Chapter 1
Gaming on the Web

in this chapter

- Figuring out what HTML5 is and where it came from
- Seeing HTML5 in the context of games
- Looking at important new features
- Enabling feature detection and dealing with legacy browsers
BEFORE I DIVE into code, I want to establish the context of the technology we use. In this first chapter, I discuss what HTML5 is as well as some of the history that led to the HTML5 specification.

One of the most interesting aspects of HTML5 is how game developers can profit from many of the new features. In this chapter, I introduce some of these features and give a few quick examples of how they are used. I talk about the canvas element and WebGL and the huge improvements these additions make in terms of our ability to create dynamic graphics. I also cover the audio element and the added multiplayer possibilities created by the WebSockets specification.

Everybody likes new toys, but we mustn’t forget that in the real world, old and outdated browsers keep many users from using these bleeding-edge features. In this chapter, I show a few helpful tools for detecting which features you can safely use as well as how you can use these feature tests to load appropriate fallback solutions when necessary.

Finally, I briefly introduce the puzzle game that I use throughout the rest of the book to take you through the creation of a complete HTML5 game.

Tracing the History of HTML5

HTML, the language of the Web, has gone through numerous revisions since its invention in the early 1990s. When Extensible Markup Language (XML) was all the rage around the turn of the millennium, a lot of effort went into transforming HTML into an XML-compliant language. However, lack of adoption, browser support, and backward compatibility left the Web in a mess with no clear direction and a standards body that some felt was out of touch with the realities of the Web.

When the W3C finally abandoned the XHTML project, an independent group had already been formed with the goal of making the Web more suitable for the type of web applications that we see today. Instead of just building upon the last specification, the Web Hypertext Application Technology Working Group (WHATWG) began documenting existing development patterns and non-standard browser features used in the wild. Eventually, the W3C joined forces with the WHATWG. The two groups now work together on bringing new and exciting features to the HTML5 specification. Because this new specification more closely reflects the reality of how web developers already use the Web, making the switch to HTML5 is easy too. Unlike previous revisions, HTML5 doesn’t enforce a strict set of syntax rules. Updating a page can often be as easy as simply changing the document type declaration.
But what is HTML5? Originally, it simply referred the latest revision of the HTML standard. Nowadays, it is harder to define as the term has gone to buzzword hell and is now used to describe many technologies that are not part of the HTML5 specification. Even the W3C got caught up in the all-inclusiveness of HTML5. For a brief period, they defined it as including, for example, Cascading Style Sheets (CSS) and Scalable Vector Graphics (SVG). This only added to the confusion. Fortunately, the W3C later backed off of that stance and went back to the original, stricter definition that refers only to the actual HTML5 specification. In a somewhat bolder move, the WHATWG simply dropped the numeral 5, renaming it simply HTML. This actually brings it much closer to reality, in the sense that specifications such as HTML are always evolving and never completely supported by any browser. In this book, I just use the term HTML for the most part. You can assume that any mention of HTML5 refers to the actual W3C specification called HTML5.

Using HTML5 for Games

Many features from the HTML5 specification have applications in game development, but one of the first features to gain widespread popularity was the canvas element. The visual nature of this element without a doubt helped it spread quickly when the first interactive animations and graphics effects started appearing. More advanced projects soon followed, giving the new standard a dose of good publicity and promising a future with a more dynamic and visually interesting web.

Canvas

Hobbyist game developers were also among the first to embrace HTML5, and for good reason. The canvas element provides web game developers with the ability to create dynamic graphics, giving them a welcome alternative to static images and animated GIFs. Sure, people have created more or less ingenious (and/or crazy) solutions in lieu of better tools for creating dynamic graphics. Entire drawing libraries rely on nothing more than coloured div elements—that may be clever, but it’s not very efficient for doing anything more than drawing a few simple shapes. Uniform Resource Identifier (URI) schemes exist that let you assign source files to img elements, for example, using a base64-encoded data string, either directly in the HTML or by setting the src or href property with JavaScript. One of the clever uses of this data: URI scheme has been to generate images on the fly and thus provide a dynamically animated image, which is not a great solution for anything but small and simple images. Wolf 5K, the winner of the 2002 contest The 5K, which challenged developers to create a web site in just 5 kilobytes, used a somewhat similar technique. The game, a small 3D maze game, generated black and white images at runtime and fed them continuously to the image src property, relying on the fact that img elements can also take a JavaScript expression in place of an actual URL.
Graphics drawn on a canvas surface cannot be declared with HTML markup but rather must be drawn with JavaScript using a simple Application Programming Interface (API). Listing 1.1 shows a basic example of how to draw a few simple shapes. Note that the full API provides much more functionality than the small portion shown in this example.

**Listing 1.1  Drawing Shapes with the Canvas API**

```html
<canvas id="mycanvas"></canvas>
<script>
  var canvas = document.getElementById("mycanvas"),
      ctx = canvas.getContext("2d");

  canvas.width = canvas.height = 200;

  // draw two blue circles
  ctx.fillStyle = "blue";
  ctx.beginPath();
  ctx.arc(50, 50, 25, 0, Math.PI * 2, true);
  ctx.arc(150, 50, 25, 0, Math.PI * 2, true);
  ctx.fill();

  // draw a red triangle
  ctx.fillStyle = "red";
  ctx.beginPath();
  ctx.moveTo(100, 75);
  ctx.lineTo(75, 125);
  ctx.lineTo(125, 125);
  ctx.fill();

  // draw a green semi-circle
  ctx.strokeStyle = "green";
  ctx.beginPath();
  ctx.scale(1, 0.5);
  ctx.arc(100, 300, 75, Math.PI, 0, true);
  ctx.closePath();
  ctx.stroke();
</script>
```

The code produces the simple drawing shown in Figure 1-1.
I revisit the canvas element in Chapter 6 and explore it in detail when I use it to create game graphics and special effects.

Audio

Just as welcome to the toolbox of the web game developer as the canvas element is the new audio element. Finally, we have native audio capabilities in the browser without resorting to plug-ins. A few years ago, you could be almost certain that if a web site had audio, some form of Flash would be involved. Libraries like the SoundManager 2 project (www.schillmania.com/projects/soundmanager2/) provide full JavaScript access to most of the audio features of Flash. But even if such a bridge allows your own code to stay on the JavaScript side, your users still need the plug-in installed. The HTML5 audio element solves this problem, making access to audio available in browsers out of the box using only plain old HTML and JavaScript.

The audio element has a few issues still to be resolved, however. The major browser vendors all seem to agree on the importance of the element and have all adopted the specification, but they have so far failed to agree on which audio codecs should be supported. So, while the theory of the audio element is all good, reality has left developers with no other option than to provide audio files in multiple formats to appease all the browsers.

The audio element can be defined both in the mark-up and created dynamically with JavaScript. (The latter option is of more interest to us as application and game developers.) Listing 1.2 shows a basic music player with multiple source files, native user interface (UI) controls, and a few keyboard hotkeys that use the JavaScript API.

Listing 1.2  A Simple Music Player with HTML5 Audio
<audio controls id="myaudio">
  <source src="Prelude In E Minor, Op. 28.ogg"/>
  <source src="Prelude In E Minor, Op. 28.mp3"/>
</audio>
<script>
  var audio = document.getElementById("myaudio");
  document.onkeydown = function(e) {
    if (e.keyCode == 83) {
      continued
Listing 1.2  continued
    audio.pause(); // Key pressed was S
} else if (e.keyCode == 80) {
    audio.play(); // Key pressed was P
}
</script>

REMEMBER  Audio data APIs that will eventually allow dynamically generated sound effects and audio filters are in the works at both the Mozilla and WebKit camps. Because these APIs are still very early in their development, I won’t be using them for the games in this book, although I briefly examine the possibilities they present in Chapter 10 when I dive into HTML5 audio.

WebSockets
Apache and the XMLHttpRequest object that is at its heart brought new life to the web with the Web 2.0 explosion of the early 2000s. Despite the many great things it has enabled, however, it is still painfully limited. Being restricted to the HTTP protocol, the action is rather one-sided, as the client must actively ask the server for information. The web server has no way of telling the browser that something has changed unless the browser performs a new request. The typical solution has been to repeatedly poll the server, asking for updates, or alternatively to keep the request open until there is something to report. The umbrella term Comet ([en.wikipedia.org/wiki/Comet_(programming)](en.wikipedia.org/wiki/Comet_%28programming%29)) is sometimes used to refer to these techniques. In many cases, that is good enough, but these solutions are rather simple and often lack the flexibility and performance necessary for multiplayer games.

Enter WebSockets. With WebSockets, we are a big step closer to the level of control necessary for efficient game development. Although it is not a completely raw socket connection, a WebSocket connection does allow us to create and maintain a connection with two-way communication, making implementation of especially real time multiplayer games much easier. In Listing 1.3, you can see that the interface for connecting to the server and exchanging messages is quite simple.

Listing 1.3  Interacting with the Server with WebSockets
    // Create a new WebSocket object
    var socket = new WebSocket("ws://mygameserver.com:4200/");

    // Send an initial message to the server
    socket.onopen = function () {
socket.send("Hello server!");

// Listen for any data sent to us by the server
socket.onmessage = function(msg) {
    alert("Server says: " + msg);
}

Of course, using WebSockets requires that we also implement a server application that is compatible with the WebSockets protocol and capable of responding to the messages we send to it. This doesn’t have to be a complex task, however, as I show you in Chapter 13 when I implement multiplayer functionality using Node.js on the server side.

**Web Storage**

Cookies are the usual choice when web applications need to store data on the client. Their bad reputation as spyware tracking devices aside, cookies have also given developers a much-needed place to store user settings and web servers a means of recognizing returning clients, which is a necessary feature for many web applications due to the stateless nature of the HTTP protocol.

Originally a part of HTML5 but later promoted to its own specification, Web Storage can be seen as an improvement on cookies and can, in many cases, directly replace cookies as a larger storage device for key-value type data. There is more to Web Storage than that, however. Whereas cookies are tied only to the domain, Web Storage has both a local storage that is similar to cookies and a session storage that is tied to the active window and page, allowing multiple instances of the same application in different tabs or windows. Unlike cookies, Web Storage only lives on the client and is not transmitted with each HTTP request, allowing for storage space measured in megabytes instead of kilobytes.

Having access to persistent storage capable of holding at least a few megabytes of data comes in handy as soon as you want store to any sort of complex data. Web Storage can only store strings, but if you couple it with a JavaScript Object Notation (JSON) encoder/decoder, which is natively available in most browsers today, you can easily work around this limitation to hold structures that are more complex. In the game I develop during the course of the book, I use local Web Storage to implement a “Save Game” feature as well as to store local high score data.

Listing 1.4 shows the very simple and intuitive interface to the storage.
Listing 1.4  Saving Local Data with Web Storage

// save highscore data
localStorage.setItem("highscore", "152400");

// data can later be retrieved, even on other pages
var highscore = localStorage.getItem("highscore");

alert(highscore); // alerts 154200

WebGL

WebGL is OpenGL for the web. The most widely used graphics API is now available for web developers to create online 3D graphics content. Of course, this has major implications for the kind of web games that are now possible. As a testament to this significance, Google developers released a WebGL port of the legendary first-person shooter, Quake II, on April 1, 2010, to general disbelief due to both the carefully chosen release date and the achievement itself.

REMEMBER  Creating a full 3D game is a rather complex task and does not fit within the scope of this book. Using WebGL also requires developers to be very aware of the platforms they plan to target because Internet Explorer, for example, has no support whatsoever and the problem can’t be worked around with polyfills. WebGL is not used much in this book because it is not yet supported on mobile platforms such as iOS or Android, although it will be one day. I do however use a bit of WebGL to create an interesting intro screen for those with WebGL-enabled browsers.

HTML5 is (not) a Flash killer

Ever since the arrival of the canvas element and the improved JavaScript engines, the Internet has seen discussions and good old flame wars over whether the new standards would replace Flash as the dominant delivery method for multimedia applications on the web. Flash has long been the favorite choice when it comes to online video, music, and game development. Although competing technologies such as Microsoft’s Silverlight have tried to beat it, they have only made a small dent in the Flash market share. HTML5 and its related open technologies now finally look like a serious contender for that top spot.

It appears that Adobe, too, has acknowledged the power of HTML5 because they are currently preparing the launch of Adobe Edge (http://labs.adobe.com/technologies/edge/), a development environment very similar to Flash but based fully on HTML5, CSS3,
and JavaScript. Adobe’s position seems to be that HTML5 and Flash can coexist for the time being but perhaps this is a sign that Flash will be phased out in the long term in favor of HTML5 and the open web.

That is not to say that HTML5 is a drop-in replacement for Flash. You must know where the new standards fall short and where alternative solutions like Flash might be more appropriate. One area where Flash is still very handy is to ensure backward compatibility with older browsers, which I talk about next.

Creating Backward Compatibility

As with most other new technologies, issues with backward compatibility inevitably show up when working with HTML5. HTML5 is not one big, monolithic thing: Browsers support features, not entire specifications. No browsers today can claim 100 percent support for all of the HTML5 feature set and Internet Explorer, still the most widely used browser, has only begun HTML5 support with its newest version, 9. Some features like WebGL are not implemented at all, and Microsoft has so far not shown any interest in doing so. However, even if the current crop of browsers fully supported HTML5 and the related standards, you still have to think about legacy browsers. With browsers like Internet Explorer 6 still seeing significant use today, a decade after its release in 2001, you can’t safely assume that the users of your applications and games can take advantage of all the features of HTML5 for many years to come.

Feature detection

No one says that the applications and games we build today must support all browsers ever released — doing so would only lead to misery and hair-pulling. We shouldn’t just forget about those users, though. The least we can do is try to tell whether the user is able to play the game or use a certain feature and then handle whatever problems we detect. Browser sniffing — that is, detecting what browser the user is using by examining its user agent string — has almost gone out of style. Today, the concept of feature detection has taken its place. Testing for available properties, objects, and functions is a much saner strategy than simply relying on a user changeable string and assuming a set of supported features.

Using the Modernizr Library

With so many discrepancies in the various implementations and features that can be tricky to detect, doing adequate feature detection is no simple task. Fortunately, you don’t usually need to reinvent the wheel because many clever tricks for detecting feature support have
already been developed and aggregated in various libraries. One collection of these detectors is available in the Modernizr library (www.modernizr.com/). Modernizr provides an easy-to-use method of testing whether a certain feature is available or not. You can detect everything from the canvas element and WebGL to web fonts and a whole slew of CSS features, allowing you to provide fallback solutions where features aren’t supported and to degrade your application gracefully.

Using the Modernizr library is dead simple. Simply include a small JavaScript file and you can start testing for features by evaluating properties on the Modernizr object, as seen in Listing 1.5.

Listing 1.5  Detecting Features with Modernizr

```javascript
if (Modernizr.localstorage){
  // local storage is supported, carry on.
} else {
  // no local storage support, use a fallback solution.
}
```

Modernizr also adds CSS classes to the html element, indicating which features are supported and which are not. Because all other elements are children of the html element, this allows you to easily style your markup appropriately according to the supported features. For example, if you want to use rgba() colors to make a semi-transparent background and fall back to an opaque background if rgba() colors are not supported, you could create styles as shown in Listing 1.6.

Listing 1.6  Applying CSS Based on Supported Features

```css
/* rgba() colors are supported */
.rgba .some_element {
  background-color : rgba(255,255,255,0.75);
}

/* rgba() colors are not supported */
.no-rgba .some_element {
  background-color : rgb(220,220,220);
}
```
Modernizr can only tell you whether something is supported; it is up to you, the developer, to make sure that the appropriate action is taken if the desired feature is unavailable. In the next section, I examine some of the options for dealing with missing features.

**Filling the gaps with polyfills**

Beginning in the early 2000s, a popular trend has been to favor so-called *progressive enhancement* when adding new features to web sites. This strategy calls for web sites to target the lowest common denominator in terms of supported features. Any technology that is not supported across the board should be used only to add enhancements to the site, never critical functionality. This ensures that everyone can access and use the web site. If the user has a modern browser, they simply get a better experience.

Progressive enhancement is a sound strategy in many cases, but sometimes you simply need to use a certain feature. If some browsers have no native support for that feature, that hole must be plugged even if it means using less than ideal or even hackish fallback solutions. These fallbacks are sometimes called *polyfills*, named after the spackling paste Polyfilla, as their function is somewhat similar. They fill out the cracks in the supported feature sets when running your code in actual browsers, bridging the gap between specifications and the reality of dealing with non-perfect browsers.

As an example, Internet Explorer had no support for *canvas* until IE9, but several polyfills exist that provide various amounts of *canvas* functionality for legacy browsers. One of the earliest of these polyfills is the ExplorerCanvas project from Google (http://code.google.com/p/explorercanvas/). It uses Vector Markup Language (VML), an old Microsoft developed XML-based language, to simulate a *canvas* element. It provides enough 2D drawing functionality that it has been used successfully in many projects. Some features are missing, however, because VML does not perfectly overlap the *canvas* specification and lacks support for, for example, patterns and clipping paths. Other polyfills use Flash or Silverlight to get even closer to the full *canvas* API, letting you use advanced features like image data access and compositing effects. With all these different options, picking the right fallback solutions is no easy task. Depending on the target platforms, sometimes even the polyfills need fallbacks.

To make things easier, you can turn to Modernizr’s built-in script loader. The script loader is based on a small library called yepnope.js (http://yepnopejs.com/) that combines dynamic script loading techniques with a feature tester like Modernizr. This combination allows you to perform automatic, conditional script loading based on available features. Listing 1.7 shows how different scripts can be loaded depending on the support of a feature.
Listing 1.7  Loading scripts conditionally with Modernizr

```javascript
Modernizr.load([{
    test : Modernizr.localstorage,
    yep : "localstorage-data.js",
    nope : "cookiebased-hack-data.js"
}]);
```

The `Modernizr.load()` function takes a list of objects, each describing a feature test that determines which script(s) should load. The script specified by the `yep` property loads if the feature is supported; the `nope` script loads if it is not. The `yepnope.js` functionality includes many other cool features that can help you streamline the loading stage, letting you load both JavaScript and CSS files based on feature support, browser make and version, or even something completely different.

**Building a Game**

Starting with Chapter 2 and throughout the rest of the book, I take you through the process of developing an HTML5 web game from scratch. I show you how to create a match-three gem-swapping game in the style of Bejeweled or Puzzle Quest, casual games that have been very popular on many platforms over the past decade. This type of game has tried-and-tested game mechanics, allowing us to focus our attention on the use of web technologies in the context of game development. Additionally, these games play well in desktop browsers as well as on mobile devices such as smart phones and tablets, giving us opportunity to explore multiplatform web game development.

The game takes advantage of several features from the HTML5 specification, but also uses related technologies such as web fonts and CSS3 for building the UI. Although the game itself might not be revolutionary, it allows me to cover many of the newest advances in open web technology. Among other things, I use the `canvas` element to generate some of the game graphics and I show you how to add sound effects using HTML5 audio. The finished game will be playable in a regular browser but I show you how to ensure that it plays just as well on mobile devices and even offline. I show you how to use Web Storage to save high score data and to allow players to pick up where they left.
The canvas element lets us create interesting dynamic graphics, but it's not always suitable for creating user interfaces. You don't really need any new tools for that part, however, because traditional HTML and CSS gives us all you need to build great UI. With the latest additions to the CSS specification, you can add animations, transforms, and other features that bring life to the UI experience. In Chapters 6 and 7, I show you how to build the display module with the canvas element. Later on, in Chapter 11, I take you a bit further as I show you how to use WebGL to add 3D graphics to the game.

In Chapter 13, I show you how to create a simple chat application using WebSockets. For this purpose, I also show you how to develop a small server application using the Node.js framework (http://nodejs.org/). However, you should note that WebSockets are only supported in a few browsers and it might be a while before it is ready for prime time use.

Summary

It's been a bumpy road but it finally looks like HTML and the web in general is on the right track again. The WHATWG brought in some fresh perspective on the standards process, and web developers are now beginning to enjoy the fruits of this undertaking in the form of a plethora of new tools and a HTML standard that is more in line with how the web is used today. As always, using new features requires dealing with older browsers but many polyfills are already available that you can use to ensure cross-browser compatibility. Using a helper like Modernizr eases the burden of this task even further.

Many of the new additions are of special interest to game developers because real alternatives to Flash-based web games are now available. Canvas and WebGL bring dynamic and hardware accelerated graphics to the table, the audio element has finally enabled native sound, and with WebSockets, it is now possible to create multiplayer experiences that are much closer that of desktop games than what was possible just a few years ago. Advances in other, related areas like CSS and the increasing support for web fonts let us create richer UI experiences using open, standardized tools.