Community Experience Distilled

Getting Started with CreateJS

Design and develop astounding animated web applications using CreateJS

Afshin Mehrabani

[PACKT] PUBLISHING open source* community experience distilled

Getting Started with CreateJS

Design and develop astounding animated web applications using CreateJS

Afshin Mehrabani

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with CreateJS

First published: February 2014

Production Reference: 1130214

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-026-0

www.packtpub.com

Cover Image by Manohar V (manoharv.cam@gmail.com)

Credits

Author
Afshin Mehrabani

Reviewers
Bryan Butler
Danilo Matamoros
Kailash Nadh

Acquisition Editors
Neha Nagwekar
Julian Ursell

Content Development Editor
Sankalp Pawar

Technical Editors
Mrunmayee Patil
Aman Preet Singh

Copy Editors
Brandt D'Mello
Adithi Shetty

Project Coordinator
Sanket Deshmukh

Proofreader
Elinor Perry-Smith

Indexer
Hemangini Bari

Graphics
Abhinash Sahu

Production Coordinator
Komal Ramchandani

Cover Work
Komal Ramchandani

About the Author

Afshin Mehrabani is a 21-year-old software engineer and an open source programmer. He is also a computer software engineering student. He started with programming and PHP web development when he was 12 years old. Later, he entered the Iran Technical and Vocational Training Organization. He was ranked first and has also bagged a golden medal in a competition on web development in his country. He also became a member of the Iran's National Elite Foundation by producing a variety of new programming ideas.

He has worked as a software engineer in the Tehran Stock Exchange and is presently the head of the web development team at Yara International. He cofounded the Usablica team in early 2012 to develop and produce usable applications. Afshin is the author of IntroJs, WideArea, Flood.js, and some other open source projects.

Also, he has contributed to Socket.IO, Engine.IO, and some other open source projects. His interests lie in creating and contributing to open source applications, writing programming articles, and challenging himself with new programming technologies.

He has already written different articles about JavaScript, NodeJS, HTML5, and MongoDB, which are published in different academic websites. Afshin has five years of experience with PHP, Python, C#, JavaScript, HTML5, and NodeJS in many financial and stock-trading projects.

I would like to thank my parents and my lovely sister Parvin for their support, which gave me the power to keep going.

About the Reviewers

Bryan Butler has developed a wide variety of digital media and web-based projects in Ireland and the UK since 2002. These include everything from brand websites, mobile solutions, and asset/administration management systems to interactive TV interfaces, e-learning simulators, and educational games. He currently specializes in providing digital and web-based solutions using the latest HTML5 technologies. For more information and examples of his work you can go to `www.brybutler.com`.

Danilo Matamoros is a web developer professional with over seven years of experience working in multiple projects in Latin America and Australia, having developed more than 20 websites and web applications for organizations in the private, public, educational, charity, tourism, and commercial sectors.

Kailash Nadh is an independent developer, researcher, and tech consultant with over a decade of experience. His research interests include computational, linguistics, and artificial intelligence. For more information about him visit his personal website at `http://nadh.in`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

CreateJS is a full-featured tool for web developers to create and maintain animations and drawings in web browsers using HTML5. It consists of some modules, each of which performs different tasks to manage a web-based application.

In this book, we will discuss the different parts of CreateJS using many interactive examples and screenshots.

What this book covers

Chapter 1, *Installing CreateJS*, serves as a quick installation reference for users new to CreateJS.

Chapter 2, *Commencing with CreateJS*, covers getting started with CreateJS and other components, using API, and configuring modules.

Chapter 3, *Working with Drag-and-drop Interactions*, discusses the drag-and-drop features of CreateJS and how to customize these features in projects or extend them.

Chapter 4, *Performing Animation and Transforming Function*, covers how to use animation and transform objects on the page using CreateJS.

Chapter 5, *Utilizing Caching in EaselJS*, covers how to use caching in CreateJS for better performance in animations.

Chapter 6, *Using Filters in EaselJS*, discusses using filters in images and objects in CreateJS.

Chapter 7, *Developing a Painting Application*, discusses how to develop a simple painting application that is drawn on the canvas using graphics.

Chapter 8, *Utilizing Vector Masks*, discusses using shapes as a clipping path on any display object.

Chapter 9, *Developing Your First CreateJS Application*, covers how to build and create a UI from scratch with CreateJS.

What you need for this book

All examples and source code present in this book work in modern web browsers. You will need to install the latest version of Google Chrome or Mozilla Firefox for all the examples to work perfectly.

Who this book is for

If you are a web developer with some experience in JavaScript development and want to enter the fascinating world of feature-rich Internet applications using CreateJS, then this book is perfect for you.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text are shown like this: "In EaselJS, when you have a shape, or even better, an instance of the `DisplayObject` that doesn't change frequently, it's better to use the `cache` function to cache it in a different `Canvas` element".

A block of code is set as follows:

```
var circle = new createjs.Shape();
circle.graphics.beginFill("red").drawCircle(0, 0, 50);
circle.x = 100;
circle.y = 100;
```

Any command-line input or output is written as follows:

```
# Install the grunt command line utility globally
npm install grunt-cli -g
```

```
# Install all the dependencies from package.json
npm install
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "With drop-down menus, users can change the **Brush Style**, **Brush Size**, **Background Color**, and **Brush Color** fields."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/0260OS_coloredimages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Installing CreateJS

Installing and using a client-side library in the right manner is really important. Using a large client-side library incorrectly could cause real problems; one obvious problem is the slow loading of web pages. In this chapter, we are going to see how we can set up and use CreateJS for both development and production environments.

In this chapter, you'll learn about the following topics:

- Requirements
- Download
- Installation and setup

Understanding CreateJS and subsets

CreateJS contains different libraries such as **SoundJS**, **TweenJS**, **EaselJS**, and **PreloadJS**. Each one has different requirements to run in browsers.

TweenJS should work in all browsers and their older versions as well. SoundJS requires **HTMLAudio**, **Flash**, or **WebAudio**, and these features need modern browsers. SoundJS should work perfectly with the following browsers:

- Google Chrome 31+
- Mozilla Firefox 25+
- Internet Explorer 9+
- Opera 18+

Among the mobile browsers, it should work with these:

- iOS Safari 4.0+
- Android browser 2.3+
- BlackBerry browser 7.0+

PreloadJS should work with all browsers, even Internet Explorer 6.

EaselJS has a dependency on HTML5 canvas, so your audiences should have one of the following browsers to be able to use EaseJS:

- Google Chrome 31+
- Mozilla Firefox 25+
- Internet Explorer 9+
- Opera 18+

For mobile, one of these browsers is required:

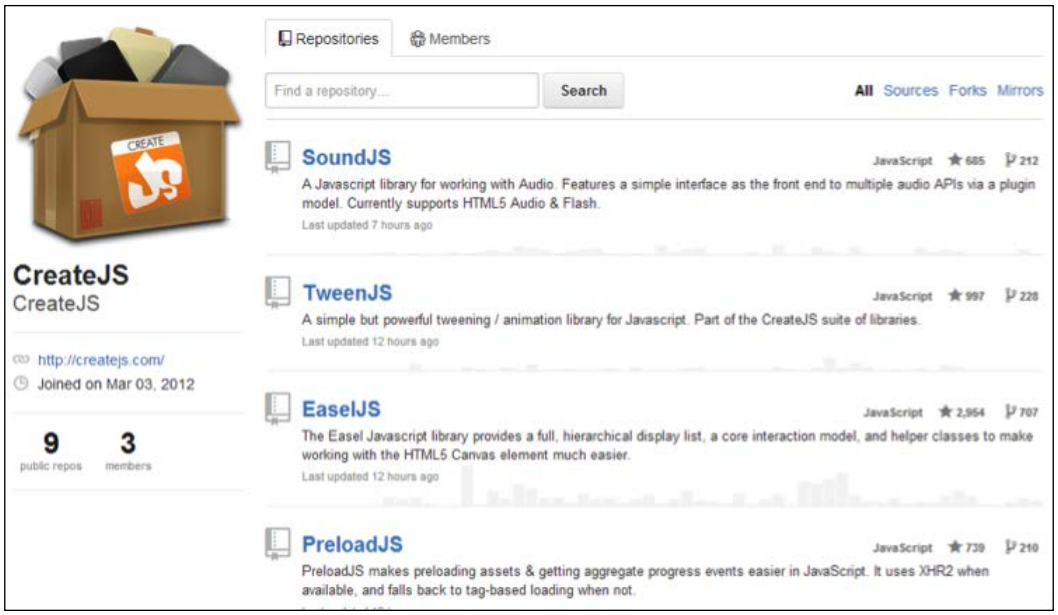
- iOS Safari 3.2+
- Opera Mini 5.0-7.0
- Android browser 2.1+
- BlackBerry browser 7.0+

Downloading CreateJS

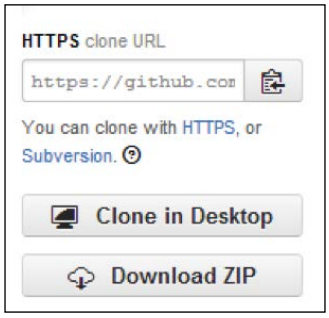
There are some ways to download and include CreateJS files into your project which are discussed in the following sections.

GitHub

You can download the latest version of CreateJS and all subsets from the GitHub, as shown in the following screenshot from `https://github.com/CreateJS/`:



After going to each repository, you can either download the ZIP folder of the latest changes or use the Git clone command to get the source from GitHub, as shown in the following screenshot:



You can find this box on the right-hand side of each repository page, which helps you to get the code.

Understanding the Content Delivery Network

CreateJS has a great **Content Delivery Network** (**CDN**), which contains all versions and all subset libraries hosted. You can access CDN servers from `http://code.createjs.com/`.

Using the CDN-hosted versions of CreateJS libraries in your project allows them to be downloaded quickly and cached across different sites using the same version of the libraries. This can reduce bandwidth costs and page load times.

You can find all versions at `http://code.createjs.com/` as shown in the following screenshot:

You can also use a combined version of all subsets and libraries that contains all of the latest stable versions of all libraries, including EaselJS, TweenJS, SoundJS, and PreloadJS.

Setting up the libraries

After choosing how to download the source, you need to set up the library to make it work. In this section, we are going to see how to set up the library for both production and development environments.

The production environment

If you want to use CreateJS in the production environment, things will be much easier. All you need to use is a single compiled source file. For this purpose, you can either use hosted files from CreateJS CDN or build locally, and then link in locally.

Using CDN

In this case, all you need to do is link `<script…>` to the source of the file in the CreateJS CDN server (see the *Downloading CreateJS* section of this chapter); after that, everything should work properly.

The development environment

If you want to debug, develop using CreateJS, or see how things are going on inside CreateJS, you need to use uncompiled source files.

For each library, you can find the uncompiled source files at `/src/`. All dependencies and source code will be here.

To use uncompiled source files, the important thing to keep in mind is that all files should be called in the correct order; otherwise, you will get some errors. Luckily, there is a file in each project that will give you hints on how to include files in the correct order. This config file is placed in `build/config.json`.

You can see an example of this config for EaselJS in the following screenshot:

```
"easel_source": [
        "../src/createjs/events/Event.js",
        "../src/createjs/events/EventDispatcher.js",
        "../src/createjs/utils/IndexOf.js",
        "../src/easeljs/utils/UID.js",
        "../src/easeljs/utils/Ticker.js",
        "../src/easeljs/events/MouseEvent.js",
        "../src/easeljs/geom/Matrix2D.js",
        "../src/easeljs/geom/Point.js",
        "../src/easeljs/geom/Rectangle.js",
        "../src/easeljs/ui/ButtonHelper.js",
        "../src/easeljs/display/Shadow.js",
        "../src/easeljs/display/SpriteSheet.js",
        "../src/easeljs/display/Graphics.js",
        "../src/easeljs/display/DisplayObject.js",
        "../src/easeljs/display/Container.js",
        "../src/easeljs/display/Stage.js",
        "../src/easeljs/display/Bitmap.js",
        "../src/easeljs/display/Sprite.js",
        "../src/easeljs/display/BitmapAnimation.js",
        "../src/easeljs/display/Shape.js",
        "../src/easeljs/display/Text.js",
        "../src/easeljs/display/BitmapText.js",
        "../src/easeljs/utils/SpriteSheetUtils.js",
        "../src/easeljs/utils/SpriteSheetBuilder.js",
        "../src/easeljs/display/DOMElement.js",
        "../src/easeljs/filters/Filter.js",
        "../src/easeljs/filters/BlurFilter.js",
        "../src/easeljs/filters/AlphaMapFilter.js",
        "../src/easeljs/filters/AlphaMaskFilter.js",
        "../src/easeljs/filters/ColorFilter.js",
        "../src/easeljs/filters/ColorMatrix.js",
        "../src/easeljs/filters/ColorMatrixFilter.js",
        "../src/easeljs/ui/Touch.js",
        "../src/easeljs/version.js"
],
```

So, you have to put `<script…>` and load files in this order. You can find the same config file in the same location for other projects too.

After loading all JS files, you can use the library and also put your breakpoints on the source code to trace or debug it.

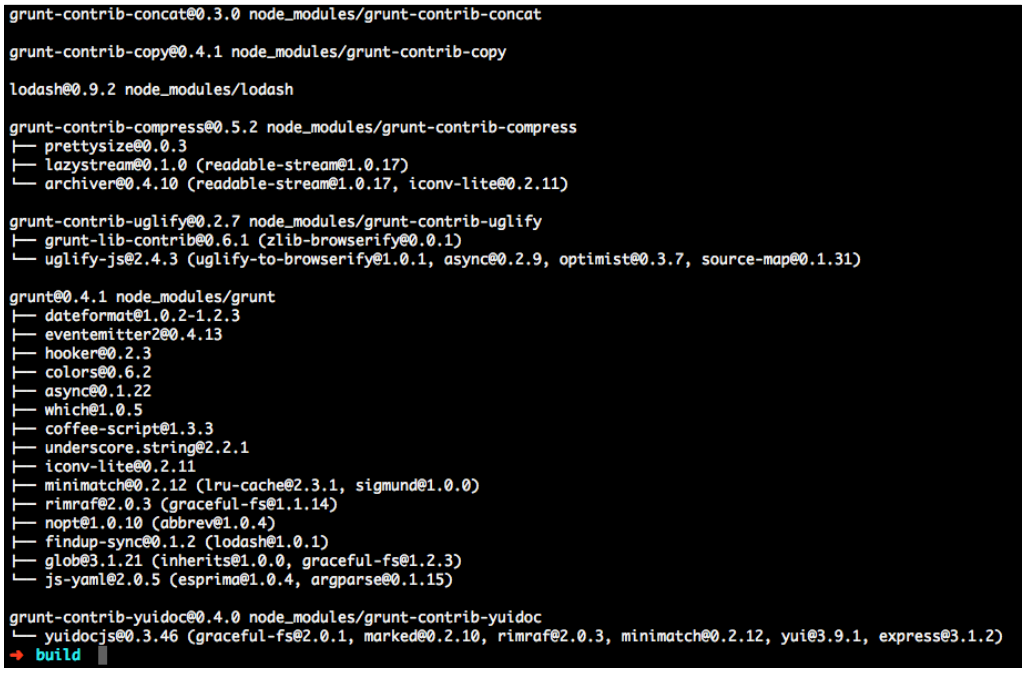
Building the source code

All CreateJS libraries use Grunt to make and build files, so you need to have the NodeJS and Grunt module installed (0.10.2 or greater is required).

First of all, get the latest version of NodeJS from `www.nodejs.org` and install it. Then go to the `/build` folder for the library (for example, EaselJS) and run the following commands in your command environment:

```
# Install the grunt command line utility globally
npm install grunt-cli -g


# Install all the dependencies from package.json
npm install
```

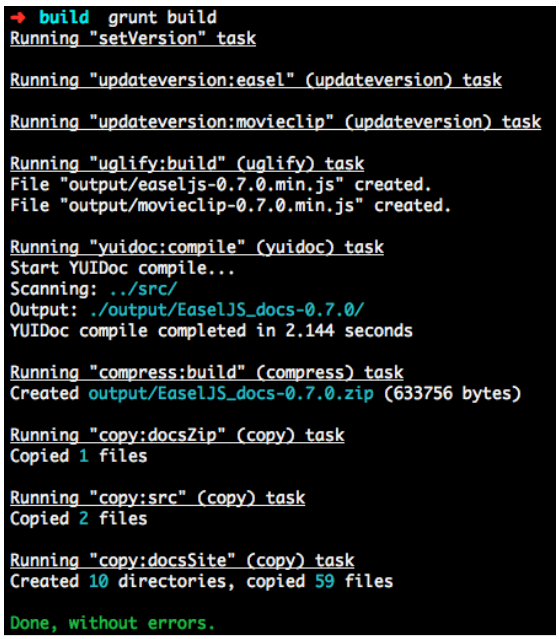
After executing these commands, you should obtain a result as shown in the following screenshot:



After installing all dependencies, you have only one step left to build the library. Go to the library folder and run the following command:

```
grunt build
```

You should see the following result in your command environment:



When you see the `Done, without errors` message, you can find your compiled file in the `/lib` folder of the library, named `{PROJECT_NAME}-{VERSION}.min.js`, for example `easeljs-0.7.0.min.js`. This is exactly same as the files in the CDN server. You can link your `script` tag locally to this file and use it.

For more information about options and how to work with them, you can read the `README.md` file in the `/build` folder of each library.

Summary

In this chapter, we have learned how to set up and prepare CreateJS for different environments, because as a programmer, you may want to make changes to CreateJS or customize it for yourself. We also discussed using CreateJS for the production environment, using CDN servers, and building the source code.

In the next chapter, we are going to discuss how to write the first working example with CreateJS and how to use the API.

2
Commencing with CreateJS

In this chapter, we are going to talk about working with basic objects and events in CreateJS and EaselJS. After learning about these topics, you can work with basic methods and functions in CreateJS to create your shapes, and control them using events.

We are going to discuss the following topics:

- Exploring CreateJS
- Working with APIs
- Methods and events

Exploring CreateJS

EaselJS is one of the main CreateJS modules, which enable developers to work with Canvas elements. To work with EaselJS, we need to have a `canvas` element, so all shapes can be rendered into this area. After creating an instance of `Stage` class, we need to add `displayObject` to the `Stage` class. EaselJS supports the following features:

- **Bitmap**: This is used for the images.
- **Shape** and **Graphics**: These are used for the vector graphics.
- **SpriteSheet** and **Sprite**: These are used for the animated Bitmaps.
- **Text**: This is used for the simple text instances.
- **Container**: These hold other DisplayObjects.
- **DOMElement**: This is used to control the HTML DOM elements.

When the `Stage` object wraps the `canvas` element, all shapes and text appear in the `Canvas` element.

For more details, check the EaselJS documentation at `http://www.createjs.com/Docs/EaselJS/modules/EaselJS.html`.

Let's go through an example of creating a basic shape in EaselJS. Here, we have a `canvas` element with a specific height and width:

```
<canvas id="demoCanvas" width="500" height="200"></canvas>
```

CreateJS has a `Stage` method, which accepts a `canvas` element in the first parameter, and we should pass the ID of our canvas element to it:

```
var stage = new createjs.Stage("demoCanvas");
```

We now have a stage for our canvas element. In the next step, we need to create a shape:

```
var circle = new createjs.Shape();
circle.graphics.beginFill("red").drawCircle(0, 0, 50);
circle.x = 100;
circle.y = 100;
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

In the first line, a `circle` variable is created. It contains the `Shape` object from EaselJS. All `Shape` objects have a `graphics` property.

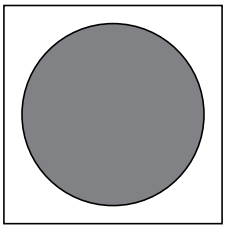
In the next line, we fill it with the color red using the `beginFill` method, and then in the line after that, we create a circle with the `drawCircle` method. The `drawCircle` method has three parameters; the first two parameters are used for positioning the circle (x and y axis values) and the last parameter is the radius in pixels. Thus, we have created a circle with position `0` (relative to the shape's position) and radius `50`.

EaselJS supports method chaining, we can call all functions one after another, just as we have seen in our previous example of creating the circle and filling in the background color.

After creating the `Shape` object, we need to add it to our `stage` object and also update the stage as follows:

```
stage.addChild(circle);
stage.update();
```

Keep in mind that after adding `child` (`shape`, `circle`, and so on), we have to call the `update` method from the `stage` object to update the stage; otherwise, the code will not run properly and we will not get the desired result. You can see the result of our simple code in the following screenshot:



Working with events

`DisplayObject` has a method to add events to shapes or objects. Using `addEventListener`, we can add an event to `DisplayObject` (for example, `shape`). This function has two mandatory arguments:

- The name of the event
- The callback function for the event

We will understand this method of working with events with the following code:

```
displayObject.addEventListener("click", handleClick);
function handleClick(event) {
  // Click happened.
}
```

In the first line, a click event is added to `displayObject` so that the `handleClick` function is called when the user clicks on the object. The `handleClick` function is empty in this example.

Let's consider our earlier example of the circle and add a click event to our circle. Inside the callback function of the click event, we move the circle 10 pixels to right. Here is the code for that:

```
circle.addEventListener("click", handleClick);
function handleClick(event) {
event.target.x += 10;
stage.update();
}
```

In the first line, we have our DisplayObject. Using `addEventListener`, the click event is added to the circle. Our callback handler is `handleClick`. Inside this function,
we can get target objects (the circle shape, in this example) and change properties of the shape (for example, width, height, or position) via the `event` variable.

`event.target` is the target shape object. In every callback function call, we add the x property with 10 and then call the `update` function from the `stage` object. We have to call the update function after changing properties in order to apply changes.

Remember that to add events to `DisplayObject`, we need to add an event listener first and then add displayObject to the stage. Here is the complete source code for our example:

```
var stage = new createjs.Stage("demoCanvas");

var circle = new createjs.Shape();
circle.graphics.beginFill("red").drawCircle(0, 0, 50);
circle.x = 100;
circle.y = 100;

circle.addEventListener("click", handleClick);
function handleClick(event) {
  event.target.x += 10;
  stage.update();
}
stage.addChild(circle);
stage.update();
```

EaselJS has many more events, and you can use all of them in the same example explained previously.

Currently DisplayObjects supports the following events:

- `click`: The user clicks and releases the mouse
- `dblclick`: The user double-clicks the mouse
- `mousedown`: The user clicks the mouse
- `mouseout`: The user moves the mouse pointer away from an object
- `mouseover`: The user moves the mouse pointer over an object
- `pressmove`: The user clicks the mouse and then moves it
- `pressup`: The user releases the mouse either over or outside the object
- `rollout`: The user rolls away from a child element
- `rollover`: The user rolls over a child element

For more details, refer to `http://www.createjs.com/Docs/EaselJS/classes/DisplayObject.html#events`.

Summary

In this chapter, we have learned how to work with the basic functions and events of CreateJS and EaselJS. We have learned how to create a stage object in EaselJS, what `DisplayObject` is, and how to append them to the `stage` object.

We also created the first simple shape in EaselJS, a circle. In the last part of the chapter, we discussed how to add an event listener to an object in EaselJS.

In the next chapter, we are going to go through complex examples to create a drag-and-drop interaction and use mouse events in EaselJS.

3
Working with Drag-and-drop Interactions

In this chapter, we will go through the basic events and callbacks of CreateJS to make a **drag-and-drop** feature. After reading this chapter, you will be able to understand common events of objects in CreateJS and also how to change properties such as width or height. We will cover the following topics in this chapter:

- Scenarios for drag-and-drop
- Mouse events
- Creating an example to drag-and-drop

The scenario

All we need to do to create a drag-and-drop feature is to bind events to `DisplayObject` on the `Stage` object (such as a circle) and then change the x and y axes of the target object continuously as the mouse is moved. Fortunately, CreateJS provides many events on mouseover, and we can use them to achieve our goals.

In further sections, we will see how we can bind events to objects and get information from the mouse cursor. We will also see how to change the properties of an object or shape in the `Stage` object.

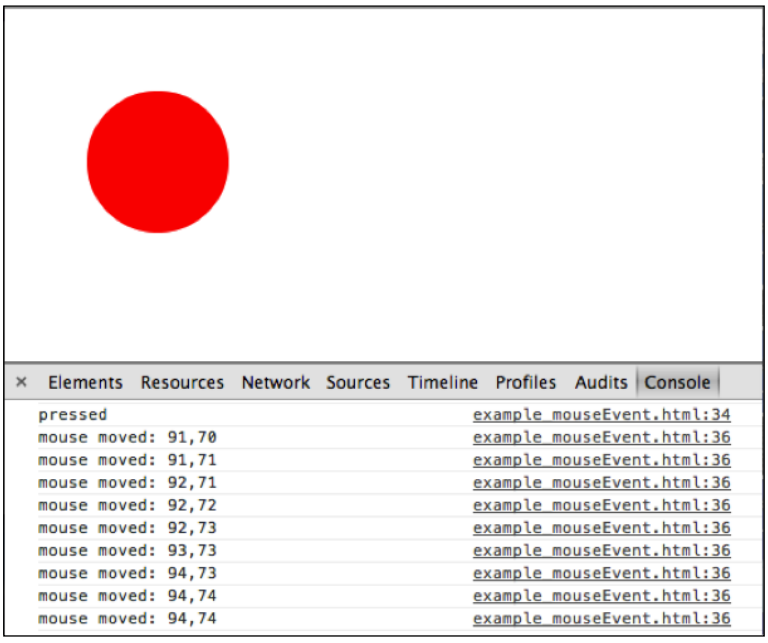
Understanding the on function

In EaselJS, you can get access to all mouse events, such as click, mouse up, and so on. A `MouseEvent` instance is passed as the only parameter for all mouse event callbacks. It includes the `stageX` and `stageY` properties, which indicate the cursor's position relative to the stage coordinates.

In the following example, `pressed` will be logged on to the console when the mouse is clicked over the circle; thereafter, mouse moves will be logged whenever the mouse moves until the mouse is released.

```
circle.on("mousedown", function(mousedownEvent) {
    console.log("pressed");
    circle.on("pressmove", function(moveEvent) {
        console.log("mouse moved:
          "+moveEvent.stageX+","+moveEvent.stageY);
    });
});
```

As you can see, we can simply bind a mouse event to our `DisplayObject` object and then read or alter the properties on each event callback. The following screenshot displays the output of the previous example along with the details of mouse events:



In the next section, we will put all these things together and create a simple drag-and-drop interaction.

Creating a drag-and-drop interaction

We learned in the previous chapter that the first requirement for working with EaselJS is to create a `Stage` object and then append all the `DisplayObject` objects to it. Suppose we have a `Canvas` element with the ID `demoCanvas`. We will need the following code for the same:

```
var stage = new createjs.Stage("demoCanvas");
```

If you want to track the movement of mouse cursor events when the mouse cursor leaves the `Canvas` element, `mouseMoveOutsideproperty` should be set to `true`:

```
stage.mouseMoveOutside = true;
```

In the next step, a circle will be created:

```
var circle = new createjs.Shape();
circle.graphics.beginFill("red").drawCircle(0, 0, 50);
```

And, of course, we have to add our shape to the `stage` element shown as follows:

```
stage.addChild(circle);
```

Now, it's time to bind functions to the `mousedown` and `pressmove` events:

```
circle.on("mousedown", function (evt) {
    var offset = {
        x: evt.target.x - evt.stageX,
        y: evt.target.y - evt.stageY
    };

    circle.on("pressmove", function (ev) {
        ev.target.x = ev.stageX + offset.x;
        ev.target.y = ev.stageY + offset.y;
        stage.update();
    });
});
```

As you can see, we have set a callback function for the `mousedown` event of our display object, `circle`. Inside the anonymous function, there is an offset variable that has two properties, `x` and `y`. These properties collect the offset values of the mouse cursor on every mouse down within the shape (circle in this example), so we can use this offset value to change the position of the circle. In this example, `x` or `y` could be between +50 and -50.

After that, an anonymous function is added to the `pressmove` event of the shape.

Inside the next anonymous function for the `pressmove` event, we have two lines of code. Both calculate the position of the mouse cursor and then alter the coordinates of the target shape. `ev.stageX` and `ev.stageY` always give you the coordinates of the mouse cursor within the stage. Therefore, using these properties, we can change the coordinates of the target shape correctly.

Everything is ready now, but there's one last step we should perform to complete the challenge. As we learned earlier in EaselJS, we should call the `update` function in order to update the stage after making any changes in objects in the `stage` event. In the drag-and-drop example, we are changing the coordinates of the target shape continuously. Therefore, we also have to update the `Stage` event continuously; the question is how. The answer to this question is that we have to call `stage.update()` on the call for each event as follows:

```
stage.update();
```

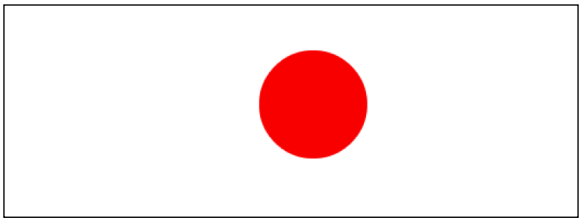
EaselJS will update the stage event after each change to the coordinates of the circle.

The complete example

Here you can see the whole source code for creating a simple drag-and-drop interaction:

```
stage = new createjs.Stage("demoCanvas");
stage.mouseMoveOutside = true;

var circle = new createjs.Shape();
circle.graphics.beginFill("red").drawCircle(100, 100, 50);
stage.addChild(circle);

circle.on("mousedown", function(evt) {
    var offset = {
        x: evt.target.x - evt.stageX,
        y: evt.target.y - evt.stageY
    };

    circle.on("pressmove", function(ev) {
        ev.target.x = ev.stageX + offset.x;
        ev.target.y = ev.stageY + offset.y;
        stage.update();
    });
});

stage.update();
```

After running the example, you should see a red circle, as shown in the following screenshot. By clicking and dragging the circle, the coordinates of the circle will be changed.



Summary

In this chapter, we have learned how to work with the `mousedown` and `pressmove` events, how to change object properties, and also how to update the stage event continuously using EaselJS features. With a combination of all the mentioned features, we can build an animation or say drag-and-drop interactions with CreateJS.

In the next chapter, we will discuss working with the animation and transformation of objects with CreateJS to develop awesome animations in browsers.

4
Performing Animation and Transforming Function

Now that you have learned how to work with events and callbacks in the previous chapter, it's time to move ahead and work with the Animation and Transforming function. This chapter is supposed to be more interactive than the previous chapter, because we are going to talk about and use the Animation and Transforming function of CreateJS. In this chapter, we use *TweenJS* and *EaselJS* to create basic animation in browsers.
You can then use these functions to create more innovative animations.

In this chapter, we will cover the following topics:

- Creating animations with CreateJS
- Understanding TweenJS
- Understanding the TweenJS API
- Creating simple animations
- Transforming shapes
- Understanding Sprite Sheets
- Developing animations using Sprite Sheet

Creating animations with CreateJS

As you may already know, creating animations in web browsers during web development is a difficult job because you have to write code that has to work in all browsers; this is called *browser compatibility*. The good news is that CreateJS provides modules to write and develop animations in web browsers without thinking about browser compatibility. CreateJS modules can do this job very well and all you need to do is work with *CreateJS API*.

Understanding TweenJS

TweenJS is one of the modules of CreateJS that helps you develop animations in web browsers. We will now introduce TweenJS.

> *The TweenJS JavaScript library provides a simple but powerful tweening interface. It supports tweening of both numeric object properties and CSS style properties, and allows you to chain tweens and actions together to create complex sequences. – TweenJS API Documentation*

For more information on TweenJS, please refer to the official documentation at:

`http://www.createjs.com/Docs/TweenJS/modules/TweenJS.html`

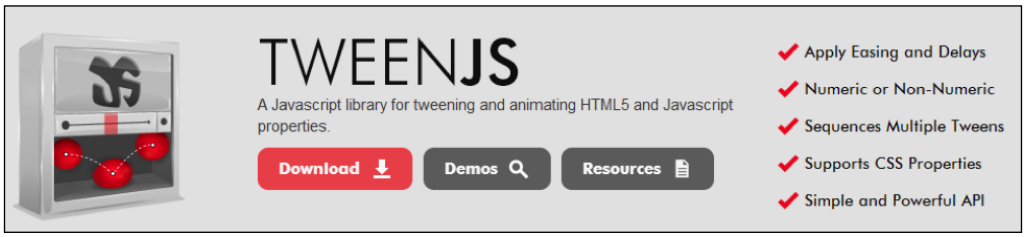
What is tweening?

Let us understand precisely what tweening means:

> *Inbetweening or tweening is the process of generating intermediate frames between two images to give the appearance that the first image evolves smoothly into the second image. – Wikipedia*

For more information on tweening, visit:

`http://en.wikipedia.org/wiki/Tweening`



The same as other CreateJS subsets, TweenJS contains many functions and methods; however, we are going to work with and create examples for specific basic methods, based on which you can read the rest of the documentation of TweenJS to create more complex animations.

Understanding API and methods of TweenJS

In order to create animations in TweenJS, you don't have to work with a lot of methods. There are a few functions that help you to create animations. Following are all the methods with a brief description:

- `get`: It returns a new tween instance.
- `to`: It queues a tween from the current values to the target properties.
- `set`: It queues an action to set the specified properties on the specified target.
- `wait`: It queues a wait (essentially an empty tween).
- `call`: It queues an action to call the specified function.
- `play`: It queues an action to play (un-pause) the specified tween.
- `pause`: It queues an action to pause the specified tween.

The following is an example of using the Tweening API:

```
var tween = createjs.Tween.get(myTarget).to({x:300},400).
set({label:"hello!"}).wait(500).to({alpha:0,visible:false},1000).
call(onComplete);
```

The previous example will create a tween, which:

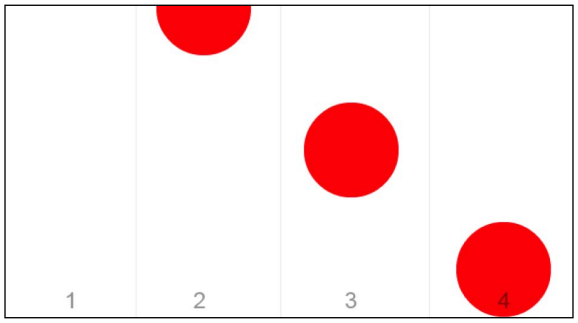
- Tweens the target to an `x` value of `300` with duration `400`ms and sets its `label` to `hello!`.
- Waits `500`ms.
- Tweens the target's `alpha` property to `0` with duration `1`s and sets the `visible` property to `false`.
- Finally, calls the `onComplete` function.

Creating a simple animation

Now, it's time to create our simplest animation with TweenJS. It is a simple but powerful API, which gives you the ability to develop animations with *method chaining*.

Scenario

The animation has a red ball that comes from the top of the `Canvas` element and then drops down.



In the preceding screenshot, you can see all the steps of our simple animation; consequently, you can predict what we need to do to prepare this animation.
In our animation,
we are going to use two methods: `get` and `to`.

The following is the complete source code for our animation:

```
var canvas = document.getElementById("canvas");
var stage = new createjs.Stage(canvas);

var ball = new createjs.Shape();
ball.graphics.beginFill("#FF0000").drawCircle(0, 0, 50);

ball.x = 200;
ball.y = -50;

var tween = createjs.Tween.get(ball) to({
  y: 300
}, 1500, createjs.Ease.bounceOut);

stage.addChild(ball);
createjs.Ticker.addEventListener("tick", stage);
```

In the second and third line of the JavaScript code snippet, two variables are declared, namely; the `canvas` and `stage` objects. In the next line, the `ball` variable is declared, which contains our `shape` object. In the following line, we drew a red circle with the `drawCircle` method (we have discussed about the `drawCircle` method in the previous chapter). Then, in order to set the coordinates of our shape object outside the viewport, we set the `x` axis to `-50` px.

After this, we created a `tween` variable, which holds the `Tween` object; then, using the TweenJS method chaining, the `to` method is called with duration of `1500` ms and the `y` property set to `300` px. The third parameter of the `to` method represents the `ease` function of `tween`, which we set to `bounceOut` in this example.

In the following lines, the `ball` variable is added to `Stage` and the `tick` event is added to the `Ticker` class to keep `Stage` updated while the animation is playing. You can also find the `Canvas` element in line 30, using which all animations and shapes are rendered in this element.

Transforming shapes

CreateJS provides some functions to transform shapes easily on `Stage`.
Each `DisplayObject` has a `setTransform` method that allows the transforming of a `DisplayObject` (like a circle).

The following shortcut method is used to quickly set the transform properties on the display object. All its parameters are optional. Omitted parameters will have the default value set.

```
setTransform([x=0] [y=0] [scaleX=1] [scaleY=1] [rotation=0] [skewX=0]
[skewY=0] [regX=0] [regY=0])
```

This was taken from:

`http://www.createjs.com/Docs/EaselJS/classes/Shape.html#method_setTransform`

Furthermore, you can change all the properties via `DisplayObject` directly (like `scaleY` and `scaleX`) as shown in the following example:

```
displayObject.setTransform(100, 100, 2, 2);
```

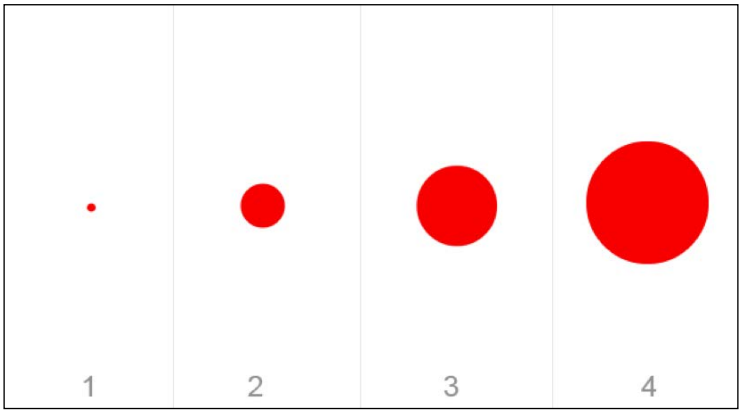
An example of Transforming function

As an instance of using the shape transforming feature with CreateJS, we are going to extend our previous example:

```
var angle = 0;
window.ball;
var canvas = document.getElementById("canvas");
var stage = new createjs.Stage(canvas);

ball = new createjs.Shape();
```

```
ball.graphics.beginFill("#FF0000").drawCircle(0, 0, 50);

ball.x = 200;
ball.y = 300;

stage.addChild(ball);

function tick(event) {
  angle += 0.025;
  var scale = Math.cos(angle);

  ball.setTransform(ball.x, ball.y, scale, scale);
  stage.update(event);
}

createjs.Ticker.addEventListener("tick", tick);
```

In this example, we have a red circle, similar to the previous example of tweening. We set the coordinates of the circle to `200` and `300` and added the circle to the `stage` object. In the next line, we have a `tick` function that transforms the shape of the circle. Inside this function, we have an `angle` variable that increases with each call. We then set the `ballX` and `ballY` coordinates to the cosine of the `angle` variable. The transforming done is similar to the following screenshot:



This is a basic example of transforming shapes in CreateJS, but obviously you can develop and create better transforming by playing with a shape's properties and values.

Understanding Sprite Sheet

In this section, we will discuss about the EaselJS feature to make an animation using a series of images. This feature is called **Sprite Sheet**. A Sprite Sheet combines a series of images or frames of an animation to produce 2D or 3D animation. For instance, if you want to animate a hero that is walking, we can combine all frames of the walking character into a single image, and then make the animation using the Sprite Sheet feature of EaselJS.

The following is a series of images (animation frames) that are combined into a single image:

In the next section, you will learn how to use the Sprite Sheet feature to develop animations.

Developing animations using Sprite Sheet

Let's start by understanding the `SpriteSheet` class. This class is used to initialize the Sprite Sheet feature and encapsulate its properties and configurations. After creating the `SpriteSheet` class, we can use its methods to control the animation.

The basic configuration of this class has three mandatory properties:

- The image or images to use for animation frames.
- The position of each image, which can be defined using a single value for all frames or even with an individual configuration for each frame.
- The representation of the animation, which can be defined by a start and end frame or with individual values for each frame.

The following is a code snippet defines the configuration for the `SpriteSheet` class:

```
data = {
  // list of images or image URIs to use. SpriteSheet can handle
preloading.
  // the order dictates their index value for frame definition.
```

```
images: [image1, "path/to/image2.png"],

  // the simple way to define frames, only requires frame size because
frames are consecutive:
  // define frame width/height, and optionally the frame count and
registration point x/y.
  // if count is omitted, it will be calculated automatically based on
image dimensions.
frames: {width:64, height:64, count:20, regX: 32, regY:64},

// OR, the complex way that defines individual rects for frames.
  // The 5th value is the image index per the list defined in "images"
(defaults to 0).
frames: [
  // x, y, width, height, imageIndex, regX, regY
  [0,0,64,64,0,32,64],
  [64,0,96,64,0]
],

  // simple animation definitions. Define a consecutive range of
frames (begin to end inclusive).
  // optionally define a "next" animation to sequence to (or false to
stop) and a playback "speed".
animations: {
  // start, end, next, speed
run: [0,8],
jump: [9,12,"run",2]
}

// the complex approach which specifies every frame in the animation
by index.
animations: {
  run: {
  frames: [1,2,3,3,2,1]
  },
  jump: {
    frames: [1,4,5,6,1],
    next: "run",
    speed: 2
  },
stand: { frames: [7] }
  }
}
```

The following is a sample configuration for the `SpriteSheet` class:

```
var data = {
  images: ["sprites.jpg"],
  frames: { width:50, height:20 },
  animations: {
    run:[0,4],
    jump:[5,8,"run"]
  }
};
```

This is explained more in detail at: `http://www.createjs.com/Docs/EaselJS/classes/SpriteSheet.html`

Now, we will develop a simple walking animation using Sprite Sheets. The following is our sprite image that will be used for the animation frames:

Sprite image

The next step is to configure our `SpriteSheet` class. The following is the configuration:

```
var data = {
  "animations":
  {
    "run": [0, 15]
  },
  "images": ["running.png"],
  "frames": {
    "height": 70,
    "width": 51,
    "regX": 0,
    "regY": 0,
    "count": 16
  }
};
```

We have a total of 16 frames for our animation; hence, the *run* frameset that defines the frames of the animation, starts from `0` and extends till `15`. We defined the path of the sprite image. We then defined the configuration of the frames with height `70`, width `51` (this is the width of each individual image), and a count of 16 that refers to the number of frames.

The following is the complete source code for the animation:

```
stage = new createjs.Stage(document.getElementById("canvas"));

varss = new createjs.SpriteSheet({
  "animations":
  {
    "run": [0, 15]
  },
  "images": ["running.png"],
  "frames": {
    "height": 70,
    "width": 51,
    "regX": 0,
    "regY": 0,
    "count": 16
  }
});

var grant = new createjs.Sprite(ss, "run");

stage.addChild(grant);
createjs.Ticker.setFPS(40);
createjs.Ticker.addEventListener("tick", stage);
```

As seen in previous examples, first we defined the stage using the `Stage` class. After that, the `SpriteSheet` class was initiated using the configuration, and then we passed the object to the `Sprite` class to start the animation. The second parameter for the `Sprite` class defines the starting frameset for animation. Finally, we added the `Sprite` object to the stage with the `addChild` method.

It's important to add the `tick` event to the `Ticker` class and pass the `Stage` object to it to start the animation; otherwise, you will see a blank screen. Furthermore, using the `Ticker` class and `setFPS` method, we can control the ratio of rendering the animation.

The following image shows a preview of our Sprite Sheet example:

Summary

In this chapter, we have learned how to develop and create animations with TweenJS and EaselJS. We discussed in detail about working with chaining methods in TweenJS, callback functions in animations, and also how to change a shape's properties with TweenJS functions. We then learned how to transform with EaselJS in order to change a shape's properties like rotation or scale. We then went on to learn about utilizing the Sprite Sheet feature to create animated characters.

In the next chapter, we will discuss about caching techniques in CreateJS and how to improve the performance of applications using cache.

5
Utilizing Caching in EaselJS

As you know, after developing an animation, it's very important to work on the performance issues to make it smooth. The performance of rendering animations varies between browsers but there are some techniques like caching that can simply improve the performance. In this chapter, we will learn how to make better and smooth animations or drawings using the *EaselJS* caching system of `DisplayObject`.

In this chapter, we will cover the following topics:

- Exploring the caching feature of EaselJS
- Understanding the cache method
- Example of using cache
- Using cache in animations
- Using cache with Bitmap

Exploring the caching feature of EaselJS

In EaselJS, when you have a shape, or even better, an instance of the `DisplayObject` that doesn't change frequently, it's better to use the *cache* function to cache it in a different `Canvas` element. This technique will help you use EaselJS in the drawing process to animate and render animations or drawings smoothly, as the shapes don't need to be rendered with every tick.

It is basically the main idea of using the `cache` method in the `DisplayObject` class. All you need to do is learn more about using the cache method in EaselJS. In further sections, we will go through methods, their usage, and how to create animations with caching.

Understanding the cache method

In order to understand how caching a `DisplayObject` works in EaselJS, we take the example of an imaginary canvas area so that the cached elements get rendered into it and each time you need to update the target shape, you call the `cache` method again.

You can see the definition of the `cache` method inside the `DisplayObject` class in the following piece of code:

```
cache (x, y, width, height, [scale=1])
```

> *Draws the display object into a new canvas, which is then used for subsequent draws. For complex content that does not change frequently (ex. a Container with many children that do not move, or a complex vector Shape), this can provide for much faster rendering because the content does not need to be re-rendered each tick. The cached display object can be moved, rotated, faded, etc freely, however if its content changes, you must manually update the cache by calling* `updateCache()` *or* `cache()` *again. You must specify the cache area via the x, y, w, and h parameters. This defines the rectangle that will be rendered and cached using this display object's coordinates.*

This has been taken from:

`http://www.createjs.com/Docs/EaselJS/classes/DisplayObject.html#method_cache`

As you can see, the `cache` method accepts four mandatory and one optional parameter. The first and second parameters define the *coordinate* of the cache area; the third and fourth parameters define the *width* and *height* of the cache area. Using the last parameter, you can define the *scale of shape* inside the cache area. By default it is set as `1`, but you can change it.

Example of using cache

Now it's time to see an example of using the `cache` method in EaselJS. The following piece of code uses the `cache` method to render a circle into a canvas element:

```
var shape = new createjs.Shape();
shape.graphics.beginFill("#ff0000").drawCircle(0, 0, 25);
shape.cache(-25, -25, 50, 50);
```

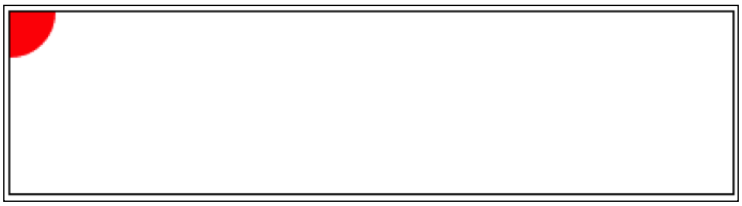
In the first line, we created a shape using the `Shape` class, filled it with red color, and then rendered it at `(0, 0)` with a radius of `25`. In the third line, you will notice the use of the `cache` method. In this line, a *cache area* is created at `-25, -25` with a width and height of `50`.

In order to update the target shape (the shape variable in the above example), you need to call the `cache` or `updateCache` method with all new parameters again.

The complete source code and result is as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html;
   charset=UTF-8">
<title>Cache method in EaselJS</title>
<script type='text/javascript' src='createjs.js'></script>

<script type='text/javascript'>
window.onload=function() {
  var canvas = document.getElementById("testCanvas");
  var stage = new createjs.Stage(canvas);

  var shape = new createjs.Shape();
  shape.graphics.beginFill("#ff0000").drawCircle(0, 0, 25);
  shape.cache(-25, -25, 50, 50);

  stage.addChild(shape);
  stage.update();
}
</script>

</head>
<body>
<canvas id="testCanvas" width="400" height="100" style="border:
1px solid black;"></canvas>
</body>
</html>
```

The above source code is the completed example of using the cache method in EaselJS. The result of this source code is as shown in the following screenshot:

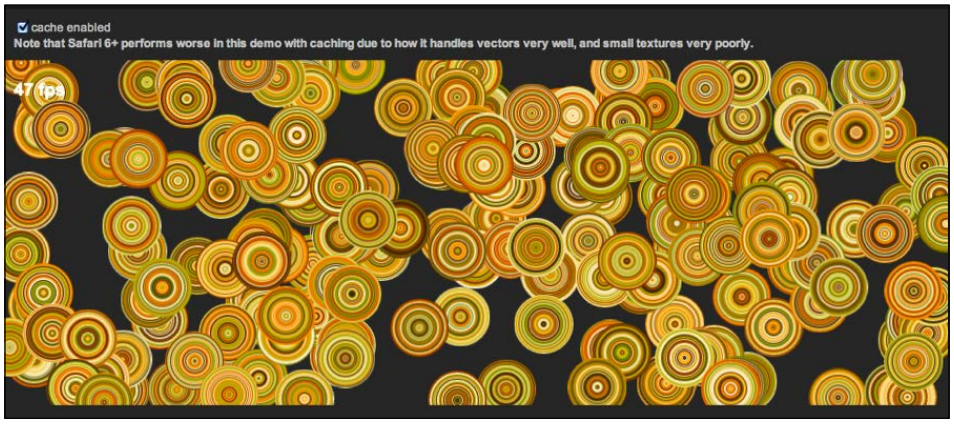


Using cache in complex shapes and animations

Caching in EaselJS comes in handy when you have a complex shape or animation in your canvas. In most cases, complex shapes with details shouldn't get rendered in every tick. So you can simply use the cache method to cache it in order to prevent rendering overhead.

Now we will see a complex example from the EaselJS library (source: `https://github.com/CreateJS/EaselJS/blob/master/examples/Cache.html`) and the effect of using a cache technique on that. In this animation, we will create about *200* complex circles and move them on each tick. There is a checkbox on the page that controls the enabling or disabling of caching for all shapes inside the canvas area using `cache` and `uncache` methods.

The following screenshot shows us a preview of our animation (notice the checkbox):



There are three main functions that handle this animation and all the logics; `init`, `tick`, and `toggleCache`. We will discuss each one of them separately.

In the first lines of the code, we will use the following variables:

```
var canvas;
var stage;
var shape;
varcircleRadius= 30;
var rings = 30;
```

The first variable holds the canvas element, the second one is used for the `Stage` object, the `shape` variable is used to draw shapes on the stage, and `circleRadius` and `rings` are used for basic settings of circles. `circleRadius` is used to define the radius of each circle and `rings` is used to define the number of rings inside each circle.

The following code shows the the basic `init` method that draws all shapes and prepares the stage:

```
Function init() {
  // create a new stage and point it at our canvas:
  canvas = document.getElementById("testCanvas");
  stage = new createjs.Stage(canvas);

  // create a large number of slightly complex vector shapes,
and give them random positions and velocities:

  var colors = ["#828b20", "#b0ac31", "#cbc53d", "#fad779", "#f9e4ad",
"#faf2db", "#563512", "#9b4a0b", "#d36600", "#fe8a00", "#f9a71f"];

  for(var i= 0; i< 200; i++) {
    shape = new createjs.Shape();
      for(var j = rings; j > 0; j--) {
        shape.graphics.beginFill(colors
        [Math.random() * colors.length | 0])
        .drawCircle(0, 0, circleRadius * j / rings);
    }
    shape.x = Math.random() * canvas.width;
    shape.y = Math.random() * canvas.height;
    shape.velX = Math.random() * 10 - 5;
    shape.velY = Math.random() * 10 - 5;

    stage.addChild(shape);
  }

  // add a text object to output the current FPS:
```

```
  fpsLabel = new createjs.Text("-- fps", "bold 18px Arial", "#FFF");
  stage.addChild(fpsLabel);
  fpsLabel.x = 10;
  fpsLabel.y = 20;

  // start the tick and point it at the window so we can do some
work before updating the stage:
  createjs.Ticker.addEventListener("tick", tick);
  createjs.Ticker.setFPS(50);
}
```

This code is used to create the stage and all shape objects. In lines 3 and 4, the `Stage` object is created. In line 8, we defined random colors for circle rings. After that, we have used a *for-loop*, which draws 200 different circles with random positions on the stage. We have another for-loop to draw rings inside the circles in line 12.

In our animation, we have a label that indicates the **Frames per Second** (**FPS)** rate on each tick. So, in lines 28 to 31, we have defined our label properties. In line 34, the `Ticker` class is created and in line 36, the FPS of the animation is set to `50`.

After the `init` function, we have a tick function that will be called on each tick by EaselJS:

```
function tick(event) {
  var w = canvas.width;
  var h = canvas.height;
  var l = stage.getNumChildren() - 1;

  // iterate through all the children and move them according to
their velocity:
  for(var i= 1; i< l; i++) {
    var shape = stage.getChildAt(i);
    shape.x = (shape.x + shape.velX + w) % w;
    shape.y = (shape.y + shape.velY + h) % h;
  }
  fpsLabel.text = Math.round(createjs.Ticker.getMeasuredFPS()) +
    " fps";
  // draw the updates to stage:
  stage.update(event);
}
```

The main job of the above `init` function is to change the position of all the circles on the stage on each tick, set the current FPS rate to the label, and then update the stage. The reason there is `-1` in line 4 is to exclude the label object from `children`; keep in mind that we only need to change the position of all the circles.

The last function is the `toggleCache` function. This method enables or disables caching for all circles:

```
function toggleCache(value) {
  // iterate all the children except the fpsLabel, and set up the
cache:
  var l = stage.getNumChildren() - 1;
  for(var i= 0; i< l; i++) {
    var shape = stage.getChildAt(i);
    if (value) {
      shape.cache(-circleRadius, -circleRadius, circleRadius * 2,
         circleRadius * 2);
    } else {
      shape.uncache();
    }
  }
}
```

This function is called only when you check or uncheck the checkbox on the page so it enables or disables caching for all circle objects on `stage`. There is a for-loop that iterates over all circle shapes and calls the `cache` or `uncache` method according to the status of the checkbox. Consequently, the caching for circle shapes will be enabled or disabled.

By clicking on the checkbox, you can obviously see that the animation rendering gets smoother when caching is enabled.

Finally, you can find the complete source code of our animation on the Packt website.

Caching Bitmap

In this section, we will utilize the `cache` method with Bitmap and `AlphaMaskFilter` to develop a reflection effect in EaselJS. The target is to load an image and create a `Bitmap` class to draw the image. Then, clone the `Bitmap` image, change the rotation and add a gradient background, and use `AlphaMaskFilter` to create the reflection effect.

The following screenshot is a preview of the result:

The following is the source code of this example:

```
function init() {
  var canvas = document.getElementById("canvas");
  var stage = new createjs.Stage(canvas);

  var image = new Image();
  image.src = "easeljs.png";

  //wait to load the image
  image.onload = function(evt) {
    var bitmap = new createjs.Bitmap(evt.target);
    var width = bitmap.image.width;
    var height = bitmap.image.height;

    //clone the existing bitmap to use as reflection
    var reflectBitmap = bitmap.clone();
    reflectBitmap.regY = height;
    reflectBitmap.rotation = 180;

    //to add a padding from the main bitmap
    reflectBitmap.y = height + 2;
    reflectBitmap.scaleX = -1;

    var maskShape = new createjs.Shape();
```

```
        var graphics = maskShape.graphics;
        //add reflection effect
        graphics.beginLinearGradientFill(["rgba(255, 255, 255, 0)"
          , "rgba(255, 255, 255, 0.6)"], [0.5, 1], 0, 10, 0, height);
        graphics.drawRect(0, 0, width, height);
        graphics.endFill();

        maskShape.cache(0, 0, width, height);

        reflectBitmap.filters =
          [new createjs.AlphaMaskFilter(maskShape.cacheCanvas)];
        reflectBitmap.cache(0, 0, width, height);

        //add both pictures
        stage.addChild(bitmap);
        stage.addChild(reflectBitmap);
        stage.update();
      }
    }
```

As seen in previous examples, firstly, we created the `Stage` class. Then, in order to load the image, we used the `Image` class and passed the address of the image to the `src` property. The `Image` class has an `onload` event, which helps developers know when the image is loaded completely. We used this event to execute other parts of the application correctly.

After that, we used a `Bitmap` class and passed the image parameter from the `Image` class to it. Because we need the width and height of the picture, we saved them into two different variables called `width` and `height`. At this moment, we have the first picture but we should have one more picture to create the reflection effect. So, we used the `clone` function to duplicate the image. In order to change the rotation, scale, and coordination of the second image, we changed the `regY`, `rotation`, `y`, and `scaleX` properties.

After that, a new shape is created using the `Shape` class. This is the mask layer that will be used for the `AlphaMaskFilter`. Then, we added a linear background to it to create the reflection effect and cached it using the `cache` function. Finally, an `AlphaMaskFilter` is added to the second picture (a cloned `Bitmap` class) and this shape is used as the mask layer. The second picture is also cached again. Both pictures are added to `Stage` using the `addChild` function and `Stage` is also updated with the `update` function.

Summary

In this chapter, we have learned how to work with the `cache` method in EaselJS to create better drawings and animations on the canvas. The reason we use caching in EaselJS is to provide better and faster animation rendering in browsers and also use fewer resources while rendering the animations or drawings. In the next section, we discussed about using the cache method with the `Bitmap` class to create the reflection effect.

In the next chapter, we will talk about applying filters in canvas with EaselJS, which is one of the best features of EaselJS, and you can make fantastic stuff with this feature.

6
Using Filters in EaselJS

One of the great features of CreateJS is the ability to play with filters and apply various filters to pictures and shapes easily. CreateJS makes it easy by providing a `Filter` class and `filters` property for `DisplayObject`; accordingly, we can simply create instances of the `Filter` class and apply them to the objects. In this chapter, we will have a look at CreateJS filters and some basic examples of working with the `Filter` class.

In this chapter, we will cover the following topics:

- Understanding the `Filter` class
- How to use built-in EaselJS filters

Understanding the Filter class

EaselJS comes with a `Filter` class, which is the base class for all other filters, and other filter classes should inherit from this class. Filters need to be applied to objects that have been cached using the `cache` method; after that, if the object gets changed again, we should use the `cache` or `updateCache` methods to update the shape.

The following is an example of applying filters to an object:

```
    /* Add canvas and stage */
    var canvas = document.getElementById("canvas");
    var stage = new createjs.Stage(canvas);

/* create and draw the shape */
var circle = new createjs.Shape();
circle.graphics.beginFill("red").drawCircle(50, 50, 40);

/* add the blur filter to filters property */
```

```
circle.filters = [new createjs.BlurFilter(5, 5, 10)];
/* cache the shape to apply the filter */
circle.cache(0, 0, 100, 100);
/* add shape to stage and update */
stage.addChild(circle);
stage.update();
```

In the first line, we have created a shape object; in the next line, we have created a circle with the color red. The next line contains the filter configurations, and in the last line we have cached the object using the `cache` function.

Using the `cache` method in examples with filtering will not only boost performance but will also apply the filter to the shape and make it work.

EaselJS contains several basic filters, such as `blur` or `color` filters, that you can use easily. Here is a list of built-in filters:

- `AlphaMapFilter`: This is used to map a grayscale image to the alpha channel of a display object.
- `AlphaMaskFilter`: This is used to map the alpha channel of an image to the alpha channel of a display object.
- `BlurFilter`: This applies vertical and horizontal blur to a display object.
- `ColorFilter`: This color transforms a display object.
- `ColorMatrixFilter`: This transforms an image using a `ColorMatrix`.

In the next section, we will discuss all these filters in detail.

Using the AlphaMapFilter class

The `AlphaMapFilter` is an built-in filter is used for applying a grayscale alpha map image (or canvas) to the target. To be clearer, the alpha channel of the result will be copied from the red channel of the map and the RGB channels will be copied from the target object.

> *Generally, it is recommended that you use* `AlphaMaskFilter`*, because it has much better performance.*

This has been taken from: `http://www.createjs.com/Docs/EaselJS/classes/AlphaMapFilter.html`

The following code snippet is the definition for this class:

```
AlphaMapFilter (alphaMap)
```

The parameters are as follows:

`alphaMap | Image | HTMLCanvasElement`

We have used the grayscale image or the canvas as the alpha channel. It should be of the same dimensions as the target.

The following code is an example of using the `AlphaMapFilter` class:

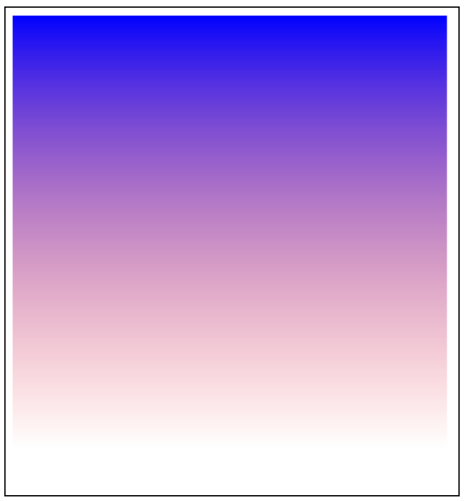
```
/* Add canvas and stage */
var canvas = document.getElementById("canvas");
var stage = new createjs.Stage(canvas);

/* Create filter */
var box = new createjs.Shape();
box.graphics.beginLinearGradientFill(["#0000ff", "#ff0000"]
  , [1, 0], 0, 0, 0, 300)
box.graphics.drawRect(0, 0, 300, 300);
box.cache(0, 0, 300, 300);

/* create the second shape */
var box2 = new createjs.Shape();
```

```
box2.graphics.beginLinearGradientFill(["#0000ff", "#ff0000"]
  , [0, 1], 0, 0, 0, 300);
box2.graphics.drawRect(0, 0, 300, 300);

/* Add filter to box2 */
box2.filters = [
  new createjs.AlphaMapFilter(box.cacheCanvas)
];
/* and finally, cache the shape to apply changes */
box2.cache(0, 0, 300, 300);

/* Add bitmap to stage and update stage */
stage.addChild(box2);
stage.update();
```

In the first few lines of the code, we created a rectangle created with a linear gradient and then cached the object using `cache` method. The reason for caching the object is to use it in the filter parameters.

Then, we created the `box2` variable; it's our parent shape. This shape is the same as the first one, but the gradient color is different. We have changed the colors for the start and end of the linear gradient. Afterward, we added `AlphaMapFilter` to the `box2` filters and the `box` variable as the parameter of the filter. Then, in to order apply the filter to the shape, we cached the shape using the `cache` method and added it to the stage.

A preview of the previous example is shown in the following image:



Using the AlphaMaskFilter class

This filter is similar in usage to the `AlphaMapFilter` class, but we will briefly talk about this filter as well. As per the CreateJS documentation, it's recommended that you use this filter instead of `AlphaMapFilter` because it has much better performance.

`AlphaMaskFilter` applies the alpha mask from the mask image (or canvas) to the target. The alpha channel of the result will be derived from the mask, and the RGB channels will be copied from the target object.

Here is how we define the `AlphaMaskFilter` class:

```
AlphaMaskFilter (mask)
```

The parameters in this code snippet are as follows:

`mask | Image`

This class is an instance of the `Image` class.

Here is an example of using this filter:

```
/* Declare variables */
var canvas, stage, img;

function init() {
  /* Add canvas and stage */
  canvas = document.getElementById("canvas");
  stage = new createjs.Stage(canvas);

  /* Load image */
  img = new Image();
  img.onload = handleimg;   //function that's called once image
has loaded
  img.src = "targetImg.png";  //image url
}

function handleimg() {
  /* Create mask layer */
  var box = new createjs.Shape();
  box.graphics.beginLinearGradientFill(["#000000", "rgba
    (0, 0, 0, 0)"], [0, 1], 0, 0, 0, 200)
  box.graphics.drawRect(0, 0, 200, 200);
  box.cache(0, 0, 200, 200);

  /* Create bitmap */
```

```
    var bmp = new createjs.Bitmap(img);

    /* Add filter to bitmap */
    bmp.filters = [
       new createjs.AlphaMaskFilter(box.cacheCanvas)
    ];
      bmp.cache(0, 0, 200, 200);

    /* Add bitmap to stage and update stage */
    stage.addChild(bmp);
    stage.update();
  }
```

As you can see, the usage of this filter is almost the same as `AlphaMapFilter`.

The example source code is divided into two functions, `init` and `handleimg`. In order to load the image properly, we used the `Image` class and the `onload` event. We then used the `handleimg` function for the `onload` event callback function.

Inside `init` function, `stage` class is created. We also configured the `Image` class and assigned the `handleimg` function to the `onload` event. A major part of the example's source code is inside the `handleimg` function. In the first few lines, we created a rectangle with a linear gradient background. The reason for using the `rgba` function to define color is to change the alpha channel of the gradient so that the filter will derive this alpha channel for the final result. Finally, we cached the shape using the `cache` method.

Then, we loaded an image using the `Bitmap` function and also applied it to the `bmp` variable with the `filters` property. We also cached this image in order to apply the filter changes.

The following screenshot illustrates the result of our example:

Implementing the BlurFilter class

The Blur filter is one of usable filters for creating innovative animations and drawings. In this section, we will have a discussion about using the `BlurFilter` class.

This filter applies a **blur box** to `DisplayObject`.

This filter is CPU intensive, particularly if the quality is set to higher than 1.

Here is the definition of the `BlurFilter` class and its parameters:

```
BlurFilter ([blurX=0] [blurY=0] [quality=1])
```

The parameters included in this code snippet are as follows:

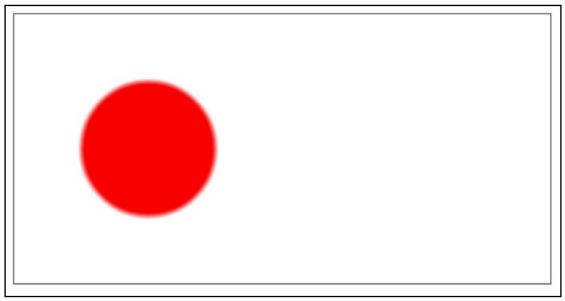
- `[blurX=0] | Number`: It is an optional parameter. It is used to set the horizontal blur radius in pixels.
- `[blurY=0] | Number`: It is an optional parameter. It is used to set the vertical blur radius in pixels.
- `[quality=1] | Number`: It is an optional parameter. It is used to set the number of blur iterations.

Here is an example of using the Blur filter with a red circle:

```
/* Add canvas and stage */
var canvas = document.getElementById("canvas");
var stage = new createjs.Stage(canvas);

/* create circle shape */
var shape = new createjs.Shape().set({x:100,y:100});
shape.graphics.beginFill("#ff0000").drawCircle(0,0,50);

/* create blur filter and add to shape */
var blurFilter = new createjs.BlurFilter(5, 5, 1);
shape.filters = [blurFilter];

/* add getbounds to give spread effect to blur */
var bounds = blurFilter.getBounds();
shape.cache(-50+bounds.x, -50+bounds.y, 100+bounds.width,
100+bounds.height);

/* add shape to stage and update */
stage.addChild(shape);
stage.update();
```

In the first line, we have the `shape` variable, which is the variable for our shape, a circle. In the next line, we filled the circle with the red color and finished drawing the shape using the `drawCircle` function.

Then, we created the blur filter using three parameters and applied it to the `shape` object with the `filters` property. In order to find the dimension of the caching area, we used the `getBounds` function because, as you know, the blur filter has some more padding after applying the `getBounds` function.



Utilizing the ColorFilter class

This filter applies a color transform to `DisplayObject`. This filter comes handy when you need to play with colors in EaselJS.

In the following code snippet, you can see the definition of this filter:

```
ColorFilter ([redMultiplier=1] [greenMultiplier=1]
   [blueMultiplier=1] [alphaMultiplier=1] [redOffset=0]
   [greenOffset=0] [blueOffset=0] [alphaOffset=0])
```

The various parameters in this code snippet are as follows:

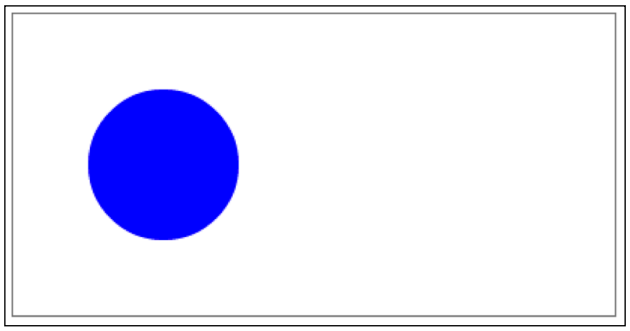
- `[redMultiplier=1]` – `Number`: It is an optional parameter. It sets the value to multiply with the red channel. The value should be between 0 and 1.
- `[greenMultiplier=1]` – `Number`: It is an optional parameter. It sets the value to multiply with the green channel. The value should be between 0 and 1.
- `[blueMultiplier=1]` – `Number`: It is an optional parameter. It sets the value to multiply with the blue channel. The value should be between 0 and 1.
- `[alphaMultiplier=1]` – `Number`: It is an optional parameter. It sets the value to multiply with the alpha channel. The value should be between 0 and 1.
- `[redOffset=0]` – `Number`: It is an optional parameter. It sets the value to add to the red channel after it has been multiplied. The value should be between -255 and 255.

- `[greenOffset=0]` – `Number`: It is an optional parameter. It sets the value to add to the green channel after it has been multiplied. The value should be between -255 and 255.
- `[blueOffset=0]` – `Number`: It is an optional parameter. It sets the value to add to the blue channel after it has been multiplied. The value should be between -255 and 255.
- `[alphaOffset=0]` – `Number`: It is an optional parameter. It sets the value to add to the alpha channel after it has been multiplied. The value should be between -255 and 255.

Here is an example of using this filter:

```
/* Add canvas and stage */
var canvas = document.getElementById("canvas");
var stage = new createjs.Stage(canvas);

var shape = new createjs.Shape().set({x:100,y:100});
shape.graphics.beginFill("#ff0000").drawCircle(0,0,50);

shape.filters = [
  new createjs.ColorFilter(0,0,0,1, 0,0,255,0)
];
shape.cache(-50, -50, 100, 100);
/* add shape to stage and update */
stage.addChild(shape);
stage.update();
```

In this example, we created a red circle and then changed its color to blue using `ColorFilter`. This is done by multiplying all the channels with 0 (except for the alpha channel, which is set to 1) and then adding the value 255 to the blue channel and 0 to the other channels.



Using the ColorMatrixFilter class

With this filter, you can play with complex color operations, such as saturation, brightness, or inversion. This filter uses the `ColorMatrix` class to perform the action.

The following code snippet defines this class:

```
ColorMatrixFilter (matrix)
```

The parameters present in this code snippet are as follows:

- `matrix- Array`: A 4x5 matrix describing the color operation to perform using the `ColorMatrix` class.

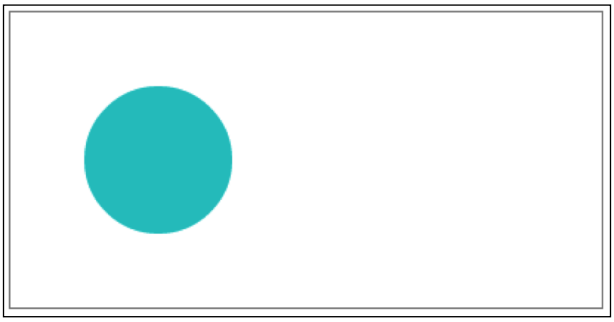
Here is an example of using this filter:

```
/* Add canvas and stage */
var canvas = document.getElementById("canvas");
var stage = new createjs.Stage(canvas);

var shape = new createjs.Shape().set({x:100,y:100});
shape.graphics.beginFill("#ff0000").drawCircle(0,0,50);

var matrix = new createjs.ColorMatrix()
  .adjustHue(180).adjustSaturation(100);
shape.filters = [
 new createjs.
  ColorMatrixFilter(matrix)
];

shape.cache(-50, -50, 100, 100);
/* add shape to stage and update */
stage.addChild(shape);
stage.update();
```

In the preceeding example, we created a red circle and then inverted the hue and changed the saturation to 100. We started by creating the `stage` class in the first line. Then, we created a circle using the `drawCircle` function. To place the circle in the viewport of the `canvas` element, we used the `set` function to change the `x` and `y` values.

Then, we initiated the `matrix` variable using the `ColorMatrix` class. We used the `adjustHue` and `adjustSaturation` functions to change the hue and saturation of the circle. An acceptable value for `adjustHue` is between -180 to 180. This value for `adjustSaturation` is between -100 and 100. We set the hue value to 180 and saturation value to 100 in our example to see the difference better.

We applied all using the `filter` property of the `shape` variable. Finally, we cached the shape using the `cache` method and updated the stage using `update` method to apply the changes.



Summary

In this chapter, we learned how to use the built-in filters in EaselJS to change `DisplayObject` properties such as color, hue, saturation, and so on. We also discussed the definition and basic usage of the filters with interactive examples and screenshots.

In the next chapter, we will discuss the creation of a web-based painting application using all the methods and functions that we have learned up to now.

7
Developing a Painting Application

In this chapter, we will create a simple painting application using almost all the EaselJS features that we have already discussed in the previous chapters. This chapter will be more interactive and challenging, as we need to wrap up everything that we have already learned. So let's get started.

In this chapter, we will cover the following topics:

- Preparing the stage
- Understanding the `mousemove`, `mouseup`, and `mousedown` events
- Implementing each callback function

Preparing the stage

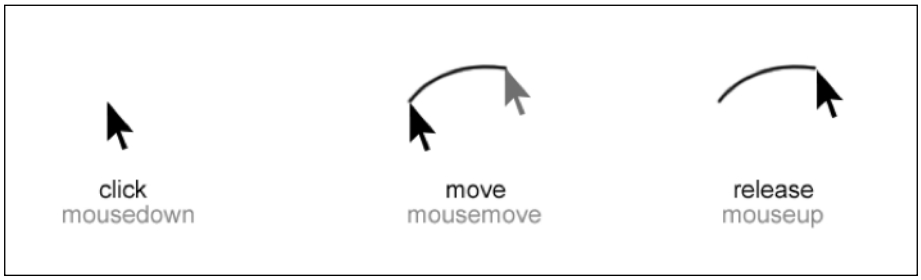
The overall functionality of this application is almost the same as drag-and-drop applications; we will use the `mousedown`, `mouseup`, and `mousemove` events to handle the painting logic. First, we will create a `stage` object and then an array of colors that will be used for the color of brushes. We will enable the *touch feature* for web browsers and for devices that support touch events. Finally, we set callback functions for the `mousedown`, `mouseup`, and `mousemove` events to handle the drawing feature and draw the lines.

The following screenshot shows a preview of our painting application:

The final source code consists of the following functions:

- `init`: It is used to create the stage and prepare other objects.
- `handleMouseDown`: It is used to handle the `mousedown` event and bind the `mousemove` event.
- `handleMouseMove`: It is used to handle the `mousemove` event and draw the line.
- `handleMouseUp`: It is used to handle the `mouseup` event and unbind the `mousemove` event to stop drawing.

The following image illustrates how the events work together to draw a line:



In further sections, we will discuss more about the source code and how it is created.

Understanding the init function

Inside the `init` function, we will setup the stage, declare the basic variables, and also attach functions to the main events like the `mousedown` event.

The following code is the source code of the `init` function:

```
function init() {
  var canvas = document.getElementById("cvs");
  var index = 0;
  var colors = ["#828b20", "#b0ac31", "#cbc53d", "#fad779", "#f9e4ad",
    "#faf2db", "#563512", "#9b4a0b", "#d36600", "#fe8a00", "#f9a71f"];

  var stage = new createjs.Stage(canvas);
  stage.autoClear = false;
  stage.enableDOMEvents(true);

  createjs.Touch.enable(stage);
  createjs.Ticker.setFPS(24);
  drawingCanvas = new createjs.Shape();

  stage.addEventListener("stagemousedown", handleMouseDown);
  stage.addEventListener("stagemouseup", handleMouseUp);

  stage.addChild(drawingCanvas);
  stage.update();
}
```

In the first line of the function's body, we have a global `canvas` variable, which refers to our `Canvas` element in the page. We have an `index` variable that holds a counter to choose a color for brushes while painting and the next line contains an array of colors. We choose one value from this array randomly using the `index` variable. After that, as seen in previous examples, we have to create a `stage` object; this is also a global variable.

After that, we set the `autoClear` property of `stage` to `false`. This property indicates whether the stage should clear the rendered elements on the canvas automatically or not. By setting this value to `false`, we can manually control the clearing.

Then, we enabled the **DOM** (**Document Object Model**) events using the `enableDOMEvents` method. This method actually enables or disables the event listener, which `stage` adds to DOM elements such as window, document, and canvas.

In the following lines, touch events and **frames per second** (**FPS**) are configured. The `setFPS` function sets the target frame rate in frames per second. This function is a member o f the `Ticker` class. The `Ticker` class is one of the major features of EaselJS that provides a centralized ticker or heartbeat and listeners can subscribe to the ticker event to be notified when time elapsed.

Then, the global variable `drawingCanvas` is initialized with a `Shape` object and it will be our painting shape. In the following events, we will use this variable to complete the drawing process.

Further, the `mousedown` and `mouseup` events are assigned to proper functions and then a painting shape is added to `stage`. There are some ways to add an event listener and one of them is using the `addEventListener` function. Pass the name of the event and the function to it. We used the `addEventListener` function in this example.

Similar to the previous examples, we have to add the shape to `stage` and update it using the `update` function. In the following lines, we added the shape to `stage` and updated it.

This is the definition of the `init` function. Actually, this function is a bootstrap function to start the painting application. Inside this function, all events to paint and draw are configured. In the following sections, we will discuss the event callback function.

Implementing the handleMouseDown function

The following code is the source code of the `handleMouseDown` function, which is used to handle the `mousedown` event:

```
function handleMouseDown(event) {
  color = colors[(index++) % colors.length];
  stroke = Math.round(Math.random() * 30 + 10);
  oldPt = new createjs.Point(stage.mouseX, stage.mouseY);
  oldMidPt = oldPt;
  stage.addEventListener("stagemousemove", handleMouseMove);
}
```

This function is used to handle the `mousedown` event and will be called after pressing the mouse button. Inside this function, we set the color and size of a stroke, and hold the current mouse position to use in the next function call.

All variables are global, so you can't see any `var` keyword before them in order to have the variables in the following function calls and other scopes. In the last line, a function also sets the `mousemove` event in order to manage the drawing lines. Actually, the `mousemove` event fires whenever the mouse cursor moves in `stage`.

The color of the brush is selected from the `colors` array that is defined in the `init` function, one after the other, using the `index` variable. What we do to select the next color from the array is increase the `index` variable and then calculate the division's remainder. With this simple hack, we can choose a value between zero and the length of the array.

The size of the brush is selected using the `random` function. The `random` function from the `Math` class in JavaScript returns a value between 0 and 1 (but not 1). By multiplying this value with 30, we can get a value between 0 and 30 (but not 30). The round function also rounds up a number in JavaScript.

And the important part of that code is that `stage.mouseX` and `stage.mouseY` return the current mouse coordinate on the Canvas element. We use those variables to get the mouse position and hold it in a global variable. These values will be used to draw the lines and the reason we save them in a global variable is to provide accessibility in other scopes and functions. As you can see, we used the `Point` class to collect the coordinate of the mouse cursor. The `Point` class represents a two-dimensional coordinate system in EaselJS and we use this class to save the cursor pointer.

Using the handleMouseMove function

This function actually draws the line and is used to handle the `mousemove` event. This is our main function to handle drawing.

The source code of the `mousemove` function is as follows:

```
function handleMouseMove(event) {
  var midPt = new createjs.Point(oldPt.x + stage.mouseX>> 1,
     oldPt.y + stage.mouseY>> 1);

  drawingCanvas.graphics.clear().setStrokeStyle
    (stroke, 'round', 'round').beginStroke(color).moveTo
    (midPt.x, midPt.y).curveTo(oldPt.x, oldPt.y,
     oldMidPt.x, oldMidPt.y);

  oldPt.x = stage.mouseX;
  oldPt.y = stage.mouseY;

  oldMidPt.x = midPt.x;
```

```
    oldMidPt.y = midPt.y;

    stage.update();
  }
```

This function is called continuously while the mouse cursor is dragging over `stage`. In this function, we draw the line using the `beginStroke` function and then save the current mouse position again in order to use it in the following function calls; actually, the following mouse move. By each move of the mouse cursor, this function is called again, so we will have the line.

In the first line you can see the right shift operator (the `>>` operator). We used this function to simplify the `Math.floor(num / 2)` operation. Actually, `num>> 1` and `Math.floor(num / 2)` have the same result.

After that, we update the stage using the `update` function to apply changes to the stage and render everything to the canvas.

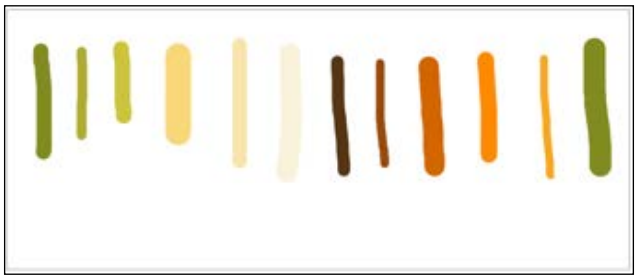
Utilizing the handleMouseUp function

This function is called when a user releases the mouse click and uses it to end the drawing lines and remove the event from `stage`. Its source code is as follows:

```
function handleMouseUp(event) {
  stage.removeEventListener("stagemousemove", handleMouseMove);
}
```

All we do in this function is call `removeEventListener` to remove the `mousemove` event and prevent calling the function anymore. After removing this event from `stage`, the `handleMouseMove` function won't call anymore. So, by moving the mouse cursor, EaselJS won't call our function until the next `mousedown` event. That's exactly what we want to handle in the painting logic.

In the following screenshot, you can see a preview of this application:



Downloading the source code

This painting example is one of the basic EaselJS samples. You can download the complete source code of the project from EaselJS's GitHub:

`https://github.com/CreateJS/EaselJS/blob/master/examples/CurveTo.html`

Summary

In this chapter, we have discussed how to create a simple painting application from scratch using the `mousemove`, `mousedown`, and `mouseup` events, and it's a good exercise to understand the concept of those events. Then, we learned how to manage mouse events inside each other to draw a line. The `addEventListener` and `removeEventListener` functions were used to add and remove an event from an object.

Also, we learned how to use the stroke feature in EaselJS to draw lines with a specific color and size. We used the `beginStorke`, `curveTo`, and `moveTo` functions to draw the lines and handle the painting logic.

In the next chapter, we will talk about *vector mask* and how to create a mask layer in CreateJS.

8
Utilizing Vector Masks

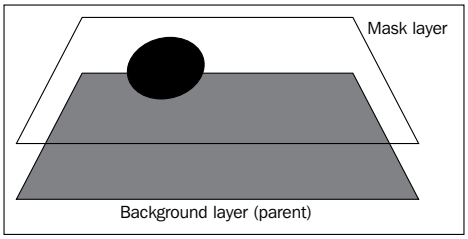
In this chapter, we will talk about utilizing vector masks in CreateJS and how to develop animation or drawings using vector masks. First off, we should know what a vector mask is and what it does. With vector masks, we can control which parts of the parent layer are hidden or revealed. We can even add a special effect to the vector mask to make that part of the parent layer different from the rest.

In this chapter we will cover:

- Learning about vector masks
- Adding a vector mask to an existing `DisplayObject` object
- Applying vector masks to pictures
- Animating the mask layer

Learning about vector masks

Vector masking is a useful feature in EaselJS that enables developers to create awesome animations or drawings easily. There is a `mask` property for every `DisplayObject` object, with which you can apply a mask layer, or in other words; create a layer over an existing shape or picture. After applying the mask layer and updating the `stage` event, you will see that a new layer masks the existing layer. In other words, you can control what part of the parent layer is hidden or revealed with vector masks.



Furthermore, mask layers are also shapes, so you can change the mask layer properties continuously to create animations.

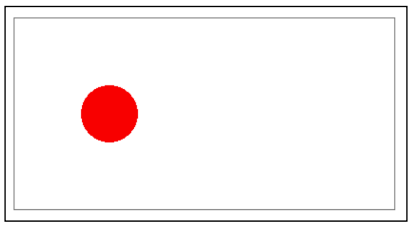
Here is an example of using a vector mask in EaselJS:

```
/* Declare stage in usual way */
var canvas = document.getElementById("canvas");
var stage = new createjs.Stage(canvas);

/* Create the mask */
var mask = new createjs.Shape();
mask.graphics.drawCircle(0, 0, 30);
mask.x = 100;
mask.y = 100;

var bg; /* Create a red background */
var bg = new createjs.Shape();
bg.graphics.clear().beginFill("red").rect(0, 0, 400, 200);

/* Add mask to background */
bg.mask = mask;

/* Add to stage */
stage.addChild(bg);

/* update stage in usual way */
stage.update();
```

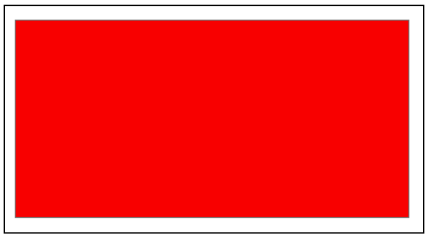
As with other examples in this book, we first get the `canvas` element from the page and then create the `Stage` object. Then, we create a simple circle using the `Shape` class and the `drawCircle` function. For the default position, we set both `x` and `y` to `100`. This circle is our vector mask layer.

Then, we create a `bg` variable that contains a rectangle filled with the color red. After that, we assign the first shape—that is, the circle—to the `mask` property of the `bg` variable. Finally, we add the `bg` variable to stage.

Here is the output of the preceding source code:

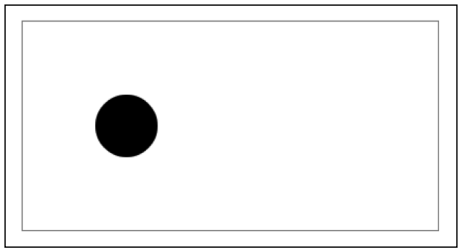


For you to understand the example better, look at the following screenshot. It is what the output will be after removing the mask layer.



As you can see in the first example, our mask layer is seen only in the circular shape, but in the next example, the whole rectangle is seen because there is no mask layer anymore.

The following screenshot shows the mask layer independently:



After assigning the mask layer to the parent layer (the red rectangle), the only visible part of the rectangle will be the visible part of the mask layer.

In the next section, we will look at a drag-and-drop example with mask layers.

Using a vector mask with Bitmap images

In this section, you will learn to use the vector mask, filters, and a `Bitmap` class with an example. The `Bitmap` class is a subset of `DisplayObject`; thus, it has almost all the properties of the `Shape` class, such as `filters`, `mask`, and so on.

Here is an example of using a vector mask with a `Bitmap` class:

```
//query the canvas element
var canvas = document.getElementById("canvas");

//create the Stage class
```

```
var stage = new createjs.Stage(canvas);

//create the mask layer
var mask = new createjs.Shape();
mask.x = img.width / 2;
mask.y = img.height / 2;
mask.graphics.drawCircle(0, 0, 100);

var bg = new createjs.Bitmap(img);
//add blur filter
bg.filters = [new createjs.BlurFilter(50,50,10)];
bg.cache(0,0,img.width,img.height);
bg.mask = mask;
stage.addChild(bg);

stage.update();
```

In the first line, we created the `canvas` variable that refers to our `canvas` element on the page. Then, we initiated the `stage` variable with the `Stage` class.

In the next line, we initiated a `mask` variable using the `Shape` class. This shape is our mask layer, and it's a circle. For the mask layer coordinates, we used `img.width / 2` and `img.height / 2` to place the mask layer in the center of the picture. Then, we created the circle using the `drawCircle` method.

Then we created the `bg` variable, which refers to our image. We initiated this variable using the `Bitmap` class; the first parameter of the `Bitmap` class is the `Image` class. We already loaded the image using the `Image` class.

Here is an example of loading an image and using an `onload` event:

```
var img = new Image();
img.src = "easlejs.png";

img.onload = function(evt) {
    //logic
}
```

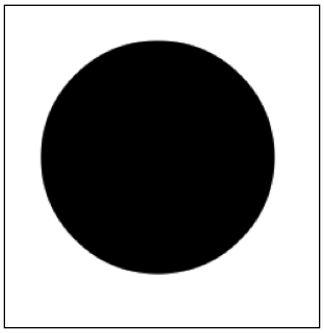
You can use the same approach to load images and pass them to the `Bitmap` class in EaselJS.

Then, we added a blur filter to the picture filters and cached the shape using the `cache` method.We used the original image dimensions for the `cache` method. Then we assigned the mask layer to the `bg` variable using the `mask` property.

Finally, we added the `bg` variable to the `stage` event and updated this event to apply the changes.

In order to understand the differences better, look at the following output screenshot of the `bg` variable without the `mask` property. This is the `Bitmap` class without the mask layer.

The following image shows the vector mask separately:



The following screenshot illustrates the final result of our example with the mask layer:

As you can see, the whole image is visible in the first screenshot. However, the only visible part in the third image is our mask layer, that is the circle. That's the way vector masks work with pictures and shapes. You can create any shape and mask an existing layer, such as a picture with this shape.

In the next example, we will create a drag-and-drop interaction using vector masks.

Playing with vector masks

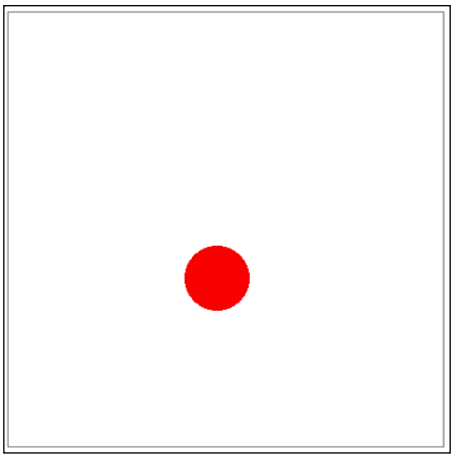
Now, we will complete our previous example to create a simple drag-and-drop example using vector masks. The idea is to change the x and y coordinates of the mask layer on the `mousemove` event of the parent layer so that we can only see the masked layer over the existing shape. It will seem that only a circular shape is being dragged, but what's actually happening is that our mask layer is changing continuously. The source code of our example is as follows:

```
var stage = new createjs.Stage("canvas");
var mask = new createjs.Shape();

mask.graphics.drawCircle(0, 0, 30);
mask.x = 100;
mask.y = 100;

var bg = new createjs.Shape();
bg.graphics.clear().beginFill("red").rect(0, 0, 400, 400);

bg.mask = mask;

function handlePress(event) {
    event.addEventListener("mousemove", handleMove);
}

function handleMove(event) {
    mask.x = stage.mouseX;
    mask.y = stage.mouseY;
    stage.update();
}

bg.addEventListener("mousedown", handlePress);

stage.addChild(bg);
stage.update();
```

As in the previous example, we created a mask layer in the shape of a circle in the first line. We specified the default coordinates for the mask layer with `x=100` and `y=100`. Then, we created a `bg` variable that contains the background or parent layer.

Because we need the coordinates of the mask layer to change continuously as we move the mouse cursor, we bound a callback function to both `mousedown` and `mousemove` events. Then, inside the `mousemove` callback function, we changed the co-ordination of the mask layer and updated the stage.

The result will look like a drag-and-drop ball over the stage, but actually, it's our mask layer that keeps changing with every mouse move.



Summary

The vector mask feature is one of the most useful features for drawing and developing animations, not only in CreateJS but in all other tools. In this chapter, we learned how to create vector mask layers in EaselJS and also how to enhance them to create animations. This feature also comes handy when you need to apply a different filter to a specific part of an existing shape or picture.

In the next chapter, we will wrap up everything to create a complete UI from scratch using all the CreateJS features that we have already talked about.

9
Developing Your First CreateJS Application

In our previous chapters, you learned everything you need to know about building impressive web applications using CreateJS. In this chapter, we will wrap up everything and learn to build an actual application using CreateJS from scratch. We will develop a simple painting application with options such as the ability to change the background color, brush color, brush style, brush size, and so on. Also, you will learn about the tricks and tips that will help you develop better JavaScript libraries and applications.

In this chapter, we will cover the following topics:

- Conceptualizing the application
- Explaining the structure of the application
- Implementing every part of the application
- Getting an image exported from the `Canvas` element

Understanding your application structure

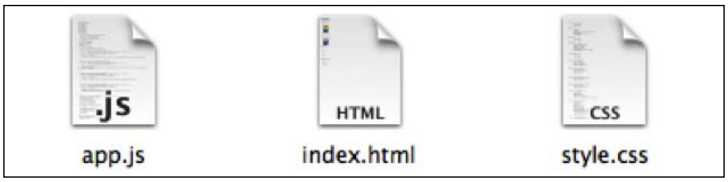
To demonstrate the application development with CreateJS, we will build a painting application. In this application, we will use the EaselJS module and some pure JavaScript snippets to get an image exported from the `Canvas` element. As you know, EaselJS renders all outputs into a `Canvas` element; there are some JavaScript functions to get an image output from the `Canvas` element.

This application contains three files:

- `index.html`

- `app.js`
- `style.css`

In the `index.html` file, we create the HTML elements and link external files to it. `app.js` is the main JavaScript file that contains almost all JavaScript code to run an application, and `style.css` is used to style the page, drop-down menus, and other minor elements.



We split the dependencies into different files to manage them better and provide better performance while loading the application. External static files will be cached in browsers, so users will not require to download them every time the page is refreshed.

Our painting application mainly works with EaselJS features, such as the `curveTo` and `beginStroke` functions. In order to control the application's global settings such as the background color and brush color, we have global variables that hold the settings. We will be using them in other events or functions. These variables are changed when the user clicks on the different options and menus present on the page.

The preview of the application interface is as follows:

The application has a navigation bar that consists of four drop-down menus and a title. With the drop-down menus, users can change the brush style, brush size, background color, and brush color. Also, there is an **Export** link that converts the `Canvas` element to a PNG image, and gives a download link to the user from where the user can download the image.

Our drop-down menus work with pure CSS code so we don't need any JavaScript code for them. In the next section, we will explain each part in detail.

Developing the index.html file

Our main HTML page has a simple structure. Following is a `head` tag in an HTML page:

```
<!DOCTYPE html>
<html>
<head>
   <meta http-equiv="content-type" content="text/html;
     charset=UTF-8">
   <title>Painting</title>
   <link rel="stylesheet" type="text/css"
     href="style.css">
</head>
```

In the `head` tag, we only link the CSS file to the page. Other JavaScript files will be connected to the page at the end of the file and the reason is to provide better performance while loading the page. When we add stylesheets in the `head` tag and JavaScript files at the end of the HTML file (before closing the `body` tag), our page interface will appear to work faster because the browser doesn't wait to load all JavaScript and CSS files. The web browser loads CSS files because they are in the `head` tag, and after rendering all the HTML elements, it loads the JavaScript files. This trick gives a better feeling about the application's performance to users.

In the following lines, we have the `body` and `wrapper` elements:

```
<body>
   <div id="wrapper">
       <div id="header">
           <h1>Painting</h1>
```

The `wrapper` layer is the container for all other elements. Also the `div` header is the container for the black-colored header at the top of the page, as seen on the output screen. This section also contains the drop-down menus and export links.

The source code for one of the drop-down menus to choose the color of the brushes is as follows:

```
<div id="colorPicker" class="pickerDropDown">
   <span class="hex">Brush color</span>
   <span class="fill" style="background-color: #004358;"></span>
   <div class="sub">
       <ul>
           <li>
```

```
                <a href="javascript:void(0);"
                   class="sphexbrushColor" style="background:
                   #FD7400;">
                   #FD7400
                </a>
            </li>
            <li>
                <a href="javascript:void(0);"
                    class="sphexbrushColor" style="background:
                    #FFE11A;">
                    #FFE11A
                </a>
            </li>
            <li>
                <a href="javascript:void(0);"
                    class="sphexbrushColor" style="background:
                    #BEDB39;">
                    #BEDB39
                </a>
            </li>
            <li>
                <a href="javascript:void(0);"
                    class="sphexbrushColor" style="background:
                    #1F8A70;">
                    #1F8A70
                </a>
            </li>
            <li>
                <a href="javascript:void(0);"
                    class="sphexbrushColor" style="background:
                    #004358;">
                    #004358
                </a>
            </li>
        </ul>
    </div>
</div>
```

Each drop-down menu has a `div` element with a subclass. Inside the `div` element, we have some `ul` and `li` elements that define the drop-down menu. For color pickers, we have a circle that shows the current color. Other drop-down menus have the same structure. After drop-down menus, we have a link to export the image.

The source code for the same is as follows:

```
<div id="exportToImage" class="pickerDropDown">
   <span class="hex">
       <a href="javascript:void(0)"
         onclick="exportToImage(this);">Export</a>
   </span>
</div>
```

As you can see, we have a function call when a user clicks on the `Export` link. We call the `exportToImage` function, which converts the `Canvas` element to a PNG image.

We will explain this function better in the next sections.

Finally, we have the definition of the `Canvas` element:

```
<div id="main">
   <canvas id="pStage"></canvas>
</div>
```

The `canvas id` object is assigned with the `pStage` value and is placed inside a `div` element. After that, we link our two JavaScript files. The first file is the CreateJS library with all subsets in a combined file and the second is the `app.js` file as shown:

```
<script type='text/javascript'
   src="http://code.createjs.com/createjs-
   2013.12.12.min.js"></script>
<script type='text/javascript' src="app.js"></script>
```

We use CreateJS CDN servers to load the main library file. This file is already minified and we can use it in the production environment.

Implementing the app.js file

The `app.js` file is the main JavaScript file that contains all functionality and logic for the painting application. This file consists of five functions; one of them is the main function that performs the set up of other events and configures and creates the stage. The next three functions are the callback functions for different mouse events, and the last function is used to create a PNG image from the `Canvas` element. But before everything else, we have the global variable's declaration shown as follows:

```
var canvas, stage, drawingCanvas, oldPt, oldMidPt, bgLayer,
   brushColor, brushSize, bgColor, brushStyle, mouseMoveFn;
```

We will explain more about the usage of each variable in the next sections.

After that, we have the `init` function, which is the main function for the application.

```
canvas = document.getElementById("pStage");

canvas.width = window.innerWidth;
canvas.height = window.innerHeight - 73;

//set default colors
brushColor = "#004358";
bgColor = "#FCFFF5";
brushSize = 12;
brushStyle = "round";
```

In the first line, we get the `Canvas` element using the `getElementById` function. Then we set the width and height of the `Canvas` element to the window's width and height to fit the canvas to the page. The reason we use `-73` for the `innerHeight` value is to prevent vertical scrolling of the page as our header height is about 73 pixels. After that, default options are set. You can change them with to do your preferred options.

In order to bind the `onclick` events to the drop-down menus, we have a simple `for` loop that iterates over the `ul` items and binds the `onclick` event to the links:

```
//bind onclick event to the brush color picker
for (var i = document.getElementsByClassName("brushColor").length
  - 1; i>= 0; i--) {
   var item = document.getElementsByClassName("brushColor")[i];

   item.onclick = function () {
       brushColor = document.querySelector("#colorPicker
         .fill").style.backgroundColor =
         this.style.backgroundColor;
   }
};
```

In the first line, we have a `for` loop that iterates over the drop-down items, and then binds an `onclick` event to each item. The same code is also used for other drop-down menus. Finally, we end the file with the following code:

```
stage = new createjs.Stage(canvas);
stage.autoClear = false;

createjs.Touch.enable(stage);

stage.on("stagemousedown", mouseDownCallback);
stage.on("stagemouseup", mouseUpCallback);

bgLayer = new createjs.Shape();
bgLayer.graphics.beginFill(bgColor).drawRect(0, 0, canvas.width,
  canvas.height);
stage.addChild(bgLayer);

drawingCanvas = new createjs.Shape();
stage.addChild(drawingCanvas);
stage.update();
```

In the first line, like our previous examples, `Stage` is the object that is created. After that, we set the `autoClear` property to `false` in order to manage the `stage` object getting cleared manually. Then, we set the `touch` feature to `enable`.

We are developing a painting application so we need to bind callback functions to the `mousedown`, `mouseup`, and `mousemove` events in order to manage and control mouse events. In the next lines, we bind callback functions to the `stagemousedown` and `stagemouseup` events, which are used to handle the mouse-click events.

In the painting application, we have a background layer where the user can change the color using the drop-down menu.

In the next lines, we create a `Shape` object that is used for the background layer and then we create the next shape to draw the painting lines. Both of these shapes are added to the stage using the `addChild` function.

The source code for the `mouseDownCallback` event is as follows:

```
oldMidPt = oldPt = new createjs.Point(stage.mouseX, stage.mouseY);
mouseMoveFn = stage.on("stagemousemove", mouseMoveCallback);
```

Inside this function, we collect the current mouse cursor's coordinates and also add a callback function to the `stagemousemove` event.

The `mouseMoveCallback` function source code is shown as follows:

```
Var midPt = new createjs.Point(Math.floor((oldPt.x + stage.mouseX)
   / 2), Math.floor((oldPt.y + stage.mouseY) / 2));

drawingCanvas.graphics.setStrokeStyle(brushSize,
   brushStyle).beginStroke(brushColor).moveTo(midPt.x,
   midPt.y).curveTo(oldPt.x, oldPt.y, oldMidPt.x, oldMidPt.y);

oldPt.x = stage.mouseX;
oldPt.y = stage.mouseY;

oldMidPt.x = midPt.x;
oldMidPt.y = midPt.y;

stage.update();
```

In the first line, we calculate the next point that we need for the `moveTo` function using the current mouse position and the old mouse position. In the next line, we create a stroke with the current options and move the point to the new coordinates that we have calculated in the first line. After that, old positions are updated and finally the `update` function is called from the `stage` object.

Our last callback function for events is the `mouseUpCallback` function. Inside this function, we unbind the callback function from `stagemousemove` to stop drawing, which is shown as follows:

```
stage.off("stagemousemove", mouseMoveFn);
```

The last function is the `exportToImage` function, which is used to get a PNG image exported from the `Canvas` element. In this function, we convert the `Canvas` element to a PNG image format with base 64 and set the output to the link's `href` object. There is a function called `toDataUrl` that converts the contents from the `Canvas` element to an image. The `exportToImage` function is called when the **Export** link is clicked by a user. The following code explains the same:

```
var dateTime = new Date();

obj.href = canvas.toDataURL();
obj.download = "paint_" + dateTime.getHours() + "-" +
   dateTime.getMinutes() + "-" + dateTime.getSeconds();
```

At the end of the file, we call the `init` function to start the application:

```
init();
```

Preview of the final application

Once the code is run, our painting application will be ready for use. The preview of our final application is as shown in the following screenshot:

Summary

In this chapter, we learned to create an actual web application using CreateJS from scratch and use the different features offered by this library. We gained experience on how to declare global options and change them using user inputs, and apply the changes in the application. Also, we learned how to export images from the `Canvas` element using the powerful JavaScript API.

Furthermore, we discussed how to include static files such as JavaScript and CSS to provide better performance while loading the application.

Every new beginning is some beginning's end and as you turn the last pages of this intuitive guide, you are empowered to explore, discover, develop, and build astounding web applications using CreateJS. In this age of the Internet citizenship, HTML5 has emerged as a powerful platform, where you can make your mark with compelling web applications. So go ahead, create the next amazing web app, and thrill the world!

Index

E

F

G

H

I

L

M

N

P

S

T

V

W

Thank you for buying

Getting Started with CreateJS

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

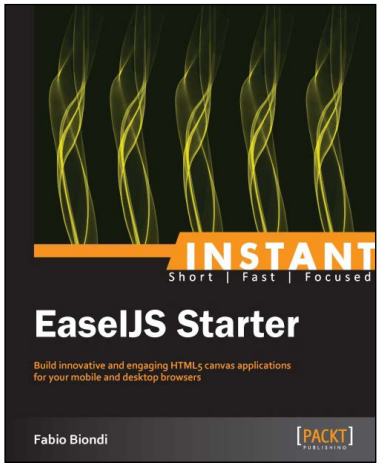
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



Instant EaselJS Starter

ISBN: 978-1-78216-518-7 Paperback: 54 pages

Build innovative and engaging HTML5 canvas applications for your mobile and desktop browsers

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Learn to create interactive web content with the latest version of EaselJS framework and the HTML5 Canvas element
3. Starts with the basics and you will soon find yourself creating responsive and customized applications
4. Learn how to use EaselJS, TweenJS, and PreloadJS to create user interfaces and interactive animations

---

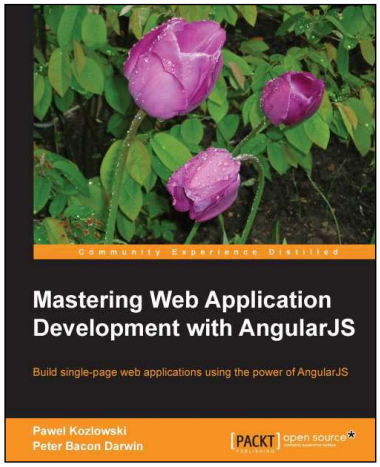
HTML5 for Flash Developers

ISBN: 978-1-84969-332-5 Paperback: 322 pages

Leverage your Flash skill set and learn to create content using a wide range of HTML5 web development features

1. Discover and utilize the wide range of technologies available in the HTML5 stack
2. Develop HTML5 applications with external libraries and frameworks
3. Prepare and integrate external HTML5 compliant media assets into your projects

[PACKT] open source
community experience distilled
PUBLISHING



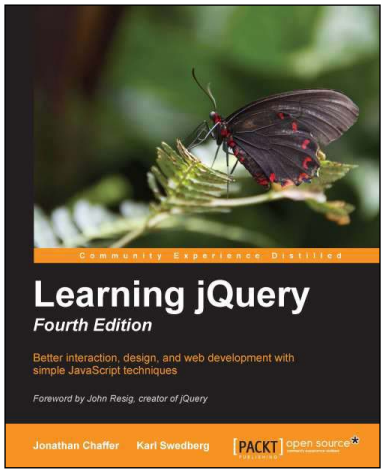
Mastering Web Application Development with AngularJS

ISBN: 978-1-78216-182-0 Paperback: 372 pages

Build signal-page web applications using the power of Angular JS

1. Make the most out of AngularJS by understanding the AngularJS philosophy and applying it to real life development tasks
2. Effectively structure, write, test, and finally deploy your application
3. Add security and optimization features to your AngularJS applications
4. Harness the full power of AngularJS by creating your own directives

---



Learning jQuery

Fourth Edition

ISBN: 978-1-78216-314-5 Paperback: 444 pages

Better interaction, design, and web development with simple JavaScript techniques

1. An introduction to jQuery that requires minimal programming experience
2. Detailed solutions to specific client-side problems
3. Revised and updated version of this popular jQuery book