Lappeenranta University of Technology
School of Industrial Engineering and Management
Degree Program in Computer Science

Rasmus Halsas

Comparing HTML5 game engines used in game development

Examiners: Associate Professor Uolevi Nikula
D.Sc. (Tech.) Jussi Kasurinen

Supervisors: D.Sc. (Tech.) Jussi Kasurinen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Tuotantotalouden tiedekunta
Tietotekniikan koulutusohjelma

Rasmus Halsas

Comparison of HTML5 game engines used in game development

Diplomityö

2017

72 sivua, 18 kuvaa ja 2 taulukkoa
Työn tarkastajat: Dosentti Uolevi Nikula
Tutkijatohtori Jussi Kasurinen

Supervisors: Tutkijatohtori Jussi Kasurinen

Hakusanat: HTML5, pelikehitys, pelimoottori, canvas, endless runner
Keywords: HTML5, game development, game engine, canvas, endless runner

Pelikehityksessä käytetään usein työkaluja auttamaan työkuorman kanssa. Pelimoottorit ovat yksi suosituimmista työkaluista ja tämän työn tavoitteena onkin tutkia miten työssä käytetyt pelimoottorit eroavat toisistaan ja mitkä ovat ne mahdolliset hyödyt, joita pelimoottoreita käyttämällä saavutetaan. Lopuksi työssä selvitetään, milloin HTML5 voidaan pitää sopivana työkaluna pelikehitykselle. Pelimoottoreiden hyödyt käyvät tutkimuksessa selväksi ja tuloksia pelimoottorien eroista vertaillaan keskenään. Peli toteutetaan neljällä erilaisella lähestymistavalla. Lopuksi pelejä vertaillaan ja tuloksia analysoimalla yritetään vastata tutkimuskysymyksiin.

ABSTRACT

Lappeenranta University of Technology
School of Industrial Engineering and Management
Degree Program in Computer Science

Rasmus Halsas

Comparison of HTML5 game engines used in game development

Master's Thesis

2017

72 pages, 18 figures and 2 tables

Examiners: Associate Professor Uolevi Nikula
D. Sc. (Tech.) Jussi Kasurinen

Supervisors: D. Sc. (Tech.) Jussi Kasurinen

Keywords: HTML5, game development, game engine, canvas, endless runner

In game development, tools are often used to help with the workload. Game engines are one of the popular choices that are used and the goal of this thesis is to study how the game engines used in the thesis differ from each other and what are the potential benefits of using them. Finally, an attempt is made to see if HTML5 is a suitable tool for game development. The benefits of the game engines are going to be clear and the results between different game engines are compared and analyzed. A game is implemented using four different approaches. The games are compared against each other and by analyzing the results there is an attempt to answer the research questions.

ACKNOWLEDGEMENTS

This Master's thesis has been done during the years 2016 – 2017 at the Lappeenranta University of Technology.

I have studied since 2010 and I have met a lot of people that I will remain friends with for the rest of my life. While these friendships weren't always beneficial for studying it was still worth it.

Firstly, I want to thank my supervisor Jussi Kasurinen for guiding me through this ordeal and giving me support to complete the thesis. I also want to thank my employer for flexibility.

The greatest support for my writing has been my fianceé Kaisa. She has helped me whenever I had problems with motivation even though she had her own thesis to write as well.

Lastly I want to thank my parents for helping and supporting in my studies. They always believed that I would do my best. My mother is the sole reason I got interested in studying in the first place. Father would always help me with everyday problems I faced when moving to a new city with new challenges. Without their support I would have had a much harder time while studying.

Rasmus Halsas
31.1.2017

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	**Application Programming Interface**
Canvas	**Method of generating fast, dynamic graphics using JavaScript**
CSS	**Cascading Style Sheets**
DOM	**Document Object Model**
FPS	**Frames per second**
HTML	**Hypertext Markup Language**
HTML5	**Hypertext Markup Language 5**
JSON	**JavaScript Object Notation**
MVC	**Model-View-Controller design pattern**
OS	**Operating System**
RAM	**Random Access Memory**
WebGL	**Web Graphics Library**
WebAudio	**JavaScript API for processing and synthesizing audio**
WebStorage	**JavaScript API for storing data locally**
WWW	**World Wide Web**

1 Introduction

In game development there are several different factors that affect the process of creating a product. Performance, development speed and portability have to be considered when choosing the tools for development. The current popular platforms such as iOS, Android, and Windows Phone have their specific environments, which reduce the portability. Web technologies can produce a platform-independent solution. (Santanchè et al., 2013, p. 14) In this study the performance, development speed, and portability are evaluated.

1.1. Background

The author of this study works in the game industry and is constantly using tools to assist in the development process. Thus it would be interesting and useful for his work to find out the concrete benefits of using tools in game development. This is also the main goal of this study. Game development used to have relatively small teams. The teams have grown and the games' complexity has increased with the teams (Blow, 2004, p. 29). Companies need suitable tools to keep the working environment feasible and the workflow doable. Many companies use external tools to build games instead of doing everything from scratch. Tools called "game engines" simplify this process by providing a toolset for control, interaction and graphical manipulation (Anderson et al., 2008).

A major part of the development can be automated with tools. An example for this automation could be following: images are packed into a sheet of sprites so that the frameworks can read them and create objects out of them. Using and manipulating 'sprites' is a popular way of approaching 2-D game development. Libraries and tools allow the user to automate most of the mundane tasks and focus on content creation and development. The combination of libraries that help to build games can be described as 'engine'. Weeks describes the game engine as "central" to his project as it controls the animation, interaction, and movement of the object. (Weeks, 2014, p. 1)

Game development can be challenging but rewarding. It is a combination of innovative technologies with creative ideas (Pirker et al., 2016, p. 54). Getting your idea through designing to a complete product is both exhausting and exciting. Hence choosing correct

tools for development is important. In this study, frameworks were compared with each other and discuss the possible benefits and differences they have.

Web techniques are the main way to display content on the web pages. Web techniques have developed rapidly in recent years with the introduction of the newest version of the HTML standard, HTML5. (Pilgrim, 2010) It is proven that 2-D game can be made with using only HTML5 and JavaScript (Weeks, 2014, p. 6).

1.2. Research problem, goals, and limitations

In this study the focus is on game development with the most popular publicly available HTML5 frameworks. The frameworks are compared with a proof-of-concept games, evaluating their documentation, support, structure, and general performance. The study is limited to three different engines: Phaser, ImpactJS and Construct 2. Additionally, the test games will also be implemented without any use of libraries to see the potential benefits of using libraries during the development. The study also compares frameworks' support for the most popular browsers excluding any lower level differences on browsers. The most popular browsers: Internet Explorer, Chrome, Firefox, Safari, and Android Default alongside their mobile counterparts are compared against each other.

The study will attempt to answer the following research questions:

- RQ1: What are the benefits of using HTML5 game engines for game development?
    - RQ1.1: How do the engines used in the study differ from each other?
- RQ2: When and where HTML5 game engines can be considered as feasible for game development?

1.3. Structure of the study

In the study the prototypes are created with non-library, Phaser, Construct 2 and ImpactJS -approaches. Data for development time, line count, the relative difficulty of development and general performance are gathered during and after the development. Difficulties and possible limitations are discussed in these sections. The possible differences between the frameworks are discussed and analyzed using the gathered results while aiming to answer how viable, beneficial and different they are.

The study first explains the architecture of the application while going through the code and explains how the functionalities are implemented in the respective framework. There are going to be slight differences between prototypes, due to the differences in frameworks. The general design of the game is an endless runner, a game that never ends (Bruce, 2015).

In the end the prototypes are benchmarked for their FPS (frames per second) consistency, loading times and memory consumption. Subjective results of relative difficulty are discussed and compared to the other frameworks alongside the possible bottlenecks and problems that occur during the development.

2 Game Industry

In recent years the cost of development has risen significantly. As the games are becoming bigger and bigger the teams that make those games increase in size and complexity. (Blow, 2004) The video game industry was valued over 93 billion dollars in 2013 (Gartner, Inc., 2013). Since a lot of money is moving in the industry, it is lucrative albeit risky idea to set up a game studio. Bigger companies can work with their existing brands, release sequels and take less risky approaches. Smaller companies need to take huge risks to be able to compete in the market space. the game industry has become a big business with little space for smaller entrepreneurs (Motley Fool, 2016).

Good development tools have become necessary for smaller teams since they can externalize many of the tasks that would require a lot of resources to complete. According to Kasurinen et al. (2013, p. 41) developers are content with existing solutions and the cost of these tools is not considered important. In the recent years the gaming market has become saturated. Nearly 34% (33.8% as of 17.1.2017) of the games on the digital market place Steam were released (Steamspy, 2017) in 2016.

2.1. Game engines

Anderson et al. (2008, p. 229) attempt to answer a question “what is a game engine” by suggesting that game engines are a collection of code that does not directly specify the behavior. It helps to understand the basic concept but perfect description remains to be unseen. In this study HTML5 game engines are used to build the prototypes as the viability of HTML5 for game development will be tested. The chosen game engines are Phaser, ImpactJS and Construct 2. Other potential non-HTML5 choices would be Unreal Engine 4 and Unity, both available for free. Unreal describes its newest iteration Unreal Engine 4 as “complete suite of game development tools made by game developers, for game developers” enabling both 2-d and 3-d development (Epic Games, Inc., 2017). Unity is a game engine adopted by smaller and bigger developers alike (Unity, 2017).

HTML5 Game Engines are young as the canvas API was not fully supported before 2009. API was to be fully supported by Safari and Opera by the end of 2009, and in 2010 both Firefox and Chrome added full support for the API as well (Deveria, 2017)

2.2. HTML5 game engines

Phaser is an HTML5 game engine built for Desktop and Mobile HTML5 game development. It is open source meaning that anyone can build new features and tools for the framework and attempt to fix the underlying issues. Phaser allows game development with WebGL including the fallback option to Canvas if the target isn't supporting WebGL. (Photon Storm Ltd., 2017)

Phaser uses a custom build of Pixi.js for WebGL and canvas rendering. The documentation is comprehensive and the examples can be helpful in solving any issues had during the development. There are over 700 examples currently with source codes available. The community is active which is necessary for a framework that is dependent on open-source development. It is easy to understand the basics of the framework through the several tutorials and examples that their community supports. (Photon Storm Ltd., 2017)

ImpactJS is a game framework built for Desktop and Mobile HTML5 game development. It is a commercial framework and developed by one man, Dominic Szablewski. The framework supports rendering only on Canvas but there are 3rd party solutions to enable WebGL rendering as well. ImpactJS also includes tile-editor that allows rapid level creation for games. ImpactJS's focus lies in creating 2D platformer games but building any type of game is possible. (Szablewski, 2017)

The framework structure is based on inheritance of its class objects, the entities. The most important of these classes is the ig.Entity class. All the gaming objects can be derived and inherited from that class. New functionality can be injected to the class thus giving the same functionality to all the objects that inherit that class. (Szablewski, 2017)

The framework itself is easily extendable and there are numerous plugins that can be used to amplify the development. One of the more popular examples of these is ImpactJS++ which adds tools for pathfinding, UI, dialogue, lighting, camera and so on (Hover, 2014). ImpactJS has a good and descriptive documentation with examples used alongside the API documents. To use ImpactJS to its fullest potential the user needs to learn how the underlying mechanics work.

As a commercial framework the price is $99 and as such the code inside the engine cannot be shared in this study. The personal code created is not restricted by copyright policies. ImpactJS is also deeply integrated to Ejecta enabling an easy way to compile an OpenGL application for an iOS device with a performance comparable to native solutions. (Szablewski, 2017)

Construct 2 is the second version of the GUI (graphical unit interface) HTML5 game creator. It is specifically designed for building 2D games without any requirements for the coding background. You can get a lot out of it by having a good knowledge of JavaScript but it is not mandatory. Construct 2 works with conditions, loops, and triggers. The editor works with the user placing logical elements, ‘blocks', on the editor that creates the logic of the application. Like engines mentioned before, the Construct 2 enables the developers to see the results instantly. (Scirra Ltd., 2017)

Construct 2 includes support for WebGL with Canvas as a fallback option if the browser does not support former. The framework is actively developed and things like object pooling, particle plugin, pathfinding, physics, audio, etc. are included in the package. If something crucial is missing it is more than likely to be found as a plugin or in the next update. The potential downside of Construct 2 is that it needs a different thinking to be able to use all of its capabilities. Most of the development is done using graphical elements. The developer can modify the engine extensively by adding plugins or modifying existing ones. (Scirra Ltd., 2017)

2.3. Game development process

While the current Software engineering process models are not suitable for the game industry, there are similarities, such as agile methodologies, between regular software development processes and game development processes (Kasurinen, 2016, p. 39). Game development starts with creating the design and prototyping based on the designed features. The design describes the project goals and what should be included in the final product. A prototype is a mock-up that tests this features and ideas in action and is evaluated on its potential and risks. The design is a crucial part of the development process since major scope changes at the end of the development process can cause delays and cost increases in the development process. A prototype can be tested with user-segments in so-called

“focus groups” to see if there is potential in the product. Prototyping also allows developers to see if features can be built to work in feasible time-window. Selecting correct tools for the project is part of the design process. Some features may have a need for special tools for implementation. Correct use of useful tools may also speed up the development process and reduce the workload on the project. Using game engines in game development can reduce work for the developers and allow the development to focus on the core functionalities of the product. (KinematicSoup Technologies Inc., 26.10, 2016; Politowski et al., 2016)

When entering production, the focus shifts to work towards completing a set of features during one state of development. Most of the content creation is done during the production. The scope of the product should already be set on the design. Hence the workflow is closer to that of normal software development. (Bethke, 2003, p. 4; KinematicSoup Technologies Inc., 26.10)

Testing is something that should be an ongoing process. It is often overlooked and done relatively late. There are several different aspects that should be considered when testing the game. Code-level unit testing could be done regularly but unit testing relies on well-defined models with expected values. Unit-test on games tend to have a lot of difficulties. Coding conventions, frequent code reviews, and diligent refactoring can keep things in order, though. (Barus et al., 2015, p. 148; Toll and Olsson, 2012, p. 378)

Testing can be a time-consuming process but in the end it will lessen the costs and save time. Finally, the product’s subjective values should be evaluated with user testing. As games are entertainment, they need to entertain. Playtests are especially important for game development and game testing in general. Games don’t necessarily have to be perfect products with zero errors in the implementation. They should be entertaining (Bethke, 2003, p. 14). Many of the bugs left in the games have become features in the final game e.g. Tribes ‘skiing’ (Hi-Rez Studios, 2017).

Playtesting can be divided into two parts: Quality Assurance (QA) and User Acceptance Testing (UAT). QA covers internal testing of the current state of the game for bugs and features that do not function while UAT can also include external tests to evaluate the entertainment value of the game. However, it should be remembered, that game testing

differs from normal software practices. Games' purpose is to entertain and to attain that goal, the game's software doesn't have to be perfect. The focus on the testing is often on the soft aspects such as feel and usability (Kasurinen and Smolander, 2014, p. 9; KinematicSoup Technologies Inc., 26.10)

3 HTML and HTML5

HTML (Hyper Text Markup Language) is a publishing language to publish documents, retrieve online information, design forms and give means to include content to the documents in the WWW (World Wide Web). The language consists of adding 'elements' to the document. The correct format for adding the elements is adding *<[element]>* - tag and closing it with *</[element]>* tag. If you were to produce a document to the web, a web page, you would need to follow these guidelines. An example web page would consist of *<html>*, *<head>* and *<body>* tags. *<img>* and *<audio>* tags could be used to bring multimedia content on the page to make it more compelling. (Castro, 2002, p. 13; Vaughan-Nichols, 2010, p. 14; W3C, 1999; w3schools, 2017a)

The elements can have several different usages based on the attributes they have. The attributes can be assigned by using '=' inside the opening tag of the element. An example page of few paragraphs of text, a clickable link and an image would need following markup: Firstly, you would add *<html></html>* tags, followed with *<head>* and *<body>* tags. For adding the link, you need to add *<a>* tags. Additionally, you should add the target of the link as an attribute for the element resulting with *click me*. The text written in between of the opening and closing tags would serve as the clickable link. The image needs to have the source of the image set. Lastly, paragraphs with * * tags were added with a text string in between the tags, resulting with following code: (Castro, 2002, p. 13; Vaughan-Nichols, 2010, p. 14; W3C, 1999; w3schools, 2017a)

```
<html>
        <head>
        </head>
        <body>
                <a href=www.testpage.com>click me</a>
                <img src=www.testpage.com/test.png></img>
                <p> Testing stuff </p>
                <p> Testing stuff again </p>
        </body>
</html>
```

Since WHATWG (Web Hypertext Application Technology Working Group) published the First Public Working Draft of the HTML5 -specification on 22.01.2008, the support has been growing for the HTML5 alongside the feature set. HTML5 was officially released as a stable W3C Recommendation on 28.10.2014. As the newest specification of HTML the HTML5 has added even more functionality and given new possibilities for the web developers. HTML5 is the next generation of HTML after HTML 4.01, XHTML 1.0, and XHTML 1.1. HTML5 provides new features that are necessary for modern web applications. It adds standards for many features of the web platform that web developers have been using for years. HTML5 is designed to be cross-platform with the only requirement being a modern web browser. Per the logs that W3C has gathered since 2002, the five most used browsers are Chrome, Firefox, Internet Explorer, Safari, and Opera, as can be seen in the table 1.

Table 1. W3schools browser usage statistics 2016 July (w3schools, 2016)

2016	Chrome	IE	Firefox	Safari	Opera
July	71.9%	5.2%	17.1%	3.2%	1.1%

The browsers included in the table 1. support HTML5 even if the support is partial. HTML5 provides the capabilities to display advanced graphics, play audio and store data efficiently, to give a toolset to implement applications that can compete with native solutions. (Pilgrim, 2010; W3C, 2014)

3.1. Cascading Style Sheets

CSS (Cascading Style Sheets) is a language used to define presentation and appearance of an HTML document. CSS is a separate entity from HTML for more easy maintenance and logical working space. Although animations can be done with CSS, the study will focus on the usage of the canvas element. (Ayesh, 2013, p. 78; W3C, 2016)

3.2. JavaScript

JavaScript is an interpreted programming language with OO (object oriented) capabilities. Syntactically, JavaScript resembles C, C++ and Java, with common programming constructs such as the ‘if’ -statement, the ‘while’ -loop, and the ‘&& / | |’ operators. The similarity ends

with this syntactic resemblance since JavaScript is not type-strict language, it is loosely typed one, meaning that variables do not need to have a type specified. JavaScript supports numbers, strings, and Boolean values as primitive datatypes, also including support for Array, Date, and regular expression -objects. (Flanagan, 2001, p. 1)

JavaScript is commonly used in web pages and its purpose is controlling the behavior and interaction on the document. This embedded version of JavaScript is more commonly known as 'client-side' JavaScript emphasizing the idea that the scripts are run client-side rather than server-side. Following example shows the 'client-side' JavaScript in action, modifying the content dynamically on the web. (Flanagan, 2001, p. 4):

```
<html>
    <head>
    </head>
    <body>
        <h1>My First JavaScript</h1>
        <script>
            function MyFirstFunction(){
                return "Hello World";
            }
        </script>
        <button type="button"
            onclick="document.getElementById('demo').innerHTML=
            MyFirstFunction()">
            Click me to display "Hello World"
        </button>
        <p id="demo"></p>
    </body>
</html>
```

First, a function is defined inside '<script>' -tags. The function returns a string "Hello World" when it is called. Next, it is defined when the function is going to be used by adding it to the 'onclick' -event of a button on the web page. When the button is pressed the value of element

‘demo,' a paragraph -element, will change to the return value of ‘MyFirstFunction’ -function displaying “Hello World.”

Because JavaScript is interpreted instead of compiled it is often considered a scripting language instead of a true programming language. It is often assumed that scripting languages are simpler and built for non-programmers. The fact that JavaScript is loosely typed does make it somewhat more forgiving for unsophisticated programmers. Many web designers have been able to use JavaScript for limited, cookbook-style programming tasks. (Flanagan, 2001, p. 2)

JavaScript is a full-featured programming language with complexity at par with other languages. Programmers who attempt to use JavaScript for nontrivial tasks often find the process frustrating due to a lack of understanding for the technique. On HTML5 game engines, JavaScript is the language used to control all important behavior. JavaScript is also used to build layout and presentation by controlling the elements created inside the canvas element. (Flanagan, 2001, p. 2; Williams, 2012, p. 16)

All the HTML5 game engines used in the study need to have following functionality: data loading, update loop and render loop (Photon Storm Ltd., 2017; Scirra Ltd., 2017; Szablewski, 2017). For non-library approach the functionalities can be implemented by creating the functions and calling them at specific time intervals. If the goal is to achieve 60 FPS the canvas needs to update with time intervals of 1000/60 millisecond. ‘requestAnimationFrame’ is an another method that tells the browser that you wish to perform an animation and requests that the browser calls a specified function to update an animation before the next repaint. The update happens 60 times per second matching the display refresh rate in most web browsers as per w3c recommendation. (Flanagan, 2001; Mozilla Developer Network, 2017a; Q. Nguyen, 2013, p. 34; w3schools, 2017b)

4 Support for HTML5 features

Canvas, WebGL, WebAudio and WebStorage (Rettig, 2012, p. 6) are widely used in web-based game development. They are used by Phaser, ImpactJS, and Construct 2 to render and play audio. (Photon Storm Ltd. 2017; Szablewski 2017; Scirra 2017) A table is included that shows current support (table 2.) across the most popular browsers for the mentioned HTML5 features. Support is checked against the newest version of the browser that has a stable version before 29.08.2016.

Full support for the newest version (29.08.2016)	
Partial support for the newest version (29.08.2016)	

Table 2. (Deveria, 2017)

	canvas API	WebGL API	WebAudio	WebStorage
Chrome				
Safari			*prefix -webKit needed*	
Firefox				
Opera				
Internet Explorer			*Uses fallback option to play audio*	
Edge				
iOS Safari			*prefix -webKit needed*	
Android Browser				
Android Chrome				

4.1. HTML5 performance and efficiency on games

The modern PCs don't have problems displaying out acceptable amount of frames per second for games that are done with HTML5 (Geary, 2012, p. 18). The web has become a lot faster in recent years. The performance bottleneck usually lies with lower-performing devices such as smartphones and tablets. Game frameworks help with this problem to a certain extent with optimization and efficient implementations of tools. Scirra has done HTML5 performance tests on its Construct 2 framework from time to time. Comparing the results from figure 1. and figure 2. a trend of improvement can be seen across all the devices. The different platforms have begun to include support for WebGL rendering, boosting the performance. (Scirra Ltd., 2016)

The HTML5 might still be lacking in performance when creating games on a larger scale but for smaller projects it performs decently even on older hardware. Construct 2's test results are relevant for other engines, as the game is tested WebGL rendering, essentially the same technique that other engines use can use as well. There are differences based on how frameworks are optimized around specific drawing logics but the test shows the potential of the HTML5 performance for games. (Scirra Ltd., 2016)

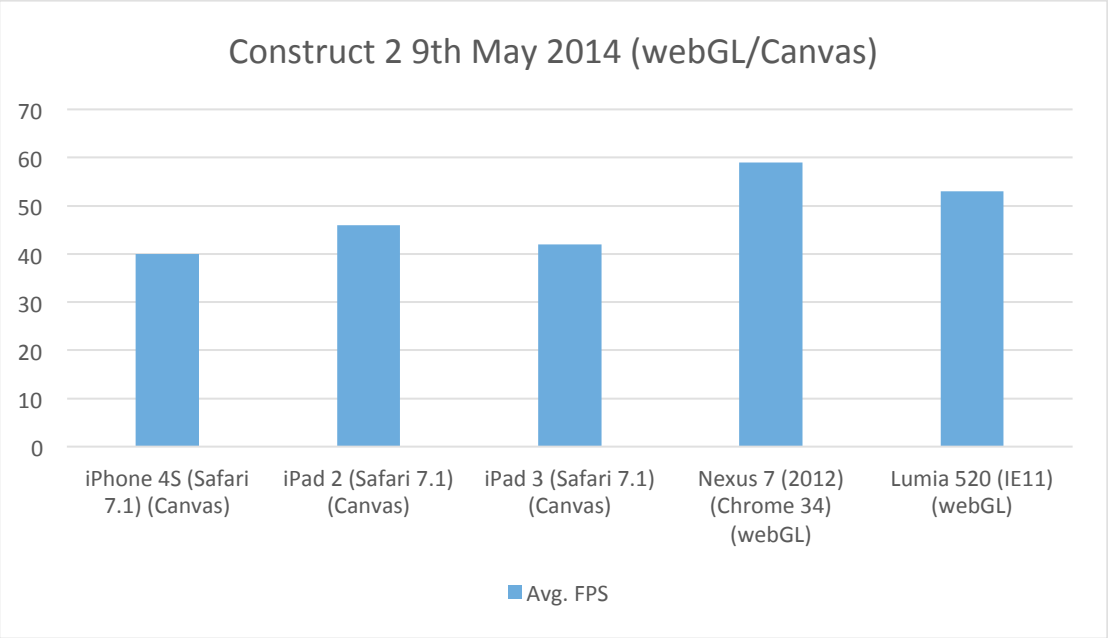

Figure 1. Performance test 1/2. (Scirra Ltd., 2016)

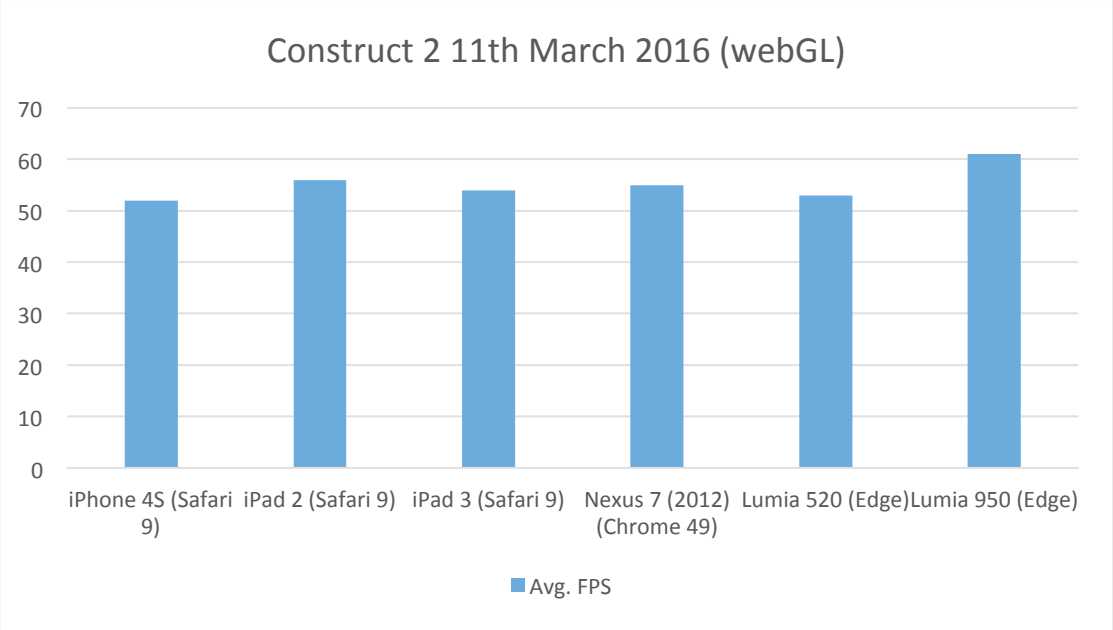

Figure 2. Performance test 2/2. (Scirra Ltd., 2016)

5 Similar research

On a higher level, game development has been a target of several studies over the years. The relative difficulty and complexity were studied by Blow (2004) in his article. The article describes the evolution of early development teams to bigger corporate-based teams. As the projects grow the complexity increases along with teams' sizes. There can be seen similarities with current indie-development where project teams tend to be smaller, due to the often simpler ideas and design.

There are several studies based on HTML5 games. Weeks (2014) studied the limitations of HTML5 technology with a game in his study. He implements an action 2-D game in JavaScript with HTML5. While he doesn't use external frameworks other than the very basic JavaScript libraries he founds HTML5 and JavaScript as viable options for building an action game. Suhonen (2016) implements a collectible card game in his thesis. The study focused on support for the data transmission protocols and support for HTML5 features improving rapidly on the browsers in the future. The first part of Kashayap's (2015) thesis studied if HTML5 could be a replacement for Flash. Finally, ImpactJS was tested and found to be a complete framework with all the functionality needed. Only disadvantages were its price and lack of direct implementation for 3-D rendering.

A non-library solution was built by Janiszewski (2014) to test out the cross-platform capabilities. When PhoneGap and CocoonJS were compared together the results showed that CocoonJS handled smaller textures (64x64) significantly better than PhoneGap. For building a game, CocoonJS was the better choice based on the results of the performance tests.

6 Literature review conclusions

The game industry is currently worth over 100 billion dollars (Gartner, Inc., 2013). In a highly saturated and competitive market space there is a need for tools that amplify the development processes. Choosing correct set of tools for the project is part of the design process and should be done with care. Game development processes share similarities with regular software processes and hence the methodologies used in the latter can also be used.

The recent years have seen HTML5 maturing to set of features that allow users to create applications that can truly be platform free. Knowing the fundamentals of JavaScript, CSS and HTML is crucial for HTML5 game development. HTML5 is built upon previous specifications, introducing new features while still including the backwards-compatibility. HTML5 brought essential features such as Canvas, WebGL, WebAudio and WebStorage are referred under umbrella term: “HTML5”.

While there are still performance constraints when developing HTML5 games it should be remembered that the web and it’s building techniques are constantly evolving, with the browsers optimizing their logic bringing out more performance. Based on the conclusions and similar research projects it can therefore be concluded that the case study can be built with HTML5 features.

7 Case study: Endless Runner

In this study four different prototypes are implemented to test core functionality of HTML5 game engines, and to see how to create a game from scratch. Phaser, ImpactJS and Construct 2 are used in the implementation process. Additionally, a non-library solution is implemented to better view the differences between having a game engine and not having one.

7.1. Endless Runner

'Endless Runner' is a type of game where the player character is constantly moving (Bruce, 2015). In the way of the player there might be goals but the game usually never ends. A lot of games of this type have been released throughout the years such as Flappy Bird (D. Nguyen, 2013) and Robot Unicorn Attack (Adult Swim Games, 2010).

7.2. Building the prototypes

The popular known design patterns are used as guidelines when implementing and designing the prototypes. ImpactJS implements its object-oriented structure resulting in different architectural structure from Phaser and Non-library solution that follow the Model-View-Controller pattern. The design of game is relatively simple and ImpactJS tools are handling much of the heavy-lifting. A lot of functionality that otherwise would be separated into different modules are instead included into a single main.js file with respective files for two entities used in the game. Construct 2 follows its way of doing this with Layout (view) and Events (logic and input) -sheets handling the work.

Global variables are used to some extent since this is a prototype and therefore encapsulation or security doesn't have to be taken into consideration. The goal is to get a working product as efficiently as possible without limiting the performance. In the parts where the code is explained there are text written with *italic* styling. Variables, objects and functions are identified in the text with italic styling. Some of the sentences might start with lower-case since in the JavaScript code following objects are separate entities and don't necessarily share anything: *objectName* and *ObjectName*. JavaScript is case-sensitive that's why it is necessary to start some of the sentences with lower-case if we want to refer to correct part of the code.

7.3. MVC

The MVC (model view controller) design pattern was developed in Smalltalk-80 in 1988 (GuangChun et al., 2003, p. 1). It is a design pattern suited for structuring an application that is reliable and easily maintainable. MVC designed to abstract out some of the calls that might otherwise cause faults and issues in the application. Objects of different classes take over operations related to the application domain, the display of the application's state and the user interaction with the model and the view. MVC -approach is used for the prototypes for the clarity that the design pattern brings to the structure. (Ocariza et al., 2015)

7.4. Comparability

The goals for prototypes are clearly set so the tools used to implement these prototypes could be easily compared and analyzed. The games are supposed to meet following requirements:

- Car image on the screen
- White boxes that move towards the car
- Collision between car and white box resets the score
- Every white box that exits the canvas from left side gives player one point
- After collision the score is saved to storage
- Keep track of best scores
- Attempt to render at target FPS of 60

Minor differences between prototypes such as the height of the jumps and speed of the animations are accepted. The process of implementing the games are reported along with the time used for development, relative performance across different browsers, and the differences in code. Phaser, ImpactJS, Construct 2 and non-library approach are analyzed to find out the main differences.

7.5. Non-library approach

The code was separated into “model”’, “view”, and ‘controller’ folders. On non-library approach everything is built from scratch by just using the toolset HTML5 gives us. Luckily, as mentioned before, the toolset is suitable for doing small projects even without any extra libraries. Drawing logic, update logic, loading, and event handling are needed for the complete game. Lastly, objects needed in the game (car, obstacles, text and so on) are to be added to the game. The final architecture of the application can be seen in figure 3. Images, controllers, views, models and sounds re separated into different folders as per MVC design pattern.

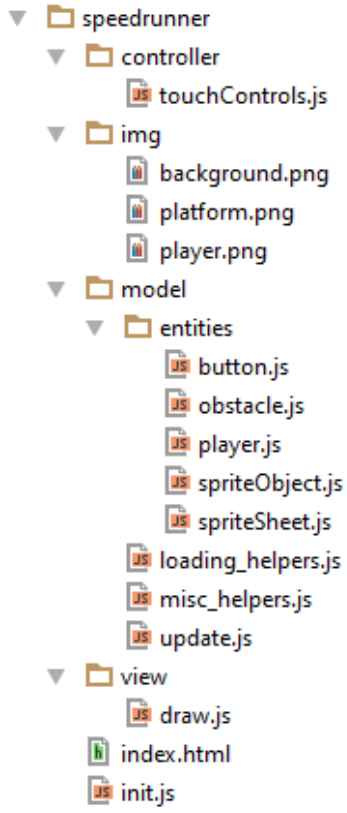

Figure 3. non-library architecture

Sublime Text 2 (Sublime Text 2 2013) is used for development. First, a file that connects all the JavaScript -files together, is created. This HTML -file was named “index.html.” A canvas element is added to the “HTML -file to draw graphics on the screen. To set up the drawing

and updating logic the game needs JavaScript functions. “init.js” -file was added to project for that reason. The “index.html” -file will be similar to following example:

```
<html>
	<title>Non-library Speedrunner</title>
	<head>
	</head>
	<body>
		<canvas id=”canvasId” width=”300” height=”200”></canvas>
		<script type=”text/javascript” src=”init.js”></script>
	</body>
</html>
```

Logic for updating the game, drawing the images and doing that at pre-defined intervals is needed. There are two suitable choices for doing this. Using time intervals or using requestAnimationFrame API. When using requestAnimationFrame, the control of the animation is given to the browser, resulting in more efficient results (Irish, 2011; Q. Nguyen, 2013, p. 34). The function is first going to load the assets and after loading sets the flag that prevents the program to load the assets again.

A variable is defined as a flag for loading so it can be used to detect whether the assets are still being loaded or not:

```
var loading = true;
```

A global variable for *canvas* element is created to access the “context”. A context is needed for drawing graphics on the *canvas* element. Finally, two variables for game’s width and height are created.

```
var canvas = document.getElementById(“canvasId”);
var context = canvas.getContext(“2d”);
var GAME_WIDTH = 300;
var GAME_HEIGHT = 200;
```

In *gameloop* function the game assets are loaded, the update and draw functions are called and the requestAnimationFrame loop is started. The implemented code is following:

```
var gameLoop = function(){
        if(loading){
                loadEntities();
        }
        update();
        draw();
        requestAnimationFrame(gameLoop, canvas);
}
requestAnimationFrame(gameLoop, canvas);
```

To display graphics on the screen a drawing logic is needed. The game needs to have background, text and entities (player and obstacles). Functions for background, text and entity drawing are created. The functions are named *drawBackground*, *drawText* and *drawEntities* respectively. *draw* function calls these functions to draw the needed graphics on the screen. The implemented code is following:

```
function drawBackground(){
        context.drawImage(_background,0,0);
}
function drawText(){
        context.font=”20px Arial Black”;
        context.fillStyle=”yellow”;
        context.fillText(“Top Score: “+currentHighScore.toString(), 100, 35);
        context.align=”left;
        context.font=”16px Arial Black”
        context.fillStyle=”white”;
        context.fillText("Score: "+currentScore.toString(),100,79);
}
function drawEntities() {
        for (var i = 0; i < entityList.length; i++) {
```

```
                entityList[i].draw();
        }
}
function draw(){
        drawBackground();
        drawText();
        drawEntities();
}
```

Before displaying any graphics on the screen the assets have to be loaded. First, a global variable *_player* is created for so that *Player* object can be accessed throughout the code. *loadEntities* function will handle loading of *Player* object and background image. In the function a *Player* object is created and added to the *entityList* array. *entityList* is used to store all the objects used in the game. Finally, *loading* variable is set to **false** signaling that loading is no longer active. The implemented code is following:

```
var _player = null;
var _background = null;
var entityList =[];
function loadEntities(){
        _player = new Player(64,100,'player',32,32);
        entityList.push(_player);
        _background = loadImg('background');
        loading = false;
}
```

loadImg function is used to create image objects so they can be drawn:

```
function loadImg(imgName){
        var img = new Image();
        img.src = "img/" + imgName + ".png";
        return img;
}
```

Game's objects (player and obstacles) logic must be constantly updated. To do that, an update function is needed. First, two variables are created to track the obstacles' spawn interval:

```
var frameCounter = 0;
var frameLimit = 60;
```

Math.random() function is a JavaScript method of generating a random number between 0 and 1. By multiplying *Math.random()* with 80, a number between 0 and 80 is generated. Hence variance can be added to the number of frames needed for spawning the obstacles. When the obstacles move out of the screen (when the x-position is less than 0), their *canRemove* property is set to **true**, signaling that these objects should be removed from the *entityList* and eventually removed by the garbage collector (Mozilla Developer Network, 2017b).

```
function update(){
    if(frameCounter > frameLimit){
        frameLimit = 80 + (Math.random() * 80);
        entityList.push(new Obstacle(GAME_WIDTH,GAME_HEIGHT-
        8,entityList.length));
         frameCounter=0;
     }
    frameCounter++;
```

updateEntities function is called to update every object inside the *entityList*. After updating the entities *gravityController function* is called. *gravityController* constantly increments the y position of the objects to create a feeling of gravity when jumping. Objects' canRemove property is constantly checked and if it is set to **true** the respective object will be removed from the *entityList* array.

```
    updateEntities();
    gravityController();
```

```
        for(var i = 0;i<entityList.length;i++){
                if(entityList[i].canRemove){
                        entityList.splice(i,1);
                }
        }
}

function updateEntities(){
        for(var i = 0; i < entityList.length; i++)
        {
                entityList[i].update();
         }
}
```

Because of the lack of tools in the non-library approach a collection of helper functions is created. Since the game is a prototype there is no need to separate these helper functions to different categories. First, three variables are created to keep track of the current scores and current gravity level.

```
var currentScore = 0;
var currentHighScore = 0;
var _gravity = 5;
```

In the game the *Player* object can jump over the obstacles. After jumping the *Player* object should return to the y position it jumped from. Hence a logic for gravitation is needed. *gravityController* function handles the gravity of the game. The acceleration speed for gravity is *_gravity* * 0.01 pixels per second, resulting 0.05 pixels per second increase on falling every second. The value of the variable *_gravity* is arbitrary.

```
function gravityController(){
        for(var i = 0; i < entityList.length; i++){
                if(distance(entityList[i].posx,entityList[i].posx,entityList[i].posy,GAME_HEIGHT-
                entityList[i].height) > 1){
```

```
                entityList[i].veloY += _gravity*0.01;
            }
            else{
                entityList[i].veloY = 0;
            }
        }
}
```

To detect if the *Player* object overlaps with the obstacles a distance calculation is needed. *distance* function is created to calculate the distance between two objects.

```
function distance(x1,x2,y1,y2){
    return Math.sqrt( ( x1 - x2 ) * (x1 - x2) + (y1 - y2)*(y1 - y2) );
}
```

checkBoundingWorld function checks if the object is on ground level, thus preventing *gravityController* function setting the position of the *Player* object outside the *canvas*.

```
function checkBoundingWorld(x,y,width,height,added){
    if(added < 0){
        y+=added;
    }
    else{
        if(distance(x,x,y,GAME_HEIGHT-height) < 1){
            y = GAME_HEIGHT - height;
            return y;
        }
        if(y < GAME_HEIGHT){
            if(y + height > GAME_HEIGHT){
                y= GAME_HEIGHT - height;
            }
            else{
                y+=added;
```

```
                    }
                }
            }
            return y;
}
```

To display the high scores there's a need to store the previous scores and compare them with each other to find out the highest value. The highest value will be drawn to the canvas. *setHighScor*e function attempts to get the previous results from the local storage. If there are no results to be found an empty array is generated. All the data stored in the local storage must be converted to string format. The resulting string is then parsed and added to an array. That array is sorted with *Math.sort()* method with ascending order from highest score to the lowest. The scores are stored if there's a need to display any of the lower scores than just the highest one.

```
function setHighScore(){
        stringArray = localStorage.getItem('highScoreList');
        if(!stringArray){
                stringArray = "[0,0,0,0,0]";
        }
        parsedArray = JSON.parse(stringArray);
        parsedArray.push(currentScore);
        parsedArray.sort(function(a, b){return b-a});
```

The last score is removed from the array so only five results remain. The array's highest value is assigned to a variable *currentHighScor*e. Lastly, the array is converted to a string and added to the local storage with a key *highScoreList*.

```
        parsedArray.splice(1,parsedArray.length-1);
        drawHighScore(parsedArray);
        localStorage.setItem('highScoreList',JSON.stringify(parsedArray));
}
```

currentHighScore variable's value is constantly being drawn on the canvas as text. *drawHighScore* function sets the first member of the array as the current high score.

```
function drawHighScore(highScoreList){
        currentHighScore = highScoreList[0];
}
```

A general object, *spriteObject*, for sprites is needed that stores positional information, size of the sprite's image and image's data.

```
var spriteObject = function(x,y,img,width,height){
        this.sourceX = x;
        this.sourceY = y;
        this.img = img;
        this.width = width;
        this.height = height;
};
```

spriteObject will be drawn to the coordinates that are received as parameters.

```
spriteObject.prototype.draw = function(x,y){
        context.drawImage(this.img,this.sourceX,this.sourceY,this.width,this.height,x,y,32,32);
};
```

To create animations the images must be drawn in a specific order. A spritesheet is a big image that has several sprites packed in it. It is parsed into an array of sprites where the sprites can be controlled per the animation logic. The size of one sprite is pre-defined in this study. A spritesheet object, *spriteSheet*, is needed to handle the logic for animations.

```
var spriteSheet = function(x,y,img,width,height,number,fps,animation){
        this.spriteObjectArray = [];
         var totalWidth = width*4;
        this.fps = fps;
        this.counter = 0;
```

```
        this.animation = animation;
        this.animationFrame = 0;
        for(var i = 0; i < number; i++){
                this.spriteObjectArray.push(new
                spriteObject(i*(totalWidth/number),0,img,width,height));
        }
};
```

animationCalc function handles the animation loop for the *spriteSheet object*. The animation speed is based on FPS in this prototype.

```
spriteSheet.prototype.animationCalc = function(){
        if(this.counter> (1000/this.fps)){
                this.counter= 0;
                if(this.animationFrame<this.animation.length-1){
                        this.animationFrame++
                }
                else{
                        this.animationFrame=0;
                }
        }
        else{
                this.counter++;
        }
};
```

A image will be drawn on the canvas based on the value of the *animationFrame* variable.

```
spriteSheet.prototype.draw = function(x,y){
        this.animationCalc();
        this.spriteObjectArray[this.animation[this.animationFrame]].draw(x,y);
};
```

The game needs to have controllable character, the *Player* object. The *Player* object stores positional information, size and a *spriteSheet* object.

```
var Player = function(x,y,img,width,height){
        this.posx = x;
        this.posy = y;
        this.width = width;
        this.height = height;
        this.spriteSheet = new spriteSheet(this.posx,this.posy,loadImg("player"),this.width
,this.height,4,260,[0,1,2,3]);
        this.veloY = 0;
        this.veloX = 0;
};
```

The *Player* object's position is updated per the y velocity with the help of function *checkBoundingWorld* function, *spriteSheet* object will be drawn to the *Player* object's position.

```
Player.prototype.update = function(){
        this.posy = checkBoundingWorld(this.posx,this.posy,this.width,this.height,this.veloY);
};

Player.prototype.draw = function(){
        this.spriteSheet.draw(this.posx,this.posy);
};
```

Since the *Player* object needs to jump over the obstacles a jumping logic is needed. First a check is made to verify that the *Player* object is on ground level. If it is on ground level its *veloY* value is decremented by 1.25, resulting upwards acceleration on the *canvas*.

```
Player.prototype.jump = function(){
        if(distance(this.posx,this.posx,this.posy,GAME_HEIGHT-this.height) < 1){
                this.veloY += -1.25;
        }
```

```
};
```

An obstacle object is needed so that *Player* object has obstacles to jump over. Implemented *Obstacle* object includes a constructor where object's position and id is specified. Its *veloX* value is randomized between 1.5 – 2 to give variance to the gameplay.

```
var Obstacle= function(x,y,id){
        this.posx = x;
        this.posy = y;
        this.canRemove = false;
        this.id = id;
        this.img = loadImg("platform");
        this.canScore = true;
        this.veloY = 0;
        this.veloX = (Math.random() * 0.5) + 1.5;
};
```

When *Obstacle* object needs to be removed its *kill* function is called and *canRemove* property will be set to **true**. As explained in the update.js implementation the objects that have their *canRemove* property set to **true**, are removed from the *entityList* array.

```
Obstacle.prototype.kill = function(){
        this.canRemove = true;
};
```

When the object exits the screen from the left edge the player's score will increase by one. Lastly the object will be drawn with *draw* function.

```
Obstacle.prototype.update = function(){
        this.posx -= this.veloX;

        if(this.posx + 16 < 0 && this.canScore){
                currentScore++;
```

```
                this.kill();
                this.canScore=false;
        }

         if(distance(this.posx + 4,_player.posx + 16,this.posy +4,_player.posy) < 32){
                this.canScore=false;
                setHighScore();
                currentScore = 0;
         }
};

Obstacle.prototype.draw = function(){
         context.drawImage(this.img,this.posx,this.posy);
};
```

To give the user control user input must be handled. For that an event listener is added to the game: “mousedown” for mouse events. The “mousedown” event is resolved resulting the *Player* object jumping.

```
canvas.addEventListener('mousedown', function(e) {
        resolvePressed();
}, false);

function resolvePressed(){
  if(distance(_player.posx,_player.posx,_player.posy,GAME_HEIGHT-_player.height) < 1){
        _player.jump();
        }
}
```

7.6. Phaser approach

The code will be separated into Model and Controller folders. Phaser handles the 'view' portion of the logic by itself. The elements are just added to the *stage* object Phaser provides. After adding an object as the child of *stage* object it is automatically updated as long as *destroy()* or *kill()* is not called. There will be white boxes that need to be spawned and destroyed multiple times. Spawning objects create a lot of work the garbage collector and result in stutter due to the high frequency of garbage collection events. Alternatively *kill()* method can be used instead of *destroy()*. *destroy()* method destroys the object while *kill()* method just removes the object from the rendering list. The object can be brought back to the game with *reset()* method after killing it with *kill()* method.

The final architecture of the application can be seen in figure 4. The file 'phaser.js' is not categorized under MVC structure since it is a relatively large (3 megabytes ~) library and its functionality parallels with model, view and controller.

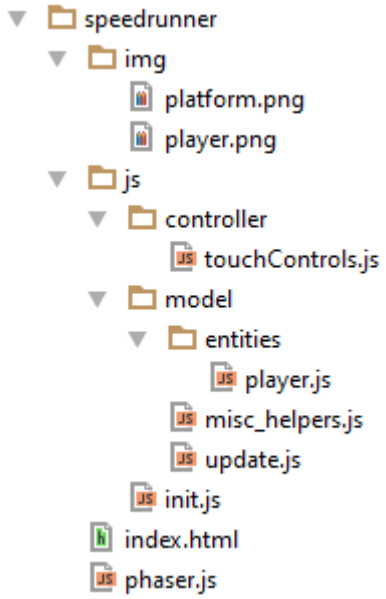

Figure 4. Phaser architecture

Again, Sublime 2 is used for development. The project starts by creating the HTML -file that links all the files together. The prototype is done with Phaser and hence there are a lot of tools ready to use reducing the complexity of the project files. Phaser's core file phaser.js itself is almost 3 megabytes sized library. The index.html -file looks like in the following example:

```
<html>
    <head>
        <script type="text/javascript" src="phaser.js"></script>
        <script type="text/javascript" src="js/controller/touchControls.js"> </script>
        <script type="text/javascript" src="js/model/entities/player.js"> </script>
        <script type="text/javascript" src="js/model/misc_helpers.js"> </script>
        <script type="text/javascript" src="js/model/update.js"> </script>
        <script type="text/javascript" src="js/init.js"> </script>
    </head>
    <body>
    </body>
</html>
```

Two constant values are set for game view's width and height. Additionally, global variables *scoreText* and *player* are created for easy referencing to the *Player* object and to the *text* object that displays the current score.

```
var GAME_HEIGHT = 200;
var GAME_WIDTH  = 300;
var scoreText = null;
var player = null;
```

Phaser needs a *game* object so its toolset can be used for the game logic. The constructor creates a context for the user that includes canvas and Phaser's toolset for development. Constructor's arguments decide the width and height of the element, rendering logic (canvas or WebGL) and the functions that will be used to initialize the game.

var game = new Phaser.Game(GAME_WIDTH, GAME_HEIGHT, Phaser.CANVAS, '',{ preload: preload, create: create, update: update});

preload function is used to load all the assets needed in the game, before the actual game is even loaded. *create* function is used to set up the physics engine, create the *Player* object, set up the gravity level, add *text* objects that are needed and create timed event for obstacle spawning.

```
function preload() {
        game.load.spritesheet('player', 'img/player.png',32,32,4);
        game.load.image('platform', 'img/platform.png');
}
function create() {
        game.physics.startSystem(Phaser.Physics.ARCADE);
        game.time.advancedTiming = true;
        player = new Player(game,64,20);
        game.add.existing(player);

        addInput();
        game.physics.arcade.gravity.y = 400;
        game.time.events.add(1500,createObstacle);

        scoreText = game.add.text(100, 64, "Score: "+currentScore.toString(),{font: "16px
        Arial Black"});
        scoreText.fill = "white";

        highScoreListText.push(game.add.text(100, 16, "Top Score: ",{fill:"yellow", font: "20px
        Arial Black", align: "left"}));
        setHighScore();
}
```

The high score text needs to be changed when user's score value is changed. The logic is similar to what was done in the non-library approach. The score value changes if the *Obstacle* object exits the screen from the left edge. When the *Obstacle* object overlaps the

Player object the score is checked against the current high score and if the score is higher it is updated as the new high score.
An array is created for *Obstacle* objects addition to variables for the current score value and the previous score value. The previous score is stored while the displayed score changes to the highest value that can be found from the storage.

```
var ObstacleArray = [];
var currentScore = 0;
var prevScore = -1;
```

update function handles updating the score text whenever it changes. Collision detection is constantly checked in the function to detect if *Player* object and *Obstacle* object overlap. Finally, the input events will be listened.

```
function update() {
    if(prevScore !== currentScore){
        scoreText.setText("Score: "+currentScore.toString());
        prevScore = currentScore;
    }
    for(var i = 0; i < ObstacleArray.length; i++){
        bufObj = ObstacleArray[i];
        bufObj.x -= bufObj.speedVal;

        game.physics.arcade.collide(player,bufObj, zeroScore);
        if(bufObj.x < bufObj.width){
            if(bufObj.canScore){
                bufObj.canScore=false;
                currentScore++;
            }
            bufObj.destroy();
            ObstacleArray.splice(i,1);
        }
    }
```

```
        checkInput();
}
```

The implementation and the logic is the same as in non-library approach in chapter 7.5

```
var highScoreListText = [];
function setHighScore(){
   stringArray = localStorage.getItem('highScoreList');
        if(!stringArray){
                stringArray = "[0,0,0,0,0]";
        }
        parsedArray = JSON.parse(stringArray);
        parsedArray.push(currentScore);
        parsedArray.sort(function(a, b){return b-a});
        parsedArray.splice(1,parsedArray.length-2);
        drawHighScore(parsedArray);
        localStorage.setItem('highScoreList',JSON.stringify(parsedArray));
}
function drawHighScore(highScoreList){
        highScoreListText[0].setText("Top Score: "+highScoreList[0].toString());
}
```

In Phaser there's already *sprite* class that can be used to load *sprite* objects. The *sprite* object's constructor receives width, height and the name of the sprite as parameters. Randomized value is added to object's *speedVal* property to create variance to the speed of the obstacles in the game. Physics are enabled for the object and finally the created *sprite* object is added to an array. Time event is added to call *createObstacle* function after 1000 – 1999 milliseconds.

```
function createObstacle(){
        sprite = game.add.sprite(GAME_WIDTH,GAME_HEIGHT-8,'platform');
        sprite.scale.set(0.25);
        sprite.speedVal = Math.random() + 3;
```

```
        sprite.canScore = true;
        game.physics.enable(sprite, Phaser.Physics.ARCADE);
        sprite.body.collideWorldBounds=true;
        ObstacleArray.push(sprite);
        game.time.events.add(1000 + (Math.random() * 1000),createObstacle);
}
```

A functionality for setting resetting the score is added. *zeroScore* function is called whenever the two objects overlap each other. *currentScore* variable is set to zero and *obj2*'s *canScore* property to false so that the *Obstacle* doesn't increase *currentScore* by one when moving out of the canvas area.

```
function zeroScore(obj1,obj2){
        obj2.canScore=false;
        setHighScore();
        currentScore = 0;
}
```

Player object is significantly reduced in size and complexity compared to the non-library approach. The object inherits the *Phaser.Sprite*'s functionality. An animation array is added to the object and the animation is started. Lastly the physics are enabled for the object and *body.collideWorldBounds* is set to **true** to prevent the *Player* object from falling through the bottom of the *canvas* area.

```
Player = function (game, x, y) {
        Phaser.Sprite.call(this, game, x, y, 'player');
        anim = this.animations.add('drive',[0,1,2,3],25);
        anim.play(25,true);
        game.physics.enable(this, Phaser.Physics.ARCADE);
        this.body.collideWorldBounds = true;
        this.body.bounce.y = 0.02;
};
```

```
Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;
```

Enabling input is relatively simple in Phaser. For mouse input, game objects input.mouse.capture property is set to true. After setting the value, the mouse events are now registered.

```
function addInput(){
        game.input.mouse.capture = true;
}
```

The mouse events can be handled with following code:

```
function checkInput(){
        if( player
        &&  game.input.activePointer.isDown
```

Whenever *Player* object's y value is 1 pixel away from the height of the canvas a negative y value set causing *Player* object to jump.

```
        && (player.y + player.height <= GAME_HEIGHT + 1)
        && (player.y + player.height >= GAME_HEIGHT - 1)
        ){
        player.body.velocity.y = -100;
        }
}
```

7.7. ImpactJS approach

Only three JavaScript -files are created for this game; main.js that handles the drawing and game logic, ‘obstacle’ -object and ‘player’ -object as ImpactJS does a lot of the heavy lifting and mandatory logic for the developer. The final architecture of the application can be seen in figure 5. Architecture follows the initial folder structure the ImpactJS providers for the user.

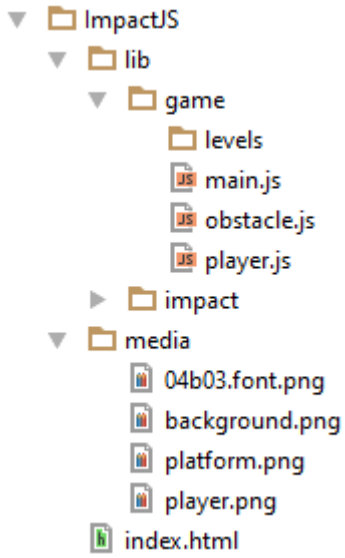

Figure 5. ImpactJS architecture

As with Phaser the Sublime 2 is used for development. The HTML file is created at the start of the project.

```
<html>
	<head>
		<script type="text/javascript" src="lib/impact/impact.js"></script>
		<script type="text/javascript" src="lib/game/main.js"></script>
	</head>
	<body>
		<canvas id="canvas"></canvas>
	</body>
</html>
```

First, *MyGame* object is created by inheriting the ImpactJS's *ig.Game* object. New variables are created for score listing, gravity value and reference for 'Player' object.

```
ig.module(
	'game.main'
)
.requires(
	'impact.game',
	'impact.font',
	'game.player',
	'game.obstacle'
)
.defines(function(){

MyGame = ig.Game.extend({

	font: new ig.Font( 'media/04b03.font.png' ),
	gravity: 200,
	currentScore: 0,
	currentHighScore: 0,
	highScoreList: [],
	player: null,
```

init function creates *Player* object and assigns it to the *MyGame*'s local property so it can be referenced throughout the application. Event listeners for touch and mouse events are added and logic for obstacle spawning. Finally, the high score is loaded from the local storage. *(The font-size of the code is reduced for readability.)*

```
	init: function() {

		this.player= ig.game.spawnEntity(EntityPlayer, 64, 200 );
		document.addEventListener( 'touchstart', this.player.jump.bind(this.player), false );

		document.addEventListener( 'mousedown', this.player.jump.bind(this.player), false );
		this.obstacleSpawner();
```

```
            this.setHighScore();
        },
        obstacleSpawner: function(){
            this.spawnEntity(EntityObstacle, 400, 192 );
        },
```

draw function draws the current and highest score values to the screen. There are also setHighScore and drawHighScore functions that were implemented in non-library approach on chapter 7.5.

```
        draw: function() {
            this.parent();
            this.font.draw( "Current Score: " + this.currentScore, ig.system.width/2, 20, ig.Font.ALIGN.CENTER );
            this.font.draw( "High Score: " + this.currentHighScore, ig.system.width/2, 10, ig.Font.ALIGN.CENTER );
        },
        setHighScore: function(){
            var stringArray = localStorage.getItem('highScoreList');
            if(!stringArray){
                stringArray = "[0,0,0,0,0]";
            }
            var parsedArray = JSON.parse(stringArray);
            parsedArray.push(this.currentScore);
            parsedArray.sort(function(a, b){return b-a});
            parsedArray.splice(1,parsedArray.length-1);
            this.drawHighScore(parsedArray);
            localStorage.setItem('highScoreList',JSON.stringify(parsedArray));
        },
        drawHighScore: function(highScoreList){
            this.currentHighScore = highScoreList[0];
        }
        });
        ig.main( '#canvas', MyGame, 60, 300, 200, 1 );
});
```

Player object is extended from the *Entity* object and it is specified as *TYPE.A* object that should check collision against *TYPE.B* objects. Its collision type is also set to *FIXED* meaning that it won't move in the event of collision.

```
ig.module(
        'game.player'
        )
        .requires(
```

```
        'impact.entity'
)
.defines(function(){
        EntityPlayer = ig.Entity.extend({
                collides: ig.Entity.COLLIDES.FIXED,
                type: ig.Entity.TYPE.A,
                checkAgainst: ig.Entity.TYPE.B,
                size: {x:32, y:32},
                animSheet: new ig.AnimationSheet( 'media/player.png', 32,32 ),
```

Spritesheet is added and the animation set up in *init*, while *update* handles the check that the objects y coordinate stays below a specific threshold.

```
                init: function( x, y, settings ) {
                        this.parent( x, y, settings );
                        this.addAnim( 'idle', 0.05, [0,1,2,3] );
                },
                update: function(){
                        this.parent();
                        if(this.pos.y > ig.system.height-32){
                        this.pos.y = ig.system.height-32;
                        }
                },
```

jump is the only non-default function to be added to this entity. It adds y velocity of -100 to the object if the object's y value is equal to the height-32.

```
                jump: function(){
                        if(this.pos.y === ig.system.height-32){
                                this.vel.y = -100;
                        }
                },
```

check is added to handle the collision events. ImpactJS supports collision checking out of the box and check function is one of the functions that are inherited from the *ig.Entity* object. In the function the object's *name* property is checked to verify that it is the correct object and if it is, the score is reset and high score updated if needed.

```
                check: function(other){
                        if(other.name==="obstacle"){
                                ig.game.setHighScore();
```

```
                                        ig.game.currentScore=0;
                                    }
                                }
                            });
                    });
```

Obstacle object is like *Player* object with differences on colliding logic. Its colliding type is set to *NONE* meaning that it will pass through other objects. Type is set to *TYPE.B* so that *Player* objects will check against it when they overlap. Gravity shouldn't affect obstacles, only the *Player* object. Hence the *gravityFactor* is set to 0.

```
ig.module(
            'game.obstacle'
            )
            .requires(
            'impact.entity'
            )
            .defines(function(){
                        EntityObstacle = ig.Entity.extend({
                        name: "obstacle",
                        collides: ig.Entity.COLLIDES.NONE,
                        type: ig.Entity.TYPE.B,
                        checkAgainst: ig.Entity.TYPE.A,
                        size: {x:8, y:8},
                        gravityFactor: 0,
                        canScore: true,
                        animSheet: new ig.AnimationSheet( 'media/platform.png', 8, 8 ),
                        timerDelta: null,
```

An initial x velocity of -150 is added for the object causing the object to move from right to left with fixed speed. A timer is added and after the set time has passed a new *Obstacle* object is spawned.

```
                        init: function( x, y, settings ) {
                                    this.parent( x, y, settings );
                                    this.addAnim( 'idle', 1, [0] );
                                    this.vel.x = -150;
                                    this.timerDelta = new ig.Timer((Math.random() * 1.5) + 1);
                        },
```

If object manages to pass the *Player* object one score is added. When *Obstacle* object exits the screen it is removed from the game with the *kill* function, a function that is also inherited from *ig.Entity*.

```
                update: function(){
                        if(this.timerDelta && this.timerDelta.delta() > 0){
                                ig.game.obstacleSpawner();
                                this.timerDelta = null;
                        }
                        if(this.pos.x < ig.game.player.pos.x - 32 && this.canScore){
                                ig.game.currentScore++;
                                this.canScore = false;
                        }
                        if(this.pos.x < -32){
                                this.kill();
                        }
                        this.parent();
                }
        });
});
```

7.8. Construct 2 approach

Construct 2 differs a lot from Phaser and ImpactJS. The user needs to add blocks that represent functionality in the code. Construct works on the general principle of conditions, actions and combination of both. First, the variables are defined. The blocks added after variable declaration are explained in order of appearance.

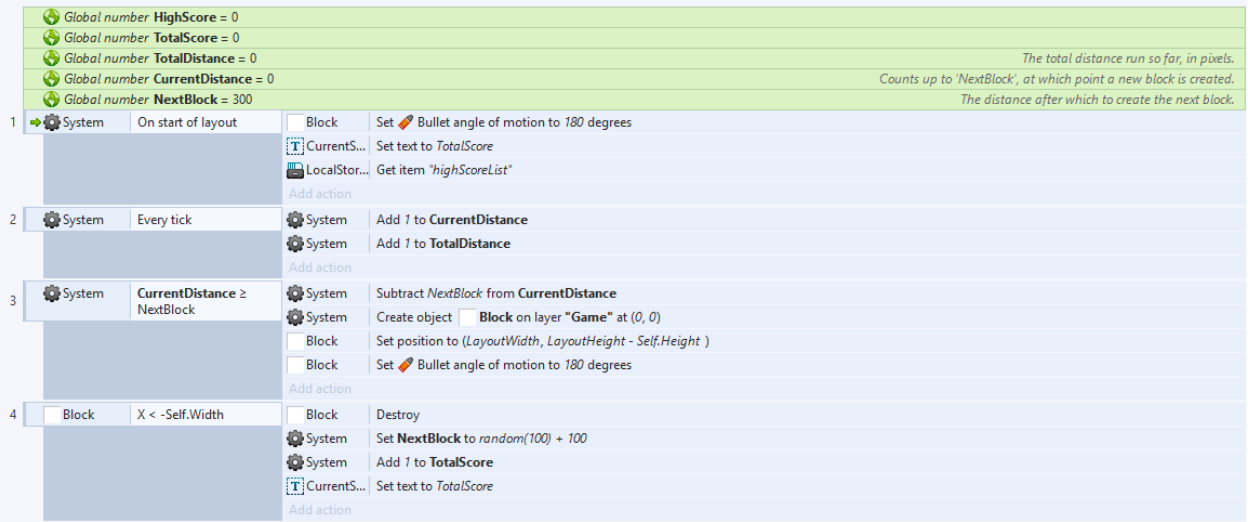

Figure 6. Event sheet 1.

Block 1. Condition block ‘On start of layout’ is added with actions that should be executed when the layout starts.

Block 2. Condition block ‘Every tick’ updates these actions every update tick, decrementing the distance variables.

Block 3. When ‘CurrentDistance’ -value exceeds set threshold ‘NextBlock,' a new ‘Block’ will spawn and its position set to the right edge of the screen.

Block 4. When ‘Block’ -object exits the screen from the left edge, a new spawning threshold is set and the current ‘Block’ destroyed. Additionally, the current score is then updated.

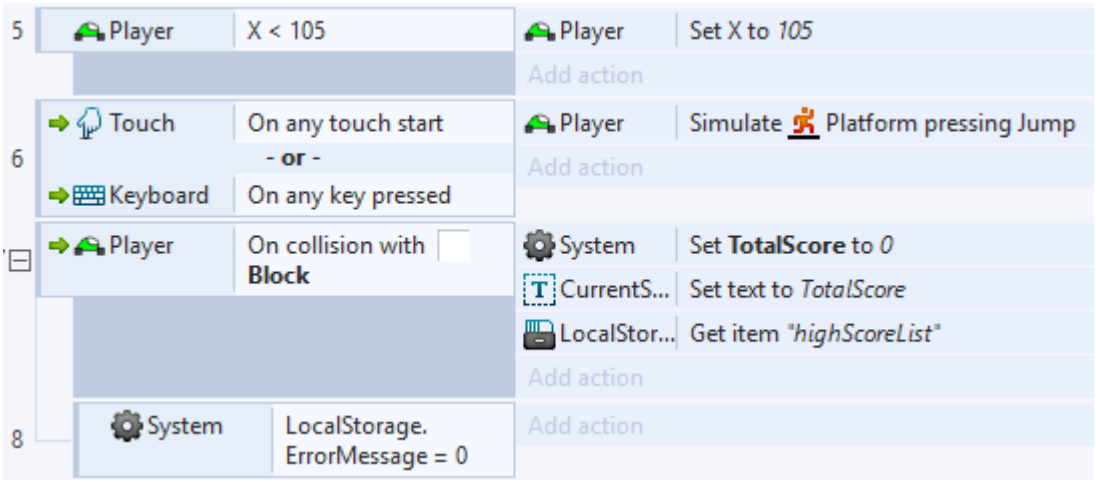

Figure 7. Event sheet 1.

Block 5. Makes sure that the “Player” stays in the set position.

Blocks 6 – 8. Block 6. Handles the jumping logic for “Player,” while 7 – 8 blocks handle updating the score to the scoreboard when a collision is detected between “Player” and “Block.”

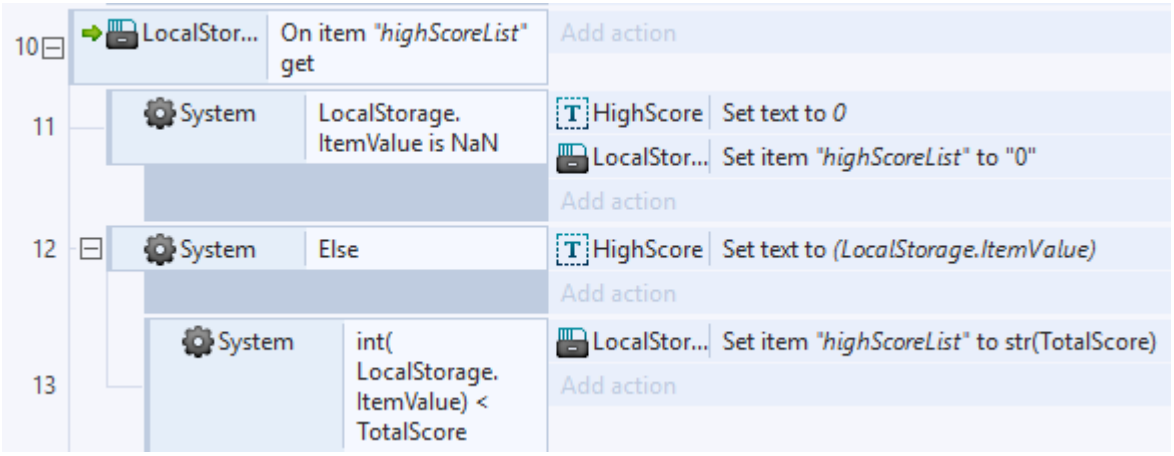

Figure 8. Event sheet 1.

Blocks 10 – 13. Handle the logic with fetching the high scores from the local storage.

8 Results

After implementation there are four different versions from the same game idea. The games can be played by clicking on the *canvas* element to jump over the white boxes. Every successful jump nets the player one point. The highest score will be locally stored so that the user can aim for better scores. Following figures x – y show how the final versions look like in each of the approaches:

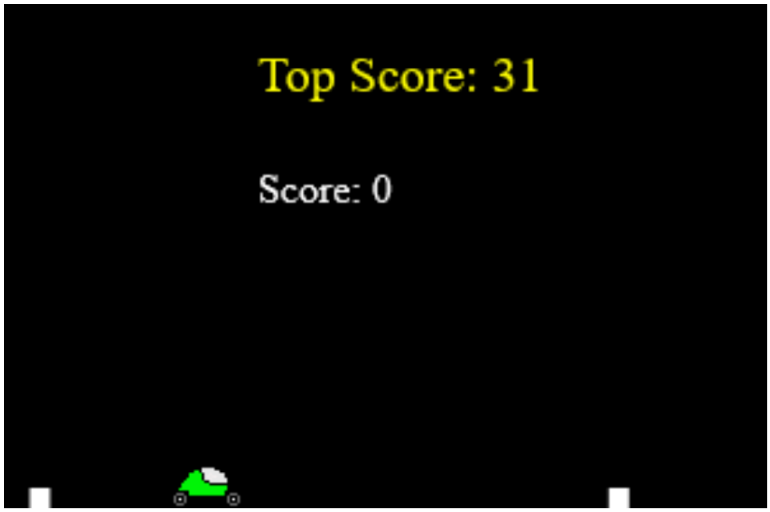

Figure 9. non-library

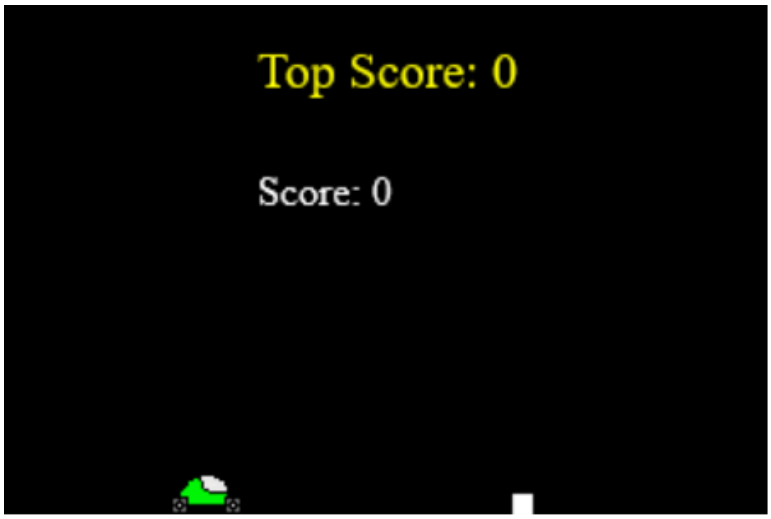

Figure 10. Phaser

Figure 11. ImpactJS

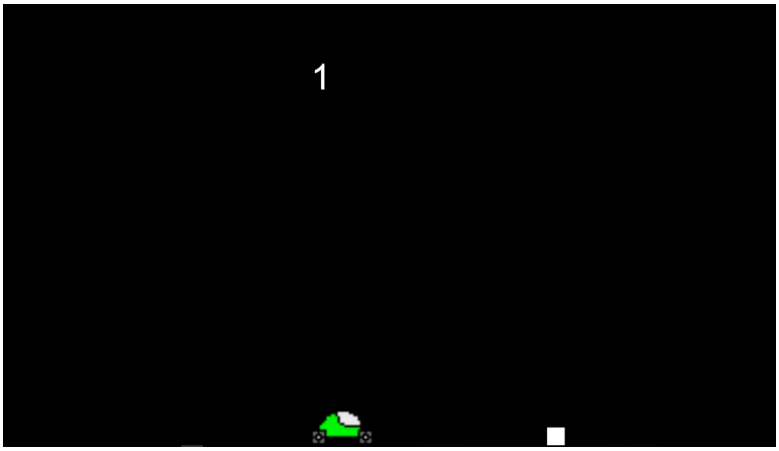

Figure 12. Construct 2

There are several different metrics that could be evaluated when comparing the game engines. Following metrics are reported during and after the implementation phase:

1. Amount of work needed on implementation
    a. Factor: time
    b. Factor: lines
2. Performance of the final product
    a. FPS
    b. Loading time
    c. Memory consumption
3. Subjective results
    a. Implementation difficulty relative to non-library
    b. Documentation support
    c. Bottlenecks

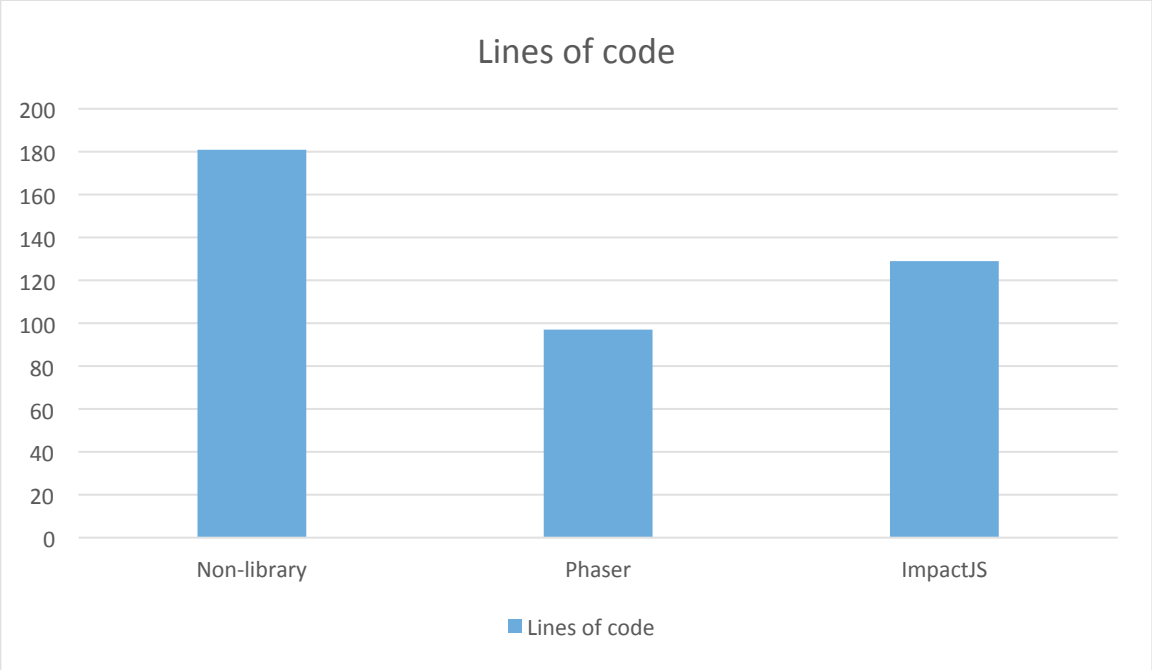

Figure 13. Lines of code produced during development

The amount of work needed can be derived from the number of lines written for the final product. It also answers the question: “How much extra work is needed to get the final

product?" Figure 13 shows the expected result; the non-library approach will net the most of the lines due to the need of doing everything from scratch. Additionally, Construct 2 uses 'drag-and-drop' styled GUI for its game development. Hence it is difficult to compare its building process to others by using lines. Empty lines are omitted from the results and only the lines added by the developer are counted.

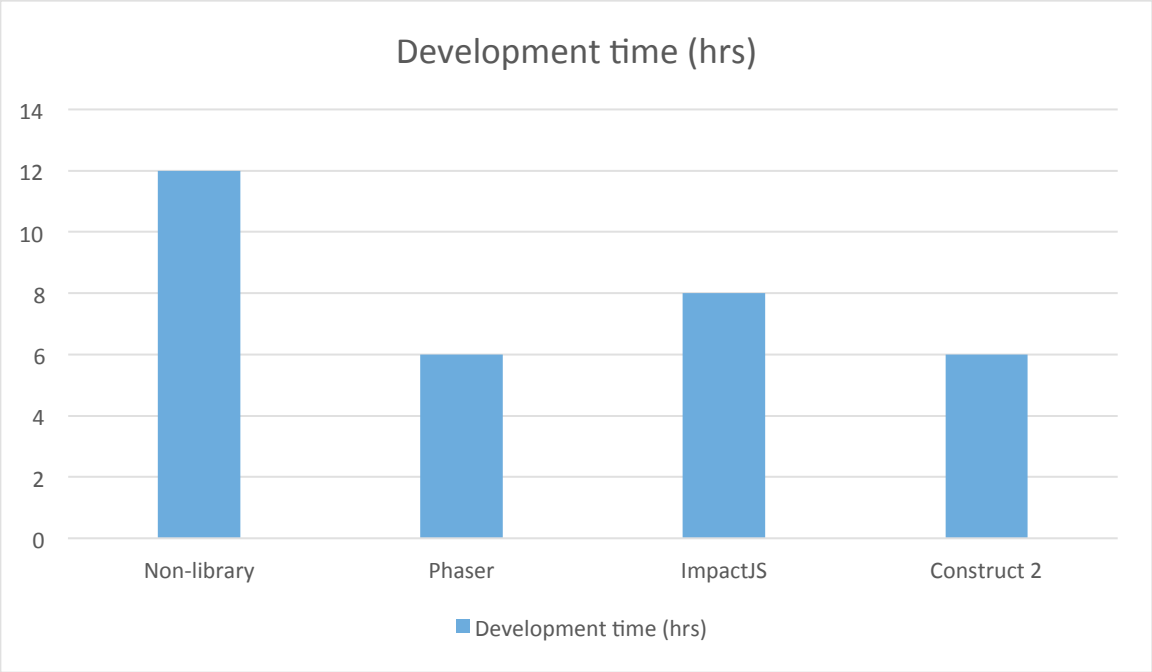

Figure 14. Total development time used for the development

The time used for development is recorded with an error margin of 30 minutes. The results can be seen in the figure 14. The results were in line with expectations with non-library solution needing more effort than others.

Construct 2 has fundamental differences from other engines which caused some confusion but its graphical UI and easy-to-use instructions help to alleviate that issue. ImpactJS and Phaser share a resemblance to each other in their structure and how they perform.

The game is benchmarked on a laptop with following specification:

OS:	Windows 10 Pro 64-bit (10.0, Build 10586)
Memory:	16384MB RAM
Processor:	Intel Core i7-6820HQ CPU @ 2.70GHz (8 CPUs)
Graphics Processing Unit (GPU):	Intel HD Graphics 530

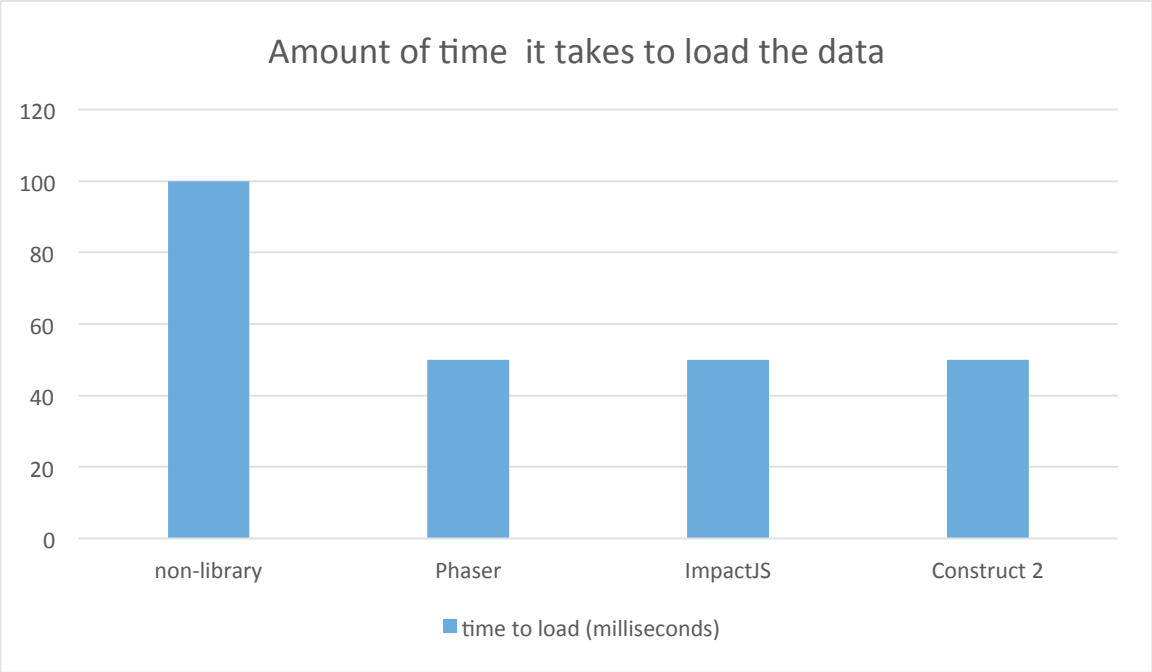

Figure 15. Loading times for the prototypes

The loading time for the game differ greatly depending on which solution is used as can see from figure 15. The non-optimized non-library solution is slower as expected.

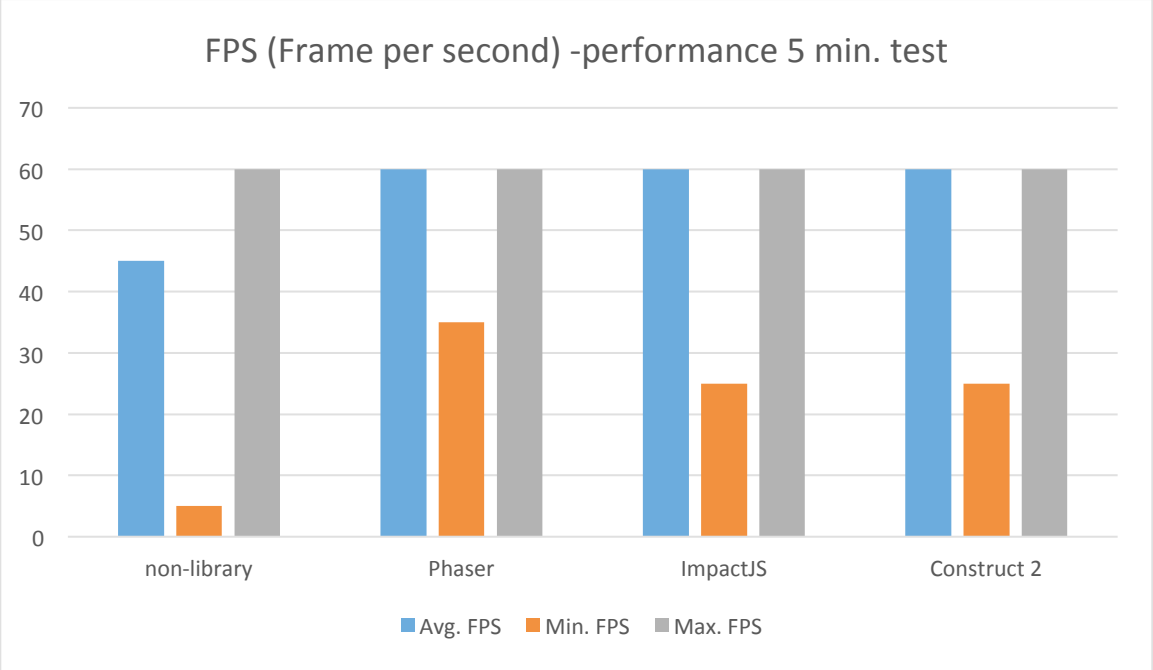

Figure 16. Performance test 1.

All the games utilize *requestAnimationFrame* instead of having *setInterval* based loop. The test laptop is equipped with relatively powerful processor being high-end of Intel's newest 'Skylake' -generation. That alongside the Intel HD 530 ensures that the average FPS stays at target FPS of 60. Non-library's non-optimized approach reduces the average FPS slightly while still being playable. Minimum FPS is an important factor for an enjoyable gameplay experience, hence it will be evaluated. If the frame rate fluctuates a lot, it causes big issues for gameplay experience. Figure 16 shows that the non-library solution it drops as low as 5 FPS. Games implemented with game engines managed to stay above 30 FPS for most of the time. Biggest drops on FPS are noted on the loading phase and when the garbage collector invoke its collection algorithms.

The games are also tested with a mobile device for FPS performance. The test devices are following:

Galaxy S6	Android 6.0.1
iPhone 6S Plus	iOS 10
iPhone 5	iOS 9.3.2
Nexus S	Android 6.0.1

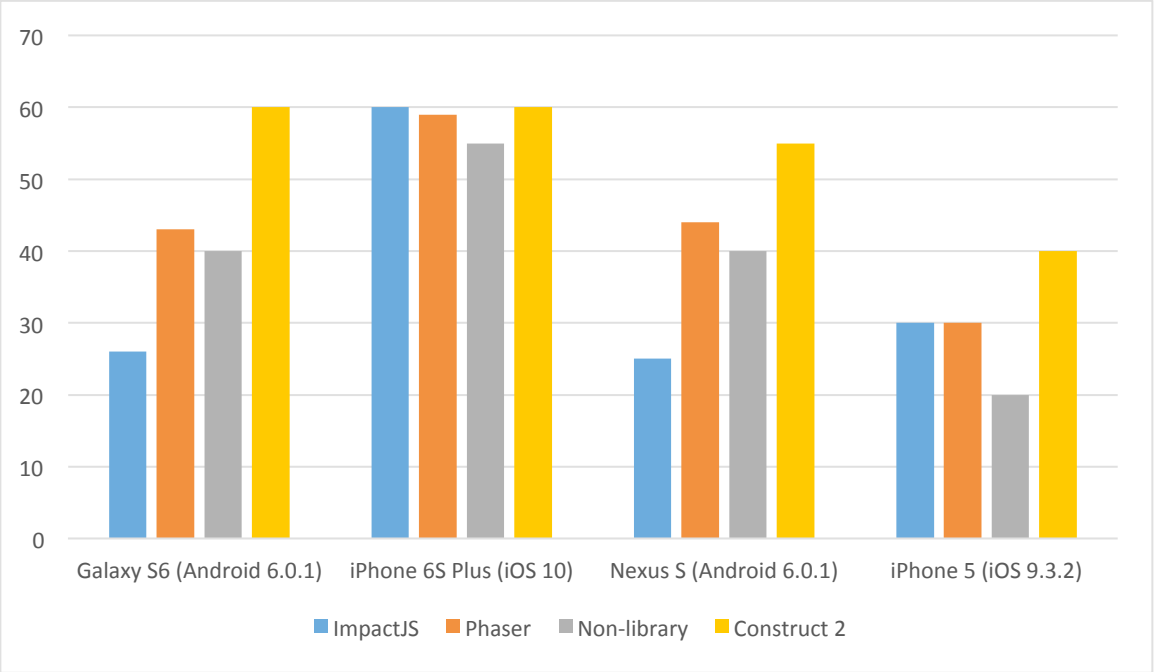

Figure 17. Performance test 2.

As can see from figure 17. on mobile devices the results are impressive as well. The game is played for 5 minutes, and all the solutions using frameworks manage to get past the 30 FPS threshold. Non-library solution slacks behind still managing to pull out playable frame rates. ImpactJS metrics show a drop to 30 FPS on Android devices.

The goal of the games is to display smooth animations that should feel and visually look good. Framerate of the films is 24 FPS while console games attempt to keep at least 30 FPS as the threshold for smooth gaming. Our results show that on a modern PC a simple endless

runner works perfectly above 30 FPS. The performance also depends on the complexity of the game and the hardware used.

Due to how the garbage collector works the memory usage is one cause of stuttering in the gameplay. The garbage collector does its cleanup process at certain thresholds showing a slight but visible stutter during the gameplay. The stutter is avoided by recycling the objects instead of deleting and creating objects, removing the need for garbage collection. The problem with object pooling is its high memory usage, due to the need of keeping everything in the memory. Memory usage is not a problem on a laptop that boasts 16GB of RAM (Random Access Memory), but on lower-end devices, the developer must be ready for compromises.

On the mobile devices, Phaser and Construct 2 stay at acceptable FPS. ImpactJS average on 26 FPS with Galaxy S6 which should be a high-end device. The iPhones perform a lot better in the tests. With the small differences that the projects have, it can be deduced that some small differences in the implementation are causing performance bottlenecks on Android devices.

Looking through the code, the biggest difference and the cause for problems can be ImpactJS's utilization of sprite fonts. Every single letter on the screen is a sprite that needs to be drawn. On iOS, this problem doesn't show up, due to better overall canvas performance. Changing the ImpactJS solution to use regular fonts instead of sprite fonts helps the game compare a lot better against Phaser and Construct 2 as can be seen in Figure 18. Regular fonts are DOM-elements, and hence the Android device can use hardware acceleration to render them, resulting in a better overall performance.

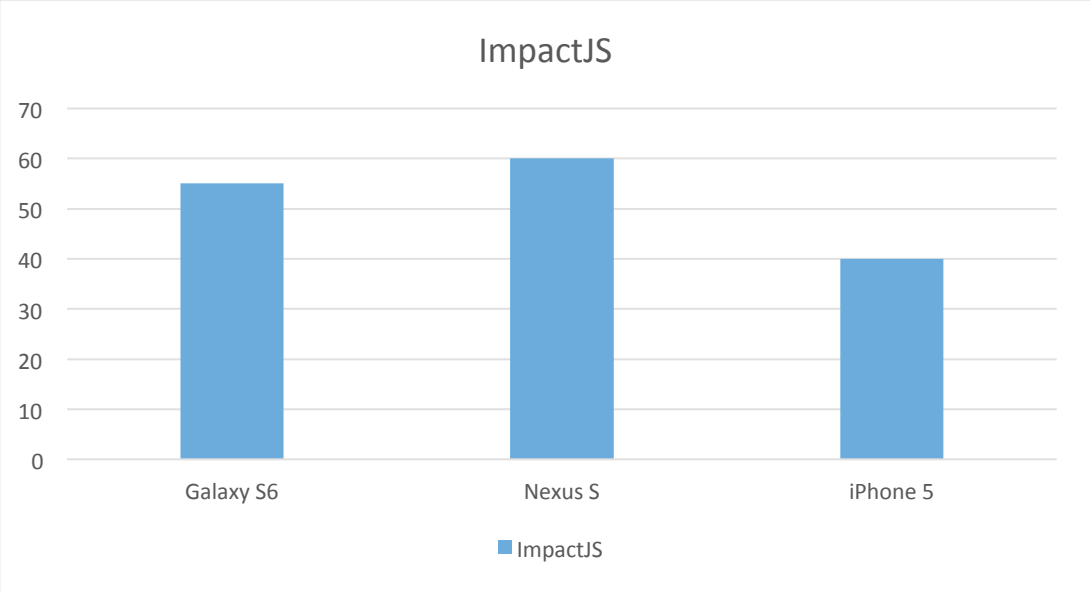

Figure 18. Performance test 3.

9 Discussion

One of the greatest benefits of using HTML5 to develop a game is its rapid development cycle. The results can be seen in seconds while completing a fully functional prototype takes few hours. The solution that doesn't use any external frameworks compares well against solutions that use game engines for development. As a prototyping tool, HTML5 frameworks seem to have immediate potential.

Are HTML5 game engines the best choice for game development? From the results, a gradual decline in the performance can be observed when changing from faster devices to slower ones. The previous observation is somewhat obvious, as the hardware and software are older, but considering that the game is just drawing a few images on the screen, it's still a noteworthy observation for scalability and portability. For a small and simple game, HTML5 is a potential choice. A good example for a small and simple game is Cut The Rope (ZeptoLab UK Limited., 2017). For bigger projects, the engine's scalability should be taken into consideration. Another good example of what can be achieved with HTML5 is CrossCode a game made by using ImpactJS. It is a complex and graphically exceptional game and it performs fluidly (Radical Fish Games 2016). It is just a matter of optimization and design.

Game engines often include a lot of functionality that may remain unknown for the developer. ImpactJS and Construct 2 both support in-app purchases and Construct 2 includes a basic implementation of networking. Even 3-D development is possible with WebGL and suitable plugins. This feature-set is expanded even more with the plugins that are built by other developers. These game engines are packed full with features that might not always utilized, but plugins can improve the capabilities of the game engine and make it an even more lucrative choice over the non-library solution.

The biggest differences between the frameworks are the most visible ones. Development on Construct 2 differs from all the other approaches greatly. Construct 2 uses GUI to build the game logic and attach the game events together. A game can be developed without any knowledge on programming while ImpactJS and Phaser require at least a basic knowledge of JavaScript. ImpactJS and Phaser have a lot in common based on how they render, update elements and on how you implement the games. The internal differences are minor and

mainly on the syntax. For example, ImpactJS uses inheritance to extend its base functions further. Its most important object, ig.Entity is used to do UI elements, player objects, enemy objects, projectiles and pretty much everything visible on the screen. You can also just draw plain images on the screen if you want.

In the Phaser, the objects are created by the user resulting in often less bloated objects. Instead of using class hierarchy for creating the objects, the objects are created using JavaScripts prototyping functionality. ImpactJS inheritance model can be a problem too. The objects are easily crowded with a lot of obsolete functionality that the child objects inherit. The bloating is especially true with the ig.Entity object. On the other hand, it is easy to introduce new functionality for a group of objects. Construct 2 uses graphical UI and allows the user to build the game without any knowledge of programming. The logic needs a while to understand if you are coming from an environment where everything is programmed, but it is one of the most popular HTML5 game engines for a reason. It is easy to use.

All the frameworks' communities are active with recent activity on the forums and with promises of new plugins to be released in the near future. From the core engines perspective, the ImpactJS is developed by one person in a closed environment, and the last version for ImpactJS was released late 2014. It is to be noted that the framework is still solid and if it works, there's little need to break and fix things.

Phaser and Construct 2, on the other hand, have regular updates to the frameworks. Better support for all the features needed in games, performance tweaks and continuous development around the engine is a sign of a healthy community. Continued support is problematic with ImpactJS as the engine is developed by a single person and that person is also working on other projects such as Ejecta. Due to the reason that ImpactJS is a commercial product and the development is in a closed environment, it is hard to recommend it to anyone. The author has personally done a lot of projects with the framework, and finds it a really good tool, but the lack of current interaction between developer and community is something that pushes new customers off. Phaser and Construct 2 are both free, although Construct 2 requires a 99€ investment to unlock all the extra features. Phaser is an open-source project powered by a custom-made version of Pixi.js with Construct 2 being somewhere in the middle of ImpactJS and Phaser on the openness.

Overall, the web is gradually getting faster. Even though theoretically it cannot ever catch up to the native performance in its current form its best benefits lie elsewhere. It is fast enough, easy to just pick up and start developing games. For prototyping, HTML5 is a great tool. There's potential that it can be used for bigger games as well. The tools are easy to use, the community is big due to easiness and popularity of HTML5 and examples can be found everywhere.

10 Conclusions

In this study, the goal was to find out the concrete benefits of using an HTML5 game engine in game development, over non-library HTML5 -solution. It could be argued that non-tool solutions give the developer more freedom since the developer has access to all the internal methods and knows the product inside-out. It could also be argued that tools let the user focus on the matters that have higher priority in game development. It was tested how much the tool-usage improves the efficiency of development. The 'soft' -aspects of the environment were compared since proper documentation can give developer enough control to modify any internal methods if needed. The second part was to compare the different implementation approaches in the study and see if they differ from each other greatly. Lastly, it was deduced if the HTML5 could be considered as a feasible tool for game development. HTML5 is popular and accessible; hence research for HTML5 game felt suitable.

All the approaches proved to produce games that are playable. The main differences were not in the performance but in the way the games are implemented. Construct 2 had the most glaring difference on having a graphical unit interface -based solution for development. Programming is only used to amplify and extend the existing functionalities. ImpactJS and Phaser have their structures and way of doing things. ImpactJS is built around its custom-built object structure, where every single object that is created inherits. For example, when creating a player -object, the object inherits the functionalities of "ig.Entity" an object that has values like "health" already initialized. Phaser, on the other hand, expects the developer to build this structure himself and only offers the tools. The non-library approach, of course, lacked any proper optimization but for a small project the differences in performance were negligible.

The benefits of using engines are the speed and efficiency of the development process. Developers can focus on the core functionalities and let the game engines to handle everything else. The developer still has relatively high control on the environment since all the game engines had good documentation. HTML5 can be considered as a feasible technology for smaller games for now. The browsers' rendering performance is constantly improving and the possibilities for bigger games are out there. The devices that are running these games

are improving as well, negating the fundamental performance differences between non-native and native products.

Game engines and other tools are important for the game industry. HTML5 and its frameworks should be seen as valid choices if entering game industry, due to its platform independency. Phaser, ImpactJS and Construct 2 were each built a bit differently but this just means that they are good tools for a different genre of games. ImpactJS, for example, includes tile-editor and basic properties like “health” that help with the development if the developer is aiming at doing 2-D platformers. Construct 2 and Phaser have other genres where they excel and hence the tools should be chosen based on the needs of the developer.

REFERENCES

Adult Swim Games, 2010. Robot Unicorn Attack [WWW Document]. [Accessed 17.1.2017]. Available http://games.adultswim.com//games/web/robot-unicorn-attack

Anderson, E.F., Engel, S., Comninos, P., McLoughlin, L., 2008. The Case for Research in Game Engine Architecture. Future Play '08 Proceedings of the 2008 Conference on Future Play: Research, Play, Share.Toronto, Ontario, Canada. November 3 - 5, 2008. New York, NY, USA. ACM. pp. 228–231.

Ayesh, A., 2013. Essential Dynamic HTML fast. 1st ed. London, New York: Springer.

Barus, A.C., Tobing, R.D.H., Pratiwi, D.N., Damanik, S.A., Pasaribu, J., 2015. Mobile game testing: Case study of a puzzle game genre. 2015 International Conference on Automation, Cognitive Science, Optics, Micro Electro-Mechanical System, and Information Technology (ICACOMIT). Bandung, Indonesia. October 29 - 30, 2015. IEEE. pp. 145–149.

Bethke, E., 2003. Game Development and Production. Plano, Texas: Wordware Publishing, Inc.

Blow, J., 2004. Game Development: Harder Than You Think. Queue 1, 28–37.

Bruce, D., 2015. Endless Runner (Concept). [WWW Document]. [Accessed 22.1.2017]. Available http://www.giantbomb.com/endless-runner/3015-7179/

Castro, E., 2002. HTML for the World Wide Web with XHTML and CSS, 5th Edition, Berkeley, CA: Peachpit Press.

de Sa, C., 2009. Basic Collision Detection in 2D - Part 1 [WWW Document]. [Accessed 17.1.2017]. Available http://devmag.org.za/2009/04/13/basic-collision-detection-in-2d-part-1/.

Deveria, A., 2017. Can I use... Support tables for HTML5, CSS3, etc [WWW Document]. [Accessed 25.10.2016]. Available http://caniuse.com/

Epic Games, Inc., 2017. Unreal Engine 4 is a professional suite of tools and technologies for building high-quality games across a range of platforms [WWW Document]. [Accessed 22.1.2017]. Available https://www.unrealengine.com/unreal-engine-4.

Flanagan, D., 2001. JavaScript: The Definitive Guide. 4th ed. Sebastopol, CA: O'Reilly Media.

Gartner, Inc., 2013. Gartner Says Worldwide Video Game Market to Total $93 Billion in 2013 [WWW Document]. [Accessed 18.1.2017]. Available http://www.gartner.com/newsroom/id/2614915

Geary, D., 2012. Core HTML5 Canvas: Graphics, Animation, and Game Development. 1st ed. Upper Saddle River, NJ: Prentice Hall.

GuangChun, L., Lu, W., Hanhong, X., 2003. A Novel Web Application Frame Developed by MVC. SIGSOFT Softw. Eng. Notes 28, 2, pp. 7.

Hi-Rez Studios, 2017. Tribes Ascend [WWW Document]. [Accessed 20.1.2017] Available http://www.tribesascend.com/.

Hover, C., 2014. Impact++ [WWW Document]. [Accessed 20.1.2017]. Available http://collinhover.github.io/impactplusplus/.

Irish, P., 2011. requestAnimationFrame for smart animating - Paul Irish [WWW Document]. [Accessed 18.1.2017]. Available https://www.paulirish.com/2011/requestanimationframe-for-smart-animating/

Janiszewski, M., 2014. HTML5 Cross-Platform Game Development. Master's Thesis. Vrije Universiteit Amsterdam.

Kashyap, P., 2015. Investigation into the use of HTML 5 game engines to create a responsive social educational game for children. Master's Thesis. Charles Darwin University.

Kasurinen, J., 2016. Games As Software: Similarities and Differences Between the Implementation Projects. CompSysTech '16 Proceedings of the 17th International Conference on Computer Systems and Technologies 2016. Palermo, Italy. June 23 - 24, 2016. New York, NY, USA. ACM. pp. 33–40.

Kasurinen, J., Smolander, K., 2014. What Do Game Developers Test in Their Products? ESEM '14 Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. Torino, Italy. September 18 - 19, 2014. New York, NY, USA. ACM. p. 1:1–1:10.

Kasurinen, J., Strandén, J.-P., Smolander, K., 2013. What Do Game Developers Expect from Development and Design Tools? EASE '13 Proceedings of the 17th International Conference

on Evaluation and Assessment in Software Engineering. Porto de Galinhas, Brazil. April 14 - 16, 2013. New York, NY, USA, ACM. pp. 36–41.

KinematicSoup Technologies Inc., 2016. Game Development Workflow [WWW Document]. [Accessed 20.1.2017]. Available http://www.kinematicsoup.com/news/2016/10/26/game-development-workflow

KinematicSoup Technologies Inc., 26.10. Game Development Workflow Part 2: Production and Post-Production [WWW Document]. [Accessed 21.1.2017]. Available http://www.kinematicsoup.com/news/2016/11/10/game-development-workflow-part2

Motley Fool, 2016. The State of the Video Game Industry Today - [WWW Document]. [Accessed 18.1.2017]. Available http://www.fool.com/investing/2016/06/21/the-state-of-the-video-game-industry-today.aspx

Mozilla Developer Network, 2017a. window.requestAnimationFrame() [WWW Document]. [Accessed 25.1.2017]. Available https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

Mozilla Developer Network, 2017b. Introduction [WWW Document]. [Accessed 20.1.2017]. Available https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction

Mozilla Developer Network, 2017c. Memory Management [WWW Document]. [Accessed 20.1.2017]. Available https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

Nguyen, D., 2013. .GEARS | Flappy Bird [WWW Document]. [Accessed 19.1.2017]. Available http://dotgears.com/apps/app_flappy.html

Nguyen, Q., 2013. HTML5 Canvas and CSS3 by Examples: Graphics, Games and Animations. The United States of America: RAMACAD INC.

Ocariza, F.S., Pattabiraman, K., Mesbah, A., 2015. Detecting Inconsistencies in JavaScript MVC Applications. 37th IEEE International Conference on Software Engineering. Florence, Italy. May 16 - 24, 2015. NJ, USA. IEEE Press Piscataway. pp. 325–335.

Photon Storm Ltd., 2017. Phaser - A fast, fun and free open source HTML5 game framework [WWW Document]. [Accessed 22.1.2017]. Available http://phaser.io/

Pilgrim, M., 2010. HTML5: Up and Running: Dive into the Future of Web Development, 1 ed. Sebastopol, CA: O'Reilly Media.

Pirker, J., Kultima, A., Gütl, C., 2016. The Value of Game Prototyping Projects for Students and Industry. GJH&GC '16 Proceedings of the International Conference on Game Jams, Hackathons, and Game Creation Events, San Francisco, CA, USA. March 13, 2016. New York, NY, USA. ACM. pp. 54–57.

Politowski, C., Fontoura, L., Petrillo, F., Guéhéneuc, Y.-G., 2016. Are the Old Days Gone?: A Survey on Actual Software Engineering Processes in Video Game Industry. GAS '16 Proceedings of the 5th International Workshop on Games and Software Engineering. Austin, Texas. May 14 - 22, 2016. New York, NY, USA. ACM. pp. 22–28.

Rettig, P., 2012. Professional HTML5 Mobile Game Development, 1 ed. Indianapolis, Indiana: Wrox.

Santanchè, A., Boulanger, R., Viana, G., Panaggio, R., Melo, B., Aboud, H., 2013. Mobile Development Using Web Technologies Focusing on Games. WebMedia '13 Proceedings of the 19th Brazilian Symposium on Multimedia and the Web. Salvador, Brazil. New York, NY, USA. ACM. pp. 13–14.

Scirra Ltd., 2017. Make Your Own 2d Games With Construct 2 [WWW Document]. [Accessed 22.1.2017]. Available https://www.scirra.com/construct2

Scirra Ltd., 2016. The Great HTML5 Gaming Performance Test: 2016 edition - Scirra.com [WWW Document]. [Accessed 22.1.2017]. Available https://www.scirra.com/blog/175/the-great-html5-gaming-performance-test-2016-edition

Steamspy, 2017. 2016 - Year Stats - [WWW Document]. [Accessed 23.1.2017). Available http://steamspy.com/year/2016

Suhonen, M., 2016. Keräilykorttipelin toteutus HTML5-ja Javascript-tekniikoilla. Master's Thesis. Lappeenranta University of Technology, School of Business and Management, Computer Science.

Szablewski, D., 2017. Impact - HTML5 Canvas & JavaScript Game Engine [WWW Document]. [Accessed 23.1.2017]. Available http://impactjs.com/

Toll, D., Olsson, T., 2012. Why is Unit-testing in Computer Games Difficult? CSMR ' 12 Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering. March 27 - 30, 2012. Washington, DC, USA. IEEE. pp. 373–378.

Unity, 2017. Unity - Game Engine [WWW Document]. [Accessed 23.1.2017]. Available https://unity3d.com

Vaughan-Nichols, S.J., 2010. Will HTML 5 Restandardize the Web? Computer, 43, 4, pp. 13–15.

W3C, 2016. Cascading Style Sheets [WWW Document]. [Accessed 20.1.2017]. Available https://www.w3.org/Style/CSS/

W3C, 2014. HTML5 [WWW Document]. [Accessed 15.11.2016). Available https://www.w3.org/TR/html5/

W3C, 1999. HTML 4.01 Specification [WWW Document]. [Accessed 15.11.2016]. Available https://www.w3.org/TR/html4/

w3schools, 2017a. Introduction to HTML [WWW Document]. [Accessed 23.1.2017]. Available http://www.w3schools.com/html/html_intro.asp

w3schools, 2017b. JavaScript Tutorial [WWW Document]. [Accessed 23.1.2017]. Available http://www.w3schools.com/js/

w3schools, 2016. Browser Statistics [WWW Document]. [Accessed 26.1.2017]. Available http://www.w3schools.com/browsers/

Weeks, M., 2014. Creating a Web-based, 2-D Action Game in JavaScript with HTML5. ACM SE '14 Proceedings of the 2014 ACM Southeast Regional Conference. Kennesaw, Georgia. March 28 - 29, 2014. New York, NY, USA. ACM. pp. 7:1–7:6.

Williams, J.L., 2012. Learning HTML5 Game Programming: A Hands-on Guide to Building Online Games Using Canvas, SVG, and WebGL. Boston, MA. Pearson Education, Inc.

ZeptoLab UK Limited., 2017. Cut the Rope games, Om Nom and Nommies Official Website [WWW Document]. [Accessed 25.1.2017]. Available http://www.cuttherope.net/