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®

HTML5 Game Development Insights

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

Apress®

HTML 5 Game Development Insights

ISBN-13 (pbk): 978-1-4302-6697-6

ISBN-13 (electronic): 978-1-4302-6698-3

President and Publisher: Paul Manning
Lead Editor: Ben Renow-Clarke
Technical Reviewer: Phil Sherry, Rob Evans, David Gash and Rita Turkowski
Developmental Editor: Gary Schwartz
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Christine Ricketts
Copy Editor: Mary Behr and Lisa Vecchione
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

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

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

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

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

Full Book Contents at a Glance

Full Book Contents

About the Authors

Colt McAnlis is a Developer Advocate at Google, focusing on games and performance. Before that, he was a systems and graphics programmer in the games industry working at Blizzard, Microsoft (Ensemble), and Petroglyph. He's a UDACITY course professor for HTML5 games, and he also spent four years as an Adjunct Professor at SMU Guildhall's school for game development. When he's not working with developers, Colt spends his time preparing for an invasion of giant ants from outer space.

Peter Lubbers is a Program Manager in the Google Developer Relations team, focused on scalable developer programs. He is the author of *Pro HTML5 Programming* (2nd Ed. Apress, 2011) and the founder of the San Francisco HTML5 User Group (SFHTML5), the largest HTML5 User Group in the world with over 10,000 members. Prior to joining Google, Peter headed up the HTML5 training division at Kaazing. A native of the Netherlands, Peter served as a Special Forces commando in the Royal Dutch Green Berets. Peter lives on the edge of the Tahoe National Forest and in his spare time he loves to run around Lake Tahoe. You can follow him on Twitter at `@peterlubbers`.

Brandon Jones works on Chrome's WebGL implementation at Google, and he is a big advocate of WebGL in any form. After taking a web development job ten years ago before eagerly moving on to "real programming," he came to his senses and now gleefully spends any time that he's not hacking on Chrome or building 3D things in JavaScript or Dart.

Duncan Tebbs is a Senior Software Engineer at Turbulenz Limited, developing technology to allow developers to bring high quality game content to the Web. He has previously held roles at Electronic Arts, NaturalMotion, and Square-Enix, where he has worked on various games technology and research projects.

Andrzej Manzur is an HTML5 game developer; founder of the Enclave Games indie development studio, which is focused on mobile HTML5 games; creator of the js13kGames competition; and Gamedev.js Meetups organizer. He's a front-end developer and JavaScript programmer, active blogger and conference speaker, HTML5 games evangelist, and a huge fan of the Firefox OS. In his free time, he loves playing Neuroshima Hex, eating sushi, running around with a paintball marker, and driving go-karts, plus everything SciFi/post-apo/cyberpunk related.

Sean Bennett is a Software Engineer at Crowdtilt, and he is passionate about using the Web as a platform for awesome, immersive experiences. Sean's background is in web development and developer education. When he isn't working on building a better Web, Sean likes running, hiking, and preparing for the inevitable zombie apocalypse.

Florian d'Erfurth is a new game developer who's been doing freelance web development for the last five years. In early 2013, he won the Udacity HTML5 Game Development contest, and ever since he's been obsessed with JavaScript and its performance so that he can deliver ubiquitous gaming experiences. Florian is currently working on Foxes and Shotguns, a video game featuring wererabbits.

Bruno Garcia is co-founder at 2DKit, helping developers build cross-platform games. Before that, he engineered browser and mobile games at Three Rings and Zynga. He is a member of the Haxe compiler team, contributes to the Flump open source project, and writes the occasional game on the side. His favorite color is #202020.

Shun Lin is the founder of the Cocos2d-HTML5 open source game engine and a Senior Technical Director at ChuKong, where he focuses on creating a cross-browser and cross-platform game engine. He spends most of his time on improving the performance and compatibility of mobile web games. Shun is also the co-founder of the Cocos2d-x game engine, working alongside the "x-men" to bring JavaScript Binding to Cocos2d-x, which provides developers an easy way to publish their HTML5 games as native apps with very little or no change to their existing code.

Ivan Popelyshev is an MMO game developer and founder of Matroid Games, a company that focuses on real-time and massively multiplayer games. He's passionate about HTML5 cutting edge technologies, building game servers and sports programming. During his career, he won the ACM as well as other contests, and he participated in many well-known competition finals, including TopCoder 2013. Previously, he worked at Wargaming and Yandex, where he supported the Russian sports programming community.

Jason Gauci is a Research Scientist at Apple. Prior to joining Apple, Jason worked in the labs of Google Research and Lockheed Martin Applied Research, focusing on behavior prediction and predictive analytics. Jason is the lead developer for several netplay-enabled emulators, including MAMEHub, NES Together, and SNES Together. He also co-hosts the Programming Throwdown podcast.

Jon Howard is the Executive Product Manager for Future Development at the BBC (CBBC & CBeebies). In this role, he works closely with interactive/TV producers, user experience designers, the BBC Research and Development department, and many of the world's best digital agencies to promote innovation and set strategy for the games portfolio on the UK's most successful children's web sites. Over the years, Jon has designed, developed, and served as tech lead on many hugely successful games for major UK and international childrens' brands, including Tree Fu Tom, Scooby Doo, Rastamouse, Shaun the Sheep, Charlie & Lola, Dick & Dom, Horrible Histories, Sarah Jane Adventures, Scorpion Island, and many more. Jon takes a keen interest in game design developments. He is a creative coder, a prolific game prototyper, and an enthusiastic mathematician/statistician with a great passion for BBC Children, rich media, innovation, and technology.

Ian Ballantyne is a Software Engineer at Turbulenz, developing HTML5 games, tools, and technology. He is responsible for managing the Turbulenz Engine and SDK. He is all too aware of the complex details of delivering high-fidelity 3D games to the web platform. Having worked with indie developers, small studios, and large publishers, he knows a thing or two about helping game developers make the best use of new technologies. When he's not giving talks on high performance HTML5 games, he's involved in their creation, making them run on anything from desktops to mobile devices. By day he is a mild-mannered software engineer; by night he is a rogue coder, hacking away at secret projects and game jams. Prior to Turbulenz, he worked at Philips Electronics on a gaming/lighting technology known as amBX. A graduate of Imperial College London, Ian is still glad to be putting his Masters of Engineering in Computing to practical use.

Jesse Freeman is a Developer Evangelist at Amazon focusing on HTML5 and gaming. He is an active leader in New York's developer community. For more than 13 years, Jesse has been on the cutting edge of interactive development, focused on the web and mobile platforms. As an expert in his field, Jesse has worked for Microsoft, MLB, HBO, the New York Jets, VW, Tommy Hilfiger, Heavy, and many more. In addition to development, Jesse has a background in Art with a Masters in Interactive Computer Art from the School of Visual Arts. He can be found on Twitter at `@jessefreeman`. Jesse also speaks at conferences and does workshops, which you can learn about on his web site, `http://jessefreeman.com`.

Takuo Kihira is the HTML5 Manager and Chief Engineer at DeNA, and he specializes in JavaScript and HTML5 development for mobile platforms. Takuo created the HTML5 Flash Player Pex, which is distributed by DeNA free of charge. Takuo is a serial entrepreneur who has started two companies.

Tyler Smith is the HTML5 Game Evangelist at Intel, focusing on helping the HTML5 game developer community grow. He has participated in numerous app and game development hackathons, both as a speaker and facilitator. Tyler was also the project manager and lead developer of Boom Town, an HTML5 game that was deployed on more than seven platforms with a single code base. He is also the founder and CTO of LocalGhost Media, a small HTML5 development studio. All opinions expressed in the chapter he wrote for this book are his and his alone. They do not reflect the opinion of his employers.

Don Olmstead is a Senior Software Engineer at Sony Network Entertainment, working on the underlying web platform for PlayStation hardware. On the native side, he advances the browser, supporting the rendering bits. For the application side, he develops the rendering engine, built using WebGL, which provides the backbone for the Playstation Store, and portions of the UX on the PS4. He is interested in how the Web can be leveraged by game developers as a target platform.

John McCutchan is a Software Engineer at Google working on the Dart Virtual Machine. While an undergraduate, John created inotify, the Linux kernel filesystem event notification system used in every Linux distribution and Android phone. After receiving a M.Sc. in Computer Science from McMaster University, John joined Sony Computer Entertainment of America, where he optimized the Bullet Physics library for the PlayStation 3. In 2013, John created a highly-performant SIMD programming model for dynamically compiled languages, such as Dart and JavaScript.

Chad Austin is a Senior Technical Director at IMVU, where he focuses on client application structure and the web platform. When he's not in meetings or evangelizing technical strategy, he likes to get his hands dirty—demonstrating what is possible and shaving milliseconds off critical execution paths as well. In a previous life, he wrote a 2D RPG game engine still in use today.

Mario Andrés Pagella is the founder of Warsteed Studios, an independent HTML5 game development studio, and he is also a Technical Project Leader at R/GA. He has authored *Making Isometric Social Real-Time Games with HTML5, CSS3, and JavaScript and HTML5 Transition and Animation*, and he advocates the use of HTML5, CSS3, and JavaScript for game development through his web site, Twitter account, and in meetups and events.

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 5

Faster Canvas Picking

Colt "MainRoach" McAnlis, Developer Advocate, Google

If you're writing a 2D game in HTML5, chances are that you'll want the user to have the ability to pick an object on the screen. More specifically, at some point, the user will need to select some item on the screen, or in your game, which may represent part of the world. We call this "picking" as the user is selecting what object they are interacting with. Consider, for instance, your standard social time-management game. The user is presented with a 2.5D play area where bitmaps (or "sprites") are rendered with some perspective distortion on the screen. For the more advanced users, you can quickly saturate the play area with these sprites, often stacking many of them together, only leaving a few pixels visible between overlapping objects. In this environment, determining the picking result of a mouse click is quite difficult. The canvas API doesn't provide any form of pixel-based selection and the large number of objects makes it difficult to brute-force the technique. This section will cover how to address performance and accuracy problems in canvas picking using a few old-school techniques that most of us have forgotten about.

Creating Pickable Objects

For the sake of simplicity, I'll first introduce the most basic form of 2D picking I can, which is simply doing a point-to-rectangle test for each object in the scene. This type of *brute force* technique will yield accurate results, but not the most *precise* results (I will cover how to get "precise" picking later), especially where two sprites are overlapping. But before we tackle that topic, let's first start off with a few definitions of objects we'll use throughout this section, the *SpritePrototype* and *SpriteInstance*.

Defining a Sprite Prototype

As with most 2D canvas games, your world will not be populated with millions of unique bitmaps, but rather millions of objects where large groups of them share similar bitmaps between them. As such, it makes no sense to load a given image into memory for each sprite that uses it; you'd have duplicate versions of the images, for each *instance* of the sprite that exists, sitting around in main memory, which could quickly become a problem.

A much more performant solution is to simply load your images a single time and create references to them as each instance is created. You effectively cache the *prototype* images into a large array, which can be referenced by the individual instances later. The sample *SpriteProto* class that follows is pretty simplistic. It contains a filename field, alongside width and height (important later for picking, or in the future when using atlases; see `http://en.wikipedia.org/wiki/Texture_atlas` for more information). The most interesting part of this class is the *load* function, which given a filename will invoke JavaScript to load the image into memory. Once loaded, a *SpriteProto* object will then retain a handle to the loaded image in the *imgHandle* member (see Listing 5-1).

Listing 5-1. The SpriteProto definition

```
function SpriteProto(){
    this.filename="";
    this.imgHandle=null;
    this.size={w:0,h:0};
    this.load= function(filename,w,h)
    {
        var targetSpriteProto = this;
        this.size.w = w;
        this.size.h = h;
        var img = new Image();
        img.onload = function(){
            targetSpriteProto.imgHandle = img;
        }
        img.src = filename;
    }
}
```

For most game build-chains, you'll generally create some sort of designer or artist-centric view of the world ahead of time. That is, before this level is actually loaded by the user, you have a pretty good idea what assets will be needed to load, simulate, and display this level. As such, you don't define a complex asset dependency hierarchy here, but rather brute-force the loading of your basic proto-sprites so that they can be used by object instances later. During your initialization, you fill the globally accessible *protoSprites* array using the *loadProtos* function (see Listing 5-2).

Listing 5-2. Loading a set of prototype images

```
var protoSprites = new Array();
//--------------
function loadProtos()
{
    //technically, this should be an atlas definition!
    var imgs=[
        {nm:"0.png",w:66,h:42},
        {nm:"1.png",w:66,h:52},
        {nm:"2.png",w:66,h:46},
        {nm:"3.png",w:70,h:65}
        ];
    for(var i =0; i < imgs.length; i++)
    {
        var sp = new SpriteProto();
        sp.load(imgs[i].nm,imgs[i].w,imgs[i].h);
        protoSprites.push(sp);
    }
}
```

Note that for your purposes, you don't just list the path to the image, but also the width and height of the bitmap in pixels. These image-specific bounding conditions are important for gross-level picking that we'll discuss a bit later in the bounding boxes section.

Representing Objects

Now that you have your images loaded and in a form you can reference, you need to generate the actual objects that will populate your canvas and be targets for picking. To this end, you create a new *SpriteInstance* class.

You'd expect the basic two parameters of this class, *position* and *size*, which you will use for picking later. You also add a few other parameters: a unique *ID* value, alongside a *ZIndex*, which is useful for sorting during rendering, and collision resolution. And of course, you need a reference to what *ProtoSprite* object you'll be using to render (see Listing 5-3).

Listing 5-3. Definition of a SpriteInstance

```
function SpriteInstance(){
    this.id=0;              //the unique ID for this object
    this.zIndex=50;         //needed for rendering and picking resolution, range [0,numObjects]
    this.pos={x:0,y:0};
    this.size={w:0,h:0};
    this.spriteHandle=null; //what sprite we'll use to render
```

For your simple uses, you need to populate the world with pickable objects, and to do so, you simply flood the given canvas with randomly placed objects. Note that you store the generated sprites in a global array for other systems to access later on (see Listing 5-4).

Listing 5-4. Populating the scene with randomly located SpriteInstances

```
var spriteInstances = new Array(); //global list of active sprites
//--------------
function generateInstances()
{
    var numSprites = 4096; //magic number
    for(var i = 0; i < numSprites; i++)
    {
        var sp = new SpriteInstance();
        sp.id = i;
        sp.zIndex = i; //just to keep my sanity...
        sp.pos.x = Math.floor(Math.random() * (500)); //random point on canvas
        sp.pos.y = Math.floor(Math.random() * (500));
                        //choose a random sprite to render this with
        var idx = Math.floor(Math.random() * (protoSprites.length));
        sp.spriteHandle = protoSprites[idx];
        sp.size = sp.spriteHandle.size;

        spriteInstances.push(sp);
    }
}
```

When you'd like to draw the sprites, you simply iterate over each instance, fetch its corresponding *spriteHandle*, and use that to draw to the canvas with the correct position data (see Listing 5-5).

Listing 5-5. Drawing the sprites in the scene

```
function drawSprites()
{
    //note that zIndex = i here, so this rendering is correct according to depth
    for(var i =0; i < spriteInstances.length;i++)
    {
        var sp = spriteInstances[i];
        var ps = sp.spriteHandle;
        ctx.drawImage(ps.imgHandle,sp.pos.x, sp.pos.y);
    }
}
```

See Figure 5-1 for an example of this code in action. Note that this is NOT the way you should be doing rendering in your production code. Real games have various types of restrictions and orderings for depth from their sprites. It's a complex dance between artist-generated content and runtime performance achieved by sorting of sprites. This simple example avoids all this, since a discussion of that size is beyond the scope of this chapter.

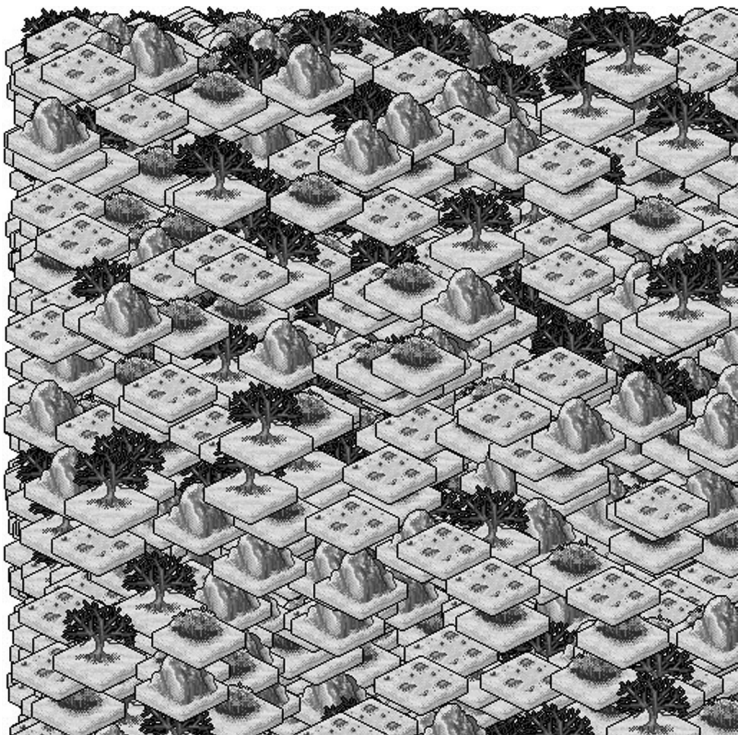


Figure 5-1. *The results of rendering sprites onto the canvas*

Basic Picking with Bounding Boxes

Since your *SpriteInstance* object supports both position and size information, the most straightforward way to determine what sprite has been picked is to test the given point against the bounding box of the sprite. In this simplistic model of picking, all that's needed is a simple loop over all the sprites in the scene, and to test which of their bounding-boxes intersect with the given x,y point on the canvas (given from the mouse); see Listing 5-6.

Listing 5-6. Find the desired sprite by looping through all instances, and determining which one contains the XY pair. Note that this function returns the FIRST instance it finds

```
function findClickedSprite(x,y)
{

 //loop through all sprites
 for(var i =0; i < spriteInstances.length; i++)
 {
  var sp = spriteInstances[i];
  //pick is not intersecting with this sprite
  if( x < sp.pos.x || x > sp.pos.x + sp.size.w ||   y < sp.pos.y || y > sp.pos.y + sp.size.h)
   return sp;
}
return null;
}
```

In Listing 5-6, you're completely ignoring z-ordering and complex depth stacking, and just returning the first sprite that contains this selection point. Don't worry; we'll get to that later.

Caveats

As mentioned, the brute-force, bounding-box method of picking is quite effective for scenes with low object count. However, once the number of objects in your scene increases, this technique will quickly reveal performance problems, especially on underpowered devices. Another particular issue is scene depth complexity—that is, how many overlapping sprites you have. Once you start adopting more aggressive versions of depth complexity, you'll quickly find that this method is unacceptable to user input and expectations, and you'll need a more fine-grain solution.

Faster Picking via Bucketing

Let's say you're building a complex app and you have thousands of images in flight when a user clicks. You'll quickly see that the performance burden lies in the inner loop of the sprite-to-point collision test, and although a simple bounding-box test will be the quickest to implement, you still end up touching a lot of items that aren't even remotely *near* the point in question.

The solution to this is to introduce a *spatial acceleration structure* to your canvas. This data structure organizes/separates your sprites spatially, such that you speed up spatial tests by only referencing items that are reasonably co-located. There's lots of variants for SAS (for example, scene graphs, bounding volume hierarchies, quadtrees, and octrees), each having a unique set of pros, cons, and trade-offs, the analysis of which is outside the scope of this section.

For your purpose, you will use a very simplistic 2D binning algorithm, which will divide your canvas into a grid of cells (of fixed sizes); each cell will contain a list of objects that touch that cell, allowing a sprite to be listed in multiple cells (see Figure 5-2).

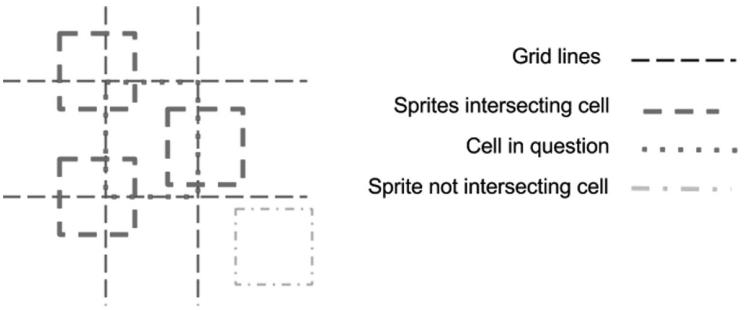


Figure 5-2. *Given a 2D grid, you can determine what sprites overlap with what cells. Later, when a pick event occurs, you can determine what cell the point resides in, and reduce the number of per-sprite checks you need to perform*

Your *BucketGrid* class will effectively create a 2D array of *arrays*, such that when a sprite is spawned, you calculate what grid cells it overlaps, and add a pointer to this instance to each of those cell's lists; see Listing 5-7.

Listing 5-7. Definition of a BucketGrid

```
function BucketGrid(){
    this.tileSize=32;
    this.numXTiles=0;
    this.numYTiles=0;
    this.tileData=null;

    this.init=function()
    {
        this.tileData = new Array();
        this.tileSize = 16;
        this.numXTiles= 512/this.tileSize;
        this.numYTiles= 512/this.tileSize;

        this.tileData.length = this.numXTiles*this.numYTiles;
        for(var k =0; k < this.tileData.length;k++)
            this.tileData[k] = new Array();
    }
```

Marking a sprite instance on the grid is pretty simple. You just run some math on the corners of the sprite to calculate the min/max boundaries of it, and what tiles those boundaries fall into; and for each grid cell, you add the sprite to the containing list (see Listing 5-8).

Listing 5-8. Given a sprite, mark which grid cells it overlaps in

```
this.markInstanceInAccelGrid=function(sp)
{
    var gridminX = Math.floor(sp.pos.x / this.tileSize);
    var gridmaxX = Math.floor((sp.pos.x + sp.size.w) / this.tileSize);
    var gridminY = Math.floor(sp.pos.y / this.tileSize);
    var gridmaxY = Math.floor((sp.pos.y + sp.size.h) / this.tileSize);
```

```
    if(gridmaxX >= this.numXTiles) gridmaxX = this.numXTiles-1;
    if(gridmaxY >= this.numYTiles) gridmaxY = this.numYTiles-1;

    for(var y = gridminY; y <=gridmaxY; y++)
    {
        var idx = y*this.numXTiles;
        for(var x = gridminX; x <=gridmaxX; x++)
        {
            this.tileData[idx+x].push(sp);
        }
    }
};
```

This makes finding entities extremely quick, because you've already pre-sorted the environment. When a mouse-click comes in, you simply find the bucket it resides in, and return the list of entities for that bucket, which you've already calculated.

```
this.getEntsForPoint=function(x,y)
    {
        var gridminX = Math.floor(x / this.tileSize);
        var gridminY = Math.floor(y / this.tileSize);
        var idx = gridminX + gridminY*this.numXTiles;
        var ents = this.tileData[idx];
        return ents;
    };
}
```

This makes modification to your existing code very nice and tidy:

```
function findClickedSprite(x,y)
{

 //loop through all target sprites
var tgtents = acclGrid.getEntsForPoint(x,y);
 for(var i =0; i < tgtents .length; i++)
 {
  var sp = tgtents [i];
  //pick is not intersecting with this sprite
  if( x < sp.pos.x || x > sp.pos.x + sp.size.w ||   y < sp.pos.y || y > sp.pos.y + sp.size.h)
   return sp;
}
return null;
}
```

The acceleration grid reduces performance overhead by reducing the number of times the *inner loop* is called; regardless of how many sprites you have universally, this bucketing will only care about what sprites reside in a grid cell. An easy trade-off in bang-for-the-buck; you didn't have to derail yourself by worrying how JavaScript is handling the math for convex hulls or the array-access times for pixel-perfect picking (both covered later in this chapter). It's a simple technique that can be used for static and dynamic environments.

Note that to use the grid, an entity needs to be responsible for updating its markings on the grid. For your simple purpose, you only need to update the entity spawning code to mark the location of the object in the grid (see Listing 5-9).

Listing 5-9. Inserting the BucketGrid into the existing code

```
//--------------
var protoSprites = new Array();
var spriteInstances = new Array();
var acclGrid = new BucketGrid();
//--------------
function generateInstances()
{
    acclGrid.init();

    var numSprites = 4096; //magic number
    for(var i = 0; i < numSprites; i++)
    {
        //.......

        spriteInstances.push(sp);
        acclGrid.markInstanceInAccelGrid(sp);
    }
}
```

Caveats

This type of spatial acceleration structure is always in demand for games. It's primary goal is to reduce the number of *potential* intersections to a defined list that is always smaller than the entire set. And to be clear, your `BucketGrid` structure did nothing to increase the *precision* of your picking, but only increased the *performance* of your picking. We will discuss precision issues more in the following sections.

One large issue to keep an eye on is that as your game evolves, your spatial structure will need to adapt to take into account objects that move, objects that have been scaled/rotated, etc. Without proper gardening, it's easy to generate bugs with object spawn/update/die lifecycles. In these cases, it may be wise to generate versions of spatial grids, each one specialized to the objects that need it. For example, a static 2D binning grid is fine for static objects, but dynamic ones may need a more aggressive quadtree hierarchy, or perhaps a k-dimensional tree.

Take care of your spatial grid, and it'll remain a strong data structure for your games.

Pixel Perfect Picking

For games that utilize picking for desktop applications, pixel-perfect response to a mouse click is crucial. Multiple images can be overlaid against each other, and each one can have varied alpha footprints which in no way match their conservative bounding box estimates. As such, to do a pixel-perfect pick of an object on the screen, you'll need to be able to determine what pixel from what image was clicked on and what object instance that image belongs to.

In order to do this in HTML5, you need to introduce two separate data structures, a *sprite prototype*, which represents a single loaded image sprite, and a *sprite instance*, which represents an instance of the prototype on the canvas—that is, you assume that a single image is used multiple times on a canvas.

1. The process for pixel-perfect picking is really straightforward. You need to load an image, somehow get access to its pixel data, and keep that around in memory for access later.
2. As objects are created and drawn to screen, you need a reference back to what image it's using.
3. When the user clicks, you need to find what objects intersect with that click, and then per-object, check if the click has hit a transparent or opaque section of the image representing that object.

Step 3 is where the real magic happens: being able to check for intersection against transparent or opaque pixels allows your game to overlay images on top of each other, and you can still determine, at pixel granularity, which image/object was selected (see Figure 5-3).

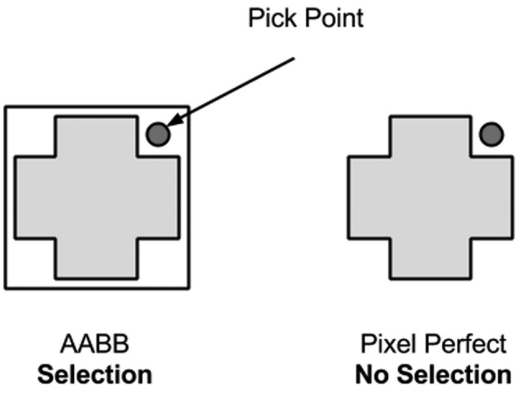


Figure 5-3. *Valid areas for bounding-box picking vs. pixel-perfect picking. Since the bounding-box for an object is larger than its discrete bounds, a selection does not need to be pixel perfect*

Loading Pixel Data

The cornerstone of your picking process is assuming that your images *contain* alpha values, and that you can access those pixels during a pick operation to determine if the selected point is in a transparent or opaque part of the image, which you generally assume is loaded this way:

```
var img = new Image();
img.onload = function(){alert(‘loaded!”);}
img.src = filename;
```

The problem here is that in JavaScript you don’t directly handle the pixel data; it’s handled behind the scenes on your behalf.

To get the data then, you need to do some extra work. You could start by writing a JavaScript PNG decoder, but that would be massive overkill, considering PNGs support lossless compression, meaning you’d have to implement the entire DEFLATE codec by hand. Since you’re really only concerned with the *alpha* values of an image, you could store the alpha channel in a separate .RAW file that you fetch in parallel; however, this would increase the transfer and asset size of the app.

For the sake of your purpose, you ignore those two options, and instead decide to keep the code footprint and transfer sizes low by using the *canvas* element to fetch the data. To do this, you create an off-screen canvas, render your image to it, and *fetch the pixels* of the canvas object back to memory (see Listing 5-10).

Listing 5-10. Fetching the image data by using an off-screen canvas

```
var offScreenCanvas= document.createElement('canvas');
var fetch_ctx = offScreenCanvas.getContext('2d');

// set a max size for our offscreen fetch canvas
offScreenCanvas.width = 128;
offScreenCanvas.height = 128;
function fetchImageData(imgObject, imgwidth,imgheight)
```

```
{
    fetch_ctx.clearRect(0,0,128,128);
    fetch_ctx.drawImage(imgObject, 0, 0);
    var last = Date.now();
    //CLM note this keeps an additional in-memory copy
    var imgDat = fetch_ctx.getImageData(0,0, imgwidth, imgheight);
    var current = Date.now();
    var delta = (current - last)
    console.log("getImageData " + delta  + "ms");

    return imgDat;
}
```

■ **Note** For some odd security reason, you aren't able to use *getImageData* to fetch the pixels for an image file that isn't being served off the same origin as the code. As such, if you try the code in Listing 5-10 by simply loading your file in a browser, it will fail. You need to serve the code from some simple web server and browse to it appropriately. The pixel-perfect picking example in the source code comes supplied with a simple Python script to create a web server to load the example from. For more information on this type of thing, try searching for "Same Origin Policy getImageData."

This allows you to transfer a smaller asset footprint, keep using your PNGs/ GIFs or whatever other compression footprint you want, and still get the RGBA data available in main memory during load time (see Listing 5-11).

Listing 5-11. Updating the SpriteProto definition to handle pixel data

```
function SpriteProto(){
    this.filename="";
    this.imgHandle=null;
    this.imgData=null;
    this.size={w:0,h:0};
    this.load= function(filename,w,h)
    {
        var targetSpriteProto = this;
        this.size.w = w;
        this.size.h = h;
        var img = new Image();
        img.onload = function(){
            targetSpriteProto.imgHandle = img;
            targetSpriteProto.imgData = fetchImageData(targetSpriteProto.imgHandle,w,h);
        }
        img.src = filename;
    }
}
```

Testing a Mouse Click

Now that you have your per-sprite image data, as well as in-game instances of *referencing* that data, you can continue with the process of determining which of those objects intersect a given point on the screen, taking into account pixel transparency.

Firstly, the *findClickedSprite* function will loop through all the sprite instances in memory and do a bounding box test against the picking point; you assume that array lookup is a performance limiting action in JavaScript, and this bounding-box test allows an early out for items that don't potentially intersect with the pick position.

Once you find a sprite instance whose bounding box intersects with the picking point, you grab the sprite prototype and translate the canvas-relative mouse position to a sprite-instance-relative position that you use to test against.

■ **Note** that you're still utilizing the bounding-box picking style covered earlier. Per-pixel picking is quite an intensive inner-loop operation: the lookup and calculation of pixel data will easily cause performance problems. As such, using a bounding-box specific test first allows you to create a list of "potential" selected sprites (omitting objects that don't intersect with the target in any way) which you can then continue forward with to do pixel-perfect testing on.

These values are passed to a function on the sprite prototype to determine if the target pixel is transparent or not. If you're clicking an opaque pixel for this sprite, you set this as the selected sprite (see Listing 5-12).

Listing 5-12. Updating the findClickedSprite function to take into account pixel-based picking

```
function findClickedSprite(x,y)
{
    var pickedSprite = null;
    var tgtents = spriteInstances;
    //loop through all sprites
    for(var i =0; i < tgtents.length; i++)
    {
        var sp = tgtents[i];
        //pick is not intersecting with this sprite
        if(    x < sp.pos.x || x > sp.pos.x + sp.size.w ||
            y < sp.pos.y || y > sp.pos.y + sp.size.h)
            continue;

        var ps = sp.spriteHandle;
        //get local coords and find the alpha of the pixel
        var lclx = x - sp.pos.x;
        var lcly = y - sp.pos.y;
        if(!ps.isPixelTransparent(lclx,lcly))
        {
            pickedSprite = sp;
        }
    }
}
```

Again, in this particular example, you're somewhat naive about z-index collisions and visible ordering to the user. The fix for this is somewhat simplistic: if you've found a pick collision, you test the `zIndex` of the new sprite with the `zIndex` of the existing picked sprite to see which one is closer to 0 (which represents closer to the camera), as shown in Listing 5-13.

Listing 5-13. Updating the picking logic to take into account zIndex depth

```
if(!ps.isPixelTransparent(lclx,lcly))
{
    //do depth test (if applicable)
    if(pickedSprite && sp.zIndex < pickedSprite.zIndex)
        continue;
    pickedSprite = sp;
}
```

The *isPixelTransparent* function for the proto-sprite does very simple logic. It effectively fetches the pixel of the image in RGBA (that is, Red, Green, Blue, Alpha) form, and tests if the alpha channel of that pixel is greater than some threshold. The threshold is important, as most artists can add gradient falloffs, drop shadows, and other items which increase the visual of the item, but shouldn't be considered for picking purposes; see Listing 5-14.

Listing 5-14. isPixelTransparent is an important function which determines if a given pixel should be considered for collision or not

```
function SpriteProto(){
    //other stuff here
    this.isPixelTransparent=function(lclx,lcly)
    {
        var alphaThreshold = 50;
        var idx = (lclx*4) + lcly * (this.imgData.width*4);

        var alpha = this.imgData.data[idx + 3];

        //test against a threshold
        return alpha < alphaThreshold;
    };
}
```

Results and Caveats

The results, as shown in Figure 5-4, are quite nice. You can select the right object out of a very complex pixel coverage area.

Figure 5-4. *The results of a pixel-perfect picking. Users can select sprites which may be complex or hidden under other objects in Z-ordering*

While this method works, and produces pixel-perfect results, it presents two primary issues.

1. Image data, which is normally stored in your JavaScript layer behind the scenes, now has to be duplicated in your scripts. As such, this results in a larger memory footprint, often more than double the size since your in-memory copy is uncompressed.
2. It's currently unclear how an array look-up affects performance in JavaScript under the hood. In C++ you have the ability to avoid CPU addressing issues like L2 Cache optimization for array traversal, which is completely missing in JavaScript. On my 12-core work machine, a single pick against 4096 images takes around ~2ms. I'd imagine on a phone it would be *significantly* longer.

And finally, it's unclear if you *really* need pixel-perfect picking for *every part* of your game. For instance, the user may benefit from a more loosely defined picking area that allows an extension of the valid picking area beyond the pixel boundaries around the object, in an attempt to reduce user picking frustration.

Convex Hull Picking

While pixel-perfect picking represents the most resolution-specific solution you can produce for selecting objects, it has a large downside of increased memory footprint and potential performance burden for slower devices. For example, if you had a 1024x1024 image, that may only be 64k in PNG form, but once you fetch it to main memory, it's now 4MB. There's really no (good) way around this, since the `getImageData` function on canvas returns RGBA data uncompressed; even if you pass in a grayscale image, you'd get the full pixel footprint.

Ideally, it would be great to get a lower-memory *representation* of the image without having to store the whole thing in memory. And to that end, I'm going to introduce the concept of using convex hulls for picking.

Effectively, a convex hull is a minimum representation of the *shape* of your sprite, without curving inward towards itself (i.e. being concave). See Figure 5-5 for a representation of a convex hull versus a traditional bounding box.

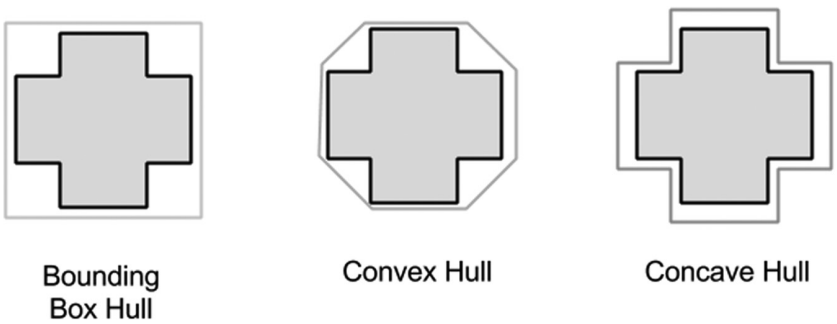


Figure 5-5. *Visualizing the difference between a bounding box (which is always some form of rectangle), a convex hull (which is not allowed to "curve inward" on itself) and a concave hull (which allows itself to curve towards the source object boundaries)*

Convex hulls have been used for years in game development as a rough estimation of shape for 2D and 3D objects; they're especially useful in physics engines, where calculating the intersection of concave objects is roughly 2x slower than intersection of convex ones. Figure 5-6 shows a convex hull of one of the example sprites.

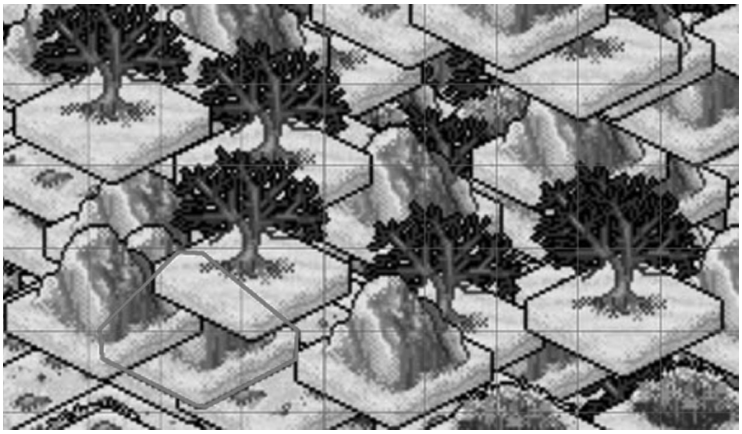


Figure 5-6. *A convex hull in action. Advanced data structures can accelerate your picking code, while in some cases sacrificing resolution perfection*

Once the mouse selection is pressed, you'll test the mouse-point against the convex hull of sprite instances to determine if it's inside or outside a target object.

You will lose some resolution on this process; that is, your results won't be pixel-perfect any longer, but there's a whole separate discussion about *how precise* your picking code needs to be, especially on mobile, where pixels are (generally) smaller than peoples' fingers. Figure 5-7 illustrates the difference in precision for the different selection methods.

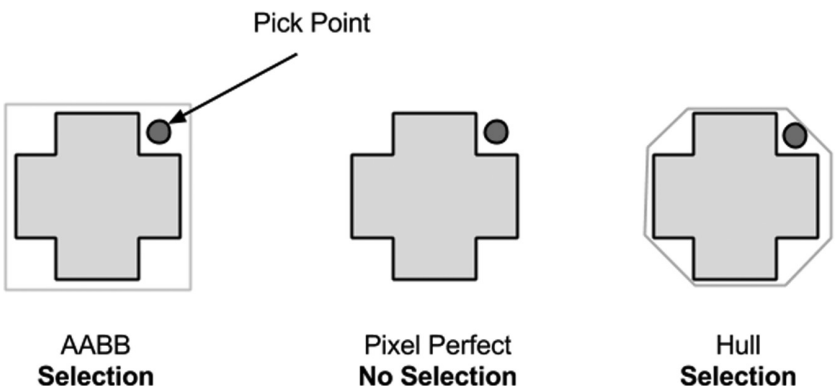


Figure 5-7. *Bounding box, pixel-perfect, and convex hull regions for the same image. Notice that convex hulls are smaller than bounding boxes, but larger than pixel-perfect. This trade-off allows it to be both semi-fast and semi-accurate when needed*

Generating the Convex Hull

Generating a convex hull should be done offline, ahead of time, so that you can reduce the loading time for your HTML5 game. As such, I threw together a simple C++ app that loads a sprite, calculates its convex hull, and outputs JSON data that you include in the HTML file. Your mileage may vary.

I'll spare you walking through the C++ details here, as the code itself is simple. It opens an image and calculates for each scan-line the min/max pixels that represent alpha boundaries (see Figure 5-8). From there, it uses a modified QHull algorithm (which you can get at `www.qhull.org`) to determine what the maximum convex hull is for the set of spatial points.

Figure 5-8. *Hull generation process. Per scan-line, you find the min-max pixels for that row and toss those at a convex hull generator*

The hull points are dumped to individual files, which for simplicity, I've manually added the hull data to the HTML file.

```
var cHullData=[

{"name":"0.png", "hull":[{"x":0,"y":16}, {"x":1,"y":15}, {"x":31,"y":0}, {"x":34,"y":0},
{"x":64,"y":15}, {"x":65,"y":16}, {"x":65,"y":25}, {"x":64,"y":26}, {"x":34,"y":41},
{"x":31,"y":41}, {"x":1,"y":26}, {"x":0,"y":25}, {"x":0,"y":16}]},

//See source code for the full list of hulls defined by our tool
```

Once again, it's worth pointing out that this method of spitting out per-file information may not be the correct way to handle the issue of getting hull information into your data chain (your mileage may vary). It's merely provided as an example to help you understand the process and determine the right integration paths that best suit your needs.

Doing Picking Against the Convex Hull

To transition from per-pixel picking to hull picking, you need to make a few small changes. Firstly, a sprite prototype needs to load the hull data, rather than fetching the image pixel data. This is an easy task, as the hull building app will spit out the image name for a hull, so that when an image loads, you can look up the proper hull for the image (see Listing 5-15).

Listing 5-15. Updating the SpriteProto definition to support convex-hull picking

```
function SpriteProto(){
//...

this.load= function(filename,w,h)
    {
        var targetSpriteProto = this;
        this.size.w = w;
        this.size.h = h;
        var img = new Image();
        img.onload = function(){

            targetSpriteProto.imgHandle = img;
            for(var u =0; u < cHullData.length; u++)
            {
                var thull = cHullData[u];
                if(thull.name == filename)
                {
                    targetSpriteProto.hullData = cHullData[u].hull;
                    break;
                }
            }
        }
        img.src = filename;
    };
})
```

To determine if a mouse click (i.e. point) is inside your new convex polygon hull, you utilize a method of *point-in-polygon testing* known as *ray casting*, which performs its test by casting a line through the 2D polygon, through the point in question (see Figure 5-9 for an example). For each line that your ray intersects, you toggle a Boolean value to determine if you're in or out of the polygon; see Listing 5-16.

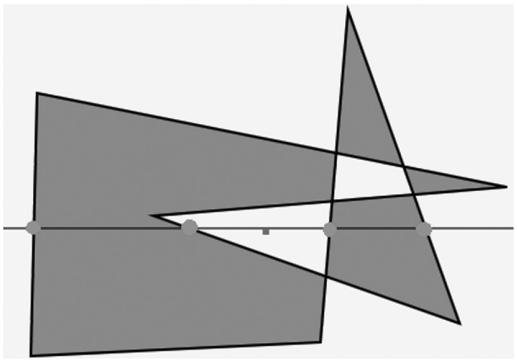


Figure 5-9. *An example of ray casting to determine if a point is in a polygon. The line runs through the polygon an even number of times, signaling that the point is not inside the polygon boundaries*

Listing 5-16. isPixelContained will determine if a given point resides inside the convex hull for this sprite

```
function SpriteProto(){
//...
//--------------
    this.isPixelContained=function(lclx,lcly)
    {
        var inPoly = false;
        var numPoints = this.hullData.length;
        var j = numPoints-2;
        var latLng={x:lclx,y:lcly};
        //from http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html
        for(var i=0; i < numPoints; i++)
        {

            var vertex1 = this.hullData[i];
            var vertex2 = this.hullData[j];

            if ((vertex1.x < latLng.x && vertex2.x >= latLng.x) ||
(vertex2.x < latLng.x && vertex1.x >= latLng.x) )
            {
                if (vertex1.y + (latLng.x - vertex1.x) / (vertex2.x - vertex1.x) *
(vertex2.y - vertex1.y) < latLng.y)
                {
                    inPoly = !inPoly;
                }
            }
            j=i;
        }
        return inPoly;

    }
} // end of SpriteProto function
```

Caveats

As mentioned, this method will reduce the overall memory footprint for your data significantly (compared to per-pixel testing), and on some platforms can have the added benefit of faster execution for a given pick operation.

The real "gotcha" with convex hulls is that it takes a bit of management in order to properly fit into your asset build pipeline the correct way. This poses a particularly large issue for those games using texture atlases, as you'll need to calculate the convex hull of each sprite *before* inserting it into the atlas.

Moving Forward

This chapter has discussed a set of techniques which, when used independently, can decrease processing time for picking and increase precision at the same time.

Industrial strength implementations of this type of code, however, will find the right and smart way to *combine* these techniques for the best results. Before you ship this technique in your title, consider some of the following modifications:

- For example, having a separate spatial grid for dynamic vs. static objects will help create early-out logic for frequently updating objects that tend to complicate your binning selection.
- Only allow visible objects to be pickable. It doesn't help your performance to check objects which are off-screen and not visible to the user, especially in large levels, where the majority of objects are outside the view bounds.
- Per sprite, define what type of picking collision it should use. Some images are fine with box-only picking; for others, convex hull should be enough; and for the rare few that need it, use pixel-perfect solutions.
- Don't forget that as your scene changes, so does the complexity of picking. For example, a tall building may hide multiple objects behind it, which may not need to be selectable any longer. Updating your scene representation to take this into account is ideal.
- Animated images complicate picking quite easily. Your art chain will need some way to output pixel/hull/bounding information per frame of animation and update your sprite representation to produce picking information for the current frame of the animation.
- Transforms and scales of bitmaps also complicate things. Most of the simplistic logic presented in this article quickly becomes error-prone when you rotate and scale bitmaps.
- One of the most annoying results of picking is pick complexity, or rather when the user clicks on objectA, meaning to get to objectB below it. If this is a common complaint, consider adding code paths to allow a multi-click within a tolerance (so if the user clicks the same location twice) to select the second or third stacked object. (In this article I only discussed returning the CLOSEST object to the pick.)

CHAPTER 23

Automating Your Workflow with Node.js and Grunt

Jesse Freeman, Developer Evangelist, Amazon

Insanity: doing the same thing over and over again and expecting different results.

—Albert Einstein

As developers, we tend to do a lot of repetitive tasks. We are constantly compiling code, packaging it, and deploying it to different places. As the project scales, the complexity of these processes continues to grow. If there is one constant among humans, it's that we are not great at doing repetitive tasks, especially complex ones. Sure, we can do one single task over and over again, but anything more complicated, and the system quickly breaks down. This isn't anything new; Henry Ford realized it when he took advantage of the assembly line to lower the cost of making cars. We can do the same thing for our own code by taking advantage of automation.

Automating Your Workflow

If you come from working with other programming languages, such as Java or ActionScript, the idea of automation may not be alien to you. When compiled, these languages rely on Java, and, as such, it is easy to integrate them into build-script languages, which are built on the same languages, such as Ant, Maven, and even Scala. As JavaScript developers, we can take advantage of `Node.js` to handle the running of all our automation scripts. `Node.js` is incredibly powerful, and, when paired with a build system, such as Grunt, , we can attain results similar to those other languages.

Right now you may be thinking, "How can a build script help me with my game?" JavaScript isn't compiled like Java, and, in order to package it, you simply upload it to the server. However, if you have tried to support multiple platforms before, or looked for ways to compress and optimize your code, you know that there is more involved than just sending your code up to a server via file transfer protocol (FTP) to get it running. Let's take a look at some of the unique advantages to automating your workflow, which will be the focus of this chapter.

Optimize and package your game: Games are the biggest abusers of resources when running. Games usually have thousands of lines of code, hundreds of assets, and lots of additional files that need to be loaded at runtime. You can minimize and compress your game's code to shrink its size and combine the requested files to cut down on load time.

Deploy to multiple platforms: HTML5 games live in more than one place nowadays. You may want to publish the game to your server, submit it to a Web store, or put it in a wrapper to run natively.

Create a reproducible build process: If you are working as part of a large team, or even on your own, you will want to make sure that you can quickly set up and run the game's code at the same time whenever you set up a project from scratch or share it with another developer.

As you can see, there is a lot that you can take advantage of with automated build scripts, but it's not easy work. It requires you to sit down and map out your process and how to optimize that process. In this chapter, I will walk you through the basics of how I create my own build scripts and give you suggestions on how to apply these solutions to your own project. To get started, you will need to download and install `Node.js`.

Installing Node.js

One of the reasons I picked `Node.js` to build my automation solution was because of how easy it is to install locally. When most people hear "Node," they think "server side." Little do they know that `Node.js` features a powerful command-line integration with an operating system (OS) that can be run locally. Moreover, `Node.js` runs on Windows, MacIntosh, and Linux, making it ideal for any developer's coding platform of choice. To get started, you will need the correct installer for your OS. You can get it from the official `Node.js` web site (`http://nodejs.org`). As you can see in Figure 23-1, you simply click the Downloads button, and the site will automatically detect your OS.

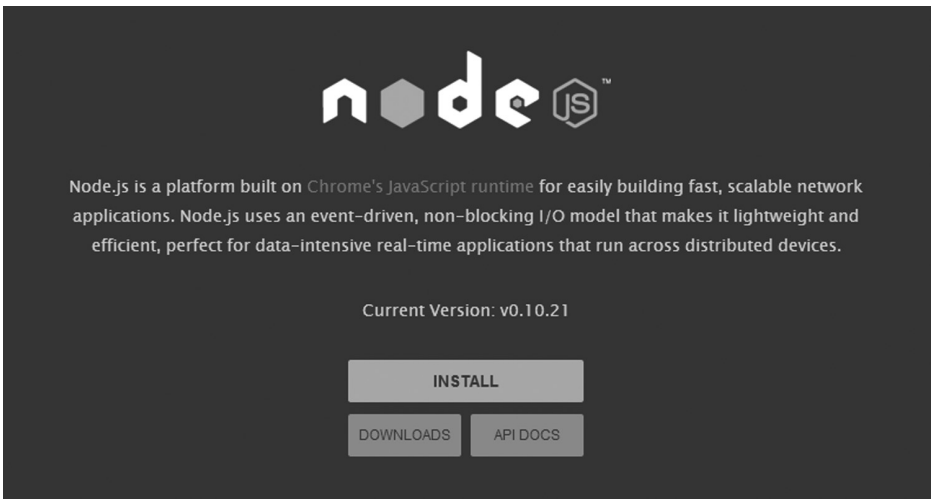


Figure 23-1. *You can get the latest build of Node.js at* `http://nodejs.org`

Once the installer has downloaded, run through the installer wizard, leaving the default settings, as shown in Figure 23-2.

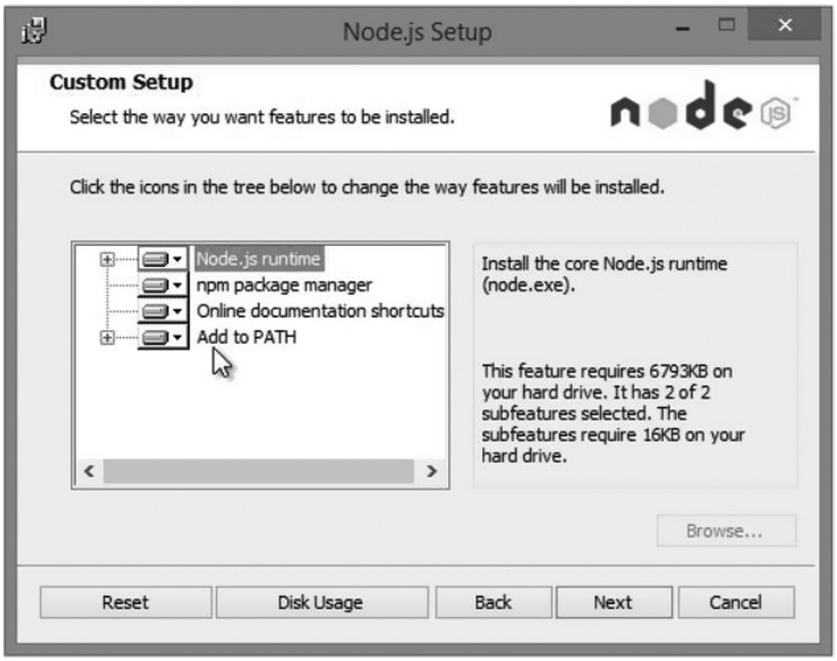


Figure 23-2. *Make sure to install all the default features so that Node.js works properly for your build script project*

There is one important thing to note if you are on Windows. I am not a big fan of the command prompt on Windows, because I come from a Linux background. That being said, I have had great success with Git Bash, which comes installed with Git. You can get them both from `http://git-scm.com`. If you want to use a Bash command line for `Node.js` on Windows, it is critical that you install Git Bash before `Node.js`.

Once you start the installation process, there are two things that you will want to make sure to configure. The first is Windows Explorer integration, which you will enable with the Git Bash Here option, as demonstrated in Figure 23-3.

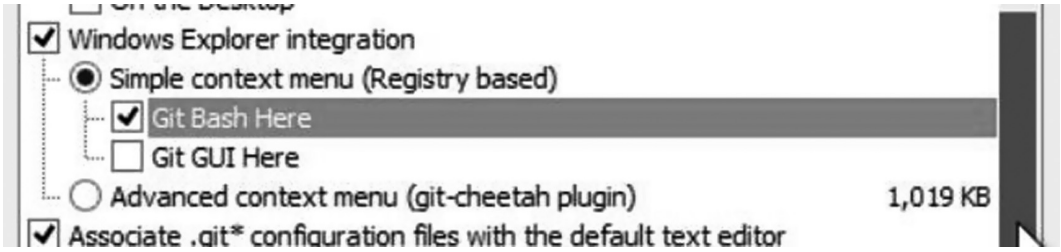


Figure 23-3. *Make sure to add the Git Bash Here option when installing Git Bash*

This allows you to right-click any folder and open Git Bash in that directory. This saves you a bit of time, as opposed to trying to find it via the command line. Next, you must adjust the PATH environment by selecting the Use Git Bash only option, as shown in Figure 23-4.

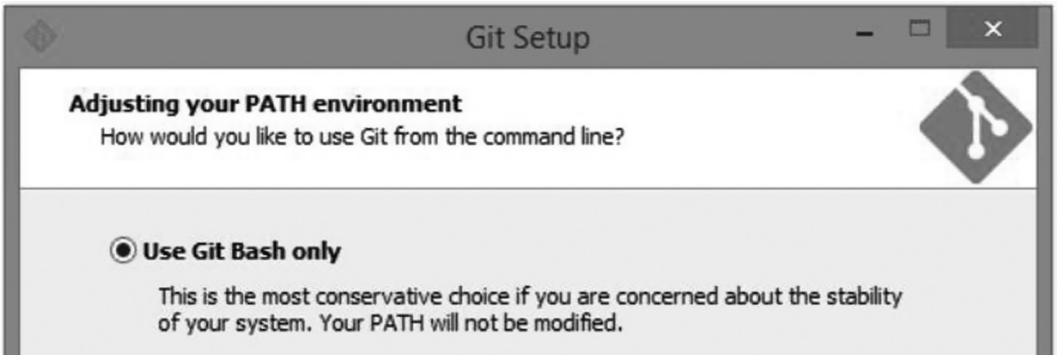


Figure 23-4. *This will make Git Bash the default command prompt for Git and the other command-line tools that you will be using in this chapter*

This option will ensure that Git Bash is your main application (app) for running Git and `Node.js` (once you have it installed) from the command line. For the rest of this chapter, I will be using Git Bash on Windows. If you are on a Mac, because it uses Bash by default in the Terminal, everything should be the same.

By now, you should have `Node.js` installed on your computer. Simply open Git Bash on Windows, or Terminal on Mac; enter the following command at the prompt; and then press Enter:

```
>npm
```

You should see something similar to the screen shown in Figure 23-5.

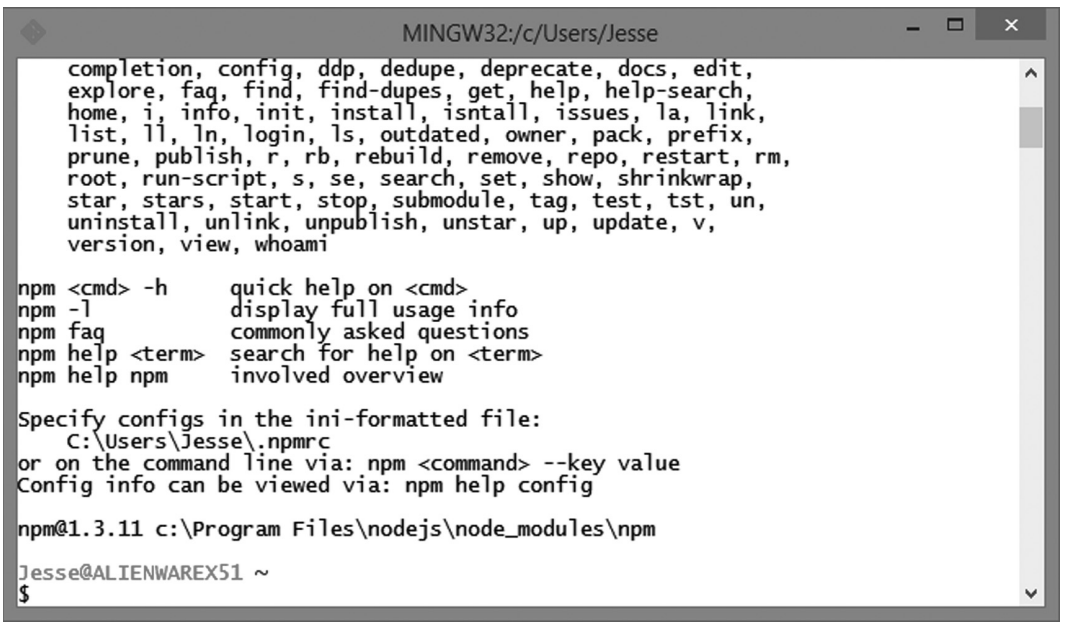


Figure 23-5. *If you don't pass npm any commands, it will print out the instructions*

Here, you are just testing that Node Package Manager (npm) is working and correctly installed. If you don't see instructions to run npm, you may need to restart your computer or try reinstalling Node.

Command-Line Primer

I could have brought this up at the beginning, but I didn't want to scare you off. If you haven't noticed by now, you will be using the command line a lot for running your build script. Some people are terribly afraid of the command line, so I thought I would just cover the basics here. The first thing to keep in mind is that nothing bad will happen, so long as you take your time and are careful about what you enter at the prompt. The command line is an incredibly powerful tool, and, as a Web developer, you should become very familiar with it, especially as you go deeper into code automation and working with servers. For reference, Table 23-1 contains a few of the most common Bash commands that you will be using plus descriptions of how they work:

Table 23-1. *Common Bash Commands and Examples*

Action	Command	Description
Print working directory	`> pwd`	This will show you your current path on the file system.
List contents of directory	`> ls`	This will let you see everything inside the current directory you are in.
Change directory	`> cd /path/to/new/folder`	This will let you navigate into a new folder.
Go back	`> cd ../`	This navigates back one directory.
Make new folder	`> mkdir FOLDER_NAME`	This allows you to make new folders on the file system.
Delete	`> rm FILE_TO_DELETE`	This allows you to delete and file a folder on the system.
Delete all files	`> rm -rf FOLDER_TO_DELETE`	If you are trying to delete a directory with multiple files and folders inside it, you will need to force it to perform the delete recursively.
Clear console	`> clear`	This command will clear the console window.
Stop running command	`Ctrl+C`	Some commands that you run will require input or will continuously run and need to be stopped manually.

There are lots of great resources on the Web that cover how the command line works. It's important to note that these are Bash commands, so if you are not using Git Bash on Windows, and instead favor the default command line, some of these will be different.

To get your feet wet, let's create a new folder for your test project. In the command line, you will use the following commands to create a new directory, move into it, and then make sure that you are in the right place:

```
> mkdir NodeSampleProject
> cd NodeSampleProject
> pwd
```

You can see these commands being run in Figure 23-6.

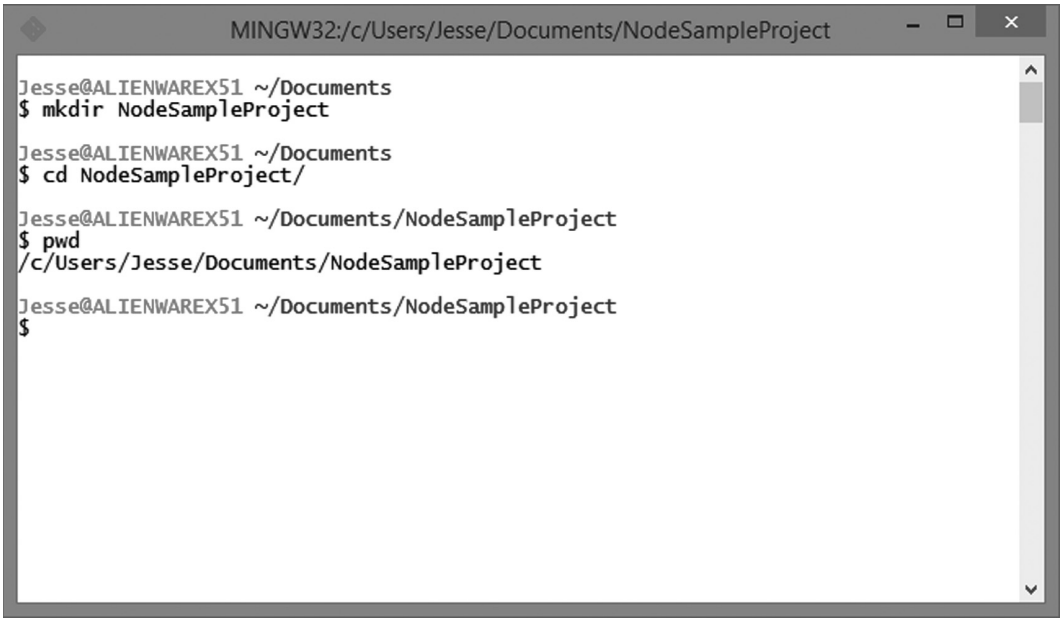


Figure 23-6. *Here, you can see that you have created a new directory, navigated into it via the cd command, and printed the working directory to make sure that you are in the right place*

Introduction to npm

Now that I have covered installing `Node.js` and how to use the command line, you are ready to look at npm, which is at the core of Node. npm allows you to add new features and functionality to your vanilla installation of `Node.js`. On its own, Node doesn't do much outside of allowing you to run it via the command line. There are literally hundreds of packages that you can download from `http://npmjs.org` that can be added to Node to make it more powerful. It is really easy to install new packages. You simply use the following command:

```
> npm install PACKAGE_NAME
```

Over the rest of this chapter, I will discuss different types of packages that you will need to install to create more powerful build scripts. The biggest advantage that npm affords is its ability to help you manage your package dependencies. By tracking which packages you install in your project, it becomes easier to share this list with other team memebers or reinstall a specific version when setting up a project from scratch. To do this, you will have to create a package JavaScript Object Notation (JSON) file for your own project. At this point, you should be inside the new folder that you created in the previous section. Run the following command to activate npm's project setup wizard:

```
> npm init
```

You should see the screen displayed in Figure 23-7.

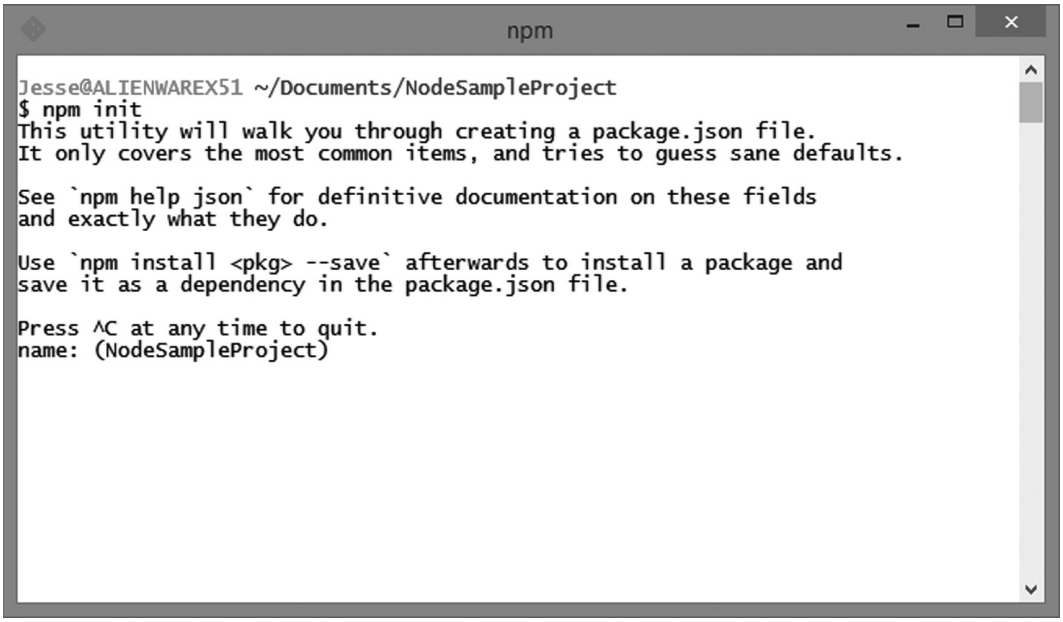


Figure 23-7. *Entering "`npm init`" will begin the process of creating the package.json file*

As you can see, the wizard will automatically create the package JSON for your project. Simply answer the questions the wizard asks. To start, you must give the package JSON a name. Once this is done, the wizard will ask you to confirm all your values, as illustrated in Figure 23-8.

Figure 23-8. *Answer all the questions, and you will see a sample of the package.json file before you save it*

Now, if you look in your directory, you will see a `package.json` file. Open it with your text/code editor of choice, and it should have the following structure:

```
{
  "name": "NodeSampleProject",
  "version": "1.0.0",
  "description": "A sample nodejs project",
  "main": "index.html",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jesse Freeman",
  "license": "BSD-2-Clause"
}
```

Over time, as you add new packages to your project and continue to flesh out your build script, this file will automatically populate with anything you install via npm. You will come back to this file later on, when you start adding more Node modules to your project. For now, let's focus on installing the first package, which is Grunt.

Installing Grunt

Grunt is a JavaScript task runner. This means that you can define a set of tasks, similar to macros in other programs, that can run in order and be strung together to build more complex operations, which will become your final build script. To get started, you will need to install Grunt's command-line tools globally on your computer, which will allow Grunt to work in any project you create moving forward. To do this, enter the following command at the prompt:

```
> npm install –g grunt-cli
```

As the tools install, you should see the downloaded output, as shown in Figure 23-9.

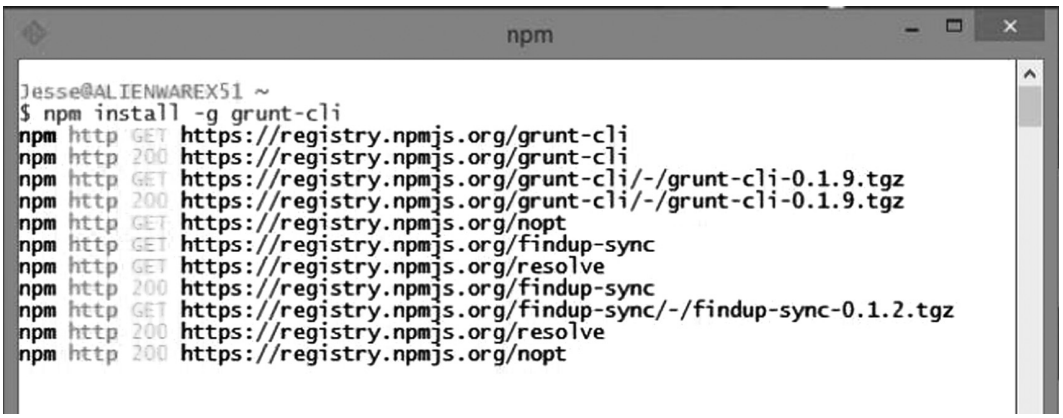


Figure 23-9. *As each part of the package is downloaded, it will be displayed at the command prompt*

Basically, npm is downloading and installing everything required to run Grunt on the command line.

■ **Tip** One thing to note is the –g you used in your command for installing Grunt. This flag indicates that it will be installed globally.

In the next section, you will install your packages locally to the project. Grunt is the only one you want to be able to access in any project on which you're working.

Creating a Grunt File

At this point, you are ready to create your build script. Grunt requires a case-specific file named `GruntFile.js` ×in the root of your project. Let's create that in your code editor of choice, and I will go over how each part of the script works plus how you can add onto it.

Once you have created your `GruntFile.js`, open it, and add to it the following code:

```
module.exports = function (grunt) {
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json')
    });
}
```

This is the basic template for your script. There is not much you can do, so let's set up a basic server and have it display an index page with "Hello World" on it. Create an `index.html` file in your project. Next, you will want to install a simple Node server module, called *connect*. Grunt has its own standard library of modules, called *tasks*. You can see a full listing of these at `http://gruntjs.com/plugins`. You can install them throughnpm. Switch back to the command line, and enter this:

```
> npm install grunt-contrib-connect --save-dev
```

There are a few things to point out here. Recall how you used npm to install Grunt on the command line. Here, though, you will notice that you are not using the global flag –g and instead add `--save-dev`. This is a special flag that tells npm not only that you are going to install this module locally, but also that you want to save a reference to it in your `project.json` file. This is very important, as it will allow you to manage module dependency as your project grows. Let's open the `project.json` file and take a look at what was added there. You should now see the following code toward the bottom of the file:

```
"devDependencies": {
    "grunt-contrib-connect": "~0.3.0",
}
```

Keep in mind that you may be using a different version from what I show here, as the plug-ins are constantly being updated by their authors. Pretty cool, right? Anything you install with `--save-dev` will automatically be added to your dev dependencies in your `project.json` file. Now, when you distribute your project, or if ever have to set it up again from scratch, npm will simply read these dependencies and install them for you. We'll talk about how that works later in this section. For now, you will need to set up the connect module in your GruntFile. Switch back to the `GruntFile.js`, and add the following code just above the `grunt.initConfig` line:

```
grunt.loadNpmTasks('grunt-contrib-connect');
```

This is how you load tasks into Grunt. The module you just installed into your script will now be imported. To configurethe module, you will need to make a new task. Add the following code below the line with `pkg: grunt.file.readJSON`. Make sure that you add a comma to that line first, because you are adding the code to an existing JSON object. The code should look like this:

```
pkg: grunt.file.readJSON('package.json'),
        connect: {
            server: {
                options: {
                    port: 8080,
                    base: './',
                    keepalive: true
                }
            }
        }
```

As you can see, the connect task has its own configuration object. You name the configuration after the task name, and, in this case, you call it connect. Connect accepts several configuration objects. For this example, you are simply going to register a port number and base it on where the root of the server should host. Here, this will be inside the current directory. Also, for this example, you need to set the `keepalive` property to `true`. Connect will only run so long as Grunt is running, so without this, the build script would execute, and the server would be shut down.

Now, at the end of the line of code, just before the closing curly brace, add the following code:

```
grunt.registerTask('default', ['connect']);
```

To run a task, you have to set a task name. Here, you are using default and an array to list the tasks to run, and you are only calling connect. Default is the main task that this script will run, and, as you can probably guess, it will run whenever you call the script. You can create all kinds of tasks with their own unique names. Later on, I'll show you some ways to configure and run them. Before you can run this script, you need to create an `index.html` page for the server to load. Add the following code to the document:

```
<!DOCTYPE html>
<html>
  <meta charset="utf-8" />
  <head>
    <title>Grunt Demo Project</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Now that you have a file to load, go back to the command line, and enter the following command:

```
> grunt
```

This is how you run the build script. The code will automatically call the default task. You should see the server start up at the command prompt, as shown in Figure 23-10.

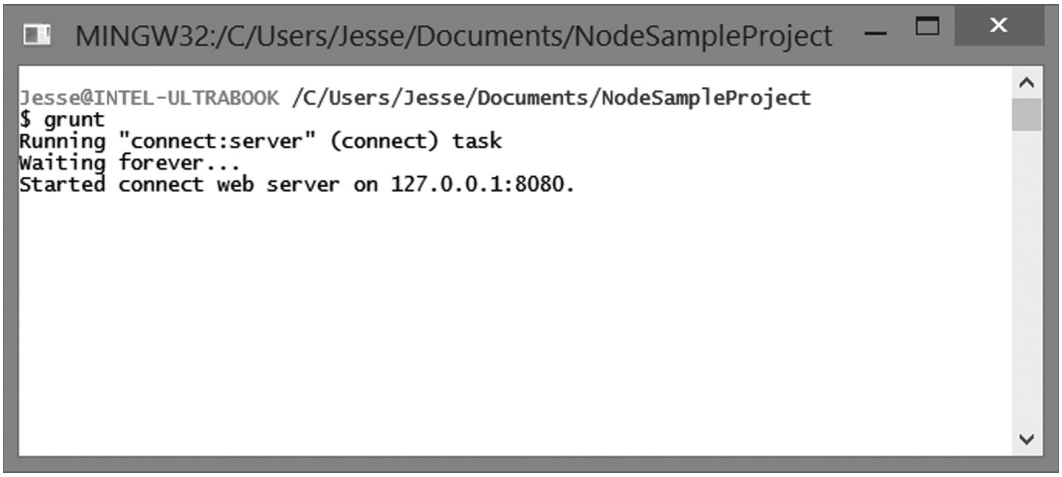


Figure 23-10. *Here, you can see that connect is starting the web server at 127.0.0.1 and port 8080; your specific Internet protocol (IP) address may be slightly different*

This tells you that you have a server running at localhost on port 8080. Open your browser, and enter `http://localhost:8080`. You should now see your Hello World page, as illustrated in Figure 23-11.

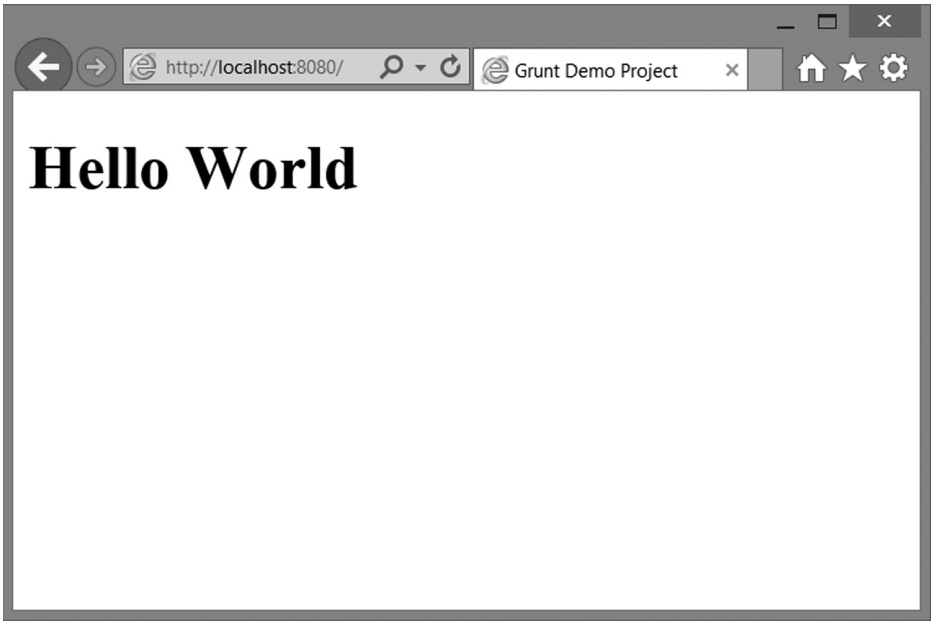


Figure 23-11. *You should now see the Hello World page when you go to `http://localhost:8080`*

As you can see, it's incredibly easy to sett up a server and host a file with Grunteasy. If you have ever had to install and configure Apache, you will appreciate how simple this is. Of course, the server is really basic. It's not running PHP or anything else, but merely hosting the page. For most HTML5 games, this is all you need to do your local testing. Now, let's stop the server. Go back to the command prompt, and press Ctrl+C. This will end the server.

At this point, you have seen the basic steps for creating a build script with Grunt. You find the task module you want to install with npm, import it into your GruntFile, create a configuration object for it, and then add it to your task or make a new one for it. Next, I'll talk about the process I go through when creating a build script and what is useful for HTML5 games.

What Should Your Build Script Do?

Sometimes, the most difficult part of creating a build script is just figuring out what you need it to do without trying to boil the ocean. At the heart of all my HTML5 game build scripts are the following tasks:

1. Copy source code to a tmp folder so that I don't make changes to the original source code or mess anything up.
2. Combine all JavaScript into a single JavaScript file.
3. Inject JSON and other external data into my combined code to cut down on unnecessary external connections and additonal load times.
4. Uglify and minimize to a single JavaScript file.
5. Delete any unnecessary files.
6. Perform builds for each platform to which I want to deploy the game.

As you can see, there isn't a lot here. Each step is critical to the end result, which is packaging the game and getting it ready to publish to a server or app store. From here, there may be subtle variations that you want to do, based on the framework that you are using.

To see how this sort of build script works with an actual project, I suggest you see Chapter 19 . This chapter walks you through how to build a simple, rogue-like game, using TypeScript. Its build process is fully automated, taking advantage of all the content covered in this chapter.

Conclusion

Creating build scripts, or any kind of automated workflow, takes a good deal of work. It's not difficult work, mind you, just a lot of planning and thought on how to build out a robust automation plan as well as how to scale up or down to fit your ever-changing needs.

Over time, you will start to understand better how to break apart your scripts to make them more modular and reusable across all your projects. Creating build scripts is an art form all its own and an integral part of any serious developer's workflow. The time saved on packaging and deploying games alone is worth the up-front investment in making the build script. I wouldn't work on a single HTML5 game without some form of build script, even if it only compressed the JavaScript and minified it before uploading.

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.

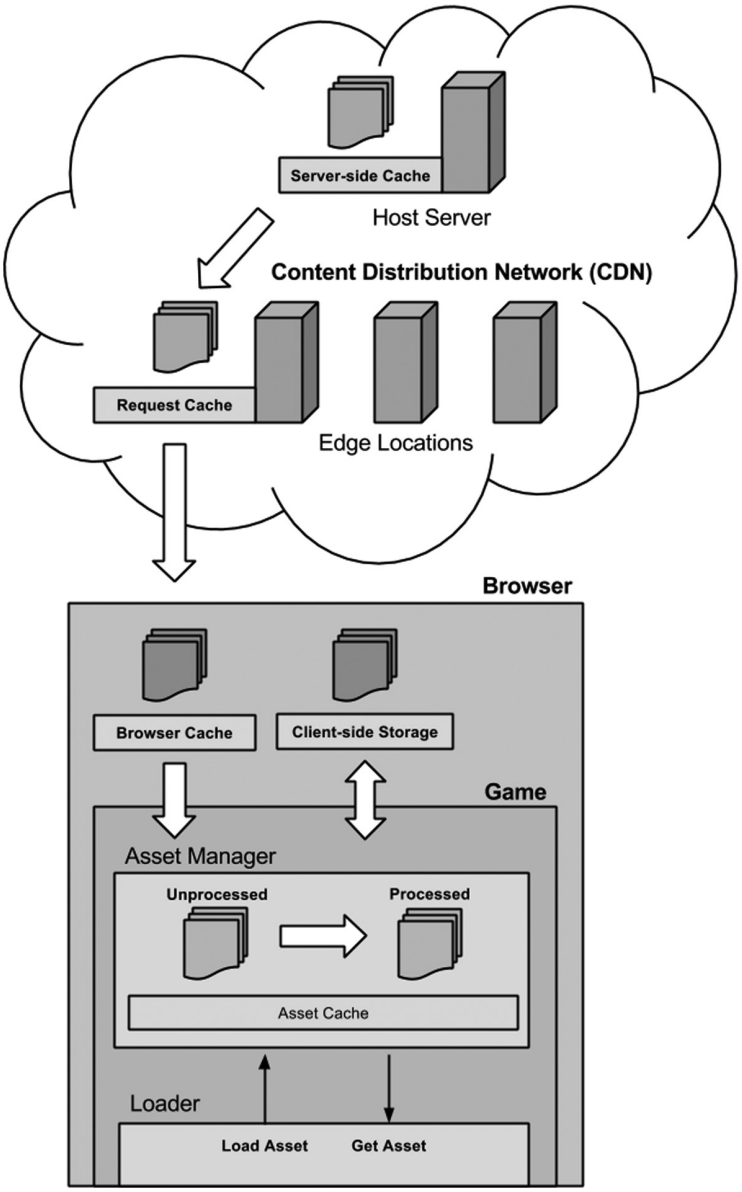


Figure 2-2. *Possible locations on the server and client side where game assets can be cached, from being stored on disk by a remote host server to being stored in memory already processed and ready to use by the game*

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`).

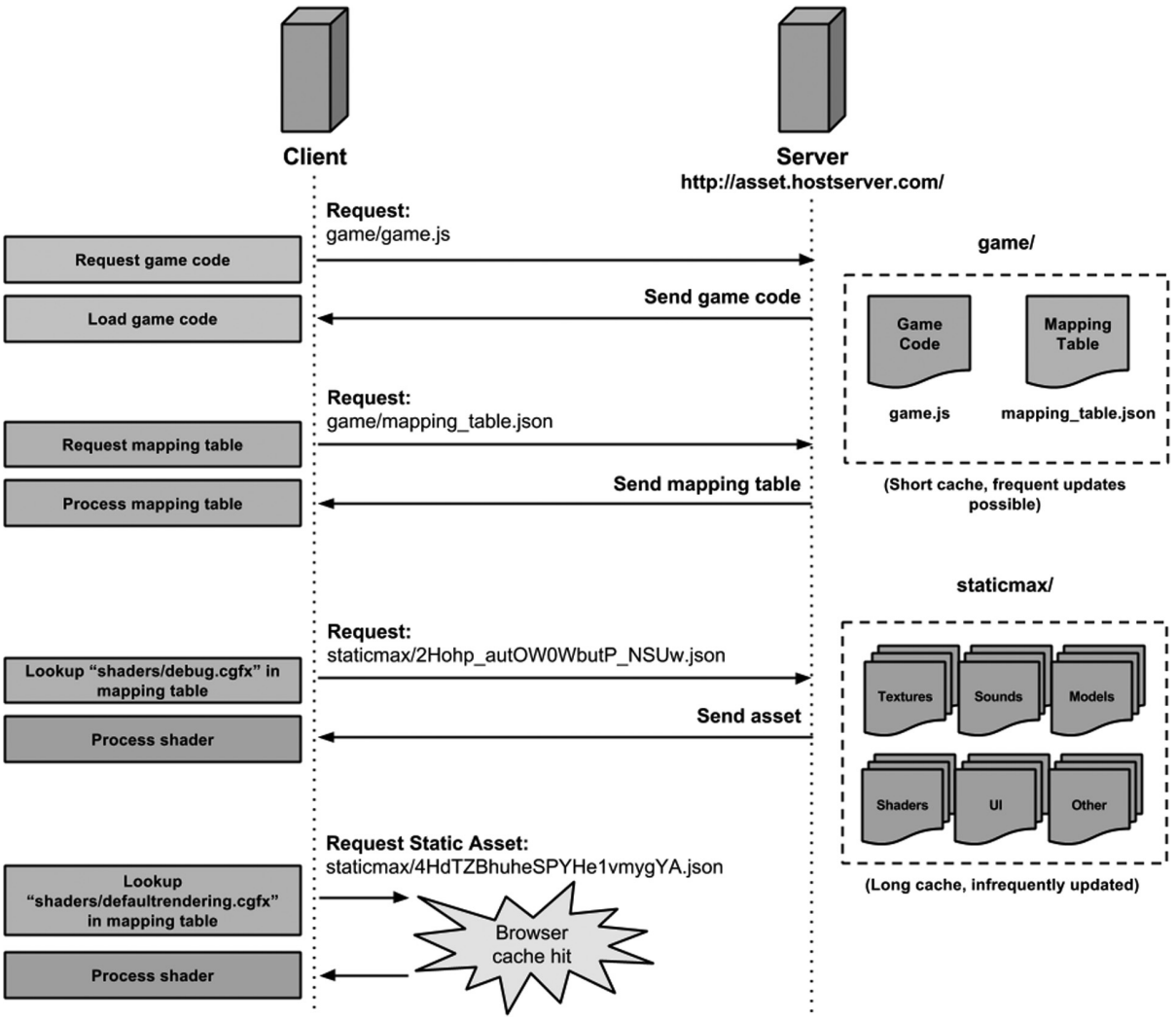


Figure 2-3. *An example of the communication process between a server and the client when using a mapping table. Game code and mapping table data served with short cache times provide the ability to request static assets served with long cache times that take advantage of HTTP caching behaviours*

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;
```

```
            // Call the onDestroy function if the cachedAsset is loaded
            if (this.onDestroy && !cachedAsset.isLoading) {
                this.onDestroy(oldestKey, cachedAsset.asset);
            }
            delete cache[oldestKey];
            // Reset the cachedAsset for the new entry
            cachedAsset.cacheHit = this.hitCounter;
            cachedAsset.asset = null;
            cachedAsset.isLoading = true;
            cachedAsset.key = key;
            cachedAsset.observer = Observer.create();
            this.cache[key] = cachedAsset;
        } else {
            // Create a new entry (up to the maxCacheSize)
            cachedAsset = this.cache[key] = cacheArray[cacheArrayLength] = {
                cacheHit: this.hitCounter,
                asset: null,
                isLoading: true,
                key: key,
                observer: Observer.create()
            };
        }
        this.hitCounter += 1;

        var that = this;
        var observer = cachedAsset.observer;
        if (callback) {
            // Subscribe the callback to be called when the asset is loaded
            observer.subscribe(callback);
        }
        this.onLoad(key, params, function onLoadedAssetFn(asset) {
            if (cachedAsset.key === key) {
                // Check the cachedAsset hasn't be re-allocated during loading
                cachedAsset.cacheHit = that.hitCounter;
                cachedAsset.asset = asset;
                cachedAsset.isLoading = false;
                that.hitCounter += 1;

                // Notify all callbacks that the asset has been loaded
                cachedAsset.observer.notify(key, asset, params);
            } else {
                if (that.onDestroy) {
                    // Destroy assets that have been removed from the cache during loading
                    that.onDestroy(key, asset);
                }
                // Notify the original observer that asset was not saved in the cache
                observer.notify(key, null, params);
            }
        });
    };
```

```
    AssetCache.create = // Constructor function
    function (cacheParams) {
        if (!cacheParams.onLoad) {
            return null;
        }

        var assetCache = new AssetCache();

        assetCache.maxCacheSize = cacheParams.size || 64;
        assetCache.onLoad = cacheParams.onLoad;
        assetCache.onDestroy = cacheParams.onDestroy;

        assetCache.hitCounter = 0;
        assetCache.cache = {};
        assetCache.cacheArray = [];

        return assetCache;
    };
    AssetCache.version = 2;
    return AssetCache;
})();

/**
 * Usage:
 *
 * A textureCache is created to store up to 100 textures in the cache.
 * The onLoad and onDestroy functions give the game control
 * of how the assets are created and destroy.
 *
 * In the render loop the texture is fetched from the cache.
 * It will be rendered if it exists, otherwise will be requested.
 *
 * The render loop is unaware of how the texture is obtained, only that
 * it will be requested as soon as possible.
 */

var textureCache = AssetCache.create({
    size: 100,
    onLoad: function loadTextureFn(key, params, loadedCallback) {
        requestTexture(key, loadedCallback);
    },
    onDestroy: function destroyTextureFn(key, texture) {
        if (texture) {
            texture.destroy();
        }
    }
});
```

```
function renderLoopFn() {
    var textureURL = "textures/wall.png";
    var texture = textureCache.get(textureURL);
    if (texture) {
        drawTexture(texture);
    }
    else if (!textureCache.isLoading(textureURL)) {
        textureCache.request(textureURL);
    }
}
```

In this example, an asset cache is created with a limited size of 100. Imagine that you only have the memory allowance to store 100 textures uncompressed in graphics memory at a given time. The game may require more than this number over its lifetime, but not at a single point in time. The aim is to provide the game with the set of textures it requires most often, which are available as it needs them. In addition, you want to avoid re-requesting a texture where possible. By allowing an asset cache to manage the texture storage, the game is able to access the texture quickly while it is in memory and make a request if it is not.

In this example, the `textureCache.get` function will be called in the render loop every time a texture is required to draw on screen. If the function returns "null", then the texture is not yet in the cache (or not available). If the texture is not already loading, then the game will request it. The request function will in turn call the `onLoad` function that the game provides for the `AssetCache`. This gives the game control over how it loads the asset that the cache will store. In this case, it simply calls a `requestTexture` function, which will request and allocate memory for the texture. It is assumed that this function will locate the texture using the quickest method, whether that be from an HTTP request, the browser cache, or client-side storage. The `onLoad` function also provides a callback to the `requestTexture` function to return the loaded asset. In the case where a request to the `textureCache` is made but exceeds the 100-texture limit, then the cache will find the least accessed texture and call the `onDestroy` function, destroying the texture and releasing the memory. If the game attempts to get a texture that has since been removed from the cache, a request will be made and the process will start again.

This allows the game to access many different textures without having to worry about filling up texture memory with unused textures. Other heuristics such as texture size could be used in conjunction with this approach to ensure the best use of texture memory. The "Leaderboards" sample, which can potentially require hundreds or thousands of textures for avatar images, is a more complete example. It is included as part of the open source Turbulenz Engine (see `https://github.com/turbulenz/turbulenz_engine`).

Data Formats

When building complex HTML5 games with an ever-increasing demand for content, inevitably the amount of asset data required will increase. The previous topic discussed methods to quickly access this data, but what about the cost of the data itself? How it is stored in memory and the processing costs play a part in how quickly the game will be able to load. Memory limitations restrict you from storing all data uncompressed and ready for use, and processing costs have a direct impact on the time to prepare the data.

The processing of data can be a native functionality provided by the browser features of the hardware, such as the GPU, or written in JavaScript and executed by the virtual machine itself. Choosing the appropriate format for the data on a given browser/platform can help utilize the processing and storage functionality available by avoiding long load times and reducing storage cost.

Texture Formats

Anyone who has written web content will be familiar with browser support for file formats such as JPEG, PNG, and GIF. In addition, some browsers support additional file formats such as WebP, which provides smaller file sizes for equivalent quality. The browser is usually responsible for loading these types of images, some of which can be used by the Canvas (2D) and WebGL (2D/3D) APIs. By using WebGL as the rendering API for your game, you may have the option to load other image formats and pass the responsibility of processing and storing the data to the graphics card. This is possible with WebGL if certain compressed texture formats are supported by the hardware. When passing an unsupported format such as JPEG to WebGL via the `gl.texImage2D` function, the image must first be decompressed before uploading to the graphics card. If a compressed texture format such as DXT is supported, then the gl.compressedTexImage2D function can be used to upload and store the texture without decompressing. Not only can you reduce the amount of memory required to store a texture on the graphics card (and hence fit more textures of equivalent quality into memory), but you can also defer the job of decompressing the texture until the shader uses it. Loading textures can be quicker because they are simply being passed as binary data to the graphics card.

In WebGL spec 1.0, the support for compressed texture formats such as DXT1, DXT3, and DXT5 that use the S3 compression algorithm is an extension that you must check for. This simplified example shows you how to check if the extension is supported and then check for the format you require. If the format is available, you will be able to create a compressed texture from your image data. See Listing 2-4 for a simplified example of how to achieve this. The assumed variables are listed in the comment at the top.

Listing 2-4. Checking if the Compressed Textures Extension Is Supported and How to Check if a Required Format is Available

```
/**
* gl - The WebGL context
* textureData - The data that will be used to create the texture
* pixelFormat - The pixel format of your texture data that you want to use as a compressed texture
* textureWidth - The width of the texture (power of 2)
* textureHeight - The height of the texture (power of 2)
*/

/**
* Request the extension to determine which pixel formats if any are available
* The request for the extension only needs to be done once in the game.
*/
var internalFormat;
var ctExtension = gl.getExtension('WEBGL_compressed_textures_s3tc');
if (ctExtension) {
   switch (pixelFormat) {
   case 'DXT1_RGB':
       internalFormat = ctExtension.COMPRESSED_RGB_S3TC_DXT1_EXT;
       break;
   case 'DXT1_RGBA':
       internalFormat = ctExtension.COMPRESSED_RGBA_S3TC_DXT1_EXT;
       break;
   case 'DXT3_RGBA':
       internalFormat = ctExtension.COMPRESSED_RGBA_S3TC_DXT3_EXT;
       break;
```

```
    case 'DXT5_RGBA':
        internalFormat = ctExtension.COMPRESSED_RGBA_S3TC_DXT5_EXT;
        break;
    }
}

if (internalFormat === undefined) {
    // If the pixel format is not supported, fall back to an option that creates an uncompressed
texture.
    return "Compressed pixelFormat not supported";
}

var texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
gl.compressedTexImage2D(
    gl.TEXTURE_2D,
    0,
    internalFormat,
    textureWidth,
    textureHeight,
    0,
    textureData);
```

This simple example does not cover all aspects of using the extension (including checking browser-specific names for the extension and robust error handling), but it should give an indication of how the extension is used. For more information about best practices for using compressed textures, see the Graphics Device implementation as part the open source Turbulenz Engine (`https://github.com/turbulenz/turbulenz_engine`). It is also worth noting that in the future more compression formats may be supported, so check up-to-date documentation.

There are a few key considerations from looking at this example. The first is what to do when a given pixel format is not supported. As mentioned throughout this chapter, the best way to load an asset is to not load it at all, so checking the available formats should happen before the data is even loaded. This allows the game to select the best option for the given platform. In the event that the WebGL extension or any of the required formats are not supported, the game will have to provide a fallback option. This could be by choosing to load one of the browser-supported file formats and accepting the cost of decompressing or, alternatively, if the file format is not supported by the browser, reading the file and decompressing the data in JavaScript. This may not be as bad as it sounds if the task is executed by Web Workers running in the background while other data loads. Remember that the result will be an uncompressed pixel format, which will take up more memory. If you have control of the compression/decompression, you may be able to choose to use a lower bit depth per pixel (e.g. 16-bit instead of 32-bit), which may affect quality but reduce storage. The advice is to experiment with different combinations for your game and instrument the loading time in different browsers on different platforms. Only then will you be able to get a true idea of what best suits your content as a fallback option.

Another consideration is how texture data is loaded onto the client if the browser doesn't support the file format and the cost of doing this. DDS is typically used as a container format for DXT compressed textures and will need to be loaded and parsed to extract the texture data. The files are usually requested using XHR as binary data using the `arraybuffer` response type and can use the Uint8Array constructor to create a byte array that can be manipulated. Once the loader has parsed the container format, the pixel data, which exists as a byte array, can be passed directly to

WebGL to use as texture data. The Turbulenz Engine includes a JavaScript implementation of such a DDS loader to provide this functionality. In most cases, the binary data is just being passed to WebGL so there is little processing cost in JavaScript, which makes this a quick way to load texture data.

For some textures in your game, you may want to include mipmaps. This common practice adds processing costs for image formats that don't include mipmap data, such as PNG and JPEG. If the game requires mipmapped textures created from these formats, they will be generated on the fly, adding to the total load time. The advantage of loading a file format that supports mipmaps such as DDS is that they can be generated offline with more control over the level of quality. This does add to the total size of the file (around 33%, as each mipmap is a quarter the size of the previous level); however, these files tend to compress well with gzip, which can be enabled as a server compression option (discussed later).

The decision for the exact format to use for texture data is often subjective when it comes to quality. The more compression, the smaller the file size and faster load times, but the more visual artifacts that can be seen. Think carefully about options for pixel formats, the use of an alpha channel, the pixel depth, and the compression depending on the image content. The choice of which S3 compressed algorithm (DXT1, DXT3, DXT5) is like choosing between PNGs for images with alpha and JPEGs for images where artifacting is less obvious: it depends on what they support best. Quick loading is the goal, while still attempting to maintain an acceptable level of quality.

Audio Formats

Similar to texture formats; balancing data size and quality of audio files will affect the overall load time. Games are among the power-users of audio on the Web when it comes to effects, music, experience, and interactivity. The many variations of sound effects and music tracks can easily add up and eclipse the total size of the other types of data combined (even mesh data). This of course depends on the game, and you should keep track of how much audio data you are transferring during development. It is therefore important to consider different options for loading sounds that best fit your game.

If you are using sound within your game, you should be familiar with HTML5 Audio and the Web Audio API as the options available for browser audio support. The history of support for both the APIs and audio codecs has been hampered due to the complex issue of patents surrounding some audio formats, which has resulted in an uneven landscape of support across different browser vendors. The upshot is that audio support in games is not just a matter selecting the optimal format for your sound data, but also choosing based on availability and licensing. Familiar formats such as MP3 and AAC are supported by the latest versions of the majority of browsers, but cannot be guaranteed due to some browser manufacturers intentionally avoiding licensing issues. Formats such as OGG and WAV are also supported, but again not by all browser manufacturers. At the time of writing, the reality is that documenting the current support of audio formats would quickly be out of date and wouldn't help address the practicalities of efficiently loading audio for games. The best advice is to look at the following references to help understand the current support for desktop and mobile browsers and to apply the suggestions in this section to your choice of formats: (`http://en.wikipedia.org/wiki/HTML5_Audio` and `https://developer.mozilla.org/en-US/docs/HTML/Supported_media_formats`).

Transferring audio data is similar to other types of binary data. It can either be loaded directly by the browser by specifying it in an `<AUDIO>` tag or via an XHR request. The latter gives the game control of when the sound file is loaded and the option of what to do with it when it is received. The availability of the Web Audio API in browsers means that most games should have a fallback to HTML5 Audio. As with texture formats, testing for support, then deciding which format to use, is preferable to loading all data upfront. This does mean that you will probably be required to have multiple encodings of the same audio data hosted on your server. This is the price of compatibility, unlike textures where the choice is based mainly on performance. See Listing 2-5.

Listing 2-5. A Simplified Example of Loading an Audio File

```
/**
* soundName - The name of the sound to load
* getPreferredFormat - A function to determine the preferred format to use for a sound with a given
name.
*                      The algorithm for this decision is up to the game.
* audioContext - The audio context instance for the Web Audio API
* bufferCreated - The callback function for when the buffer has been successfully created
* bufferFailed - The callback function for the audio decode has failed
*/

// Check once at the start of the game which audio types are supported
var supported = {
   ogg: false,
   mp3: false,
   wav: false,
   mp4: false
};
var audio = new Audio();

if (audio.canPlayType('audio/ogg')) {
   supported.ogg = true;
}
if (audio.canPlayType('audio/mp3')) {
   supported.mp3 = true;
}
if (audio.canPlayType('audio/wav')) {
   supported.wav = true;
}
if (audio.canPlayType('audio/mp4')) {
   supported.mp4 = true;
}

// Audio element is thrown away after the support query
audio = null;

var soundPath = getPreferredFormat(soundName, supported);

var xhr = new window.XMLHttpRequest();
xhr.onreadystatechange = function () {
   if (xhr.readyState === 4) {
       var xhrStatus = xhr.status;
       var response = xhr.response;
```

```
        if (xhrStatus === 200) {
            if (audioContext.decodeAudioData) {
                audioContext.decodeAudioData(response, bufferCreated, bufferFailed);
            } else {
                var buffer = audioContext.createBuffer(response, false);
                bufferCreated(buffer);
            }
        }
        xhr.onreadystatechange = null;
        xhr = null;
    }
};
xhr.open('GET', soundPath, true);
xhr.responseType = 'arraybuffer';
xhr.send(null);
```

In Listing 2-5, the `getPreferredFormat` function is specified by the game to best decide which format to use. The choice should be based on the knowledge the game has about the required sound and how it is used. For example, if the file is a short sound effect (a few seconds in length) that needs to be played quickly, such a bullet fire sound, then selecting an uncompressed WAV file might be the best option, because unlike most MP3/OGG/AAC files it doesn't need to be decoded before use. It depends on the size and length of the sound. This example is unlikely to hold true for music files, which are usually minutes, not seconds, in length and hence are much larger when uncompressed.

Since the Web Audio API requires compressed audio files to be decompressed before use, storage of the uncompressed audio can quickly become an issue, especially on mobile where resources are more limited. Depending on the hardware, this decoding can be expensive for large files, so loading all large data files, such as music, upfront and then decoding is considered a bad idea. One option is to only load the music immediately required and wait until later to load other music.

If the sound needs to be played immediately, then an alternative approach is required for larger audio files. It is possible to combine streaming HTML5 Audio sounds with the Web Audio API, but at the time of writing, support is limited and unstable but should get better with improvements to media streaming in the future. In the absence of such features, one possible approach to speed up loading times is to load a smaller lower quality version of an audio file and start playing, only to replace it with a higher quality version after it has been loaded. The ability to seek and cross-fade two sources in the Web Audio API should make this a seamless transition. This type of functionality could be written at an audio library level. Figure 2-4 shows an example of loading and playing audio data with respect to user-driven events within the game.

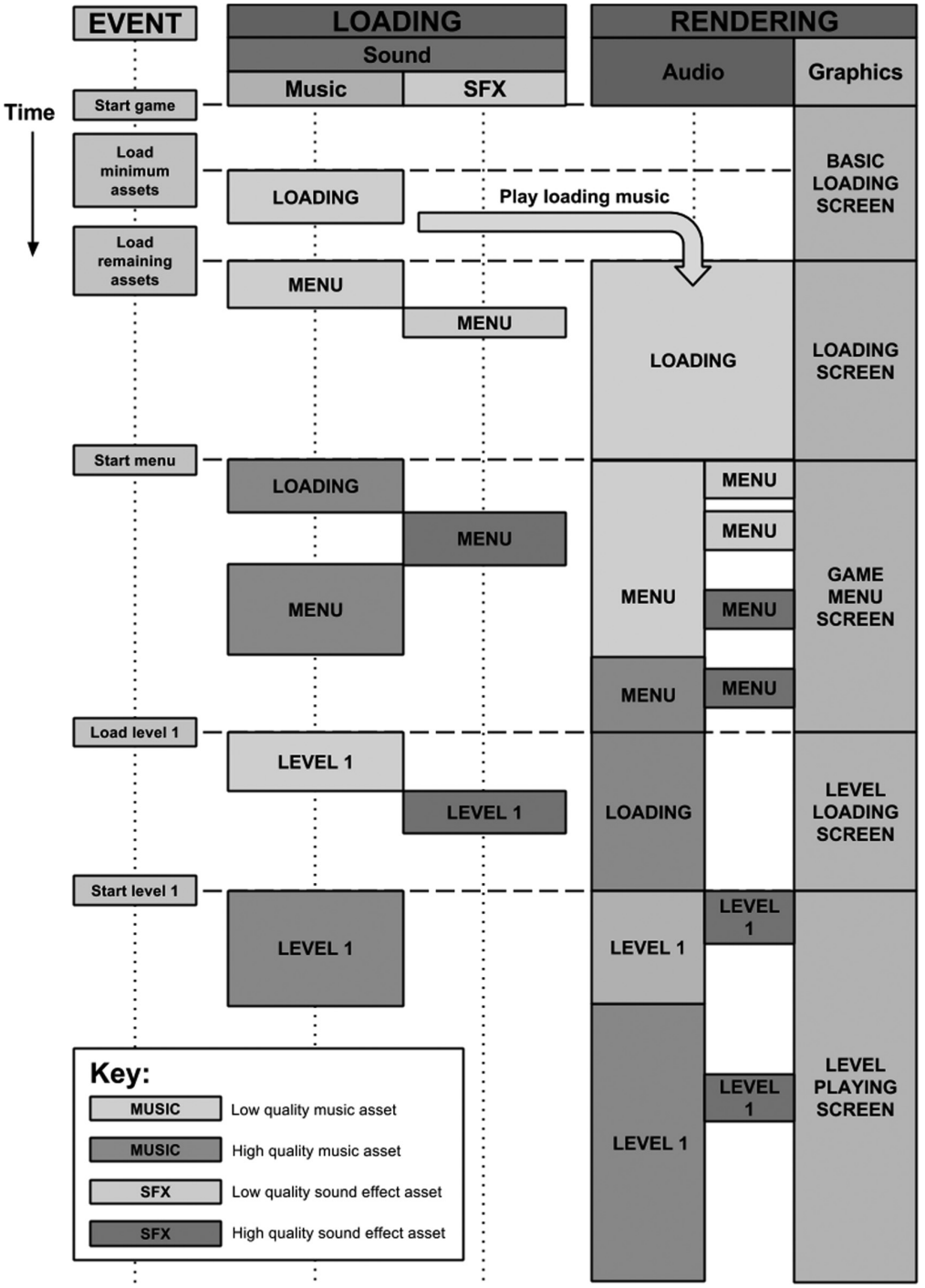


Figure 2-4. *A timeline showing when audio assets are loaded and when they are played. By loading lower-quality audio assets first, sound effects and music can be played sooner and long load times can be avoided. Loading higher-quality sound effects and music can be deferred, leaving the game to decide when is the most appropriate time to process them*

Having dealt with the issues of quality, format support, and encoding, the main choice that impacts loading is how late to defer it, assuming that a connection to the asset data server will still be available. Games providing a combination of MP3, OGG, and WAV audio files should cover the question of browser compatibility. A time will come where the available choices outweigh the limitations and at that point selecting an optimal audio format will be more important.

Other Formats

On the web, JSON is commonly used as a data-interchange format for sites. JSON data has the benefit that, once parsed, it can easily be manipulated as a JavaScript object. This makes it a simple way to transfer text data such as "string tables" for localization, where looking up the data is convenient because it is already in the appropriate format for use. It lends itself to defining extensible data formats where manipulating parts of the data is easy to do by setting properties on objects.

The downside is although `JSON.parse` is a native function in modern browsers, the cost of processing large assets with complex data structures is high, so much so that it can take anything from a few milliseconds to a few seconds to process, depending on the data. This has an impact on loading time, but also on the performance of the game. If the processing takes longer than 16.6ms on a 60fps game, then it can affect the frame-rate of the game, which makes the process of background loading problematic. It is possible to use Web Workers to ensure that the processing is done in a separate thread, which can help reduce the impact of parsing.

Some data, however, lends itself to using a binary format. Accessing this is possible in JavaScript with the help of the `arraybuffer` XHR transfer type and typed arrays such as `Uint8Array`, `Int32Array`, `Float64Array`, and `Float32Array`. This allows you to intentionally transfer floating-point values at a given precision or define a fixed size for your data. If you use a `Uint8Array` view on a transferred `arraybuffer`, you effectively have a byte array for your file format to manipulate as you require. This allows you to write your own binary data parsers in JavaScript. The DataView interface is designed specifically for doing this and to handle the endianness of the data. For more information for how to use these interfaces for manipulating binary data, see `www.html5rocks.com/en/tutorials/webgl/typed_arrays/`.

In addition to structuring your own binary file data to be used in JavaScript, proposals are emerging to standardize many of the common formats that are used by games. One example is the Khronos proposal for **glTF**, a format for transferring and compressing 3D assets that is designed to work with WebGL (`www.khronos.org/news/press/khronos-collada-now-recognized-as-iso-standard`). Another example is the **webgl-loader** project that aims to provide mesh compression for WebGL (`https://code.google.com/p/webgl-loader/`). Proposals like these combine text and binary data and aim to provide a strategy to deliver complex mesh data in a format optimized for web delivery. At the time of writing there is no standardized approach, but it is worth being aware of the data formats that are specifically designed to help deliver certain file content to the Web.

Asset Hosting

When running your game on your local machine, the performance of loading the asset data from a file or in most cases from a locally run web server can appear to be very quick. As soon as you start loading from a server hosted on a local network and then eventually from where your game will be hosted online, it becomes apparent that what you considered an acceptable load time is no longer acceptable in the real world. There are many factors involved when downloading assets; considering how best to host them is key to having consistent behavior for all players of your game. The decisions are not only about where they are hosted, but how they are hosted. Web sites employ many strategies for delivering content quickly to millions of users all over the globe. Since HTML5 games operate as web sites/web apps, the same strategies can be applied to make loading your assets as quick as possible. There are many other considerations for asset hosting, such as cost and storage size, that should not be ignored, but for now let's consider performance.

Server Compression

The transfer time of a file is longer the larger the file, given a fixed bandwidth, so anything that can be done to reduce the file size that doesn't modify the data within the asset can be a benefit. The previous section discussed compression techniques specific to the type of data, but there is also the option of generalized compression algorithms. The majority of browsers support receiving content gzip compressed, such as HTML, CSS, and JavaScript, so compressing large text-based data, such as JSON objects, in the same way is a good idea.

There are many choices of web server technology available to use for hosting, such as Apache, IIS, and nginx. Most servers support enabling gzip as a configuration option and usually allow you to specify which file types to apply it to when a request comes in with the header `Accept-Encoding: gzip`. If a server has it turned on, it should serve the asset gzipped with the header `Content-Encoding: gzip`. There is usually an associated CPU cost of encoding and decoding the gzip on server and client side, but this assumes that the cost is less than the transfer cost of a larger file. Server-side compression costs can usually be eliminated if the response is cached with the gzipped file. Also, not all gzip compression utilities are considered equal. At Turbulenz, we have found that 7-Zip provides very good gzip compression, which can be done offline before uploading the assets to the server. In this case, the server should be configured to serve up the pre-compressed version of the file when gzip is requested. It is worth trying different compression tools to find which one works best for you. For more information about enabling HTTP compression, see `http://en.wikipedia.org/wiki/HTTP_compression`.

Gzip compression is considered essential for uncompressed text formats, but for compressed formats such as PNG it can result in files that are larger than the original compressed file. However, some types of compressed data do benefit from additional gzip compression, such as DXT texture data. The preferred approach should be to compress the file offline and compare with and without compression. If the resulting file sizes are similar, then the added decompression overhead may make loading times slower, and it is probably not worth compressing. If the compressed file size is much smaller, it might be worth considering. If you have limited hosting space, it is worth doing a few tests before choosing which version to upload and serve.

Geolocating Assets

The latency of a request (the time it takes the server to receive, process, and respond) has an impact on how quickly the game will load, regardless of whether the asset is actually transferred or not. Assuming a player is accessing the game for the first time and can't take advantage of any of the client-side caching techniques previously described, what can be done to reduce this latency? Since the latency is related to how far the client is away from the server, the logical solution is to reduce the distance from the player to the host server. As an example, assume that the latency to the server is 100ms (not including transfer time) and assume your game needs to transfer 100 assets. Worst case, that's 10000ms (10 seconds) to query the host server to even find out about each asset (assuming a single server with a single sequential request). In reality, browsers can make more than one request at a time to a given server, but there is a limit to how many requests can be made. If, however, the base latency can be reduced, then savings can be made for every asset request. If the player's bandwidth available to transfer a given asset is the same for two different servers, then the performance benefit will be with the one with lower latency. Having host servers in multiple locations around the world helps reduce the start time for all your players. It's easy to forget this if you only test run the game from a single location.

Using a Content Distribution Network

Providing geolocated assets in multiple locations around the world would require you to run multiple servers and distribute assets among those locations. Luckily, CDNs exist to provide this type of service. The infrastructure provided by companies running these networks can serve internet content up to millions of users with more hosting servers than would be feasible for an individual games company to provide. Pricing, performance, and interfaces vary depending on the provider, so it is worth investigating before making a decision.

As a model for game asset hosting, CDNs can effectively be configured to cache requests made for certain assets server-side, with a given set of request headers to edge locations around the world, which are closer to the players. The advantage is you don't need to distribute your assets to these locations; that task is usually handled by the CDN. Another advantage to CDNs is that they can help provide geolocated DNS lookups. If you request an asset from `cdn.company.com`, some CDN services will use Anycast to connect you to the host that is identified as being closest to the machine that made the request. This is usually managed by the provider's DNS service and the route with the lowest latency may not always be the geographically closest. This allows all requests for an asset to be made to a single URL, but then handled by different servers. It is worth noting that the behaviours of CDNs in terms of configuration differ between providers, so this is just one example of how a CDN can be used.

Imagine, similar to the local server approach, which you are able to set the cache time to be a long period of time for assets that you consider static and are rarely modified. When you first upload the asset to your server and request it via the CDN, which is configured to respect the cache times of the asset in the same way the client would, the CDN edge locations make a request for the file and from that point onwards any identical requests (with the same request headers) will be served from the CDN and not from your asset server. Your server should now have a manageable number of requests from the CDN edge locations and the actual players will only be loading assets from edge locations with the lowest latency for them. If you want to force the CDNs to re-request the asset, you would need a different type of request to be made or the length of cache time to be smaller. By hashing the files as described in the mapping table section, a new hash will indicate a change of file and hence there is no need to update the CDN edge locations. This assumes that the cost of storage for more data is low, but the cost to invalidate the cache and ask the CDN to re-request the files is high.

To replace an existing file with a modified version, a unique id is generated and the new file is uploaded with the id in the URL. The mapping file is also modified to include a reference to the id and uploaded. Unlike the static asset, the mapping table has a shorter cache time before the CDN will re-request it from the game asset server. This allows it to quickly propagate to each of the edge locations when clients request it. Once a client receives a new mapping table, it will request the new asset from the nearest edge location, which will in turn request it from the game asset server. This will need to happen for all edge locations. Once the new asset has been requested and propagated to all edge locations, then all players can access the new asset. The time it takes for this to happen is effectively the "server-side cache" length (i.e., the time from having uploaded the new asset and mapping table to the time when all users can get the new version of the game asset). Because the old mapping table can still be accessed, both versions can be requested. This has the advantage that it can be used to stagger the roll-out of a new update, by selecting to only serve the new mapping table to a percentage of clients. In this scenario, some players will be given the new game with new assets and some the old game with old assets.

Figure 2-5 shows an example scenario in which clients are requesting different mapping tables and assets from a range of edge locations. It also shows the data that needs to be requested by the clients and the edge locations to satisfy the requirements of the game. Client 1 wants to load version 2 of the game. From the mapping table m2, it requires the group of assets known as a2 and aC. Since the client has already played version 1 of the game, it has both a1 and aC in its cache. This means that it will only need to request a2 in order to play the game. It does so by requesting the assets from Edge Location 1. That server has already downloaded the assets from the Host Server, and therefore no additional requests to the host are required. Client 2 wants to load version 1 of the game. From mapping table m1, it requires the group of assets known as a1 and aC. Since the client has not played the game it will have to request both a1 and aC. It requests these from its preferred server, Edge Location 1. Again, the server already has a1 and aC cached, so it does not need to make any requests to the Host Server. Client 3 wants to load version 2 of the game. From mapping table m2, it requires a2 and aC, which it must request because there are no cached assets. This time, the preferred server is Edge Location 2. This server does have the common assets aC, but does not have a2. The server can quickly respond with aC, but it needs to request a2 from the Host Server. This will only happen once for a single client per edge location. After that, all other clients requesting from that server will be served with the cached assets. This simplified example shows how shared content and the distribution among edge locations can reduce the number of requests when serving multiple versions of a game, resulting in less required data transfer.

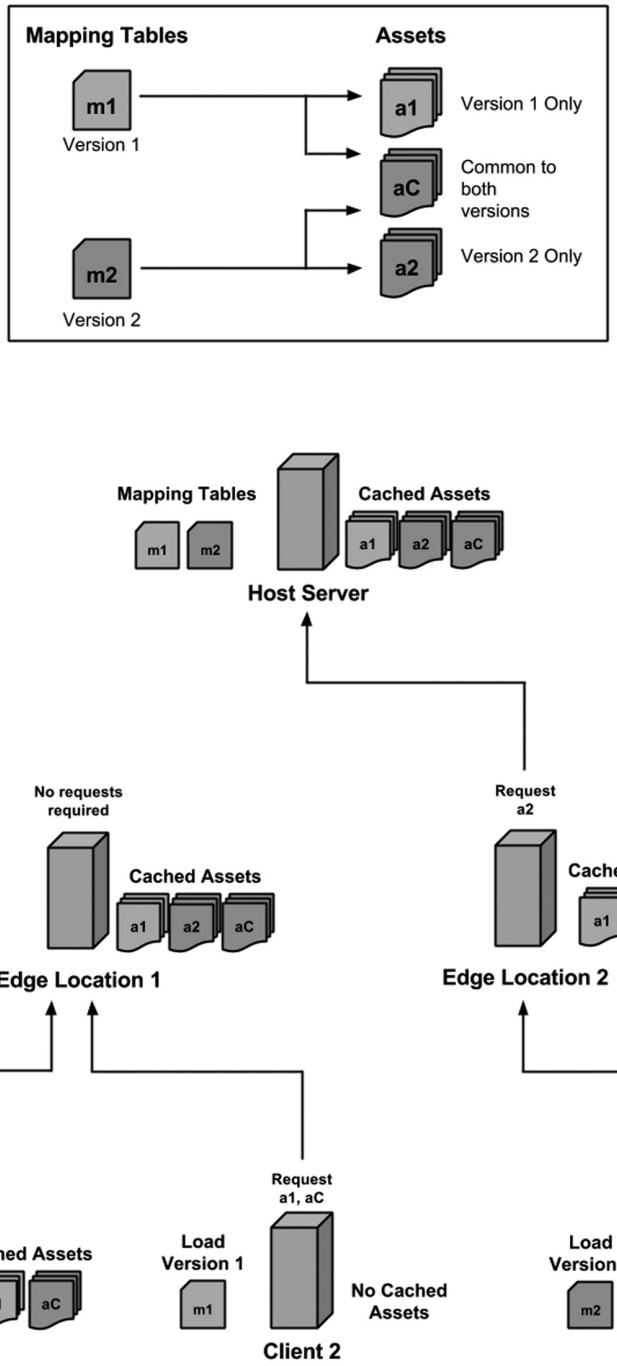


Figure 2-5. *A mapping table strategy combined with a CDN allows you to serve multiple versions of a game with minimal transfer cost between machines. The examples show different scenarios where each client requests assets as referenced by a given version of a mapping table. The requests made vary on the local availability of assets in cache*

Servers play an important role in getting content to your players. You don't need to be a large game developer to experience a runaway success with your game. The last thing you will want is to have hosting and performance issues impeding the playability of your game. A good hosting strategy will allow another service to do the heavy lifting while you focus on providing exciting content.

Effective Asset Grouping

Understanding how files are loaded by the browser and how best to process them is important to have a quick loading game, but reasoning at a higher level about whether to load them is the most effective way of reducing load times. As the game developer, you know when you need an asset, how frequently it is used, and what its dependencies are. For example, to render a simple model you will likely require shaders for all the materials used, one or more textures for the material, and possibly other associated meshes with their own shader and texture requirements. If you are aware of the dependencies between the assets, you can consider grouping them into a single request instead of making separate HTTP requests for each asset. This can reduce the total number requests the browser has to handle and ensures that all asset content is available at the same time. Once again there is a trade-off with the ability to process assets in parallel, but this section assumes that the cost of retrieving the assets separately is higher.

One option is to group assets by type (e.g. sounds, textures, shaders). The advantage is that if you can identify a commonality between them, you may be able to group them in a way that reduces redundant data. The most obvious example is the use of spritesheets to group a number of 2D character animations. In a single texture there would be no duplication of start frame of the animation, for example. Depending on the compression method, you may also see improvements in compressing assets together compared to compressing separately. Another advantage is the ability to easily replace one set of assets with another, for example a texture pack for a model that provides a different skin for the same mesh without having to replace the mesh. This might be useful if you have customizable characters. One disadvantage of grouping by type is that you can easily end up waiting for a single asset type to be processed before the game can progress. Players may end up waiting longer for the game to load before they have a visual/audible indication that progress is being made.

Another option is to group by dependency (e.g. any assets that need to co-exist to be used). One example is grouping a model with associated assets specific to that mesh (e.g. shaders, textures, animation data, sounds). The advantage is that you could be animating the model while loading other models in the background. It is important to make sure that background loading of assets doesn't have an impact on the performance of rendering the model, but that topic deserves its own chapter. One disadvantage of this approach is that shared assets may need to be duplicated or the dependency tree ends up encapsulating the majority of the game assets, losing the granularity of loading in small chunks. To avoid duplicating shared assets, it might make sense to group common assets together, which can then be loaded first before any other group.

Another option is to group by association. One example would be grouping by game level. If the game only requires certain sounds, textures, and other data for a specific level, then it might make sense to only load the data when the level is being loaded. This is especially true if players have to unlock levels or buy additional content. Grouping in this way assumes that levels are mainly independent of each other and that the game will have the ability to access that data at a later point. These high level associations depend massively on the design of the game and how it is played, but by thinking about grouping at this granularity you can make more impactful decisions on load time. Figure 2-6 presents a few scenarios in which grouping may occur.

By Dependency		
Player	Enemy	Weapon
models/player.dae.json	models/enemy.dae.json	models/weapon.dae.json
animations/player_idle.json	animations/enemy_idle.json	animations/weapon_idle.json
textures/player_diffuse.dds	textures/enemy_diffuse.dds	animations/weapon_raise.json
By Type		
Sounds	Textures	Models
sounds/sfx/hit.ogg	textures/player/diffuse.dds	models/player.dae.json
sounds/sfx/blast.ogg	textures/enemy/diffuse.dds	models/weapons/sword.dae.json
sounds/music/intro.ogg	textures/walls/brick.dds	models/map/city/statue.dae.json
By Association		
Common	Menus	Level 1
textures/sky/clouds.dds	ui/menus/main_menu.json	models/boss1.dae.json
sounds/music/background.ogg	sounds/sfx/menu_disabled.ogg	sounds/music/level1.ogg
models/player.dae.json	textures/menus/main_menu.dds	models/map/level1/hut.dae.json
Combined		
Common	Player Common	Player Skin 1
models/world_shield.dae.json	models/player.dae.json	models/player_cloak.dae.json
textures/shield/diffuse.dds	animations/player_idle.json	sounds/sfx/cloak_swoosh.ogg
sounds/sfx/hit_steel.ogg	sounds/sfx/player_jump.ogg	textures/player_diffuse.dds

Figure 2-6. *An example of grouping assets by various scenarios. Combining the options based on your own assets can help reduce the total number of requests required to load your game*

Grouping Using Tar Files

If you are interested in grouping assets together as a single file but maintaining some structure, then using the tar archive format is one option. Tar files allow you to archive files of different types by concatenating them together in a single bitstream, which can later be compressed. This process allows you to group assets while preserving the file structure into a single HTTP request, and allows you the choice of compression techniques to transfer it. Since the resulting file is binary data, it can be transferred and processed as an `arraybuffer`. Tar files contain header information for each file which if processed as binary data gives information about the files contained, for example the filename and filetype. This gives the loading code the option to choose which files to process if and when required. There is an example implementation of a JavaScript based tar loader in the open source Turbulenz Engine (`https://github.com/turbulenz/turbulenz_engine`).

Conclusion

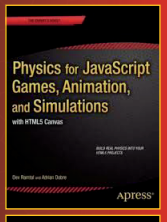
You should now be familiar with different areas that affect the loading times of HTML5 games. By considering the techniques mentioned for caching, data compression, hosting, and data arrangement applied to your game content, we hope it helps you achieve performance improvements in your HTML5 game.

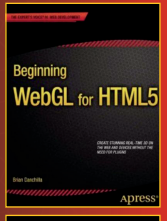
The Turbulenz team has worked hard in the area of optimization to make sure that not only our own games, but games of other developers, including those using the Turbulenz Engine, take advantage of these techniques. Surely there will be many more cost-saving measures in the future as the technology progresses, which we hope to exchange with other developers to ensure that HTML5 and the Web continues to be a powerful and exciting medium for distributing games.

Acknowledgements

I would personally like to thank Michael Braithwaite, David Galeano, Joe Kilner, Blake Maltby, and Duncan Tebbs for their insights into the inner workings of the Turbulenz Engine, their in-depth knowledge of browser technologies, and their feedback. I would also like to thank the rest of the Turbulenz engineering team and Wonderstruck team for constantly pushing the boundaries of what is possible with HTML5 and for making great browser games, which are the driving force for much of the content of this chapter.

RELATED





HTML5 Game Development Insights

HTML5 Game Development Insights is a from-the-trenches collection of tips, tricks, hacks, and advice straight from professional HTML5 game developers. The 24 chapters here include unique, cutting edge, and essential techniques for creating and optimizing modern HTML5 games. You will learn things such as using the Gamepad API, real-time networking, getting 60fps full screen HTML5 games on mobile, using languages such as Dart and TypeScript, and tips for streamlining and automating your workflow. Game development is a complex topic, but you don't need to reinvent the wheel. *HTML5 Game Development Insights* will teach you how the pros do it.

The book is comprised of six main sections: Performance; Game Media: Sound and Rendering; Networking, Load Times, and Assets; Mobile Techniques and Advice; Cross-Language JavaScript; Tools and Useful Libraries. Within each of these sections, you will find tips that will help you work faster and more efficiently and achieve better results.

Presented as a series of short chapters from various professionals in the HTML5 gaming industry, all of the source code for each article is included and can be used by advanced programmers immediately.

What You'll Learn:

- "From The Trenches" tips, hacks, and advice on HTML5 game development
- Best practices for building Mobile HTML5 games
- Actionable advice and code for both professional and novices

Shelve in
Web Development/JavaScript

User level:
Intermediate–Advanced

SOURCE CODE ONLINE

ISBN 978-1-4302-6697-6
54999
9 781430 266976

Apress®
www.apress.com