Simple asset management for HTML5 games

- The Problem
- The Solution
- Queuing
- Start Downloads
- Tracking Success and Failure
- Signaling When Done
- Easy Retrieval of Assets
- Bonus: Bug Fix
- Example Usage
- Areas for Improvement
- Full Source
- Summary

HTML5 has provided many useful APIs for building modern, responsive, and powerful web applications in the browser. This is great, but you really want to build and play games! Luckily, HTML5 has also ushered in a new era of game development that uses APIs like Canvas and powerful JavaScript engines to deliver gaming straight to your browser without the need for plugins.

This article will walk you through building a simple Asset Management component for your HTML5 game. Without an asset manager, your game will have a hard time compensating for unknown download times and asynchronous image loading. Follow along to see an example of a simple asset manager for your HTML5 games.

The Problem

HTML5 games can’t assume their assets such as images or audio will be on the player’s local machine, as HTML5 games imply being played in a web browser with assets downloaded over

HTTP. Because the network is involved, the browser isn't sure when the assets for the game will be downloaded and available.

The basic way to programmatically load an image in a web browser is the following code:

```
var image = new Image();
image.addEventListener("success", function(e) {
  // do stuff with the image
});
image.src = "/some/image.png";
```

Now imagine having a hundred images that need to be loaded and displayed when the game starts up. How do you know when all 100 images are ready? Did they all successfully load? When should the game actually start?

The Solution

Let an asset manager handle the queuing of assets and report back to the game when everything is ready. An asset manager generalizes the logic for loading assets over the network, and it provides an easy way to check on the status.

Our simple asset manager has the following requirements:

- queue up downloads
- start downloads
- track success and failure
- signal when everything is done
- easy retrieval of assets

Queuing

The first requirement is to queue up downloads. This design lets you declare the assets you need without actually downloading them. This can be useful if, for example, you want to declare all assets for a game level in a configuration file.

The code for the constructor and queuing looks like:

```
function AssetManager() {
  this.downloadQueue = [];
```

```
}

AssetManager.prototype.queueDownload = function(path) {
    this.downloadQueue.push(path);
}
```

Start Downloads

After you have queued up all the assets to be downloaded, you can ask the asset manager to start downloading everything.

The web browser can parallelize the downloads, luckily—usually up to 4 connections per host. One way to speed up asset downloading is to use a range of domain names for asset hosting. For example, instead of serving everything from assets.example.com, try using assets1.example.com, assets2.example.com, assets3.example.com, and so on. Even if each of those domain names is simply a CNAME to the same web server, the web browser sees them as separate servers and increases the number of connections used for asset downloading. Learn more about this technique from Split Components Across Domains at Best Practices for Speeding Up Your Web Site.

Our method for download initialization is called `downloadAll()`. We'll build it up over time. For now, here is the first logic to just start the downloads.

```
AssetManager.prototype.downloadAll = function() {
    for (var i = 0; i &lt; this.downloadQueue.length; i++) {
        var path = this.downloadQueue[i];
        var img = new Image();
        var that = this;
        img.addEventListener("load", function() {
            // coming soon
        }, false);
        img.src = path;
    }
}
```

As you can see in the code above, `downloadAll()` simply iterates through the downloadQueue and creates a new Image object. An event listener for the load event is added and the src of the image is set, which triggers the actual download.

With this method you can start the downloads.

Tracking Success and Failure

Another requirement is to track both success and failures, because unfortunately not everything always works out perfectly. The code so far only tracks successfully downloaded assets. By adding an event listener for the error event, you will be able to capture both success and failure scenarios.

```
AssetManager.prototype.downloadAll = function(downloadCallback) {
  for (var i = 0; i &lt; this.downloadQueue.length; i++) {
    var path = this.downloadQueue[i];
    var img = new Image();
    var that = this;
    img.addEventListener("load", function() {
        // coming soon
    }, false);
    img.addEventListener("error", function() {
        // coming soon
    }, false);
    img.src = path;
  }
}
```

Our asset manager needs to know how many successes and failures we've encountered, or it will never know when the game can start.

First up, we'll add the counters to the object in the constructor, which now looks like this:

```
function AssetManager() {
<span class="highlight">    this.successCount = 0;
    this.errorCount = 0;</span>
    this.downloadQueue = [];
}
```

Next, increment the counters in the event listeners, which now look like this:

```
img.addEventListener("load", function() {
    <span class="highlight">that.successCount += 1;</span>
}, false);
img.addEventListener("error", function() {
    <span class="highlight">that.errorCount += 1;</span>
}, false);
```

The asset manager is now tracking both successfully loaded and failed assets.

Signaling When Done

After the game has queued up its assets for download, and asked the asset manager to download all the assets, the game needs to be told when all the assets are downloaded. Instead of the game asking over and over and over if the assets are downloaded, the asset manager can signal back to the game.

The asset manager needs to first know when every asset is finished. We will add an isDone method now:

```
AssetManager.prototype.isDone = function() {
    return (this.downloadQueue.length == this.successCount + this.errorCount);
}
```

By comparing the successCount + errorCount to the size of the downloadQueue, the asset manager knows if every asset either finished successfully or had some sort of error.

Of course knowing if it’s done is only half the battle; the asset manager also needs to check this method. We will add this check inside both of our event handlers, as the code below shows:

```
img.addEventListener("load", function() {
    console.log(this.src + ' is loaded');
    that.successCount += 1;
    if (that.isDone()) {
        // ???
    }
}, false);
img.addEventListener("error", function() {
    that.errorCount += 1;
if (that.isDone()) {
        // ???
    }
}, false);
```

After the counters are incremented, we will see if that was the last asset in our queue. If the asset manager is indeed done downloading, what should we do, exactly?

If the asset manager is done downloading all the assets, we will call a callback method, of course! Let’s change `downloadAll()` and add a parameter for the callback:

```
AssetManager.prototype.downloadAll = function(downloadCallback) {
    ...
```

We will call the downloadCallback method inside of our event listeners:

```
img.addEventListener("load", function() {
    that.successCount += 1;
    if (that.isDone()) {
        downloadCallback();
    }
}, false);
img.addEventListener("error", function() {
    that.errorCount += 1;
    if (that.isDone()) {
        downloadCallback();
    }
}, false);
```

The asset manager is finally ready for the last requirement.

Easy Retrieval of Assets

Once the game has been signaled that it can start, the game will begin to render images. The asset manager is not only responsible for downloading and tracking the assets, but also for providing them to the game.

Our final requirement implies some sort of getAsset method, so we'll add it now:

```
AssetManager.prototype.getAsset = function(path) {
    return this.cache[path];
}
```

This cache object is initialized in the constructor, which now looks like this:

```
function AssetManager() {
    this.successCount = 0;
    this.errorCount = 0;
    this.cache = {};
    this.downloadQueue = [];
}
```

The cache is populated at the end of `downloadAll()`, as shown below:

```
AssetManager.prototype.downloadAll = function(downloadCallback) {
  ...
      img.addEventListener("error", function() {
          that.errorCount += 1;
          if (that.isDone()) {
              downloadCallback();
          }
      }, false);
      img.src = path;
      <span class="highlight">this.cache[path] = img;</span>
  }
}
```

Bonus: Bug Fix

Did you spot the bug? As written above, the isDone method is only called when either load or error events are triggered. But what if the asset manager doesn't have any assets queued up for download? The isDone method is never triggered, and the game never starts.

You can accommodate this scenario by adding the following code to `downloadAll()`:

```
AssetManager.prototype.downloadAll = function(downloadCallback) {
    if (this.downloadQueue.length === 0) {
      downloadCallback();
  }
 ...
```

If no assets are queued, the callback is called immediately. Bug fixed!

Example Usage

Using this asset manager in your HTML5 game is quite straightforward. Here is the most basic way to use the library:

```
var ASSET_MANAGER = new AssetManager();
```

```
ASSET_MANAGER.queueDownload('img/earth.png');

ASSET_MANAGER.downloadAll(function() {
    var sprite = ASSET_MANAGER.getAsset('img/earth.png');
    ctx.drawImage(sprite, x - sprite.width/2, y - sprite.height/2);
});
```

The above code illustrates:

1. Creates a new asset manager
2. Queue up assets to be downloaded
3. Start the downloads with `downloadAll()`
4. Signal when the assets are ready by invoking the callback function
5. Retrieve assets with `getAsset()`

Areas for Improvement

You will no doubt outgrow this simple asset manager as you build out your game, although I hope it provided a basic start. Future features could include:

- signaling which asset had an error
- callbacks to indicate progress
- retrieving assets from the File System API

Please post improvements, forks, and links to code in the comments below.

Full Source

The source for this asset manager, and the game it's abstracted from, is open source under the Apache License and can be found in the Bad Aliens GitHub account. The Bad Aliens game can be played in your HTML5 compatible browser. This game was the subject for my Google IO talk titled Super Browser 2 Turbo HD Remix: Introduction to HTML5 Game Development (slides, video).

Summary

Most games have some sort of asset manager, but HTML5 games require an asset manager that loads assets over a network and handles failures. This article outlined a simple asset manager that should be easy for you to use and adapt for your next HTML5 game. Have fun, and please let us know what you think in the comments below. Thanks!

□□□ #2
□ □□□ □□□ 2026-10-31 21:37:23 UTC
□ □□□ □□□ 2026-10-31 21:37:25 UTC