THE EXPERT'S VOICE® IN WEB DEVELOPMENT

JavaScript Creativity

EXCITING, CREATIVE, AND INTERACTIVE JAVASCRIPT TECHNIQUES FOR YOUR PROJECTS

Shane Hudson
Foreword by Jeremy Keith

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

CHAPTER 1

Introduction

In this book, we will go on a journey down the rabbit hole of HTML5 and modern JavaScript, exploring a lot of different subjects as we go. We will start with canvas, which many of you will already be familiar with, and use it for two examples: a flocking animation and a coloring book app. Each chapter shows you how each topic can be used in the real world; how HTML5 and JavaScript can be used together to create useful applications and not just for making games or cool demos (although it will also help with those too). The book is split into three main projects, with some additional examples along the way. Every example (especially the main projects) aims to be a good starting point from which to play around and explore, because as useful as books are with their structured approach to learning, there is nothing better than just diving into the code yourself.

After canvas we will, in Chapter 3, delve into using audio and video on the web; this chapter primarily focuses on the Web Audio API, as well as both the audio and video HTML elements. The API is extremely powerful, so the chapter is only a brief introduction and is mostly about visualization of audio, which ties in nicely with Chapter 2, which uses canvas as the base for the graphics. We then move onto 3D graphics, exploring the basics of 3D on the web. I've decided to use the popular library Three.js for handling the WebGL, since pure WebGL "shader code" is rather complicated and very similar to the C language. There are multiple examples along the way in Chapter 4 but the main example is a 3D node graph that shows relationships between films and actors/actresses. I chose this example because data visualization is an extremely useful technique and one of the areas that I have a strong interest in, it is also a good way to show what these new technologies have allowed us to do natively on the web.

In Chapter 5, we will be going on a bit of a tangent into the realm of music theory because the first main project is going to be a music player that includes 3D visualization and music creation; this will nicely tie together all the chapters up until this point. The chapter builds heavily on the introduction to the Web Audio API in Chapter 3, however instead of visualization it generates sound (which can hopefully be used as a starting point for music). Of course, as with all the chapters, it is only a peek into a large discipline so I aim for it to be a good starting point rather than a comprehensive guide. The music theory aspect of Chapter 3 is one of the reasons why this book is called *JavaScript Creativity*; it is not about design or even focused on what you can do with canvas—it is about the creativity of applying web technologies to more than just a blog or an online store.

The music player project is put together in Chapter 6. I use Backbone as a way to bind together all the data, although you are welcome to convert it and instead use something else such as Ember or even just ES6.

The next project is comprised of Chapters 7 and 8. The project is a video chat application using WebRTC and Node.js. Chapter 7 will be a brief introduction to using Node.js for a text-based chat room and Chapter 8 will be a fairly in-depth look into how WebRTC works and how to use it. This will be a useful chapter for many because video chat is a common project that people want to make, especially within some businesses. The project also provides the introduction to getUserMedia that will be needed for the final project (Chapters 9 and 10) involving motion and object detection using the webcam.

Object detection is not a field that is particularly common within the web industry, but it has always been an interest of mine and we now have native capabilities for accessing the webcam (and even more complex techniques such as Web Workers, although I will not be using them in this book) so it makes sense that I would be doing computer vision on the web! The final project, involving object detection, is possibly less useful than the others but more interesting and hopefully it will be the perfect starting place for anybody interested in the subject as well as

showing how powerful the web platform really is. The majority of computer vision algorithms are math based and confusing, so I've gone for naïve algorithms (simple to understand but have many edge cases that are not catered for), as well as using a library for more advanced object detection so that you can have an introduction without being bombarded with academia.

The final project ties together all the chapters. It is a real time multi-user, computer-generated, gesture-controlled musical band. This is an ambitious project and is by no means perfect. It is a great way to practice everything in the book and is (I hope) a fun and unique project. The music generated will not be great, it will probably lag and not sound very good but I think it will be the perfect end to the book; after all, my main purpose of the book is to get you involved with projects that you may have not even thought about before!

What You Need to Know

Now that you know the journey we will be going on with this book, it is important to be able to start it. The book is being sold as requiring a working knowledge of JavaScript. This is quite true, but it is written in such a way that, although some of the code is quite complicated, it should be fairly accessible to anybody with some programming background.

■ **Note** This chapter will focus on the basics, namely debugging, so most of you will probably want to continue to chapter 2 for the introduction to canvas.

CSS

I just wanted to say that knowledge of CSS is expected, but not particularly needed. I use CSS in some chapters to style parts of a page but I tend to focus on the code throughout the book so nothing is polished design-wise and CSS use is minimal. You can get by fine without knowing any, but just be aware that if I do use some it will not be explained.

Debugging

Debugging is, at least in my opinion, a programmer's best skill because quite often code will not work as expected (especially across browsers) and even if it does it can always be improved. Most of the code in this book relies on cutting-edge technologies and many specs have changed even while I wrote the book—so if anything doesn't work, then it could be due to the implementation having changed since I wrote it or the browser you are using doesn't support it.

Browser Compatibility

The first thing to check is whether your browser is compatible with the features being used. This is something you should already be familiar with if you've ever used any modern technologies in production because many users still use browsers that do not support it. Of course, in this case you should be writing code in a "progressive enhancement" way so that the modern browsers get the modern features but it does not break the older browsers. However, throughout this book (because it is about cutting-edge technologies and features) I do not support older browsers and quite often at the time of writing the code only works in one or two browsers. Unless otherwise stated, I test all the code using Google Chrome 31. Most of it should also work in Mozilla Firefox.

The browsers change rapidly these days, in fact while I was writing the book Chrome forked Webkit into Blink and it has already affected specifications because they no longer always agree to go the same route. So what we need is a way to know when different versions of browsers change the implementation. Unfortunately, apart from looking through the change logs or bug tracker, there is no way to know if something has been changed. Luckily, we do know when features are added (which is enough for the majority of cases) due to two open source projects: `www.html5please.com` and `www.caniuse.com`. HTML5 Please is used to easily know if using a feature in production is recommended, and Can I Use is used to know in which browsers a feature is supported.

JavaScript Console

Every browser has different developer tools, some are similar and some quite different but one similarity that all browsers have is the JavaScript Console. This is where all errors are logged, but it is often much more powerful than just an error log. I discuss it from a Google Chrome viewpoint, but most browsers have very similar commands.

Access to variables

Often you will want to know a variable's value so that you can verify it is what you would expect or even just to learn how the data is structured (especially for objects). To do this, simply type the name of the variable into the JavaScript console and the value will be output below the name, this works for all types including objects and arrays. There are a few ways to access the variables via the console:

- After the code has run – If you are only interested in the final outcome, just do as I explained above and type the variable name.
- Breakpoint - I will discuss breakpoints in more detail shortly but as with debuggers in many other languages, you are able to add a breakpoint that allows you to examine the variables at the point that the code at the breakpoint is executed.
- Debugger Statements - You can manually add breakpoints in your code by writing `debugger;` in your code. It is generally easier to use breakpoints, but worth knowing about the debugger statement.
- Inside the code – You can log to the console from within the code. It is usually easier to use breakpoints but if you wish to you can log by writing `console.log('foo');` within the code.
- As a table – This is currently only available in Google Chrome and Firebug but it is a very useful way to view arrays as tabular data where each index is a row. By running `console.table(array);` from either within the code or within the console (I would recommend straight from console, during a breakpoint, since it is not compatible with other browsers).

Prompt

The area that you are able to type into within the console is known as the prompt, since it is used in much the same way as a shell (command line) prompt is used. This is very useful for accessing the console API, such as `console.log` and `console.table`. But it is also far more powerful, since it can evaluate expressions. These could be as basic as simple calculation (the answer gets outputted straight away) or something more specific to the page you're on such as modifying the DOM. In fact, you can write any JavaScript within the prompt, which makes it an invaluable tool for debugging since you can for example take a function that is not working as you would expect it to and modify it without affecting the rest of the script.

Sources

As with the console, most JavaScript debugging tools (within browsers and/or plugins) have a way to view the source code - which makes sense since you usually need to see the code to understand and debug it – but it is not just read-only. These tools allow you to manipulate and debug the code line by line. In Google Chrome, it is found under the 'Sources' tab, although most of the features are available in most other tools too.

Live editing

Sometimes you want to be able to write a line or code or test out a function, that is what the prompt is for, but sometimes you want to change the code itself a variety of times and to see how it does affect the rest of the codebase. Rather than using the prompt or repeatedly saving in your text editor, you can go into the source tab and directly edit the code. This will run it in a new instance of the JavaScript Virtual Machine so that you can test and debug all you need to without overwriting the original file (which would then need to be refreshed, and you may lose useful code et cetera).

Breakpoints

You are likely already familiar with basic breakpoints, either from using JS developer tools or a previous language. It is one of the most useful tools in a developer's toolkit. While in code view (sources tab in Chrome) you can add a breakpoint by clicking the line count, which leaves a marker (the breakpoint). As I explained earlier, this default type of breakpoint pauses the JavaScript and allows you to see the current state, including the values of the variables. There are a few other types of breakpoints and not all debuggers can deal with all types of breakpoints, which can be useful in different situations.

- **DOM breakpoints:** These let you pause the code for changes such as modification of an attribute or removal of an element. In Chrome this can be found by right-clicking a DOM none and selecting one of the options listed under Break on….
- **Exceptions:** Quite often you will have code that is set up to throw an exception but rather than handling it in the code, you will want to debug it properly and figure out what caused the exception. Instead of having breakpoints on every exception, which soon gets annoying, you can set the debugger to pause on either all exceptions or just uncaught exceptions. In Chrome this is accessed by clicking the pause button that has a hexagonal background. The color changes dependent on which mode it is set to.
- **Events:** Breakpoints can be set for when an event gets triggered, such as click or mousemove, which can be very useful for making sure events are triggered where they should be and that they do as expected. In Chrome, these are available from the sources panel under the Event Listener Breakpoints heading.
- **Conditional breakpoints:** These are the same as the default breakpoint except that you can set conditions on which the breakpoint pauses, which work the same way as if you put a breakpoint within an `if` statement. These are, in my opinion, the most useful kind of breakpoint because you can use them to only pause the code when the results are not what you would expect them to be. To place a conditional breakpoint, right-click the line count and it will give you a text field to type the condition as shown in Figure 1-1.

Figure 1-1. *Adding a conditional breakpoint*

Timeline

So far we have discussed functionality in debuggers that allow you to fix code and make sure it is working as you would expect it to. Sometimes however, getting the code to work is the easy bit—making it fast is often far harder. We are still on the subject of debuggers but now rather than inspecting to make sure each part of the code is working, we instead need to record the entire process so that we can spot patterns in performance (looking at the overall running of the code rather than each individual function/line). This is a subject that could easily fill an entire book, so I will go over the basics and it should give you a good starting point for figuring out how to understand and improve the performance.

In Chrome, the tool that we use to measure performance of the code is the timeline. I should note that the network pane is also related, as it lets us see which resources are causing a delay. But I'm going to focus on the running of the code rather than the resources, because it is more likely that you have experience with resources (on a normal website) than with performance testing of the code (crucial for the more creative of web apps).

To start using the timeline, you need to record the running of the code by clicking the circle button. As soon as it starts recording you will notice that some graphs are created. There are (at the time of writing) three aspects that are recorded: Events, Frames, and Memory; all these are recorded simultaneously but can only be viewed separately by selecting them. At the basic level, these features give you a good way to test the speed of parts of your code because it breaks down the code like a stack, showing you each function call. This is a fairly easy way to find bottlenecks, but it only really scratches the surface of what the timeline can do.

Rather than just showing the amount of time the function takes, it also splits it into loading, scripting, rendering, and painting. These are extremely useful for knowing why a function is taking longer than it should. The loading category is where any XHR (AJAX) calls are made as well as parsing the HTML. The logic within the function is included in the scripting category. Rendering is where the DOM manipulation takes place, primarily recalculating styles and layout. Paint is where the browser updates what is being shown to match the outcome of the rendering, including any changes on canvas and other elements that are not directly DOM related.

In Chapter 2, we use canvas for a simulation of birds flocking (known as Boids) and so I have recorded that using the timeline to show you how it works. Figure 1-2 shows the events timeline, useful for timing functions and visually seeing the stack that they produce. There are only two different colors shown in the screenshots. This is because there are very few DOM elements due to using a canvas and so there is no loading and rendering required. The lack of DOM is also the same reason for the patterns within the timeline, if we were recording performance of a regular website, then we might see changes on scrolling or hovering over a button; so the timeline is definitely powerful for more than canvas animations! Figure 1-3 shows the frames timeline. This measures the Frames Per Second (FPS) of the page because a low FPS can make a site or animation look "janky" instead of the smooth experience that people expect. The third screenshot of this section, Figure 1-4, shows the memory timeline; you can see from the spikes that the memory stores more and more until the garbage collector comes along and clears the memory that is storing old data. You will notice that there is quite a lot of whitespace below the timeline. This space is for a graph showing the amount of DOM nodes and event handlers (as you can see in Figure 1-5, which shows the timeline for a basic portfolio website I once made).

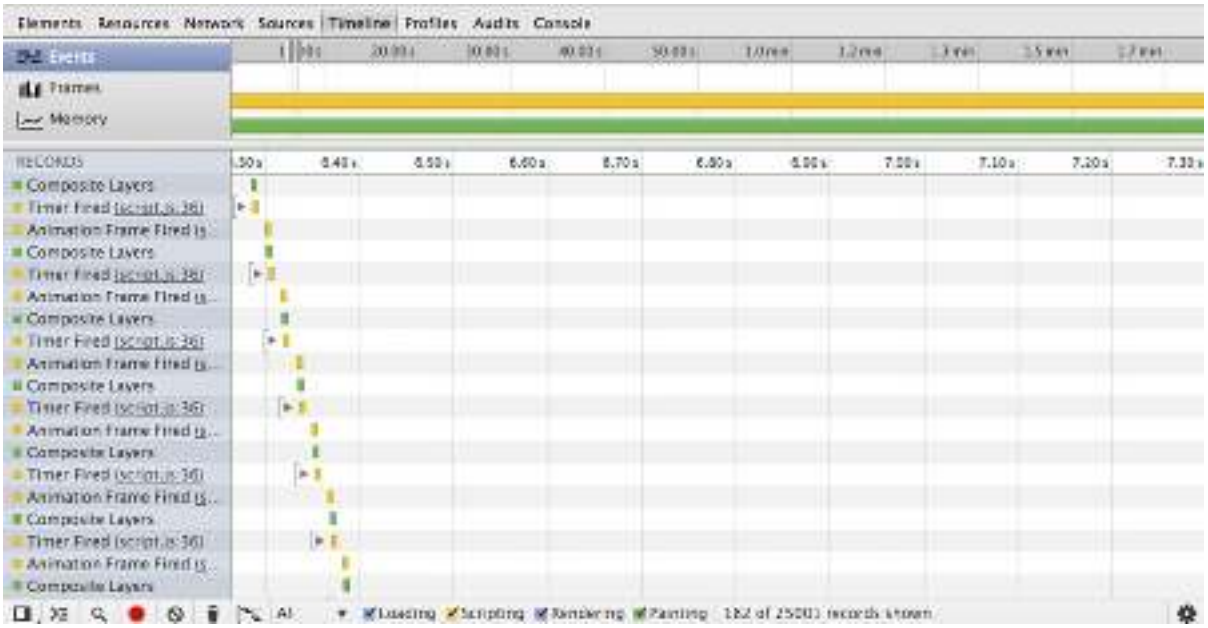

Figure 1-2. *Showing Chrome Developer Tools' events timeline*

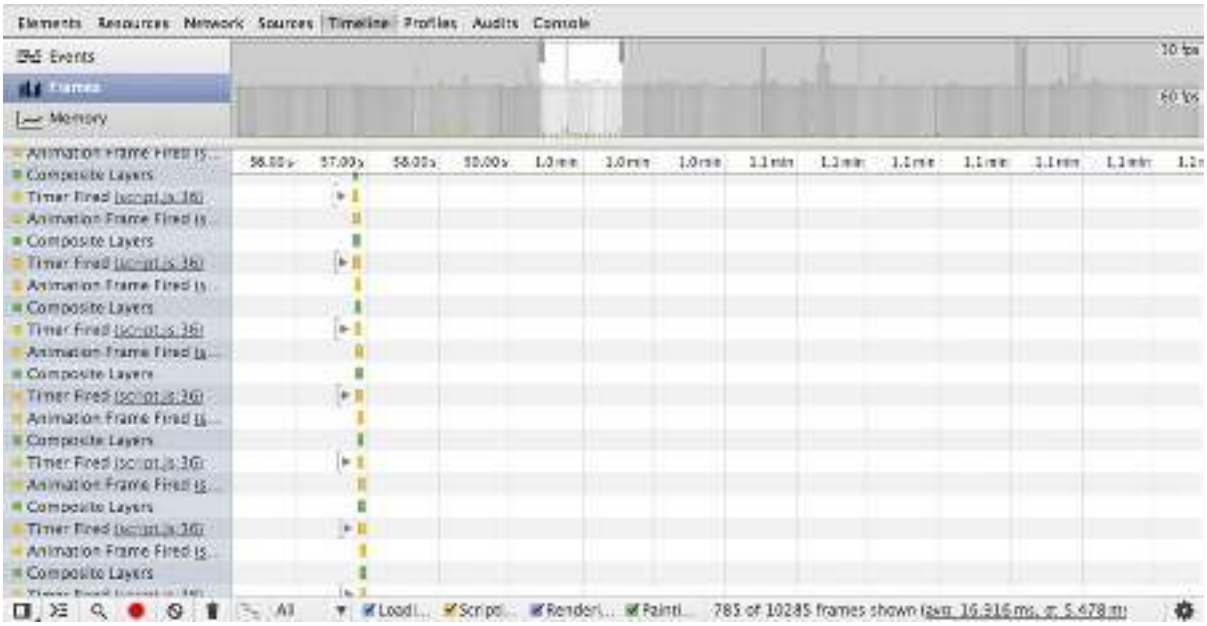

Figure 1-3. *Showing Chrome Developer Tools' frames timeline*

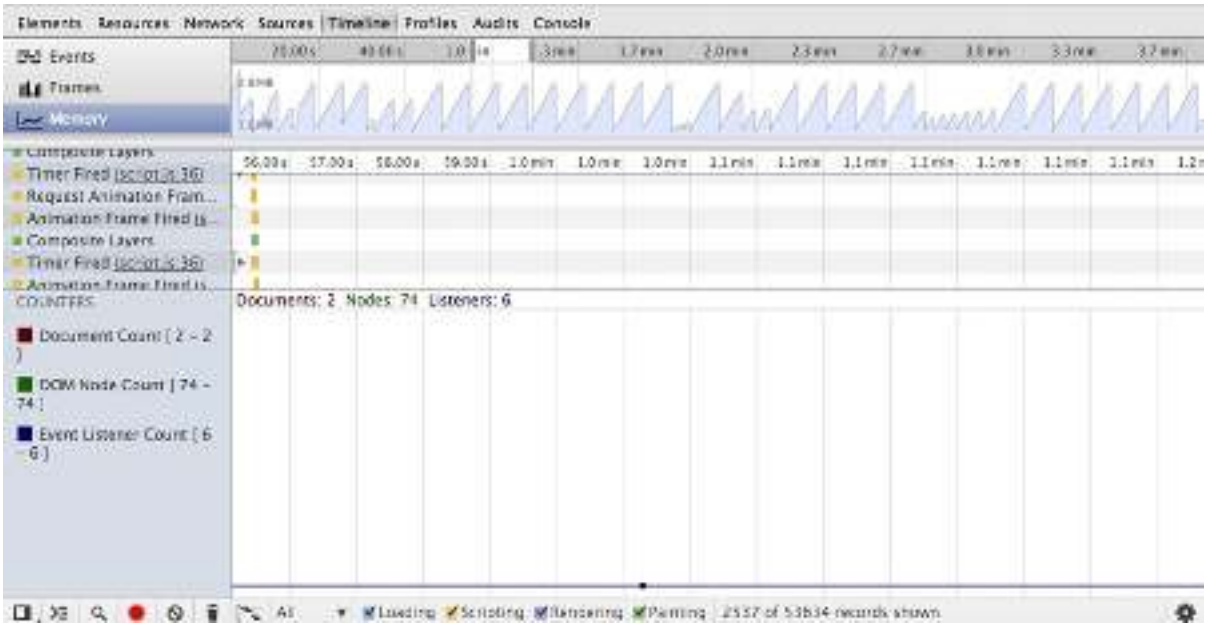

Figure 1-4. *Showing Chrome Developer Tools' memory timeline*

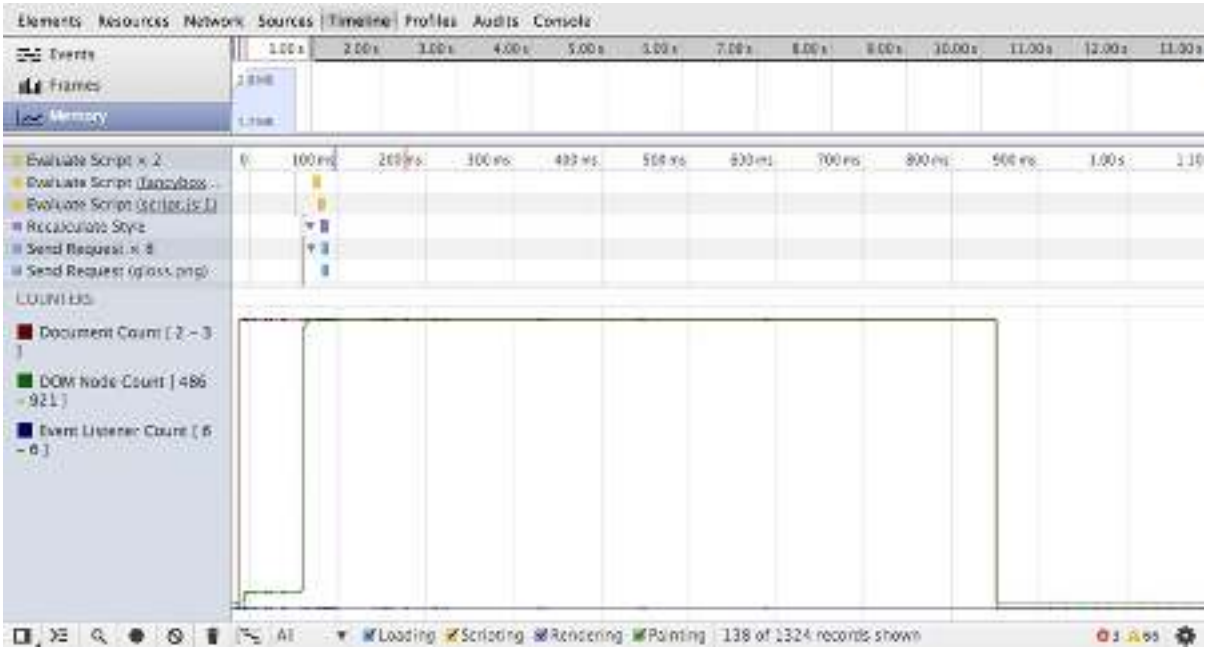

Figure 1-5. *A memory timeline showing DOM elements rendering on a portfolio website*

Canvas Profiles

The timeline is an incredibly powerful tool, but when it comes to debugging a canvas animation it can often feel like guess work (because most animations are scripted to be dynamic, we don't really know what each function is actually doing). Luckily, there is a brand new tool in Chrome Developer Tools that lets you capture a canvas frame (at the time of writing it is still only in the Canary version of Chrome). It is available under the Profiles tab (you may need to enable the experiment under the developer tools' settings).

A captured frame profile is very similar to a breakpoint, except instead of pausing the code it records a frame and allows you to step through each call that is made. This is incredibly powerful because it lets you see the exact order of execution and shows the state of the canvas at that particular step, as shown in Figure 1-6.

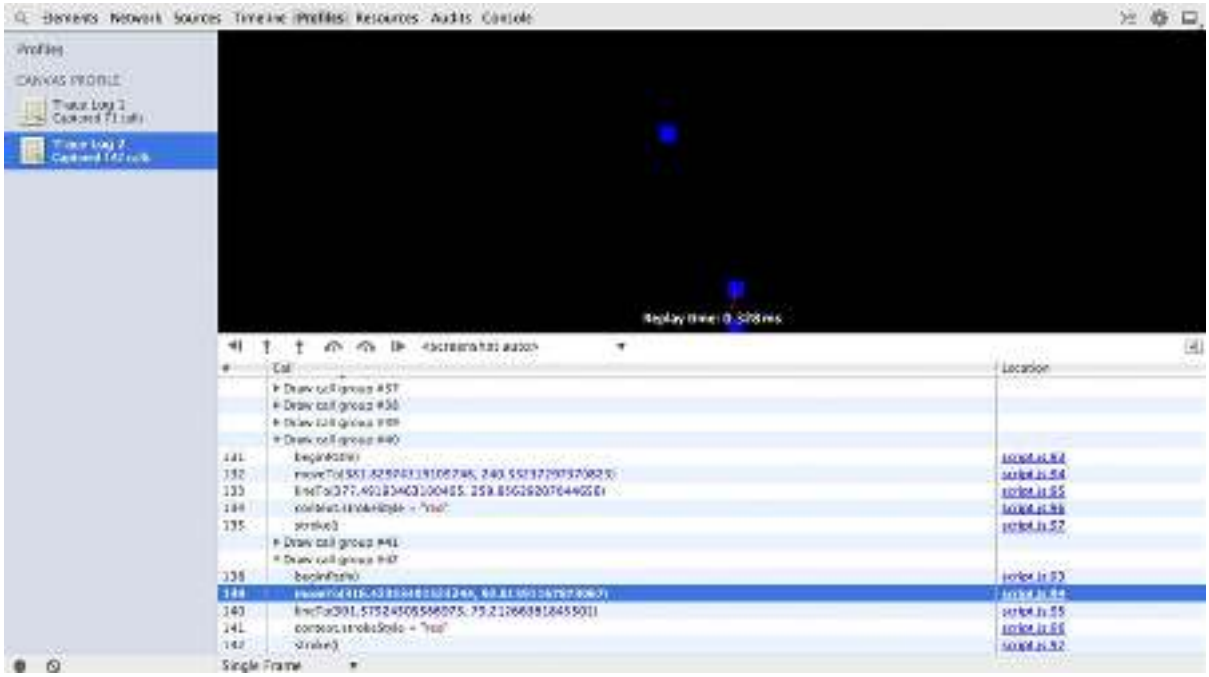

Figure 1-6. *Capturing a canvas frame*

Summary

This opening chapter has hopefully prepared you for the rest of the book. I am a strong believer that you cannot learn by purely reading a book, so I have written in such a way that I hope to guide you to different techniques and ways of learning about each subject rather than writing a ”how to” book. So the debugging section of this chapter should be very useful to you throughout the rest of the book because debugging isn’t just about fixing broken code, it is about exploring how the code actually works and how you can improve it.

CHAPTER 2

Canvas and Animation Basics

In the world of interactivity, a simple animation can be both a great introduction to learning as well as effective for conveying messages to the user. Many people believe that all interactive features on websites are games, but this just is not true! I hope that from this chapter you will gain an insight of how you can do something a little bit differently–how to stand out from the standard websites with professional animations.

There are many ways we can add animation these days, with the ever-more impressive HTML and CSS technologies, including: transforms, canvas, and SVG. I focus on canvas as it is used heavily throughout the book, though you should feel comfortable using all three and understand the pros and cons of each. In brief, you can think of canvas as raster and SVG as vector, with transforms used to manipulate DOM elements.

What Is Canvas?

Canvas is well named, as it can be likened to a painter's canvas because it is empty until you start painting. On its own, canvas is no different than a div–it is a blank non-semantic block element. The difference however is that canvas has a *context* that can be used for drawing on with JavaScript, this context can be extracted using `var ctx = canvas.getContext('2d');` where canvas is a variable containing reference to a canvas element. You may have noticed that I have specified a 2d context, `getContext` allows you to choose between multiple contexts, currently the 2d context is the most common and most well developed; however WebGL and other technologies can be used for a 3d context.

Because canvas is not supported in legacy browsers, it is possible to declare a fallback so that other content can be shown instead–such as a static image or a flash animation. To add a fallback you simply just write html between the canvas tags. I would recommend you avoid writing "Canvas is not supported in your browser" because the experience of a website should be tailored to the capabilities of the device/browser rather than to criticize the user for their browsing habits; however for simplicity's sake I will be breaking that rule for many exercises in this book.

```
<canvas id="animation">
    <img src="images/canvas-fallback.png">
</canvas>
```

Once you have a context, you can start drawing. All drawing features of the context use the Cartesian grid system in which you have both the x and y axes, it should be noted that–as with most computer-based graphics–the context uses the bottom-right quadrant, so the top-left corner is its origin (0,0).

```
var ctx = canvas.getContext('2d');
//fillRect(x, y, width, height);
ctx.fillRect(10, 10, 50, 50);
```

The preceding code paints a 50px × 50px square at 10px away from the top-left corner on each axis. Do note that any four-sided shapes are called rectangles in canvas. By default this rectangle is black. Because the drawing onto context is procedural, it is required that we change color before we draw the object. After we have changed the color, all objects will use the same color until it is changed again.

```
// This sets the fill color to red
ctx.fillStyle = "#ff0000";

// fillRectangle(x, y, width, height);
ctx.fillRect(10, 10, 50, 50);
```

As you can see in Figure 2-1, the preceding code simply draws the same rectangle in red.

Figure 2-1. *A simple red rectangle*

So now, since we are concerned with animation, let's get this square moving!

RequestAnimationFrame

If you have ever needed repetitive tasks in JavaScript, such as polling, then you will probably have experience with `setInterval` and `setTimeout`. Both of these functions are unreliable because they can trigger at anytime after the timeout rather than on it, which can cause trouble if you need them to be triggered immediately (something that is quite important for smooth animations). Also, many browsers have a minimum timer resolution of about 16ms that can cause delays especially when not expected. Another problem with using timeouts is that you need to set the timeout manually; so any testing will be specific to that computer. On the other hand, `requestAnimationFrame` (rAF) can be used to trigger at the best time for the computer it is running on. Rather than a set time, it runs up to 60 frames per second but fires less frequently (such as when the browser tab is in the background). A further optimization that can be used is the second parameter, which can be a DOM element to constrain the rAF.

Due to a lack of browser support of `requestAnimationFrame` in some browsers at the time of writing, it is recommended to use a shim so that it works across browsers that implement a version of it. I will not be including this in the code examples, but you should take it as required until all browsers support the standards correctly.

```
// Shim for RequestAnimationFrame
(function() {
  var requestAnimationFrame = window.requestAnimationFrame || window.mozRequestAnimationFrame ||
                              window.webkitRequestAnimationFrame || window.msRequestAnimationFrame;
  window.requestAnimationFrame = requestAnimationFrame;
})();
```

I find that it is important to separate code, so while I will not be using any particular design patterns yet (being small amounts of code), I encourage you to organize your code appropriately. For this section I think `logic()` and `draw()` are appropriate names for functions.

```
// This is a way of selecting
var ele = document.querySelector("#animation");
var ctx = ele.getContext('2d');
var x = y = 10;
var width = height = 50;

function logic () {
    x += 10;
    if (x < ele.width - width) requestAnimationFrame(draw);
}

function draw()  {
    ctx.clearRect(0, 0, ele.width, ele.height);

    // This sets the fill colour to red
    ctx.fillStyle = "#ff0000";

    // fillRectangle(x, y, width, height);
    ctx.fillRect(x, y, 50, 50);
}

requestAnimationFrame(draw);
setInterval(logic, 1000/60);
```

You will notice that I have used both `requestAnimationFrame` and `setInterval`, this is due to the fact that you rarely want the animation to be running at full speed (dependent on the computer) so this allows the speed to be set while still using `requestAnimationFrame` to improve the performance of the rendering. Now, I mentioned animation does not have to run at full speed, but I have not yet showed you how to easily change that speed–for that we are going to use Linear Interpolation.

Linear Interpolation

Linear Interpolation (lerp) is used to define subcoordinates of the particular path. This can be likened to taking a long journey and stopping off for food along the way. In animation, it is used to smooth out the path, so instead of jumping from one point to another it appears to slide between them (of course, it actually just makes much smaller jumps).

```
n = start + (end - start) * speed
```

The preceding code equation is used to work out the next point of the interpolation. This should be iterated so that every frame the start variable is the `n` of the previous frame, because it is gradually moving toward the end point (now dependent on the speed set rather than the fps).

```
var ele = document.querySelector("#animation");
var ctx = ele.getContext('2d');
var startX = 10;
var startY = 10;
var endX = ele.width - 50;
```

```
var x = startX;
var y = startY;
var duration = 0;
var width = height = 50;

function lerp(start, end, speed) {
    return start + (end - start) * speed;
}

function logic () {
    duration += 0.02;
    x = lerp(startX, endX, duration);
    if (x < ele.width - width)
        requestAnimationFrame(draw);
}

function draw()  {
    ctx.clearRect(0, 0, ele.width, ele.height);

    // This sets the fill colour to red
    ctx.fillStyle = "#ff0000";

    // fillRectangle(x, y, width, height);
    ctx.fillRect(x, y, 50, 50);
}
requestAnimationFrame(draw);
setInterval(logic, 1000/60);
```

This is not too different from the previous version, except we now have a `lerp()` function almost identical to the earlier equation. I have also defined duration, `startX`, and `startY` variables to be plugged into the function. The biggest change is `logic()`. Rather than adjusting `x` (which causes it to jump), I increased duration by a tiny amount, which I then simply plugged into the `lerp()` function to get the new value for `x` (which is somewhere on the path to the final x destination).

Follow the Mouse

Quite often, you will need interactive animation rather than just a video. A common way to add interactivity within canvas is by using the mouse. To start with, I will show you how you can get your little red square to follow your movements.

The mouse is handled by the operating system and so triggers an event for all programs to access, which means of course that the term “mouse” actually means any device capable of triggering the event. For this case, we need to look for the event `'mousemove'`. To do so, we use `addEventListener` to add the listener to a specific element.

```
element.addEventListener(type, listener, useCapture boolean optional);
```

`Type` is the name of the event (see the list that follows of mouse-specific events). `Listener` can be implemented in two ways, either as an object that implements `EventListener` or as a simple callback function. The `useCapture` is a Boolean that is true if the event should be triggered on the capture stage; otherwise it will be triggered on target and bubbling phases. `useCapture` will default to false on modern browsers though some older browsers will require it.

```
click
dblclick
mousedown
mouseenter
mouseleave
mousemove
mouseover
mouseout
mouseup
```

To make the mouse move, you need a callback that sets the new start and end points based on the mouse, like so:

```
ele.addEventListener('mousemove', function(evt) {
    startX = x;
    endX = evt.clientX;
});
```

This sets the new path for the square, so now we just need to modify it slightly so that it is smooth. I have decided to add constraints to make sure the square doesn't try to go off the canvas or glitch if the mouse sets x to where the square is already heading. If it doesn't match this constraint, then duration gets set to 0.

```
function logic (evt) {
    var max = ele.width - width;
    duration += 0.02;
    var l = lerp(startX, endX, duration);
    if (l < max && l > 0 && endX != x)
    {
        x = l;
        requestAnimationFrame(draw);
    }
    else {
        duration = 0;
    }
}
```

This should show you just how easily a few tiny changes can completely change the whole animation and how the user interacts with it. See Listing 2-1 to see how it all fits together.

Listing 2-1.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Chapter 2 - Basics of Canvas</title>
    </head>

    <body>
        <canvas id="animation">
            <p>Fallback not supported.</p>
        </canvas>
        <script src="script.js"></script>
    </body>
</html>
```

```
// Polyfill for RequestAnimationFrame
(function() {
  var requestAnimationFrame = window.requestAnimationFrame || window.mozRequestAnimationFrame ||
                              window.webkitRequestAnimationFrame || window.msRequestAnimationFrame;
  window.requestAnimationFrame = requestAnimationFrame;
})();

var ele = document.querySelector("#animation");
var ctx = ele.getContext('2d');
var width = height = 50;
var startX = 10;
var startY = 10;
var endX;
var x = startX;
var y = startY;
var duration = 0;

function logic (evt) {
    var max = ele.width - width;
    duration += 0.02;
    var l = lerp(startX, endX, duration);
    if (l < max && l > 0 && endX != x)
    {
        x = l;
        requestAnimationFrame(draw);
    }
    else {
        duration = 0;
    }
}

function draw()  {
    ctx.clearRect(0, 0, ele.width, ele.height);

    // This sets the fill colour to red
    ctx.fillStyle = "#ff0000";

    // fillRectangle(x, y, width, height);
    ctx.fillRect(x, y, 50, 50);
}

function lerp(start, end, speed) {
    return start + (end - start) * speed;
}

ele.addEventListener('mousemove', function(evt) {
    startX = x;
    endX = evt.clientX;
});

requestAnimationFrame(draw);
setInterval(logic, 1000/60);
```

Bouncing Box

Now that we have looked at how to animate in a static way (moving from left to right) as well as using an input, I would like to do one more example using the box by adding a dynamic element to it. There are so many options for how an animation works. The most common is bouncing an object off the boundaries of the canvas. It is not much of a step up in difficulty from the previous example but will tie in nicely with the next example (which will be a naïve implementation of flocking boids).

Instead of "lerping" between current and desired positions, to bounce a box we need to make it go in a specific direction until it hits an edge. This means that we need to use basic trigonometry to take an angle (it will begin as a random angle under 360 degrees) and to find the position. As you know from lerping, it is best to move in lots of small steps rather than a big one, so we can define the distance for the trigonometry function to find. To find the x direction, we find the cos of the angle (in radians) multiplied by the distance and for the y direction we do the same but using the sin of the angle.

```
function degreesToRadians(degrees) {
    return degrees * (Math.PI / 180);
}

function dir_x(length, angle) {
    return length * Math.cos(degreesToRadians(angle));
}

function dir_y(length, angle) {
    return length * Math.sin(degreesToRadians(angle));
}
```

Once we implement these functions, it is similar to the previous examples. We need to initialize variables for both x and y axes for distance and heading (direction). Distance should start as 0, and heading needs to be a random angle up to 360 degrees. Within logic, it is best to start with a simple check to make sure both `heading` variables are between –360 and 360. Then we need to check whether the object is within the boundaries, if it isn't, then bounce it back. After that, we simply lerp between current position and the position that is found by using the degrees and direction functions above.

```
var ele = document.querySelector("#animation");
ele.height = window.innerHeight;
ele.width = window.innerWidth;
var ctx = ele.getContext('2d');
var x = 10;
var y = 10;
var duration = 0;
var width = height = 50;
var heading_x = heading_y = Math.random() * 360;
var distance_x = distance_y = 0;

function logic () {
    if (heading_x > 360 || heading_x < -360) heading_x = 0;
    if (heading_y > 360 || heading_y < -360) heading_y = 0;

    if (x <= 0 || x >=ele.width - width) {
        heading_x = heading_x + 180;
    }
```

```
    if (y <= 0 || y >=ele.height - height) {
        heading_y = -heading_y;
    }

    distance_x = dir_x(2, heading_x);
    distance_y = dir_y(2, heading_y);
    if (duration < 10) duration += 0.05;
    x = lerp(x, x + distance_x, duration);
    y = lerp(y, y + distance_y, duration);
    requestAnimationFrame(draw);
}
```

And that's that. You can find the full code listing for all the examples in the download that complements this book on the Apress website at www.apress.com/9781430259442 or my own website at www.shanehudson.net/javascript-creativity.

"Clever" Animation

Next we are going to create a "clever" animation–that is to say, extending on the previous concept of dynamic animation found in the bouncing box example by making objects aware of their surroundings. In 1986, Craig Reynolds created an incredible simulation of birds flocking that he called Boids. It was built on just three rules that allowed each boid to be responsible for its own movement, but also allowed it to see local neighbors and move toward them. The best explanation for each rule I have found was this:

- **Separation**: Steer to avoid crowding local flockmates
- **Alignment**: Steer toward the average heading of local flockmates
- **Cohesion**: Steer to move toward the average position (center of mass) of local flockmates

Our animation is based on Boids, though some of it will be simplified for brevity (I encourage you to modify my code to improve it–to do so you will probably want to use vectors). We will start with our usual set up code, getContext. etc., as well as creating an object for Boid that holds the data about location, direction, and size of the boid. Once that is done we create a function called `setup()`, which adds each boid (depending how many we set) onto the canvas at a random position with a random direction. I have made a simple function to wrap the JavaScript Math.random, just to make the code a bit neater.

```
(function()  {
 var canvas = document.querySelector("#flocking");
 var ctx = canvas.getContext('2d');

 canvas.height = window.innerHeight;
 canvas.width = window.innerWidth;
 var flock = [];

 var flockRadius = 250;
 var neighborRadius = 10;

 var Boid = function(x, y, heading, size) {

   this.x = x;
   this.y = y;
```

```
    this.heading = heading
    this.size = size;

  };

  function setup()  {
    for (var i = 0; i < 50; i++)
    {
       flock.push(new Boid(rand(canvas.width), rand(canvas.height), rand(360), 15));
    }
    setInterval(logic, 1000/60);
  }

  function logic () {
    for (var i = 0; i < flock.length; i++)  {
      // Do something with each boid
    }
    requestAnimationFrame(draw);
  }

  function draw()  {
    // Drawing goes here
  }

  function rand(max)  {
    return Math.random() * max;
  }
  setup();
})();
```

You may notice that for direction instead of "up" or "right" we use degrees, 0 to 360. This makes the animation much more fluid and also allows us to use radian math later. We use a variable `flockRadius` to control the distance at which boids can see each other (used for heading the same way, etc.). Now let's add some code to the drawing method. For this we are going to need to start by clearing the canvas each frame (if you don't do this, you will just draw over the previous frame). Once the canvas is cleared, it is time to draw the boids! So we need to iterate over the flock array, while storing the current boid in a variable called `b`, drawing each one to the canvas (that is, the context of the canvas) at the correct position. To find the position, we take `b.x` and `b.y` but we need to add half of its size onto it, because the position is the center of the boid rather than the top left of it.

```
function draw()  {
  ctx.clearRect(0,0,canvas.width, canvas.height);
  for (var i = 0; i < flock.length; i++)
  {
    var b = flock[i];
      ctx.fillStyle = "blue";
      ctx.fillRect(b.x,b.y, b.size, b.size);
      ctx.beginPath();
      ctx.moveTo(b.x + (b.size / 2),b.y + (b.size / 2));
```

```
      ctx.lineTo((b.x + (b.size / 2)) + dir_x(20,flock[i].heading),(b.y + (b.size / 2)) +
dir_y(20,flock[i].heading));
      ctx.strokeStyle = "red"
      ctx.stroke();
  }
}
```

In `lineTo` I have used a couple of functions used to get the position when given a distance and direction. In this case I have used the functions to draw a line pointing 20px in the direction each boid is heading. Here you can see the helper functions. They use basic trigonometry and Pythagoras, so they should be fairly easy to follow.

```
function distanceBetween(a, b)  {
  var dx = a.x - b.x;
  var dy = a.y - b.y;
  return Math.sqrt(dx * dx + dy * dy);
}

function angleBetween(x1, y1, x2, y2)
{
  return Math.atan2(y1 - y2, x1 - x2) * (180.0 / Math.PI);
}

function angleDifference(a1, a2)
{
  return ((((a1 - a2) % 360) + 540) % 360) - 180;
}

function degreesToRadians(degrees){
   return degrees * (Math.PI / 180);
}

function dir_x(length, angle){
  return length * Math.cos(degreesToRadians(angle));
}

function dir_y(length, angle){
  return length * Math.sin(degreesToRadians(angle));
}
```

Don't be put off by all the math–it just makes it much easier to write our logic function, which we are going to do now! We need to start by setting up some variables within the for loop for the position of where the boid is headed. This is called `centerx` and `centery`, but it is not the center of the canvas (I think of it as the "center of attention"). I also provide a variable `b` to make it easier to access `flock[i]`. With the variables set up, we can now loop through the boids again to find the neighbors, which is any boid within a distance of less than flockRadius. With these we can find the average position (adding the x and y of each boid to `centerx` and `centery`, then dividing by the amount of boids in the radius). Of course, if there is only one boid in the flock we might as well give it a random position to head toward.

```
for (var i = 0; i < flock.length; i++)  {
  var centerx = 0;
  var centery = 0;
  var count = 0;
```

```
  var b = flock[i];

  for (var j = 0; j < flock.length; j++)
  {
    var distance = distanceBetween(b, flock[j]);
      if (distance < flockRadius)
      {
        centerx += flock[j].x;
        centery += flock[j].y;
        count++;
      }
  }

  if (count > 1) {
    centerx = centerx / count;
    centery = centery / count;
  }
  else  {
    centerx = Math.random() * canvas.width;
    centery = Math.random() * canvas.height;
  }
  // Set heading and x/y positions
}
```

Now that we have our center of attention/gravity (I suppose gravity is a better word for it) we can work out the angle the boid needs to turn to head in the correct direction. We do this by using the `angleBetween` and `angleDifference` helper functions to get the angle needed, then linear interpolating it so that every iteration it gets closer to the point it is heading. Of course, due to the way we have set up the code, the point it is heading to may change depending on its proximity to its neighbors (remember we change the center point in the nested loop). Lastly, so that the boids don't just fly off the page, we need to define a "wrap around" so that if the boid goes off the canvas it appears on the opposite side (just like Asteroids or Snake).

```
if (count > 1) {
  centerx = centerx / count;
  centery = centery / count;
}
else  {
  centerx = Math.random() * canvas.width;
  centery = Math.random() * canvas.height;
}

var angleToCenter = angleBetween(b.x,b.y,centerx,centery);
var lerpangle = angleDifference(b.heading, angleToCenter);

b.heading += lerpangle * 0.01;

headingx = dir_x(2,b.heading);
headingy = dir_y(2,b.heading);

b.x += headingx;
b.y += headingy;
```

```
if (b.x < 0) b.x = canvas.width;
if (b.y < 0) b.y = canvas.height;

if (b.x > canvas.width) b.x = 0;
if (b.y > canvas.height) b.y = 0;
```

If you put the above code where the comment was on the previous code, you should find that you now have blue boids with red lines, indicating direction, which fly around the canvas and join groups. Figure 2-2 shows how this should look. The animation can be improved in a number of ways, such as adding separation so they fly next to each other rather than through one another.

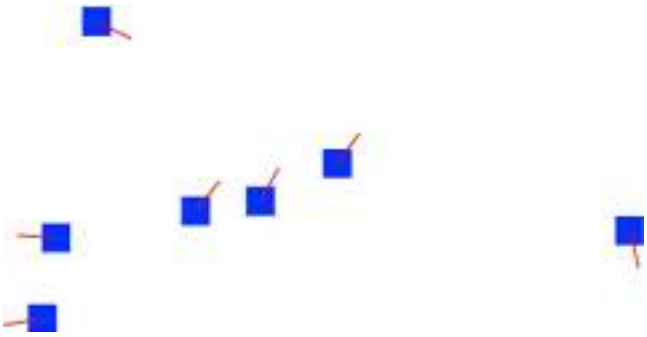

Figure 2-2. *This shows each boid heading a specific direction. Also note the three boids in the middle heading the same way as a flock*

In listing 2-2 you can see the full code for this example, which is available for download online as with the previous examples.

Listing 2-2.

```
(function()  {
  var canvas = document.querySelector("#flocking");
  var ctx = canvas.getContext('2d');

  canvas.height = window.innerHeight;
  canvas.width = window.innerWidth;
  var flock = [];

  var flockRadius = 250;

  var Boid = function(x, y, heading, size) {

    this.x = x;
    this.y = y;
    this.heading = heading
    this.size = size;

  };
```