TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Informatics

Chair of Knowledge-Based Systems

Methodology for Web Application Performance Analysis and Methods to Improve it

Master thesis

Student: Alexey Gudz

Student code: 132456

Supervisor: Jaak Henno

Tallinn
2015

Author’s declaration.

I confirm that this work was composed personally by me and it has never been presented for defense by anyone else. Other authors’ works, key positions and other literature sources that were used, are shown.

_____ _____

(Date) (Signature)

Annotation.

In this master thesis I will be explaining how a browser's internal logic works, how to measure the performance of your web application and provide advices on how to improve it. The idea appeared from work experience as a mobile web developer - too many times the lack of understanding of web browser logic resulted in development dilemmas and inefficient, poor performance apps. An existing live web game is used as a practical example for the in-depth performance analysis. The goal of the thesis is to acquaint the reader with a web browsers structure and mechanics, its rendering and script engines, explain how to use a browser's developer tools for performance analysis and demonstrate this using a real-life example - an online game developed by the author and the company he works for.

Annotatsioon.

Selles töös uuritakse brauseri sisemist loogikat ja veebi rakenduste jõudluse analüüsimist ning pakkutakse soovitusi selle oprimeerimiseks. Idee tuli mobiilse veebi arendaja töö kogemusest. Brauseri loogika teadmatuse tulemusena on tihti dilemmad ja halvasti toimivad rakendused. Kasutatakse olemasolevat veebi mängu praktilise näidisena sügava jõudluse analüüsimiseks. Töö eesmärk on tutvustada lugejat veebi brauseri struktuuriga ja mehaanikaga, selle renderdamise ja skriptimise mootoridega, seletada kuidas kasutada brauseri arendaja vahendeid jõudluse analüüsimiseks ning demonstreerida seda reaalsel näitel, milleks on autori ja tema kolleegida poolt arendatud online mängu.

List of Figures.

List of acronyms and abbreviations

CPU	central processing unit
GPU	graphics processing unit
HTML	hypertext markup language
CSS	cascading style sheets
SMIL	synchronized multimedia markup language
UI	user interface
HTTP	hypertext transfer protocol
URI	uniform resource identifier
URL	uniform resource locator
DOM	document object model
CSSOM	cascading style sheets object model
API	application programming interface
WTF	web template framework
FPS	frames per second
MB	megabytes
QA	quality assurance

1. Table of Contents

INTRODUCTION

A web browser is one of the popular programs being used by most people today. It is also one of the most complex ones. This program processes and presents an impressive amount of different media such as text, images, videos, sounds, animations, interactive content and even games.

To be able to do that, the browser uses extremely complicated underlying logic of processing, translating, interpreting input content and interacting with the memory, the CPU, the GPU, the soundcard, 3rd party plugins, all to present the content to the end user to enjoy.

Web industry has been prospering for quite some time already and with the introduction of HTML5, multimedia capabilities of websites have reached a new level. Web game development in particular has experienced a remarkable increase in popularity as a result of new canvas technology.

It isn't just the websites that have been improving – the platforms that run them have also undergone changes. Nowadays, a web page can be accessed through many different browsers on wide array of devices with fundamentally different hardware and operating systems.

The large diversity of technologies used by the browser have also been growing very fast. Not only HTML but also CSS, JavaScript and SVG are all being improved at a remarkable rate. The problem, however, is that these technologies are developed independently from each other, often resulting in overlapping functionality. This complicates things and confuses developers. For example, one may write an animation using only HTML canvas. He can also do it purely in JavaScript with the help of "RequestAnimationFrame". Similar results can be achieved with SVG and SMIL. The latest standard for style sheets, CSS3, also offers animation functionality – "keyframes".

With so many options, all having different syntax and unique behavior, a developers become baffled when thinking of which way to animate things would be best for him and produce smoothest experience. This is why understanding how the web browser translates these technologies is crucial if we want to write high performance web apps.

Web pages are becoming more complex, interactive and rich in media while the hardware and software running the page – more diverse. Performance has become a critical topic. Whether your website or web game is enjoyable for the viewer or not heavily depends on how smooth it runs on their machine.

Unfortunately, most browsers have different underlying logic - they utilize different rendering engines for translating input HTML and media into an on-screen image, use different JavaScript engines for interpreting and executing JavaScript code.

Google Chrome web browser with its WebKit rendering and V8 JavaScript engines is the most popular web browser to date and has been chosen for study in this thesis.

The topic of the thesis comes from real issues the author stumbles into in his job as a

game developer. The task is to make mobile web games that provide a smooth enjoyable play experience. Smartphones and tablets, however, have limited hardware capabilities and often fail to maintain stable decent frame rate. For example, an image fading it while multiple animations are playing produces hiccups on lower tier devices.

The goal of this research work is to understand what transformations a web page undergoes in Chrome browser to reach the client's screen. After the process is clarified, methods on how to evaluate and optimize the performance, make the animations smoother and the page rendering earlier, will be explained.

This thesis involves an analysis of an existing complex web game developed by the company the author works in and took part in development. As the game was developed in HTML5 for different devices, mostly mobile, the performance and smoothness of the game is of vital importance making it a perfect practical example for this research.

2. WHAT THIS IS ABOUT

Practical significance

The author of this thesis is a web developer with some years of experience in the field, working for a company that produces web games for mobile devices, such as smartphones and tablets.

The problems discussed in this research work are real life issues that arise for the author as well as his work colleagues while programming those games on a daily basis.

In addition to the problems being practical real life issues, the solution will also find use in the company the author works in. He and his colleagues will benefit from the analysis impacting the quality of future games and projects.

The problem

There are a few problems addressed in this thesis.

First of all, web developers often lack solid understanding of behind-the-scenes browser mechanics. When a programmer does not fully comprehend how the program runs his code, it is very difficult to write it efficiently. For web pages it may mean that the loading times will be unnecessarily slow or the whole experience not smooth. Programmers should understand the browser engine and write-code that runs well.

The second issue or "knowledge gap" is the inability to analyze the performance of a web page or app. A developer, author included, needs to know how to use the available tools to understand how well his creation runs, what are the weakest links that need optimizing.

The other problem is something many IT companies face - the lack of time or resources to allocate to improving the performance of their web apps. The deadlines are usually strict and everything is developed in a hurry. There is often no time for learning, researching, evaluating the performance or optimizing the code. This is why it is a good opportunity for the author to learn and analyze the performance of the company's game.

Understanding Chrome browser mechanics and being able to analyze target web application is highly beneficial for any single developer or company involved in making graphically intensive web applications or web games.

The solution

The solution to the problem of suboptimal inefficient code lies in raising the developer's awareness of how browsers work, the technologies behind it as well as learning how to measure performance and find flaws in it. This knowledge is applied

in the practical part, the game analysis. The result is a methodology to analyze browser applications performance and common ways to improve it.

To sum up, this research work is about understanding how browser works, being aware of what can be done to optimize performance, knowing how to measure it and applying it in practice The work will benefit the author and the company in future projects as well as improve the skill of the author and the readers.

3. BROWSER LOGIC OVERVIEW

Before we get down to Chrome browser specifics, a general explanation of browser logic, its components and processes is necessary. We will try to understand how the page rendering happens and how we can improve it.

What is the browser

A web browser is an interpreter, a program that requests and receives a web page, parses it into a structure it can work with, and presents to the user.

Figure 1: Top 5 most used browsers [1]

How does it know how to present the page correctly? The rules are actually standard HTML and CSS specification which are written and maintained by the World Wide Web Consortium. [2]

It may sound simple on the outside, but the underlying logic is fairly complex. There are so many input formats and technologies the browser has to handle: texts, images, video, audio, style sheets, scripts, SVG, canvas. How does all of this end up as instructions to the GPU, CPU, soundcard?

This is complex question and requires a more low-level understanding of browser structure.

Architecture and flow

Generally, a web browser consists of the following components:

- User Interface – the browser's interface, e.g. address bar, buttons, menus.
- Browser engine – an intermediary component handling communication between the UI and the Render engine. Handles plug-ins and extensions.
- Rendering engine – main component responsible for displaying the requested web page on screen.
- Networking layer for requesting HTTP content.

- UI backend handles the drawing of common UI elements such as windows, buttons, text controls.
- JavaScript engine is needed for interpreting and executing JavaScript code.
- Data persistence layer provides mechanism for saving data. E.g. cookies, html5 localStorage,

This is illustrated in the following diagram:

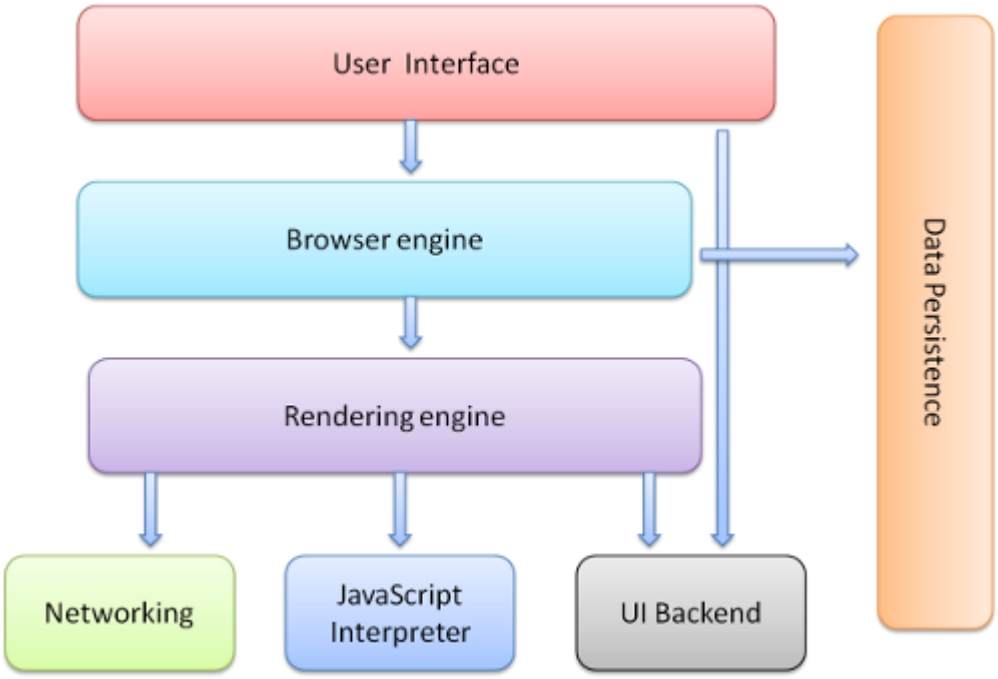


Figure 2: The structure and components of a web browser [2]

What we are going to be focusing more on in this thesis is the Rendering engine part.

After the networking component has made an HTTP request for the specified URI and HTML has been downloaded, the parsing of the HTML into a Document Object Model tree by the render engine is started. [2]

Styling information is also parsed by the same rendering engine to create a CSS Object Model tree. As the DOM tree is traversed, relevant CSS is applied. Together these DOM and CSSOM trees are merged to form the Render Tree (also called the "Frame tree" in Mozilla logic). This tree starts with the Viewport object and contains sequential blocks with styling information. [2]

At this point, we know which nodes should be drawn as well as their styles, but not their size or positions. This is computed using geometrical calculations in the next phase – the Layout (or "reflow") stage. [2]

The flow is concluded after the pixels are actually painted on screen. All of this is done by the rendering engine.

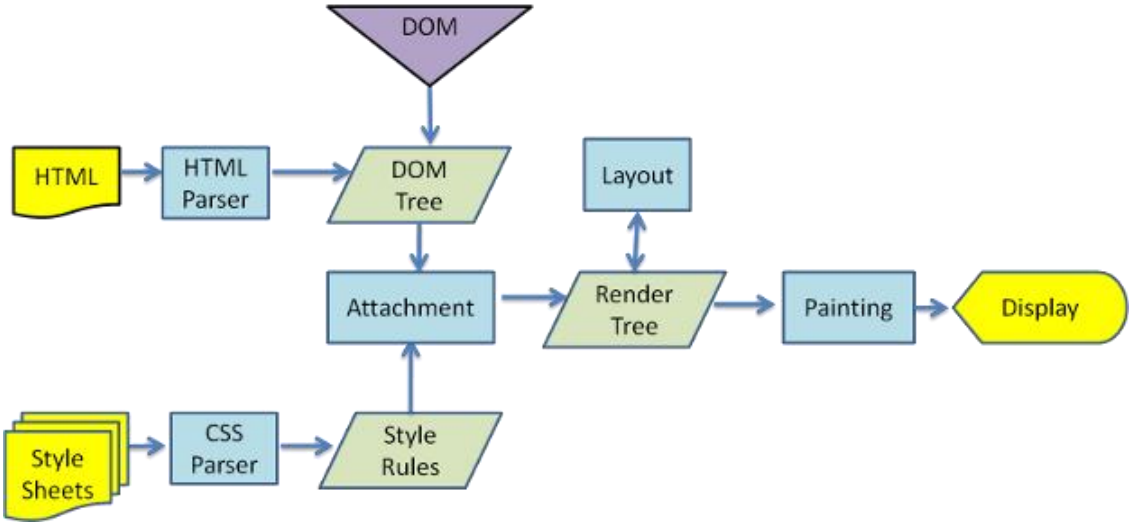


Figure 3: The main flow of page rendering [2]

This flow from the downloading of your page to the moment it is rendered, is actually called the "Critical rendering path". It is something very useful for a developer to understand, and even more useful if you know how you can optimize it.

Multimedia dilemmas

One of the many complexities of the web browser is the wide arrange of **independently** developed technologies and formats available.

Figure 4: Different browser multimedia technologies

The problem with that is the introduction of dilemmas for web developers. Here are some of them:

- Should I use Flash or HTML5?
- Is GIF animation faster than video?
- Should I use spritesheets with CSS or gifs?

- CSS animations vs transitions
- Is Canvas faster than SVG?
- Do DIVs have better performance than Canvas?
- Should I do my animation in CSS or JavaScript?
- What is the difference between WebAudio and HTML5 audio?

The big question in most of such dilemmas is usually "*Which is faster*?

It is of course also important to understand all the technologies your site utilizes, as each of them have their own cons and pros. However, what I want to stress the most in this thesis is the importance of knowing how to **measure the performance** of your web app. This will be covered in sections "Using the tools" and "Analyzing our game".

What can we optimize

The loading of the page

The first thing a user pays attention to when loading your web page or game is how long it takes for the browser to display it. This creates the first important impression. If the page loaded quickly, it will surely be a pleasant one. If the page is too slow the user may even leave it before checking the content. No one likes to wait. User satisfaction is heavily dependent on page speed.

If we sum up what happens when we enter a URL, we get the following chain of events:

- User inputs the URL
- Browser downloads the HTML
- Browser scans the HTML
- Browser finds links to images, CSS, JavaScript files
- Browser starts downloading those
- Browser does not start rendering the page until CSS and JavaScript are loaded
- Only then the rendering starts

Many developers are totally unaware of what the steps the browser takes to get that initial page view rendered, the critical rendering path, and have no idea about what can be done to speed up that process.

There actually are some simple things that can be done to dramatically reduce the loading times. The easiest tips are to stop non-critical resources from blocking the

Non parser blocking JavaScript – by default, if JavaScript is encountered, the browser stops the construction of the DOM until the script is parsed. The obvious reason is that the script may affect the DOM structure. However, if we know that the JavaScript is not absolutely critical in initial view, we can add a property called "async" to the script tag.

<script src="ourscript.js" async></script>

This tells the browser that the HTML parsing and DOM construction should not be blocked while this script is loading.

Deferring or lazy-loading the resources – similarly to "async" property, if we know that our styles, scripts or images are not crucial in initial page view, we might as well postpone their loading. Lazy loading a resource usually means we start loading it only when we reach that part of the page. Deferring basically means delaying until the page rendering is complete. [3]

Combining CSS and JavaScript files – more HTTP requests means slower loading times. By combining our files into one CSS and one JavaScript file, we remove some network overhead latency. It is also possible to entirely inline all our CSS and scripts directly in HTML, but it is a good idea only if the styles and scripts are small.

Repaints and reflows

Repaints happens when an element's appearance changes and is a costly operation. Reflow is the calculating of the geometry and positions of the elements in the DOM and an even more expensive process. To get smooth browsing experience, the number of reflows should be minimized. For that, we need to know what triggers it.

In most cases, reflows are caused by changing the CSS or DOM, class name, resizing and scrolling the window, changing the content on the page.

That is a lot of easy to trigger cases. It may make more sense to focus on optimizing the speed of the reflow. For that, some measures can be taken:

- Minimizing the depth of the DOM makes reflows faster [4]
- Group DOM and style changes so they are done in 1 reflow
- Sacrifice animation steps, that is, updating the frames less
- If animations are used in a container, set its position to "absolute" or "fixed". This way it will not affect the page layout.
- Avoiding complex CSS selectors, those are slow
- Avoid tables as they are much slower than block layouts. [5]

Using spritesheets

We have previously mentioned network overhead and how less HTTP requests is better. This can also be applied for images.

One useful although, for some reason, not very commonly used practice is to merge

multiple separate images into single “spritesheet” files. This can be used for grouping many small images, such as icons together or sequential frames of an animation.

Apple advises using spritesheets as one of image delivery best practices. [6]

Here is an example of a spritesheet, taken from Apple’s developer blog:

Figure 5: Example of a spritesheet image

By packing 20 images into 1 large, 19 slow network HTTP requests have been avoided.

There is one thing to bear in mind when creating spritesheets. If a web page or app is aimed at mobile devices, memory is a factor that needs to be considered. The author and his colleagues work with spritesheet images and mobile devices on a daily basis. One problem they have run into is their web game crashing due to some spritesheet image being too large for the device. This is due to memory limitation. The megapixel size of the image matters instead of the file size, as it gets loaded into memory for processing. Avoid large images.

To sum it up, spritesheets can be a nice way to optimize your web app performance, but some consideration needs to be taken.

4. CHROME

Introduction to Chrome

Chrome is a freeware web browser developed by the technology giant Google, first introduced on the PC back in 2008. It has been the leading browser for the last few years, easily overshadowing all of its rivals.

2015	Chrome	IE	Firefox	Safari	Opera
March	63.7 %	7.7 %	22.1 %	3.9 %	1.5 %
February	62.5 %	8.0 %	22.9 %	3.9 %	1.5 %
January	61.9 %	7.8 %	23.4 %	3.8 %	1.6 %

The browser is available on multiple platforms:

- Windows
- Mac
- Linux
- iOs
- Android

Most of the code is available in an open-source projected called “Chromium”.

Chrome has been chosen for closer study in this thesis as the author uses it in both home and work environments, and because Chrome offers amazing tools for developers to help with performance measuring of a web page, which is what the goal of this thesis is.

WebKit and Blink

As discussed in the Browser Internals Quick Overview section, each browser utilizes a rendering or in other terms, “layout”, engine. Google Chrome uses “Blink”.

Blink is actually a more recent fork of the popular WebKit layout engine and was

implemented in Chrome only starting version 28. Up until version 27, original WebKit was used. The only exception is the iOs platform where Chrome still uses WebKit. The reason for this is an Apple iOs app policy requiring all browsers to use WebKit rendering engine.

The reason why Google decided to dump the popular engine for its own version was to simplify and optimize the code. WebKit contains millions of lines of outdated redundant code which makes it unnecessarily complex to maintain. There are also other reasons, like having better flexibility to add own features. Blink is being developed as part of the Chromium project.

We will not be going too deep into the technical aspects of WebKit as it is a wide and complex area by itself. Briefly speaking, WebKit consists of different components:

- WebCore, which is the core library responsible for handling HTML, DOM, rendering and layout, networking, multimedia and more.

- The Javascript engine – a virtual machine that interprets and executes JavaScript. In WebKit, the default Javascript engine is JavaScriptCore. It is replaced by Google's V8 engine Chromium based browsers.

- The API – application programming interfaces referred by the browsers called WebKit and WebKit2

- Web Template Framework, or WTF in short – a library containing utility data used by the above components

- Different bindings

The image below should illustrate the structure:

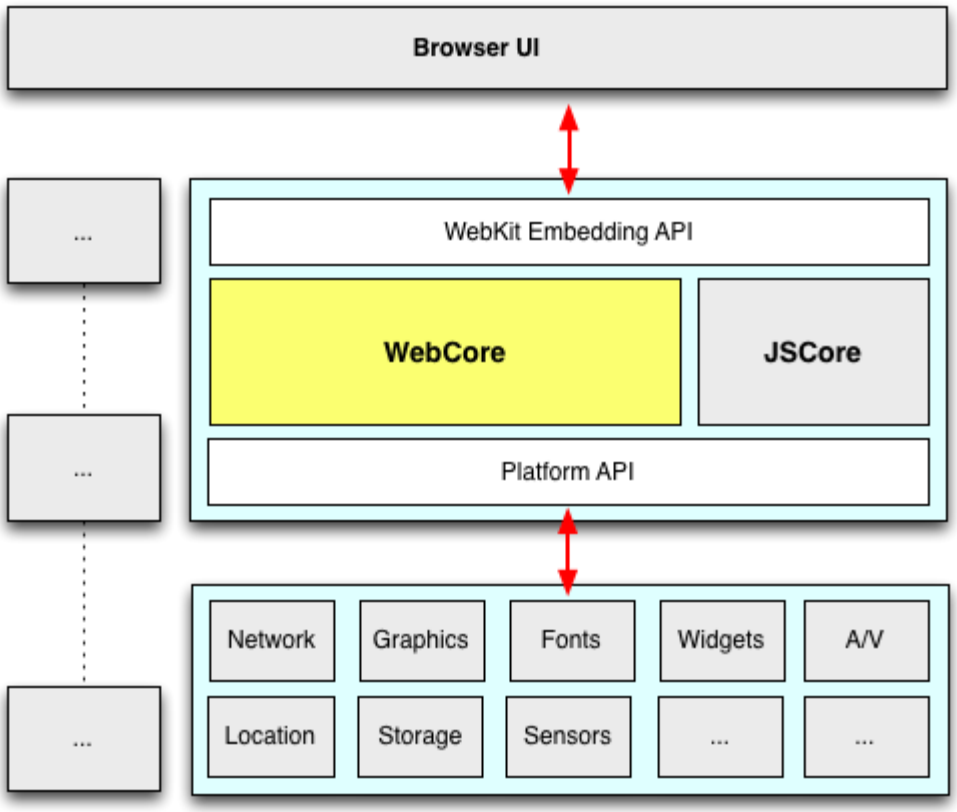


Figure 6: The internal structure of WebKit engine [7]

It is also worth mentioning that there exist many other ports of WebKit for many different platforms. Although the core "WebCore" component is the same, a lot of platform-dependant aspects differ for each port, such as handling networking and drawing on screen logics.

Chrome Developer Tools

The more important goal of the thesis is to help understand how to perform an analysis of a web application. For that we need effective tools. Luckily, Google offers a nice set of tools called DevTools as part of Chrome browser.

We will not be covering all of them in detail, but it is useful to have a quick overlook:

- Elements – this is where you inspect the DOM of the page, being able to check any element, its styles and change it in real time.
- Network – the screen where you can see all the network requests, resources loading, timings etc.
- Sources – your main tool for debugging. Here you see all the files used, can analyze the code and set breakpoints for debugging.

- Timeline is your best friend for performance analysis. It is here where you can monitor all the actions taking place and analyze them.

- Profiles is another valuable tool to evaluate performance. Allows you to record and check the CPU and memory usage of your web page or app.

- Resources is the place to check local data sources such as cached resources, cookies, local databases.

- Audits – yet another helper in improving your page performance. It runs a scan and offers suggestions such as removing unused CSS rules and more.

- Console is your log as well as a command entering screen.

All of these tools are very useful and well documented by Google in their developer's page. [8]

5. USING THE TOOLS

The main developer tools we will look at will be the Timeline, Profiles, Network and the Audit. Timeline is the most important one and this is where we start.

Understanding the Timeline

Timeline is a powerful recording tool for analyzing performance that displays a lot of useful information regarding our page. If we select all the checkboxes – Causes, JS Profiler, Memory and Paint, we get the following screen:

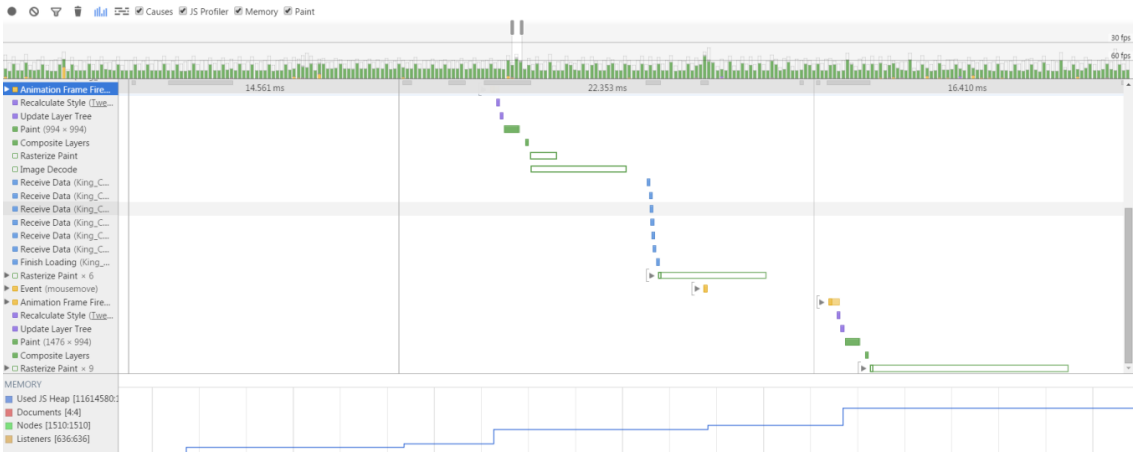


Figure 7: The Timeline screen

Using it is simple - record the action on your page you want analyzed, understanding it – not so much. There are a lot of sections of information which can be confusing at first. Let us look at the most important ones:

Frames view- enabling it allows us to see the frame rate of the browser while performing the measured action. This graph is basically the rendering performance of our page. We can see the spikes in performance and get information on what causes. If during these spikes we receive less than 30 FPS, they are worth looking into.

The event screen – the main area of the tool, displays all the events and function calls that happened in the timeframe we specify along with the time they took to complete. On the left there is the **Records** panel which displays the names of the events and functions we are looking at. We can also enable the "Flame chart" view to get a combined screen.

Below it is the **Memory** part of the screen which shows us a graph of changes in Heap memory size, Documents, Nodes and Listeners in the timeframe that we are studying.

Bottom most section contains a **Summary** and a **Paint Profiler**. The former display a pie chart showing how much time was spent on Scripting, Rendering, Other processes and idling in the selected timeframe. The latter - a helpful screen with its own timeline

that shows the order in which our elements were painted on the screen.

All the information regarding the meaning of events, the colors and more can be found in Google's own tutorial of the Timeline. [9]

Network resource loading and events

The Network screen is the place where we track all of the networking requests our page or app has been performing. We can either record while performing specific actions that we want to study or check the loading of resources on initial page load.

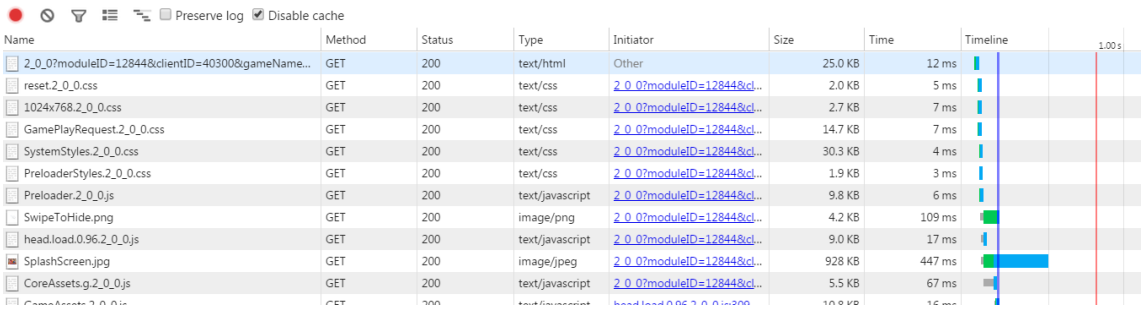


Figure 8: The Network tool

In the Network tool, we are able to see how much time (in milliseconds) each of our resources took to download and find what's slowing our initial page loading down. It is useful to check the "Disable cache" flag to get accurate results in assessing our page load time.

Since we are concerned about the performance and the speed of our app, we will look at 2 interesting events shown in the Network screen: *domContentLoaded* and Load events.

Load - this event is fired when all of our resources have finished downloading and is therefore, less useful.

domContentLoaded - a very important event for a web developer, signifying that our initial HTML has been fully loaded and parsed. This is what we usually hook initialization logic to. The goal is to make this event fire as early as possible. The sooner - the earlier our user will see the initial page.

In the "The loading of the page" section we covered the techniques to optimize page speed. Using them we actually cause the *domContentLoaded* event and our initial page render to appear earlier.

To conclude, the Network screen is ideal for checking our *domContentLoaded* event when optimizing our page speed and for tracking all network requests, possibly finding things that slow us down.

Reading the CPU and Heap profiles

Using Chrome's profiler to get CPU and memory usage of your page is as easy as pressing a button. What's important is understanding the results that the profiler gives us.

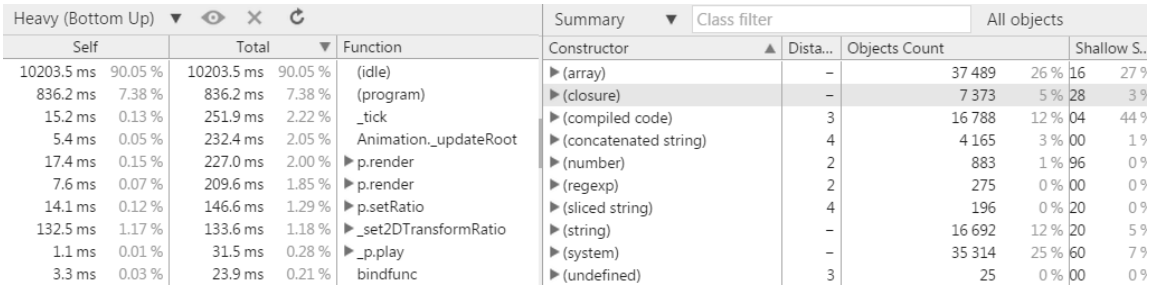


Figure 9: CPU and Heap profile examples

In both cases, what we receive is a crude table of complicated statistics or a graph. It will not show you where your problems are - those results have to be analyzed. A developer needs to know what to look for. There are, however, some common practices helping you pin-point suspicious bits.

The most obvious one is to:

- Start the CPU or Heap recording
- Perform an action on the page or game that is suspected to be slow or problematic
- Stop the recording

As a developer, you should already have a hunch of what part of the logic needs such investigation. If you don't, you may record that actions that are most used.

In Heap profile, what we want to look for are "leaks". Although the Garbage Collector does its job well, irresponsible programming often leads to the page or app gradually growing in memory which may result in slow performance and even crashing. Memory has to be returned - clear that which is not needed anymore.

A useful practice for finding memory leaks is the **comparison** view in Heap profiler:

- Think of an action or a process in your app that can be repeated or reverted
- Take a heap snapshot
- Repeat or revert that action
- Take another heap snapshot
- Select the 2nd snapshot and switch to "comparison" view

Usually, when you repeat the same action or revert it, there shouldn't be much

difference in memory print between the states. If you see a lot of “new”, or the “delta” is positive”, it may be worth looking in to. You may be forgetting to clean some old references and new ones keep piling up.

6. ANALYZING OUR GAME

This is the part where we apply gained knowledge and perform a practical analysis of an existing commercial web game the development of which the author of this thesis took part in. For the analysis, latest version of Chrome, “42.0.2311.135 m”, was used.

The game under study is an actual game developed for HTML5. Although the target clients are mobile device users that enjoy gaming experience on their phones and tablets, the game can be and frequently is played on the desktop computers as well.

First, we will look at how the game looks and plays, we will check its DOM. Then we will start with our analysis.

The Game

The game is called “Lucky Leprechaun” and it is a typical slot game with reels and symbols. You spin the wheels and hope for some of the possible winning combinations to land.

The game is available to play on different game web sites. One example, which was used by the author to test the game is Olybet [10]. The game can be played for free in “Fun mode” [11].

Below is a screenshot taken directly from the main game screen.

Figure 10: Lucky Leprechaun main game screen

We can see that the game is made purely in HTML, CSS and JavaScript. The main playing area is run on canvas and there are no signs of Flash or other plugins anywhere.

Checking the frame rate

As explained in the "Using the tools" section, there are 2 ways to check the frame rates – in the timeline and by enabling a hidden flag.

To find out the weak spots of the game we actually need to play it and record various game aspects using the Timeline tool. Due to lack of test data it was hard evaluate all game features. Here are the results of 3 main features the user may use:

Spinning the reels:

Figure 11: FPS snapshot when spinning the reels

The performance is smooth and FPS rarely drops below the 60 mark. There is a lot of idle time.

Animated win combinations playing:

Figure 12: FPS snapshot when multiple animations are playing

Same smooth experience as reels spinning. No problems here.

Swiping the screen:

Figure 13: FPS snapshot when swiping the screen

As we can see, swiping the screen results in a lot of paint and layer compositing calls (green). FPS often fell beyond the 30 mark and visual lags were easily noticeable. Not a very smooth experience.

This is definitely an area to look into although there may not necessarily be a way to optimize it. Looking at the DOM in the "Elements" tool or by using Chrome's "inspect" feature, we easily find the relevant container:

* *

This code implies that there is a very wide container and multiple screens nest inside it. By swiping, we change the screen "x" and "y" positions.

One possible optimization that springs to mind is adding style property "*display: none*" to screens non adjacent to the current one. That way they are guaranteed to not be processed in render tree construction.

The result of this test produces a somewhat more promising FPS graph:

Figure 14: FPS snapshot when swiping the screen after optimization

Further tests and analysis of this logic will need to be performed, outside the scope of this thesis.

Memory usage

The next goal of our analysis is trying to understand what the biggest memory usages are and whether there is room for optimizations.

From the author's perspective, this is very important as lower end devices have less

memory and, during QA testing phases, are often reported to crash.

Heap memory

By recording the heap size, we get the total size of our reachable objects in memory – 5.9mb. This is not suspicious by itself, what we need to look out for are **memory leaks**.

Let us check for any leaks using the **comparison view**

Comparing heap size while playing the game

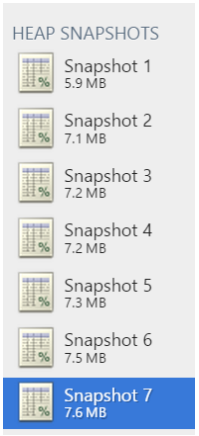


Figure 15: Heap snapshots after intervals of spinning the reels

What we see here is that while playing the game and taking heap snapshots between few spins, we notice that heap size keeps growing, which may point to a memory leak – some references are not being cleared most likely.

The next step would be to analyze what kind of objects keep growing in memory after each spin and understand whether something unnecessary can be cleared on next spin. This obviously requires access to the source code and will not be performed in the scope of this thesis.

Page total memory usage

Being aware of a possible memory leak, it was a good idea to check how much memory the game's Chrome page process uses. For that, Chrome's memory-redirect tool was used:

chrome://memory-redirect/

After loading the game, we get a rough number of 200 megabytes, which is fine for a game containing so many images, sounds, animations.

We saw that the memory footprint grows in size the more you play. However, the games has other features which may also suffer from not cleared references. Bonus is one such feature. Luckily, it was triggered while testing the game with random data. After the bonus was completed and normal game resumed, it was a good idea to see how the memory usage was affected.

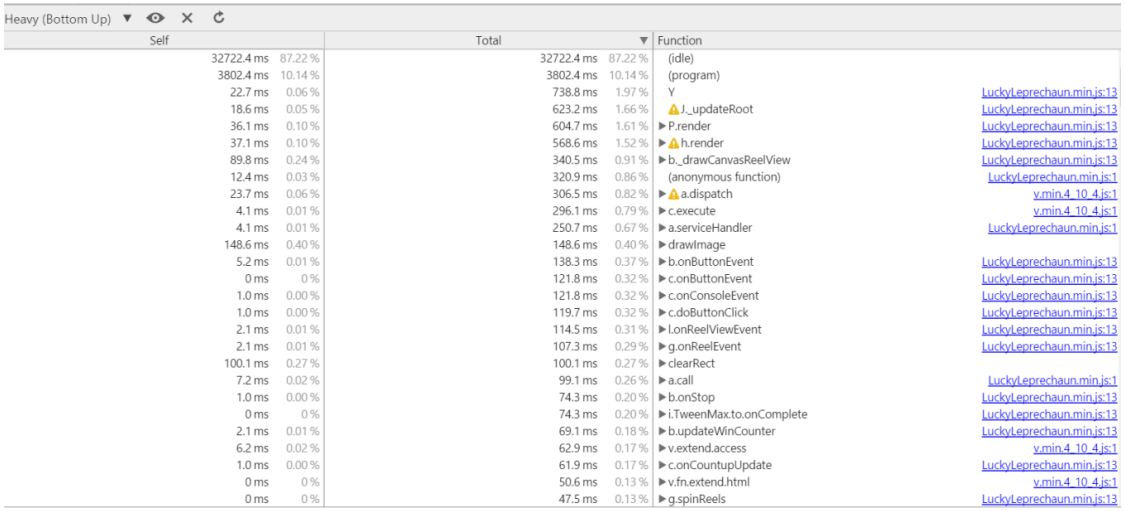
796	Tab Lucky Leprechaun	350576 k	70656 k	421232 k

Figure 16: Game memory consumption after Bonus feature ended

The total size is roughly 411mb, twice as large. This is understandable as new data had to be loaded in memory. However, the question is "Could it have been cleared after the rare feature has ended?"

CPU profile

To understand if there is any process in our game that takes too much CPU time, we run the "Collect JavaScript CPU profile". We record the CPU usage while casually playing the game, that is - spinning the reels. The result we get is:



Heavy (Bottom Up)

Self		Total		Function	
32722.4 ms	87.22 %	32722.4 ms	87.22 %	(idle)	
3802.4 ms	10.14 %	3802.4 ms	10.14 %	(program)	
22.7 ms	0.06 %	738.8 ms	1.97 %	Y	LuckyLeprechaun.min.js:13
18.6 ms	0.05 %	623.2 ms	1.66 %	⚠ J._updateRoot	LuckyLeprechaun.min.js:13
36.1 ms	0.10 %	604.7 ms	1.61 %	▶ P.render	LuckyLeprechaun.min.js:13
37.1 ms	0.10 %	568.6 ms	1.52 %	▶ ⚠ h.render	LuckyLeprechaun.min.js:13
89.8 ms	0.24 %	340.5 ms	0.91 %	▶ b._drawCanvasReelView	LuckyLeprechaun.min.js:13
12.4 ms	0.03 %	320.9 ms	0.86 %	(anonymous function)	LuckyLeprechaun.min.js:1
23.7 ms	0.06 %	306.5 ms	0.82 %	▶ ⚠ a.dispatch	v.min.4_10_4.js:1
4.1 ms	0.01 %	296.1 ms	0.79 %	▶ c.execute	v.min.4_10_4.js:1
4.1 ms	0.01 %	250.7 ms	0.67 %	▶ a.serviceHandler	LuckyLeprechaun.min.js:1
148.6 ms	0.40 %	148.6 ms	0.40 %	▶ drawImage	
5.2 ms	0.01 %	138.3 ms	0.37 %	▶ b.onButtonEvent	LuckyLeprechaun.min.js:13
0 ms	0 %	121.8 ms	0.32 %	▶ c.onButtonEvent	LuckyLeprechaun.min.js:13
1.0 ms	0.00 %	121.8 ms	0.32 %	▶ c.onConsoleEvent	LuckyLeprechaun.min.js:13
1.0 ms	0.00 %	119.7 ms	0.32 %	▶ c.doButtonClick	LuckyLeprechaun.min.js:13
2.1 ms	0.01 %	114.5 ms	0.31 %	▶ l.onReelViewEvent	LuckyLeprechaun.min.js:13
2.1 ms	0.01 %	107.3 ms	0.29 %	▶ g.onReelEvent	LuckyLeprechaun.min.js:13
100.1 ms	0.27 %	100.1 ms	0.27 %	▶ clearRect	
7.2 ms	0.02 %	99.1 ms	0.26 %	▶ a.call	LuckyLeprechaun.min.js:1
1.0 ms	0.00 %	74.3 ms	0.20 %	▶ b.onStop	LuckyLeprechaun.min.js:13
0 ms	0 %	74.3 ms	0.20 %	▶ i.TweenMax.to.onComplete	LuckyLeprechaun.min.js:13
2.1 ms	0.01 %	69.1 ms	0.18 %	▶ b.updateWinCounter	LuckyLeprechaun.min.js:13
6.2 ms	0.02 %	62.9 ms	0.17 %	▶ v.extend.access	v.min.4_10_4.js:1
1.0 ms	0.00 %	61.9 ms	0.17 %	▶ c.onCountupUpdate	LuckyLeprechaun.min.js:13
0 ms	0 %	50.6 ms	0.13 %	▶ v.fn.extend.html	v.min.4_10_4.js:1
0 ms	0 %	47.5 ms	0.13 %	▶ g.spinReels	LuckyLeprechaun.min.js:13

Figure 17: CPU profiler results

Apart from the "idle", which is irrelevant, and "program", which is something that can't be attributed to JavaScript code (such as rendering, dome creating, native code), we get a list of JavaScript functions.

From this result we see that most CPU usage is attributed to some render functions and "drawCanvasReelView" function.

The latter is probably worth investigating.

Audit

Performing an audit while easy and automated, produced a number of helpful advices in speeding up the page load (Network Utilization) and page smoothness (Web Page Performance).

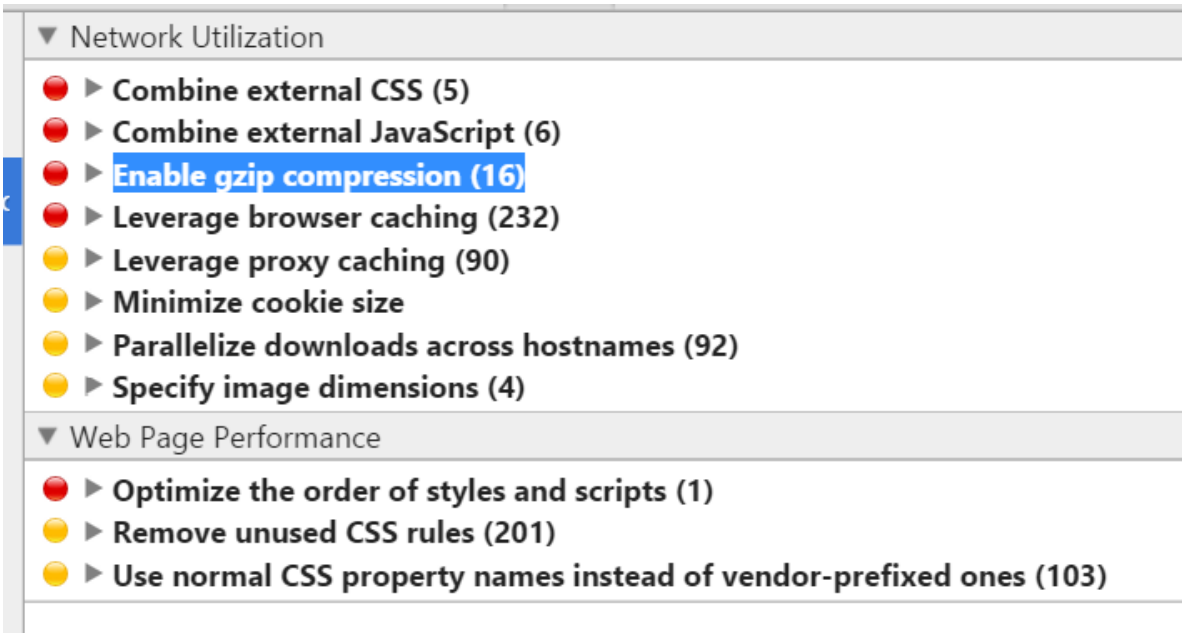


Figure 18: Results of running the Audit

Let us look at the suggestions proposed:

Combine external CSS – as explained in the "What can we optimize" section, reducing the number of CSS files will result in less HTTP requests.

Combine external JavaScript – same as with CSS. Less network requests – faster load time.

Enable gzip compression – "Compressing the following resources with gzip could reduce their transfer size by about two thirds (~1.7 MB):

Gzip compression is a useful way to reduce the size of our resources, thus making their downloading faster. Apparently in our game the server is not set up to use compression. Solid advice. [12]

Leverage browser caching and **Leverage proxy caching** – provides a list of resources that have suboptimal cache settings. Efficient caching of resources is a great way to reduce game loading times.

Minimize cookie size – large cookie size means slower HTTP requests. In our case, it is only 18 bytes which is very small. It is assumed this suggestions is always offered by the Audit tool.

Parallelize downloads across hostnames – this suggestion basically means downloading requested resources in parallel from different hosts. This should speed up the time it takes for the game to load resources.

Specify image dimensions – as the audit tool explains it, "A width and height should be specified for all images in order to speed up page display".

Optimize the order of styles and scripts – the tool notified that an inline JavaScript code between 2 links to external CSS files in the "header" of the HTML blocks possibility for their parallel download.

Remove unused CSS rules - as much as 201 CSS rules defined are not actually used and need to be cleaned up.

Use normal CSS property names instead of vendor-prefixed ones - although this was done to support a wide range of browser, most likely this was implemented improperly

Methodology for Interactive web application performance study and ways to improve it

While learning to use performance analysis tools and applying it in a real game, we worked out some efficient methods to help us with the analyzing process. These methods can be used by any web developer and should guide them with the analysis:

Although we used Chrome's developer tools to demonstrate the analysis, other browsers should also have similar tools available.

Let us sum up the methods we used:

Frame rate study - looking for problematic areas in our page rendering:

- Use the Timeline tool, Frame views
- Record an action or process
- Look for the FPS spikes in the graph
- Analyze the functions and events behind them

Memory study - aiming to find possible memory leaks:

- Take a Heap snapshot in Profiles tool
- Perform the action that should be analyzed (it can be a suspicious or just commonly used action in the application)
- Revert to the same state before that action
- Take another Heap snapshot
- Use Heap "comparison" view to find if our used memory size has grown
- If some objects keep growing in size, that action may be causing a memory leak
- Also check total Heap size and page memory usage changes after using the application for some time

CPU study - finding the processes that utilize the most CPU time

- Record JavaScript CPU usage

- Perform the action that should be analyzed
- Sort by Total CPU usage
- Find and analyze functions that use the most CPU, except for Idle and Program

Audit – running the automated analysis tool.

- Run the tool
- Analyze and consider generated advices – information on each can be easily found online

While these methods still require the developer to study the code or process, they should be useful for finding where the problems or problematic areas lie.

Using this methodology as a base, it is planned to release a commercial tool for web application performance analysis and optimization.

7. Conclusions

In this master thesis we have raised problems the author and most web developers run into that are all result of the lack of understanding of the web browser logic. To be able to write efficient code or optimize existing apps and pages, programmers needs to know what is going on behind the scenes - the structure and mechanics of the browser.

Today, the browser is a very powerful multimedia tool which can process and present us with not only text and images, but also videos, sounds and even fully interactive 2d or 3d games. The capabilities of the web browser as well as involved technologies have grown immensely. Unfortunately, many technologies, such as HTML and Canvas, CSS, SVG, Flash have been developed independently which causes complexities and dilemmas regarding their interaction and performance.

What concerns the structure of the browser, we have highlighted that a typical browser consists of a browser engine, a rendering engine, user interface, data persistence and networking layers, a JavaScript interpreter and a UI backend.

We have looked at the rendering flow which shows the steps the browser takes to deliver and present the page to our screen. It is a process of requesting, downloading and parsing the HTML document as well as the stylesheets into DOM and CSSOM trees respectively, which, in turn, are merged into a Render Tree. This tree contains only the nodes and their visual data. However, the next step, the Layout, calculates their size and positions on screen and, finally, they are painted on the screen by the Rendering Engine.

We have also learned about the significance of page load times and page performance. We covered some popular steps regarding their optimizations, such as merging JavaScript and CSS files and using spritesheets to reduce the number of HTTP requests.

To have a closer look at a web browser and how to use it for performance analyzing, we chose the popular Google Chrome browser. Chrome uses formerly WebKit and now Blink as its rendering engine and its own JavaScript engine which is called V8.

Google Chrome offers an amazing set of developer tools to help with the web page analysis. Developer tools include, but are not limited to the DOM inspector to look at or modify the HTML, the Timeline to measure the frame rate and catch the events and function calls behind it, the Profiler to see what uses or misuses our CPU time and memory, the Audit tool to get Google’s list of splendid advices on optimizing the page.

Having stressed the importance of knowing the browser and how to optimize our page or app, as well as how to use the tools available to measure the performance and find the weak spots, we got down to the practical part of our thesis – the game analysis.

The analyzed game is a real project the development of which the author took part in. The game suffers from usual performance problems and the company where the author works, as many others, do not have the time or resources to allocate to performance analysis. This is the reason why this thesis topic was chosen.

As a result of study performed in this thesis a methodology for analyzing browser application performance and ways to improve this were created. In the analysis of the game, the same Chrome developer tools were used. We found what causes the most frame rate issues in the game. We used Heap profiling to look for notorious memory leaks and found suspicious bits. The CPU profiler showed us what was causing the most stress for our CPU. Finally, in the Audit section we summarized all the useful advices the tool generated for us.

In conclusion, it should be mentioned that understanding the browser and knowing how to analyze and optimize the performance is a great benefit to the author, and hopeful the readers as well. The achieved analysis and the improvement in the knowledge of the author will prove very helpful to the company he works for.

Kokkuvõte.

Selles töös me oleme tõstnud probleemi, millesse autor ja paljud veebi arendajad tihti sattuvad, ja mille põhjus on teadmatus sellest, kuidas töötab veebisirvija sisemine loogika. Selleks, et kirjutada effektiivse koodi või optimeerida olemasolevaid lehti, peab arendaja aru saama, kuidas veebisirvija käitub ning millest koosneb.

Tänapäeval oskab veebisirvija väga palju sisendeid töödelda, pakkudes kasutajale väljundina teksti, pilte, videosid, heli ja isegi interaktiivseid 2d või 3d mänge. Brauser ehk veebisirvija ja tehnoloogiad, mida ta kasutab, on kõvasti ja kiiresti arendanud. Kahjuks paljud neist, nagu HTML ja Canvas, CSS, SVG, Flash, olid sõltumatult üksteisest arendatud. See toob keerukust ja dilemmasid arendajale kuna ta peab aru saama, kuidas nad suhtlevad üksteisega ning mis töötab kiiremini selles või teises olukorras.

Mis puudutab struktuuri, tavaline brauser koosneb browser engine, rendering engine, user interface, data persistence, networking layer, JavaScript interpreteerija ja UI backend komponentidest.

Me uurisime rendering flow – protsessi, mille tulemusena brauser kuvab pildi ekraanile. Sellest protsessis brauser küsib ja laeb alla HTML ja stiilide faile, töötleb seda ja ehitab DOM ja CSSOM puid. Neid liidetakse kokku, et saada Render tree. Pärast arvutatakse saadud elementide suurused ja asukohad, kuhu neid joonistada. Lõpuks, rendering engine joonistab neid ekraanile.

Rääkisime veel lehe laadimise ja töötamise kiirusest ning selle tähtsusest. Vaatasime mõned tehnikad, kuidas lehe kiiremaks teha, näiteks JavaScript, ja stiili failide kokku panemine selleks, et vähem HTTP request'e teha.

Valisime selles töös jõudluse analüüsiks Google Chrome brauseri. Chrome on kõige populaarsem brauser tänapäeval, mis kasutab WebKit ja Blink rendering engine tehnoloogiat ning oma V8 JavasScript interpretaatorit. Chrome pakkub suurepäraseid töövahendeid arendajale. mille abil saab jõudluse analüüsida. Nad lubavad DOMi uurida, frame rate ehk lehe kiiruse analüüsida, vaadata, mis kasutab kõige rohkem meie CPU aega ja mälu, ning Audit tooli abil genereerida hea optimeerimise soovituste listi.

Rõhutades analüüsi ja optimeerimise tähtsusi, jõudsime mängu analüüsini. Mäng, mis oli selles töös analüüsitud, on reaalne online mäng, mille arendamisel selle töö autor osales. Mängus eksisteerivad tüüpilised jõudluse probleemid ja firmal, kus autor töötab, nagu enamustel firmadel, ei ole aega ning ressurse mida pühendada selle analüüsimiseks. Seepärast oli selline töö teema ning eesmärgid valitud.

Analüüsis kasutasime Chrome developer tools. FPS uurimise tulemusena saime kohti, kus jõudlus on kõige madalam, Heap profiler abil otsisime kohti kus võib olla memory leak ning leidsime kahtlase koha. CPU profiler näitas mis kasutas kõige rohkem CPU aega. Lõpuks, Audit'i abil saime palju kasulikke soovitusi.

Kokkuvõtteks võib öelda, et brauseri aru saamine ning jõudluse analüüsimise ja optimeerimise oskus on väga kasulik autorile ja loodetavasti ka lugejatele. Mängu analüüsi tulemused ning autori uued teadmised ilmtingimata toovad kasu autorile ning firmale, kus ta töötab.

8. Bibliography

1. Pingdom, Report: The most common web browsers and browser versions today [WWW] http://royal.pingdom.com/2011/06/17/report-the-most-common-web-browsers-and-browser-versions-today/ (11.05.2015)

2. P. I. Tali Garsiel, How Browsers Work: Behind the scenes of modern web browsers [WWW] http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/ (10.05.2015)

3. P. Sexton, Deferring images for faster pages [WWW] https://www.feedthebot.com/pagespeed/defer-images.html. (12.05.2015)

4. L. Simon, Minimizing browser reflow [WWW] https://developers.google.com/speed/articles/reflow. (12.05.2015)

5. L. Lew, Repaints and Reflows: Manipulating the DOM responsibly [WWW] http://blog.letitialew.com/post/30425074101/repaints-and-reflows-manipulating-the-dom#notes. (12.05.2015)

6. Apple inc, Reducing HTTP Requests with Sprite Sheets [WWW] https://developer.apple.com/library/safari/documentation/NetworkingInternet/Conceptual/SafariImageDeliveryBestPractices/ReducingHTTPRequestswithSprites/ReducingHTTPRequestswithSprites.html#//apple_ref/doc/uid/TP40012449-CH4-SW5. (11.05.2015)

7. P. Irish, WebKit for Developers [WWW] http://www.paulirish.com/2013/webkit-for-developers/. (17.04.2015)

8. Google, Chrome DevTools Overview [WWW] https://developer.chrome.com/devtools. (18.04.2015)

9. Google, Performance profiling with the Timeline [WWW]. https://developer.chrome.com/devtools/docs/timeline. (14.05.2015)

10. Olybet [WWW] https://eemobile.olybet.com/en/Games/. (22.04.2015)

11. Microgaming, Lucky Leprechaun [WWW] http://mobile3.gameassists.co.uk/MobileWebGames/game/mgs/4_10_1?lobbyName=iForiumDEMO&languageCode=en&casinoID=1866&loginType=VanguardSessionToken&bankingURL=&gameName=luckyLeprechaun&clientID=40300&moduleID=10374&clientTypeID=40&xmanEndPoints=https%3A%2F. (6.05.2015)

12. I. Grigorik, Optimizing encoding and transfer size of text-based assets1 April 2014. [WWW] https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer?hl=en. (10.05.2015)