Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Master's thesis

Petteri Pekonen

Fundamentals of HTML5 game optimization

Examiners : Associate Professor Jussi Kasurinen
D.Sc. (in tech) Jani Rönkkönen

Supervisors : Associate Professor Jussi Kasurinen
D.Sc. (in tech) Jani Rönkkönen

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Petteri Pekonen

Fundamentals of HTML 5 game optimization

Master's Thesis

62 pages, 13 figures, 19 code snippets, 17 tables, 1 appendix

Examiners : Associate Professor Jussi Kasurinen
D.Sc. (in tech) Jani Rönkkönen

Keywords: HTML 5 game, PixiJS, Optimization, JavaScript

This thesis studied optimization of HTML 5 game when using PixiJS game engine, which is currently most used game engine by Seepia Games. This work found Three different categories of optimization. These categories are rendering performance, JavaScript code performance and memory efficiency and package size. This work goes through theory and a few experiments of each of the three categories. In rendering gathering many draw calls into one draw call was found to have significant effect. In JavaScript JIT-friendly code was found to have better performance. Also, few techniques were identified that help with memory management. In size compressing asset was found have significant effect. Also found that file count can have effect on load speed. Lastly in a short comparison of a few HTML 5 game engines PixiJS version 4 was fastest at rendering sprites, while PixiJS version 5 produced smaller package size.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Petteri Pekonen

Fundamentals of HTML 5 game optimization

Diplomityö

2019

62 sivua, 13 kuvaa, 19 koodi pätkää, 17 taulukkoa, 1 liite

Työn tarkastajat: Apulaisprofessori Jussi Kasurinen
TkT Jani Rönkkönen

Hakusanat: HTML 5 peli, PixiJS, optimointi, JavaScript
Keywords: HTML 5 game, PixiJS, Optimization, JavaScript

Tämä opinnäytetyö tutkii HTML 5 peli optimointia PixiJS pelimoottorilla, joka tällä hetkellä Seepia Games:n käytetyin pelimoottori. Työ löysi kolme optimointi kategoriaa. Nämä kategoriat ovat renderöinti nopeus, JavaScript koodin nopeus ja muistin tehokkuus ja paketin koko. Tämä työ käy läpi teorian ja muutaman kokeen, jokaisesta kategoriasta. Renderöinnissä löydettiin, että monen piirtokutsun kokoaminen yhteen piirtokutsuun olevan huomattava vaikutus. JavaScript:ssä JIT-ystävällisellä koodilla löydettiin olevan parempi suorituskyky. Löydettiin myös muutama tekniikka muistin hallintaan. Koossa löydettiin, että tiedostojen pakkaamisella on huomattava vaikutus kokoon. löydettiin myös, että tiedostojen määrällä on vaikutus lataus nopeuteen. Lopuksi lyhyessä HTML 5 pelimoottorien vertailussa PixiJS versio 4 oli nopein renderöimään spritejä, mutta PixiJS versio 5 tuotti pienimmän paketti koon.

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisors Dr. Jani Rönkkönen and Associate Professor Jussi Kasurinen for guidance. I would also like to thank Seepia Games for making it possible to write this thesis while working.

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

CPU	Central Processing Unit
ECS	Entity Component System
FPS	Frames Per Second
HTML	HyperText Markup Language
HTML 5	HyperText Markup Language version 5
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
JIT	Just-In-Time
JPEG	Joint Photographic Experts Group
MDN	Mozilla Developer Network
ms	Millisecond
OpenGL	Open Graphics Library
PNG	Portable Network Graphics
W3C	The World Wide Web Consortium
WebGL	Web Graphics Library
XHTML	eXtensible HyperText Markup Language

1 INTRODUCTION

1.1 Background

Hypertext markup language (HTML) defines structure and content of a web page. It uses different elements to describe structure and type of content. These elements are made from tags surrounded by “<” and “>”. [1] Browsers then read HTML and render it’s content as web page.

Hypertext markup language version five (HTML 5) was first introduced in 2008 and made into a recommendation in 2014 [2] [3]. HTML 5 includes canvas element which allows rendering of shapes and images using JavaScript programming language. This allowed games to be developed using HTML 5.

Web Graphics Library (WebGL) was released in year 2011. WebGL allows access to Graphics processing unit (GPU) through HTML 5 canvas element. This enables 3D rendering and faster 2D rendering. WebGL is based on open graphics library for embedded systems version 2.0(OpenGL ES 2.0). [4] [5] [6] This enables the usage of many OpenGL techniques in WebGL.

In year 2016 Facebook published Instant Games platform. Games for this platform would be made with HTML 5, which enables easy cross platform gaming. These games would be playable from Facebook and Facebook Messenger making them able to reach Facebooks 1.79 billion users and Messengers 1 billion users. [7] In year 2018 Facebook opened Instant Games platform to all game developers [8]. This has created a new large market that has significantly increased HTML 5 game development. Although Instant Games can be played on desktop computers, they should be optimized for mobile since Facebook and Facebook Messenger have significant number of mobile users. 96% of Facebook users used it on mobile device [9] and 47% of Facebook users used it only from mobile device [10]. Also, Facebook Messenger is second most popular mobile messenger app by monthly active users in July 2019 [11].

In recent years playable ads have been getting increasingly popular [12] [13]. These ad games are also made with HTML 5. Playable ad games have strict packet size limitations making size optimization much more important than in regular HTML 5 game. Of course, ad games also must have adequate performance. But since they are so small, they are often simple enough that performance is not major problem.

1.2 Goals and delimitations

This work studies the fundamentals of HTML 5 game optimization. The focus of this work is in PixiJS version 4, although many methods can be used in other game engines. The goal of this work is to find ways to optimize rendering performance, JavaScript performance and packet size of a HTML 5 games. The secondary is to compare the performance of chosen game engines.

In rendering this work focuses on 2D games using WebGL. Although WebGL allows 3D rendering, it is outside the scope of work. Also, shader optimization is not considered in this work and therefore uses game engines default shader which is reasonably optimized.

1.3 Structure of the thesis

Chapter 2 covers earlier work on related optimizations. These are on subjects of rendering, JavaScript and load times. Chapter 3 covers history of used web technologies. Chapter 4 covers theory of rendering. It also covers few other concepts needed to optimize rendering performance. Chapter 5 covers few core concepts of JavaScript. It also covers few techniques to optimize JavaScript. Chapter 6 covers theory of file size and load times. It also covers few techniques to optimize file size. Chapter 7 covers experiments done with PixiJS version 4. Chapter 8 covers comparison between few chosen HTML 5 game engines / graphics libraries. Chapter 9 covers result from experiments and comparisons. Chapter 10 covers found techniques and results. It also covers suggestions for future work. Chapter 11 covers conclusions

2 LITERARY REVIEW

Literary review was done using google scholar to find scientific papers on WebGL rendering, JavaScript optimization and webpage size.

2.1 geometry batching using texture-arrays

Matthias Trapp, and Jürgen Döllner in their work on “geometry batching using texture-arrays” introduces a method of using texture arrays to increase batch sizes. They have identified Application Programming Interface (API) draw calls as performance limiting factor. With increased batch size fewer draw calls are needed. [14]

Their method has three main parts. First one is pre-processing part that gathers all meshes and sorts textured used by these meshes. Second part is batching process which creates batches from geometry and texture data. These batches contain geometry data and mapping to their used textures. These textures are grouped into texture arrays. Third part is evaluating texture mapping during runtime rendering process. [14]

They measured performance in frames per second(fps), with two test computers. They used complex 3D virtual city model of Boston as base for rendering data sets. On slower test computer they got average speed-up of 11.1 times without batching. on faster text computer they got average speed-up of 2.8 times without batching. According to them this difference was caused by different central processing unit (CPU)speeds. [14]

2.2 JavaScript optimization

Vincent St-Amour and Shu-yu Guo in their work on “JavaScript optimization coaching” they used their program that analyses code and gives programmers suggestions on how to change program to optimize it. They run this program in SpiderMonkey JavaScript engine, that firefox uses. They then applied top five suggestions if they were feasible to implement. As test programs they selected Richards, DeltaBlue, RayTrace, Splay, NavierStokes, PdfJS, Crypto and Box2D. On SpiderMonkey they got speed up of 2-17%. They also run these on

Googles v8 and Apples JavaScriptCore and got speed up of 2-20%. Only one program on JavaScriptCore had slowdown. From this they derived that doing optimization coaching on single engine can have speedups on other engines. [15]

Liang Gong, Michael Pradel, and Koushik Sen in their work on "pinpointing JIT-unfriendly JavaScript code" provides a method to identify and measure just-in-time (JIT)-unfriendliness. They do this through dynamic runtime analysis they call JITProf. It identifies code that prevents JIT optimizations. it does this by adding profiles code to program and looking for certain patterns. They had seven code patters that prevent JIT optimization. They applied JITProf to 50 most popular websites and found significant amount of JIT-unfriendly code. They fixed few JavaScript libraries based on JITProf suggestions and gained speedups ranging from 1% up to 26.3% [16]

2.3 Load time

In Fiona Fui-Hoon Nah's work on a study on tolerable waiting time, he creates an experiment to measure how long people will wait for web page to load. Experiment was done by two groups of sophomore business major students who were savvy Web users. They were tasked to retrieve information from specific web pages. They were provided a page with ten links to these specific web pages, but only seven of them worked. They measured time between clicking link and clicking stop button. First groups browsers had feedback telling it was working and second groups browsers didn't have any feedback. On first non-working link first groups average wait time was 38 seconds while second groups wait time was 13 seconds. On second non-working link first groups average wait time was 17 seconds while second groups wait time was 4 seconds. On third non-working link first groups average wait time was 7 seconds while second groups wait time was 3 seconds. Fiona Fui-Hoon Nah speculated that lower wait times were caused by lowered expectations. 61% of no feedback group gave up of first access to second non-working link between 2-4 seconds. 26% of feedback group gave up of first access to second non-working link between 2-3 seconds. [17]

3 WEB TECHNOLOGIES

3.1 HTML

HTML is standardized markup language used to create web pages. It uses elements as basic building blocks to describe structure and content type. These elements are represented by tags surrounded by “<” and “>”. These elements usually come in pairs, where first one is starting tag and second is ending tag. Between these tags is elements content which can contain more elements. HTML standards define all used elements and their meanings which browsers then use to render web pages. HTML has ability to link to either other web pages or different parts of same web page. [1] [18]

In year 1989 Tim Berners-Lee wrote the first proposal for internet-based hypertext system at CERN [19]. The first publicly available document about HTML was “HTML tags”, that defined several elements. It was first mentioned by Tim Berners-Lee in October 1991. [20] [21] This was then further refined by Tim Berners-Lee and Daniel Connolly into an Internet Engineering Task Force (IETF) draft in year 1993 [22]. In year 1995 IETF released HTML 2.0. This was a standard that combined, clarified and formalized HTML features in common use. [23] After IETF working group on HTML was closed in 1996 [24] World Wide Web Consortium (W3C) started to make their own HTML standard. In January 1997 W3C published HTML 3.2 W3C recommendation [25]. In December 1997 they published HTML 4.0 W3C recommendation [26] and HTML 4.01 in year 1999 [27].

After this W3C started focusing on developing eXtensible HyperText Markup Language (XHTML) and published XHTML 1.0 in year 2000 as W3C recommendation [28]. Because W3Cs lack of interest in HTML Web Hypertext Application Technology Working Group (WHATWG) was formed by individuals from Apple, the Mozilla Foundation, and Opera Software [29]. In year 2008 first draft for HTML 5 was published by W3C and with updates made into recommendation in year 2014. This specification was done with WHATWG by trying to combine HTML specification with WHATWG living standard within W3C regulations. [30] [31] This created two groups developing HTML 5 that was resolved in year 2019 when W3C and WHATWG signed an agreement. They agreed that WHATWG will be

developing HTML living standard and produce snapshots which then go through W3C recommendation process. [32]

3.2 JavaScript

JavaScript is interpreted and dynamically typed language. interpreted language means that it has not been compiled to machine code instead each line of code is interpreted. This means that actual code in text is send to clients' which browsers JavaScript engine then interprets. Now days all modern browsers use JIT compilers to compile JavaScript. Even though JavaScript is compiled it is still considered to be interpreted since compilation happens at runtime. [33]

JavaScript being dynamically typed language means that each variable can have any data type. Same variable can even be assigned different data types. Meaning variable can contain one type of data first and then be assigned different type of data. Current JavaScript has seven primitive data types that are listed in Table 1. In addition to primitive data types JavaScript has object data type. [34]

Type	Comment
Boolean	true / false
Null	
Undefined	no assigned value
Number	
BigInt	Can go over Number.MAX_VALUE
String	
Symbol	Unique value that can be used as key for object property

Table 1JavaScript primitive data types

Normal objects are collection of key value pairs. These values can be of any data type thus allowing complex data structures. JavaScript uses prototypes as when creating objects to inherit methods and properties from. These prototype object can have their own prototype object creating a prototype chain. This makes objects to have all methods and properties defined in all prototypes in chain. JavaScript also includes some objects with additional capabilities such as functions and arrays. Functions are normal object that can be called.

Arrays are normal objects which inherit from Array.prototype. This gives them length property and special relationship with integer keyed properties. It also gives some useful functions for manipulating the array. [35] [34]

JavaScript was first published in year 1995 as part of Netscape Navigator 2.0. It was meant to be open and gross-platform scripting language designed to be used in HTML pages. It was then later given to Ecma International standards organization to be standardized. They then produced first edition of ECMAScript specification in year 1997. From there on Ecma has released updated editions of ECMAScript. Since publishing third edition in year 2002 it has become programming language supported by almost all web browsers. All modern java scripts implement ECMAScript specification. [36] [37]

3.3 WebGL

WebGL is a standard for 3D graphics API through HTML 5 canvas element. It is low-level API based on OpenGL 2 ES. This enables hardware accelerated graphics on canvas element, without any browser plugins. The differences between WebGL and OpenGL ES were made for WebGL to work as expected in memory managed environment such as JavaScript. WebGL 2.0 is based on OpenGL ES 3.0. [4] [5] [38]

One of the earliest canvas 3D experiments were done by Vladimir Vukićević in year 2006 [39]. In 2007 Opera also had its own experimental 3D canvas [40]. In year 2009 Khronos Group started a working group to develop WebGL specification. They then published WebGL 1.0 specification in year 2011. Khronos Group also authors OpenGL and OpenGL ES standards. [41] [4]

3.4 PixiJS

PixiJS is an opensource 2D rendering library made by GoodBoy Digital. It is lightweight and fast with full WebGL support and canvas2d fallback. Although it’s not a game engine it provides many useful features that help making HTML 5 games. These features include full

scene graph, asset loading and mouse and touch interactions. All these features allow higher level coding, where you don’t need to do any low-level WebGL coding. [42]

PixiJS was first released in year 2013. This was followed with many performance updates and version 2.0 in year 2014. Version 3.0 was a big refactoring of PixiJS released in year 2015. Next big release was version 4.0 in year 2016, which brought multi texture batching on WebGL. In year 2019 Version 5.0 was released. This major update changed PixiJS structure into components that JavaScript packaging systems, such as webpack, can use. It also removed canvas2d fallback support and focuses even more on WebGL. [43] [44] [45]

4 RENDERING

Rendering is the process of calculating pixel color values of an image that is then shown to user. WebGL based HTML5 games use shader based rendering pipeline as seen in Figure 1 [46] and it is almost identical to OpenGL rendering pipeline as seen in Figure 2 [47]. They both have many stages that start by application sending vertex data and ending in framebuffer that will contain rendered image. [46] [47] These stages allow GPU to work on parallel.

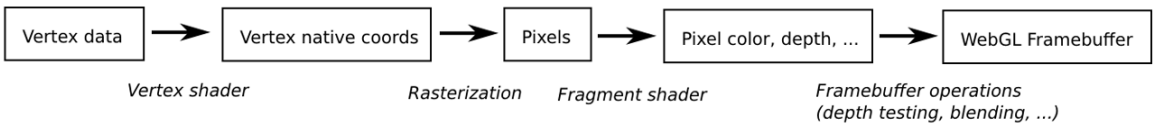


Figure 1 WebGL rendering pipeline [46]

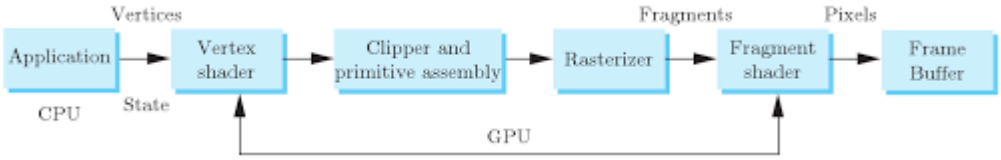


Figure 2 OpenGL renderingpipeline [47]

Out of these many stages all except the application sending vertex data is done in GPU and optimizing these steps are outside the scope of this work. Therefore, this work considers only the first stage which is the application stage. This stage consists of CPU processing scene and making draw calls to GPU through drivers. Draw calls send vertex data and state changes to GPU. These data transfers and state changes are slow operations. These state changes can be for example different texture, blend mode or shader. Since GPUs are very good at doing parallel calculations, small draw calls can't take advantage of this [46]. Slowness of draw call makes reducing number of them to be one of the most common rendering optimizations. Even Mozilla Developer Network (MDN) list few large draw calls better that many small draw calls [48].

4.1 Batching

Since draw calls are major limiting factor in rendering, batching can be used to reduce draw calls. Batching is the process of grouping meshes together to be able to draw them together instead of drawing them individually. This can significantly reduce needed draw calls thus increasing performance. Batching requires that batched meshes share same state. [14]

Effect of batching can be demonstrated with a simple test using PixiJS in legacy mode. Both Code snippet 1 and Code snippet 2 create 8 cino textured sprites and 8 spark textured sprites. Different draw order these snippets create can be seen in Figure 3. In Figure 3 we can see that no interleaving draw order always draws spark textured sprites on top of cino textured ones. Table 2 shows that there is significant difference in draw calls, because batching is not happening in interleaved draw order, but no interleaving is batching nicely.

```
01:  for (let i = 0; i < 8; ++i){
02:    const sprite = new PIXI.Sprite(cinoTexture)
03:    sprite.position.set(i*20, i*20);
04:    app.stage.addChild(sprite);
05:
06:    const sprite2 = new PIXI.Sprite(sparkTexture)
07:    sprite2.position.set(50 + i*20, i*20);
08:    app.stage.addChild(sprite2);
09:  }
```

Code snippet 1 interleaving draw order

```
01:  for (let i = 0; i < 8; ++i){
02:      const sprite = new PIXI.Sprite(cinoTexture)
03:      sprite.position.set(i*20, i*20);
04:      app.stage.addChild(sprite);
05:  }
06:
07:  for (let i = 0; i < 8; ++i){
08:      const sprite = new PIXI.Sprite(sparkTexture)
09:      sprite.position.set(50 + i*20, i*20);
10:      app.stage.addChild(sprite);
11:  }
```

Code snippet 2 no interleaving draw order

(a) *(b)*

Figure 3 Draw call test: a) interleaving; b) no interleaving.

draw order	interleaving	no interleaving
draw calls	16	2

Table 2 Draw calls of interleaving and no interleaving draw order.

4.2 Texture atlas

One of the most common reasons for batches to break into smaller ones is having different textures. This problem can be solved with texture atlas, which is one large texture containing many smaller ones. This texture atlas then makes it possible to make a larger batch, because there is no need to change texture state. [14].

This can be demonstrated using same code as batching demonstration only changing textures to be in atlas. In both cases using texture atlas result is one draw call.

4.3 Texture atlas size

Maximum atlas size is determined by maximum supported texture size of hardware and driver. Texture size of 2048x2048 pixels is a safe size that any reasonable device should support. Figure 4 shows that 4096x4096 pixels textures are supported by almost all currently used devices. Although WebGL Stats is gathers statistics from limited set of sites and is not representative of all users, it should still be good enough indicator. iPhones have supported 4096x4096 pixels textures since iPhone 4s [49], which was released in year 2011 [50].

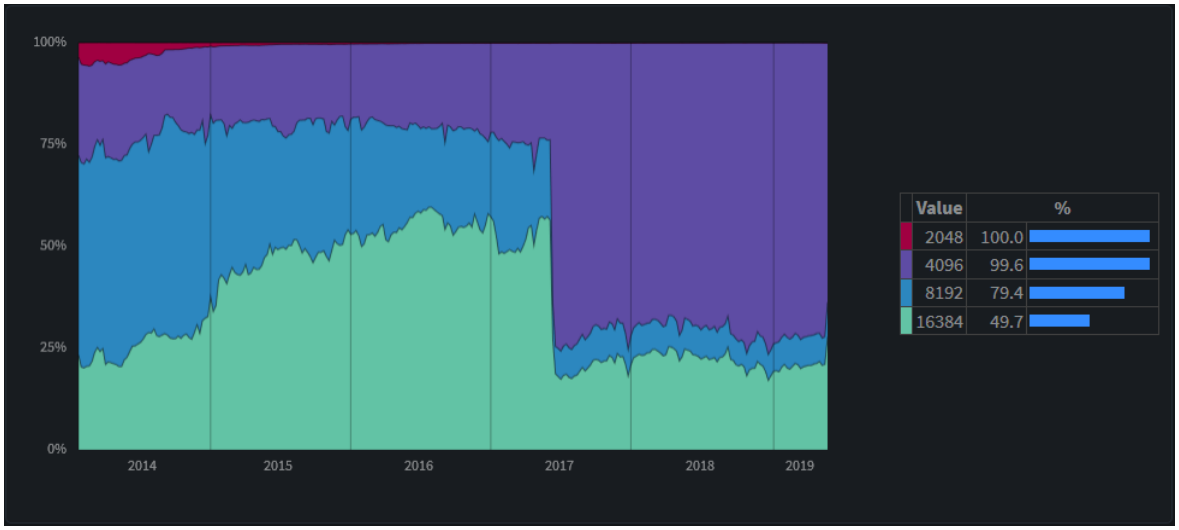


Figure 4 WebGL MAX_TEXTURE_SIZE statistic [51]

4.4 Batching in PixiJS

In PixiJS actual texture data is in base texture and texture is always a region of some base texture [42]. This division fits very well into texture atlas concept. PixiJS rendering starts from a root container from which renderer then iterates through all the container's children in recursive manner. During this travel through container hierarchy renderer generates draw calls that will generate the image user will see.

PixiJS batches only sprites so any other renderable object will be drawn with its own draw call. Sprites on the other hand are batched together with following logic. When renderer finds a sprite and there is no active batch it will start new batch. Then renderer adds sprite's vertex data to batch and base texture to a list. If there is an active batch renderer will check if it can add sprite to the active batch. Renderer will be able to add sprite to active batch if it's not batch breaking. Also, if there is non-sprite renderable objects between sprite, they will break batch. Batch also ends if batch limit or end of container hierarchy is reached. The default batch size limit in PixiJS is 4096 objects [42]. [52]

Sprite is batch breaking if

- It has base texture that is not in batch texture list and batch texture list is full.
- It has different blend mode from batch
- It has mask
- It has filter

4.5 WebGL bound textures limit

From Table 3 we can see how many textures WebGL allows to be bound to GPU simultaneously on test devices (Appendix 1). WebGL will get this limit from GPU drivers. PixiJS also uses internal maximum sprite textures which is four on phones and tablets and thirty-two on other platforms [53]. PixiJS uses this ability to have multiple textures bound to GPU to batch sprites with different base textures. PixiJS uses smaller number of maximum sprite textures and WebGL maximum textures as a limit of how many textures PixiJS will use to batch.

Device	Sony	Honor 7	iPhone se	Dell 7010
Pixi max sprite textures	4	4	4	32
WebGL max textures	16	16	8	16
Pixi max textures	4	4	4	16

Table 3 Texture limits of PixiJS and WebGL

Devices in Table 3 contain three mobile devices and one desktop. The mobile devices are Sony, Honor 7 and iPhone se.

5 JAVASCRIPT

HTML 5 games use JavaScript as their programming language. For this reason, it is good idea to have some understanding of few concepts in JavaScript, that can have significant performance impact.

5.1 Garbage collection

JavaScript allocates memory automatically when needed and uses garbage collection for automatically releasing it. Garbage collection does this by monitors memory usage and determining when some parts of allocated memory is no longer used. This means that developers can not choose when memory is released only to remove references and wait for garbage collector to release it. [54] This does make it easier for developers because there is no memory leaks because developer forgot to release memory [55]. Downside is that some developers can start thinking that they don’t need to think about memory management at all.

Memory allocating and releasing is considered slow. For this reason, it is good to generate as little garbage as possible.

Since JavaScript has no access to garbage collector, it is impossible to give it hints on when to run. Therefore, garbage collector runs when system decides so. This usually happens periodically or when memory usage gets close to a limit. These garbage collector runs can cause unpredictable pauses. [55]

5.1.1 Reference

Garbage collection algorithms all rely on references. The concept of reference in context of garbage collection is when one object has access to other object it has a reference to that object. For this context function and lexical scopes are also objects. [54]

5.1.2 Reference counting

Reference counting is a simple garbage collection algorithm that some older JavaScript engines use. This algorithm changes the problem of is this part of allocated memory used to is there a reference to this part of allocated memory. If there are no references, then that part is considered garbage and can be released. [54]

Code snippet 3 shows an example on how reference counting works. On line 1 an object is allocated and variable ‘a’ references to it. On line 2 variable ‘b’ is set to variable ‘a’, which means variable ‘b’ now also references to the object. On line 3 variable ‘a’ is set to null and is therefore no longer referencing the object. Because variable ‘b’ is still referencing to it, it is not yet released. On line 4 variable ‘b’ is set to null and since it’s last reference to the object it will be released when garbage collector runs.

```
01:  let a = {};
02:  let b = a;
03:  a = null;
04:  b = null;
```

Code snippet 3 Reference counting example

This works on straightforward data structures, but it fails on circular references. Circular reference is when an object references itself through a chain of zero or more other objects. Reference counting algorithm doesn’t release these objects because they still have reference to them even though there is no reference to the object outside of circular reference.

Code snippet 4 shows an example of a circular reference. On lines 1 and 2 two objects are created. On lines 3 and 4 these objects are assigned properties that reference each other. This creates a circular reference. On lines 5 and 6 variable references to objects are removed, but since they still reference each other their reference count is not zero and they are not released.

```
01:    let a = {};
02:    let b = {};
03:    a.x = b;
04:    b.x = a
05:    a = null;
06:    b = null;
```

Code snippet 4 Circular reference example

5.1.3 Mark and sweep

Mark and sweep is a modern garbage collection algorithm. This algorithm changes the problem of is this part of allocated memory used to is this part of allocated memory reachable. If it is not reachable then that part is considered garbage and can be released. According to MDN from year 2012 all modern browsers use mark and sweep algorithm. [54] [56]

In mark and sweep algorithm all local and global variables are called roots and stored in the global object. The algorithm determines if object is reachable by starting from these roots and marking them. Then it marks all objects referenced by these roots. Then it recursively marks objects referenced by previous objects. Objects that have already been marked are not checked again to prevent forever loops and to optimize since there is no need to process same object twice. After the marking comes sweep where all objects that were not marked are released. [54] [56]

5.1.3.1 Optimizations

There have been many implementation optimizations to this algorithm over the years, but there have not been any improvements to algorithm itself [54]. These optimizations include idle-time collection, generational collection and incremental collection. Idle-time collection means that garbage collector tries to run when CPU is idle. [56] This reduces the chances of garbage collector interfering with normal execution of JavaScript.

Generational collection is a method where allocated objects are divided into two sets of objects. One is for new objects and second one is for old objects. this is because most objects

are created and discarded fast. For this reason, generational collection is made to optimize allocation and release of these short lifetime objects. When new objects survive long enough, they are moved to old objects heap. For example, Googles v8 JavaScript engine doesn't use mark and sweep for these newly created objects because it's too slow. instead it uses Cheney's algorithm [57] which is much faster, but since it has large space overhead it is not suitable for large heaps. In Googles v8 JavaScript engine collection of short lifetime objects is called minor garbage collection and collection of old objects is called major garbage collection. [55] [56]

Incremental Collection is a method where marking of objects is done in small increments. this makes it so that there is series of small pauses instead of one large pause, when garbage collector runs. Because object references can change between these incremental markings, it requires some additional processing and bookkeeping. [55] [56]

5.1.3.2 Example

Using same example from Code snippet 4, lines 1 and 2 creates objects a and b. References to these objects are stored in global object as seen in Figure 5.

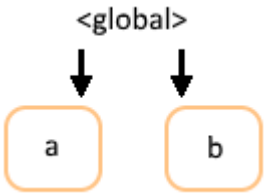


Figure 5 Mark and sweep example part 1

Lines 3 and 4 create a circular reference between objects a and b as seen in Figure 6.

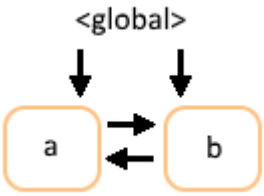


Figure 6 Mark and sweep example part 2

Lines 5 and 6 remove references from global to objects a and b as seen in Figure 7.

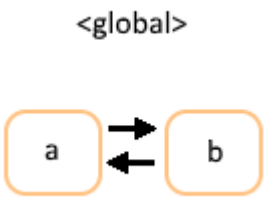


Figure 7 Mark and sweep example part 3

After this when mark and sweep algorithm runs it starts from global and since there is no references to objects a and b it doesn't mark them. Then sweep will release these objects, because they were not marked. As a result, circular references are not a problem with mark and sweep algorithm.

5.2 Object pooling

Object pooling is a method where objects are reused instead of creating new ones and destroying after use. Object are requested from pool and returned to pool after use. With this method saves both releasing memory during garbage collect and memory allocation during object creation. Only thing needed is reinitializing of object properties. [58]

Object pooling can reduce how often garbage collector runs. This is because object pooling reduces the amount of memory allocation and release operations. For example, Googles v8 JavaScript engine triggers minor garbage collection when certain number of memory allocations have been made.

5.3 Entity component system

Entity component system (ECS) is a design pattern in which game object are composed instead of inherited. It consists of three parts entities, components and systems. Entities are usually only an identifier and container for components. Components contain data and mark entity to have some behavior. Systems are where all the code is. They work on entities that have components they require. [59] [60] This allows engine level pooling of components and entities. Some implementations have fixed size pools allowing allocating all needed memory at start.

5.4 JIT

Most modern JavaScript engines use JIT compilers to increase JavaScript performance. JIT compilers profile code and if a piece of code is run often enough just before it is used it will be compiled. Compilation takes more time than simple interpretation, but compiled code will run much faster. If that piece of code is run even more often JIT compiler will try to optimize it. Optimization again takes some time, but optimized code can run much faster. [16] [33] [61]

Because JavaScript is a dynamic language JITs optimizations must make some assumptions. For this reason, JIT compiler must check if these assumptions are still valid before running optimized code. If these assumptions are not valid it has to deoptimize that piece of code. This deoptimized code can later be optimized again with potentially different results. [15] [61]

5.4.1 JIT-unfriendly code

JIT-unfriendly code is code that prevents JIT compilers from doing beneficial optimizations. This is done by using JavaScripts dynamic features. Liang Gong, Michael Pradel, and Koushik Sen in their work on “pinpointing JIT-unfriendly JavaScript code” describe seven JIT-unfriendly code patterns. [16]

These seven JIT-unfriendly patterns are:

- Inconsistent object layouts
- Polymorphic operations
- Binary operations on undefined
- Non-contiguous arrays
- Accessing undefined array elements
- Storing non-numeric values in numeric arrays
- Unnecessary use of generic arrays

In addition to these seven Vincent St-Amour and Shu-yu Guo in their work on “JavaScript optimization coaching” found property location to be JIT-unfriendly [15].

5.4.1.1 Inconsistent object layouts

Objects of same type with properties in different order have inconsistent object layout. This happens for example when object is assigned properties a and b in different order based on some condition. This prevents JIT compiler from creating optimized code where properties are at known offsets. This results in property access from being fast fixed offset to be hash table lookup. [16]

5.4.1.2 Polymorphic operations

Polymorphic operations are pieces of code that apply operation to different data types depending from where it is called. Changing these to multiple operations that are always called with same data type allows JIT compiler to optimize it for that specific data type. [16]

5.4.1.3 Binary operations on undefined

When doing binary operations, such as +, -, *, /, %, |, and & on undefined values is slower than doing it on defined value. This is because JavaScript engine must do type checking and type conversions instead of using specialized code. [16]

5.4.1.4 Non-contiguous arrays

Non-contiguous arrays are arrays where some elements have not been initialized. If these arrays are sparse enough, they are implemented as hash table instead of continuous memory location. Hash table lookup is slower than lookup from continuous memory location. JIT compilers also have specialized code that assume that there are no missing elements in array and have fallback to slower more generalized code. [16]

5.4.1.5 Accessing undefined array elements

Trying to access uninitialized, deleted or out of bounds array element gives undefined. These are slow for similar reasons as non-contiguous arrays. [16]

5.4.1.6 Storing non-numeric values in numeric arrays

JavaScript arrays can contain elements any data type. If it contains only numbers JavaScript engine can use efficient C-like fixed size arrays of integers or doubles. It is a slow operation to change number only array to generalized array. Single type arrays also allow JIT compilers to optimize them. [16]

5.4.1.7 Unnecessary use of generic arrays

JavaScript generic arrays are normal arrays where one can put any kind of data. Since EcmaScript 2015 specification for types arrays were given. Typed arrays allow programmers to tell what specific type of data is in these arrays. These allow JIT compilers to use C-like specialized arrays instead of complex data structures. This leads to better memory representation and removes the need for type checking. JIT-compilers can optimize general arrays into typed arrays, but do not always pick the best type. [16] [62]

5.4.1.8 property location

JavaScript objects property can be in object instance or in prototype. If property can be found in either instance or prototype JIT compiler can’t optimize it to either case. It must use more generalized code that has to check to see if it’s found in instance and fallback to access prototype property [15]

5.4.2 Optimization coaching

Optimization coach is a program that provides recommendations that would allow more optimizations. These are often hard to diagnose performance problems since optimizers fail silently. Optimization coach reads optimizers logs and finds near misses. Near misses are optimizations that were not applied but could be applied if program was changed in some way. Near misses are not applied because of either lack of information or they would change program to behave wrong in some cases. [15]

Optimization coach then creates recommendations from these near misses, that programmer can implement if so decides. These recommendations can require changing program's semantic therefore can't be automatically applied. This helps programmers to create code that is easier for optimizers to optimize. [15]

5.4.3 Typescript

Typescript is nearly identical programming language to JavaScript with only difference being type information. In Typescript all variables can be assigned a type. It then gives errors if function is called with wrong type of variable. [63] [64]

When Typescript is compiled into a JavaScript Typescripts variable typing can help JavaScripts JIT compilers. One benefit is that they don't need to deoptimize because variable type changes from optimized version.

5.5 WebAssembly

WebAssembly is a new language for the web. It is assembly like low level intermediate language that can achieve near native performance. It is abstract syntax tree in binary format. Being intermediate language makes it platform independent and binary format makes it's file size smaller than equivalent text formatted JavaScript. Being low-level language WebAssembly allows ahead of time optimizations, which can run faster than JavaScript. WebAssembly allows compilation target for languages such as C/C++ and rust so they can be run in web. WebAssembly is designed to be able to run with JavaScript allowing programmers to use best parts from both. [65] [66] [67] [68]

Micha Reiser and Luc Bläser were able to get between 100% and 345% speedup on chrome and between 98% and 464% speedup on firefox with WebAssembly than typescript. They used eight computationally heavy test cases. Typescripts few near same performance is because Typescript already allows JIT compilers to do significant optimizations. They expect WebAssembly performance to increase in future because browsers will probably improve WebAssembly support. [65]

6 SIZE

HTML 5 games size consist mostly of three parts. These parts are code, images and audio. Usually code is not very big part of size, but on very small games it can be significant part. Two usual reasons for optimizing HTML 5 game size is loading speed and requirements. These requirements can be for example 2 megabytes for advertisement game. In some cases, bandwidth cost can also be reason for size optimization.

6.1 File count

In addition to file size file count can also have significant impact on loading speed. It is faster to load one big file instead of many small files. This comes from browsers having a limit to how many simultaneous connections it can have. This comes from Hypertext Transfer Protocol (HTTP) 1.1 specification that single client should maintain only two connections to same server. Now a days most browsers allow six simultaneous connections. This means that loading many files only the amount of connections can be loaded at once and others must wait for their turn to load. There is also additional time taken by establishing new connection for each file. [69] [70] [71]

Number of mage and audio files can be reduced with two methods. First method is to embed data into html file. This is done by first transforming image or audio file into data url using base 64 encoding. This data url is then used as a source url in html image or audio tag. Second method is to combine many small files into one big one. Combining multiple images into one big one is called image atlas or sprite sheet. Combining multiple audio files into one long one is called audio sprite. [69] [72]

6.2 Code

Programmers usually program JavaScript in human readable form. and since JavaScript is transferred as is, it contains many unnecessary characters. Minimizers can compress JavaScript by methods such as renaming used variables and functions, removing white

spaces. Code snippet 5 shows a generic function to calculate sum from array of numbers. This code is 140 characters of text. Code snippet 6 shows same code after it has been minimized. it is only 68 characters of text. This is reduction of 51,4% in character count.

```
01:  function sum(numbersArray) {
02:    let count = 0;
03:    for(let i = 0; i < numbersArray.length; ++i) {
04:      count += numbersArray[i];
05:    }
06:    return count;
07:  }
```

Code snippet 5 Sum function

```
01:  function sum(t){let e=0;for(let n=0;n<t.length;++n)e+=t[n];return e}
```

Code snippet 6 Minimized sum function

Another way to reduce size of JavaScript code is to not include unused code. It is possible to manually remove one's own code that is not needed. But if JavaScript is using modules it is not that easy. If modules are used with EcmaScript 2015 syntax it is possible for compilation tools such as WebPack to not include unused code. This is known as tree shaking. [73] [74]

6.3 Image format

Image format effects how image is compressed and/or encoded. This effects image features and size. Two often used image formats are portable network graphics (PNG) and Joint Photographic Experts Group (JPEG).

6.3.1 JPEG

JPEG format was published in year 1992. It is popular lossy format with photographs. It can do gradients well, but it has trouble with images requiring sharpness. It also has good compression. JPEGs problem in HTML 5 games is its lack of support for alpha channel. This

means that it can't have any see through parts in images. [75] [76] This makes JPEG good format for example backgrounds.

6.3.2 PNG

PNG was published in 1996 by W3C. Second edition in year 2003 was made to make PNG into International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) international standard. [77]

PNG uses lossless compression to compress actual image pixels. This allows good quality images with good compression. PNG has five types of images, which are TrueColor with and without alpha, greyscale with and without alpha and indexed palette. Additionally, each image can have between 1 to 16 bits per color. [76] [77]

One of the best ways to reduce image size is to reduce color space. PNG8 is usually good compromise with image quality and size. PNG8 is an PNG image that uses indexed color palette of 256 colors. This means that each pixel is 8-bit index to color palette table. [77] [78] This limited amount of colors means that it is not very good with gradients and other images with lots of colors.

6.3.3 WebP

WebP is relatively new format released in 2010 by Google. It uses predictive coding by predicting pixel value from neighbors and then only saving difference. According to Google WebP is almost 30% smaller than PNG or JPEG at close to same image quality. [76] [79]

According Giaime Ginesu, Maurizio Pintus and Daniele D. Giusto work in 2012 WebP is good with natural images, but is no longer good with high resolution, highly detailed or synthetic images. According to them WebP images had blurry effect. According. They also concluded that it doesn't provide enough improvement to replace JPEG. [80].

WebP has been adopted by many modern browsers. With exception being Apple's Safari browser, that does not support WebP. Also, Firefox version that supports WebP was released

in January 2019. [76] [79] [81] Because of these factors WebP might not be supported widely enough for HTML 5 games yet and fallback images should be provided.

6.3.4 Browser support

HTML 5 specification doesn't define any image formats [31]. This leaves browser developers to choose which image formats to implement. This allows new formats but can take some time for them to be widely adopted. Table 1 shows which version of few popular browsers [82] supports PNG, JPEG and WebP formats.

	Firefox	Chrome	Safari	Edge	IE
PNG	1	1	1	12	5
JPEG	all	all	all	all	all
WebP	65	17	no	18	no

Table 4 Browser version supporting image formats [76]

6.4 Audio

Any decent quality uncompressed audio file can be rather large. Since HTML 5 games usually want to have smaller files, it's good idea to compress audio. It is also usually good idea to convert stereo audio to mono, since it will reduce file size. [83]

6.4.1 WAV

Waveform Audio File Format (WAV) was published in year 1991. Although wav file can contain different data formats [84] [85], it's usually used for raw uncompressed pulse-code modulated data. This makes them to have large file size.

6.4.2 Mp3

Mp3 is an audio format specified by MPEG standard first introduced in year 1991. More specific it's MPEG Audio layer III. It supports wide range of bit rates and sample rates. These allow HTLM 5 game developers to choose audio quality. It uses lossy compression to get even smaller file size. [83]

Mp3 was patented and because of this it was not always supported by all browsers. These patents expired in 2012 in the EU and 2017 in the United States. It is now free for everyone to use. [83]

6.4.3 Ogg vorbis

Ogg vorbis is an open format for wide range of applications. Its format was decided in year 2000. It also supports wide range of bit and sample rates. It uses lossy compression that produces smaller files than mp3 at close to same quality. Although Ogg vorbis is open format it is not supported by all browsers. [83] [86]

6.4.4 Browser support

Just like image tag HTML 5 audio tags does not define audio format [31]. This means that browser developers choose which formats to implement. Table 5 shows which version of browsers support each audio format in HTML 5 audio.tag.

	Firefox	Chrome	Safari	Edge	IE
Wav	3.5	8	4	12	no
Mp3	22	4	4	12	9
Ogg Vorbis	3.5	4	no	17	no

Table 5 Browser version supporting audio formats in HTML 5 audio tag [83] [87] [88]

7 EXPERIMENTS

7.1 Moving sprites in PixiJS rendering test

This test shows how many moving sprites PixiJS can draw. Sprite moving code is very simple as to have minimal impact to the test since goal is to test rendering performance. For this reason, test was done in with and without batching.

7.1.1 Test setup

Tests were done using Code snippet 7 which creates sprites in interleaving draw order. Line 4 was changed to match how many sprites was to be drawn. Canvas size in the test is 512x512 pixels.

```
01:    const cinoTexture = PIXI.loader.resources["images/cino.png"].texture
02:    const sparkTexture = PIXI.loader.resources["images/spark.png"].texture
03:    const sprites = [];
04:    const spriteCount = 1000;
05:
06:    // create sprites in interleaving draw order
07:    for (let i = 0; i < spriteCount/2; ++i){
08:      const sprite = new PIXI.Sprite(cinoTexture)
09:      sprite.direction = 1;
10:      sprite.position.set(412 * Math.random(), 412 * Math.random());
11:      app.stage.addChild(sprite);
12:      sprites.push(sprite);
13:
14:      const sprite2 = new PIXI.Sprite(sparkTexture)
15:      sprite2.direction = 1;
16:      sprite2.position.set(412 * Math.random(), 412 * Math.random());
17:      app.stage.addChild(sprite2);
18:      sprites.push(sprite2);
19:    }
```

Code snippet 7 Moving sprites test sprite creation

For non-batching test Code snippet 8 was added to before creating PixiJS application. This in combination with interleaving draw order makes it so that each sprite is drawn with its own draw call.

```
01:  PIXI.SPRITE_MAX_TEXTURES = 1;
```

Code snippet 8 Set sprite batching texture limit to 1

Code snippet 9 was used to move these sprites with minimal code while still fully remaining inside canvas.

```
01:    for (let i = 0; i < sprites.length; ++i) {
02:      const sprite = sprites[i];
03:      sprite.position.y += sprite.direction;
04:      if (sprite.position.y < 0) {
05:        sprite.direction = 1;
06:      }else if (sprite.position.y > 412) {
07:        sprite.direction = -1;
08:      }
09:    }
```

Code snippet 9 Moving sprites test sprite moving

7.1.2 Measuring

Code snippet 10 Initializes measured times array. It also sets length of the test to 600 frames.

```
01:    const times = [];
02:    for(var i=0; i<600; i++){
03:      times[i]=0;
04:    }
```

Code snippet 10 Moving sprites test time initialization

Code snippet 11 shows how the test frame times were measured. It uses JavaScripts performance.now() function [89] on every frame to save that frames time. Result frame time is then calculated from these times by first calculating differences and then taking average from them.

```
01:    let tick = 0;
02:    let running = true;
03:    app.ticker.add(() => {
04:      if (running) {
05:        times[tick] = performance.now();
06:        tick++;
07:        if (tick >= times.length) {
08:          running = false;
09:        }
10:      }
11:    });
```

Code snippet 11 Moving sprites test time measurement

7.1.3 Result

Table 6 shows resulting frame times when PixiJS was batching all to a single draw call. All test devices were able to draw one thousand sprites with 60 fps, which corresponds to 16.67 millisecond(ms) frame time. Test devices were also able draw four thousand sprites while to maintain 30fps, which corresponds to 33.34ms frame time.

Sprite Count	Sony	Honor 7	iPhone se	Dell 7010
1000	15.15 ms	16.90 ms	16,71 ms	16,73 ms
2000	19.69 ms	17.34 ms	16,67 ms	16,72 ms
3000	25.65 ms	24.06 ms	16,66 ms	16,79 ms
4000	32.96 ms	31,04 ms	16,66 ms	16,74 ms

Table 6 Moving sprites test with batching frame times

Table 7 shows resulting frame times when PixiJS was not batching making every sprite it's own draw call. All test devices were still able to draw one thousand sprites faster than 30 fps. with two thousand sprites Sony and Honor 7 are already significantly slower than with batching. Sony being 93.80 percent and Honor 7 being 195.90 percent slower without batching. One possible explanation to iPhone se anomaly is that it does batching somehow despite receiving individual draw calls from PixiJS.

Sprite Count	Sony	Honor 7	iPhone se	Dell 7010
1000	17.67 ms	22.82 ms	16.69 ms	16.80 ms
2000	38.16 ms	51.31 ms	16.67 ms	20.21 ms
3000	58.14 ms	75.87 ms	16.90 ms	27.69 ms
4000	75.18 ms	98.03 ms	16.67 ms	36.12 ms

Table 7 Moving sprites test without batching frame times

Table 8 shows how much slower test devices were at rendering sprites without batching versus batching all sprites to one draw call. These are frame time differences so for example four thousand sprites on test device Sony had a drop from 30.34 fps to 13.30 fps.

Sprite Count	Sony	Honor 7	iPhone se	Dell 7010
1000	16.63%	35.03%	-0.12%	0.42%
2000	93.80%	195,90%	0%	20.87%
3000	126.67%	215.34%	1.44%	64.92%
4000	128.10%	215.82%	0.06%	115.77%

Table 8 Moving sprites test how much slower without batching is

These results are average result from 600 frames so any spikes in performance would be evened out. There are also many results near 16.67ms, which is because these were mostly capped at 60 fps. This cap comes from PixiJS using requestAnimationFrame function for update loop [90]. According to MDN web docs requestAnimationFrame generally matches display refresh rate, which is usually 60 times in a second [91].

7.2 pooling test

This test compares duration of creating PixiJS sprites and using pool of already created sprites.

7.2.1 Setup

Code snippet 12 shows how non pooled sprites are created. Code snippet 13 shows how pooled sprites are retrieved from pool and then returned to pool.

```
01:  for (let i = 0;  i < 2000;  ++i)  {
02:    const sprite = new PIXI.Sprite(cinoTexture);
03:    sprites.push(sprite);
04:  }
```

Code snippet 12 No pooling new PixiJS sprite creation

```
01:  for (let i = 0; i < 2000; ++i)  {
02:    const sprite = new getSprite();
03:    sprites.push(sprite);
04:  }
05:  for (let i = 0; i < sprites.length; ++i) {
06:    releaseSprite(sprites[i]);
07:  }
```

Code snippet 13 Pooled get and release sprite

Code snippet 14 shows function to retrieve sprite from pool and Code snippet 15 show how it is returned to pool. The pool is JavaScript array with two thousand sprites created beforehand.

```
01:  function getSprite() {
02:    if (spritePool.length > 0) {
03:      const sprite = spritePool.pop();
04:      sprite.position.set(0, 0);
05:      sprite.scale.set(0, 0);
06:      return sprite;
07:    } else {
08:      return new PIXI.Sprite(cinoTexture);
09:    }
10:  }
```

Code snippet 14 getSprite function for pooled test

```
01:  function releaseSprite(sprite) {
02:    spritePool.push(sprite)
03:  }
```

Code snippet 15 releaseSprite for pooled test

7.2.2 Measuring

Duration is measured using JavaScripts performance.now() function before and after Code snippet 12 and Code snippet 13. These were repeated five times and then calculated average from them.

7.2.3 Result

Table 9 shows how that using pooling is significantly faster than always create new sprites. Also, during testing it was noticed that no pooling triggered garbage collection during sprite creation. Pooled test did not trigger garbage collection.

	Sony	Honor 7	iPhone se	Dell 7010
No pooling	51.42ms	55.70ms	22.12ms	11.27ms
Pooling	10.60ms	17.70ms	10.60ms	2.05ms

Table 9 Pooling test results

7.3 References to null test

This test compares garbage collection speed of large linked list with and without setting its references to null.

7.3.1 Setup

Memory is reserved by creating a linked list of 10 million nodes. Code snippet 16 shows how linked list is created. Each node has 1024 characters of data. and reference to next node. Reference to this linked list is kept and ten more linked lists are created to push this first one to old objects memory space. After two seconds reference to the first linked list is set to null. In nulling version just before setting reference to null linked list gone through and set all nodes next references to null.

```
01:  function getLinkedList() {
02:    const first = {
03:      data: <1024 character string>,
04:      next: null
05:    }
06:    let prev = first
07:    for (let i = 0; i < 10000000; i++) {
08:      const node = {
09:        data: <1024 character string>,
10:        next:  null
11:      }
12:      prev.next = node;
13:      prev = node;
14:    }
15:    return first;
16:  }
```

Code snippet 16 Function to create linked list of data.

7.3.2 Measurement

After releasing linked list more memory is reserved. Every 300 milliseconds one million 1024-character string are stored into an array. This is done to trigger major garbage collection that uses mark and sweep algorithm. This major garbage collection was measured using Google chromes developer consoles performance tab. From this data first major garbage collection is searched and its duration is used as measurement. Figure 8 shows an example of this.

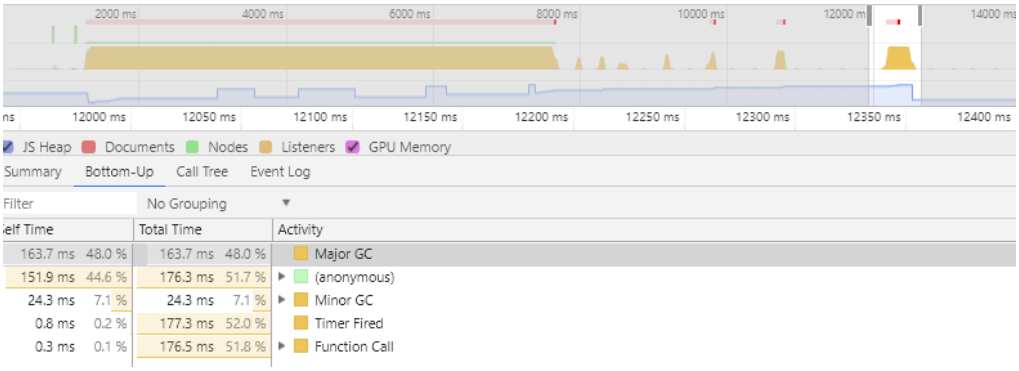


Figure 8 Google Chormes developer consoles performce tab

7.3.3 Result

Table 10 shows average of three test on both with and without nulling linked list references. From this we can see that nulling is 30.2% faster than without nulling. This is significant increase.

	no nulling	nulling
average duration	162.1 ms	113.2 ms

Table 10 Major garbage collection duration with and without nulling linked list references.

7.4 For loops comparison test

This test compares JavaScript engines speed to go through an array in three different styles. These are forward for loop, backwards for loop and foreach function.

7.4.1 Test setup

Test uses an array with one million random numbers. This array is then given to for loop functions. Code snippet 17 show function for forwards for loop. Code snippet 18 show function for backwards for loop. Code snippet 19 show function for foreach loop.

```
01:  function forwardFor(arr) {
02:    for(let i = 0; i < arr.length; ++i)  {
03:      const t = 1 + arr[i];
04:    }
05:  }
```

Code snippet 17 Forward for function

```
01:  function backwardsFor(arr) {
02:    for(let i = arr.length - 1;  i >= 0; --i)  {
03:      const t = 1 + arr[i];
04:    }
05:  }
```

Code snippet 18 Backward for function

```
01:    function foreachFor(arr) {
02:      arr.forEach(elem => {
03:        const t = 1 + arr[i];
04:      })
05:    }
```

Code snippet 19 Foreach function

7.4.2 Measuring

Measuring is done using JavaScripts performance.now() function before and after calling loop function. Difference is then calculated from these to get how long the function took to execute. For warmup each function is called one hundred times before measuring. This is to get JIT to optimize functions. Each test was done five times and average time is calculated as result.

7.4.3 Result

Table 11 shows results in milliseconds. From this we can see that warmup has big effect. Forward for is 73.0% faster, backward for is 58.0% faster and foreach is 9.5% faster than without warmup. This is because warmup causes JIT to optimize function. Foreach is optimized less than others probably because foreach callback function is created each time and not optimized.

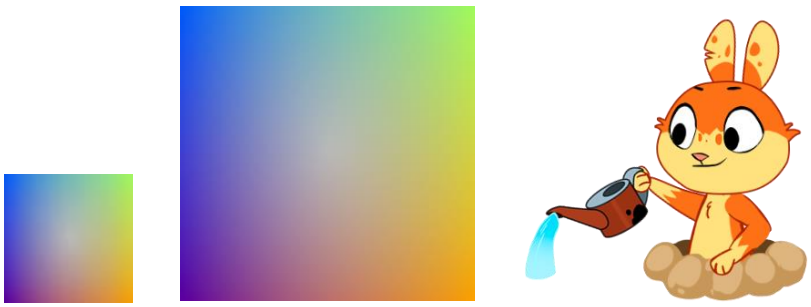
From these results we can see that without warmup backwards for is 6.8% faster than forwards for. With warmup forward for is 30.9% faster than backwards for.

	forward	backward	foreach
no warmup	2.07 ms	1.93 ms	19.08 ms
warmup	0.56 ms	0.81 ms	17.26 ms

Table 11 For loop test results

7.5 Image format comparison test

In this test three images are compressed into PNG8, JPEG and WebP formats. These three images are 64x64 pixels gradient, 256x256 gradient and bunny images. Figure 10 shows these original uncompressed images. Compression is done using Texture packer program. Both JPEG and WebP compression use quality setting of 80.



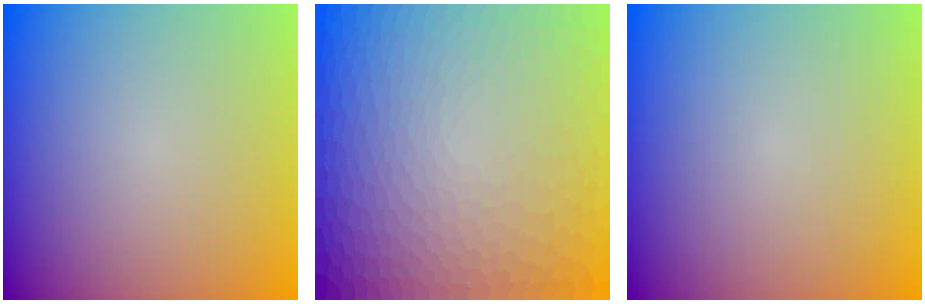
(a) (b) (c)

Figure 9 Image format size source images: (a) 64x64 gradient (b) 256x256 gradient (c) bunny

Figure 10 shows compressed small gradient images. JPEG and WebP are indistinguishable from each other. On other hand PNG8 has some noticeable ridges. These are easier to see in Figure 11 since it is same image but larger.

(a) (b) (c)

Figure 10 Compressed 64x64 gradient (a) jpg (b) png (c) webp



(a) (b) (c)

Figure 11 Compressed 256x256 gradient (a) jpg (b) png (c) webp

Figure 12 shows compressed bunny image. JPEG image has black background since it does not support alpha channel. JPEG image also has some artifacts and fuzziness in watering can. WebP also has this fuzziness but much less. Figure 13 shows zoomed versions of Figure 12 watering cans that shows this fuzziness much.

(a) (b) (c)

Figure 12 Compressed bunny (a) jpg (b) png (c) webp

(a) (b) (c)

Figure 13 Compressed bunny cropped and zoomed to watering can (a) jpg (b) png (c) webp

Table 12 shows image sizes after compression. From it we can see that PNG is not as good with gradient images. We can also see that WebP is significantly smaller that JPEG. All three have similar size with bunny image. This is probably because it has large areas of same color. PNG is good with these large areas of same color whereas JPEG and WebP are not. JPEG also has no alpha channel, so it has less data.

	Original PNG	PNG8	JPEG	WebP
small gradient	4.88KB	2.56KB	951B	266B
large gradient	74.6KB	10.2KB	3.53KB	1.03KB
bunny	50.7KB	15.5KB	15.4KB	15.3KB

Table 12 Image compression sizes

8 ENGINE COMPARISONS

This chapter compares four HTML 5 game engines. They are compared by usage, packet size and rendering performance. These engines are PixiJS v4 using version 4.8.7, PixiJS v5 using version 5.1.5, Phaser 3 using version 3.19.0 and unity project tiny using version 0.14.5-preview.

8.1 Usage

Both PixiJS v4 and v5 are used in similar way. They are JavaScript libraries. First in code pixi application is created which creates all necessary parts to render. It also creates container called stage which in default configuration is automatically rendered. Containers, sprites and other render able objects are then added to stage to be rendered. It is possible to add own update function to pixi applications ticker.

Phaser 3 is also JavaScript library. First in cade phaser game object is created with configurations. This creates all necessary parts to render. In configuration a scene or array of scenes is given. Game then start from first scene. Each scene has preload, create and update functions. Preload is called only once per scene. Create is called each time game changes scene to this scene. Update is called on every tick for active scene. Images and other game objects are then added to scene which will then be rendered.

Unity project tiny is used from unity editor by first installing project tiny package. Project tiny uses ECS. Entities and entity groups are made in editor. Components are added to these entities. Also new components can be created in editor. Project tiny comes with some build in core systems, but new systems can be created. These systems are where all code is. Each system has reference to world which contains all entities. Systems have update function which is called on every tick. In update system can query world for all entities with specific components and then do something with them.

8.2 size

Table 13 shows engine sizes. From it we can see that PixiJS v5 is smaller than PixiJS v4. One possible explanation for this is that version 5 no longer has support for canvas 2D. It therefore only supports WebGL. Both Phaser 3 and project tiny are almost same size. They are also much bigger than either PixiJS. The difference in size comes from Phaser 3 and project tiny being full game engines whereas PixiJS is not. These differences include for example physics, tween and bone animation, such as spine, libraries. These do exist for PixiJS as either own separate module or third-party library.

PixiJS v4	PixiJS v5	Phaser 3	Project tiny
428KB	347KB	910KB	912KB

Table 13 Engine sizes

8.3 Rendering performance

Moving sprites test was done on all four engines. Table 14 shows result for PixiJS v4. Table 15 shows result for PixiJS v5. Table 16 shows results for Phaser 3. Table 17 shows result for project tiny. From these we can see that PixiJS v5 is slightly slower PixiJS v4. We can also see that Phaser 3 is between PixiJS v4 and v5. Also project tiny is slightly slower than Phaser 3. iPhone se was able to maintain 60fps on all test. Dell desktop was able to maintain 60 fps on almost all test. Exception being Phaser 3 with 3000 and 4000 sprites.

Sprite Count	Sony	Honor 7	iPhone se	Dell 7010
1000	15.15 ms	16.9 ms	16.71 ms	16.73 ms
2000	19.69 ms	17.34 ms	16.67 ms	16.72 ms
3000	25.65 ms	24.04 ms	16.66 ms	16.79 ms
4000	32.96 ms	31.04 ms	16.66 ms	16.74 ms

Table 14 PixiJS V4 rendering performance

Sprite Count	Sony	Honor 7	iPhone se	Dell 7010
1000	15.20 ms	16.97 ms	16.72 ms	16.99 ms
2000	19.21 ms	19.63 ms	16.67 ms	16.72 ms
3000	27.28 ms	29.10 ms	16.68 ms	16.73 ms
4000	36.09 ms	45.04 ms	16.67 ms	16.78 ms

Table 15 PixiJS v5 rendering performance

Sprite Count	Sony	Honor 7	iPhone se	Dell 7010
1000	15.16 ms	16.85 ms	16.69 ms	16.71 ms
2000	19.09 ms	17.11 ms	16.69 ms	16.71 ms
3000	26.89 ms	21.93 ms	16.74 ms	25.20 ms
4000	35.47 ms	39.59 ms	16.75 ms	35.88 ms

Table 16 Phaser 3 rendering performance

Sprite Count	Sony	Honor 7	iPhone se	Dell 7010
1000	15.15 ms	16.85 ms	16.83 ms	16.70 ms
2000	21.42 ms	18.03 ms	16.77 ms	16.68 ms
3000	29.55 ms	24.32 ms	16.76 ms	16.68 ms
4000	36.06 ms	30.94 ms	16.85 ms	16.75 ms

Table 17 Project tiny rendering performance

9 RESULTS

This study found batching have effect on all test devices except iPhone se. Sony is from 16.63% to 128.10% slower without batching. Honor 7 is from 35.03% to 215.82% slower without batching. Dell 7010 is from 0.42 to 115.77% slower without batching After reaching high enough sprite count batching typically halves execution time. This is more noticeable on Honor 7 than Sony, because it has faster CPU with slightly slower GPU. This is because batching is done on CPU and it enables better utilization of GPU

This study found object pooling to have effect on all test devices. All test devices were faster to retrieve objects from pool than creating new ones. Sony is 79.39%, Honor 7 is 68.22%, iPhone se is 52.08% and Dell 7010 is 81.81% faster.

Setting references to null was found to have effect on garbage collection. It made garbage collection 30.17% faster on Dell 7010.

Different ways to do a for loop was found to have effect. Without warmup backward for is 7.25% faster than forward for and 888.60% faster than foreach. Forward for is 821.74% faster than foreach. With warmup forward for is 44.64% faster than backwards for and 2982.14% faster than foreach. Backward for is 2030.84% faster than foreach.

Different image formats were found to have effect on file size. On small gradient test image JPEG is 65.23% smaller than PNG8 and WebP is 89.84% smaller than PNG8. On large gradient test image JPEG is 65.39% smaller than PNG8 and WebP is 89.90% smaller than PNG8. On bunny test image JPEG is 0.65% smaller than PNG8 and WebP is 1.29% smaller than PNG8.

Different game engines were found to have different performances. With Sony test device with 1000 sprites PixiJS v4 is shared fastest with project tiny. PixiJS is 0.33% slower than PixiJS v4 and Phaser 3 is 0.07% slower than PixiJS v4. With 2000 sprites Phaser 3 is fastest. PixiJS v4 is 3.14% slower than Phaser 3, PixiJS v5 is 0.63% slower than Phaser 3 and project tiny is 12.21% slower than Phaser 3. With 3000 sprites PixiJS v4 is fastest. PixiJS v5 is

6.25% slower than PixiJS v4, Phaser 3 is 4.83% slower than PixiJS v4 and project tiny is 15.20% slower than PixiJS v4. with 4000 sprites PixiJS v4 is fastest. PixiJS v5 is 9.50% slower than PixiJS v4, Phaser 3 is 7.62% slower than PixiJS v4 and project tiny is 9.41% slower than PixiJS v4. Overall with Sony PixiJS v4 was fastest.

With Honor 7 test device with 1000 sprites Phaser 3 is shared fastest with project tiny. PixiJS v4 is 0.30% slower than Phaser 3 and PixiJS v5 is 0.71% slower than Phaser 3. With 2000 sprites Phaser 3 is fastest. PixiJS v4 is 1.34% slower than Phaser 3, PixiJS v5 is 14.73% slower than Phaser 3 and project tiny is 5.38% slower than Phaser 3. With 3000 sprites Phaser 3 is fastest. PixiJS v4 is 9.62% slower than Phaser 3, PixiJS v5 is 32.69% slower than Phaser 3 and project tiny is 10.90% slower than Phaser 3. With 4000 sprites project tiny is fastest. PixiJS v4 is 0.32% slower than project tiny, PixiJS v5 is 45.57% slower than project tiny and Phaser 3 is 27.96 % slower than project tiny. Overall with Honor 7 PixiJS v4 was fastest

With iPhone se test device all game engines were within 1.14%. With Dell 7010 test device all game engines were within 0.66% except Phaser 3. With 3000 sprites Phaser 3 is 51.08% slower than project tiny, which is fastest. With 4000 sprites Phaser 3 is 114.34% slower than PixiJS v4, which is fastest.

10 DELIBERATION AND FUTURE

Batching has positive effect because GPUs are very good at parallel processing. With larger batches GPU can process it more effectively. Also, CPUs are usually much slower than GPUs and with small batches GPU must wait for CPU to send more data to process.

Garbage collection can be slow and unpredictable. Also, programmers can't directly affect garbage collector. Therefore, setting references to null and object pooling were found to be positive things. Object pooling is also faster because it doesn't have to reserve memory.

JIT-unfriendly code should be avoided. This is because most of JavaScripts speed comes from JIT compilers optimizations and JIT-unfriendly code prevents these optimizations. Different for loops are also optimized differently. Especially foreach which doesn't optimize much. Probably because it always used new function and was therefore not optimized.

File count was found to have negative effect. This is because HTTP specification limits number of simultaneous connections. Therefore, only a number of files can be downloaded at one time and others have to wait for their turn.

JPEG and WebP are better at gradient colors since they encode cosine waves which are then compressed. WebP is smaller with same quality as JPEG but is not widely adopted. PNG on other hand encodes bitmap which is then compressed. This makes PNG better with continuous colors and sharp edges. With these qualities' JPEG is often better choice with large background images and PNG with other game images.

With engine comparisons PixiJS v4 was found to be fastest at drawing sprites. PixiJS v5 can still be better because it can batch more than just sprites. Phaser 3 was also faster than PixiJS v5 but is larger packet size. Therefore, if size is not big limiter and a full game engine is preferred then Phaser 3 might be best. Unitys project tiny is promising but requires different way of thinking, because it uses ECS.

3D rendering, meshes and shader performance is recommended for future study.

11 CONCLUSION

This study found several techniques to optimize HTML 5 games. Batching was found to be significantly better than without batching on rendering performance. Although with few enough sprites batching becomes not significant Pooling was found to have significant performance boost on object creation. Setting references to null was found to make garbage collection faster. JIT-unfriendly code was found to be less performant than JIT-friendly code. Foreach was found to be significantly slower than forward and backward for.

Combining assets into fewer but bigger files were found to improve load times and batching. Image and audio compression were found to significantly reduce file size.

Of the four compared game engines PixiJS v4 was found to be fastest at rendering sprites, while PixiJS v5 produced smallest package size. Phaser 3 and project tiny both have more features than either PixiJS version, but they produce significantly larger package size.

REFERENCES

[1] Mozilla, "HTML: Hypertext Markup Language," 23 july 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML. [Accessed 1 August 2019].

[2] W3C, "A vocabulary and associated APIs for HTML and XHTML," 22 1 2008. [Online]. Available: https://www.w3.org/TR/2008/WD-html5-20080122/. [Accessed 2 4 2019].

[3] W3C, "Open Web Platform Milestone Achieved with HTML5 Recommendation," 28 10 2014. [Online]. Available: https://www.w3.org/2014/10/html5-rec.html.en. [Accessed 2 4 2019].

[4] KHRONOS GROUP, "Khronos Releases Final WebGL 1.0 Specification," 3 3 2011. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification. [Accessed 2 4 2019].

[5] KHRONOS GROUP, "WebGL Specification," 27 31 2014. [Online]. Available: https://www.khronos.org/registry/webgl/specs/1.0.3/. [Accessed 2 4 2019].

[6] P. Cozzi and C. Riccio, "WebGl for OpenGL Developers," in *OpenGL Insights*, CRC Press, 2012, p. 27.

[7] C. Grunewald, "Instant Games," 30 11 2016. [Online]. Available: https://developers.facebook.com/blog/post/2016/11/30/instant-games-closed-beta/. [Accessed 31 3 2019].

[8] M. Weingert, "Instant Games Platform Now Open for All Developers," 14 3 2018. [Online]. Available: https://developers.facebook.com/blog/post/2018/03/14/instant-games-platform-open. [Accessed 31 3 2019].

[9] We Are Social, & Hootsuite, & DataReportal, "Device usage of Facebook users worldwide as of January 2019," 9 August 2019. [Online]. Available: https://www.statista.com/statistics/377808/distribution-of-facebook-users-by-device/. [Accessed 15 August 2019].

[10] K. Smith, "53 Incredible Facebook Statistics and Facts," 1 June 2019. [Online]. Available: https://www.brandwatch.com/blog/facebook-statistics/. [Accessed 15 August 2019].

[11] We Are Social, Hootsuite, DataReportal., "Most popular global mobile messenger apps as of July 2019, based on number of monthly active users," 9 August 2019. [Online]. Available: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/. [Accessed 15 August 2019].

[12] AdColony, "Fall 2017 App Install Marketing Survey," 2017. [Online]. Available: https://www.adcolony.com/reports/fall-2017-app-install-marketing-survey/. [Accessed 2 4 2019].

[13] Marketing charts, "Top App Publishers Are Excited About Playable Ads in 2018," 18 12 2017. [Online]. Available: https://www.marketingcharts.com/digital/mobile-phone-81668. [Accessed 2 4 2019].

[14] . M. Trapp and J. Döllner, "Geometry Batching Using Texture-arrays," in *Proceedings of the 10th International Conference on Computer Graphics Theory and Applications*, Berlin, Germany, 2015.

[15] V. St-Amour and S.-y. Guo, "Optimization Coaching for JavaScript," in *29th European Conference on Object-Oriented Programming (ECOOP'15).*, Prague, Czech Republic, 2015.

[16] L. Gong, M. Pradel and K. Sen, "JITProf: pinpointing JIT-unfriendly JavaScript code," in *ESEC/FSE 2015 Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* , Bergamo, Italy, 2015.

[17] F. F.-H. Nah, "A study on tolerable waiting time: how long are," *Behaviour & Information Technology*, vol. 23, p. 153–163, 2004.

[18] w3school, "HTML Introduction," [Online]. Available: https://www.w3schools.com/html/html_intro.asp. [Accessed 1 August 2019].

[19] T. Berners-Lee, "Information Management: A Proposal," March 1989. [Online]. Available: https://www.w3.org/History/1989/proposal.html. [Accessed 1 August 2019].

[20] CERN, "HTML Tags," [Online]. Available: http://info.cern.ch/hypertext/WWW/MarkUp/Tags.html. [Accessed 1 August 2019].

[21] T. Berners-Lee, "Re: status. Re: X11 BROWSER for WWW," 29 October 1991. [Online]. Available: http://lists.w3.org/Archives/Public/www-talk/1991SepOct/0003.html. [Accessed 1 August 2019].

[22] T. Berners-Lee and D. Connolly, "Hypertext Markup Language (HTML)," 1993.

[23] T. Berners-Lee and D. Connolly, "rfc1866 Hypertext Markup Language - 2.0," 1995.

[24] E. W. Sink, "HTML WG," 12 September 1996. [Online]. Available: https://www.w3.org/MarkUp/HTML-WG/. [Accessed 1 August 2019].

[25] D. Raggett, "HTML 3.2 Reference Specification," 14 January 1997. [Online]. Available: https://www.w3.org/TR/2018/SPSD-html32-20180315/. [Accessed 1 August 2019].

[26] D. Raggett, A. L. Hors and I. Jacobs, "HTML 4.0 Specification," 18 December 1997. [Online]. Available: https://www.w3.org/TR/REC-html40-971218/. [Accessed 1 August 2019].

[27] D. Raggett, A. L. Hors and I. Jacobs, "HTML 4.01 Specification," 24 December 1999. [Online]. Available: https://www.w3.org/TR/html401/. [Accessed 1 August 2019].

[28] S. Pemberton and e. al., "XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)," 1 August 2002. [Online]. Available: https://www.w3.org/TR/xhtml1/#xhtml. [Accessed 1 August 2019].

[29] WHATWG, "WHATWG - FAQ," 2018. [Online]. Available: https://whatwg.org/faq. [Accessed 1 August 2019].

[30] I. Hickson and A. Hyatt, "HTML 5," 22 January 2008. [Online]. Available: https://www.w3.org/TR/2008/WD-html5-20080122/. [Accessed 1 August 2019].

[31] "HTML 5," 28 October 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-html5-20141028/. [Accessed 1 August 2019].

[32] J. Jaffe, "W3C and WHATWG to work together to advance the open Web platform," 28 May 2019. [Online]. Available:

https://www.w3.org/blog/2019/05/w3c-and-whatwg-to-work-together-to-advance-the-open-web-platform/. [Accessed 1 August 2019].

[33] Mozilla, "What is JavaScript?," 30 july 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript. [Accessed 2 August 2019].

[34] Mozilla, "JavaScript data types and data structures," 18 july 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures. [Accessed 2 August 2019].

[35] Mozilla, "Object prototypes," 23 April 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes. [Accessed 3 August 2019].

[36] Netscape Communications Corporation, "NETSCAPE AND SUN ANNOUNCE JAVASCRIPT, THE OPEN, CROSS-PLATFORM OBJECT SCRIPTING LANGUAGE FOR ENTERPRISE NETWORKS AND THE INTERNET," 4 December 1995. [Online]. Available: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html. [Accessed 3 August 2019].

[37] Ecma International, "ECMA-262, 10th edition," june 2019. [Online]. Available: https://www.ecma-international.org/ecma-262/10.0/index.html. [Accessed 3 August 2019].

[38] KHRONOS GROUP, "WebGL Overview," [Online]. Available: https://www.khronos.org/webgl/. [Accessed 4 August 2019].

[39] V. Vukićević , "Canvas 3D: GL power, web-style," 26 November 2007. [Online]. Available: https://web.archive.org/web/20110717224855/http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/. [Accessed 4 August 2019].

[40] T. Johansson, "Taking the canvas to another dimension," 13 November 2007. [Online]. Available: https://web.archive.org/web/20071117170113/http://my.opera.com/timjoh/blog/2007/11/13/taking-the-canvas-to-another-dimension. [Accessed 4 August 2019].

[41] KHRONOS GROUP, "Khronos Details WebGL Initiative to Bring Hardware-Accelerated 3D Graphics to the Internet," 4 August 2009. [Online]. Available: https://www.khronos.org/news/press/khronos-webgl-initiative-hardware-accelerated-3d-graphics-internet. [Accessed 4 August 2019].

[42] GoodBoy Digital, "PixiJS API Documentation," [Online]. Available: http://PixiJS.download/v4.8.7/docs/index.html. [Accessed 22 4 2019].

[43] Github, "PixiJS releases," 3 August 2019. [Online]. Available: https://github.com/PixiJS/pixi.js/releases. [Accessed 7 August 2019].

[44] J. Walker, "PixiJS v4 Arrives," 1 April 2016. [Online]. Available: https://medium.com/goodboy-digital/PixiJS-v4-arrives-a5db2f244442. [Accessed 7 August 2019].

[45] M. Groves, "PixiJS v5 lands," 11 April 2019. [Online]. Available: https://medium.com/goodboy-digital/PixiJS-v5-lands-5e112d84e510. [Accessed 7 August 2019].

[46] R. Nyman, "The concepts of WebGL," 22 4 2013. [Online]. Available: https://hacks.mozilla.org/2013/04/the-concepts-of-webgl/. [Accessed 17 4 2019].

[47] E. Angel, "Teaching Computer Graphics Starting with Shader-Based OpenGL," in *OpenGL Insights,* P. Cozzi and C. Riccio, Eds., CRC Press, 2012, pp. 3-8.

[48] MDN Web Docs, "WebGL best practices," 23 3 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_best_practices. [Accessed 10 4 2019].

[49] S. Itterheim and A. Lw, "Table 6-1. The Technical Specifications of iOS Devices," in *Learn cocos2d 2,* Apress, 2012, p. 160.

[50] gsmarena, "Apple iPhone 4s," [Online]. Available: https://www.gsmarena.com/apple_iphone_4s-4212.php. [Accessed 3 june 2019].

[51] F. Bösch, "MAX_TEXTURE_SIZE," [Online]. Available: https://webglstats.com/webgl/parameter/MAX_TEXTURE_SIZE?platforms=0000ffffcfbfaaff01. [Accessed 3 june 2019].

[52] R. Davey, "Multi Texture Batching," 25 11 2016. [Online]. Available: https://phaser.io/tutorials/advanced-rendering-tutorial/part2. [Accessed 22 4 2019].

[53] Goodboy Digital, "PixiJS v4.8.7 source code," 22 March 2019. [Online]. Available: https://PixiJS.download/v4.8.7/pixi.js. [Accessed 14 September 2019].

[54] Mozilla, "Memory Management," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management. [Accessed 7 june 2019].

[55] J. Conrod, "A tour of V8: Garbage Collection," 25 January 2014. [Online]. Available: http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection. [Accessed 7 june 2019].

[56] I. Kantor and et al, "Garbage collection," 1 june 2019. [Online]. Available: https://javascript.info/garbage-collection. [Accessed 11 june 2019].

[57] C. J. Cheney, "A Nonrecursive List Compacting Algorithm," *Commun. ACM,* vol. 13, no. 11, pp. 677--678, 1970.

[58] A. Jolin, "Can J2EE manage cache and pool memory?," *Dr. Dobb's Journal,* vol. 26(10), pp. 117-120, 2001.

[59] A. Martin, "Entity Systems are the future of MMOG development – Part 2," 11 November 2007. [Online]. Available: http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/. [Accessed 16 August 2019].

[60] D. Wiebusch and M. E. Latoschik , "Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems," in *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2015.

[61] L. Clark, "A crash course in just-in-time (JIT) compilers," 28 February 2017. [Online]. Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/. [Accessed 17 August 2019].

[62] Ecma international, "Standard ECMA-262 6th Edition," June 2015. [Online]. Available: http://www.ecma-international.org/ecma-262/6.0/. [Accessed 17 August 2019].

[63] Microsoft, "TypeScript in 5 minutes," [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. [Accessed 17 August 2019].

[64] Microsoft, "Migrating from JavaScript," [Online]. Available: https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html. [Accessed 17 August 2019].

[65] R. Micha and L. Bläser, "Accelerate JavaScript Applications by Cross-compiling to WebAssembly," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, Vancouver, BC, Canada, 2017.

[66] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai and J. Bastien, "Bringing the Web Up to Speed with WebAssembly," *SIGPLAN Not.*, vol. 52, no. 6, pp. 185-200, 2017.

[67] E. Elliot, "What is WebAssembly?," 19 June 2015. [Online]. Available: https://medium.com/javascript-scene/what-is-webassembly-the-dawn-of-a-new-era-61256ec5a8f6. [Accessed 18 August 2019].

[68] Mozilla, "WebAssembly," 18 March 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly. [Accessed 18 August 2019].

[69] J. Bryant and M. Jones, "Performance Guidelines," in *Pro HTML5 Performance*, Apress, 2012, pp. 21-36.

[70] R. Fielding, U. Irvine, J. Gettys, Compaq/W3C, J. Mogul, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee and W3C/MIT, "rfc2616 Hypertext Transfer Protocol -- HTTP/1.1," 1999.

[71] Push Technology, "Browser connection limitations," 25 April 2019. [Online]. Available: https://docs.pushtechnology.com/cloud/latest/manual/html/designguide/solution/support/connection_limitations.html. [Accessed 22 August 2019].

[72] P. Sexton, "Base64 encoding images," 25 June 2015. [Online]. Available: https://varvy.com/pagespeed/base64-images.html. [Accessed 22 August 2019].

[73] WebPack, "Tree Shaking," [Online]. Available: https://webpack.js.org/guides/tree-shaking/. [Accessed 22 August 2019].

[74] A. Bachuk, "What is tree shaking?," 22 November 2017. [Online]. Available: https://medium.com/@netxm/what-is-tree-shaking-de7c6be5cadd. [Accessed 22 August 2019].

[75] G. K. Wallace, "The JPEG Still Picture Compression Standard," *IEEE Transactions on Consumer Electronics*, vol. 38, pp. xviii-xxxiv, 1992.

[76] Mozilla, "Image file type and format guide," 14 July 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Image_types. [Accessed 23 August 2019].

[77] W3C, "Portable Network Graphics (PNG) Specification (Second Edition)," 10 11 2003. [Online]. Available: https://www.w3.org/TR/2003/REC-PNG-20031110/. [Accessed 11 04 2019].

[78] P. Hansen, "png-8-24-32-what," 4 2 2011. [Online]. Available: http://www.patrickhansen.com/2011/02/04/png-8-24-32-what/. [Accessed 29 August 2019].

[79] Google, "A new image format for the Web," 11 February 2019. [Online]. Available: https://developers.google.com/speed/webp/. [Accessed 29 August 2019].

[80] G. Ginesu, M. Pintus and D. D. Giusto, "Objective assessment of the WebP image coding algorithm," *Signal Processing: Image Communication,* vol. 27, no. 8, pp. 867-874, 2012.

[81] Mozilla, "Firefox Release Notes," 29 January 2019. [Online]. Available: https://www.mozilla.org/en-US/firefox/65.0/releasenotes/. [Accessed 29 August 2019].

[82] W3Counter, "Browser & Platform Market Share July 2019," july 2019. [Online]. Available: https://www.w3counter.com/globalstats.php?year=2019&month=7. [Accessed 29 August 2019].

[83] Mozilla, "Web audio codec guide," 26 August 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Audio_codecs. [Accessed 30 August 2019].

[84] P. Kabal, "WAVE File Specifications," 2 May 2017. [Online]. Available: http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html. [Accessed 30 August 2019].

[85] E. Fleischman, "rfc 2361:WAVE and AVI Codec Registries," 1998.

[86] xiph.org, "Vorbis audio compression," 2016. [Online]. Available: https://www.xiph.org/vorbis/. [Accessed 5 September 2019].

[87] A. Deveria and community, "Wav audio format," 29 july 2019. [Online]. Available: https://caniuse.com/#feat=wav. [Accessed 30 August 2019].

[88] A. Deveria and community, "MP3 audio format," 29 July 2019. [Online]. Available: https://caniuse.com/#feat=mp3. [Accessed 30 August 2019].

[89] Mozilla, "performance.now()," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now. [Accessed 26 May 2019].

[90] GoodBoy Digital, "PixiJS Ticker Sourcecode," 22 March 2019. [Online]. Available: PixiJS.download/v4.8.7/docs/core_ticker_Ticker.js.html. [Accessed 26 may 2019].

[91] Mozilla, "window.requestAnimationFrame()," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. [Accessed 3 june 2019].

[92] M. Reiser and L. Bläser, "Accelerate JavaScript Applications by Cross-Compiling to WebAssembly," in *VMIL 2017 Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, Vancouver, BC, Canada, 2017.

[93] M. Selakovic and M. Pradel, "Performance Issues and Optimizations in JavaScript:," in *ICSE '16 Proceedings of the 38th International Conference on Software Engineering*, Austin, Texas, 2016.

APPENDIX 1. Test devices

	Sony Xperia XA	Honor 7	iPhone se 64GB	Dell 7010
model	F3111	PLK-L01	MLXQ2KS/A	7010
operating system	Android 7.0	Android 6.0	iOS 11.0	Windows 10
CPU	4x 2.0GHz 4x1.0GHz	4x 2.2GHz 4x1.5GHz	1.80GHz dual core	i7-3770 3.4GHz
GPU	GPU Mali-T860MP2	Mali-T628 MP4	PowerVR GT7600	AMD Radeon HD 7570