defines how far will the curve reach.

Getting ready

As done in previous recipes, we are skipping the HTML part here too—not that it's not needed, it just repeats itself in every recipe and if you need to refresh yourself on how to get the HTML setup, please take a look at the *Graphics with 2D Canvas* recipe in *Chapter 1*, *Drawing Shapes in Canvas*.

How to do it...

In this example, we will create a closed shape that looks like a very basic eye.
Let's get started:

1. We always need to start with extracting our canvas element, setting up our width and height variables, and defining a radian (as we find it useful to have one around):

```
var canvas = document.getElementById("eye");
  var wid = canvas.width;
  var hei = canvas.height;
  var radian = Math.PI/180;
```

2. Next, fill our canvas with a solid color and after that begin a new shape by triggering the `beginPath` method:

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

3. Define the line width and stroke color for our eye shape:

```
  context.lineWidth = 1;
  context.strokeStyle = "#000000"; // line color
  context.fillStyle = "#ffffff";
```

4. Move our drawing pointer to the left-centered point as we will need to draw a line from left to right in the center of the screen and back (only with a curve):

```
  context.moveTo(0,hei/2);
```

5. Draw two quadratic curves from our initial point to the other side of the canvas and back to the initial point by using an anchor point, which is in the extreme top followed by the extreme bottom of the canvas area:

```
  context.quadraticCurveTo(wid / 2, 0, wid,hei/2);
  context.quadraticCurveTo(wid / 2, hei, 0,hei/2);
```

6. Close the path. Fill the shape and use the `stroke` method on the shape (`fill` for filling the content and `stroke` for outlines):

```
  context.closePath();
  context.stroke();
  context.fill();
```

Great job! You have just created your first shape by using the `quadraticCurveTo` method.

How it works...

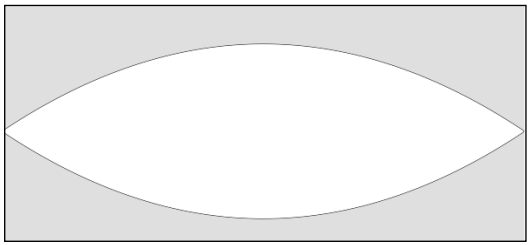
Let's look at this method closely:

```
context.quadraticCurveTo(wid / 2, 0, wid,hei/2);
```

As we are already at the origin point (point A), we input two other points—the control point and point B.

```
context.quadraticCurveTo(controlX, controlY, pointB_X, pointB_Y);
```

In our sample, we create a contained shape—the starting point to create an eye. Play with the controller to see how it affects the direction and size of the curve. The thumb rule is that the closer to the vertical line the less steep the curve will be, and the further away it is from the center point the more curved shape would be to the offset.



Creating a Bezier curve

We've just learned that with the quadratic curve we have one control point. Although we can do many things with one control point, we don't really have full control over the curve. So let's take it one step further by adding one more control point. Adding a second control point actually adds the relationship between these two points as well making it three control factors. If we include the actual anchor points (we have got two of them), we end up with five points that control the shape of the curve. That does sound complicated; it's because the more control we get the more complicated it is to actually understand how it works. It's really not easy to figure out complicated curves by code alone and as such we actually use other tools to help us figure out the right curves.

To prove the preceding point, we can find a very complex shape and start with that one (don't worry, later on in this recipe, we will practice on a very simple shape to make the concept clear). We will pick to draw the flag of Canada and mainly the maple leaf.

Getting ready

This recipe is difficult to understand, but we will break it down into details in the following *How it works...* section. So if you are new to curves, I strongly encourage you to start learning from this *How it works...* section before implementing it.

How to do it...

Let's create the flag of Canada. Let's jump right into the JavaScript code:

1. Create the canvas and context:

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

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

2. Fill the background to match the background of the Canadian flag:

```
context.fillStyle="#FF0000";
context.fillRect(0,0,50,100);
context.fillRect(wid-50,0,50,100);
```

3. Begin a new path and move the pointer to `84,19`:

```
context.beginPath();
context.moveTo(84,19);
```

4. Draw curves and lines to create the maple leaf:

```
context.bezierCurveTo(90,24,92,24,99,8);
context.bezierCurveTo(106,23,107,23,113,19);
context.bezierCurveTo(108,43,110,44,121,31);
context.bezierCurveTo(122,37,124,38,135,35);
context.bezierCurveTo(130,48,131,50,136,51);
context.bezierCurveTo(117,66,116,67,118,73);
context.bezierCurveTo(100,71,99,72,100,93);
context.lineTo(97,93);
context.bezierCurveTo(97,72,97,71,79,74);
context.bezierCurveTo(81,67,80,66,62,51);
context.bezierCurveTo(67,49,67,48,63,35);
context.bezierCurveTo(74,38,75,37,77,31);
context.bezierCurveTo(88,44,89,43,84,19);
```

5. Close the path and fill the shape:

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

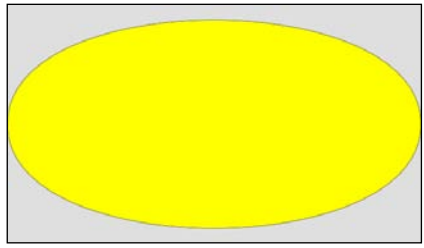
Now, you have created the flag of Canada. I don't know if you already know why it works or how we got to the apparently random numbers that we put into our curves to create the flag, but you've created the flag of Canada! Don't worry, we are about to decrypt the magic of curves right away in the next section.

How it works...

Before we can explain the details of how the Canadian flag works, we should take a step back and create a simpler example. In this short example, we will create an oval shape by using the `bezierCurveTo` method.

```
context.moveTo(2,hei/2);
  context.bezierCurveTo(0, 0,wid,0, wid-2,hei/2);
  context.bezierCurveTo(wid, hei,0,hei, 2,hei/2);
  context.closePath();
  context.stroke();
  context.fill();
```

That's it. The following is the outcome you get out of this:



If you get this, you are in great shape. We will now explain how this works and then move into how we figured out all the points for the Canadian flag. We are taking advantage of the full canvas again and we are keeping our controllers under control by setting two of our controllers to be the corners of the canvas:

```
context.bezierCurveTo(controlPointX1, controlPointY1, controlPointX2,
controlPointY2, pointBX, pointBY);
```

Play around with the controllers to see how much more control you get by using two dots—this is very useful when you need more detailed control over a curve.

This is the heart of the full example of our full flag. I strongly encourage you to explore the effects of changing the values of the control points to get a better understanding and sensitivity to it. It's time for us to get back to our flag and see how we structured it.

It's time to take our most complex drawing style—Bezier curves—and put them to action with something a bit more interesting than an oval. I have a confession: when I decided to create the flag of Canada from scratch I got scared. I was thinking "How am I going to get this done? This is going to take me hours," and then it hit me... it was clear that this flag needs to be created with a lot of Bezier points but how would I know where the points should be? So for a shape this advanced, I opened up my graphics editor (in my case, Flash Editor) and added pivot points to the maple shape:

If you closely look at the previous diagram, I basically traced the flag of Canada and placed a black dot on every sharp corner. Then I created a canvas and drew lines to see if the base shape I got was in the right overall position (by the way, I got the dots just by selecting the dots in Flash to see if their (x, y) coordinates as Flash and canvas coordinate systems are the same).

```
var context = canvas.getContext("2d");
context.beginPath();
context.moveTo(84,19);
context.lineTo(99,8);
context.lineTo(113,19);
context.lineTo(121,31);
context.lineTo(135,35);
context.lineTo(136,51);
context.lineTo(118,73);
context.lineTo(100,93);
context.lineTo(97,93);
context.lineTo(79,74);
context.lineTo(62,51);
context.lineTo(63,35);
context.lineTo(77,31);
context.lineTo(84,19);

context.closePath();
context.stroke();
```

I got a shape that was far from what I was looking for. But now I knew that my shape was getting formed in the right direction. What were missing were the curves to connect between the dots. If you look at the preceding diagram again, you will notice that I've placed two blue points between each sharp corner to define where the curves would be and how sharp or soft they would be. I then moved back into canvas and then updated the values to have the two control points. I added all the curves and switched from creating strokes to creating a fill.

```
var context = canvas.getContext("2d");
  context.fillStyle="#FF0000";
  context.fillRect(0,0,50,100);
  context.fillRect(wid-50,0,50,100);

  context.beginPath();
  context.moveTo(84,19);
  context.bezierCurveTo(90,24,92,24,99,8);
  context.bezierCurveTo(106,23,107,23,113,19);
  context.bezierCurveTo(108,43,110,44,121,31);
  context.bezierCurveTo(122,37,124,38,135,35);
  context.bezierCurveTo(130,48,131,50,136,51);
  context.bezierCurveTo(117,66,116,67,118,73);
  context.bezierCurveTo(100,71,99,72,100,93);
  context.lineTo(97,93);
  context.bezierCurveTo(97,72,97,71,79,74);
  context.bezierCurveTo(81,67,80,66,62,51);
  context.bezierCurveTo(67,49,67,48,63,35);
  context.bezierCurveTo(74,38,75,37,77,31);
  context.bezierCurveTo(88,44,89,43,84,19);
  context.closePath();
  context.fill();
```

Bingo! I just got an almost perfect flag and I feel this is enough for this sample.

Don't try to create very complex shapes on your own just yet. Maybe there are a few people out there who can do that, but for the rest of us the best way to do it is to trace the elements by using a visual editor of some sort. We can then grab the graphic information and update the values in canvas as I've done with the Canadian flag example.

At this stage, we have covered the most complex shapes that we can cover in canvas. The rest of the chapter is dedicated to other ways of manipulating content on the screen.

Integrating images into our art

Lucky for us, we don't need to start from scratch always and we can leave the more complex art for external images. Let's figure out how we can integrate images into our canvas.

Getting ready

We've been in a flag theme in this chapter and getting another flag under our belt sounds real good to me right now. So let's turn our heads to Haiti and get their flag up and running. To create this flag, we need to have the image of the symbol that is placed in the center of the flag.

In the source files, you will find an image of the center graphic (at `img/haiti.png`). By the way, when integrating art into canvas it's always best to avoid resizing the image whenever possible via code to preserve the image quality.

How to do it...

We will prepare the background to match the flag and then put the entire image above it in the center of the canvas/flag:

1. Follow the basic steps that we need to access the canvas. Set the width, height, and the actual context:

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

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

2. Draw the background elements:

```
context.fillStyle="#00209F";
context.fillRect(0,0,wid,hei/2);
context.fillStyle="#D21034";
context.fillRect(0,hei/2,wid,hei/2);
```

3. Create a new `Image` object:

```
var oIMG = new Image();
```

4. Create an `onLoad` function (that will be called when the image is loaded):

```
oIMG.onload = function(){
context.drawImage(this, (wid-this.width)/2, (hei-this.height)/2);
};
```

5. Set the source of the image:

```
oIMG.src = "img/haiti.png";
```

Yes, it's that easy to add images into canvas, but let's review more deeply what we have just done.

How it works...

The steps involved in creating an image are downloading its data and then creating a new image container in the same way as it is done in canvas:

```
var oIMG = new Image();
```

The next step is to create a listener that will be triggered when the image is loaded and ready to be used:

```
oIMG.onload = theListenerFunctionHere;
```

The last step in the loading process is to tell canvas what image should be loaded. In our case we are loading `img/haiti.png`:

```
oIMG.src = "img/haiti.png";
```

Loading an image and having it ready to be used is only the first step. If we ran our application without actually telling canvas what to do with it, nothing would happen beyond the loading of the image.

In our case, when our listener is triggered, we add the image as it is to the center of the screen:

```
context.drawImage(this, (wid-this.width)/2, (hei-this.height)/2);
```

That is all it takes to integrate an image into a canvas project.

There's more...

There are more operations that we can do with images in canvas beyond using them as backgrounds. We can define exactly what parts of the image we want (scaling). We can resize and manipulate the full image (scaling). We can even pixel manipulate our images. There are many things that we can do with images, but in the next few topics we will cover some of the more often used ones.

Scaling images

We can scale the image by adding two more parameters to the `drawImage` function, which sets the width and height of our image. Try the following:

```
context.drawImage(this, (wid-this.width)/2, (hei-this.height)/2 , 100,
120);
```

In the previous sample, we are loading the same image but we are forcing a resized image (note that the positions are not going to be in the actual center of the stage).

Adding even more control

You can control many aspects of an image. If you need more control than the preceding sample, you would need to input the full number of possible coordinates:

```
context.drawImage(this, sourceX, sourceY, sourceWidth, sourceHeight,
destX, destY, destWidth, destHeight);
```

In this case, the order has changed (notice that!). Now, the first two parameters after `this` are the local x and y coordinates of the image followed by the width and height (creating the crop that we were talking about) followed by the position on the canvas and its controlling information (x, y, width, and height).

In our case:

```
context.drawImage(this, 25,25,20,20,0,0,50,50);
```

The preceding code line means we want to take the image from its internal position of (25,25) and we want to slice a 20 x 20 rectangle out of there. We then want to position this new cropped image at (0,0) that is, the top corner of the canvas and we want that output to be a 50 x 50 rectangle.

Using images as a fill

We can use our loaded image as a way to fill up objects as well:

```
var oIMG = new Image();
  oIMG.onload = function(){
    var pattern = context.createPattern(this, "repeat");
    createStar(context,wid/2,hei/2,20,50,20,pattern,"#ffffff",20);
  };
  oIMG.src = "img/haiti.png";
```

After the image is loaded (always after the image is loaded, you start manipulating it), we create a pattern that repeats based on our image:

```
var pattern = context.createPattern(this, "repeat");
```

We can then use this pattern as our fill. So in this case, we are calling the `createStar` that we created in an earlier task—drawing a star in the center of the screen—by using the following pattern:

```
createStar(context,wid/2,hei/2,20,50,20,pattern,"#ffffff",20);
```

This ends our flag obsession to move on to shapes that just don't appear in flags. By the way, at this stage you should be able to create all the flags in the world and take advantage of integrating images when it's just not fun to draw it yourself from scratch—such as detailed country logos.

Drawing with text

I agree, we've been working on some complicated things. Now, its time for us to lay back, kick off the shoes, and do something a bit easier.

Getting ready

The good news is, if you are on this page, you should already know the basics of getting a canvas up and running. So there isn't much more that you need to do besides picking the font, size, and position of your text.

Here, we aren't covering how you can embed fonts that aren't created within JavaScript, but instead, via CSS, we will use a basic font and hope for the best in this sample.

How to do it...

In this example, we are going to create a text field. In this process, we are going to use gradients and shadows for the first time. Perform the following steps:

1. Gain access to the canvas 2D API:

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

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

2. Create a gradient style and fill the background with it:

```
var grd = context.createLinearGradient(wid/2, hei/2, wid, hei);
  grd.addColorStop(0, "#8ED6FF");
  grd.addColorStop(1, "#004CB3")
  context.fillStyle= grd;
  context.fillRect(0,0,wid,hei);
```

3. Create a gradient to be used by the text:

```
  grd = context.createLinearGradient(100, hei/2, 200,
  hei/2+110);
  grd.addColorStop(0, "#ffff00");
  grd.addColorStop(1, "#aaaa44");
```

4. Define the font to be used and set the style:

```
  context.font = "50pt Verdana, sans-serif";
  context.fillStyle = grd;
```

5. Add shadow details before drawing the text:

```
  context.shadowOffsetX = 0;
  context.shadowOffsetY = 0;
  context.shadowBlur    = 8;
  context.shadowColor   = 'rgba(255, 255, 255, 0.5)';
```

6. Use `fillText` to fill the shape and `strokeText` for outlines of the shape (notice that I call the text a shape; this is because as soon as we draw it, it will just be a part of our canvas and not live text):

```
  context.fillText("Hello World!", 100, hei/2);
  context.strokeStyle = "#ffffff";
  context.strokeText("Hello World!", 100, hei/2);
```

That's it, we just integrated our first drawn text into canvas.

How it works...

Until now, we were stuck with the solid colors. We will now break out of that and move to a new world of gradient colors. Refer to the following code snippet:

```
var grd = context.createLinearGradient(wid/2, hei/2, wid, hei);
  grd.addColorStop(0, "#8ED6FF");
  grd.addColorStop(1, "#004CB3");
```

There are a few steps involved with creating a gradient. The first step is defining its scope:

```
var grd = context.createLinearGradient(x1, y1, x2, y2);
```

Contrary to many other languages, it's really easy to define the rotation and size of a gradient in canvas. If you have worked with Photoshop before, you will find this really easy (even if you haven't, it will be easy).

All you need to do is define where you want the gradient to start and where you want it to end. You can send two dots into the method `createLinearGradient`:

```
grd.addColorStop(0, "#8ED6FF");
grd.addColorStop(1, "#004CB3");
```

In this transition, we are using only two colors. Position them at a value between 0 and 1. These values are ratios, so we are, in other words, requesting to spread the color transition from the start of the gradient area all the way to the end. We could add more colors, but our goal is to bind them all within the ratio 0 to 1. The more colors you add, the more playing around you would need to do with the values sent into the first parameter.

You just completed creating a gradient. Time to use it:

```
context.fillStyle= grd;
context.fillRect(0,0,wid,hei);
```

In this part again, we will use the `fillStyle` method and then create a rectangle.

Please note the importance of the range of values that you may send to the `addColorStop` method. As you add more colors into your gradient, the more noticeable the importance of the values sent here will be. The points are not counters but ratios of colors in our sample. The transition is between the two colors' range from 0 to 1 or in other words they transition all the way through from our first point that we send into the `createLinearGradient` method all the to the last point. As we are working with two colors, this is the perfect ratio for us.

Although we are not getting into radial gradients, they should be really easy for you as we have already learned a lot about radial shapes and gradients. The signature of this method is as follows:

```
context.createRadialGradient(startX,startY,startR, endX,endY,endR);
```

The only difference here is that our shape is a radial shape. We also want to add the starting radius and ending radius into it. You might be wondering why we need two or even more radii. So why can't we figure out the radius based on the distance between the two dots (start point and end point)? I hope you are wondering about that and if you are not, wonder about it for a second before reading the next paragraph.

We have a separate control over the radius, mainly to enable us to separate the radius and to enable us to move the focal point within the drawing without changing the actual art or recalculating the ratios of colors. A really great way to see this in use is when drawing the moon. The moon's gradients over time will change or more accurately the radius of the colors and position of the radius would change over time depending on the moon's position compared to the sun.

We are not done yet. We just mastered everything that we need to know about gradients and it's time for us to integrate some text into it.

```
context.font = "50pt Verdana, sans-serif";
context.fillText("Hello World!", 100, hei/2);
```

We set the global font value and then create a new text element. The `fillText` method gets three parameters; the first is the text to be used while the other two are the x and y positions of the new element.

```
context.strokeStyle = "#ffffff";
context.strokeText("Hello World!", 100, hei/2);
```

In our example, we are giving our text drawing both a fill and an outline. The two functions are called separately. The `fillText` method is used to fill the content of the shape while the `strokeText` method is called to outline the text. We can use any one of them or both of the methods and they can get the exactly same parameters.

There's more...

There are some more options that you can explore.

Using gradients in your text

If you can do anything to any graphical element in canvas, you can do it to text as well—for example, in our sample we are using a gradient for our text.

```
grd = context.createLinearGradient(100, hei/2, 200, hei/2+110);
  grd.addColorStop(0, "#ffff00");
  grd.addColorStop(1, "#aaaa44");

  context.font = "50pt Verdana, sans-serif";
  context.fillStyle = grd;
```

Notice that we are updating our gradient. Our last gradient was too big for such a small text area. As such, we are drawing a line from around the start of our text going horizontally for 110 pixels.

Adding shadows and glows

You can add a shadow/glow to any filled element:

```
context.shadowOffsetX = 0;
  context.shadowOffsetY = 0;
  context.shadowBlur    = 8;
  context.shadowColor   = 'rgba(255, 255, 255, 0.5)';
  context.fillText("Hello World!", 100, hei/2);
```

You can control the position of the offset of the shadow. In our case, we want it to be a glow, so we placed our shadow exactly under our element. When setting the blur values to a shadow, try using values that are powers of 2 for efficiency (its easier to render values that are powers of 2).

Notice that when we defined our shadow color, we opted to use an RGBA as we wanted to set that alpha value to 50 percent.

Understanding pixel manipulation

Now that you have mastered drawing in canvas, it's time for us to turn to a new aspect of working with canvas. In canvas, you can manipulate pixels. It's not only a vector drawing tool, but a very smart pixel editor (raster).

Getting ready

Now that we are about to start reading data that is present on the canvas, we need to understand how security works when it comes to pixels. In an effort to protect content that isn't yours, there are security issues involved in working with data that isn't hosted on the same host as yours. We will not cover these security issues in this section and will be always working with images in the same domain as our code (or all locally).

Your first step is to find an image that you wish to work with (I've added an old image of my own into the source files). In this sample, we will recreate a pixel fade-out animation—really cool and useful for slides.

How to do it...

Let's get our code working and then break it down to see how it works. Perform the following steps:

1. Create a few helper global variables:

```
var context;
var imageData;
var pixelData;
var pixelLen;
var currentLocation=0;
var fadeOutImageInterval;
```

2. Create an `init` function (for the rest of the steps, all code will be in this function):

```
function init(){
  //all the rest of the code will go in here
}
```

3. Create a context variable for the 2D Canvas API:

```
function init(){
  var canvas = document.getElementById("textCanvas");
  var wid = canvas.width;
  var hei = canvas.height;

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

4. Create a new image:

```
var oIMG = new Image();
```

5. Add the `onload` listener logic:

```
oIMG.onload = function(){
  context.drawImage(this,
  0,0,this.width,this.height,0,0,wid,hei);
  imageData = context.getImageData(0, 0, wid, hei);
  pixelData = imageData.data;
  pixelLen = pixelData.length;
  fadeOutImageInterval = setInterval(fadeOutImage, 25);
};
```

6. Define the image source:

```
oIMG.src = "img/slide2.jpg";

} //end of init function
```

7. Create a new function called `fadeOutImage`. This image will transition our image in:

```
function fadeOutImage(){
  var pixelsChanged=0;
  for (var i = 0; i < pixelLen; i +=4) {
    if(pixelData[i]) {
      pixelData[i] =  pixelData[i]-1; // red
      pixelsChanged++;
    }
    if(pixelData[i + 1]){
      pixelData[i + 1] = pixelData[i+1]-1; // green
      pixelsChanged++;
    }
    if(pixelData[i + 2]){
      pixelData[i + 2] = pixelData[i+2]-1; // green
      pixelsChanged++;
    }

  }
  context.putImageData(imageData, 0, 0);

  if(pixelsChanged==0){
    clearInterval(fadeOutImageInterval);
    alert("we are done fading out");
  }
}
```

Your outcome should look something like the following screenshot:

How it works...

We will skip explaining things that we have already covered in earlier samples such as how to load images and how to work with the `drawImage` method (covered in the *Integrating images into our art* recipe discussed earlier in this chapter).

```
var context;
var imageData;
var pixelData;
var pixelLen;
var currentLocation=0;
var fadeOutImageInterval;
```

We will see the usage of these variables in our code, but all these variables have been saved as global variables so there is no need to redefine them in our functions. By defining these variables once, we improve the efficiency of our application.

The real new logic starts within the `onLoad` listener. Right after we draw our image onto the canvas, our new logic is added. It is highlighted in the following code snippet:

```
var oIMG = new Image();
  oIMG.onload = function(){
    context.drawImage(this,
    0,0,this.width,this.height,0,0,wid,hei);
    imageData = context.getImageData(0, 0, wid, hei);
    pixelData = imageData.data;
```

```
    pixelLen = pixelData.length;
    fadeOutImageInterval = setInterval(fadeOutImage, 25);
  };
  oIMG.src = "img/slide2.jpg";
```

We are now taking advantage of storing information in our canvas area and storing it globally. The first variable we are storing is `imageData`. This variable contains all the information of our canvas. We get this variable by calling the `context.getImageData` method.

```
context.getImageData(x, y, width, height);
```

The `getImageData` function returns every single pixel for a rectangular area. We need to set it by defining the area we want. In our case, we want the full canvas area as our image is set in the full canvas area.

The returned object (`imageData`) stores the direct access to the pixel data information in its data property (`imageData.data`) and this is our main focus while working directly with pixels. This object contains all the color information for each pixel in our canvas. The information is stored in four cells (red, green, blue, and alpha). In other words, if there are 100 pixels in total in our application, we would expect our array to contain 400 cells in the `imageData.data` array.

The last thing left to do before finishing the logic in our `onLoad` listener is to trigger our animation that will transition our image; to do that we will add an interval as follows:

```
fadeOutImageInterval = setInterval(fadeOutImage, 25);
```

Our animation is triggered after every 25 milliseconds until it's completed. The logic that fades our view happens within our `fadeOutImage` function.

Now that we have got all the prep work done, it's time to delve into the `fadeoutImage` function. Here, we will be doing the actual pixel manipulation logic. The first step in this function is to create a variable that will count how many changes our `imageData.data` array has made. When we hit the required number of changes, we terminate our interval (or in a real application maybe animate the next image):

```
var pixelsChanged=0;
```

We now start to run through all the pixels by using a `for` loop:

```
for (var i = 0; i < pixelLen; i +=4) {
  //pixel level logic will go in here
}
```

Each pixel stores RGBA values, thus, every pixel gets four positions in our array and as such we are jumping four steps at a time to move between pixels.

```
context.putImageData(imageData, 0, 0);
```

When we are done with manipulating our data, it's time for us to update our canvas. To do that we just need to send our new data back into our context. The second and third parameters are for the x and y starting point.

```
if(pixelsChanged==0){
  clearInterval(fadeOutImageInterval);
  alert("we are done fading out");
}
```

When we have no more changes (you can adjust that to fit your wishes such as when there are less than 100 pixels changed), we terminate the interval and trigger an alert.

In our `for` loop, we will lower the values of red, green, and blue until they get to 0. In our case, as we are counting changes we also add the counter into the loop:

```
for (var i = 0; i < pixelLen; i +=4) {
  if(pixelData[i]) {
    pixelData[i] =  pixelData[i]-1; // red
    pixelsChanged++;
  }
  if(pixelData[i + 1]){
    pixelData[i + 1] = pixelData[i+1]-1; // green
    pixelsChanged++;

  if(pixelData[i + 2]){
    pixelData[i + 2] = pixelData[i+2]-1; // blue
    pixelsChanged++;
  }

}
```

Earlier we mentioned that each pixel gets four cells of information in the array. The first three cells store the RGB values, while the fourth stores the alpha channel. As such I thought it would be important to notice that we are skipping position `i+3` as we don't want the alpha channel to get affected. Every element in the `pixelData` array is a value between `0` and `255`. In other words, if that pixel's value was `#ffffff` (white), all three RGB cells would be equal to `255`. By the way, it would take 255 calls to our function to get the values down to `0` as the value in the cells would start from `255` and go down by 1 each time.

We always skip the position `i+3`, as we don't want to change anything in our array. Our values are between `255` and `0`; in other words, if our image has a value `#ffffff` (totally white pixel), it would go down `255` times for our function to get `0`.

Making an image grayscale

To make an image or our canvas grayscale, we need to take all of our colors (red, green, blue) into account and mix them together. After mixing them together, get to a brightness value, which we can then apply to all the pixels. Let's see it in action:

```
function grayScaleImage(){
  for (var i = 0; i < pixelLen; i += 4) {
    var brightness = 0.33 * pixelData[i] + 0.33 * pixelData[i + 1]
    + 0.34 * pixelData[i + 2];
    pixelData[i] = brightness; // red
    pixelData[i + 1] = brightness; // green
    pixelData[i + 2] = brightness; // blue
  }
  context.putImageData(imageData, 0, 0);
}
```

In this case, we are taking the red (`pixelData[i]`), green (`pixelData[i+1]`), and blue (`pixelData[i+2]`), and using a one third of each to combine together to get one color and then we are assigning them all with this new averaged value.

Try only changing two out of the three values and see what comes out.

Pixel reversing

Color reversing an image is very easy as all we need to do is flip its value pixel by pixel by taking the maximum possible value (`255`) and subtracting the current value from it:

```
function colorReverseImage(){
  for (var i = 0; i < pixelLen; i += 4) {
    pixelData[i] = 255-pixelData[i];
    pixelData[i + 1] = 255-pixelData[i+1];
    pixelData[i + 2] = 255-pixelData[i+2];
  }
  context.putImageData(imageData, 0, 0);
}
```

There you go! We visited a few options of pixel manipulation, but the limit is really just up to your imagination. Experiment, you never know what might come out of it!