This book will teach you how to create awesome video games. Games from scratch. Games that run cross-platform, in web browsers, and on phones. Games filled with dynamic sound and music. Games overflowing with impressive visual effects. Fun games.

More importantly, this book will teach you how to think about making games. You’ll learn to analyze and dissect games—to understand what it is that makes great games great. By the end of the journey you’ll have all the knowledge and tools needed to produce engaging, polished products that people will love to play.

• Learn the basics: game loops and input
• Draw graphics on the screen using Canvas
• Add amazing sound effects and music using the Web Audio API
• Develop several fun games: a platformer, a shoot’em up, a dungeon crawler, and a physics-based game
• Create your own JavaScript game library
• Jazz up your game with “juice”: screen shakes, particle effects, and more

And much more…

DESIGN AND BUILD THE NEXT INDIE GAME HIT!

Your Author

Earle Castledine—JavaScript flâneur and internet flibbertigibbet—is back! By day it’s serious business while Earle steers the ship as lead engineer at hot startups. By night his attention turns to the world of weird art and wonderful game design.

Author of Jump Start CoffeeScript, and the cult classic jQuery: Novice to Ninja—as well as speaker at countless JavaScript conferences and workshops—Earle is passionate about building engaging, fast, and interactive creator tools.

Now, having been making games with JavaScript for more than a decade, he’s here to guide you on your journey to gamedev superstardom.

SITEPOINT BOOKS
✓ Advocate best practice techniques
✓ Lead you through practical examples
✓ Provide working code for your website
✓ Make learning easy and fun

Visit us on the Web at sitepoint.com or for sales and support email books@sitepoint.com

WEB DEVELOPMENT
Print ISBN: 978-0-9941826-1-4
USD $39.95  CAD $52.95

ALL SOURCE CODE AVAILABLE FOR DOWNLOAD

CREATE SMASH HIT GAMES
About Earle Castledine

Earle Castledine—JavaScript flâneur and internet flibbertigibbet—is back! By day it's serious business while Earle steers the ship as lead engineer at hot startups. By night his attention turns to the world of weird art and wonderful game design.

Author of *Jump Start CoffeeScript*, and the cult classic *jQuery: Novice to Ninja*—as well as speaker at countless JavaScript conferences and meetups all around the globe—he's no novice to brewing up engaging, interesting, and informative creative works.

Now, having been making games with JavaScript long before it was cool, he's here to guide you on your journey to gamedev superstardom.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit [http://www.sitepoint.com/](http://www.sitepoint.com/) to access our blogs, books, newsletters, articles, and community forums. You’ll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.
For Kim Castledine (1948 - 2016)
Thanks for everything, Dad.
# Table of Contents

**Preface** .......................................................................................................................... viii

- Who Should Read This Book? ....................................................................................... viii
- Conventions Used ........................................................................................................... ix
- Supplementary Materials .............................................................................................. x

**Chapter 1: Press Space to Start** ................................................................................. 1

- Game Design .................................................................................................................. 4
- A Brief History .................................................................................................................. 6
- Games and JavaScript ...................................................................................................... 7
- Enough Talk; Let's Make Games ..................................................................................... 11
- Get Ready! ......................................................................................................................... 24

**Chapter 2: Drawing Things** ....................................................................................... 25

- DOM vs Canvas vs WebGL .............................................................................................. 26
- Plain Ol' DOM ................................................................................................................... 27
- Canvas API ....................................................................................................................... 33
- Game Engines vs Reinventing the Wheel ....................................................................... 49
- Quick, Draw ...................................................................................................................... 52

**Chapter 3: Game Loops & User Input** ....................................................................... 53
Chapter 4: Animation, Levels, Maps, Cameras... 104

- An Unexpected Proposition 105
- De-boilerplating Our Prototypes 106
- Sprites with Zing! 109
- An Unexpected Assignment 119
- Sprite Sheets & Animation 120
- The Animation Manager 131
- Tile Maps 135
- Scrolling Maps with a Camera 147
- Screens & Game Life Cycle 153
- Leveled Up 161

Chapter 5: Collision Detection & AI 162

- Colliding with Things 163
- Procedural Level Generation 175
- Colliding with Maps 181
- AI: The Bots Strike Back 192
Pathfinding .......................................................................................................................... 207
Enemies alive .................................................................................................................... 211

Chapter 6:  **Mathematics & Physics** ................................................................. 212

Jump Everybody, Jump ............................................................................................... 213
Fixing Our Time Step .................................................................................................. 221
Triangles and Vectors ................................................................................................. 223
Building a Vector Library ............................................................................................ 224
The Need for Speed (and Direction) ........................................................................... 230
Billiard Ball Physics ...................................................................................................... 242
Polar Coordinates ........................................................................................................ 245
Physics Libraries .......................................................................................................... 248

Chapter 7:  **Audio** ............................................................................................ 269

Old-school Techniques ................................................................................................. 270
Audio on the Web .......................................................................................................... 271
An Asset Manager ......................................................................................................... 281
Sound Production .......................................................................................................... 288
The Web Audio API ...................................................................................................... 303
Post Production: Mixing and Mastering ...................................................................... 317
Fade to Silence .............................................................................................................. 320

Chapter 8:  **Bringing a Game to Life with “Juice”** .... 322
Screen Effects.................................................................................................................. 324
Animations, Tweens, and Easing .................................................................................... 332
Particle Effects ............................................................................................................. 343
Camera Tricks ............................................................................................................... 348
Platformer Tricks ......................................................................................................... 352
More Tiles ..................................................................................................................... 362
Dialogs and Screen Transitions .................................................................................. 371
Designing Levels with a Level Editor ......................................................................... 376
Got Game ...................................................................................................................... 387

Chapter 9: **Optimizing & Packaging** ..................................................................... 389
Debugging ...................................................................................................................... 390
Profiling ......................................................................................................................... 396
Speeding Up Your Code ............................................................................................... 401
Speeding Up Rendering ............................................................................................... 413
Getting on Devices ....................................................................................................... 423
Getting on the Stores ................................................................................................... 437
Releasing to the World ................................................................................................. 439

Chapter 10: **Bonus Round: The Epilogue** ................................................................. 442
Preface

This book will teach you how to create awesome video games. Games from scratch. Games that run cross-platform, in web browsers, and on phones. Games filled with dynamic sound and music. Games overflowing with impressive visual effects. Fun games.

More importantly, this book will teach you how to think about making games. You'll learn to analyze and dissect games—to understand what makes great games great. By the end of the journey you'll have all the knowledge and tools needed to produce engaging, polished products that you can unleash on the internet, sell on app stores, or just use to impress your friends.

Aside from creating something cool, gamedev also tantalizes us with the possibility of becoming world-famous indie game superstars! In the ‘90s, if you wanted fame and fortune you had to start an alternative rock band. In the 2000s it was all about being in an electronic music production outfit. But today, all the cool kids are getting a gamedev studio together. The bad news is that you may have wasted all those years learning the guitar, but the good news is that you already know a bunch of JavaScript—and JavaScript is an excellent instrument in the quest for gamedev superstardom.

Whether you want to create games to express yourself and have an emotional impact on your audience, or whether you’re just digging for gold looking for the next Flappy Bird money-making hit, we all have to start at the same place: the beginning.

“The beginning” for us means establishing a solid codebase for our projects. We’ll use it to create a challenging little side-scrolling shoot-‘em-up. Then (having got our feet wet) we’ll evolve it into a tile-based action game. From there, we’ll add gravity to get a fast-paced, dungeon-crawler platform game. And finally, we’ll make an entertaining, mobile, penguin-based, physics-based, procedurally generated golf game. Each game builds gradually from the previous, leaving you with a solid foundation for constructing whatever quixotic ideas are swirling around your brain.

Then you’ll be a game developer.

Who Should Read This Book?

This book is for anyone who wants to create their own games using tools such as a browser and a text editor. It’s expected that you’re proficient in HTML and CSS, and are reasonably experienced with JavaScript, but no prior game development experience is assumed.
Acknowledgments

I'd like to thank Amelia for supporting (and tolerating) me throughout this long long journey. Really, thank you!

Also thanks to Simon Mackie for your professionalism, good ideas, and for being the voice of reason whenever I was stuck.

Thanks to Gaëtan Renaudeau for helping kickstart this project and giving it a great foundation to build on, and Franco Ponticelli and Andrzej Mazur for all your fantastic advice and formidable tech knowledge: it was great working with you both.

A huge high-five to everyone who let me use screen shots of their beautiful games and artworks in the book - it wouldn’t be the same without it, and I can’t wait to see what you all come up with next.

Shout-outs to the early-morning café crew: Warren and Madoo (for keeping my spirits up), and Anthony (for making good coffee).

Finally, thanks to Disasterpeace for the FEZ OST. I listened to it a million times over the course of this project. The perfect soundtrack for writing a gamedev book.

Conventions Used

You’ll notice that we’ve used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```html
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```
Tips, Notes, and Warnings

**Hey, You!**

Tips provide helpful little pointers.

**Ahem, Excuse Me ...**

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

**Make Sure You Always ...**

... pay attention to these important points.

**Watch Out!**

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- [https://github.com/spbooks/html5games](https://github.com/spbooks/html5games) is the book’s code archive, which contains code examples found in the book, plus game library that we’ll build in the book.
- [https://www.sitepoint.com/community/](https://www.sitepoint.com/community/) are SitePoint’s forums, for help on any tricky problems.
- books@sitepoint.com is our email address, should you need to contact us to report a problem, or for any other reason.
Press Space to Start
If you only have a few seconds to learn game development, here’s the super-condensed version. Fetch user input. Move everything just a tiny bit. Check if anything overlaps. Draw everything. Repeat this 60 times every second.

If you have a couple of extra minutes, then here’s the expanded version. A computer can do a lot of things, very quickly. It can update the individual positions of an army of enemy tanks, or flock of virtual bunnies. It can launch missiles or grow carrots or spawn beautiful particle effects. It can detect any collisions that occur. And it can draw everything on screen. All in one sixtieth of a second.

Game development is analogous to orchestrating a real-time, interactive, stop-motion animation film. Every screen update (called a frame, and generally lasting only around 16 milliseconds) the actors all move just a tiny amount. The tiny amounts add up, so playing them back at high speed creates the illusion of motion. Game development is almost entirely about creating illusions. If you want something to “walk”, first display an image of a character with open legs for a few frames, then switch to an image of closed legs for a few frames. If you’re careful about timing (and art style) you’ll have a convincing walk cycle.

![A headless walk cycle](image)

Animating and moving is only one thing we must do every frame. At the highest level, a main game loop (that runs once per frame) looks like this:
This high-level loop doesn’t explain how to coordinate and manage all those operations (we’ll cover them soon enough), but it’s a simple model for starting to think about how games work at their core.

This is all you need for a game. Don’t overthink it, don’t over-architect it. Just realize that a computer can do a lot of things in one sixtieth of a second, and a bunch of small animations and interactions quickly becomes more than the sum of its parts: the human brain is great at attributing life to inanimate objects. Start with a square moving across the screen, add other squares moving in the opposite direction, and it’s not much of a stretch of the imagination to see how it becomes *Super Mario Bros.*!

---

**This Might Seem Strange to You**

If you have a functional programming or web background, the idea of doing everything sequentially in a loop might seem a bit “procedural”, but that’s only at a high level. Games are ultimately a big ball of state. You can model that state however you want. But seriously: *keep it as simple as possible* for your first few games!
Game Design

How can we apply this high-level idea of a “main loop” to making something that actually resembles a game? Video game design, as a field, is still very young, but it’s had enough time to develop and refine a set of, well, not “laws”, but more like rules, guidelines, adages, best practices, academic theories, myths, and legends.

Game design (as distinct from game development) is about exploiting this collective knowledge in the pursuit of improving your games. Fortunately, you don’t need to be versed in the entire field of game design to start doing it effectively—just as you don’t need to know music theory to start a band and make great songs. But the more you learn, the faster you can figure out why some things do or don’t work. The more rules you know, the more rules you can break—and be reasonably confident in your choices.

I’m not going to delve too far into the nitty-gritty of game design; there are plenty of amazing resources for that. And, like all academic endeavors, it’s a deep rabbit hole once you get started. There are volumes devoted to simply defining what a “game” is! If you want to kick off the adventure, start with The Art of Game Design: A Book of Lenses by Jesse Schell. At this early stage, we’re primarily concerned with picking up the language of game design so we can move through our gamedev journey and all be on the same page.

Arguably the most important aspect of a game is its mechanics. A game mechanic refers to a specific gameplay element. It’s the goal or purpose of a single facet of your game—the thing that enforces the rules or encourages exploration to understand the game. Typically, you’ll have a whole bunch of mechanics that collectively comprise the overall gameplay, but the element (or elements) that define the essence of what makes your game special is called the core mechanic.

For example, in a platform game “jumping” is a mechanic (a pretty crucial one!). Perhaps in your game your core mechanic is that the player can jump further when they’re also running. This allows you, as the game designer, to invent situations where the player needs to choose if they should do a long jump or a short jump, and react accordingly. You need to be very careful when adding, modifying or removing mechanics, as these drastically change the entire shape of your game.

Closely related to game design is level design. Designing levels isn’t so much about making levels as thinking about making levels! Once you’ve defined your game’s mechanics, you can start to sculpt them into the overall experience you’re aiming to offer the player. Perhaps the first few levels will require a few long jumps, but they become more and more necessary as the game progresses—until the player is almost only performing long jumps. Then, at a crucial moment, throw in some alternating long/short/long jumps to shock and surprise them and keep them on their toes.
Level design is used to control the game’s pacing. **Pacing** is how the game flows over time. Where are moments of high tension? Where are moments of peacefulness? How naturally are they combined? Figure 1 shows the classic movie plot tension cycle. It can’t just be all about explosions and car chases, or it’ll quickly get dull. You have to vary the difficulty—perhaps starting easily and then ramping it up. But not for too long, else the player will get exhausted. There should be periods of relief, and surprise—highs and lows. Pacing is what stops the player from getting bored after they master the core mechanics.

1-4. Classic movie plot tension cycle

With our mechanics and pacing sorted, we need to talk about art. Throughout this book I’ll make a distinction between **graphics** and **aesthetics**. They are related, but very different. **Graphics** comprises the images that the artist creates—the pixels that make up the characters, the logos, the backgrounds. **Aesthetics** is concerned with the look and feel of the game at a higher level—the overall themes presented, the mood it generates, the use of colors, symbols, and imagery for artistic purposes.

Aesthetics is almost always more important than graphics. A game with a consistent and coherent aesthetic can be visually enjoyable and impressive even with less-than-stellar graphics. Which is great news for a lot of us who are terrible at Photoshop.

Finally, **juice**. This term probably won’t turn up in much academic literature, but it’s essential for creating great-feeling games. Juice is the polish, the attention to detail, the necessary unnecessaries, the love. It’s the shaking screen when things explode, the small particle trails following the player as they run, the smooth bouncing animations and effects in the user interface. Many of these are extremely easy to add on top of your core mechanic, and we’ll be adding heaps of them as we go along.
Now that we have the lingo down, let’s define our primary goal for the book (and for your first few games): to create a simple but interesting core mechanic, coupled with compelling level design, good pacing, a pleasing aesthetic, and a whole bunch of juice!

A Brief History

The viability of producing (and selling) games by yourself has waxed and waned over the years. The original commercial video games and home consoles (such as the Atari 2600, released in 1977) birthed many of the game genres we know today: platformers (running and jumping on platforms, collecting coins and treasure, and searching for the end of the level), RPGs (role-playing games, exploring rich worlds and developing your character’s skills), and side-scrolling shoot-'em-ups (blasting bad guys and getting more powerful weapons). All of these had their genesis as simple blocky squares in the ’70s.

The machines were new and complex beasts with very limited capabilities, and programming them required intimate hardware knowledge. The home computer revolution in the ’80s brought more power, better graphics, and easier development. It sparked the first wave of solo developers and small teams making smash-hit games.

The ’90s brought us the 16-bit era, and things got a lot more complicated. Mastering the hardware architecture was far more difficult, and the graphic and sound capabilities of the new machines were an order of magnitude more impressive than their 8-bit brethren. Game development teams got bigger and bigger, and the games themselves became expensive productions. One-person hits (such as Éric Chahi’s Another World, or Out of This World) became increasingly rare.
Advancements in hardware gave us the likes of the Nintendo 64 and the PlayStation—machines you couldn’t develop for even if you if you were a coding and graphics god. They required expensive, hard-to-obtain licensed dev kits and specialized development hardware. Giant game production houses ruled the gaming world. Employees in teams of thousands worked as tiny cogs in a mighty game-development machine, creating epic, movie-like productions. The era of the “AAA game” was here, and the days of the lone coder was over—banished to the world of “shareware” and small Flash games.

But then, in the mid-2000s, something strange happened. The world began to tire of AAA game franchises rehashing the same titles over and over again. Games started feeling stale and boring. Suddenly there was an opening for interesting and fresh ideas—even if they didn’t have the massive development budgets of Tired Shooter, XVII! Around the same time, the release of the original iPhone opened up a whole new market for mobile games—and this time players were actually willing to pay for them (unlike shareware). In another part of the internet, the opening of a digital-only game distribution store called Steam facilitated the online delivery of new games to millions of customers. The perfect storm had hit, bringing with it a tide of games made by small teams, without publishers or financial backing. The era of the “indie game developer” had begun.

Games and JavaScript

JavaScript and the Web are notably absent from this history. Games used the internet as a data transfer mechanism (and of course web browsers needed to host the Flash plugin) but that was about it. Where JavaScript was used for games, it was for simple point-'n'-click affairs—card and board games, or small experiments. That’s not to say it wasn’t possible to make “real” games back then. The 2004 release of DHTML Lemmings proved it was possible to create arcade-style games, a faithful reproduction of a classic game—including animations, sounds, and even particle explosions. As impressive as it was, it was regarded as a novelty rather than as the birth of a promising game development platform.
The biggest hurdle to games on the Web was the lack of pixel-level access. Everything had to be done via the DOM with standard HTML elements such as `div` containers and images. It wasn’t possible to create or manipulate existing graphics in any meaningful way. This changed with the introduction of the Canvas 2D API (which we’ll look at next chapter) and then WebGL (which we’ll also look at later). Suddenly you could draw shapes, manipulate bitmaps and create impressive effects, all at runtime. Small experiments were hatched, and the JavaScript games groundswell began.

It wasn’t so long ago that mentioning “games” and “JavaScript” in the same sentence would have you laughed out of any respectable game development forum. “Real games are written in C++”, the reasoning went. “C# may be acceptable, but just forget about dynamic languages: they’re too slow to do anything good!” … which, admittedly, was very true once, but much less so now. The introduction of Google Chrome sparked a whole new era of browser wars, each browser maker implementing faster and faster JavaScript and rendering engines, bringing near-native speeds to the Web.

The arguments against JavaScript have been evaporating … though don’t expect many of the low-level, old-school coders on the gamedev forums to have noticed yet! The shift of JavaScript from toy to seriously productive tool happened pretty fast, and most people weren’t watching. Now it’s a viable language for making big games that can easily be deployed to any platform.

### Strengths of JavaScript

The most obvious strength of JavaScript is that it’s everywhere. It’s unavoidable. It’s almost mandatory. Thanks to the twists and turns of computer history, JavaScript is one of the most popular languages in the world. If a device can be programmed, it can most likely run JavaScript.

Being ubiquitous is nice, but it’s not the reason why JavaScript is good for games.

JavaScript is a dynamic, prototype-based language. As games get larger, people are realizing the problems associated with traditional object-oriented programming. It can lead to complicated, incomplete class hierarchies that are inflexible and limit code reuse. There are many patterns and best practices surfacing to avoid these problems in gamedev, but the good news is that JavaScript comes with flexibility baked in by default. It lets us sculpt our requirements as we go, rather than define them from the beginning. We can change, tweak, and compose components and functions easily—which is excellent for prototyping and experimenting with game ideas.

We’re also witnessing a grand period of core language improvement. Since ECMAScript 2015 (aka ES2015, ECMAScript 6 and ES6), the language approval process has been sped up and streamlined, allowing for a whole host of improvements and features that address many of
JavaScript’s shortcomings, while building on the language’s strengths. We now have modules, classes, improved scoping powers, nicer syntax (and new features) for functions and objects, maps and sets, proxies ... the list is huge. But the upshot is that we can now write clearer, terser code with less chance of bumping into the historical warts of JavaScript.

The other huge benefit of JavaScript is its implicit partnership with the Web. Writing games in JavaScript means we get all the power of the Web for free. Every new feature and API added to the browser is immediately available for us to use in our games. Whatever we make can be easily distributed to, and accessed by, a huge global audience! We even get a few APIs specifically designed for making games: the Gamepad API implements game controller support; the Pointer Lock API lets us handle mouse movements like a native app (for making mouse-controlled games without losing focus); and, though not intended solely for games, the Web Audio API finally makes sound a first-class citizen on the Web.

Limitations of JavaScript

Every language and coding environment has its trade-offs. (That’s why there’s more than one language!) So it’s important to understand the good and the bad parts of your toolkit. The biggest weakness of JavaScript is also its strength: its dynamic nature.

Dynamic typing is great for playing and sculpting a game into shape. But once the shape is known, it’s hard to enforce it. For example, we might know that the player’s lives should be a number (`player.lives = 3`), but there’s nothing that stops us from saying `player.lives = “hello”`, even though it doesn’t make sense. Static typing helps us reason about the correctness of a program. If the compiler knows the “rules” of what every value can be, it can analyze your source and detect errors before the code is even run. A language with a strong type system avoids runtime messages like the classic `undefined is not a function`.

Bringing Static Typing to JavaScript

There are currently several attempts to bring static typing to JavaScript—the most popular being TypeScript and Flow. These tools allow you to add type information to your source code, which it uses to enforce the correct types. This code is converted to standard JavaScript before it’s deployed to the browser. Personally, I embrace the rock-and-roll dynamic nature of JavaScript—but you should certainly evaluate it for yourself.

Another fun part of web life is getting things working 100% correctly across all browsers. This can be especially hard when relying on bleeding-edge features not yet formalized by the W3C and not fully implemented by the browser makers. So although you can still play DHTML
Lemmings from well over a decade ago, many demos using brand new HTML5 features are prone to breaking, as it takes time for the specifications to be finalized, and browser makers interpret the requirements differently.

The cross-browser issues also extend to other platforms you might want to deploy to. JavaScript can run everywhere, but the capabilities and deployment process will be radically different between your browser and, say, your JavaScript-enabled toaster! It’s possible (and fun) to run your HTML5 games as native apps on phones, tablets, and consoles, but each platform has its own idiosyncrasies. While there are frameworks and services to make things easier, there’s no one-size-fits-all solution.

**The Future of HTML5 Games**

If you’re exploring your options for starting your game development career, HTML5 is an excellent choice. The features and power available to web-based games is constantly increasing and improving. It may not yet be as mind-blowingly fast as native C++ code targeting the latest low-level graphics APIs, but the gap is always closing. If you’re just starting out now, the Web is going to be ahead of your needs for a long time to come.

The Web has been a viable deployment target for many game studios for a long time now. Way back in 2012, Green Heart Games released their addictive “gamedev simulator”, Game Dev Tycoon on Steam, to almost universal praise; it still has 95% positive ratings today. A gamedev simulator game written in JavaScript in 2012 is pretty subversive!

Other early pioneers include Lost Decade Games’ action RPG, A Wizard’s Lizard, which was also released on Steam. These games were written from scratch (just as we’ll do in this book)—because there was no alternative! Later impressive efforts like Airscape: The Fall of Gravity, The Next Penelope and The Curious Expedition are all exquisite, very different games with plenty of depth. They proved that limited technology didn’t have to be a barrier to creating polished games.

More recently, there’s been a rise in so-called .IO games, which are casual MMOs (massively multiplayer online games) that generally have a .io domain name, hence the name. Examples include Slither.io, and the beautiful pixel-art, hack-‘n’-slash game Wilds.io, by JavaScript gamedev pathfinder Przemysław Sikorski (aka Rezoner).
If you look at these games through the various app stores, or read their promotional material, you won’t see a single mention of HTML5 or JavaScript—not because they aren’t proud of it, but because it’s not important! Ultimately, nobody cares what technology you use to create your game, as long as it’s good.

HTML5 simply provides a powerful, flexible ecosystem that serves as the foundation for making great games.

**Enough Talk; Let’s Make Games**

If you’re still here then you’re burning to make some games. It’s best to keep things as simple as possible so you can easily start working whenever some idea randomly pops into your brain. Our plan is to start completely from scratch and write everything from the ground up. As we progress, we’ll separate out elements we encounter that aren’t game-specific (that is, things that can be reused from game to game). These elements will go into a small library that we’ll create and build upon for future projects.

**These Are Just Guidelines**

PLEASE NOTE! Every single thing we do in this book—how we organize our project, the library we build, how we name our methods and write every line of code, the overall architecture, the number of enemy spaceships we spawn, and how many bullets it takes to explode them—it’s all just a guideline, a suggestion. Don’t be afraid to rip things up, do things your own way, move things around. Experiment relentlessly. The best way to get comfortable with new ideas is to rewrite them yourself. Feel free to follow along on our beaten path, but don’t be afraid to go off-road.
Let’s start our game with an HTML skeleton that all of our examples will derive from. Although we’re making games, we’re still using web technologies. We still need HTML, CSS, and JavaScript. The only difference is that we’ll combine them in a way that looks more like a game than a web page. Game development is all about creating illusions!

The general structure of our games will go like this:

![Game project structure diagram]

1-8. Base project structure

We have a source directory `src` for our game-specific files, `vendor` for any third-party libraries we might need, and finally `res` for game resources such as images, stylesheets, and audio files. The `index.html` file contains some very simple markup to get us started:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Game</title>
  <link rel="stylesheet" href="res/main.css" />
</head>
<body>
  <div id="board"></div>
</body>
</html>
```

Here’s a first reminder: this is just a guideline! You probably already know how you like to structure your HTML documents—so go with that. The only interesting piece is the `board` div that we’ll use as the container element for our games. (It’s called “board” because that’s what I started calling my “game board” area a million years ago, and it stuck.) It has some minimal styling via the stylesheet in `res/main.css` to center it on the page, and to stretch it to a desired dimension:
There are many tricks we can use to maintain the illusion our game is a cohesive, seamless artifact—but the power of CSS is one of the easiest. The background of the web page can be left a solid color, or it can be styled to match the aesthetic of the game. The area surrounding #board might be blank, or it might display colorful panels like the ones that decorated old arcade game cabinets. Use whatever works for your game, and whatever helps the player get lost in the experience. We won’t spend too much time on CSS—but don’t forget it’s always available to you.

The last step in our project skeleton is to add some code—a simple JavaScript file that will be the main entry point for our games. In honor of JavaScript’s shared C language heritage, we’ll save this in our src directory as main.js:

```javascript
const gameType = "awesome";
alert("Let's make ${gameType} games!");
```

Our first snippet of code may be unfamiliar to some. The `const` keyword (which defines a variable as constant, non-mutable) and the template literal strings (strings enclosed with backticks that allow embedded substitutions) are part of the ECMAScript 2015 standard for JavaScript (see the SitePoint book *JavaScript: Novice to Ninja* if you’d like to read more about newer JavaScript features). Our games will take advantage of modern JavaScript, so it’s important to make sure it’s executing correctly. To verify that it runs, include the source file as a JavaScript module in index.html:

```html
<body>
  <div id="board"></div>
  <script type="module" src="src/main.js"></script>
</body>
```
If you’ve got the popup alert working, that’s good enough for a project base! Let’s quickly hack some logic to make one of the least exciting games in the whole world: the good ol’ “What number am I thinking of?” game. Remove the `alert` and replace it with our game code:

```javascript
const myGuess = Math.floor(Math.random() * 20) + 1;
let guesses = 0;
let guess;
```

First we set up variables to hold the total number of guesses and pick a number between 1 and 20. Now we ask for the user’s guess and compare it to ours:

```javascript
while (guess !== myGuess) {
    guess = parseInt(prompt("What number am I thinking of?"), 10);
    guesses++;
    if (guess < myGuess) {
        alert("Higher.");
    } else if (guess > myGuess) {
        alert("Lower.");
    }
}
alert(`Well done! You got it in ${guesses}!`);
```

First parse the user’s guess as an integer. If the guess is too high or too low, alert the appropriate message. If it’s correct, say “Well done!” The game is boring (though slightly more funny if you change `myGuess` from 1-in-20 to 1-in-99999999!) and the user interface based on `alerts` and `prompt` dialogs is extremely annoying. But despite this, it contains the main elements we need for all games: an infinite loop, some game logic, and a render system.

---

**Modules**

At the time of writing, the JavaScript module implementation has been finalized as part of the ECMAScript 2015 standard, but browser support is not universal. Code samples for the book have also been compiled to “regular old JavaScript” via the Babel JavaScript compiler—but this is intended as a short-term solution that will not, eventually, be necessary. Consult the “Build Tools and Workflow” section later in this chapter for more information.

---

14 HTML5 Games: Novice to Ninja
1-9. What number am I thinking of?

The \texttt{while} loop runs until we get the answer correct: it’s our \textbf{main game loop}. The \texttt{prompt} is our \textbf{user input mechanism}; the game logic will determine if we have a \textbf{collision} (a correct guess); and \texttt{alert} is our \textbf{render system}, giving feedback to the player. To make a real game, we only need to swap out the alerts with image drawing, and input prompts with keyboard controls ... and hey presto—a video game!

Hopefully at this stage you’re psyched to get your big ideas on screen. But first we have a few administrative tasks we need to get in order before we go wild. We need to choose some basic tools, and we need to set up a productive project \textbf{workflow}. If at any point you’re feeling sleepy, skim forward to the aptly-titled section “Staying Motivated”.

\textbf{Making Art}

For our first few games, we can get by with basic geometric shapes—such as squares and circles—but we’ll quickly want to move on to something more impressive. And don’t be too quick to dismiss simple shapes either, as they can be used to good effect. The hit indie game \textit{Thomas Was Alone} was populated entirely by rectangles and circles—yet the \textit{aesthetic choices}, storytelling, and narration infused so much emotion and personality into those simple shapes.
But let’s assume that rectangles ain’t gonna cut it. We want some art. Perhaps you’re an artist who can draw well and you’re a wizard with graphical editing software ... But more likely (like me) you’ll be capable of producing programmer art. I could try to draw, say, a horse—and anyone I asked would probably guess that it’s a horse—but it’s not going to win any awards for “Best Horse Drawing”. If that’s also the case for you, it’s best to start off with either simple vector illustrations or low-res pixel art. Neither are “easy”, but with a bit of work you can produce some nice results—and with a lot of work you can produce some great results.

**Vector illustrations** are created by combining and manipulating basic shapes, polygons, and paths. Because the image is described by points (rather than actual pixels), things can be colored, stretched, resized, and edited without losing any information. Objects can easily be copy-pasted and reused without loss of quality.
There are several good vector-based editing software packages around. **Inkscape** is the go-to free and open-source vector editor I’d recommend. It’s under constant development and improvement, and you’ll find plenty of resources around to help master it. There are also some great commercial alternatives. Adobe Illustrator (the Photoshop of vector editing) is used by a lot of professional design houses, as are **Affinity Designer** and **Sketch**.

The concepts between all of these suites are basically the same—and general tutorials for any software will often be applicable to all. If you want to learn to start making graphics for your games, head over to the very excellent **2D Game Art for Programmers** site for some beginner-friendly vector editing tutorials on making all sorts of game assets.

---

**Rasterizing**

When using vector software, you’ll most likely need a **rasterizing** step, because the vectors will ultimately be displayed in your game as bitmap images (unless you’re using SVGs—which I’ll say a bit about in the next chapter). Usually, rasterizing simply requires selecting the **export as PNG** function from your vector software package.

---

If you’re looking for a different aesthetic, but still want to keep things **reasonably** simple, you can try your hand at low-res pixel art. **Pixel art** is artwork created at the **pixel level**—placing individual pixels by hand, as opposed to using large paint brushes that do things like antialiasing and smoothing for you. It’s a style that originated back in the early days of video games, when video game characters had to be designed on graph paper!
The actual technical definition of pixel art is not critical (you can argue about that on the internet), but for us it’s useful as an aesthetic option for creating games that feel “8-bit” and retro. It’s a style that I’m a big, unapologetic fan of!

Good pixel art is incredibly difficult to do. Reasonable-and-usable programmer pixel art is not so hard … and very good fun to try. The number-one tip when starting out is this: the worse you are at art, the lower you should keep the image resolution. If you have a large canvas for your game’s character, you need to hand place a lot of pixels, and you have to be good at drawing. If you make a tiny canvas—say 16x16 pixels—you can make more abstract, iconic representations of game characters. With a bit of work, they can even look pretty good! As you improve, you can up the canvas size. 16 pixels is really small, though, so you might need to resize the images before using them in the game.

To create pixel art, you have a few options. You can use a dedicated pixel art editor such as the excellent Aseprite editor, which is designed especially for pixel art. All of its tools and features work at the pixel level. You can also use more general image editors, such as GIMP (free and open source), Paint.NET (for Windows), and Pixelmator (for macOS)—which can work as pixel art editors, though you’ll need to configure their tools to draw and erase single pixels.
An easier option, of course, is to just use someone else’s art—a very tempting option when you can’t draw! There are various online resources that supply game art under various licenses. The most well known is Open Game Art, which has a large collection of user-submitted content that can be used in your games. Another is the free, ever-growing art collection of Kenney, the asset-producing virtuoso.

Please be careful when using art you haven’t created. Artwork (including music and sound effects) is covered by licensing and copyright law. Be sure to check the license attached to these works. You don’t want to get in legal trouble for releasing a game with assets you don’t have the right to use.
Using preexisting art can be good for motivation when you first start making games. Things instantly improve as your boring “blue square” becomes a cool, futuristic cyborg chicken ninja. Or whatever. The problem is, this approach doesn’t scale well. The chicken might be great, but when you want to add an underwater level for the chicken to explore, you’ll need to go looking for more assets. Even if you find some great underwater assets, they’ll have a style of their own. As your game grows (and you pull from more sources), things will become less consistent. You’ll have nice graphics, but bad aesthetics.

I promise that, in the end, it will be more rewarding to struggle through with your own programmer art (or at least find an artist to collaborate with). Any futuristic cyborg chicken ninjas will be your own.

Build Tools and Workflow

The games we’ll create will be written in standard JavaScript. To run them, you just load the game page in your web browser—in theory. If you’re reading this book in “the future” (that is, in the years following 2018, when I wrote this book) it will be true in practice too. Each code example has an `index-native.html` file that will run in modern browsers (you may need to still serve the page so it’s accessed via `http://` rather than `file://`). More details are available in the code repo). At the time of writing, support for some of the new JavaScript language features is not yet universal, so as boring as it sounds, we need to set up some tools for managing our games project.

The solution for working with tomorrow’s features today is to transpile our code into a form that works on older targets. Transpiling is a term that means to compile code to the same level of abstraction (rather than to a lower level, such as C to assembly). Transpiling is like converting source code to source code. We convert from source code that will only run in modern browsers to source code that will run in both new and older browsers.

As your projects become more complex, getting them into their final, runnable form becomes harder. Tools that manage your overall project are called build tools. They build your project from source and resources into the final product—transpiling and minifying your files, copying over art
assets, stripping out debugging information, running code-formatting verification tools across your codebase, and so on. Although a lovely aspect of JavaScript and the Web is that it’s not necessary to use any build tools, in reality you either need to run them (for example, to transpile to older targets) or often want to run them (to streamline the game-making workflow). In JavaScript land, this pretty much means using npm.

npm is a general package manager for JavaScript that’s part of the Node.js ecosystem. It’s a set of tools for working with JavaScript packages and projects—as well as a giant repository of other people’s code you can conveniently slot into your own projects! To install the ecosystem, head over to nodejs.org and follow the instructions on the front page. There are installers (and source code) available for macOS, Windows, and Linux.

Assuming the installation process went swimmingly and was pain free (the communities for Node and npm are very friendly if you get stuck), we can take npm for a test run. npm keeps track of meta information (such as which files belong to a specific project) by means of a special package.json file that lives in the root directory. You can manually add this file, or use the command line tool to generate one for you via npm init (from inside your project folder):

```bash
npm init
```

npm will respond with a wizard that asks you for some required and optional data about the project. Once done, it creates the package.json file with the metadata you entered. Third-party packages and tools can now be brought into your project by installing them with another npm command. For example, a handy package is the small web server (which we’ll be using in all of the examples in this book) called Budo. Budo is “a dev server for rapid prototyping”. It serves up our game page as if it were up in the cloud. It also has some extra cool features, like live reloading: the web page will magically reload whenever you save a change in your code. To install it, use the npm install command:

```bash
npm install --save-dev budo
```

`npm install <package name>` is how we install third-party packages for inclusion in our project. But notice we also added the flag --save-dev. This indicates that Budo is only needed while developing—that is, it’s not to be included when our game is finished and we want to distribute a final build. To run Budo, we’ll take advantage of another benefit of npm—the ability to run scripts for managing our project flow. In package.json, find the scripts object and add the following start command:

```json
"scripts": {
```
It instructs Budo to run and to load the `src/main.js` file. If this file includes any other files, it will also load them—and serve everything as a single `build.js` file.

Typing `npm start` will execute the command and serve our game up at a URL. If you navigate to the URL, the game will load! But actually, it will still be pointing at the module `src/main.js`—because that’s what we told it to do in `index.html`. To make it use our new `build.js` file, change the script tag in `index.html` to `<script src="build.js"></script>`. This will now fetch the bundled `build.js` project file.

That’s not very interesting on its own (though the live reloading is pretty cool), but we can now extend Budo to use features like Babel. Babel is the primary tool in the JavaScript world for transpiling our code for older targets, allowing us to use all the latest and greatest features from the new versions of the JavaScript language.

Babel is an extremely modular tool that makes it very flexible … but also unbelievably confusing to new users. If you install those three packages above (`babelify`, `babel-core`, and `babel-preset-env`) you should be good to go. To make Babel compile our code for older targets, modify the `start` command to pass flags we need for babelification:

```
  budo src/main.js:build.js --live -- -t [ babelify --presets [ env ] ]
```

**Bypassing Budo**

When it comes time to release your game, you’ll need to bypass Budo and use the mechanism it uses under the hood for running code in the browser: Browserify. This whole build-setup thing is getting boring, though, so I won’t go into detail. Have a look at the example repo for how I’ve done this, and run `npm run build` to save the `build.js` file to disk!
Phew, okay, yes. Setting up tools can be a drag—but once everything’s running smoothly, these tools become invaluable. We can grab helpful libraries from other people and easily integrate them into our games, we can extend the `npm scripts` sections to automate and simplify our development workflow, and we can export our own code packages that other people around the world can use. Pretty cool.

**Version Control Systems**

Regardless of your build system (or lack thereof), you’ll also eventually need to master a version control system (VCS). A VCS tracks changes in your codebase over time. When you make some changes to your code, you commit the change with a message about what you did. This might sound a bit annoying, but it allows you to revert to any point in your project’s history and run your game from there. This feature is an absolute lifesaver when you make some breaking change at 3:00 a.m. before a hard deadline and can’t for the life you figure out what you did to break it. It also makes working with teams a lot more manageable, as you can all work on the same codebase and merge your changes as you go.

There are lots of systems available, but one of the most popular is Git. It was created by Linus Torvalds (creator of Linux) as a tool for distributed developers to work on the Linux kernel. It’s mostly a command line tool, but there are graphical interfaces available for all platforms. These days, Git support is often built directly into your code editor, so you can commit changes without leaving your code.

Git also spawned the fantastic online service GitHub. GitHub provides an easy way to store and share your version-controlled code with the world. There are lots of resources available if you’re just getting started, such as ReadWrite’s [GitHub For Beginners](https://www.readwrite.com/2014/04/25/what-is-git-how-to-get-started) article.

**Valuable Time Spent**

Getting up to speed with a VCS and various build tools is one of the best investments you’ll make in your developer life. Once you master them, you’ll wonder how you survived without them. But to play devil’s advocate for a moment, I’ll also point out that time spent learning project tools is time spent not developing your game. If you’ve never made a game before, I won’t tell anyone if you postpone learning Git for a while in favor of getting something moving on the screen!

**Staying Motivated**

The final thing you need before we begin is motivation. Motivation is what drives you to get stuff done. Perhaps your motivation is just money—creating a breakout hit on the App Store and never
having to work again. Perhaps it’s fame—to be recognized in the gamedev field as a developer rockstar. Perhaps it’s art—to make something the world has never seen before, or challenge society’s beliefs. Perhaps you just want to finish a game.

Because we’re all different, and we have a long way to journey together, I’ll just assign us a common group motivation for the duration of this book. Here goes …

The evil AAA game studio conglomerate, Exploitative Games, Inc., has decided it’s time to crush the indie game market … and all the developers in it. This concerns you directly, because you’re a professional indie game developer. You’ve decided to quit your day job (and in spectacular fashion: you can never go back) to start your very own game development studio, MomPop Games.

Having procured some co-working space, an old laptop, and some takeout coffee, you arrive on your first day of your new life.

Upon entering the lobby, you bump into an ominous, white-suited figure who you recognize instantly as Guy Shifty, CEO and owner of the world-renowned Exploitative Games, Inc. (which just happens to be located in all of the 34 floors above your office). He pauses and looks at you suspiciously for some time. An associate leans over and whispers something into his ear. Finally he says, cryptically, “Don’t get too settled in. I plan to become king of the indie scene and get rid of the amateurs like you once and for all … MomPop Games.”

He and his entourage shuffle past, leaving you staring at your coffee.

Looks like we’ve got a bit of friendly competition on our hands! We’d better get started.

Get Ready!

Now that we’re all suitably motivated, here we go. Our mission is clear: we need to learn how to create amazing, captivating, successful video games, and we need to learn fast. We already have our project set up and ready to go. Our workflow is in place. We’ve got the tools, we’ve got the talent. Now we just need some good ideas, and a tiny bit of knowledge.

Fortunately, in just a few hundred pages you’ll walk out of here with the know-how for creating absolutely any hair-brained game you can imagine.