THE EXPERT'S VOICE® IN GAME DEVELOPMENT

HTML5 Game Development Insights

TAKE YOUR HTML5 GAME DEVELOPMENT TO THE NEXT LEVEL WITH TIPS AND ADVICE, RIGHT FROM THE PROS!

Colt McAnlis, Peter Lubbers, Brandon Jones, Andrzej Mazur, Sean Bennett, Bruno Garcia, Shun Lin, Ivan Popelyshev, Jon Howard, Ian Ballantyne, Takuo Kihira, Jesse Freeman, Tyler Smith, Don Olmstead, Jason Gauci, John McCutchan, Chad Austin, Mario Andres Pagella, Florian d'Erfurth, and Duncan Tebbs

Apress®

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

Apress®

Contents at a Glance

Introduction

Making games is *hard*.

Even most veteran game developers don't fully grasp the scale of how difficult it is to weave together technology, code, design, sound, and distribution to produce something that resonates with players around the world. As industries go, game development is still fairly young, only really gaining traction in the early 1980s. This makes it an even more difficult process, which, frankly, we're still trying to figure out.

In 30 years of game development, we've seen the boom of console games, computer games, Internet bubbles, shareware, social gaming, and even mobile gaming. It seems that every five to eight years, the entire industry reinvents itself from the core in order to adjust to the next big thing.

As hardware trends shift and user tastes change, modern game developers scramble to keep up, producing three to four games in a single year (a feat unheard of in 2001, when you thought in terms of shipping two to three games in your entire *career*). This rapid pace comes at a high cost: engineers often have to build entire virtual empires of code, only to scrap them a mere six weeks later to design an entirely different gameplay dynamic. Designers churn through hordes of ideas in a week in order to find the smallest portion of fun that they can extract from any one idea. Artists also construct terabytes of content for gameplay features that never see the light of day.

A lot of tribal knowledge and solutions get lost in this frantic process; many techniques, mental models, and data just evaporate into the air. Tapping into the brains of game developers, cataloging their processes, and recording their techniques is the only real way to grow as an industry. This is especially relevant in today's game development ecosystem, where the number of "indie" developers greatly outnumbers the "professional" developers.

Today we're bombarded with messaging about how "it's never been easier to *make* a game," which is true to some extent. The entry barrier to *creating* a game is pretty low; eight-year olds can do it. The real message here is what it takes to make a *great* game. Success comes from iteration; you can't just point yourself in a direction, move toward it, and expect your game to be great. You have to learn. You have to grow. You have to *evolve*. Moreover, with less and less time between product shipments, the overhead available to grow as a developer is quickly getting smaller and smaller. Developers can't do it on their own; they need to learn, ask questions, and see what everyone else is doing. As a developer, you have to find mentors in design, marketing, and distribution. You have to connect with other people who feel your pain, and who are trying to solve the same problems and fight the same battles. Evolve as a community, or die as an individual.

Making games *is* hard. That's why we wrote this book; even the best of us must find time to learn.

—Colt McAnlis

HTML5 has come a long way.

It might be hard to believe today, but getting publisher support for *Pro HTML5 Programming*, the book I co-authored with Brian Albers and Frank Salim in 2009, and released as one of the first books on the subject in 2010, was quite hard. Publishers were just not sure if this new HTML5 thing had a future or if it was just a passing fad.

The launch of the iPad in April 2010 changed all that overnight and drove the curiosity and excitement about HTML5 to a whole new level. For the first time, many developers started to look seriously at the new features and APIs, such as canvas, audio, and video. The possibility of many kinds of new web applications with real native feature support seemed almost too good to be true. And, to a certain extent, it was.

When developers seriously started to dig into the new APIs, they discovered many missing pieces. Features that had long been staples of other platforms were now lacking, or were implemented in such a way that they were not very useful. This disappointed many developers, and yet they were eager to improve on the HTML5 feature set. That cycle is, of course, the nature of development and the impetus for innovation.

Game software, perhaps more than any other genre, tends to stress its host platform to the max, so it was not surprising that there was some backlash to the initial hype that HTML5 was the be-and-end-all for every application on the web. However, that was never the intention of HTML5. In fact, one of the core design principles behind HTML5 is "evolution not revolution," and it is the slow but steady progress of features, spanning many years, that has changed the HTML landscape.

Nevertheless, browser vendors and spec authors have not been sitting still. Instead, they have developed many new and more powerful APIs. One example is the Web Audio API, now shipping in many of the major browsers. This API offers fine-grained audio manipulation, which the regular audio element could not provide. With this and other new APIs, it is now much easier to develop applications and web-based games that, until recently, would have been hard to imagine, let alone code.

That is why I believe we're just at the beginning of a future full of great possibilities in web-based game software. Of course, we'll never be "done." There will always be room for improvement but, as my esteemed co-authors clarify in this book, you can now build compelling games that leverage the power and flexibility of the web platform in ways that were unheard of even a few years ago.

Code, learn, improve, and repeat. Be a part of software evolution at its best.

—Peter Lubbers

CHAPTER 1

JavaScript Is Not the Language You Think It Is

Sean Bennett, Course Architect, Udacity

JavaScript is a deceptively familiar language. Its syntax is close enough to C/C++ that you may be tricked into thinking it behaves similarly.

However, JavaScrit has a number of gotchas that trip up developers coming from other languages. In this chapter, I'll go over some of the more egregious offenders, teach you how to avoid them, and showcase the hidden power in the language.

This chapter is for game programmers who are coming from other languages and who are using JavaScript for the first time. It's the tool I wish I'd had when I first started using JavaScript.

Variables and Scoping Rules

You wouldn't think that declaring variables would be at all hard or error prone. After all, it's a fundamental part of any language, right? The problem is that with JavaScript, it's

- very easy to accidentally declare variables after they're used, which leads to accidentally accessing undefined variables
- deceptively difficult to restrict access to variables, leading to naming collisions as well as memory allocation issues

I'll discuss the issues with and limitations of JavaScript scoping and then present a well-known solution for modularizing your JavaScript code.

Declaration Scoping

The first thing you need to realize about JavaScript is that there are only two different scopes: global and function level. JavaScript does not have any further lexical or block scoping.

A variable is declared on the global scope like so:

```
zombiesKilled = 10;
```

A variable is declared on the function scope as follows:

```
var antiPokemonSpray = true;
```

That's not entirely true, actually. Using the `var` keyword attaches the variable to the nearest closing scope, so using it outside any function will declare the variable on the global scope as well.

Note that the lack of any block-level scoping can cause bugs that are pretty hard to track down. The simplest example of this is the use of loop counters; for instance,

```
for (var i = 0; i < 10; i++) {
    console.log(i);
}
console.log(i);
```

That last logging statement won't output `null` or `undefined`, as you might expect. Because of the lack of block scope, `i` is still defined and accessible. This can cause problems if you don't explicitly define the value of your loop counter variables on every reuse.

Global scope is something to be avoided in JavaScript. Not only do you have all the usual reasons, such as code modularity and namespacing issues, but also JavaScript is a garbage-collected language. Putting everything in global scope means that nothing ever gets garbage collected. Eventually, you'll run out of memory, and the memory manager will constantly have to switch things in and out of memory, a situation known as memory thrashing.

Declaration Hoisting

Another concern with variable declarations is that they're automatically hoisted to the top of the current scope. What do I mean by that? Check this out:

```
var myHealth = 100;

var decrementHealth = function() {
    console.log(myHealth);

    var myHealth = myHealth - 1;
};

decrementHealth();
```

So, you would think that this would

- output `100`
- declare a new, function-scoped variable, `myHealth`, shadowing the globally scoped variable `myHealth`
- set the function-scoped `myHealth` to `99`

And, it would be totally reasonable to think that. Unfortunately, what you actually output is `undefined`. JavaScript will automatically lift the declaration of `myHealth` to the top of the function, but not the assignment.

After the JavaScript engine gets done with that, here is the code you're actually left with:

```
var myHealth = 100;

var decrementHealth = function() {
    var myHealth;
    console.log(myHealth);

    myHealth = myHealth-1;

};
```

Suddenly, that `undefined` output makes sense. Be careful! Declare all your variables up front so that this scenario doesn't catch you unawares, and make sure they have sane default values.

As a further illustration, let's take a look at the following example:

```
var myHealth = 100;

var decrementHealth = function(health) {
    var myHealth = health;
    myHealth--;
    console.log(myHealth);
};

decrementHealth(myHealth);
console.log(myHealth);
```

This will output `99` first, then `100`, because you're setting `myHealth` to the value of health inside the function rather than setting by reference.

JavaScript Typing and Equality

Now that you understand the basics of variables, let's talk about what types of values those variables can take.

JavaScript is a loosely typed language, with a few base types and automatic coercion between types (for more information, see the section "Type Coercion").

Base Types

JavaScript only has a few basic types for you to keep in mind:

1. Numbers
2. Strings
3. Booleans
4. Objects
5. Arrays
6. `null`
7. `undefined`

Numbers

Numbers are fairly self-explanatory. They can be any number, with or without a decimal point or described using scientific notation, such as `12e-4`.

Most languages treat at least integers and floating-point numbers differently. JavaScript, however, treats all numbers as floating point.

It would take too long to go into the potential problems with floating-point numbers here. Suffice it to say that if you're not careful, you can easily run into floating-point errors. If you want to learn more about the pitfalls of floating-point arithmetic, I'd recommend checking out the Institute of Electrical and Electronics Engineers (IEEE) spec *IEEE 754: Standard for Binary Floating-Point Arithmetic*.

The two additional values numbers can take on are `Infinity` and `NaN`. That's right, `NaN` is a number. (for more information, see the section "Equality Checking").

Strings

Strings are quite a bit simpler. As in most languages, you can treat a string like an array of characters. However, strings are also objects, with numerous built-in properties and methods, such as `length` and `slice`:

```
> "example string"[0]
"e"
> "example string".length
14
> "example string".slice(7)
" string"
```

I should point out that what I'm doing in the previous example is particularly bad. The memory behavior of where hard-coded strings are allocated isn't part of the language specification. Being allocated on the global heap is actually one of the better scenarios. Depending on the browser, each individual use could be allocated separately on the heap, further bloating your memory.

Booleans

Booleans, as in most languages, can take on the values `true` and `false`. Both are reserved keywords in JavaScript. The main difference here between JavaScript and many other languages lies in which values can be coerced to either `true` or `false` (for further details, see the section "Truthiness," later in this chapter).

Objects

Objects are the bread and butter of JavaScript, but they behave a bit differently from those in several other languages. In many ways, objects are similar to dictionaries in modern interpreted languages:

```
x = {};
```

Curly braces indicate that you're defining an object. Nothing inside suggests that this is the empty object. You can assign key-value pairs to an object like so:

```
player = { health: 10 };
```

Pretty simple, really. You can assign multiple key-value pairs to an object by separating them with a comma:

```
player = {
    health: 10,
    position: {
        x: 325,
        y: 210
    }
};
```

Note in this example that you're assigning another object to the `position` property of `player`. This is entirely legal in JavaScript and incredibly simple to do.

To access the object, you can use either dot or bracket notation:

```
> player.health
10
> player['position'].x
325
> player['position']['y']
210
```

Note that when using bracket notation, you need to enclose the key in quotes. If you don't do this, then the key will instead be treated like a variable:

```
> x = "unknownProperty";
"unknownProperty"
> player['position'][x]
undefined
```

This code returns `undefined`, because the interpreter can't find `player['position']['unknownProperty']`. As a side note, you should minimize use of bracket notation whenever possible. Dot notation uses fewer bytes to represent the same thing and can be more effectively minified over the wire (for more information, see the section "Inheritance the JavaScript Way").

Arrays

Arrays act similarly to other languages you may be familiar with:

```
> x = [1, 10, 14, "15"]
[1, 10, 14, "15"]
> x[0]
1
> x[3]
"15"
> x.length
4
> x.push(20, "I'm last!")
Undefined
> x
[1, 10, 14, "15", 20, "I'm last!"]
```

As you can see, arrays are heterogenous, meaning that you can have arbitrary types in a single array. Note that this is a bad idea; the internal representation for heterogenous arrays causes some serious performance headaches. Always try to keep a single type in a given array.

Arrays have a number of convenience functions as well, such as `push`, `pop`, and `slice`. I'm not going to go into much detail on these here. To learn more about them, check out the coverage of the `Array` object by the Mozilla Developer Network (MDN) (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array).

I do, however, want to sound a note of caution regarding the memory performance of these convenience functions. Most, if not all, act by allocating an entirely new array from the heap rather than modifying things in place.

In general, garbage collection and memory management are going to be huge performance concerns in JavaScript, so you want to avoid allocating new arrays and causing object churn as much as possible.

Yet, there isn't a good way to modify arrays in JavaScript without creating newly allocated objects on the heap. You can do some things to mitigate this, and, to that end, a great resource is *Static Memory JavaScript with Object Pools* (www.html5rocks.com/en/tutorials/speed/static-mem-pools/). Unfortunately, it won't completely solve your problems, but keeping these performance considerations in mind will go a long way toward mitigating your biggest memory performance issues.

null

The `null` type is a special value similar to `None` in Python. null signifies when a value has been emptied or specifically set to nothing. Note that this is distinct from the value that unknown variables are equal to or that declared but unassigned variables are set to. For that, we have `undefined`.

undefined

Variables are initially set to `undefined` when declared. Remember from declaration hoisting that declarations are automatically hoisted to the top of the function but that any accompanying assignments are not. This means that any variables will be set to `undefined` between where they're declared at the top of a function and where they're assigned to.

Let's take a look at the difference between `undefined` and `null`:

```
var player = {
    health: 100,
    damage: 5,
    hit: function() {
        console.log('poke');
    }
};
console.log(enemy);

var enemy = {
    health: 100,
    damage: 50,
    hit: function() {
        console.log('SMASH');
    }
};
console.log(enemy.health);
console.log(player.shields);
```

This code will output as follows:

```
> undefined

> 100
> undefined
```

The typeof Operator

JavaScript has a handy operator, `typeof`, which can tell you, as you'd guess, the type of its operator. Let's examine a few of these:

```
> typeof 2
"number"
> typeof 2.14
"number"
> typeof Infinity
"number"
> typeof NaN
"number"
```

So far so good; as you might expect, all of these return the string "number".

```
> typeof ""
"string"
> typeof "coconuts"
"string"
> typeof '2.4'
"string"
> typeof true
"boolean"
> typeof false
"boolean"
```

Strings and booleans also behave as expected. The challenge comes when looking at objects, `undefined`, and `null`.

```
> typeof {}
"object"
> typeof { key: "value" }
"object"
> typeof undefined
"undefined"
> typeof null
"object"
```

The issue is that `null` is treated as an object rather than its own type. This can be a problem when using the `typeof` operator, so make sure only to use `null` in situations in which this isn't a concern.

Note as well that `typeof` makes no distinction between different kinds of objects; it just tells you whether a value is an object. To distinguish between different types of objects, we have the `instanceof` operator.

The instanceof Operator

`instanceof` compares two objects and returns a boolean indicating whether the first inherits from the second. Let's look at a few examples:

```
> String instanceof Object
true
> Object instanceof String
False
```

```
> var a = {};
> a instanceof Object
true
> a instanceof String
false
```

In JavaScript all objects inherit from the base `Object`, so the first result, comparing `String`, makes sense, as does the third, comparing `a`, the empty object. In contrast, neither `Object` nor `a` inherits from `String`, so this should return `false` (for details on how to structure inheritance and object-oriented (OO) code in JavaScript, see the section "Inheritance the JavaScript Way").

Type Coercion

JavaScript is a dynamically typed language, with automatic type conversion, meaning that types are converted as needed, based on the operations being performed. Now, this type conversion is a little . . . misbehaved. Let's take a look at the following example:

```
> x = "37" + 3
"373"
> x = 3 + "7"
"37"
> x = 10 - "3"
7
```

Wait, what?

JavaScript will automatically convert between numbers and strings, but the way it does so depends on the operators involved. Basically, if the + operator is involved, it will convert any numbers to strings and assume that + means "concatenate."

However, any other operators will instead convert strings to numbers and assume that the operators involved are arithmetic.

What about an expression with more operators?

```
> x = "10" + 3 / 4 - 2
98.75
```

Can you tell what steps JavaScript took to get the result `98.75`? Personally, it took me a few seconds to step through and figure it out.

In general, you should avoid automatic coercion between types, and instead be explicit. JavaScript has a couple of handy built-in functions to convert from strings to numbers, `parseInt` and `parseFloat`:

```
> parseInt("654", 10)
654
> parseInt("654.54", 10)
654
> parseInt("654", 8)
428
> parseFloat("654")
654
> parseFloat("654.54")
654.54
```

The first parameter for both functions is the string you want to convert to a number. Note that `parseInt` automatically truncates anything after the decimal point rather than rounding.

`parseInt` also takes an optional second parameter, which is the radix, or base of the number system being converted to. The default is the standard base-10 number system.

It's worth pointing out the reverse process, converting a number to a string. The primary way to do this is to call `String(value)`, where `value` is what you want converted to a string.

Equality Checking

One of the greatest challenges for new JavaScript developers is, without a doubt. equality checking. Thankfully, the key to avoiding getting tripped up can be summed up very easily:

Always use === and !== to do equality checking rather than == and !=.

But, why must you do that? What's the deal with this === nonsense, and why are there two different equality operators?

The answer has to do with our friend automatic type coercion. == and != will automatically convert values to different types before comparing them for equality. The === and !== operators do not and will return `false` for different types.

However, what are the rules for how == converts types?

- Comparing numbers and strings will always convert the strings to numbers.
- `null` and `undefined` will always equal each other.
- Comparing booleans to any other type will always cause the booleans to be converted to numbers.
- Comparing numbers or strings to objects will always cause the numbers or strings to be converted to objects.
- Any other type comparisons are automatically `false`.

Once the types have been converted, the comparison continues the same as with `===`. Numbers and booleans are compared by value, and strings, by identical characters. `null` and `undefined` equal each other and nothing else, and objects must reference the same object.

Now that you know how == works, the earlier advice never to use it can be relaxed, at least a little bit. == can be useful, but you must be absolutely sure you know what you're doing.

Truthiness

Using various types in conditional statements is similarly problematic. Because of type coercion, you can use any type in a conditional, and that type is converted to a boolean.

The rules for converting other types to booleans are actually relatively straightforward:

- `undefined` and `null` are always `false`.
- Booleans are just treated as booleans (obviously).
- Numbers are `false` if they equal `0` or `NaN`; otherwise, they're `true`.
- Strings are `true`, except for the empty string `""`, which is `false`.
- Objects are always `true`.

The biggest thing to watch out for is that, whereas the empty string is `false`, the empty object is `true`. Be aware of this when using objects in comparisons, and you'll have solved 90 percent of your problems with truthiness.

Inheritance the JavaScript Way

If you're coming from traditional game development, you're probably very familiar with object-oriented programming (OOP) and specifically, class-based OOP, the model that C++ and Java use.

JavaScript uses a different OOP model, prototypical inheritance, which is derived from `self`'s object model.

I'll close out this chapter by discussing what prototypical inheritance is and how to use it instead of the more classical inheritance you may be used to.

Prototypical Inheritance

Prototypical inheritance, at its core, is concerned with only two things:

1. How do you create a new object?
2. How do you extend a new object from an existing one?

Creating a bare new object is simple, using object literal notation. Let's say you wanted to create the following ship:

```
var myAwesomeShip = {
    health: 100,
    shields: 50,
    guns: [{
        damage: 20,
        speed: 5
    },{
        damage: 5,
        speed: 9000
    }],
    fire: function() {
        console.log('PEW PEW');
    }
};
```

Simple enough. But, what if you wanted to create a new ship, using `myAwesomeShip` as a template, but with better shields? Obviously, you could just copy and paste things, but that's no good. Instead, you can create a clone of `myAwesomeShip`, using prototypical inheritance, like so:

```
var myMoreAwesomeShip = Object.create(myAwesomeShip);
myMoreAwesomeShip.shields = 100;
```

And, you're done. Now, if you wanted, you could roll this into a ship template object, as follows:

```
var ship = {
    manufacture: function(shields) {
        var newShip = Object.create(this);
        newShip.shields = shields;
        return newShip;
    },
```

```
    health: 100,
    shields: 50,
    guns: [{
        damage: 20,
        speed: 5
    },{
        damage: 5,
        speed: 9000
    }],
    fire: function() {
        console.log(PEW PEW');
    }
};

var myWayMoreAwesomeShip = ship.manufacture(150);
```

Voilà: you have a ship template that you can build off of, using any given ship as the template.

Of course, there is still one very important question that must be answered: Can you somehow combine these steps in order to extend the base ship with arbitrary properties?

It turns out that you can do this by writing an `extend` function and attaching it to all objects. The code for this is short, but dense:

```
Object.prototype.extend = function(extendPrototype) {
    var hasOwnProperty = Object.hasOwnProperty;
    var object = Object.create(this);

    for (var property in extendPrototype) {
        if(hasOwnProperty.call(extendPrototype, property) || typeof object[property] ===
'undefined') {
            object[property] = extendPrototype[property];
        }
    }
    return object;
};
```

Whew! There's a lot going on there. Here's what's happening:

1. You create a function, `extend`, attached to the base `Object`.
2. The function `hasOwnProperty` checks whether the object has the passed-in property or whether it's inherited from somewhere else, for example, the base `Object`.
3. You create a clone of `this`; in the previous example, `this` would be `ship`.
4. Now, you loop through all the properties in the extension; for each property, you perform these tasks:
    a. You check whether the base `Object` does not have the given property or whether the extension has the property directly.
    b. You then you assign the value from the extension to the cloned object.
5. Once you're done, you return the completed object.

Reread that a few times if you need to; it's a lot to digest. Now that you know the steps, however, you can create a new ship template, with any additional property changes you want, as shown:

```
var newShipModel = ship.extend({
    health: 200,
    shields: 100,
    fire: function() {
        console.log('TRIPLE PEW!!!');
    }
});
```

You can think of this as a new model of ship that you're going to have your shipyards build:

```
var oldShip = ship.manufacture(100);
var newShip = newShipModel.manufacture(150);
```

this

You may be a little confused by the use of the `this` keyword in the prior `extend` function. `this` behaves somewhat differently in JavaScript than it does in many other languages, primarily because JavaScript is not a class-based language.

`this` can behave differently, depending on where it's called, and can even change from function call to function call in the same function.

Let's walk through the different values `this` can take on:

- If you call `this` globally, then it refers to the global object, which is the `window` object, if running inside a browser.
- If you call `this` inside a function that is not attached to an object, `this` refers to the global object, because, by default, functions are attached to the global object.
- If you call `this` inside a function that is attached to an object, such as in the `fire` method of `ship`, then it refers to the object the function is attached to (in this case, `ship`).
- If you call `this` inside a constructor function, then a new object is created when you call it with `new`, and `this` refers to that object.
- If you call `this` in a function that is then called from an event handler, then `this` refers either to the document object model (DOM) element in the page that triggered the event or to the global object, if there is no such DOM element.

Note that this last value is where you're most likely to run into trouble, because the behavior concerning event handlers specifically overrides the behavior you would otherwise expect.

Fortunately, you can get around some of this behavior with the `call` and `apply` functions. These are attached to every function object and are used to explicitly set what `this` refers to. For instance, if you called `myAwesomeShip.fire.call(myMoreAwesomeShip)`, then `this` would refer to `myMoreAwesomeShip`.

In general, it's useful to explicitly declare what you expect `this` to be whenever you call a function that uses it.

■ **Note** Often, developers coming from a class-based OOP language rail against the lack of proper OOP in JavaScript. The truth is that JavaScript has a very flexible OOP model that can be used in much the same way as a more classical language if required. If you don't need it, then the flexibility of JavaScript's prototypical inheritance can actually be a huge boon, making it far simpler to build up the complex inheritances necessary for a game.

Conclusion

This has been a whirlwind tour of JavaScript, detailing all the pieces you need to start building a basic game architecture and pointing out some of the pitfalls along the way.

JavaScript gets a bad rap for some of its quirks and idiosyncrasies. I've detailed a few of the more nuanced issues here, and awareness of these should keep you from making some of the more painful mistakes I made when first starting out.

CHAPTER 2

Optimal Asset Loading

Ian Ballantyne, Software Engineer, Turbulenz Limited

Designing an efficient method of loading game asset data for HTML5 games is essential in creating a good user experience for players. When games can be played immediately in the browser with no prior downloading there are different considerations to make, not only for the first time play experience but also for future plays of the game. The **assets** referred to by this chapter are not the usual HTML, CSS, JavaScript, and other media that make up a web site, but are the game assets required specifically for the game experience. The techniques mentioned in this chapter go beyond dealing with the standard image and sound data usually handled by the browser and aim at helping you consider assets such as models, animations, shaders, materials, UIs, and other structural data that is not represented in code. Whether this data is in text or binary form (the "Data Formats" section will discuss both) it somehow needs to be transferred to the player's machine so that the JavaScript code running the game can turn it into something amazing that players can interact with.

This chapter also discusses various considerations game developers should make regarding the distribution of their game assets and optimizations for loading data. Structuring a good loading mechanism and understanding the communication process between the client's browser and server are essential for producing a responsive game that can quickly be enjoyed by millions of users simultaneously. Taking advantage of the techniques mentioned in the "Asset Hosting" section is essential for the best first impressions of the game, making sure it starts quickly the first time. The tips in the "Caching Data" section are primarily focussed on improving performance for future runs, making an online, connected game feel like it is sitting on the player's computer, ready to run at any time. The final section on "Asset Grouping" is about organizing assets in a way that suits the strengths and weaknesses of browser-based data loading.

The concepts covered by each section are a flavor of what you will need to do to improve your loading times. Although the concepts are straightforward to understand, the complexity lies in the details of the implementation with respect to your game, and which services or APIs you choose. Each section outlines the resources that are essential to discover the APIs in more detail. Many of the concepts have been implemented as part of the open source Turbulenz Engine, which is used throughout this chapter as a real world example of the techniques presented. Figure 2-1 shows the Turbulenz Engine in action. The libraries not only prove that the concepts work for published games, but also show how to handle the capabilities and quirks of different browsers in a single implementation. By the end of the chapter you should have a good idea of which quick improvements to make and what new approaches are worth investigating.

Figure 2-1. *Polycraft is a complete 3D, HTML5 game built by Wonderstruck Games (http://wonderstruckgames.com) using the Turbulenz Engine. With 1000+ assets equating to ~50Mbs of data when uncompressed, efficiently loading and processing assets for this amount of data is essential for a smooth gaming experience. The recommendations in this chapter come from our experiences of developing games such as Polycraft for the Web. The development team at Turbulenz hope that by sharing this information, other game developers will also be able to harness the power of the web platform for their games*

Caching Data

Caching content to manage the trade-off between loading times and resource limitations has always been a game development concern. Whether it be transferring information from optical media, hard disks, or memory, the amount of bandwidth, latency, and storage space available dictates the strategy required. The browser presents another environment with its own characteristics and so an appropriate strategy must be chosen. There is no guarantee that previous strategies will "just work" in this space.

In the world of browser-based gaming, caching can occur in the following locations: server side and client side. On the server side, the type of cache depends on the server configuration and the infrastructure behind it, for example whether a content distribution network (CDN) is being used to host the files. On the client side, it depends on the browser configuration and ultimately the game as it decides what to do with the data it receives. The more of these resources you have control over, the more optimizations you can make. In some occasions, certain features won't be available so it's always worth considering having a fallback solution. Figure 2-2 shows a typical distribution configuration. The server-side and request caches on the remote host servers, either on disk or in memory, ensure that when a request comes in, it is handled as quickly as possible. The browser cache and web storage, typically from the local disk, reduce the need to rely on a remote machine. The asset cache, an example of holding data (processed or unprocessed) in memory until required, represents the game's own ability to manage the limited available memory, avoiding the need to request it from local disk or remote host server.

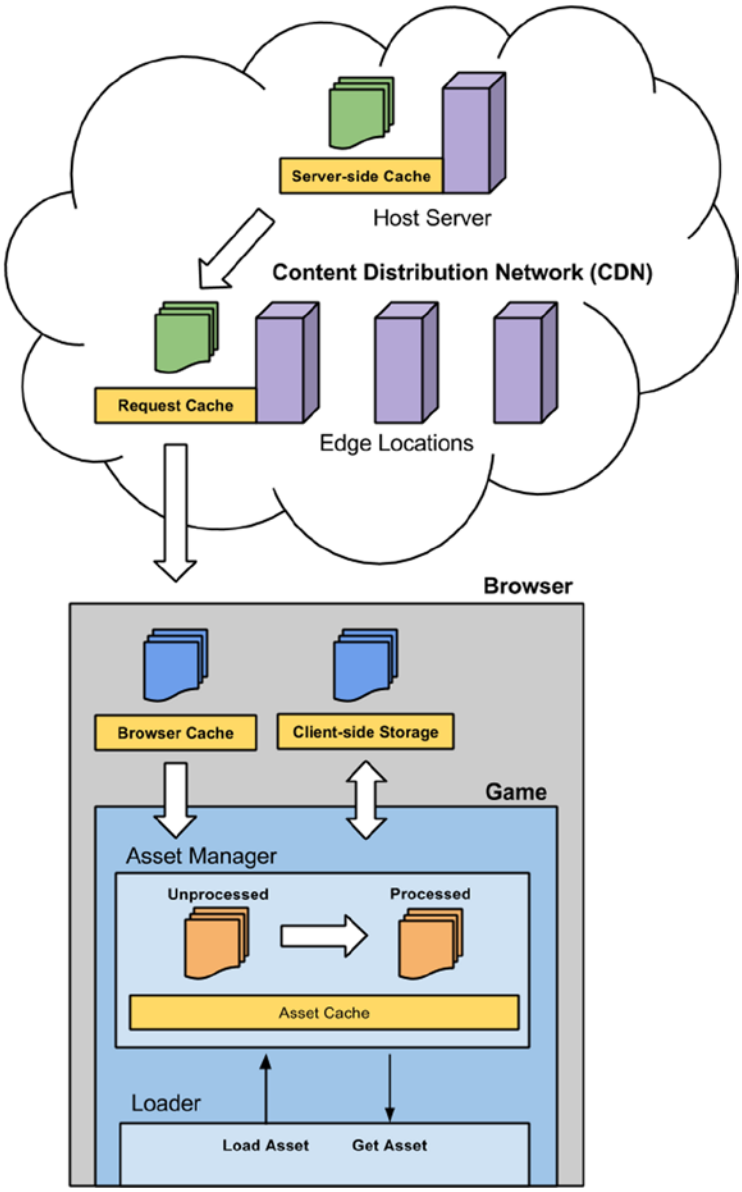

Figure 2-2. *Possible locations on the server and client side where game assets can be cached, from being stored on disk by a remote host server to being stored in memory already processed and ready to use by the game*

HTTP Caching

The most prevalent client-side caching approach is HTTP caching in the browser. When the browser requests a file over HTTP, it takes time to download the file. Once the file has been downloaded, the browser can store it in its local cache. This means for subsequent requests for that file the browser will refer to its local cached copy. This technique eliminates the server request and the need to download the file. It has the added bonus that you will receive fewer server requests, which may save you money in hosting costs. This technique takes advantage of the fact that most game assets are static content, changing infrequently.

When a HTTP server sends a file to a client, it can optionally include metadata in the form of headers, such as the file encoding. To enable more fine-grained control of HTTP caching in the browser requires the server to be configured to provide headers with caching information for the static assets. This tells the browser to use the locally cached file from the disk instead of downloading it again. If the cached file doesn't exist or the local cache has been cleared, then it will download the file. As far as the game is concerned, there is no difference in this process except that the cached file should load quicker. The behavior of headers is categorized as **conditional** and **unconditional**. Conditional means that the browser will use the header information to decide whether to use the cached version or not. It may then issue a conditional request to the server and download if the file has changed. Unconditional means that if the header conditions are met and the file is already in the cache; then it will return the cached copy and it won't make any requests to the server. These headers give you varying levels of control for how browsers download and cache static assets from your game.

The HTTP/1.1 specification allows you to set the following headers for caching.

- Unconditional:
  - **Expires: HTTP-DATE**. The timestamp after which the browser will make a server request for the file even if it is stored in the local cache. This can be used to stop additional requests from being made.
  - **Cache control: max-age=DELTA-SECONDS**. The time in seconds since the initial request that the browser will cache the response file and hence not make another request. This allows the same behavior as the Expires header, but is specified in a different way. The max-age directive overrides the Expires header so only one or the other should be used.
- Conditional:
  - **Last-Modified: HTTP-DATE**. The timestamp indicating when the response file was last modified. This is typically the last-modified time of the file on the filesystem, but can be a different representation of a last modified date, for example the last time a file was referenced on the server, even if not modified on disk. Since this is a conditional header, it depends on how the browser uses it to decide whether or not a request is made. If the file is in cache and the HTTP-DATE was long ago, the browser is unlikely to re-request the file.
  - **ETag: ENTITY-TAG**. An identifier for the response. This can typically be a hash or some other method of file versioning. The ETag can be sent alongside a request. The server can use the ETag to see if the version of the file on the client matches the version on the server. If the file hasn't changed, the server returns a HTTP response with a status code of 304 Not Modified. This tells the client that a new copy of the file is not required. For more information on ETags, see http://en.wikipedia.org/wiki/HTTP_ETag.

The ability to control caching settings is not always available from every server and behaviors will differ depending on the server. It is worth referring to the documentation to discover how to enable and set these headers.

HTTP Caching Example

Since the caching works per URL, you will need to serve your asset files in a way that can take advantage of these headers. One approach is to uniquely name each file and set the expires header/max-age to be as long as possible. This will give you a unique URL for each version of the asset file, allowing you to control the headers individually. The unique name could be a hash based on the file contents, which can be done automatically as part of an offline build process. If this hash is deterministic, the same asset used by different versions of your games can be given the same unique URL. If the source asset changes, a new hash is generated, which can also be used to manage versioning of assets.

This approach exhibits the following behaviors:

- You can host assets for different versions of your game (or different games entirely) in the same location. This can save storage space on the server and make the process of deploying your game more efficient as certain hashed assets may have previously been uploaded.
- When a player loads a new version of your game for the first time, if the files shared between versions are already in the local cache, no downloading is required. This speeds up the loading time for game builds with few asset changes, reducing the impact of updating the game for users. Updates are therefore less expensive and this encourages more frequent improvements.
- Since the file requested is versioned via the unique name, changing the request URL can update the file. This has the benefit that the file is not replaced and hence if the game needs to roll back to using an older version of the file, only the request URL needs to change. No additional requests are made and no files need to be re-downloaded, having rolled back (provided the original file is still in local cache).
- Offline processing tools for generating the asset files can use the unique filename to decide if they need to rebuild a file from source. This can improve the iterative development process and help with asset management.

Loading HTTP Cached Assets

Once a game is able to cache static assets in this way, it will need a process to be able to manage which URLs to request. This is a case of matching the name of an asset with a given version of that asset. In this example, you can assume that the source path for an asset can be mapped directly to the latest required version of that asset. The asset contents can be changed, but the source path remains the same, so no code changes are required to update assets. If the game requires a shader called `shaders/debug.cgfx`, it will need to know the unique hash so it can construct the URL to request. At Turbulenz, this is done by creating a logical mapping between source path and asset filename, and storing the information in a mapping table. A mapping table is effectively a lookup table, loaded by the game and stored as a JavaScript object literal; see Listing 2-1.

Listing 2-1. An Example of a Mapping Table

```
var urlMapping = {
   "shaders/debug.cgfx" : "2Hohp_autOWOWbutP_NSUw.json",
   "shaders/defaultrendering.cgfx" : "4HdTZBhuheSPYHe1vmygYA.json",
   "shaders/standard.cgfx" : "5Yhd75LjDeV3WEvRsKnGSQ.json"
};
```

Each source path represents an asset the game can refer to. Using the source path and not the source filename avoids naming conflicts and allows you to structure your assets like a file system. If two source assets generate identical output, the resulting hash will be the same, avoiding the duplication of asset data. This gives you some flexibility when importing asset names from external tools such as 3D editors.

In this example, the shaders for 3D rendering are referenced by their source path, which maps to a processed JSON formatted object representation of the shader. Since the resulting filename is unique, there is no need to maintain a hierarchical directory structure to store files. This allows the server to apply the caching headers to all files in a given directory, in this case a directory named `staticmax`, which contains all files that should be cached for the longest time period; see Listing 2-2.

Listing 2-2. A Simplified Example of Loading a Static Asset Cached as Described Above

```
/**
* Assumed global values:
*
* console - The console to output error messages for failure to load/parse assets.
*/

/**
* The prefix appended to the mapping table name.
* This is effectively the location of the asset directory.
* This will eventually be the URL of the hosting server/CDN.
*/
var urlPrefix = 'staticmax/';

/**
* The mapping of the shader source path to the processed asset.
* If an asset is not yet loaded this mapping will be undefined.
*/
var shaderMapping = {};

/**
* The function that will make the asynchronous request for the asset.
* The callback will return with the status code and response it receives from the server.
*/
function requestStaticAssetFn(srcName, callback) {

   // If there is no mapping, a URL request cannot be made.
   var assetName = urlMapping[srcName];
   if (!assetName) {
       return false;
   }

   function httpRequestCallback() {
       // When the readyState is 4 (DONE), call the callback
       if (xhr.readyState === 4) {
           var xhrResponseText = xhr.responseText;
           var xhrStatus = xhr.status;
           if (callback) {
               callback(xhrResponseText, xhrStatus);
           }
           xhr.onreadystatechange = null;
           xhr = null;
           callback = null;
       }
   }
```

```
    // Construct the URL to request the asset
    var requestURL = urlPrefix + assetName;

    // Make the request using XHR
    var xhr = new window.XMLHttpRequest();
    xhr.open('GET', requestURL, true);
    if (callback) {
        xhr.onreadystatechange = httpRequestCallback;
    }
    xhr.send(null);
    return true;
}

/**
* Generate the callback function for this particular shader.
* The function will also process the shader in the callback.
*/
function shaderResponseCallback(shaderName) {
    var sourceName = shaderName;
    return function shaderResponseFn(responseText, status) {
        // If the server returns 200, then the asset is included as responseText and can be
processed.
        if (status === 200) {
            try {
                shaderMapping[sourceName] = JSON.parse(responseText);
            }
            catch (e) {
                console.error("Failed to parse shader with error: " + e);
            }
        }
        else {
            console.error("Failed to load shader with status: " + status);
        }
    };
}

/**
* Actual request for the asset.
* The request should be made if there is no entry for shaderName in the shaderMapping.
* The allows the mapping to be set to null to force the shader to be re-requested.
*/
var shaderName = "shaders/debug.cgfx";
if (!shaderMapping[shaderName]) {
    if (!requestStaticAssetFn(shaderName, shaderResponseCallback(shaderName))) {
        console.error("Failed to make a request for: " + shaderName);
    }
}
```

For a live server, the `urlPrefix` will be something like "`http://asset.hostserver.com/game/staticmax/`". This example shows how it may work for a text-based shader, but this technique could be applied to other types of assets. It is also worth noting that this example doesn't handle the many different response codes that are possible during asset loading, such as 404s, 500s, etc. It is assumed that this code will be part of a much more complex asset handling and automatic retry system. Figure 2-3 shows an example of how the process may play out: loading and running the game, loading the latest mapping table, requesting a shader that is sent from the server, and finally, requesting a shader that ends up being resolved by the browser cache. For a more complete example of this, see the Request Handler and Shader Manager classes in the open source Turbulenz Engine (`https://github.com/turbulenz/turbulenz_engine`).

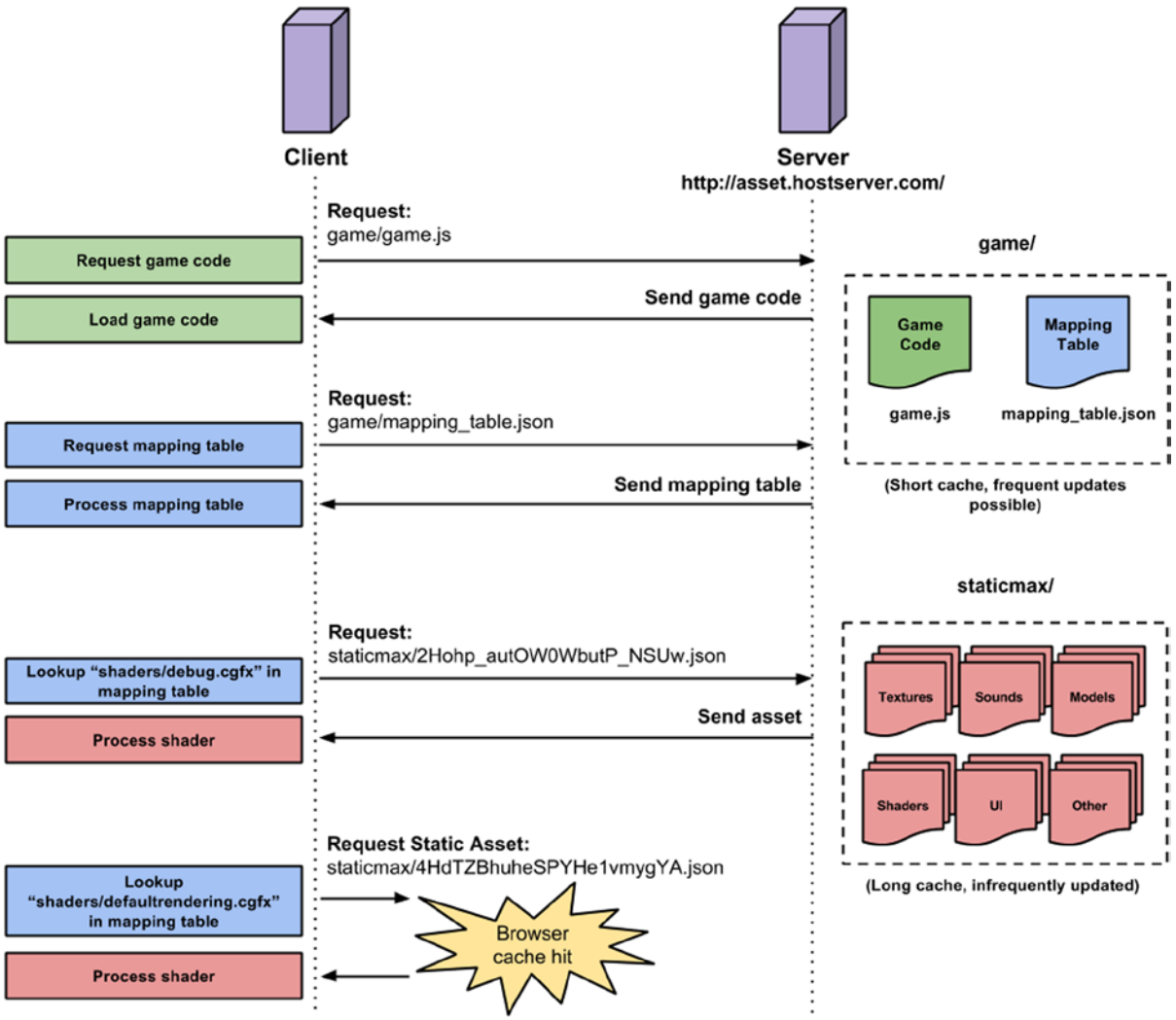

Figure 2-3. *An example of the communication process between a server and the client when using a mapping table. Game code and mapping table data served with short cache times provide the ability to request static assets served with long cache times that take advantage of HTTP caching behaviours*

Client-Side Storage

At this point it is worth mentioning a little bit about client-side storage. The techniques described for HTTP caching are about using a cache to avoid a full HTTP request. There is another option for storing data that sits between server requests and memory that could potentially achieve this if used correctly. Client-side storage usually refers to a number of different APIs that allow web sites to store data on a local machine; these are Web Storage (sometimes referred to as Local Storage), IndexedDB, Web SQL Database, and FileSystem. They ultimately promise one thing: persistent storage between accesses to a web site. Before you get excited and start preparing to save all your game data here, it is important to understand what each API provides, the limitations, the pros and cons, and the availability. Two very good articles that explain all of this very well are at www.html5rocks.com/en/tutorials/offline/whats-offline/ and www.html5rocks.com/en/tutorials/offline/storage/.

What is important is how these APIs are potentially useful for games. Web Storage allows the setting of key/value pairs on a wide range of devices, but has limited storage capacity and requires data to be stored as strings. At the other end of scale, file access via the Filesystem API allows applications to request persistent storage much larger than the limitations of Web Storage with the ability to save and load binary content, but is less widely supported across browsers. If you have asset data that makes sense to be cached by one of these APIs, then it may potentially save you loading time.

Ideally a game would be able use client-side storage to save all static asset data so that it can be accessed quickly without server requests, even while offline, but the ability to do this across all browsers is not consistent and at the time of writing it is difficult to write a generic storage library that would be able to utilize one of the APIs depending on what is available. For example, if you wanted to store binary data for quick access, then it would be possible via "Blobs" for IndexedDB, Web SQL Database, and FileSystem API, but would require data to be base64-encoded as plain text for Web Storage. The amount of storage available and the cost of processing this data would differ depending on which API was used; for example, Web Storage is synchronous and will block during loading, unlike the others, which are asynchronous. If you are happy to acknowledge that only users of browsers that support a given API would have the performance improvements, then client-side storage may still be useful for your game.

One thing client-side storage is potentially useful for across all available APIs is the storing and loading of small bits of non-critical data, such as player preferences or storing temporary data such as save game information if the game gets temporarily disconnected from the cloud. If storing this data locally means that HTTP requests are made less frequently or at not all, then this can certainly speed up loading and saving for your game. In the long term, client-side storage is essential in being able to provide offline solutions to HTML5 games. If your game only partially depends on being able to connect to online services, then players should be able to play it without having a persistent Internet connection. Client-side storage solutions will be able to provide locally cached game assets and temporary storage mechanisms for player data that can be synchronized with the cloud when the player is able to get back online.

Memory Caching

Having loaded an asset from an HTTP request, cache, or client-side storage, the game then has the opportunity to process the asset appropriately. If the asset is in JSON form, then at this point you might typically parse the data and convert it to the object format. Once parsed, the JSON is no longer needed and can be de-referenced for the garbage collector to clean up. This technique helps the application minimize its memory footprint by avoiding duplicating the data representation of an object in memory. If, however, the game frequently requests common assets in this way, the cost of reprocessing assets, even from local cache, can accumulate. By holding a reference to commonly requested assets, the game can avoid making a request entirely at the cost of caching the data in memory. The trade-off between loading speed and memory usage should be measured per asset to find out which common assets would benefit from this approach.

An example of such an asset might be the contents of a menu screen. During typical gameplay, the game may choose to unload the processed data to save memory for game data, but to have a responsive menu, it may choose to process the compressed representation because it is faster than requesting the asset again. Another case where this is convenient is when two different assets request a shared dependency independently of each other, such as a texture used by two different models. If assets are referred to in a key (in this example, the path to an asset, such

as "textures/wall.png"), then the game could use a generic asset cache in memory that has the ability to release assets if they are not being used. Different heuristics can be used to decide if an asset should be released from such a cache, such as size. In the case of texture assets, where texture memory is limited, that cache can be used as a buffer to limit the storage of assets that are used less frequently. Releasing these assets will involve freeing it from the texture memory on the graphics card. Listing 2-3 shows an example of such a memory cache.

Listing 2-3. A Memory Cache with a Limited Asset Size That Prioritizes Cached Assets That Have Been Requested Most Recently

```
/**
 * Assumed global values:
 *
 * Observer - A class to notify subscribers when a callback is made.
 *            See https://github.com/turbulenz/turbulenz_engine for an example.
 * TurbulenzEngine - Required for the setTimeout function.
 *                   Used to make callbacks asynchronously.
 * requestTexture - A function responsible for requesting the texture asset.
 * drawTexture - A function that draws a given texture.
 */

/**
 * AssetCache - A class to manage a cache of a fixed size.
 *
 * When the number of unique assets exceeds the cache size an existing asset is removed.
 * The cache prioritizes cached assets that have been requested most recently.
 */
var AssetCache = (function () {
    function AssetCache() {}

    AssetCache.prototype.exists = function (key) {
        return this.cache.hasOwnProperty(key);
    };

    AssetCache.prototype.isLoading = function (key) {
        // See if the asset has a cache entry and if it is loading
        var cachedAsset = this.cache[key];
        if (cachedAsset) {
            return cachedAsset.isLoading;
        }
        return false;
    };

    AssetCache.prototype.get = function (key) {
        // Look for the asset in the cache
        var cachedAsset = this.cache[key];
        if (cachedAsset) {
            // Set the current hitCounter for the asset
            // This indicates it is the last requested asset
            cachedAsset.cacheHit = this.hitCounter;
            this.hitCounter += 1;
```

```
        // Return the asset. This is null if the asset is still loading
        return cachedAsset.asset;
    }
    return null;
};

AssetCache.prototype.request = function (key, params, callback) {
    // Look for the asset in the cache
    var cachedAsset = this.cache[key];
    if (cachedAsset) {
        // Set the current hitCounter for the asset
        // This indicates it is the last requested asset
        cachedAsset.cacheHit = this.hitCounter;
        this.hitCounter += 1;
        if (!callback) {
            return;
        }
        if (cachedAsset.isLoading) {
            // Subscribe the callback to be called when loading is complete
            cachedAsset.observer.subscribe(callback);
        } else {
            // Call the callback asynchronously, like a request response
            TurbulenzEngine.setTimeout(function requestCallbackFn() {
                callback(key, cachedAsset.asset, params);
            }, 0);
        }
        return;
    }

    var cacheArray = this.cacheArray;
    var cacheArrayLength = cacheArray.length;

    if (cacheArrayLength >= this.maxCacheSize) {
        // If the cache exceeds the maximum cache size, remove an asset
        var cache = this.cache;
        var oldestCacheHit = this.hitCounter;
        var oldestKey = null;
        var oldestIndex;
        var i;

        // Find the oldest cache entry
        for (i = 0; i < cacheArrayLength; i += 1) {
            if (cacheArray[i].cacheHit < oldestCacheHit) {
                oldestCacheHit = cacheArray[i].cacheHit;
                oldestIndex = i;
            }
        }

        // Reuse an existing cachedAsset object to avoid object re-creation
        cachedAsset = cacheArray[oldestIndex];
        oldestKey = cachedAsset.key;
```