Learn Pixi.js

Create Great Interactive Graphics for Games and the Web

Rex van der Spuy

Apress®

Learn Pixi.js

ISBN-13 (pbk): 978-1-4842-1095-6

ISBN-13 (electronic): 978-1-4842-1094-9

Managing Director: Welmoed Spahr
Lead Editor: Ben Renow-Clarke
Development Editor: Matthew Moodie
Technical Reviewer: Jason Sturges
Editorial Board: Steve Anglin, Mark Beckner, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Steve Weiss
Coordinating Editor: Jill Balzano
Copy Editor: Michael G. Laraque
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science+Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

For Freya, Queen of the Pixies!

Contents at a Glance

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

■ Chapter 1: Making Sprites 1

■ Chapter 2: Moving Sprites 43

■ Chapter 3: Shapes, Text, and Groups 69

■ Chapter 4: Making Games 105

■ Chapter 5: Animating Sprites 121

■ Chapter 6: Visual Effects and Transitions 145

■ Chapter 7: Mouse and Touch Events 175

■ Appendix: Pixie Perilousness!—Complete Code 199

Index 207

Contents

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

Chapter 1: Making Sprites 1

Creating the Renderer and Stage 1

Render Options 2

Customizing the Canvas 3

Scaling the Canvas to the Browser Window 4

Pixi Sprites 6

Understanding Textures and the Texture Cache 7

Loading Images 8

Displaying Sprites 9

Removing Sprites 11

Using Aliases 12

A Little More About Loading Things 13

Making a Sprite from an Ordinary HTML Image Object or Canvas 13

Assigning a Name to a Loading File 14

Monitoring Load Progress 14

More About Pixi's Loader 16

Positioning Sprites 18
X and Y Properties 18
Size and Scale 20
Rotation 22
Making a Sprite from a Tileset Sub-image 24
Preventing Texture Bleed 28
Using a Texture Atlas 30
Creating the Texture Atlas 30
Loading the Texture Atlas 33
Creating Sprites from a Loaded Texture Atlas 34
Making the Blob Monsters 37
The Complete Code 39
Summary 42
Chapter 2: Moving Sprites 43
Create a Game Loop 43
Using Velocity Properties 46
Game States 48
Keyboard Movement 51
Adding Acceleration and Friction 57
Adding Gravity 60
Containing Movement Inside a Fixed Area 61
Using ES6 Sets 64
The contain Function 66
Summary 67
Chapter 3: Shapes, Text, and Groups 69
Making Shapes 70
Rectangles 72
Circles 74

Ellipses 75
Straight Lines 76
Polygons 76
Curved Lines 77
Drawing Arcs 80
Improving Graphics Rendering 81
Antialiasing for WebGL Graphics 81
Drawing on a Single Graphics Context 83
Redrawing Animated Graphics Each Frame 84
Displaying Text 87
The Text Object 87
Loading Font Files 90
Using Bitmap Fonts 92
Grouping Sprites 95
Using a Container 95
Using a ParticleContainer 101
Summary 103
Chapter 4: Making Games 105
Collision Detection 105
Installing and Setting Up Bump 106
Using the hitTestRectangle Method 106
Collision Detection in Action 106
Treasure Hunter 108
The Code Structure 109
Initialize the Game in the Setup Function 110
Playing the Game 115
Summary 120

Chapter 5: Animating Sprites 121

Using SpriteUtilities 121

MovieClip Sprites 122

The Even Easier Way 123

Using MovieClip Sprites 124

MovieClip Properties and Methods 125

Make MovieClip Sprites Using a Texture Atlas 126

Using the frameSeries Utility Function 128

Animation States 128

Making a Sprite with a State Player 128

Defining Sprite States 129

Making a Walking Sprite 134

Creating Frames for Animations 137

Particle Effects 138

Add a Little Pixi Dust 138

Making Particles 139

Using ParticleContainer 140

Customizing the Particle Options 141

Using a Particle Emitter 142

Summary 144

Chapter 6: Visual Effects and Transitions 145

Tiling Sprites 146

Tools for Working with Textures 149

Using generateTexture 149

Using cacheAsBitmap 150

Using RenderTexture 150

Tinting 151
Masking 152
Blend Modes 153
Filters 154
Video Textures 159
Working with Multiple Resolutions 160
Rope Mesh 161
Tweening and Transitions 163
Setting Up and Running Charm 163
Sliding Tweens 164
Setting the Easing Types 165
Using slide for Scene Transitions 166
Following Curves 169
Following Paths 170
A Few More Tween Effects 173
Summary 174
Chapter 7: Mouse and Touch Events 175
Setting Up Tink 175
Setting the Optional Scale 176
A Universal Pointer 176
Pointer Interaction with Sprites 177
Drag-and-Drop Sprites 179
Buttons 181
What Are Buttons? 181
Making Buttons 182
Making an Interactive Sprite 185

Case Study: Pixie Perilousness! 185
Creating the Scrolling Background 186
Creating the Pillars 188
Making Pixie Fly 190
Emitting Pixie Dust 192
Fine-Tuning the Pixi's Animation 193
Collisions with the Blocks 194
Resetting the Game 195
Taking It Further 195
Your Next Steps 196
Appendix: Pixie Perilousness!—Complete Code 199
The HTML Code 199
The JavaScript Code 200
Index 207

About the Author

Rex van der Spuy is a leading expert on video game design and interactive graphics and the author of the popular Foundation and Advanced series of books about how to make video games. Rex has designed games and performed interactive interface programming with Agency Interactive (Dallas), Scottish Power (Edinburgh), DC Interact (London), Draught Associates (London), and the Bank of Montreal (Canada). He's also built game engines and interactive interfaces for museum installations for PixelProject (Cape Town, South Africa), as well as "Ga," the world's smallest full-featured 2D game engine. He created and taught advanced courses in game design for many years at the Canadian School of India (Bangalore). The highlight of his career was programming video games on the Annapurna glacier at 4,500 meters (which, to his delight, was 1,000 meters higher than the maximum permissible operating altitude of his laptop).

About the Technical Reviewer

Jason Sturges is a cutting-edge technologist focused on ubiquitous delivery of immersive user experiences. Coming from a visualization background, he's always pushing the boundaries of computer graphics to the widest reach cross-platform, while maintaining natural and intuitive usability per device. From interactivity, motion, animations, and creative design, he has worked with numerous creative agencies on projects from kiosks to video walls to Microsoft Kinect games. Most recently, the core of his work has been mobile apps.

Committed to the open source community, he is also a frequent contributor to GitHub and Stack Overflow as a community resource, leveraging modern standards, solid design patterns, and best practices in multiple developer tool chains for Web, mobile, desktop, and other platforms.

Acknowledgments

Most illustrations and game characters for this book were created by the extraordinarily talented Kipp Lightburn (www.waymakercreative.com, waymakercreative@gmail.com). Thanks so much, Kipp!

The game graphics for Treasure Hunter were designed by Lanea Zimmerman and are from her brilliant Tiny 16 tileset (opengameart.org/content/tiny-16-basic).

For the game Pixie Perilousness! the green block graphic was designed by the author GameArtForge (opengameart.org/content/blocks-set-01).

Thanks to Chad Engler (github.com/englercj), one of Pixi's lead developers, for patiently and generously answering all of my technical questions in the Pixi discussion forum on html5gamedevs.com, and for single-handedly acting as Pixi's tireless one-man tech support team.

Thanks to Mat Groves (www.goodboydigital.com), the creator of Pixi..., for creating Pixi (!) and also for his enthusiastic support of this project and permission to use images and code samples from Pixi's web site. In particular, the sample code and image for the Rope object in Chapter 6 are based on Mat's original work.

The photograph of me was by taken by Sivan Ritter, in Arambol, Goa.

Introduction

If you want to start making games or applications for the Web, desktop, or mobile devices, Pixi is the best place to start. Pixi is an extremely fast 2D sprite rendering engine that helps you to display, animate, and manage interactive graphics, so that it's easy for you to make any visually rich interactive software you can imagine, using JavaScript and other HTML5 technologies. Pixi has a sensible, uncluttered API and includes many useful features, such as supporting texture atlases, and provides a streamlined system for animating sprites (interactive images).

■ **Note** What is an API? The acronym stands for "Application Programming Interface." The API refers to all of Pixi's objects and methods that let you make a bunch of cool stuff, without having to worry about how the underlying code actually works.

Pixi also gives you a complete **scene graph**, so that you can create hierarchies of nested sprites (sprites inside sprites), as well as letting you attach mouse and touch events directly to sprites. And, most important, Pixi gets out of your way, so that you can use as much or as little of it as you want, adapt it to your personal coding style, and integrate it seamlessly with other useful frameworks.

Pixi's API is actually a refinement of a well-worn and battle-tested API pioneered by Macromedia/Adobe Flash. Old-skool Flash developers will feel right at home. Other current sprite rendering frameworks use a similar API: CreateJS, Starling, Sparrow, and Apple's SpriteKit. The strength of Pixi's API is that it's general-purpose: it gives you total expressive freedom to make anything you like without dictating a specific workflow or architecture. That's good, because this means you can build your own architecture around it.

In this book, you'll learn everything you need to know to start using Pixi quickly. Although you can use Pixi to make any kind of interactive media, such as apps or web sites, I'm going to show you how to use it to make games. Why games? Because if you can make a game with Pixi, you can make anything else with Pixi. Besides, I like making games—and you will too!

But Pixi doesn't do everything! In this book, you're also going to learn to use a suite of easy-to-use helper libraries that extend Pixi's functionality in all kinds of exciting ways.

Bump: 2D collision functions

`github.com/kittykatattack/bump`

Charm: Tweening animation effects

`github.com/kittykatattack/charm`

Tink: Mouse and touch interactivity

`github.com/kittykatattack/tink`

Dust: Particle effects

`github.com/kittykatattack/dust`

SpriteUtilities: Advanced sprite creation utilities

`github.com/kittykatattack/spriteUtilities`

Sound.js: Music and sounds effects

`github.com/kittykatattack/sound.js`

It's everything you need to create anything you can imagine!

What Do You Have to Know?

To make use of this book, you should have a reasonable understanding of HTML and JavaScript. You don't have to be an expert or think of yourself as a "programmer." You're just an ambitious beginner with an eagerness to learn—just like I am!

If you don't know HTML and JavaScript, the best place to start learning is the book *Foundation Game Design with HTML5 and JavaScript*. I know for a fact that it's the best book, because I wrote it! It will give you all the coding and conceptual background you need to begin using this book. In fact, if you've read *Foundation Game Design*, then *Learn Pixi.js* is the perfect sequel.

This is not a tech-heavy book! It's a fun, practical book that gets to the point—fast. I'm not going to bog you down with mountains of heavy code or theory. I've kept the code simple and architecturally flat and expect that you're smart enough to be able to look at the examples and apply them to your own projects, in your own way. This book is just the map; it's up to you to take the journey.

Note If you're curious and want to take a deeper technical dive into video game programming, make sure to check out this book's hip and sophisticated older sister: *Advanced Game Design with HTML5 and JavaScript*. It shows how you can code a display engine similar to Pixi from scratch, as well as all the essential code you must know to make all kinds of 2D action games. It's a great complement to *Learn Pixi.js*, and you can share concepts and code between both books.

JavaScript ES6

This book is written in the latest version of JavaScript: ECMAScript 6, or ES6 for short. I'll introduce any new ES6 features in the code as you stumble upon them. You'll find them easy to learn. ES6 is really just a better, friendlier version of JavaScript that gets more done with less code. You won't look back once you start using it.

But, just to get you started, following are the two most important things you need to know about ES6.

1. Use let Instead of var

In most cases, you can declare a variable using the new keyword `let`.

```
let anyValue = 4;
```

In older versions of JavaScript (ES3 and ES5), you have to use `var`.

The `let` keyword gives the variable **block scope**. That means the variable can't be seen outside the pair of curly braces that it's defined in. Here's an example:

```
let say = "hello";
let outer = "An outer variable";

if (say === "hello") {
  let say = "goodbye";

  console.log(say);
  //Displays: "goodbye"

  console.log(outer);
  //Displays: "An outer variable"

  let inner = "An inner variable";
  console.log(inner);
  //Displays: "An inner variable"
}

console.log(say);
//Displays: "hello"

console.log(inner);
//Displays: ReferenceError: inner is not defined
```

A variable defined outside the `if` statement can be seen inside the `if` statement. But any variables defined inside the `if` statement can't be seen outside it. That's what block scope is. The `if` statement's curly braces define the block in which the variable is visible.

In this example, you can see that there are two variables called `say`. Because they were defined in different blocks, they're different variables. Changing the one inside the `if` statement doesn't change the one outside the `if` statement.

2. "Fat arrow" Function Expressions

ES6 has a new syntax for writing function expressions, as follows:

```
let saySomething = (value) => {
  console.log(value)
};
```

The => symbol represents an arrow pointing to the right, like this: ➔. It's visually saying "use the value in the parentheses to do some work in the next block of code that I'm pointing to."

You define function expressions in the same way you define a variable, by using `let` (or `var`). Each one also requires a semicolon after its closing brace. A function expression must be defined before you use it, like this:

```
let saySomething = (value) => {
  console.log(value)
};

saySomething("Hello from a function statement");
```

That's because function expressions are read at **runtime**. The code reads them in the same order, from top to bottom, that it reads the rest of the code. If you try to call a function expression before it has been defined, you'll get an error.

If you want to write a function that returns a value, use a `return` statement, such as the following:

```
let square = (x) => {
  return x * x;
};

console.log(square(4));
//Displays: 16
```

As a convenience, you can leave out the curly braces, the parentheses around parameters, and the `return` keyword, if your function is just one line of code with one parameter, as in the following:

```
let square = x => x * x;

console.log(square(4));
//Displays: 16
```

This is a neat, compact, and readable way to write functions.

■ **Note** A nice feature of arrow functions is that they make the scope inside a function the same as the scope outside it. This solves a big problem called **binding** that plagued earlier versions of JavaScript. Briefly, a whole class of quirks you had to work around are no longer issues. (For example, you no longer have to use the old `var self = this;` trick.)

You can write a function expression without using a fat arrow, as follows:

```
let saySomething = function(value) {
  console.log(value)
};
```

This will work the same way as the previous examples, with one important difference: the function's scope is local to that function, not the surrounding code. That means the value of "this" will be undefined. Here's an example that illustrates this difference:

```
let globalScope = () => console.log(this);

globalScope();
//Displays: Window...

let localScope = function(){
  console.log(this);
};

localScope();
//Displays: undefined
```

This difference is subtle but important. In most cases, I recommend that you use a fat arrow to create a function expression, because it's usually more convenient for code inside a function to share the same scope as the code outside the function. But in some rare situations, it's important to isolate the function's scope from the surrounding scope, and I'll introduce those situations when we encounter them.

■ **Note** This book was written when ES6 was brand new. So new, in fact, that no web browsers had yet fully implemented it. If you're in that same position, use an ES6 to ES5 transpiler such as Traceur or Babel to run the book's source code. In the book's source files, you'll find folders called ES5 that contain ES5 versions of all the source code and examples in this book.

Running a Web Server

To use Pixi, you'll also have to run a web server in your root project directory. The best way is to use node.js (`nodejs.org`) and then install the extremely easy-to-use http-server (`github.com/nodeapps/http-server`). However, you have to be comfortable working with the Unix command line, if you want to do that.

Note Are you afraid of the Unix command line? Don't be! Unix is a wonderfully retro-future way to scare your parents, and you can learn it in a few hours. Maybe start with the classic tutorial "Learn Unix in 10 minutes" (`freeengineer.org/learnUNIXin10minutes.html`) and follow it up with the "Unix Cheat Sheet" (`www.rain.org/~mkummel/unix.html`). Install a great little script called "Z" (`github.com/rupa/z`) to help you navigate the file system, and then start playing around. You'll also find dozens of videos on the Web for Unix for beginners, including Michael Johnston's excellent two-part series.

But if you don't want to mess around with the command line just yet, try the Mongoose web server (`cesanta.com/mongoose.shtml`) or just write all your code using one of the many HTML5-based text editors: Brackets, Light Table, or Atom. Any of these will launch a built-in web server for you automatically, when you run your code in a browser.

A Survival Guide for Future Pixi API Versions

This book was written when Pixi was in version 3, but Pixi is a fast-changing, living code library. What that means is that if you're using a future version of Pixi and run some code that isn't completely compatible with the code in this book, you'll have to use your judgment about how to adapt it to the new version. The good news is that Pixi's core user-facing API has been stable since version 1, so most of the code and techniques in this book should be relevant to future versions. But you'll have to stay on your toes! Following are some Future Pixi survival tips.

- **The object or method you're trying to use might have been renamed.** Pixi's development team sometimes likes to rearrange the furniture a bit, and they'll do so on a whim. That means they might change the names of some object or method names or give them new locations. For example, in version 2.0, the `TextureCache` object was located in `PIXI.TextureCache`, while in version 3.0, it was moved to `PIXI.utils.TextureCache`. Likewise, the `setFrame` method became the `frame` property. These aren't deal breakers. They're just cosmetic differences, and the code still works in the same way. But you'll have to be prepared to research possible future changes such as these and update your code if the JavaScript console gives you any errors or warnings.

- **Look for a deprecation document or script.** Pixi v3.0 has a file called `deprecation.js` that logs a message to the console if the code you're using has been changed from earlier versions. That's helpful, but you can't count on it being in future versions. If there's no `deprecation.js` file in the version of Pixi you're using, look for any document in Pixi's code repository that might list differences between versions. If you can't find one, post an issue in the code repository asking for help.
- **Use aliases for Pixi's objects and methods.** A way to slightly buffer yourself against a changing API is to create your own custom set of object and method names that just reference Pixi's. These are called **aliases**. For example, here's how you might create aliases for Pixi's `Sprite` class and `TextureCache` object:

```
let Sprite = PIXI.Sprite,
    TextureCache = PIXI.utils.TextureCache;
```

Do this right at the beginning of your program and then write the rest of your code using those aliases (`Sprite` and `TextureCache`) instead of Pixi's originals. This is helpful, because if Pixi's API changes, you only have to change what the alias is pointing to in one location, instead of every instance where you've used it throughout your entire program. Your own code base will be stable, even if Pixi's API fluctuates.

■ **Note** Another advantage to using aliases is that your code becomes more succinct: you don't have to prefix `PIXI` or `PIXI.utils` to everything. That can considerably shorten some complex lines of code and make your whole program more readable. For all these reasons, the sample code in this book uses aliases that follow this same format. You'll learn more about how to create and use aliases in Chapter 1.

Setting Up a Pixi Coding Environment

Let's find out how to set up a basic coding environment that you can use to run all the sample code in this book, as well as write your own original Pixi code. Do you have your web server running in your project's root directory, a text editor that you like to code in, and a web browser to run your code? Great, now you're ready to start working with Pixi!

Installing Pixi

Grab the latest version of the `pixi.min.js` file from Pixi's code repository (Pixi v3.0 was hosted at `github.com/pixijs/pixi.js`). You'll find the `pixi.min.js` file in the "bin" folder: `github.com/pixijs/pixi.js/tree/master/bin`. This one file is all you need to use Pixi. You can ignore all the other files in the repository; *you don't need them*.

Note If you prefer, you could alternatively use the un-minified `pixi.js` file. The minified file (`.min.js`) might run slightly faster, and it will certainly load faster. But the advantage to using the un-minified plain JS file is that if the compiler thinks there's a bug in Pixi's source code, it will give you an error message that displays the questionable code in a readable format. This is useful while you're working on a project, because even if the bug isn't actually in Pixi, the error might give you a hint as to what's wrong with your own code.

You can also use Git to install and use Pixi. (What is Git? If you don't know you can find out here: `github.com/kittykatattack/learningGit`.) Using Git has some advantages: you can just run `` `git pull` `` from the command line to update Pixi to the latest version. And, if you think you've found a bug in Pixi, you can fix it and submit a pull request to have the bug fix added to the main repository.

To clone the Pixi repository with Git, install `` `cd` `` into your project's root directory and type the following:

```
git clone git@github.com:pixijs/pixi.js.git
```

This automatically creates a folder called `pixi.js` and loads the latest version of Pixi into it.

Note You can also install Pixi using Node (`nodejs.org`) and Gulp (`gulpjs.com`), if you have to do a custom build of Pixi to include or exclude certain features. See Pixi's code repository for details on how to do this.

Create a Basic HTML Container Page

Next, create a basic HTML page and use a `<script>` tag to link the `pixi.min.js` file that you've just downloaded. The `<script>` tag's `src` property should be relative to your root directory on which your web server is running. Your `<script>` tag might look something like this:

```
<script src="pixi.min.js"></script>
```

Here's a basic HTML page that you could use to link Pixi and test that it's working:

```
<!doctype html>
<meta charset="utf-8">
<title>Hello World</title>
```

```
<body>
<script src="pixi.min.js"></script>
<script>

//Test that Pixi is working
console.log(PIXI);

</script>
</body>
```

This is the minimal amount of valid HTML you need to start creating projects with Pixi. If Pixi is linking correctly, `console.log(PIXI)` will display something such as this in your web browser's JavaScript console:

```
Object { VERSION: "3...
```

If you see that (or something similar) you know everything is working properly. Now you can start working with Pixi!