

Bachelorthesis

Thema:

“Performance Optimizations in JavaScript for Canvas2d Rendering”

An der Fachhochschule Dortmund
im Fachbereich Informatik
erstellte Bachelorthesis
im Studiengang Praktische Informatik

zur Erlangung des Grades *Bachelor of Science* von

Jöran Tesse

geboren am: 21.12.1986

Matrikelnummer: 7077519

Betreuer: **Dr. S. Betermieux**

Zweitprüfer: **Dr. K. Koll**

Dortmund, 9. September 2013

Markenrechtlicher Hinweis

Die in dieser Arbeit wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenzeichen usw. können auch ohne besondere Kennzeichnung geschützte Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Sämtliche in dieser Arbeit abgedruckten Bildschirmabzüge unterliegen dem Urheberrecht © des jeweiligen Herstellers.

Trademark Notice

Any trade names, trademarks etc. mentioned in this thesis may be subject to copyright protection and/or other legal regulation without explicit notice.

Abstract

In the course of HTML5, several techniques have been developed that allow JavaScript to render animations and complex scenes. This thesis takes an in-depth look at the way that browsers execute JavaScript code, evaluates its performance in rendering-related tasks and evaluates the general capabilities of popular JavaScript engines, and specifically analyzes the V8 JavaScript engine used in the Chromium project. Several of the proposed optimizations are analyzed in the context of a simple HTML5 game, which has been created for this thesis to identify performance threats in time-critical rendering operations. The main conclusion from this thesis is that even naive, unoptimized JavaScript code is generally fast enough to allow for a sufficient frame rate in HTML5 games, while highly optimized JavaScript is capable of computing complex scenes and physics models. However, the thesis also identifies several cases, where browsers do not yet optimize the executed code as well as they could, so JavaScript performance can be expected to continue improving over the next years.

Kurzfassung

Mit HTML5 wurden etliche Techniken eingeführt, die es JavaScript erlauben, Animationen und komplexe Szenen zu rendern. Diese Thesis nimmt einen tiefen Einblick in die Art und Weise, wie Browser JavaScript ausführen, bewertet dessen Performanz bei Aufgaben im Bereich des Renderings und analysiert die generellen Möglichkeiten populärer JavaScript-Engines, speziell die der V8 JavaScript-Engine des Chromium Projekts. Etliche der vorgeschlagenen Optimierungen werden mit Hilfe eines einfachen HTML5-Spiels untersucht, welches für diese Thesis erstellt wurde, um verschiedene Aspekte der Echtzeitfähigkeit von JavaScript einordnen zu können. Das wichtigste Ergebnis dieser Arbeit ist, dass selbst naiver, nicht optimierter JavaScript-Code zumeist schnell genug ausgeführt wird, um HTML5-Spiele mit einer hinreichenden Bildrate darstellen zu können, während stark optimierter Code sogar in der Lage ist, komplexe Szenen und Physikmodelle zu berechnen und darzustellen. Trotzdem identifiziert diese Thesis auch mehrere Fälle, in denen die Browser den Code noch nicht so gut optimieren, wie es theoretisch möglich wäre, sodass zu erwarten ist, dass die Performanz von JavaScript auch in den nächsten Jahren noch weiter verbessert werden wird.

Table of Contents

1	Introduction	1
1.1	Motivation and Background	1
1.2	History of JavaScript's Role in Browsers	1
1.3	Recent Demands on JavaScript Features	2
2	JavaScript Fundamentals	4
2.1	JavaScript Compilation and Interpretation	4
2.2	Primitives and Objects	6
2.3	Arrays and Dictionaries	7
2.4	Object-Orientation in JavaScript	7
2.5	The Event Loop	8
3	Creating a Game	10
3.1	Software Environment	10
3.2	Client-side Structure	12
3.2.1	Managing Resources	13
3.2.2	Enlarging a Fixed-Size Scene to Fill the Available Space	15
3.2.3	CSS Transitions	16
3.2.1	Rendering Quirks: UI-Events and Garbage Collection	18
3.2.2	Rendering a Scene	18
3.2.3	Applicability of Threads in Rendering	22
3.3	Server-side Structure	23
3.3.1	WebSockets	24
3.3.2	Asynchronous IO and the C10K-Problem	25
3.3.3	Advantages of a Modular and Asynchronous Architecture	26
3.3.4	Communication through WebSockets	27
3.4	Advanced Optimizations for V8	28
3.4.1	Self-modifying Code	28
3.4.2	A Look at V8's Generated Assembly	31
3.4.3	Optimizing Compiler	32
4	Conclusions	34
4.1	Rendering Performance	34
4.2	What Gaps are not yet Closed	34
4.3	Is the Nature of JavaScript Slowing it Down?	35
4.4	What Key Elements is JavaScript Missing?	36

Eidesstattliche Erklärung	37
Appendices	38
A.1 Resource Management	38
A.2 Rendering a Scene	38
A.3 Self-Modifying Code	39
A.4 Source-Code (CD)	40
References	41

List of Figures

Figure 1: Implicit object creation	6
Figure 2: A screenshot of the game.....	10
Figure 3: Conceptual map of the game's client-server-infrastructure	12
Figure 4: The user interface's four different screens.	13
Figure 5: Loading the game's resources	14
Figure 6: A rendering bug	16
Figure 7: Font rendering in Google Chrome.....	17
Figure 8: Dropped frames	18
Figure 9: Occurrence of unused frames	19
Figure 10: Scene composition.....	20
Figure 11: The most important server-side modules and classes.....	24
Figure 12: Visualized hydrogen output.....	30

1 Introduction

This thesis evaluates the overall performance and possible performance optimizations of JavaScript code related to rendering operations and the HTML5 canvas element. In order to properly examine the necessity of optimizations, and to evaluate the optimizations, a simple game has been created. Some of the examined aspects could not be usefully integrated into the actual game, so their effects were examined in separate test cases instead. Although some sections specifically refer to the V8 JavaScript engine, all popular JavaScript engines are considered equally important, and its performance was monitored in every browser, given the browser offered the interfaces required for reviewing specific performance aspects.

1.1 Motivation and Background

HTML uses JavaScript as a client-side scripting language. Therefore, the increasing popularity of rich media content created with new HTML5 technologies also boosts the popularity of JavaScript itself. Over the years, projects have emerged where the JavaScript language is used outside its original environment, for example the Mozilla Rhino project, which allows JavaScript to be easily embedded into Java programs, or software like node.js or vert.x, which provide a server-side JavaScript execution environment. The Tiobe Community Index for July 2013, which tries to measure the popularity of programming languages by counting the search hits on various popular websites, lists JavaScript at rank 10. Additionally, the language becomes more important due to the fact that writing apps for mobile devices in HTML with frameworks like Phonegap or Appcelerator Titanium becomes increasingly popular, and web-based clients are even expected to be a more future-safe development target than traditional thick clients¹. Since JavaScript becomes more and more important as a scripting language used mostly for client-side GUI programming, this thesis aims to evaluate JavaScript's performance in the context of a small game.

1.2 History of JavaScript's Role in Browsers

JavaScript was developed in 1995 by Brendan Eich for the Netscape Navigator, with the intention to add some basic interactivity to HTML pages, and to be the glue between a document's object model and applications written in Java. The first version of this

¹ [Blom et al., 2008]

lightweight language was finished after just 10 days of development, which well describes JavaScript's long time role of being merely a small scripting language to create some very basic active content. Two years after its creation, the language became standardized by Ecma International, and was officially named EcmaScript. The implementations by the different browser vendors were and still are called JavaScript (or JScript in the case of Microsoft's Internet Explorer), though they actually all refer to the EcmaScript specification. Changes to the language are discussed very publicly, allowing JavaScript (or EcmaScript) to be a "truly open language"².

1.3 Recent Demands on JavaScript Features

Dynamic page layouts utilizing Ajax allow websites to add animated transitions between different pages of a website. Due to the improvements in rendering speed and especially because all modern browsers leverage GPU acceleration, scripted animations have become quite smooth and get increasingly dependent on quick JavaScript to calculate new positions, dimensions and other transformations for animated objects. With the introduction of Canvas2d, WebGL and especially the `requestAnimationFrame` API, HTML5 established an infrastructure for scripted animations and rich content with JavaScript at its core.

Today, browsers are available for almost any kind of device, making JavaScript and HTML an attractive target for multi-platform developments. Apps for mobile devices can be written in HTML and JavaScript, which may then be distributed to any device utilizing frameworks like PhoneGap or Appcelerator Titanium. Those apps are called hybrid apps, run on almost any mobile device available, and are expected to surpass native apps by the year of 2016³, according to Gartner, Inc.

Recent performance improvements on the runtime performance of JavaScript have also made it a desirable target for compilation. Utilizing Mozilla's most recent performance-achievement, the Odin Monkey JavaScript engine, several projects like EmScripten attempt to compile software written in C++ to JavaScript by first compiling it to LLVM (low level virtual machine) byte code, which already contains most static code optimizations as well as its own memory management, and then rewriting the byte code in JavaScript. The most noteworthy developments in this field are an implementation of the bullet physics framework, achieving a performance that is off by less than a factor of two

² [Rauschmayer, 2012]

³ [Gartner, 2013]

compared to a native C++ implementation, and the yet to be released port of the Unreal Engine, a popular 3D game engine.

2 JavaScript Fundamentals

When trying to tune JavaScript code for performance, it is important to understand how the code is executed by browsers and what language constructs increase or decrease the overall performance of a script. This section will give an introduction to typical JavaScript execution environments, and will explain implications certain properties of the JavaScript language have on the execution environment.

2.1 JavaScript Compilation and Interpretation

JavaScript code is, like many other scripting languages, not compiled to any kind of machine or byte code. Instead, the source code is delivered as plain text. This means that a file containing JavaScript code is not stored in a way that allows for efficient or quick execution and does not contain either checks for correctness, nor any kind of optimization. Since the program executing the script file has to parse human-readable words during execution, this method of running JavaScript is inherently slow, and is called interpretation. In contrast to code interpretation, compilation aims to translate the plain source code into machine instructions, allowing the compiled JavaScript program to run directly on the CPU. In the context of JavaScript, there are two relevant compilation techniques: ahead-of-time and just-in-time compilation. Ahead-of-time compilation refers to the most common compilation method, where the complete source code is translated to machine code before it is run, which is common for languages like C. A just-in-time compiler, on the other hand, translates only small parts of code, while the surrounding parts of code are already in execution. The architecture of any modern JavaScript engine incorporates both of these methods, sometimes even multiple times. Chrome's JavaScript engine "V8", for example, has an ahead-of-time compiler that is optimized for a fast compilation process, allowing Chrome to almost immediately start executing any JavaScript code. While this relatively slow running code is executed, Chrome keeps track of how certain functions are used and especially how often they are used. If during execution V8 finds a function that gets executed on a regular basis, its just-in-time compiler will try to optimize the pressured function, optimistically leveraging any statistics related to the function's parameters or contents. This concept allows the browser to get started very quickly, thus reducing the overall page load time, while heavily-used functions will execute faster as time passes by. Therefore, functions that are executed at each step of an animation can be assumed to be optimized by the just-in-time compiler.

While animations and games benefit mostly from further developments in the field of just-in-time compilation, some browser vendors have also put a lot of effort in fast-running code that is compiled ahead of time. The most noteworthy, yet also most insecure attempts at pre-compiled code and efficient rendering are browser plug-ins like the Adobe Flash Player or Microsoft Silverlight. Due to several reasons, the most prevalent one being that most browser plug-ins are not compatible with mobile devices, the use of plug-ins is on the decline.

Google's attempt to bring pre-compiled code to browsers was open-source and integrated into the browser's architecture, unlike the Flash Player and other plug-ins that used plug-in-interfaces to deliver their content to the browser. Nevertheless, Google's solution still imposed a significant number of security-related questions, because "Native Client", or NaCl, allows the developer to distribute locally compiled modules written in C++. For security reasons, these modules only allow a subset of what C++ is actually capable of, but it still allows developers to keep intense computation in very efficient machine code that even allows the usage of posix threads. Despite several attempts, Google has not yet been successful at persuading other browser vendors to implement NaCl. For the sake of completeness, it can be mentioned that "Portable Native Client", or PNaCl, is an extension of NaCl's concept, where the delivered module is not yet fully compiled machine code, but a byte code using a subset of the "LLVM bitcode". Therefore, PNaCl allows a less machine-dependant approach than NaCl but requires the browser to implement an LLVM compiler.

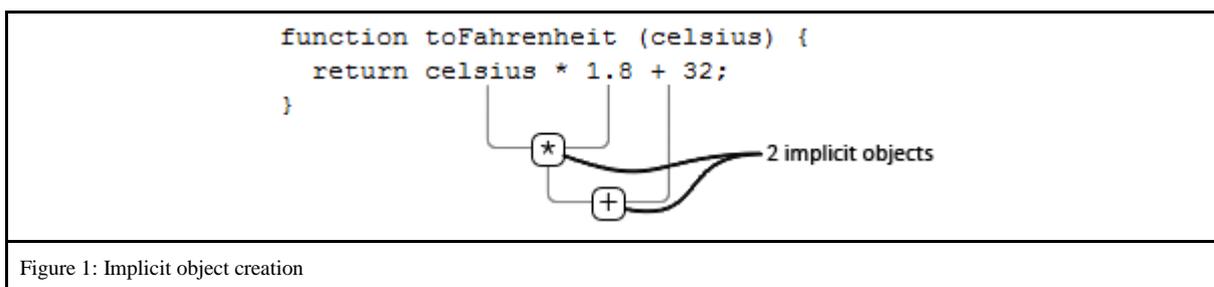
Quite recently, Mozilla has taken a look at pre-compiled code, by adding a very specific ahead-of-time compiler to their browser's JavaScript engine, called Odin Monkey. Instead of allowing machine code or C/C++-applications to run in Firefox, Odin Monkey compiles specially formatted JavaScript. This formatted JavaScript code is commonly referred to as "asm.js", because the first line of code is required to be the quoted string "asm.js". In contrast to Flash and NaCl, this approach allows the distributed script file to run in any other browser, because it technically still is regular JavaScript. The main reason for asm.js code to be compiled ahead-of-time so efficiently, is that the asm.js ruleset stipulates that asm.js code must store any variables inside a typed array, which can be compared to a byte code that implements its own memory management. Therefore, it is not surprising to see several projects arise that attempt to convert LLVM bitcode to asm.js code, thus allowing C++ programs to be compiled to very efficient JavaScript code. Still, the strict ruleset of

asm.js does not allow asm.js modules to modify the document object model, leaving asm.js the niche of functions with intense computation.

2.2 Primitives and Objects

There are several different types of data in JavaScript, although this is mostly not part of the language specification and instead refers to the way that JavaScript engines internally handle variables. Any variable may hold any type of value at any point of time, which is called dynamic typing. This propagates a lot of freedom to the developer, but slows down the code as the interpreter needs to continuously check what type of value any variable is actually holding, before it can evaluate an operation. JavaScript engines typically gather information about what types of value any variable holds, so that its just-in-time compiler is able to create optimized code by forming certain assumptions implied by the gathered information, for example the assumption that a variable always holds a certain type of value. This is very important, because this enables the compiler to “unbox” values, i.e. that a variable no longer holds an implicit object but an actual value. Unboxing variables greatly reduces the code size, since unoptimized code contains several sections of code handling the different types of values that may occur, while the optimized function only implements some mechanism to de-optimize the function in case the variable holds an incompatible type of value.

Implicit objects are another threat to a script’s performance. They occur in any section of code that is not optimized, and are used because the code does not take any assumptions about the variables used, and therefore stores each variable as an object containing both the variable’s value and information about its type. Even small lines of code with just a little arithmetic may take a long time to execute due to the fact that the browser allocates several objects for temporary values during computation, which often become obsolete immediately after being used.



2.3 Arrays and Dictionaries

Arrays in JavaScript are somewhat different to array implementations in other languages. They are not a language construct, but actual objects. In JavaScript, an array may hold numeric indices from 0 to n like in any other language, but it may as well hold indices scattered across the number space of any valid integer, or even use strings as indices. This enables the programmer to create easily readable and abstract code, but also bears the potential to significantly slow down code execution, since most JavaScript engines have two different internal modes for arrays.

The first mode is called array-mode and works just like any C-programmer would expect it to. The array holds a certain amount of items, all indexed by ascending numbers starting with zero, increasing by one for each element. The second mode, called dictionary-mode, is applied when strings are used as indices. The browser creates a map, usually a hash map, storing all the keys used in the array. When accessing an element, the JavaScript engine has to first find the key in its dictionary, which yields the position inside the actual array. In contrast, the array-mode indices already correspond to their actual position in memory. Obviously, the array-mode's performance outmatches the dictionary-mode, due to the significantly reduced overhead in element access. Still, a browser will change an array to dictionary-mode despite the fact that it only uses numbers as indices, if the JavaScript engine detects that the used indices are scattered too far apart from each other. This optimization saves memory, since the array size then corresponds to the number of elements it contains rather than the highest index used. However, the cost for the reduced memory usage is a slower access to array elements.

2.4 Object-Orientation in JavaScript

Although JavaScript does not feature a sophisticated model for inheritance like other modern languages, it is still possible to create code in an object-oriented manner. There are also ways to create inheritance and static variables, but prototypal inheritance strongly differs from classical inheritance in languages like C++ or Java. The main difference is that JavaScript does not provide a class-based layout for objects, but instead applies object properties at runtime, implying that any object may gain or lose properties at any point of time. JavaScript does not enforce a coding style where objects keep a certain set of properties. Since an object uses the "function" keyword, the syntax may be confusing, but

the way that object properties are added already indicates the mutable nature of JavaScript objects (see code example 1).

```
function Car (description, color) {
  this.description = description;
  this.color = color;
}
var blackVW = new Car ('VW Golf', 'black');
```

Code example 1: Creating an object

For a JavaScript engine, this concept makes it difficult to actually optimize class usage, since a class-name does not say anything about the properties or functions an object has. Therefore, engines use the concept of “hidden classes”. At creation, any object in JavaScript is of the same, plain object type, because the properties and methods are attached during runtime. The engine then watches the names and types of the added properties, and applies a hidden class id, which identifies all objects of this shape. This implies that it is not only important what properties are added, but also in which order they are added (see code example 2).

```
function Car (description, color) {
  if (color === 'default') this.color = color = 'red';
  this.description = description;
  this.color = color;
}
var car1 = new Car ('Lamborghini Diablo', 'yellow');
var car2 = new Car ('Dodge Challenger', 'default');
// car1 and car2 now have different hidden classes
```

Code example 2: One constructor function to create two different hidden classes.

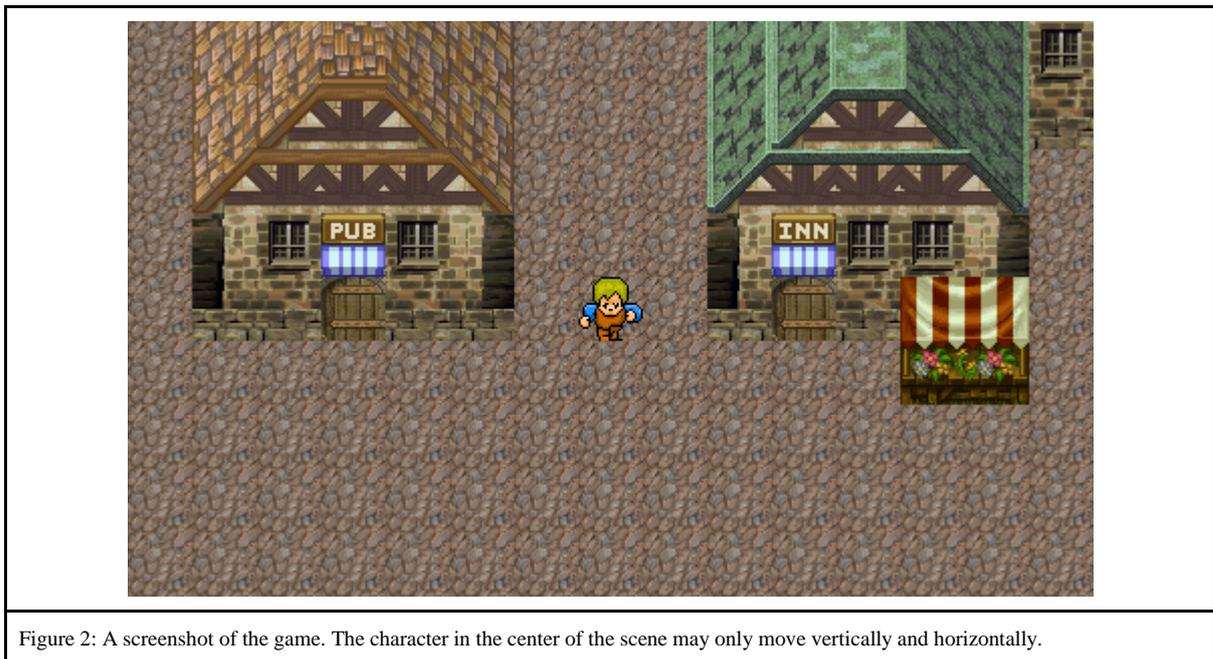
2.5 The Event Loop

Being developed for browsers, JavaScript was to create a lightweight language reacting to user interface events. JavaScript code is commonly built around several event listeners, but despite JavaScript’s asynchronous nature, it is inherently single-threaded. Except when using WebWorkers, there is one thread dedicated to executing all JavaScript code and JavaScript-related work, like for example garbage collection. Whenever an event outside of this thread occurs that tries to run a JavaScript function, the event reference is put at the

end of the thread's event queue. Once the thread finishes executing a piece of code, it will take the first item of the queue and continue by executing the referenced function. If the event queue is empty, this so-called event loop pauses and waits for new events to be triggered.

3 Creating a Game

To implement, test and evaluate most of the performance optimizations proposed in this thesis, I have created a simple, browser-based game, showing a character from above and allowing the user to move freely within a virtual world as seen in figure 2. The game is synchronized through a server, so multiple players are able to see each other. Players may walk through doors to enter buildings, but there are no further game mechanics implemented in this game.



A game requires JavaScript to be able to render the scene within the time between two frames displayed by the monitor. Additionally, a game scene is usually created by compositing many images, so creating a game is a good example of a context in which JavaScript's performance should be optimized for rendering on a Canvas.

3.1 Software Environment

The game is supposed to run inside a browser, so the implemented code already utilizes a substantial amount of publicly available third-party software. When evaluating the performance of JavaScript, it is important to minimize overhead imposed by unnecessary frameworks, therefore, the main goal for the software environment is to use as few frameworks as possible, while keeping as much of the computation as possible in JavaScript, to allow a better evaluation of the overall performance of complex JavaScript

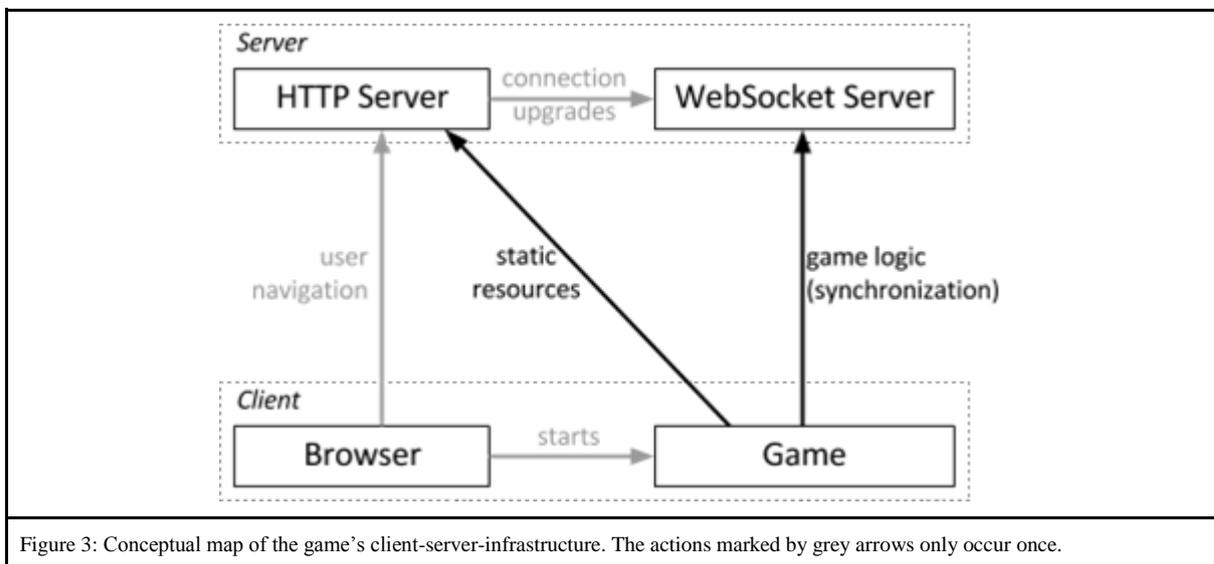
applications. On the client-side, there were no frameworks used except for one package for which a detailed analysis can be found in chapter 3.4.1. The execution environment includes most popular browsers, namely Microsoft Internet Explorer, Google Chrome and Mozilla Firefox. Chrome recently switched to the “Blink” rendering engine, but since this is a very young fork of the popular WebKit rendering engine, the game should also run on any WebKit-based browser, for example Apple Safari and Safari-based mobile browsers such as the iPhone or Android browser.

On the server-side, there are two projects that can be utilized for creating a server in JavaScript: Node.js and the vert.x project. I have decided to use node.js, because it provides very direct access to the underlying network and file system interfaces while having most of its framework code written in JavaScript. Node.js is built around Google Chrome’s V8 JavaScript engine and tries to keep the additional interface code to a minimum. In contrast, vert.x relies on well-tested open source projects such as Netty for network IO, Hazelcast for synchronization and Mozilla Rhino as its JavaScript engine. Vert.x is less platform-dependent, as it runs inside any Java Virtual Machine. Most of its framework functions are implemented in Java, and additional custom modules may be written in several different languages, for example Java, JavaScript or Python. In conclusion, V8 appears to be a more light-weight approach, which is more dependent on JavaScript’s performance, making it a better fit for the goals of this thesis. Additionally, the V8 JavaScript engine is also used in one of the client-side execution environments, so any conclusions from node.js code may be directly applied to the Google Chrome browser. In terms of game design, the game is built leveraging free and publicly available open-source software and images. I chose the “Tiled” map editor⁴ to create the maps used in the game, because it is capable of exporting maps in a JSON-based format (JavaScript Object Notation). This provides an easy way to import maps to JavaScript, since the language provides a native API for deserializing JSON-objects. The Tiled map editor supports creating maps with multiple layers, which I decided to utilize for storing certain meta-information about the map, by adding an invisible layer to the map. This layer contains information about whether or not a position may be walked upon or whether stepping onto a specific position triggers an event. The client uses this information to determine if the character may move into a certain direction, for example, a character may move on grass, but would be stopped when attempting to walk “onto” a tree or building. Trigger

⁴ Official website <http://www.mapeditor.org/> (also contains a link to the project’s source code at github).

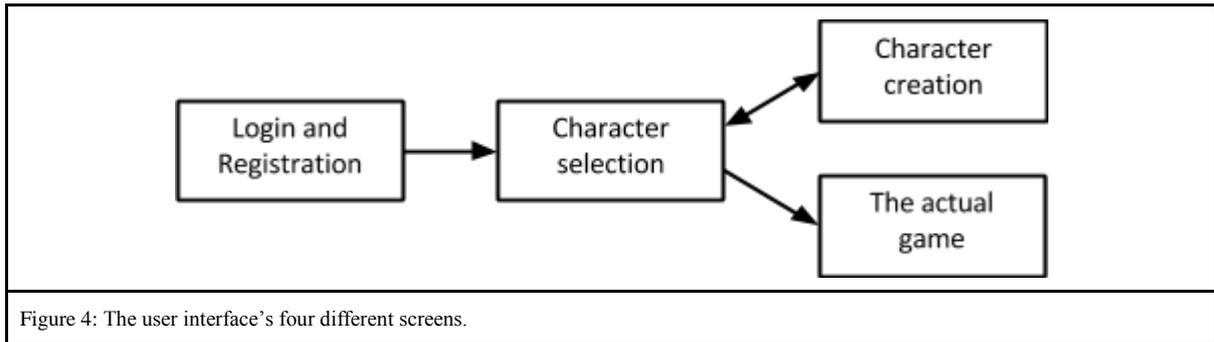
information, however, is only evaluated by the server, and only the server holds the information about what happens when stepping on the marked positions. As a basic concept, the client should only be able to raise events related to direct user interaction, such as changing the character’s position.

The server is implemented as a node.js-application, consisting of both a classical web server and a server for WebSocket connections. Basic content like source code or images will be delivered by an http server, while the game itself communicates through a persistent connection to the server implemented using a WebSocket connection, as illustrated in figure 3. Since the WebSocket protocol only allows the script to send strings, there is a custom protocol built on top that allows an event-based communication between client and server.



3.2 Client-side Structure

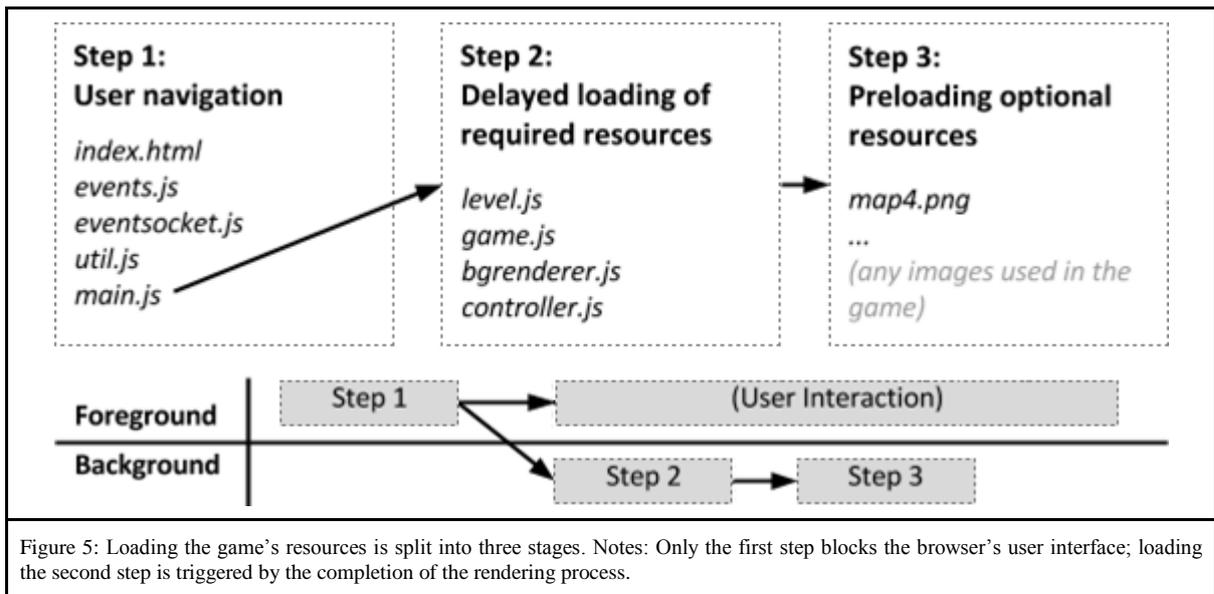
Before the actual game starts, the client-side user interface consists of four different screens, as seen in figure 4. The first page is visible upon navigating to the homepage containing the game, and shows a combined login and registration dialog. After logging in, the character selection screen is displayed and has a link to the third screen, which is for creating new characters. After those three, relatively static screens, the fourth and final screen is the game itself, which is used to display the visible portion of the game’s world, the scene.



These screens are not independent HTML-pages, but are instead one single HTML page that displays different contents. This architectural choice has two different reasons, the first being that this design pattern allows quick switches between the screens, as well as offering the opportunity to preload and initialize resources needed for a later screen. The second consideration is that the first screen may already establish the persistent WebSocket connection, which allows the server to keep the complete application logic, from login to moving around in the game's world, attached to the WebSocket server. Consequently, the server's http server is only used for static content and is not required to implement any dynamic content coupled to the WebSocket server, thus reducing the overall code complexity.

3.2.1 Managing Resources

To reduce the initial page load time, the landing page should load only those resources that are required for rendering the first screen. On the other hand, loading all resources in advance would allow the game to start instantaneously, once the player has chosen a character. Therefore, it is best to let the initial page load only the required resources, but, after the page is rendered, start preloading any scripts or images that might be used later on, thus combining the advantages of both strategies. In the game's implementation, the resources are loaded in three stages, as seen in figure 5.



Since the deferrable files (step 2 in figure 5) are exclusively JavaScript files, the initial page would pause any parsing or rendering to download and parse all those script files, due to the possibility of them containing the “document.write” command. This synchronous handling of JavaScript files is mutual among all relevant browsers, so the page load time can be expected to increase by exactly the amount of time needed to download and parse those files. Downloading those files through already open TCP connections simultaneously takes at least one full round-trip-time (best case), but having to open a TCP connection for at least one of the files would approximately double the time needed to finish downloading all files (probably but arguably the most common case). Since the time required to parse those files is negligible compared to the time needed to download them, an argument can be made that the delayed loading of these resources saves around 50 milliseconds, meaning that the initial page will display 50ms earlier than without this optimization.

The preloadable files (step 3 in figure 5) include the necessary spritemap “map4.png”, containing all the tiles used for painting the scene’s background. The sprite-maps for characters are grouped into images containing 4 different character appearances each. Consequently, the game needs at least the background images and the user’s character to display the complete scene, and more character images if there are other players present in the scene. Even this minimum amount of images sums up to 982 KiB of data and takes around half a second to load over an average broadband connection, although downloading

all resources at once would take at least .8 seconds⁵. Since preloading the resources does not interrupt user interaction, the only negative side effect of this technique is an increase in server traffic, because not every user visiting the game’s website would necessarily start playing. The benefit, on the other hand, is that a player may start the game seemingly instantaneously, as the browser is able to display the first rendered scene within the access time to the browser’s local cache, which is less than 1ms for an SSD, around 10ms for an average magnetic hard-disk drive and up to 160ms when using a mobile device⁶. Therefore, the preloading of optional resources allows the game to start immediately, instead of waiting at least half a second while downloading the required resources.

3.2.2 Enlarging a Fixed-Size Scene to Fill the Available Space

The game uses pixel-based images and renders a fixed region of the game world to the screen. The region’s size is defined by the number of individual background images, resulting in constant dimensions for the rendered scene. To use all of the available screen area inside the browser window, the painted pixel-graphics need to be enlarged to fill the screen. This can be achieved in several ways, for example by enlarging every single image as it is drawn. However, this technique increases the coupling between the rendering functions and the rest of the code, increases code complexity, and may potentially be the source of display errors. To prove the latter, I have created a test case utilizing an overloaded method of the regular Canvas API function “drawImage”, which allows drawing the individual images to specified dimensions, thus drawing an enlarged version of each image. A canvas element is capable of drawing to coordinates described by floating point numbers and will automatically handle any resampling and alpha-blending that is needed. Nevertheless, since every single image will be rendered individually, visible, half-translucent lines will appear between the images, as seen in figure 6.

⁵ The exact measurement is available in appendix A.1 on page 35.

⁶ Measurement available in appendix A.1 on page 35, classification in reference to [Tesse, 2013].

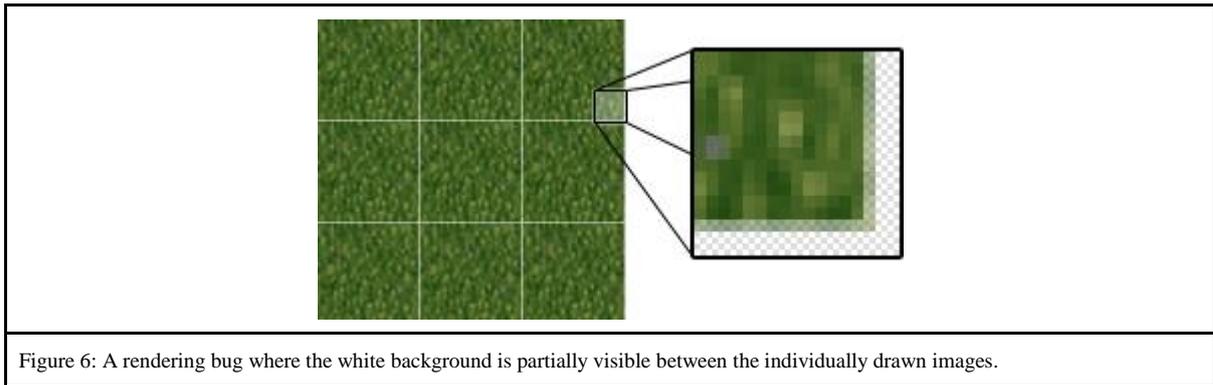


Figure 6: A rendering bug where the white background is partially visible between the individually drawn images.

The same bug occurs, if this method is slightly modified. Instead of resizing the images individually, it is also possible to set a constant zoom-factor to the canvas, which is then automatically applied to any operation invoked on the canvas. The implementation is very similar to the state machine of OpenGL and DirectX, so even though this method is still prone to the same bugs, this model at least passes some of the computation from JavaScript to the graphics hardware, thus improving the overall performance. Instead of fixing the bug by manually rounding positions and utilizing a model where not every tiled image of the scene is evenly sized, it is possible to set the canvas element's CSS property "transform" to "scale(factor)". This way, the images are not immediately enlarged when drawing the images, but instead the whole scene is drawn in regular size and scaled appropriately as a whole. This reduces the number of calls to graphics hardware accelerated operations, correctly blends images into each other without the background being visible in between, and reduces the code complexity by completely removing any size-related adjustments or computation from the rendering function.

In addition, there are indications ⁷ that CSS transforms force browsers to render the transformed element and all sub-elements on the graphics hardware, which may be an advantage if, for example, parts of the user interface are implemented in HTML. The canvas element triggers this mode itself, so there is no benefit in regard to canvas rendering performance.

3.2.3 CSS Transitions

Introduced with HTML5, CSS transitions offer an easy way to animate CSS property changes. A transition attached to the CSS property "width", for example, would gradually increase or decrease the elements width to a new value, instead of immediately applying

⁷ Applying transforms causes Google Chrome to add matching compositing regions.

any changes in its value. Since these transitions are plain CSS, they may be applied through style sheets, or, as seen in code example 3, directly by JavaScript.

```
// Get a specific element from the DOM
var element = document.getElementById ('div1');

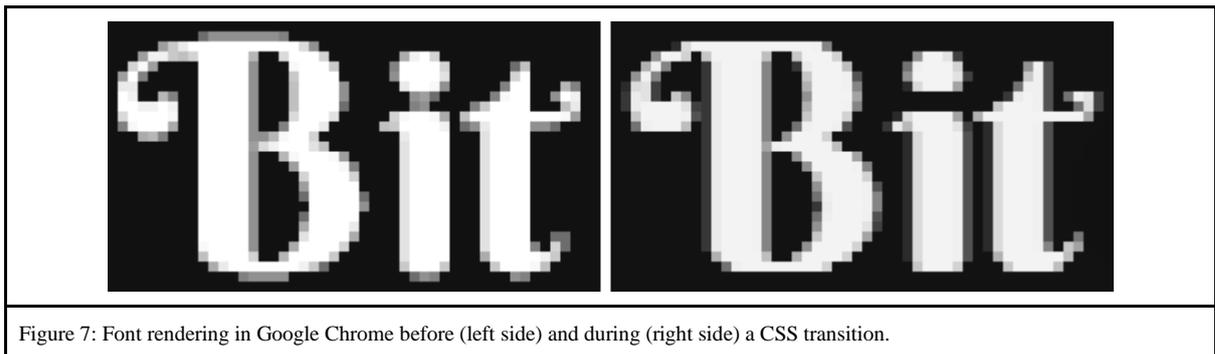
// Set an initial opacity (0 -> invisible)
element.style.opacity = 0;

// Attach a CSS3 transition to animate the opacity
element.style.transition = 'opacity 1s';

// The following statement triggers the transition
element.style.opacity = 1;
```

Code example 3: Creating a CSS3 transition through JavaScript.

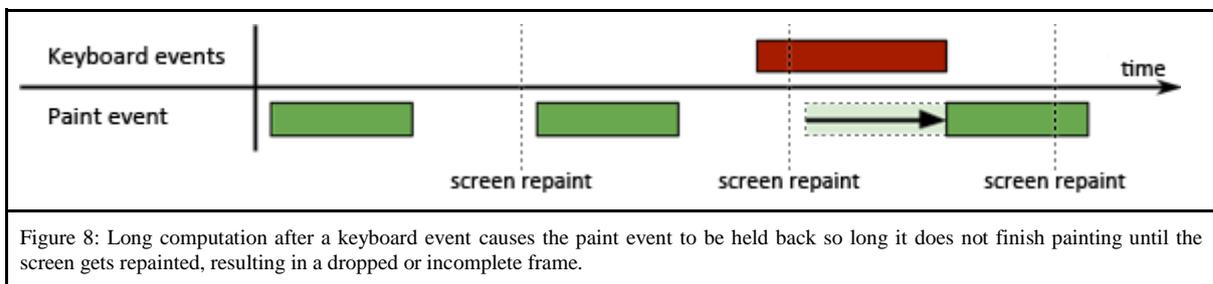
CSS transitions do not offer new features that could not be implemented in JavaScript, but it allows the browser to make certain assumptions during the rendering process. For a transition that is programmed in JavaScript, the browser only sees the current value for a certain property, but does not know how long this value would remain constant. Still, among the tested browsers, only one actually uses this information to its advantage: Google Chrome switches to a different font renderer while a CSS transition transforms an element, as seen in figure 7.



This different anti-aliasing method for font rendering only takes place if the animated region is of sufficient size, i.e. if it is large enough to yield a sufficient performance improvement. Additionally, the rendered text in figure 7 (right side) looks, as if no vertical anti-aliasing took place, so, in conclusion, it should be safe to assume that the alternate font renderer uses a simpler algorithm that renders the text faster.

3.2.1 Rendering Quirks: UI-Events and Garbage Collection

Once the game has started, there are three sources of events: server messages, user interaction (keyboard / mouse / touch) and the paint event. JavaScript is single-threaded, so intense computation upon receiving a server message or user input may cause a frame drop, as illustrated in figure 8. Whenever the screen gets repainted, the browser will invoke the paint event handler (a JavaScript function) to repaint the screen. If the script is already running, the event handler will be queued to run after the currently running piece of code has finished. Consequently, any event handler blocks the paint event handler and may cause a dropped frame.

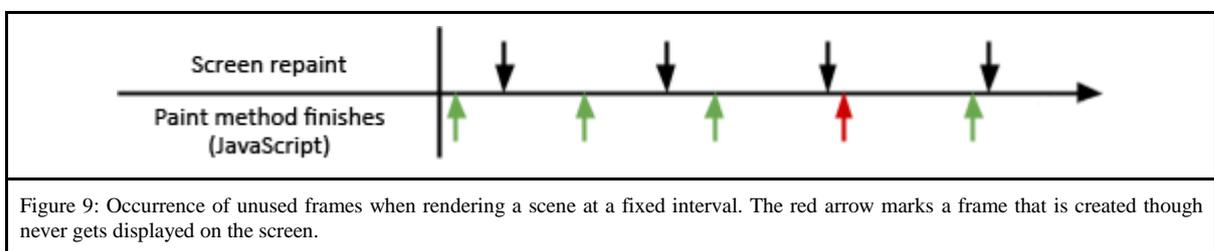


Therefore, all computation outside the paint method should be kept as small and simple as possible. Similar to any other JavaScript code, the paint event may also be delayed due to garbage collector activity. To free up memory, the garbage collector operates on the memory that is in use by JavaScript, causing the code execution to halt while the garbage collector is freeing up memory. The garbage collector will try to only clean up memory if there is no more memory available, or if the JavaScript event loop is currently idle. However, if an event occurs while the garbage collector is doing its work, the event will not be executed until the garbage collector finishes. If, for example, the garbage collector frees up a lot of memory and takes a second to do so, this is a whole second where an html-based game does not accept user input and would not even be able to update the displayed contents. The game “freezes” whenever the garbage collector is busy, therefore the generated garbage should be closely monitored and reduced if possible.

3.2.2 Rendering a Scene

The game has to continuously render the scene, which used to be achieved by setting up a regular timer through JavaScript’s “setTimeout” or “setInterval”. Even today, popular frameworks like the jQuery project rely solely on this mechanism to render

animations, which might be loosely compared to the rendering of a game’s scene. Nevertheless, HTML5 introduced a new function for triggering paint events called “requestAnimationFrame”. With this function, a script can ask the browser to execute a certain function right after the monitor displays the scene, ideally synchronizing the function calls with the screen refresh rate. Additionally, this method of painting the scene allows the browser to make certain implications about the function being called, for example that it performs an atomic operation in regard to the display, i.e. there would be no sense in displaying intermediate results. Although certain commands cause a reflow, meaning a rendering engine tries to re-render part of the displayed page, the browser may delay those rendering operations until after the function has finished, since the function is not intended to produce any meaningful output until it has finished. In the context of a game, the paint function would render the scene, and a scene only makes sense once the rendering process is completed. An unfinished scene, on the other hand, does not make any sense to a viewer and would be worse than a low frame rate in regard to an immersive gameplay. Another benefit to this method is that it reduces the amount of unused frames on fast machines. The most obvious situation for this is when the user changes the browser’s tab, minimizes the browser or when the screensaver starts, since the browser will not call the given callback function until the page actually gets redrawn. Hence, when in background, the game’s frames per second will drop to zero. Likewise, this method reduces unused frames on monitors with refresh rates below the assumed rate when using fixed timers to paint the scene. If the rendering method gets called every 16 milliseconds while the screen refresh rate is actually 60 Hz, or 16 2/3 ms, there will eventually be a frame that never gets displayed at all, as illustrated by figure 9.



The game scene can be split into three groups of objects that need to be painted. One group consists of the characters and any objects that may change their position within the world. The two other layers show the map and any static objects, differing only in that some parts of the map need to be painted in front of the characters, while the other layer is drawn

behind the characters. The necessity of a layer for overlay images is illustrated in figure 10, which separately shows the three layers and the resulting, composited scene. The background layer displays most of the map’s contents, but if the character were simply painted in front of the background, the tree in figure 10 would be rendered behind the character, and the character would consequently appear to have climbed the tree instead of standing behind it. Only by painting a mostly transparent overlay layer on top of the character layer can an illusion of depth be created.

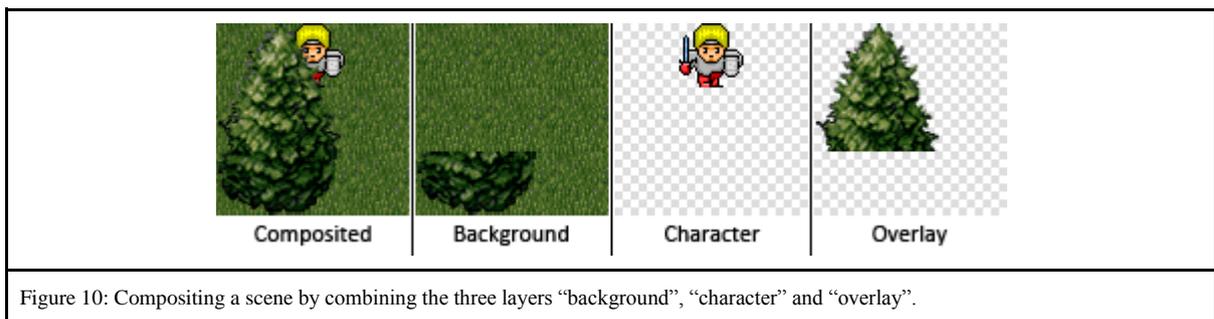


Figure 10: Compositing a scene by combining the three layers “background”, “character” and “overlay”.

There are mainly two different ways to render these groups of content. Since the game uses the HTML5 canvas element to render the scene or characters, the naive algorithm to paint the scene would be to paint those groups of images one after the other onto a single canvas. There is potential for optimizations, however, since the background and overlay layers do not change their content, they only move as the player changes his or her position. Drawing the background or overlay layer means painting 187 small images, depending on the tile numbers in the level data. However fast those 187 images are drawn, it is faster to cache the result, since every single image has to be looked up in the level data. This caching can be achieved by drawing the background onto either a separate canvas or an offscreen-canvas, as seen in code example 4. Using canvas elements in this manner, i.e. not displaying them on the screen, allows JavaScript to implement display lists, a classical technique in computer graphics ⁸.

⁸ Further reading: [Computergrafik und Bildverarbeitung]

```
function offscreenCanvas (width, height, paintFunction) {
    var canvas = document.createElement ('canvas');
    canvas.width = width;
    canvas.height = height;
    paintFunction (canvas.getContext ('2d'));
    return canvas;
}

// Usage:
var blackSquare = offscreenCanvas (10, 10, function (ctx) {
    ctx.fillStyle = '#000000';
    ctx.fillRect (0, 0, 10, 10);
});
```

Code example 4: Offscreen canvases in JavaScript.

As a method to further reduce the number of full repaints required for the background and overlay layer, the cached version of the background is slightly larger than the display area, which allows the game to just slightly move the cached image instead of completely repainting it. On the test machine using Google Chrome, there was no immediately obvious difference in the time needed for rendering the scene, but, nevertheless, both the algorithm utilizing offscreen-canvases and the one painting onto separate canvas elements significantly reduced the amount of garbage generated from about 2.4 MiB/S to less than 10% of that value, thus preventing any frame skips from occurring, instead of skipping a frame every 5.5 seconds. Looking at the profiling tools to compare the performance of the different browsers, the most meaningful comparison appears to be the time spent inside the main paint method called for each frame the browser renders⁹. Sadly, Google Chrome displays these numbers as a percentage, and shows the actual time with a measurement accuracy of 1ms, but this still is good enough for a reasonable comparison. According to Chrome's profiling data, the buffered algorithm with multiple canvases runs about 15 times faster than the naive implementation. Since the Chrome development tools always show 3ms for the naive implementation and 1ms for the buffered algorithm, math suggests that the values in milliseconds are neither rounded nor floored. Values below 1ms get ceiled to 1ms, with an unknown operation (possibly rounding) for higher intervals, so it is safe to say that the average time Google Chrome spends inside the naive paint function is anywhere between 2 and 4 milliseconds. Firefox needs 2.97ms to execute the same function, and the Internet Explorer takes an average of 0.95ms. In the optimized state, all browsers score very similar results at around 0.2 to 0.3ms. The conclusion of these

⁹ The complete measurement data can be found in appendix A.2 on page 35.

numbers is that buffering the rendered scene yields a performance improvement of factor 3 to 15. The Internet Explorer shows the best performance in terms of directly painting images onto a Canvas element, although the difference between the browsers diminishes once the paint function buffers the results for the images.

3.2.3 Applicability of Threads in Rendering

Threads in JavaScript are called WebWorkers, and are created by instantiating a WebWorker object and attaching a JavaScript file. The created WebWorker object will then execute the attached script file in a background thread that has no access to the document object model (DOM) or any variables, functions or objects declared in the main JavaScript thread. The communication between threads is limited to sending primitives as messages, so threads usually implement a listener function to receive those messages. Since it is not allowed to send objects to other threads, objects must be serialized before sending them.

To determine the applicability of threads, I have created an isolated test to measure the performance of each step. Unsurprisingly, the most costly part about threads is their creation, as the concept behind WebWorkers already suggested. Creating a thread, starting an instance of the JavaScript engine and setting up the synchronization is not a lightweight task. Therefore, setting up threads should always occur during the game's initialization, as the creation of WebWorkers during the game temporarily freezes the game. After the thread was created, the overall performance for passing messages between the threads was good across all browsers. It is safe to say that a game can benefit from letting a background thread handle certain computation within the 16 milliseconds available for rendering the scene, because the measurements showed that passing a message to another thread and back to the main thread almost always occurs within one millisecond. However, there is a downside when using threads: they are capable of generating massive amounts of garbage. When passing many messages between threads, the browser needs to copy various parts of memory resulting in lots of obsolete copies. Therefore, communication between threads should be kept as sparse as possible. Since threads do not have access to the document's object model, the suggestion of lightweight communication implies that there is no reasonable benefit in rendering content in any thread other than the main thread. If part of the scene were to be rendered inside a WebWorker, it would not be able to display it in the browser and should not transfer it to the main thread, thus leaving WebWorkers the niche of doing heavy computation. In the context of 3D games with

complex physics engines, it would be possible to keep the physics model inside a WebWorker, which then passes only the effective position changes or collisions to the main thread. A background thread in JavaScript may assist the main thread, but with the current WebWorker model, it will never be equally important as the main thread or responsible for any rendering.

3.3 Server-side Structure

The game's server is a node.js application written in JavaScript, and utilizes only frameworks supplied with a default node.js installation. Continuing the event-based, asynchronous architecture provided by the node.js APIs, the server is built upon the http package, which offers a very basic http-server. The supplied server module is extended to allow caching, compression and WebSocket connections, and adds an interface to conveniently add static or dynamic files to the http-server, as seen in code example 5.

```
// Load the server module:
var server = require ( './socksrv.js' );

// Add the physical directory "httpdocs" to the servers root:
server.addDir ( '', 'httpdocs' );

// Handle WebSocket connections:
server.onWebSocket (function (socket) {

    // Catch the WebSocket event "ping"
    socket.on ( 'ping', function () {

        // Send the "pong" event to the client
        socket.emit ( 'pong' );

    } );

} );
```

Code example 5: Using the server module (socksrv.js).

The game's logic communicates solely through WebSocket connections, and does not extend to any dynamic HTTP files. However, the used server abstraction does provide a basic interface to add dynamic files, but this feature is only used for logging and debugging purposes. Also integrated into the server module is a JavaScript implementation of the WebSocket protocol, which is capable of upgrading HTTP requests and offers a mandatory modular interface for adding higher abstraction layers on the basic protocol,

that is to say, the WebSocket server is not usable if no higher-level protocol is provided. This is in accordance with the WebSocket protocol definition, which specifically requires the browser to enclose a requested protocol name to any upgrade request. Upon receiving such a request, the server will try to find the requested protocol, and, if it is found, will initiate a WebSocket connection using the specified protocol.

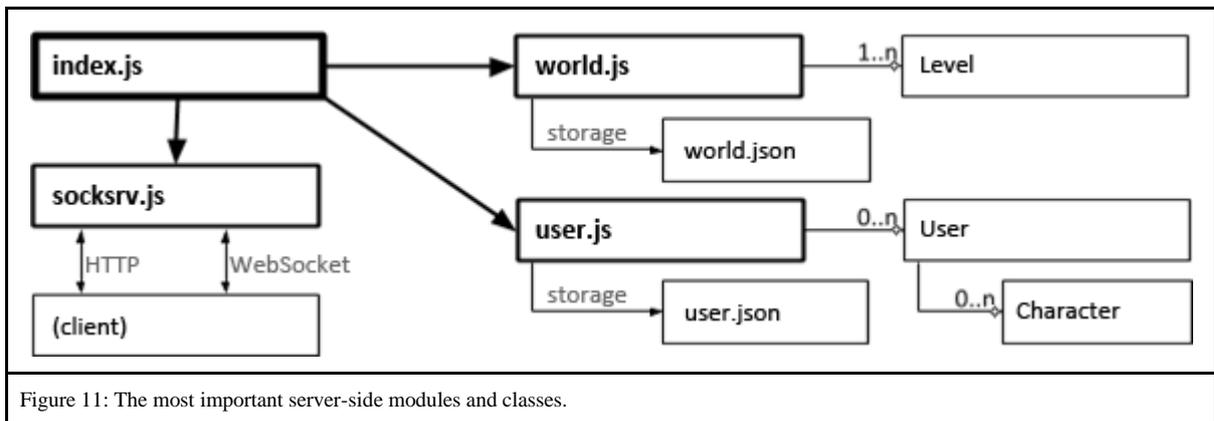


Figure 11: The most important server-side modules and classes.

Besides the server module, figure 11 shows two other important core modules, the “world” and the “user” module. The world module represents the game’s world and merges all the playable map areas to one big world, as defined in the world.json file. The user module, on the other hand, does not directly provide an object, but offers a class named “User”, which is used by the game’s logic. This class is created whenever a WebSocket connection is established, so each “User” is an abstraction of a connected browser (client). The login and registration process are encapsulated in the user module as well as a persistent storage for all users that ever created an account, which is stored in the users.json file. Each user may log in to an account saved in the users.json file, which also stores any characters attached to their account, including each character’s name, appearance and position in the world.

3.3.1 WebSockets

In a game where a player wants to see other players interacting with the world, the player position needs to be continuously updated. These updates should occur within 500ms¹⁰, but the faster these updates occur, the more immersive the gameplay is. There are several techniques to transfer data between the server and the client, the two most important ones being Ajax and WebSockets. Currently, Ajax is the most widely used technique, simply because WebSockets were just recently introduced to HTML5 and are therefore not

¹⁰ [Claypool, 2006]

supported by the majority of an average website's visitors. The problem with this approach is that Ajax was built for the client to actively send or fetch data from a server. Since most events in a game are triggered by the server, the client would need to use Ajax requests in a way that allows the server to send events to the client. Ajax long polling, as this technique is generally referred to, continuously initiates Ajax requests, which the server holds back by not immediately sending an answer. When a server-side event occurs, the server propagates this event by sending it as the answer to an open Ajax request. Inherent to this method is a significant overhead both in network traffic and in code complexity, and a strongly varying, unpredictable latency between client and server, as at any given point of time, an open request may or may not exist and sending a client event may or may not require a new TCP connection to be established. WebSockets, on the other hand, offer full-duplex TCP based connections that can be used to both send and receive messages in the form of strings of almost any size. These connections are established by asking an HTTP server to upgrade a request to a WebSocket connection, similar to the way that HTTPS connections are initiated. Like Ajax, a WebSocket connection will have a small initial delay upon connecting, but instead of requiring a new HTTP request for each message, a WebSocket connection will remain open, waiting to send the data with the minimal overhead of at most 14 bytes per message ¹¹.

3.3.2 Asynchronous IO and the C10K-Problem

Although a permanent connection between client and server is generally a good thing, for example in regard to the delay between triggering a server-side event and actually seeing the effect on the client's machine, it may impose scalability issues to the server, depending on the server's architecture. A regular HTTP server handles short requests and closes the connection after a relatively short amount of time, usually within several seconds, while a WebSocket server has to keep every single connection open for minutes or even hours. This is a serious problem, because apache, one of the most commonly used web servers ¹², implements the classical, blocking IO-model and spawns a new process for every connection handled. The idea of blocking IO is that a thread tries to read from an initially empty input stream and gets put to sleep (blocked) by the operating system, until there is data to read. This method implies that for every input stream (i.e. for every connected user) there must be one thread dedicated solely to reading from it. In contrast, asynchronous IO

¹¹ According to the current version of the WebSocket specification, namely [RFC 6455].

¹² [Netcraft, 2013]

has a single thread looping over all the available input streams, checking whether individual streams have data available to be read and, if that is the case, reads the data. This concept dramatically reduces the overhead in memory and CPU usage by removing the necessity to spawn new threads or processes as the pressure on the server increases. The overhead of new connections is mostly reduced to letting the operating system create new sockets, which is the required minimum to accept a new connection. Due to the reduced overhead on handling connections, asynchronous (or non-blocking) IO allows the server to handle significantly more simultaneous connections, thus addressing the apparent bottleneck of most web servers¹³. This so-called C10K (“connection 10.000”) problem¹⁴ means that a web server may fail to serve thousands of connections (more than 10.000) simultaneously because the system’s memory would get cluttered with information about thousands of threads reading from thousands of input streams. Using the asynchronous IO model, a server is able to fully utilize its network connection until eventually the server’s CPU becomes the bottleneck¹⁵.

Servers following the blocking IO model are also vulnerable to the “thundering herd” problem¹⁶, which states that whenever a new connection request is detected, every thread that is listening for new connections is woken up by the operating system, despite the fact that only one thread can actually fetch the resource and handle the connection. The overhead introduced by the unnecessary waking of threads has been identified as a potential bottleneck in high-load server performance by [Molloy, 2000]. As a side-note, the term “thundering herd problem” is probably related to the inefficient movement of wildebeests crossing the Mara River in Africa.

3.3.3 Advantages of a Modular and Asynchronous Architecture

JavaScript itself is an event-driven language, and most of the newer APIs reflect the asynchronous nature of this scripting language. While an asynchronous architecture feels quite natural in terms of code design, when considering performance, there are also certain properties of the optimizing compilers that promote this style of coding. A modular architecture and event-based coding style encourages the use of more and smaller functions. Since compilers optimize on function level, this increases the chance for any

¹³ [Welsh et al., 2000]

¹⁴ First formulated by [Kegel, 2006]

¹⁵ According to [Welsh et al., 2000]

¹⁶ [Stevens, 1998]

part of the code to be optimized, because lines of code that prevent optimization or cause a function to get de-optimized affect less code. Although this may not be true for every JavaScript engine, Chrome does take into account how much code a function contains when deciding whether or not to optimize a function.

In addition to the technical aspects, it is important to keep in mind that computer programs are written by human beings and will almost certainly contain lines of code that perform poorly. Modular, loosely coupled coding models increase the testability¹⁷ and thus increase the developer's chances to identify performance bottlenecks or otherwise improvable code. Also, when looking at what functions are optimized and what functions are not, it is easier to identify the line of code preventing the optimization process, if any function is kept as small and simple as possible.

3.3.4 Communication through WebSockets

In its current version, the WebSocket protocol provides two message types for sending either plain text or binary data. The browser API is kept very close to the low-level nature of WebSocket communication itself, so the basic implementation found in browsers offers only one message-related event that is triggered whenever any message arrives. Although it is possible to register several listener functions to this event and let every listener decide whether to handle individual messages, there would still need to be some sort of agreement on how to identify messages in order to correctly process them. The game created in the course of this thesis uses an arbitrary protocol which applies a message identifier to each message sent and allows any type of variable to be sent. To keep the protocol simple, each message is represented by an array, containing the message identifier as its first element and the value that is to be transmitted as its second element. This array is then serialized using the built-in function `JSON.stringify` and deserialized by the receiver using `JSON.parse`. Attempting to blend in with the asynchronous architecture of JavaScript, the protocol's implementation distributes the messages by letting any part of the source code register directly to certain message identifiers. As an example, a function inside the chat module listens to the "chat" message identifier, while the game logic registers a listener to the "position update" message identifier. Each registered listener will only see the relevant messages, which weakens the coupling between independent parts of code, increases the modularity and thus reduces the overall code complexity of the application.

¹⁷ [Testable JavaScript, 2013]

To call the listeners in an efficient way, and to integrate well with other parts of the code where listeners are used, the protocol's implementation relies on the event-module supplied with node.js.

3.4 Advanced Optimizations for V8

The performance characteristics of JavaScript differ between browsers and browser versions, which generally means that some performance optimizations may actually slow down the script in a different browser or a different version of the same browser. Therefore, optimizations that focus on specific mechanics and capabilities of a single compiler engine should have a low priority in the overall optimization process, and should be kept as loosely coupled to the rest of the code as possible, to allow the code to be easily replaced if the optimization becomes obsolete. The following sections will take an in-depth look at how the V8 JavaScript engine is affected by certain optimizations. Other browsers may behave in a very similar pattern, yet only V8 was specifically analyzed and benchmarked in the following sections, unless otherwise noted. The reason to choose V8 was that it offers a stand-alone version that runs without the Chrome browser, and provides advanced debugging modes giving access to both the result of a high-level analysis as well as the actual machine code generated by the compiler. Crankshaft, as V8's optimizing compiler is called, uses a static single assignment (SSA) form to represent the basic code blocks of the JavaScript code. This form is internally called "Hydrogen" and is used to apply further high-level code optimizations, like function inlining and variable unboxing (applying a fixed type to a JavaScript variable). In addition to the Hydrogen intermediate representation, V8 is also capable of outputting the actual machine code generated by Crankshaft, which contains several comments including the block names used during the Hydrogen phase. In conclusion, the Hydrogen output may be analyzed to look at how the original code's control structures are interpreted and optimized, while the machine code reveals how much computation each (Hydrogen-) block actually imposes on the CPU.

3.4.1 Self-modifying Code

Historically, the "eval" function has always had a bad reputation. The parsing overhead caused any code using this function to be inherently slow, and many books¹⁸ even suggest never using the function at all. However, modern JavaScript engines can parse any script

¹⁸ e.g. [JavaScript: The Good Parts, 2008]

very fast, including code fragments passed to the `eval` function. Optimizations are done at function level, so functions created by the `eval` function will get optimized by the just-in-time compiler, if they meet the requirements for triggering the optimization process. If a JavaScript engine spends a certain amount of time executing such functions, they will be marked for optimization, which means that JavaScript may create certain parts of code that can be more easily optimized by the just-in-time compiler.

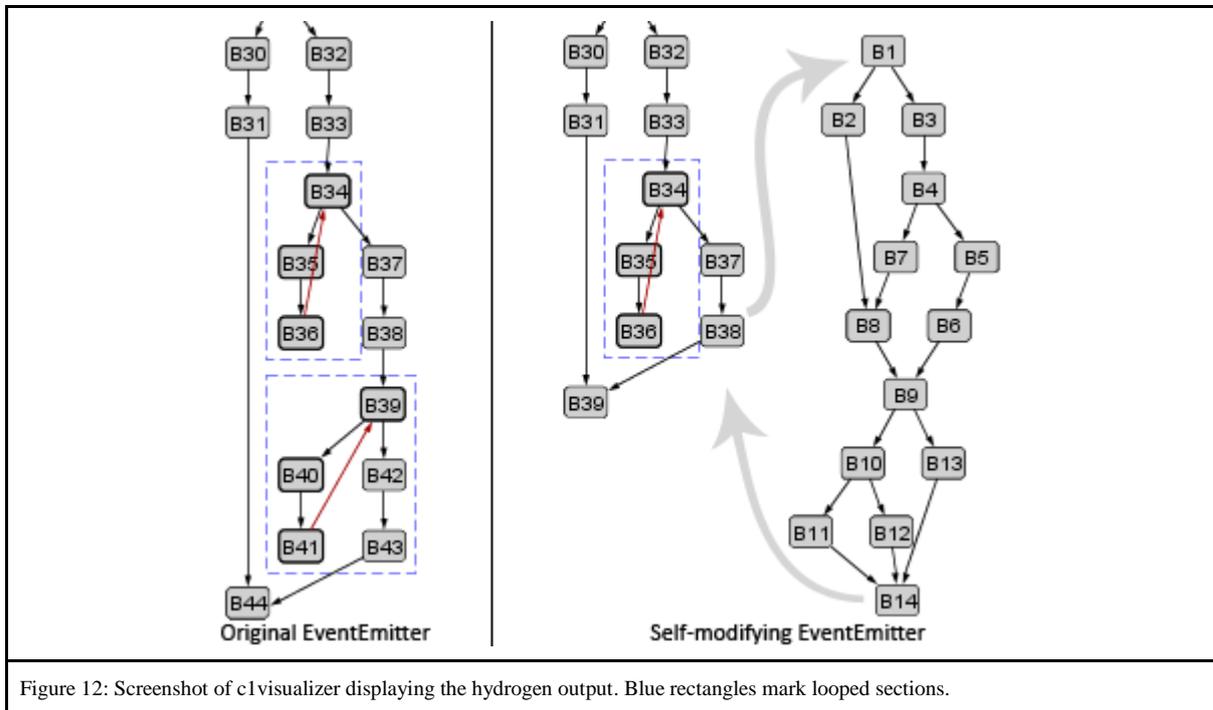
It should be said that the main purpose of self-modifying code in JavaScript would be to avoid code patterns that today's JavaScript engines are not yet optimized for. As this style of coding naturally introduces a high amount of code complexity, it should be used with caution. Browsers are constantly updated, and any effort in creating self-modifying code may be rendered useless if the browser's JavaScript engines are updated to optimize the avoided code pattern.

The game created for this thesis utilizes the `EventEmitter` class supplied with the `node.js` framework at several locations to dispatch events by invoking listener functions. Many of these instantiated event emitters are used in a similar way, where they initially attach several listeners, usually one to three listeners per event, which are then called frequently. The way the supplied class internally handles this is by storing those listeners in an array, and then iterating over that array whenever an event is to be emitted. Although JavaScript engines are able to inline functions, they are commonly unable to inline functions that are called in a semi-dynamic pattern, such as iterating over an array. I have created a modified version of the supplied `EventEmitter` class, which uses the `eval` function to manually unfold the for-loop, replacing it with static calls to fixed indices of the array. This allows the JavaScript engines to call statically addressed functions instead of looping over an array of function pointers, at the cost of creating and parsing a piece of JavaScript code whenever an event listener is registered.

Analyzing the output of V8's high-level optimizer¹⁹, the Hydrogen intermediate representation, in `c1visualizer`, as seen in figure 12, proves that this method does in fact reduce the number of loops, yet it also reveals that this improvement comes at the cost of additional instruction blocks. While the original for-loop produces a fixed amount of code, the optimized version creates more code as the number of listeners attached to the event increases (in linear proportion). The intermediate representation also shows that V8 refuses

¹⁹ A full copy of the captured results can be found on the CD (see appendix A.4 on page 37).

to inline the generated method, which may possibly be attributed to the function’s code size.



Besides the hydrogen analysis, it is also possible to track function optimizations and de-optimizations (“bail-outs”), and to display reasons for each operation. These hints revealed that no function containing a call to “eval” is optimized by V8’s just-in-time compiler. Although this is not surprising, it does strengthen the suspicion that the “addListener” function, which creates the source code for part of the “emit” function, would perform very poorly, compared to the unoptimized version.

To evaluate the actual performance, I have created an example implementation of the suggested algorithm as well as several (arbitrary) test cases²⁰. The first test case, where the emitted event only calls a single event handler, serves as a control, because this use-case is already optimized in the original EventEmitter code served with node.js. The expectedly different results for both the original and the self-modifying emit function are most probably caused by the fact that the latter function consists of a larger amount of code. To evaluate the effect of self-modifying code, the tests after the first test each call one additional listener function more than the previous one, up until five, at which point the overhead in code size balances out the drawback of looping over an array of function

²⁰ For the measured results, see appendix A.3 on page 36. The test script as well as the captured results can be found on the enclosed CD (appendix A.4 on page 37).

pointers. As a result, the self-modifying EventEmitter introduced a 68% increase in the time needed to attach the listener functions, while only causing a significant performance increase for events with two attached listener functions, although this performance increase amounted to a substantial 39%.

3.4.2 A Look at V8's Generated Assembly

During the compilation process of V8, the open source JavaScript engine used in the Chromium project, it is possible to set certain parameters, enabling the compiled V8 shell to output the generated assembly. Whenever V8's just-in-time compiler optimizes a function, V8 writes the generated assembly in a human-readable format to a file, including many helpful code comments. Thus, it is possible to see which functions are updated at what point of time, and how well V8 does at optimizing JavaScript. Likewise, it is also possible to understand why certain constructs cause significantly slower code.

As of the time writing, measurements suggest that Chrome is currently the only browser to optimize functions containing a `try...catch` block, probably not because these structures create code that is hard to optimize, but more likely because there were more important topics on the agenda of the developer teams. This simple example illustrates the most prevalent issue with most JavaScript engines: Although JavaScript itself dates back to 1995 ²¹, some browser's optimization techniques still look immature, because some code patterns that just slightly differ from others remain unoptimized.

To look at V8's assembly, I examined a simple algorithm computing matrix multiplications. The most obvious conclusion from this work was that V8 optimizes the same function several times, which may be due to several reasons. One reason is that V8 converts arrays of variable type to typed arrays, so if a developer manually uses typed arrays of uneconomical types (for example an `int32` array for storing numbers between 0 and 255), the code actually runs more slowly than leaving the choice of type to V8. Once an array is converted to a certain type, any optimized function may be further optimized, as V8 then knows more about the variables and values used inside the function. Another reason for repeated optimizations is that V8 is sometimes too optimistic at optimizing a function or variable. If this happens, the function or variable is apparently reverted to its completely unoptimized state, which V8 may then try to optimize again. In its final optimized version, the function for multiplying matrices took slightly less than twice the

²¹ [Rauschmayer, 2012]

time that a native C application required to compute the results. Even though a JavaScript variable is capable of holding any type of value, from number to object, V8 generally did a good job unboxing these, meaning that V8 actually stored the numbers as integers instead of objects with attached type checks.

The generated assembly no longer contained several code paths that handled different types of variables, but instead contained simple checks, whether or not the values were of the expected type. If an incompatible type was found, the code would branch out of the function into some V8 code, which then completely deoptimized the function. According to the code comments next to the generated assembly, these type checks are internally called “bail-outs”. Usually, those comparisons nearly always resolve to “false”, which allows modern CPUs with sophisticated jump prediction units and many pipelining stages to nicely pipeline the function causing these statements to not have such a strong impact on the overall performance of the script.

3.4.3 Optimizing Compiler

While the previous sections about V8 looked at how its optimizing compiler handled specific code, it is also important to look at how well the gathered knowledge helped at creating a well-performing HTML5 game. Google Chrome offers command-line options to pass configuration parameters to the V8 JavaScript engine, and others to disable security features. The browser consists of several threads and separated processes to achieve high security standards, but this also means that processes like V8’s JavaScript shell do not have access to the original file descriptors, so that V8 could not output any information to the standard output or the file system. Therefore, Chrome must be run with disabled security features like sandboxing, if V8 is supposed to log any information. In this case, the desired output was a log of when and why V8 optimizes or de-optimizes JavaScript function. Running the final version of the Canvas2d-game with the necessary command-line options revealed that an important JavaScript function was constantly being optimized and de-optimized while the game ran. The function was responsible for drawing the background of the game’s scene, taking the player’s current position as arguments in the form of two numbers, x and y. To optimize the function, V8 kept track of the types of arguments the function was called with, which were, according to the game’s code design, floating point numbers. A look at the logged output showed that V8 actually called the function with altering argument types, as the details given when a function was marked for optimization stated “ICs with typeinfo: 9/43 (20%)”. The reason for this is in the way that

the new character positions are calculated, because whenever the character moves horizontally/vertically, the y-/x-coordinate was rounded, causing V8 to optimize this variable from floating point to integer. Although integer operations are generally faster than floating point operations, this “optimization” required another part of code (the paint function) to handle constantly altering variable types. Consequently, the aforementioned paint function was not always called with floating point numbers as arguments, but rather with a random combination of floats and integers. Whenever the character’s direction of movement changed, the compiler would de-optimize the heavily used paint function. As it then takes a little while to gather the information, and again a little more to actually compile the optimized function, the seemingly unimportant issue of sometimes using integers instead of floating point numbers introduced a significant threat to the game’s performance.

This revealed yet another situation where the optimizing compilers were unable to optimize a function that appeared to be very simple and did not feature any complex code design. V8 obviously lacks the ability to use floating point numbers as a fallback, whenever optimizing floats to integers results in an unstable state that continuously leads back to the state where the value was stored as a floating point number.

4 Conclusions

JavaScript's overall rendering performance was good, and any popular browser is capable of rendering the scene fast enough to let the game run at most monitors' default frame rate. There is a lot of space for complex game logic, as the data shows that of the available 16.7ms (for a 60 Hz monitor) less than half a millisecond is actually spent inside the paint function for this simple game. Generally, most browsers did well in optimizing the JavaScript code, while certain code patterns are not yet optimized as well as they could be, simply because their compilers are not yet optimized for those cases. An example of this immaturity would be the Internet Explorer's implementation of the `Math.round(x)` function, which runs around 33 times more slowly than a manual implementation²².

4.1 Rendering Performance

When painting several small images onto a screen, there is a small delay for passing the image to the rendering thread, and additionally each call produces a small amount of garbage. The best performance, in terms of raw rendering power, was achieved by the Microsoft Internet Explorer. However, painting many small images causes a certain overhead for any browser, and an improved version diminishes the difference between the browsers to an insignificant amount. The proposed optimization was to buffer the paint calls, drawing the background of the scene onto a separate canvas. Small changes in position were then displayed by moving this background-canvas, rather than by repainting the whole scene with a slight position offset for each image. Although all browsers perform very well in rendering-related tasks, it is still advisable to minimize the amount of calls to rendering functions, for example `Canvas.drawImage`. Still, there is no significant disadvantage in keeping several off-screen images in memory for a longer amount of time. In general, using CSS transitions allows browsers to render more efficiently, but apparently only Google Chrome actually leverages the implications that CSS transitions impose.

4.2 What Gaps are not yet Closed

There are several parts of the JavaScript language that should not be used. The problem is not that they are not intended to be used or cause a high amount of code complexity. The issue here is that the JavaScript engines optimize at function-level, meaning that they

²² The manual implementation used a 0-bit-shift ("`>> 0`") to implicitly convert the number to an integer.

either optimize a function, or that they do not - they are unable to optimize only parts of a function. Therefore, if the optimizing compiler stumbles upon a code pattern it does not fully understand, the whole function will remain completely unoptimized. Code patterns that were found and proven to cause functions to not optimize are “`eval(..)`”, “`try .. catch`” and sometimes “`Array.forEach(..)`”, although some of the patterns may be optimized by some of the tested browsers. Besides those triggers, a function may be slowed down if the developer stores different types of objects or values in the same variable, because this keeps the compiler from efficiently optimizing that part of the code, although the impact is far less prevalent than the above code patterns.

In terms of keeping up with native programs, the worst part about JavaScript is that no browser offers an intuitive way to inform the developer about these code-patterns or display reasons for deoptimizations. JavaScript does have the potential to be used in time-critical applications like games, but if browser vendors want to promote those use cases, I strongly suggest adding said notifications to their developer tools. A programmer cannot correct code errors that he or she is not aware of.

4.3 Is the Nature of JavaScript Slowing it Down?

The test cases created for this thesis, especially the matrix multiplication test, showed that JavaScript benefits from code optimizations in a similar way that native programs do. All popular browsers generally did a good job at compiling and optimizing JavaScript code. Even if not every code construct of the language can yet be transformed to efficient machine code, the current stage of JavaScript compiler development suggests that most gaps will eventually be closed. There is still a lot of room for optimizations to the way that JavaScript engines compile and run code, so it is doubtful, at best, that this is already the right time to look for a replacement for JavaScript like Google Dart or NativeClient. As for asm.js, I believe that it does not harm the JavaScript language, because it fits well with the model of backward-compatibility that has always been inherent to the web. JavaScript engines optimize certain code constructs in arbitrary ways, so it can be argued that optimizing asm.js code is merely an advanced optimization case for specific JavaScript, since even if, at a later point of time, asm.js turns out not to be necessary, and browsers no longer optimize these JavaScript code patterns, the code will still run in any browser. There are parts in the JavaScript language that currently cause some code to perform poorly, but the concept behind the language does not slow JavaScript down to an extent where a replacement for the language remains the only alternative.

4.4 What Key Elements is JavaScript Missing?

There are many features in other languages that JavaScript does not possess but that are doubtful to make any sense in the context where JavaScript is used.

Looking at the kinds of performance improvements achieved by recent microprocessors shows that chip manufacturers like Intel seem to optimize their processors for SIMD (single instruction multiple data) instructions²³. JavaScript does not yet offer any code constructs that could possibly be optimized in a way that leverages SIMD instructions, but it may be possible to utilize these instructions in “comprehensions”, which are in discussion to be part of the next version of JavaScript. In addition, there might be the possibility to add certain features to the asm.js framework that allow code to be optimized in a way that allows the use of SIMD instructions.

Besides said instructions, there is another important technique to reduce the pressure on the CPU, which is deferring computation to other parts of the computer. Several attempts have been made to allow JavaScript code to run on the GPU (graphics processing unit), which is generally optimized for highly parallel computation. In the context of web development, the GPU’s main purpose would be to run shader code, which can be marked for execution through CSS custom filters (formerly CSS shaders) or WebGL shaders in JavaScript. These shaders are not written in JavaScript, but instead use the OpenGL ES shading language maintained by the Khronos Group.

²³ [c’t 14, 2013]

Eidesstattliche Erklärung

Gemäß § 17 (5) der BPO erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Ich habe mich keiner fremden Hilfe bedient und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

Erklärung

Mir ist bekannt, dass nach § 156 StGB bzw. § 16 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

Ort, Datum

Unterschrift

Appendices

A.1 Resource Management

Measured using the “ResourceManagement.php” test case found on the enclosed CD (see appendix A.4). The test machine for the download test (caching disabled) was a desktop computer with Google Chrome installed, connected to the internet through a wired LAN connection and a 1.7 MiB/S broadband connection. In this arbitrary setup the average time to download the game’s files was $0,823 \pm 0,024$ Seconds over 20 measurements.

For the caching test (caching enabled), the average durations over 20 measurements were as follows.

Desktop PC (Samsung SSD 830 series) [MS]	Desktop PC (magnetic hard disk drive) [MS]	Mobile Device (Samsung Galaxy Nexus) [MS]
0,349 ± 0,105	1,655 ± 0,149	6,590 ± 0,251

A.2 Rendering a Scene

Time spent within the main rendering function in JavaScript using the final version of the game. The naive algorithm individually draws every tile on the same canvas, the buffered version draws the scene onto a separate canvas to update the content less frequently. The characters are redrawn at every function call. The standard deviation for Google Chrome cannot be calculated due to the inability to display the needed values inside the Chrome development tools. The values represent the average over a dataset size of 2000 samples.

	Google Chrome [% of measurement duration]	Mozilla Firefox [ms]	Microsoft Internet Explorer [ms]
Naive algorithm	2.27% ± ?	2.97 ± 0.17	0.95 ± 0.34
Buffered on separate on-screen canvases	0.15% ± ?	0.31 ± 0.08	0.24 ± 0.09

A.3 Self-Modifying Code

Measured using the script “eventhandling.js” which can be found on the enclosed CD (appendix A.4). The script was run in D8, the stand-alone version of V8 compiled with the “debug” command line option. Over 5 test runs, each method was invoked multiple times.

	Original EventEmitter (5 x 1,000 samples) [calls/S]	Self-Modifying EventEmitter (5 x 1,000 samples) [calls/S]
addListener function	4.639 ± 0.219	7.807 ± 0.134

The same test runs as above, showing the measurements of the emit function.

	Original EventEmitter (5 x 100,000,000 samples) [calls/S]	Self-Modifying EventEmitter (5 x 100,000,000 samples) [calls/S]
1 listener	28,722 ± 664	28,696 ± 175
2 listeners	5,685 ± 85	7,882 ± 48
3 listeners	9,813 ± 12	9,713 ± 299
4 listeners	3,903 ± 32	4,065 ± 20
5 listeners	3,348 ± 38	3,294 ± 14

A.4 Source-Code (CD)

References

[Claypool, 2006]

Mark and Kajal Claypool; *On Latency and Player Actions in Online Games*; July 2006
Worcester Polytechnic Institute, Worcester, USA

[RFC 6455]

Internet Engineering Task Force (IETF), I. Fette, Google, Inc., A. Melnikov, Isode Ltd.;
The WebSocket protocol; December 2011
<http://tools.ietf.org/html/rfc6455#page-28>

[Netcraft, 2013]

Netcraft Limited; *June 2013 Web Server Survey*; June 2013
<http://news.netcraft.com/archives/2013/06/06/june-2013-web-server-survey-3.html>

[Kegel, 2006]

Dan Kegel; *The C10K problem*; 2006
<http://www.kegel.com/c10k.HTML>

[Rauschmayer, 2012]

Axel Rauschmayer; *The Past, Present and Future of JavaScript*; O'Reilly Media; July
2012

[JavaScript: The Good Parts, 2008]

Douglas Crockford; *JavaScript: The Good Parts*; O'Reilly Media; May 2008

[Gartner, 2013]

Gartner, Inc.; *Mobile and Wireless Predictions Reflect Mobility's Impact on the
Broader*; February 2013
<http://www.gartner.com/newsroom/id/2324917>

[c't, 14/2013]

Benjamin Benz, Florian Müssig; *Marathonprozessor*; c't 14/2013; Heise
Zeitschriftenverlag GmbH; June 2013

[Welsh et al., 2000]

Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler; *A Design Framework for Highly Concurrent Systems*; April 2000
University of California, Berkeley, USA

[Tesse, 2013]

Jöran Tesse; *Performanz von Internetseiten*; March 2013
Fachhochschule Dortmund, Germany

[Stevens, 1998]

W. Richard Stevens, Bill Fenner, Andrew M. Rudoff; *UNIX Network Programming: Networking APIs: Sockets and XTI*; Prentice Hall International; January 1998

[Molloy, 2000]

Steve Molloy; *Accept() Scalability on Linux*; June 2000
Center for Information Technology Integration, University of Michigan, USA

[Testable JavaScript, 2013]

Mark Ethan Trostler; *Testable JavaScript*; O'Reilly Media; February 2013

[Cytron et al., 1991]

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck; *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*; October 1991
IBM Research Division and Brown University, Providence, USA

[Blom et al., 2008]

Sören Blom, Matthias Book, Volker Gruhn, Ruslan Hrushchak, André Köhler, „Write Once, Run Anywhere – A Survey of Mobile Runtime Environments“, May 2008
Applied Telematics/e-Business Group, University of Leipzig, Germany

[Computergrafik und Bildverarbeitung, 2011]

Alfred Nischwitz, Max Fischer, Peter Haberäcker, Gudrun Socher; *Computergrafik und Bildverarbeitung*; Vieweg+Teubner Verlag; September 2011