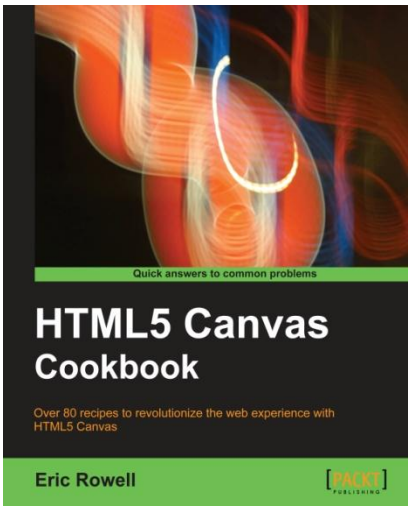
[PACKT] PUBLISHING

HTML5 Canvas Cookbook

Eric Rowell



Chapter No. 3
"Working with Images and Videos"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO. 3"Working with Images and Videos"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Eric Rowell is a professional frontend web developer and entrepreneur who is fascinated with the web industry, business, technology, and how they fit together. He's the founder and chief editor of `http://www.Html5CanvasTutorials.com`, an HTML5 canvas resource that's designed to complement the recipes in this book, and is also the creator of the KineticJS library, a lightweight JavaScript library that extends the 2D context by enabling canvas interactivity for desktop and mobile applications. When he's not building software, he loves spending time with his beautiful wife, Andie, and his spunky little dog, Koda. If you're feeling social, you can follow him on Twitter at `@ericdrowell`.

HTML5 Canvas Cookbook

The HTML5 canvas is revolutionizing graphics and visualizations on the Web. Powered by JavaScript, the HTML5 Canvas API enables web developers to create visualizations and animations right in the browser without Flash. Although the HTML5 Canvas is quickly becoming the standard for online graphics and interactivity, many developers fail to exercise all of the features that this powerful technology has to offer.

The **HTML5 Canvas Cookbook** begins by covering the basics of the HTML5 Canvas API and then progresses by providing advanced techniques for handling features not directly supported by the API such as animation and canvas interactivity. It winds up by providing detailed templates for a few of the most common HTML5 canvas applications—data visualization, game development, and 3D modeling. It will acquaint you with interesting topics such as fractals, animation, physics, color models, and matrix mathematics.

By the end of this book, you will have a solid understanding of the HTML5 canvas API and a toolbox of techniques for creating any type of HTML5 canvas application, limited only by the extent of your imagination.

What This Book Covers

Chapter 1, Getting Started with Paths and Text, begins by covering the basics of sub-path drawing and then moves on to more advanced path drawing techniques by exploring algorithms to draw zigzags and spirals. Next, the chapter dives into text drawing and then completes with an exploration of fractals.

Chapter 2, Shape Drawing and Composites, begins by covering the basics of shape drawing and also shows you how to use color fills, gradient fills, and patterns. Next, the chapter takes an in-depth look at transparencies and composite operations, and then provides recipes for drawing more complex shapes such as clouds, gears, flowers, card suits, and even a full vector-style jet complete with layers and shading.

Chapter 3, Working with Images and Videos, covers the basics of image and video handling, shows you how to copy-and-paste sections of the canvas, and covers different types of pixel manipulation. The chapter also shows you how to convert images into data URLs, save a canvas drawing as an image, and load a canvas with a data URL. Finally, the chapter ends with a pixilated image focus algorithm that can be used to focus and blur images dynamically with pixel manipulation.

Chapter 4, Mastering Transformations, explores what's possible with canvas transformations, including translations, scaling, rotations, mirror transforms, and free-form transformations. In addition, the chapter also explores the canvas state stack in detail.

Chapter 5, Bringing the Canvas to Life with Animation, begins by constructing an Animation class to handle an animation stage, and shows you how to create a linear motion, a quadratic motion, and an oscillating motion. Next, it covers some more complex animations such as the oscillation of a soap bubble, a swinging pendulum, and rotating mechanical gears. Finally, the chapter ends with a recipe for creating your own particle physics simulator, and also provides a recipe for creating hundreds of microscopic organisms inside the canvas to stress performance.

Chapter 6, Interacting with the Canvas: Attaching Event Listeners to Shapes and Regions, begins by constructing an Events class which extends the canvas API by providing a means for attaching event listeners to shapes and regions on the canvas. Next, the chapter covers techniques for getting the canvas mouse coordinates, detecting region events, detecting image events, detecting mobile touch events, and drag-and-drop. The chapter ends by providing a recipe for creating an image magnifier and another recipe for creating a drawing application.

Chapter 7, Creating Graphs and Charts, provides production-ready graph and chart classes, including a pie chart, a bar chart, an equation grapher, and a line chart.

Chapter 8, Saving the World with Game Development, gets you started with canvas game development by showing you how to create an entire side-scroller game called Canvas Hero. The chapter shows you how to create sprite sheets, create levels and boundary maps, create classes to handle the hero, the bad guys, the level, and the hero's health, and also shows you how to structure the game engine using an MVC (model view controller) design pattern.

Chapter 9, Introducing WebGL, begins by constructing a WebGL wrapper class to simplify the WebGL API. The chapter introduces WebGL by showing you how to create a 3D plane and a rotating cube, and also shows you how to add textures and lighting to your models. The chapter ends by showing you how to create an entire 3D world that you can explore in first person.

Appendices A, B, and C discuss other special topics such as canvas support detection, security, canvas vs. CSS3 transitions and animations, and the performance of canvas applications on mobile devices.

3
Working with Images and Videos

In this chapter, we will cover:

- Drawing an image
- Cropping an image
- Copying and pasting sections of the canvas
- Working with video
- Getting image data
- Introduction to pixel manipulation: inverting image colors
- Inverting video colors
- Converting image colors to grayscale
- Converting a canvas drawing into a data URL
- Saving a canvas drawing as an image
- Loading the canvas with a data URL
- Creating a pixelated image focus

Introduction

This chapter focuses on yet another very exciting topic of the HTML5 canvas, images and videos. Along with providing basic functionality for positioning, sizing, and cropping images and videos, the HTML5 canvas API also allows us to access and modify the color and transparency of each pixel for both mediums. Let's get started!

Drawing an image

Let's jump right in by drawing a simple image. In this recipe, we'll learn how to load an image and draw it somewhere on the canvas.

Follow these steps to draw an image in the center of the canvas:

How to do it...

1. Define the canvas context:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

2. Create an `image` object, set the `onload` property to a function that draws the image, and then set the source of the image:

```
    var imageObj = new Image();
    imageObj.onload = function(){
        var destX = canvas.width / 2 - this.width / 2;
        var destY = canvas.height / 2 - this.height / 2;

        context.drawImage(this, destX, destY);
    };
    imageObj.src = "jet_300x214.jpg";
};
```

3. Embed the canvas tag inside the body of the HTML document:

```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

To draw an image, we first need to create an `image` object using new `Image()`. Notice that we've set the `onload` property of the `image` object *before* defining the source of the image.

It's good practice to define what we want to do with the image when it loads *before* setting its source. Theoretically, if we were to define the source of the image before we define the `onload` property; the image could possibly load before the definition is complete (although, it's very unlikely).

The key method in this recipe is the `drawImage()` method:

```
context.drawImage(imageObj,destX,destY);
```

Where `imageObj` is the `image` object, and `destX` and `destY` is where we want to position the image.

There's more...

In addition to defining an image position with `destX` and `destY`, we can also add two additional parameters, `destWidth` and `destHeight` to define the size of our image:

```
context.drawImage(imageObj,destX,destY,destWidth,destHeight);
```

For the most part, it's a good idea to stay away from resizing an image with the `drawImage()` method, simply because the quality of the scaled image will be noticeably reduced, similar to the result when we resize an image with the width and height properties of an HTML image element. If image quality is something you're concerned about (why on earth wouldn't you be?), it's usually best to work with thumbnail images alongside bigger images if you're creating an application that needs scaled images. If, on the other hand, your application dynamically shrinks and expands images, using the `drawImage()` method with `destWidth` and `destHeight` to scale images is a perfectly acceptable approach.

Cropping an image

In this recipe, we'll crop out a section of an image and then draw the result onto the canvas.

Follow these steps to crop out a section of an image and draw the result onto the canvas.

How to do it...

1. Define the canvas context:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

2. Create an image object, set the `onload` property to a function that crops the image, and then set the source of the image:

```
    var imageObj = new Image();
    imageObj.onload = function(){
    // source rectangular area
        var sourceX = 550;
        var sourceY = 300;
        var sourceWidth = 300;
        var sourceHeight = 214;

    // destination image size and position
        var destWidth = sourceWidth;
        var destHeight = sourceHeight;
        var destX = canvas.width / 2 - destWidth / 2;
        var destY = canvas.height / 2 - destHeight / 2;

        context.drawImage(this, sourceX, sourceY, sourceWidth,
sourceHeight, destX, destY, destWidth, destHeight);
    };
    imageObj.src = "jet_1000x714.jpg";
};
```

3. Embed the canvas tag inside the body of the HTML document:

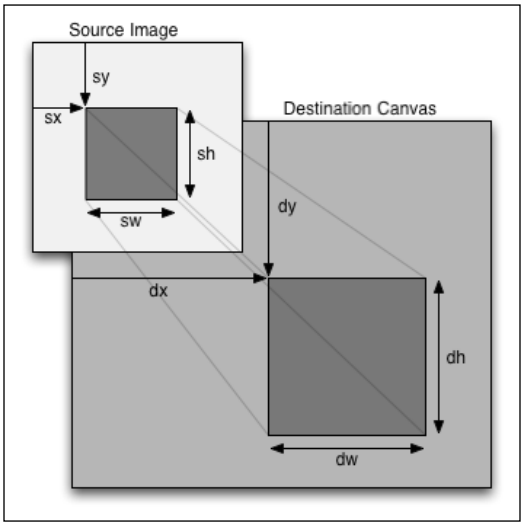
```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

In the last recipe, we discussed two different ways that we can use the `drawImage()` method to draw images on the canvas. In the first case, we can pass an `image` object and a position to simply draw an image at the given position. In the second case, we can pass an `image` object, a position, and a size to draw an image at the given position with the given size. Additionally, we can also add six more parameters to the `drawImage()` method if we wanted to crop an image:

```
Context.drawImage(imageObj,sourceX,sourceY,sourceWidth, sourceHight,
sourceHeight,sourceHeight, destX, destY, destWidth, destHeight);
```

Take a look at the following diagram:

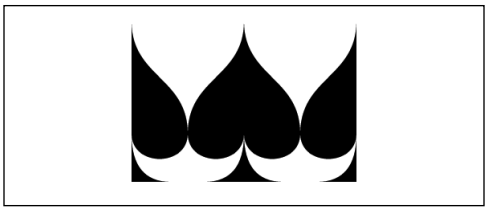


As you can see, `sourceX` and `sourceY` refer to the top-left corner of the cropped region in the source image. `sourceWidth` and `sourceHeight` refer to the width and height of the cropped image from the source. `destX` and `destY` refer to the position of the cropped image on the canvas, and `destWidth` and `destHeight` refer to the width and height of the resulting cropped image.

If you don't intend to scale a cropped image, then `destWidth` equals `sourceWidth` and `destHeight` equals `sourceHeight`.

Copying and pasting sections of the canvas

In this recipe, we'll cover yet another interesting usage of the `drawImage()` method—copying sections of the canvas. First, we'll draw a spade in the center of the canvas, then we'll copy the right side of the spade and then paste it to the left, and then we'll copy the left side of the spade and then paste it to the right.



Follow these steps to draw a spade in the center of the canvas and then copy-and-paste sections of the shape back onto the canvas:

How to do it...

1. Define the canvas context:

```
window.onload = function(){
    // drawing canvas and context
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

2. Draw a spade in the center of the canvas using the `drawSpade()` function that we created in *Chapter 2*, *Shape Drawing and Composites*:

```
    // draw spade
    var spadeX = canvas.width / 2;
    var spadeY = 20;
    var spadeWidth = 140;
    var spadeHeight = 200;

    // draw spade in center of canvas
    drawSpade(context, spadeX, spadeY, spadeWidth, spadeHeight);
```

3. Copy the right half of the spade and then paste it on the canvas to the left of the spade using the `drawImage()` method:

```
    context.drawImage(
    canvas,
    spadeX,          // source x
    spadeY,          // source y
    spadeWidth / 2,      // source width
```

```
    spadeHeight,       // source height
    spadeX - spadeWidth,  // dest x
    spadeY,          // dest y
    spadeWidth / 2,     // dest width
    spadeHeight        // dest height
  );
```

4. Copy the left half of the spade and then paste it on the canvas to the right of the spade using the `drawImage()` method:

```
    context.drawImage(
    canvas,
    spadeX - spadeWidth / 2,  // source x
    spadeY,           // source y
    spadeWidth / 2,       // source width
    spadeHeight,         // source height
    spadeX + spadeWidth / 2,   // dest x
    spadeY,           // dest y
    spadeWidth / 2,       // dest width
    spadeHeight          // dest height
  );
};
```

5. Embed the canvas inside the body of the HTML document:

```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

To copy a section of the canvas, we can pass the `canvas` object to the `drawImage()` method instead of an `image` object:

```
Context.drawImage(canvas,sourceX,sourceY,sourceWidth, sourceHight, sou
rceHeight,sourceHeight, destX, destY, destWidth, destHeight);
```

As we'll see in the next recipe, not only can we copy sections of an image or a canvas with `drawImage()`, we can also copy sections of HTML5 video.

Working with video

Although the HTML5 canvas API doesn't provide a direct method for drawing videos on the canvas like it does for images, we can certainly work with videos by capturing frames from a hidden video tag and then copying them onto the canvas with a loop.

Getting ready...

Before we get started, let's talk about the supported HTML5 video formats for each browser. At the time of writing, the video format war continues to rage on, in which all of the major browsers—Chrome, Firefox, Opera, Safari, and IE—continue to drop and add support for different video formats. To make things worse, each time a major browser adds or drops support for a particular video format, developers have to once again re-formulate the minimal set of video formats that's required for their applications to work across all browsers.

At the time of writing, the three major video formats are Ogg Theora, H.264, and WebM. For the video recipes in this chapter, we'll be using a combination of Ogg Theora and H.264. When working with video, it's strongly advised that you do a search online to see what the current status is for video support as it is a subject to change at any moment.

There's more! Once you've decided which video formats to support, you'll probably need a video format converter to convert the video file that you have on hand to other video formats. One great option for converting video formats is the Miro Video Converter, which supports video format conversions for just about any video format including Ogg Theora, H.264, or WebM formats.

Miro Video Converter is probably the most common video converter available at the time of writing, although you can certainly use any other video format converter of your liking. You can download Miro Video Converter from: `http://www.mirovideoconverter.com/`.

Follow these steps to draw a video onto the canvas:

How to do it...

1. Create a cross-browser method that requests an animation frame:

```
window.requestAnimFrame = (function(callback){
    return window.requestAnimationFrame ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame ||
    window.oRequestAnimationFrame ||
    window.msRequestAnimationFrame ||
    function(callback){
        window.setTimeout(callback, 1000 / 60);
    };
})();
```

2. Define the `drawFrame()` function which copies the current video frame, pastes it onto the canvas using the `drawImage()` method, and then requests a new animation frame to draw the next frame:

```
function drawFrame(context, video){
    context.drawImage(video, 0, 0);
    requestAnimFrame(function(){
        drawFrame(context, video);
    });
}
```

3. Define the canvas context, get the video tag, and draw the first video frame:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
    var video = document.getElementById("myVideo");
    drawFrame(context, video);
};
```

4. Embed the canvas and the video tag inside the body of the HTML document:

```
<video id="myVideo" autoplay="true" loop="true"
style="display:none;">
    <source src="BigBuckBunny_640x360.ogv" type="video/ogg"/
><source src="BigBuckBunny_640x360.mp4" type="video/mp4"/>
</video>
<canvas id="myCanvas" width="600" height="360" style="border:1px
solid black;">
</canvas>
```

How it works...

To draw a video on an HTML5 canvas, we first need to embed a hidden video tag in the HTML document. In this recipe, and in future video recipes, I've used the Ogg Theora and H.264 (mp4) video formats.

Next, when the page loads, we can use our cross-browser `requestAnimFrame()` method to capture the video frames as fast as the browser will allow and then draw them onto the canvas.

Getting image data

Now that we know how to draw images and videos, let's try accessing the image data to see what kind of properties we can play with.

WARNING: This recipe must run on a web server due to security constraints with the `getImageData()` method.

Getting ready...

Before we get started working with image data, it's important that we cover canvas security and the RGBA color space.

So why is canvas security important with respect to accessing image data? Simply put, in order to access image data, we need to use the `getImateData()` method of the canvas context which will throw a `SECURITY_ERR` exception if we try accessing image data from an image residing on a non-web server file system, or if we try accessing image data from an image on a different domain. In other words, if you're going to try out these demos for yourself, they won't work if your files reside on your local file system. You'll need to run the rest of the recipes in this chapter on a web server.

Next, since pixel manipulation is all about altering the RGB values of pixels, we should probably cover the RGB color model and the RGBA color space while we're at it. RGB represents the red, green, and blue components of a pixel's color. Each component is an integer between 0 and 255, where 0 represents no color and 255 represents full color. RGB values are often times represented as follows:

```
rgb(red,green,blue)
```

Here are some common color values represented with the RGB color model:

```
rgb(0,0,0) = black
rgb(255,255,255) = white
rgb(255,0,0) = red
rgb(0,255,0) = green
rgb(0,0,255) = blue
rgb(255,255,0) = yellow
rgb(255,0,255) = magenta
rgb(0,255,255) = cyan
```

In addition to RGB, pixels can also have an alpha channel which refers to its opacity. An alpha channel of 0 is a fully transparent pixel, and an alpha channel of 255 is a fully opaque pixel. RGBA color space simply refers to the RGB color model (RGB) plus the alpha channel (A).

Be careful not to confuse the alpha channel range of HTML5 canvas pixels, which are integers 0 to 255, and the alpha channel range of CSS colors, which are decimals 0.0 to 1.0.

Follow these steps to write out the image data properties of an image:

How to do it...

1. Define a canvas context:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

2. Create an `image` object, set the `onload` property to a function which draws the image:

```
    var imageObj = new Image();
    imageObj.onload = function(){
        var sourceWidth = this.width;
        var sourceHeight = this.height;
        var destX = canvas.width / 2 - sourceWidth / 2;
        var destY = canvas.height / 2 - sourceHeight / 2;
        var sourceX = destX;
```

```
            var sourceY = destY;

        // draw image on canvas
            context.drawImage(this, destX, destY);
```

3. Get the image data, write out its properties, and then set the source of the `image` object outside of the `onload` definition:

```
        // get image data from the rectangular area
        // of the canvas containing the image
            var imageData = context.getImageData(sourceX, sourceY,
sourceWidth, sourceHeight);
            var data = imageData.data;

        // write out the image data properties
            var str = "width=" + imageData.width + ", height=" +
imageData.height + ", data length=" + data.length;
            context.font = "12pt Calibri";
            context.fillText(str, 4, 14);
        };
        imageObj.src = "jet_300x214.jpg";
};
```

4. Embed the canvas tag into the body of the HTML document:

```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

The idea behind this recipe is to draw an image, get its image data, and then write out the image data properties to the screen. As you can see from the preceding code, we can get the image data using the `getImageData()` method of the canvas context:

```
context.getImageData(sourceX,sourceY,sourceWidth,sourceHeight);
```

Notice that the `getImageData()` method only works with the canvas context and not the `image` object itself. As a result, in order to get image data, we must first draw an image onto the canvas and then use `getImageData()` method of the canvas context.

The `ImageData` object contains three properties: `width`, `height`, and `data`. As you can see from the screenshot in the beginning of this recipe, our `ImageData` object contains a `width` property of 300, a `height` property of 214, and a `data` property which is an array of pixel information, which in this case has a length of 256,800 elements. The key to the `ImageData` object, in all honesty, is the `data` property. The `data` property contains the RGBA information for each pixel in our image. Since our image is made up of 300 * 214 = 64,200 pixels, the length of this array is 4 * 64,200 = 256,800 elements.

Introduction to pixel manipulation: inverting image colors

Now that we know how to access image data, including the RGBA for every pixel in an image or video, our next step is to explore the possibilities of pixel manipulation. In this recipe, we'll invert the colors of an image by inverting the color of each pixel.

WARNING: This recipe must be run on a web server due to security constraints with the `getImageData()` method.

Follow these steps to invert the colors of an image:

How to do it...

1. Define the canvas context:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

2. Create an `image` object and set the `onload` property to a function that draws the image and gets the image data:

```
    var imageObj = new Image();
    imageObj.onload = function(){
        var sourceWidth = this.width;
        var sourceHeight = this.height;
        var sourceX = canvas.width / 2 - sourceWidth / 2;
        var sourceY = canvas.height / 2 - sourceHeight / 2;
        var destX = sourceX;
        var destY = sourceY;
```

```
            context.drawImage(this, destX, destY);

            var imageData = context.getImageData(sourceX, sourceY,
sourceWidth, sourceHeight);
            var data = imageData.data;
```

3. Loop through all of the pixels in the image and invert the colors:

```
            for (var i = 0; i < data.length; i += 4) {
                data[i] = 255 - data[i]; // red
                data[i + 1] = 255 - data[i + 1]; // green
                data[i + 2] = 255 - data[i + 2]; // blue
                // i+3 is alpha (the fourth element)
            }
```

4. Overwrite the original image with the manipulated image, and then set the source of the image outside of the `onload` definition:

```
            // overwrite original image with
            // new image data
            context.putImageData(imageData, destX, destY);
        };
        imageObj.src = "jet_300x214.jpg";
};
```

5. Embed the canvas tag into the body of the HTML document:

```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

To invert the color of an image using HTML5 canvas, we can simply loop through all of the pixels in an image and then invert each pixel using a color inverting algorithm. Don't worry it's easier than it sounds. To invert a pixel's color, we can invert each of its RGB components by subtracting each value from 255 as follows:

```
data[i  ] = 255 - data[i  ]; // red
data[i+1] = 255 - data[i+1]; // green
data[i+2] = 255 - data[i+2]; // blue
```

Once the pixels have been updated, we can redraw the image using the `putImageData()` method of the canvas context:

```
context.putImageData(imageData, destX, destY);
```

This method basically allows us to draw an image using image data instead of a source image with the `drawImage()` method.

Inverting video colors

The purpose of this recipe is to demonstrate how to perform pixel manipulations on videos in much the same way as we did with images. In this recipe, we'll invert the colors of a short video clip.

WARNING: This recipe must be run on a web server due to security constraints with the `getImageData()` method.

Follow these steps to invert the colors of a video:

How to do it...

1. Create a cross-browser method that requests an animation frame:

```
window.requestAnimFrame = (function(callback){
    return window.requestAnimationFrame ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame ||
    window.oRequestAnimationFrame ||
    window.msRequestAnimationFrame ||
    function(callback){
        window.setTimeout(callback, 1000 / 60);
    };
})();
```

2. Define the `drawFrame()` function that captures the current video frame, inverts the colors, draws the frame on the canvas, and then requests a new animation frame:

```
function drawFrame(canvas, context, video){
    context.drawImage(video, 0, 0);

    var imageData = context.getImageData(0, 0, canvas.width,
canvas.height);
    var data = imageData.data;

    for (var i = 0; i < data.length; i += 4) {
        data[i] = 255 - data[i]; // red
        data[i + 1] = 255 - data[i + 1]; // green
        data[i + 2] = 255 - data[i + 2]; // blue
        // i+3 is alpha (the fourth element)
    }

    // overwrite original image
    context.putImageData(imageData, 0, 0);

    requestAnimFrame(function(){
        drawFrame(canvas, context, video);
    });
}
```

3. Define the canvas context, get the video tag, and draw the first animation frame:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
    var video = document.getElementById("myVideo");
    drawFrame(canvas, context, video);
};
```

4. Embed the video and canvas element into the body of the HTML document:

```
<video id="myVideo" autoplay="true" loop="true"
style="display:none;">
    <source src="BigBuckBunny_640x360.ogv" type="video/ogg"/
><source src="BigBuckBunny_640x360.mp4" type="video/mp4"/>
</video>
<canvas id="myCanvas" width="640" height="360" style="border:1px
solid black;">
</canvas>
```

How it works...

Similarly to the previous recipe, we can perform pixel manipulations on video in much the same way that we did with images because the `getImageData()` method gets the image data from the canvas context regardless of how the context was rendered. In this recipe, we can simply invert the color of each pixel on the canvas for each video frame provided by the `requestAnimFrame()` method.

Converting image colors to grayscale

In this recipe, we'll explore another common pixel manipulation algorithm, converting colors to grayscale.

WARNING: This recipe must be ran on a web server due to security constraints with the `getImageData()` method.

Follow these steps to convert the colors of an image to grayscale:

How to do it...

1. Define the canvas context:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

2. Create an `image` object and set the `onload` property to a function that draws the image and gets the image data:

```
    var imageObj = new Image();
    imageObj.onload = function(){
        var sourceWidth = this.width;
```

```
            var sourceHeight = this.height;
            var destX = canvas.width / 2 - sourceWidth / 2;
            var destY = canvas.height / 2 - sourceHeight / 2;
            var sourceX = destX;
            var sourceY = destY;

            context.drawImage(this, destX, destY);

            var imageData = context.getImageData(sourceX, sourceY,
sourceWidth, sourceHeight);
            var data = imageData.data;
```

3. Loop through the pixels in the image and convert the colors to grayscale using the equation for brightness:

```
            for (var i = 0; i < data.length; i += 4) {
                var brightness = 0.34 * data[i] + 0.5 * data[i + 1] +
0.16 * data[i + 2];

                data[i] = brightness; // red
                data[i + 1] = brightness; // green
                data[i + 2] = brightness; // blue
                // i+3 is alpha (the fourth element)
            }
```

4. Overwrite the original image with the manipulated image and then set the image source after the `onload` definition:

```
            // overwrite original image
            context.putImageData(imageData, destX, destY);
        };
        imageObj.src = "jet_300x214.jpg";
};
```

5. Embed the canvas element inside the body of the HTML document:

```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

To convert an RGB color into a gradation of gray, we need to obtain the brightness of the color. We can use the equation of brightness to obtain the grayscale value of a colored pixel. This equation is based on the fact that humans are most sensitive to green light, followed by red light, and are least sensitive to blue light:

```
Brightness = 0.34 * R + 0.5 * G + 0.16 * B
```

To account for physiological effects, notice that we've added more weight to the green value (most sensitive) followed by the red value (less sensitive) followed by the blue value (least sensitive).

With this equation in hand, we can simply loop through all of the pixels in our image, calculate the perceived brightness, assign this value to each of the RGB values, and then re-draw the image onto the canvas.

Converting a canvas drawing into a data URL

In addition to image data, we can also extract an image data URL which is basically just a very long text string containing encoded information about the canvas image. Data URLs are extremely handy if we want to save the canvas drawing in local storage or in an offline database. In this recipe, we'll draw a cloud shape, get its data URL, and then insert it into the HTML page so that we can see what it looks like.

Follow these steps to convert a canvas drawing into a data URL:

How to do it...

1. Define the canvas context and draw a cloud shape:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");

    var startX = 200;
    var startY = 100;

    // draw cloud shape
    context.beginPath();
    context.moveTo(startX, startY);
    context.bezierCurveTo(startX - 40, startY + 20, startX - 40,
startY + 70, startX + 60, startY + 70);
    context.bezierCurveTo(startX + 80, startY + 100, startX + 150,
startY + 100, startX + 170, startY + 70);
    context.bezierCurveTo(startX + 250, startY + 70, startX + 250,
startY + 40, startX + 220, startY + 20);
    context.bezierCurveTo(startX + 260, startY - 40, startX + 200,
startY - 50, startX + 170, startY - 30);
    context.bezierCurveTo(startX + 150, startY - 75, startX + 80,
startY - 60, startX + 80, startY - 30);
    context.bezierCurveTo(startX + 30, startY - 75, startX - 20,
startY - 60, startX, startY);
```

```
        context.closePath();

        context.lineWidth = 5;
        context.fillStyle = "#8ED6FF";
        context.fill();
        context.strokeStyle = "#0000ff";
        context.stroke();
```

2. Get the data URL of the canvas using the `toDataURL()` method of the `canvas` object:

```
        // save canvas image as data url (png format by default)
        var dataURL = canvas.toDataURL();
```

3. Insert the (long) data URL into a `<p>` tag so that we can see it:

```
        // insert url into the HTML document so we can see it
        document.getElementById("dataURL").innerHTML = "<b>dataURL:</
b> " + dataURL;
};
```

4. Embed the canvas tag inside the body of the HTML document and create a `<p>` tag which will be used to store the data URL:

```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
<p id="dataURL" style="width:600px;word-wrap: break-word;">
</p>
```

How it works...

The key to this recipe is the `toDataURL()` method which converts a canvas drawing into a data URL:

```
var dataURL = canvas.toDataURL();
```

When running this demo, you'll see a very long data URL that looks something like this:


```

AAAD6CAYAAAB9LTkQAAAgAElEQVR4Xu3dXbAUxd3H8f+5i09
VrEjuDlRFBSvoo1ETD/HmEcQIXskRc6FViaA+N7woRlNJUDQm4
kueeiS+INz4wEGfilwocLxSUASvDMf4XokpQbFKuAtYSdWT3PXz
/885C3t2Z3dndntme3q+W7Uehn2e7k/3sj96enpGhGCAcCCCCCAAAI
IIICAV4ERr6VRGAIIIIAAAggggIAQsBgECCCCCAAAIIIICAZwECl
mdQikMAAQQQQAABBAhYjAEEEEEAAAQQQQMCzAAHLMyjFIYAAAgg
ggAACBCzGAAIIIIAAAggg4FmAgOUZlOIQQAABBBBAAAECFmMAA
QQQQAABBBDwLEDA8gxKcQgggAACCCCAAAGLMYAAAAggggAACCHgWI
```

```
GB5BqU4BBBAAAEEEECAgMUYQAABBBBAAAEEPAsQsDyDUhwCCCCAA
AIIIEDAYgwggAACCCCAAAKeBQhYnkEpDgEEEEAAAQQQIGAxBhBAA
AEEEEAAAc8CBCzPoBSHAAIIIIAAAggQsBgDCCCAAAIIIICAZwECl
mdQikMAAQQQQAABBAhYjAEEEEAAAQQQQMCzAAHLMyjFIYAAAgggg
AACBCzGAAIIIIAAAggg4FmAgOUZlOIQQAABBBBAAAECFmMAAQQQQ
AABBBDwLEDA8gxKcQgggAACCCCAAAGLMYAAAggggAACCHgWIGB5
BqU4BBBAAAEEEECAgMUYQAABBBBAAAEEPAsQsDyDUhwCCCCAAAI
IIEDAYgwggAACCCCAAAKeBQhYnkEpDgEEEEAAAQQQIGAxBhBAAA
EEEEAAAc8CBCzPoBSHAAIIIIAAAggQsBgDCCCAAAIIIICAZwECl
mdQikMAAQQQQAABBAhYjAEEEEAAAQQQQMCzAAHLMyj
```

What you're looking at here is just a small snippet of the entire data URL. The important part to pay attention to in the URL is the very beginning, which starts with `data:image/png;base64`. This means that the data URL is a PNG image which is represented by a base 64 encoding.

Unlike image data, which is a native array of pixel data, an image data URL is special because it's a string that can be stored with local storage, or it can be passed to a web server to be saved in an offline database. In other words, image data is useful for inspecting and manipulating each individual pixel that makes up an image, while image data URLs are intended to be used for storing the canvas drawing and to be passed between the client and server.

Saving a canvas drawing as an image

In addition to saving the canvas drawing in local storage or in an offline database, we can also use an image data URL to save the canvas drawing as an image so that a user can then save it to their local computer. In this recipe, we'll get the image data URL of the canvas drawing and then set it to the source of an `image` object so that a user can right click and download the image as a PNG.

Follow these steps to save a canvas drawing as an image:

How to do it...

1. Define the canvas context and draw a cloud shape:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");

    // draw cloud
    context.beginPath(); // begin custom shape
    context.moveTo(170, 80);
    context.bezierCurveTo(130, 100, 130, 150, 230, 150);
```

```
    context.bezierCurveTo(250, 180, 320, 180, 340, 150);
    context.bezierCurveTo(420, 150, 420, 120, 390, 100);
    context.bezierCurveTo(430, 40, 370, 30, 340, 50);
    context.bezierCurveTo(320, 5, 250, 20, 250, 50);
    context.bezierCurveTo(200, 5, 150, 20, 170, 80);
    context.closePath(); // complete custom shape
    context.lineWidth = 5;
    context.fillStyle = "#8ED6FF";
    context.fill();
    context.strokeStyle = "#0000ff";
    context.stroke();
```

2. Get the data URL:

```
    // save canvas image as data url (png format by default)
    var dataURL = canvas.toDataURL();
```

3. Set the source of an image tag to the data URL so that a user can download it:

```
    // set canvasImg image src to dataURL
    // so it can be saved as an image
    document.getElementById("canvasImg").src = dataURL;
};
```

4. Embed the canvas tag in the body of the HTML document and add an image tag which will contain the canvas drawing:

```
<canvas id="myCanvas" width="578" height="200">
</canvas>
<p>
    Image:
</p>
<img id="canvasImg" alt="Right click to save me!">
```

How it works...

After drawing something on the canvas, we can create an image that the user can save by getting the image data URL using the `toDataURL()` method, and then setting the source of an `image` object to the data URL. Once the image has loaded (which is nearly instantaneous because the image is being loaded directly and doesn't have to make a request to a web server), the user can right click on the image to save it to their local computer.

Loading the canvas with a data URL

To load the canvas with a data URL, we can extend the previous recipe by creating an `image` object with the data URL and then drawing it on the canvas using our good friend `drawImage()`. In this recipe, we'll make a simple Ajax call to get the data URL from a text file and then use the URL to draw the image on the canvas. In the real world of course, you'll probably be fetching the image data URL from local storage or by calling a data service.

Follow these steps to load a canvas drawing with a data URL:

How to do it...

1. Define the `loadCanvas()` function which takes a data URL as input, defines a canvas context, creates a new image using the data URL, and then draws the image onto the canvas once it has loaded:

```
function loadCanvas(dataURL){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");

    // load image from data url
    var imageObj = new Image();
    imageObj.onload = function(){
        context.drawImage(this, 0, 0);
    };

    imageObj.src = dataURL;
}
```

2. Make an AJAX call to get a data URL stored on your server, and then call `loadCanvas()` with the response text when the response is received:

```
window.onload = function(){
    // make ajax call to get image data url
    var request = new XMLHttpRequest();
    request.open("GET", "dataURL.txt", true);
    request.onreadystatechange = function(){
        if (request.readyState == 4) {
            if (request.status == 200) { // successful response
                loadCanvas(request.responseText);
            }
        }
    };
    request.send(null);
};
```

3. Embed the canvas tag inside the body of the HTML document:

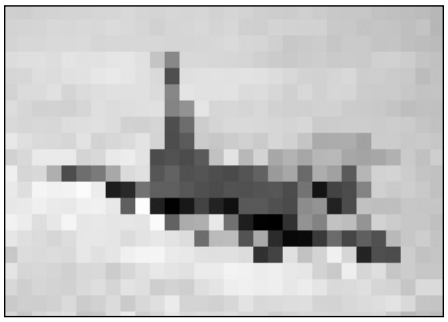
```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

To get the image data URL from a web server, we can set up an AJAX call (Asynchronous JavaScript and XML) to make a request to a web server and get the data URL as a response. When we get a status code of 200, which means that the request and response was successful, we can get the image data URL from `request.responseText`, and then pass it to the `loadCanvas()` function. This function will then create a new `image` object, set its source to the data URL, and then draw the image onto the canvas once it has loaded.

Creating a pixelated image focus

Looking for a fancy way to focus an image? How about a pixelated image focus? In this recipe, we'll explore the art of image pixelation by looping through an algorithm that pixelates an image less and less until it's completely focused.



WARNING: This recipe must be run on a web server due to security constraints with the `getImageData()` method.

Follow these steps to create a pixilation function that slowly focuses an image:

How to do it...

1. Define the `focusImage()` function which de-pixelates an image based on a pixilation value:

```
function focusImage(canvas, context, imageObj, pixelation){
    var sourceWidth = imageObj.width;
    var sourceHeight = imageObj.height;
    var sourceX = canvas.width / 2 - sourceWidth / 2;
    var sourceY = canvas.height / 2 - sourceHeight / 2;
    var destX = sourceX;
    var destY = sourceY;

    var imageData = context.getImageData(sourceX, sourceY,
sourceWidth, sourceHeight);
    var data = imageData.data;

    for (var y = 0; y < sourceHeight; y += pixelation) {
        for (var x = 0; x < sourceWidth; x += pixelation) {
            // get the color components of the sample pixel
            var red = data[((sourceWidth * y) + x) * 4];
            var green = data[((sourceWidth * y) + x) * 4 + 1];
            var blue = data[((sourceWidth * y) + x) * 4 + 2];

            // overwrite pixels in a square below and to
            // the right of the sample pixel, whos width and
            // height are equal to the pixelation amount
            for (var n = 0; n < pixelation; n++) {
                for (var m = 0; m < pixelation; m++) {
                    if (x + m < sourceWidth) {
                        data[((sourceWidth * (y + n)) + (x + m)) *
4] = red;
                        data[((sourceWidth * (y + n)) + (x + m)) *
4 + 1] = green;
                        data[((sourceWidth * (y + n)) + (x + m)) *
4 + 2] = blue;
                    }
                }
            }
        }
    }

    // overwrite original image
    context.putImageData(imageData, destX, destY);
}
```

2. Define the canvas context, fps value that determines how fast or slow the image focuses, the corresponding time interval, and the initial pixilation amount:

```
window.onload = function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
    var fps = 20; // frames / second
    var timeInterval = 1000 / fps; // milliseconds

    // define initial pixelation.  The higher the value,
    // the more pixelated the image is.  The image is
    // perfectly focused when pixelation = 1;
    var pixelation = 40;
```

3. Create a new `image` object, set the `onload` property to a function that creates a timed loop that calls the `focusImage()` function and decrements the pixilation value for each call until the image is focused, and then set the image source outside of the `onload` definition:

```
    var imageObj = new Image();
    imageObj.onload = function(){
        var sourceWidth = imageObj.width;
        var sourceHeight = imageObj.height;
        var destX = canvas.width / 2 - sourceWidth / 2;
        var destY = canvas.height / 2 - sourceHeight / 2;

        var intervalId = setInterval(function(){
            context.drawImage(imageObj, destX, destY);

            if (pixelation < 1) {
                clearInterval(intervalId);
            }
            else {
                focusImage(canvas, context, imageObj, pixelation--
);
            }
        }, timeInterval);
    };
    imageObj.src = "jet_300x214.jpg";
};
```

4. Embed the canvas tag into the body of the HTML document:

```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

How it works...

Before jumping into the pixelation algorithm, let's define pixelation. Pixelation of an image occurs when the human eye can detect the individual pixels that make up the image. Old school video game graphics and small images that have been enlarged are good examples of pixilation. In layman terms, if we define pixilation as a condition in which the pixels that make up the image are visible, this simply means that the pixels themselves are fairly large. In fact, the larger the pixels are, the more pixelated the image becomes. We can use this observation to create a pixilation algorithm.

To create an algorithm that pixelates an image, we can take color samples of the image and then draw oversized pixels in its place. As pixels need to be square, we can construct pixel sizes of 1 x 1 (standard pixel size), 2 x 2, 3 x 3, 4 x 4, and so on. The larger the pixels are, the more pixelated the image will look.

Until now, our recipes have simply looped through all of the pixels in the `data` property and converted them with a simple algorithm, without paying much attention to which pixels are being updated. In this recipe, however, we'll need to inspect sample pixels by looking at specific areas in the image based on x,y coordinates. We can use the following equations to pick out the RGBA components of a pixel based on the x, y coordinates:

```
var red = data[((sourceWidth * y) + x) * 4];
var green = data[((sourceWidth * y) + x) * 4 + 1];
var blue = data[((sourceWidth * y) + x) * 4 + 2];
```

With these equations in hand, we can use `setInterval()` to render a series of pixelated images over time, in which each successive pixelated image is less pixelated than the previous image, until the pixilation value equals 0 and the image is restored to its original state.

Where to buy this book

You can buy HTML5 Canvas Cookbook from the Packt Publishing website: `http://www.packtpub.com/html5-canvas-cookbook-recipes-to-revolutionize-web-experience/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/html5-canvas-cookbook-recipes-to-revolutionize-web-experience/book