

# 1

## Getting Started with Paths and Text

In this chapter, we will cover:

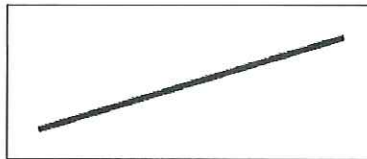
- ▶ Drawing a line
- ▶ Drawing an arc
- ▶ Drawing a Quadratic curve
- ▶ Drawing a Bezier curve
- ▶ Drawing a zigzag
- ▶ Drawing a spiral
- ▶ Working with text
- ▶ Drawing 3D text with shadows
- ▶ Unlocking the power of fractals: Drawing a haunted tree

### Introduction

This chapter is designed to demonstrate the fundamental capabilities of the HTML5 canvas by providing a series of progressively complex tasks. The HTML5 canvas API provides the basic tools necessary to draw and style different types of sub paths including lines, arcs, Quadratic curves, and Bezier curves, as well as a means for creating paths by connecting sub paths. The API also provides great support for text drawing with several styling properties. Let's get started!

## Drawing a line

When learning how to draw with the HTML5 canvas for the first time, most people are interested in drawing the most basic and rudimentary element of the canvas. This recipe will show you how to do just that by drawing a simple straight line.



### How to do it...

Follow these steps to draw a diagonal line:

1. Define a 2D canvas context and set the line style:

```
window.onload = function(){  
    // get the canvas DOM element by its ID  
    var canvas = document.getElementById("myCanvas");  
    // declare a 2-d context using the getContext() method of the  
    // canvas object  
    var context = canvas.getContext("2d");  
  
    // set the line width to 10 pixels  
    context.lineWidth = 10;  
    // set the line color to blue  
    context.strokeStyle = "blue";
```

2. Position the canvas context and draw the line:

```
    // position the drawing cursor  
    context.moveTo(50, canvas.height - 50);  
    // draw the line  
    context.lineTo(canvas.width - 50, 50);  
    // make the line visible with the stroke color  
    context.stroke();  
};
```

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>
```

### Downloading the example code



You can run the demos and download the resources for this book from [www.html5canvastutorials.com/cookbook](http://www.html5canvastutorials.com/cookbook) or 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.

### How it works...

As you can see from the preceding code, we need to wait for the page to load before trying to access the canvas tag by its ID. We can accomplish this with the `window.onload` initializer. Once the page loads, we can access the canvas DOM element with `document.getElementById()` and we can define a 2D canvas context by passing `2d` into the `getContext()` method of the canvas object. As we will see in the last two chapters, we can also define 3D contexts by passing in other contexts such as `webgl`, `experimental-webgl`, and others.

When drawing a particular element, such as a path, sub path, or shape, it's important to understand that styles can be set at any time, either before or after the element is drawn, but that the style must be applied immediately after the element is drawn for it to take effect. We can set the width of our line with the `lineWidth` property, and we can set the line color with the `strokeStyle` property. Think of this behavior like the steps that we would take if we were to draw something onto a piece of paper. Before we started to draw, we would choose a colored marker (`strokeStyle`) with a certain tip thickness (`lineWidth`).

Now that we have our marker in hand, so to speak, we can position it onto the canvas using the `moveTo()` method:

```
context.moveTo(x,y);
```

Think of the canvas context as a drawing cursor. The `moveTo()` method creates a new sub path for the given point. The coordinates in the top-left corner of the canvas are (0,0), and the coordinates in the bottom-right corner are (canvas width, canvas height).

Once we have positioned our drawing cursor, we can draw the line using the `lineTo()` method by defining the coordinates of the line's end point:

```
context.lineTo(x,y);
```

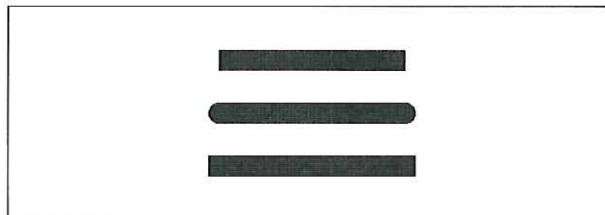
Finally, to make the line visible, we can use the `stroke()` method. Unless, otherwise specified, the default stroke color is black.

To summarize, here's the typical drawing procedure we should follow when drawing lines with the HTML5 canvas API:

1. Style your line (like choosing a colored marker with a specific tip thickness).
2. Position the canvas context using `moveTo()` (like placing the marker onto a piece of paper).
3. Draw the line with `lineTo()`.
4. Make the line visible using `stroke()`.

### There's more...

HTML5 canvas lines can also have one of three varying line caps, including **butt**, **round**, and **square**. The line cap style can be set using the `lineCap` property of the canvas context. Unless otherwise specified, the line cap style is defaulted to `butt`. The following diagram shows three lines, each with varying line cap styles. The top line is using the default `butt` line cap, the middle line is using the `round` line cap, and the bottom line is using a `square` line cap:



Notice that the middle and bottom lines are slightly longer than the top line, even though all of the line widths are equal. This is because the round line cap and the square line cap increase the length of a line by an amount equal to the width of the line. For example, if our line is 200 px long and 10 px wide, and we use a round or square line cap style, the resulting line will be 210 px long because each cap adds 5 px to the line length.

### See also...

- ▶ *Drawing a zigzag*
- ▶ *Putting it all together: Drawing a jet in Chapter 2*

## Drawing an arc

When drawing with the HTML5 canvas, it's sometimes necessary to draw perfect arcs. If you're interested in drawing happy rainbows, smiley faces, or diagrams, this recipe would be a good start for your endeavor.



### How to do it...

Follow these steps to draw a simple arc:

1. Define a 2D canvas context and set the arc style:

```
window.onload = function() {  
    var canvas = document.getElementById("myCanvas");  
    var context = canvas.getContext("2d");  
    context.lineWidth = 15;  
    context.strokeStyle = "black"; // line color
```

2. Draw the arc:

```
    context.arc(canvas.width / 2, canvas.height / 2 + 40, 80, 1.1 *  
    Math.PI, 1.9 * Math.PI, false);  
    context.stroke();  
};
```

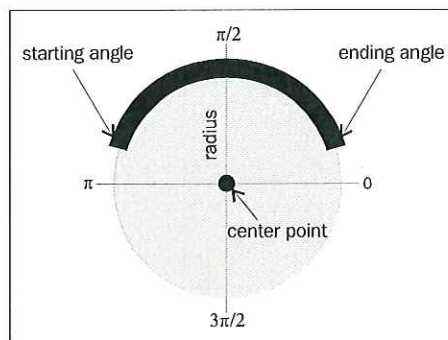
3. Embed the canvas tag inside the body of the HTML document:

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



## How it works...

We can create an HTML5 arc with the `arc()` method which is defined by a section of the circumference of an imaginary circle. Take a look at the following diagram:



The imaginary circle is defined by a center point and a radius. The circumference section is defined by a starting angle, an ending angle, and whether or not the arc is drawn counter-clockwise:

```
context.arc(centerX, centerY, radius, startingAngle,
            endingAngle, counterclockwise);
```

Notice that the angles start with  $0\pi$  at the right of the circle and move clockwise to  $3\pi/2$ ,  $\pi$ ,  $\pi/2$ , and then back to  $0$ . For this recipe, we've used  $1.1\pi$  as the starting angle and  $1.9\pi$  as the ending angle. This means that the starting angle is just slightly above center on the left side of the imaginary circle, and the ending angle is just slightly above center on the right side of the imaginary circle.

## There's more...

The values for the starting angle and the ending angle do not necessarily have to lie within  $0\pi$  and  $2\pi$ . In fact, the starting angle and ending angle can be any real number because the angles can overlap themselves as they travel around the circle.

For example, let's say that we define our starting angle as  $3\pi$ . This is equivalent to one full revolution around the circle ( $2\pi$ ) and another half revolution around the circle ( $1\pi$ ). In other words,  $3\pi$  is equivalent to  $1\pi$ . As another example,  $-3\pi$  is also equivalent to  $1\pi$  because the angle travels one and a half revolutions counter-clockwise around the circle, ending up at  $1\pi$ .

Another method for creating arcs with the HTML5 canvas is to make use of the `arcTo()` method. The resulting arc from the `arcTo()` method is defined by the context point, a control point, an ending point, and a radius:

```
context.arcTo(controlPointX1, controlPointY1, endingPointX,
              endingPointY, radius);
```

Unlike the `arc()` method, which positions an arc by its center point, the `arcTo()` method is dependent on the context point, similar to the `lineTo()` method. The `arcTo()` method is most commonly used when creating rounded corners for paths or shapes.

### See also...

- ▶ *Drawing a circle in Chapter 2*
- ▶ *Animating mechanical gears in Chapter 5*
- ▶ *Animating a clock in Chapter 5*

## Drawing a Quadratic curve

In this recipe, we'll learn how to draw a Quadratic curve. Quadratic curves provide much more flexibility and natural curvatures compared to its cousin, the arc, and are an excellent tool for creating custom shapes.



### How to do it...

Follow these steps to draw a Quadratic curve:

1. Define a 2D canvas context and set the curve style:

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

    context.lineWidth = 10;
    context.strokeStyle = "black"; // line color
```

2. Position the canvas context and draw the Quadratic curve:

```
context.moveTo(100, canvas.height - 50);
    context.quadraticCurveTo(canvas.width / 2, -50, canvas.width
- 100, canvas.height - 50);
    context.stroke();
};
```

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

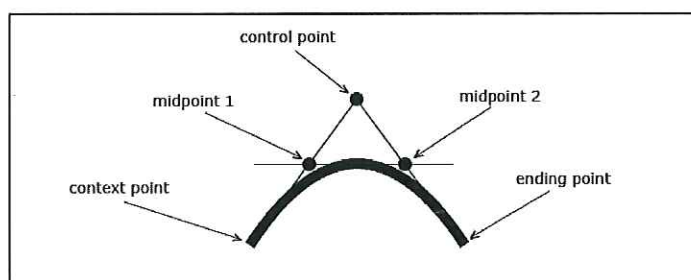
```
<canvas id="myCanvas" width="600" height="250" style="border:1px
solid black;">
</canvas>
```

### How it works...

HTML5 Quadratic curves are defined by the context point, a control point, and an ending point:

```
context.quadraticCurveTo(controlX, controlY, endPointX,
endPointY);
```

Take a look at the following diagram:



The curvature of a Quadratic curve is defined by three characteristic tangents. The first part of the curve is tangential to an imaginary line that starts with the context point and ends with the control point. The peak of the curve is tangential to an imaginary line that starts with midpoint 1 and ends with midpoint 2. Finally, the last part of the curve is tangential to an imaginary line that starts with the control point and ends with the ending point.

### See also...

- *Putting it all together: Drawing a jet*, in Chapter 2
- *Unlocking the power of fractals: Drawing a haunted tree*



## Drawing a Bezier curve

If Quadratic curves don't meet your needs, the Bezier curve might do the trick. Also known as cubic curves, the Bezier curve is the most advanced curvature available with the HTML5 canvas API.



### How to do it...

Follow these steps to draw an arbitrary Bezier curve:

1. Define a 2D canvas context and set the curve style:

```
window.onload = function() {  
    var canvas = document.getElementById("myCanvas");  
    var context = canvas.getContext("2d");  
  
    context.lineWidth = 10;  
    context.strokeStyle = "black"; // line color  
    context.moveTo(180, 130);
```

2. Position the canvas context and draw the Bezier curve:

```
    context.bezierCurveTo(150, 10, 420, 10, 420, 180);  
    context.stroke();  
};
```

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

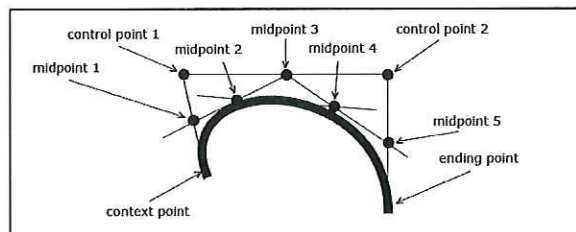
```
<canvas id="myCanvas" width="600" height="250" style="border:1px  
solid black;">  
</canvas>
```

## How it works...

HTML5 canvas Bezier curves are defined by the context point, two control points, and an ending point. The additional control point gives us much more control over its curvature compared to Quadratic curves:

```
context.bezierCurveTo(controlPointX1, controlPointY1,  
                      controlPointX2, controlPointY2,  
                      endingPointX, endingPointY);
```

Take a look at the following diagram:



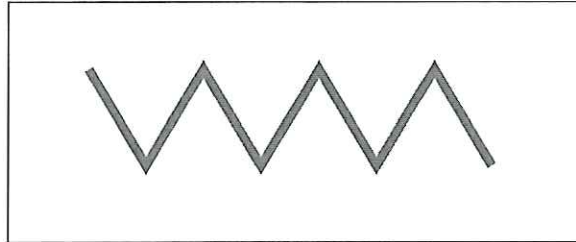
Unlike Quadratic curves, which are defined by three characteristic tangents, the Bezier curve is defined by five characteristic tangents. The first part of the curve is tangential to an imaginary line that starts with the context point and ends with the first control point. The next part of the curve is tangential to the imaginary line that starts with midpoint 1 and ends with midpoint 3. The peak of the curve is tangential to the imaginary line that starts with midpoint 2 and ends with midpoint 4. The fourth part of the curve is tangential to the imaginary line that starts with midpoint 3 and ends with midpoint 5. Finally, the last part of the curve is tangential to the imaginary line that starts with the second control point and ends with the ending point.

## See also...

- ▶ *Randomizing shape properties: Drawing a field of flowers in Chapter 2*
- ▶ *Putting it all together: Drawing a jet in Chapter 2*

## Drawing a zigzag

In this recipe, we'll introduce path drawing by iteratively connecting line subpaths to draw a zigzag path.



### How to do it...

Follow these steps to draw a zigzag path:

1. Define a 2D canvas context and initialize the zigzag parameters:

```
window.onload = function() {  
    var canvas = document.getElementById("myCanvas");  
    var context = canvas.getContext("2d");  
  
    var startX = 85;  
    var startY = 70;  
    var zigzagSpacing = 60;
```

2. Define the zigzag style and begin the path:

```
context.lineWidth = 10;  
context.strokeStyle = "#0096FF"; // blue-ish color  
context.beginPath();  
context.moveTo(startX, startY);
```

3. Draw seven connecting zigzag lines and then make the zigzag path visible with `stroke()`:

```
// draw seven lines  
for (var n = 0; n < 7; n++) {  
    var x = startX + ((n + 1) * zigzagSpacing);  
    var y;  
  
    if (n % 2 == 0) { // if n is even...  
        y = startY + 100;  
    }  
    else { // if n is odd...  
        y = startY;  
    }  
    context.lineTo(x, y);  
}
```

```
        context.lineTo(x, y);  
    }  
  
    context.stroke();  
};
```

4. 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 a zigzag, we can connect alternating diagonal lines to form a path. Programmatically, this can be achieved by setting up a loop that draws diagonal lines moving upwards and to the right on odd iterations, and downwards and to the right on even iterations.

The key thing to pay attention to in this recipe is the `beginPath()` method. This method essentially declares that a path is being drawn, such that the end of each line sub path defines the beginning of the next sub path. Without using the `beginPath()` method, we would have to tediously position the canvas context using `moveTo()` for each line segment while ensuring that the ending points of the previous line segment match the starting point of the current line segment. As we will see in the next chapter, the `beginPath()` method is also a required step for creating shapes.

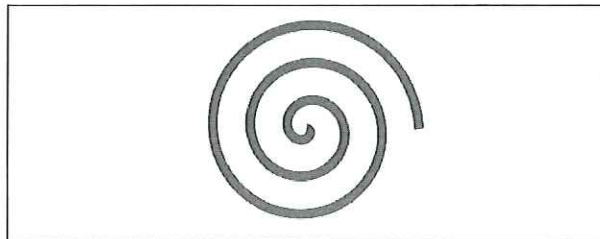
### Line join styles

Notice how the connection between each line segment comes to a sharp point. This is because the line join style of the HTML5 canvas path is defaulted to **miter**. Alternatively, we could also set the line join style to **round** or **bevel** with the `lineJoin` property of the canvas context.

If your line segments are fairly thin and don't connect at steep angles, it can be somewhat difficult to distinguish different line join styles. Typically, different line join styles are more noticeable when the path thickness exceeds 5 px and the angle between line sub paths is relatively small.

### Drawing a spiral

Caution, this recipe may induce hypnosis. In this recipe, we'll draw a spiral by connecting a series of short lines to form a spiral path.



### How to do it...

Follow these steps to draw a centered spiral:

1. Define a 2D canvas context and initialize the spiral parameters:

```
window.onload = function(){  
    var canvas = document.getElementById("myCanvas");  
    var context = canvas.getContext("2d");  
  
    var radius = 0;  
    var angle = 0;
```

2. Set the spiral style:

```
context.lineWidth = 10;  
context.strokeStyle = "#0096FF"; // blue-ish color  
context.beginPath();  
context.moveTo(canvas.width / 2, canvas.height / 2);
```

3. Rotate about the center of the canvas three times (50 iterations per full revolution) while increasing the radius by 0.75 for each iteration and draw a line segment to the current point from the previous point with `lineTo()`. Finally, make the spiral visible with `stroke()`:

```
for (var n = 0; n < 150; n++) {  
    radius += 0.75;  
    // make a complete circle every 50 iterations  
    angle += (Math.PI * 2) / 50;  
    var x = canvas.width / 2 + radius * Math.cos(angle);  
    var y = canvas.height / 2 + radius * Math.sin(angle);  
    context.lineTo(x, y);  
}  
  
context.stroke();  
};
```



4. 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 a spiral with HTML5 canvas, we can place our drawing cursor in the center of the canvas, iteratively increase the radius and angle about the center, and then draw a super short line from the previous point to the current point. Another way to think about it is to imagine yourself as a kid standing on a sidewalk with a piece of colored chalk. Bend down and put the chalk on the sidewalk, and then start turning in a circle (not too fast, though, unless you want to get dizzy and fall over). As you spin around, move the piece of chalk outward away from you. After a few revolutions, you'll have drawn a neat little spiral.

## Working with text

Almost all applications require some sort of text to effectively communicate something to the user. This recipe will show you how to draw a simple text string with an optimistic welcoming.



Hello World!

### How to do it...

Follow these steps to write text on the canvas:

1. Define a 2D canvas context and set the text style:

```
window.onload = function() {  
    var canvas = document.getElementById("myCanvas");  
    var context = canvas.getContext("2d");  
  
    context.font = "40pt Calibri";  
    context.fillStyle = "black";
```

2. Horizontally and vertically align the text, and then draw it:

```
// align text horizontally center
context.textAlign = "center";
// align text vertically center
context.textBaseline = "middle";
context.fillText("Hello World!", canvas.width / 2, 120);
};
```

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 text with the HTML5 canvas, we can define the font style and size with the `font` property, the font color with the `fillStyle` property, the horizontal text alignment with the `textAlign` property, and the vertical text alignment with the `textBaseline` property. The `textAlign` property can be set to left, center, or right, and the `textBaseline` property can be set to top, hanging, middle, alphabetic, ideographic, or bottom. Unless otherwise specified, the `textAlign` property is defaulted to left, and the `textBaseline` property is defaulted to alphabetic.

### There's more...

In addition to `fillText()`, the HTML5 canvas API also supports `strokeText()`:

```
context.strokeText("Hello World!", x, y);
```

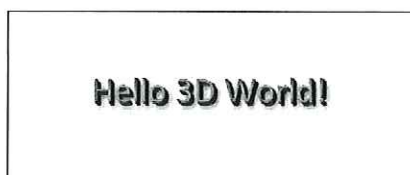
This method will color the perimeter of the text instead of filling it. To set both the fill and stroke for HTML canvas text, you can use both the `fillText()` and the `strokeText()` methods together. It's good practice to use the `fillText()` method before the `strokeText()` method in order to render the stroke thickness correctly.

### See also...

- ▶ *Drawing 3D text with shadows*
- ▶ *Creating a mirror transform in Chapter 4*
- ▶ *Drawing a simple logo and randomizing its position, rotation, and scale in Chapter 4*

## Drawing 3D text with shadows

If 2D text doesn't get you jazzed, you might consider drawing 3D text instead. Although the HTML5 canvas API doesn't directly provide us with a means for creating 3D text, we can certainly create a custom `draw3dText()` method using the existing API.



### How to do it...

Follow these steps to create 3D text:

1. Set the canvas context and the text style:

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

    context.font = "40pt Calibri";
    context.fillStyle = "black";
```

2. Align and draw the 3D text:

```
// align text horizontally center
context.textAlign = "center";
// align text vertically center
context.textBaseline = "middle";
draw3dText(context, "Hello 3D World!", canvas.width / 2, 120,
5);
};
```

3. Define the `draw3dText()` function that draws multiple text layers and adds a shadow:

```
function draw3dText(context, text, x, y, textDepth){
    var n;

    // draw bottom layers
```

```

    for (n = 0; n < textDepth; n++) {
        context.fillText(text, x - n, y - n);
    }

    // draw top layer with shadow casting over
    // bottom layers
    context.fillStyle = "#5E97FF";
    context.shadowColor = "black";
    context.shadowBlur = 10;
    context.shadowOffsetX = textDepth + 2;
    context.shadowOffsetY = textDepth + 2;
    context.fillText(text, x - n, y - n);
}

```

4. 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 3D text with the HTML5 canvas, we can stack multiple layers of the same text on top of one another to create the illusion of depth. In this recipe, we've set the text depth to five, which means that our custom `draw3dText()` method layers five instances of "Hello 3D World!" on top of one another. We can color these layers black to create the illusion of darkness beneath our text.

Next, we can add a colored top layer to portray a forward-facing surface. Finally, we can apply a soft shadow beneath the text by setting the `shadowColor`, `shadowBlur`, `shadowOffsetX`, and `shadowOffsetY` properties of the canvas context. As we'll see in later recipes, these properties aren't limited to text and can also be applied to sub paths, paths, and shapes.

## Unlocking the power of fractals: Drawing a haunted tree

First thing's first—what are fractals? If you don't already know, fractals are the awesome result when you mix mathematics with art, and can be found in all sorts of patterns that make up life. Algorithmically, a fractal is based on an equation that undergoes recursion. In this recipe, we'll create an organic-looking tree by drawing a trunk which forks into two branches, and then draw two more branches that stem from the branches we just drew. After twelve iterations, we'll end up with an elaborate, seemingly chaotic mesh of branches and twigs.



### How to do it...

Follow these steps to draw a tree using fractals:

1. Create a recursive function that draws a single branch that forks out into two branches, and then recursively calls itself to draw another two branches from the end points of the forked branches:

```
function drawBranches(context, startX, startY, trunkWidth, level){
  if (level < 12) {
    var changeX = 100 / (level + 1);
    var changeY = 200 / (level + 1);

    var topRightX = startX + Math.random() * changeX;
    var topRightY = startY - Math.random() * changeY;

    var topLeftX = startX - Math.random() * changeX;
    var topLeftY = startY - Math.random() * changeY;
```



```

        // draw right branch
        context.beginPath();
        context.moveTo(startX + trunkWidth / 4, startY);
        context.quadraticCurveTo(startX + trunkWidth / 4, startY
- trunkWidth, topRightX, topRightY);
        context.lineWidth = trunkWidth;
        context.lineCap = "round";
        context.stroke();

        // draw left branch
        context.beginPath();
        context.moveTo(startX - trunkWidth / 4, startY);
        context.quadraticCurveTo(startX - trunkWidth / 4, startY -
trunkWidth, topLeftX, topLeftY);
        context.lineWidth = trunkWidth;
        context.lineCap = "round";
        context.stroke();

        drawBranches(context, topRightX, topRightY, trunkWidth *
0.7, level + 1);
        drawBranches(context, topLeftX, topLeftY, trunkWidth *
0.7, level + 1);
    }
}

```

2. Initialize the canvas context and begin drawing the tree fractal by calling `drawBranches()`:

```

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

    drawBranches(context, canvas.width / 2, canvas.height, 50, 0);
};

```

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

```

<canvas id="myCanvas" width="600" height="500" style="border:1px
solid black;">
</canvas>

```

### How it works...

To create a tree using fractals, we need to design the recursive function that defines the mathematical nature of a tree. If you take a moment and study a tree (they are quite beautiful if you think about it), you'll notice that each branch forks into smaller branches. In turn, those branches fork into even smaller branches, and so on. This means that our recursive function should draw a single branch that forks into two branches, and then recursively calls itself to draw another two branches that stem from the two branches we just drew.

Now that we have a plan for creating our fractal, we can implement it using the HTML5 canvas API. The easiest way to draw a branch that forks into two branches is by drawing two Quadratic curves that bend outwards from one another.

If we were to use the exact same drawing procedure for each iteration, our tree would be perfectly symmetrical and quite uninteresting. To help make our tree look more natural, we can introduce random variables that offset the ending points of each branch.

### There's more...

The fun thing about this recipe is that every tree is different. If you code this one up for yourself and continuously refresh your browser, you'll see that every tree formation is completely unique. You might also be interested in tweaking the branch-drawing algorithm to create different kinds of trees, or even draw leaves at the tips of the smallest branches.

Some other great examples of fractals can be found in sea shells, snowflakes, feathers, plant life, crystals, mountains, rivers, and lightning.

# 2

## Shape Drawing and Composites

In this chapter, we will cover:

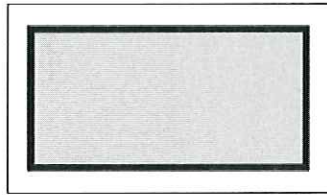
- ▶ Drawing a rectangle
- ▶ Drawing a circle
- ▶ Working with custom shapes and fill styles
- ▶ Fun with Bezier curves: drawing a cloud
- ▶ Drawing transparent shapes
- ▶ Working with the context state stack to save and restore styles
- ▶ Working with composite operations
- ▶ Creating patterns with loops: drawing a gear
- ▶ Randomizing shape properties: drawing a field of flowers
- ▶ Creating custom shape functions: playing card suits
- ▶ Putting it all together: drawing a jet

### Introduction

In *Chapter 1, Getting Started with Paths and Text*, we learned how to draw sub paths such as lines, arcs, Quadratic curves, and Bezier curves, and then we learned how to connect them together to form paths. In this chapter, we'll focus on basic and advanced shape drawing techniques such as drawing rectangles and circles, drawing custom shapes, filling shapes, working with composites, and drawing pictures. Let's get started!

## Drawing a rectangle

In this recipe, we'll learn how to draw the only built-in shape provided by the HTML5 canvas API, a rectangle. As unexciting as a rectangle might seem, many applications use them in one way or another, so you might as well get acquainted.



### How to do it...

Follow these steps to draw a simple rectangle centered on the canvas:

1. Define a 2D canvas context:

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

2. Draw a rectangle using the `rect()` method, set the color fill with the `fillStyle` property, and then fill the shape with the `fill()` method:

```
    context.rect(canvas.width / 2 - 100, canvas.height / 2 - 50,  
200, 100);  
    context.fillStyle = "#8ED6FF";  
    context.fill();  
    context.lineWidth = 5;  
    context.strokeStyle = "black";  
    context.stroke();  
};
```

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...

As you can see from the preceding code, we can draw a simple rectangle by using the `rect()` method:

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

The `rect()` method draws a rectangle at the position `x,y`, and defines its size with `width` and `height`. Another key thing to pay attention to in this recipe is the usage of `fillStyle` and `fill()`. Similar to `strokeStyle` and `stroke()`, we can assign a fill color using the `fillStyle` method and fill the shape using `fill()`.



Notice that we used `fill()` before `stroke()`. If we were to stroke a shape before filling it, the fill style would actually overlay half of the stroke style, effectively halving the line width style set with `lineWidth`. As a result, it's good practice to use `fill()` before using `stroke()`.

## There's more...

In addition to the `rect()` method, there are two additional methods that we can use to draw a rectangle and also apply styling with one line of code, the `fillRect()` method and the `strokeRect()` method.

### The `fillRect()` method

If we intend to fill a rectangle after drawing it with `rect()`, we might consider both drawing the rectangle and filling it with a single method using `fillRect()`:

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

The `fillRect()` method is equivalent to using the `rect()` method followed by `fill()`. When using this method, you'll need to define the fill style prior to calling it.

### The `strokeRect()` method

In addition to the `fillRect()` method, we can draw a rectangle and stroke it with a single method using the `strokeRect()` method:

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

The `strokeRect()` method is equivalent to using the `rect()` method followed by `stroke()`. Similar to `fillRect()`, you'll need to define the stroke style prior to calling this method.





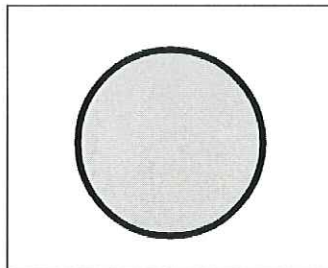
Unfortunately, the HTML5 canvas API does not support a method that both fills and strokes a rectangle. Personally, I like to use the `rect()` method and apply stroke styles and fills as needed using `stroke()` and `fill()` because it's more consistent with custom shape drawing. However, if you're wanting to apply both a stroke and fill to a rectangle while using one of these short-hand methods, it's good practice to use `fillRect()` followed by `stroke()`. If you were to use `strokeRect()` followed by `fill()`, you would overlay the stroke style by the fill, halving the stroke line width.

### See also...

- ▶ *Creating a linear motion in Chapter 5*
- ▶ *Detecting region events in Chapter 6*
- ▶ *Creating a bar chart in Chapter 7*

## Drawing a circle

Although the HTML5 canvas API doesn't support a circle method, we can certainly create one by drawing a fully enclosed arc.



### How to do it...

Follow these steps to draw a circle centered on the canvas:

1. Define a 2D canvas context:

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

2. Create a circle using the `arc()` method, set the color fill using the `fillStyle` property, and then fill the shape with the `fill()` method:

```
context.arc(canvas.width / 2, canvas.height / 2, 70, 0, 2 *  
Math.PI, false);  
context.fillStyle = "#8ED6FF";  
context.fill();  
context.lineWidth = 5;  
context.strokeStyle = "black";  
context.stroke();  
};
```

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...

As you might recall from Chapter 1, we can create an arc using the `arc()` method which draws a section of a circle defined by a starting angle and an ending angle. If, however, we define the difference between the starting angle and ending angle as 360 degrees ( $2\pi$ ), we will have effectively drawn a complete circle:

```
context.arc(centerX, centerY, radius, 0, 2 * Math.PI, false);
```

### See also...

- ▶ *Creating patterns with loops: drawing a gear*
- ▶ *Transforming a circle into an oval in Chapter 4*
- ▶ *Swinging a pendulum in Chapter 5*
- ▶ *Simulating particle physics in Chapter 5*
- ▶ *Animating a clock in Chapter 5*
- ▶ *Detecting region events in Chapter 6*
- ▶ *Creating a pie chart in Chapter 7*