Acknowledgements

After a year of research around a subject that I have found interesting for a long time I would like to thank my promotor Prof. dr. Peter Quax and co-promotor Prof. dr. Wim Lamotte for giving me the chance to work on this subject as part of a non-conventional master-thesis and for the guidance throughout the year.

I also thank dr. Jori Liesenborgs for his help and support with the implementations and the feedback on my work, and Arno Barzan for proof-reading and providing valuable feedback on drafts of this text several times which helped me further refine this text.

Special thanks also go to Peter Beverloo who originally sparked my interest in browser development and for his feedback and help guiding me through the more complex parts of the Chromium and Blink source-code.

Finally, I would of course like to thank my family. Especially my parents who stuck with me through my somewhat eventful school career.

Thanks!

Abstract

The World Wide Web, traditionally only used for static content, has grown in size, reach, and capability over the past few years. Web pages have turned into full featured web applications and web browsers have dramatically improved their interoperability, performance, stability, and security. This means developers can move more and more advanced applications to the web. This move initially started with simple applications like e-mail clients. Nowadays, developers can use new web standards to move very complex and high-performance applications to the web as well.

In this thesis we perform a case-study of the state of the web as an application platform, with a focus on performance. We start with an in-depth analysis of the architecture of a modern web browser and try to understand key architectural and performance concepts. We also look at the developer tools available in modern web browsers. Some aspects of the internal workings of the Google Chrome web browser are investigated in more detail as well: the performance of a *getImageData* call on a hardware accelerated canvas and the garbage collection of *WebGLRenderingContext* objects.

We also propose and implement two proof of concepts. The first extends WebGL with support for text rendering and handwritten lines (annotations). The second implements a remote WebGL rendering solution where the actual rendering of a WebGL canvas is done by a remote system and the result is streamed back to the client. The goal here is to make complex WebGL applications usable on low-end devices.

Contents

Chapter 1

Introduction

The World Wide Web, invented by Tim Berners-Lee at CERN in the late 20th century, originally started as a simple text based internet service. The first web browsers had a very limited feature set and were only able to display text and images with basic styling. Since the original introduction, the web has grown immensely with the help of new technologies such as JavaScript. There are an estimated 4.2 billion web pages on the web right now[22]. A lot of new features have been added to the web and the browsers used to access it[65] in recent years. For example things like WebGL, WebSockets, or WebRTC enable advanced graphics and bi-directional communication on the web and are used throughout this thesis.

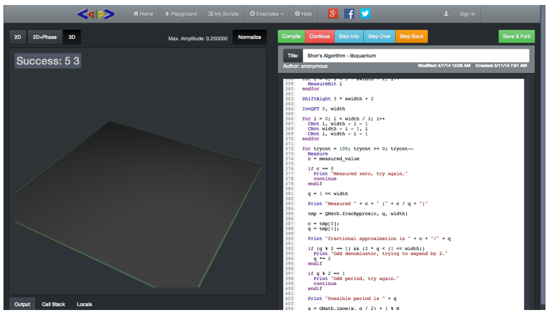
These new features turned the World Wide Web in an attractive application platform. A couple years after the first introduction, developers started porting simple desktop applications to the web. Examples of these are Hotmail and Gmail, two web-based email clients. Further advances in browser performance (JIT JavaScript engines) and technologies like WebGL opened the door for high-performance and advanced graphics applications. Examples include a web-based version of the popular Angry Birds game (figure 1.1b) and the JavaScript port of the Unreal Engine 3 by Mozilla and Epic[51]. Non-entertainment examples include The X Toolkit for scientific visualizations[15] and Google's Quantum Computing Playground (figure 1.1a).

1.1 The Web as an Application Platform

As the web continues to grow, more and more types of applications will be ported to the web. To support these applications a lot of new standards have been developed in the the last 6 years[65] and browser vendors moved to just-in-time JavaScript compilers and GPU accelerated rendering. Now is a good time to discover what problems developers encounter when porting applications to the web, identify the underlying causes, and look for possible solutions. Performance of web applications has become very important with the recent rise in mobile web-traffic[58]. Mobile devices have very limited hardware resources and battery life. At the moment most applications are still developed for native platforms[71]. Although the web has always had some advantages over native platforms, a lot of work is being done to make the web more attractive to application developers.

1.1.1 Cross-Platform

The fragmentation in operating system usage and the adoption of mobile devices[57] means developers have to develop their applications for multiple platforms in order to support a large user base. This is a huge disadvantage of all native platforms: applications have to be build for



Title: Shor's Algorithm - libquantum
Author: anonymous
Modified: 4/7/14 12:28 AM Created: 3/11/14 7:51 AM

```
  MeasureBit i
endfor

ShiftRight 3 * swidth + 2

InvQFT 0, width

for i = 0; i < width / 2; i++
  CNot i, width - i - 1
  CNot width - i - 1, i
  CNot i, width - i - 1
endfor

for trycnt = 100; trycnt >= 0; trycnt--
  Measure
  c = measured_value

  if c == 0
    Print "Measured zero, try again."
    continue
  endif

  q = 1 << width

  Print "Measured " + c + " (" + c / q + ")"

  tmp = QMath.fracApprox(c, q, width)

  c = tmp[0];
  q = tmp[1];

  Print "fractional approximation is " + c + "/" + q

  if (q % 2 == 1) && (2 * q < (1 << width))
    Print "Odd denominator, trying to expand by 2."
    q *= 2
  endif

  if q % 2 == 1
    Print "Odd period, try again."
    continue
  endif

  Print "Possible period is " + q

  a = QMath.ipow(x, q / 2) + 1 % N
```

(a) Quantum Computing Playground is a browser-based, GPU-accelerated quantum computer with an IDE and its own scripting language with debugging and 3D quantum state visualization features.[21]

(b) A web-based version of the popular Angry Birds game. The application works in desktop and mobile browsers and has the same features as the native versions.[56]

Figure 1.1: Two WebGL based web applications. Both use WebGL and advanced JavaScript game engines.

multiple, often closed and proprietary platforms. Because the web is cross-platform developers only have to build and support one version of their application. This cuts down on development costs and saves time. This time can be invested in new features and other improvements instead. The old and well known browser quirks, most notably those in Internet Explorer, have disappeared.

1.1.2 Developer Tools

All native platforms have advanced developer tools, often integrated into platform specific Integrated Development Environments (IDEs). The web has lacked proper developer tools for years. Debugging a web application had to be done with alert statements (printf debugging for JavaScript) which was far from ideal. This made the development of large web application a burden. All modern web browser nowadays ship with built-in developer tools, some more advanced than others.

1.1.3 New Web Standards

Browser vendors and web authors teamed up at the *World Wide Web Consortium[74]* and the *Web Hypertext Application Technology Working Group[76]* to develop new web standards that bring native capabilities to the web in a usable, performant, and secure way. These efforts have led to several new standards that brought support for bi-directional communication (WebRTC, WebSockets), advanced graphics (WebGL), parallel computing (WebCL), accelerated video decoding (Video element), threading (WebWorkers), and many more to the web. Some browser vendors also opted for shorter release cycles (Chrome for example ships new versions every 6 weeks) to make these new features available to end users faster.

1.1.4 JavaScript

The only programming language available to web applications is JavaScript. JavaScript is an ECMAScript[16] implementation. ECMAScript is a dynamically typed, classless language. Old browsers used an interpreter to execute JavaScript. This made web applications a lot slower than native applications because native applications can be compiled. Property access for example is a lot slower in JavaScript because JavaScript does not has the concept of a *class* which gives structure to objects. In JavaScript, the structure of objects can be changed dynamically during program execution. This means a slow hashmap lookup is required to find the exact location of a property in an object. On native platforms, the compiler can determine the exact location of each property in an object during compilation because the objects have a fixed structure. This is one of the reasons why browsers now use Just-In-Time compilers to execute JavaScript.

1.1.5 Distributing and Updating Applications

Distributing a web application is as simple as providing a Uniform Resource Locator (URL). Everyone with access to the internet can use the same URL to access a web application. No installation is required, it just works. Because all users use the same URL to access a web application they will also always use the latests version of the application. Uploading or deploying a new version of a web application makes it available to all users immediately. Facebook for example deploys new versions daily and GitHub deploys new versions hundredths of times each day without their users noticing[19].

1.2 Performance on the Web Platform

The web has always been criticized as being slow. Because the web is a complex environment this slowness has multiple causes. There are three aspects to web performance. On one side there is back-end performance. The back-end stores all resources (images, CSS & JavaScript code) and application data. The server has to transfer all this to the client which introduces a network component. Once all the data and code have arrived at the client, there is the client-side performance as well.

Focussing on the performance of web applications should be an important aspect of the development process. Multiple studies and experiments have shown that the slower a website loads, the lower the conversion rate and user-satisfaction will be[39]. Users spend less time on slow a website and sometimes leave before the application is usable (fully initialized). This makes performance optimization a commercial benefit. Although human eyes have no refresh rate, recent research has shown that 60 frames per second is ideal for a smooth experience of motion, or in our case, interaction[53]. Applications with frame rates below 24fps are perceived as slow.

The recent rise in mobile web usage makes a further case for performance optimizations, mobile devices have less powerful hardware than desktop machines and will benefit greatly from optimizations. Battery usage is a large factor here as well, unoptimized code unnecessarily drains the user's battery. A lot of work is being done the make the web platform faster. Some of this has already been mentioned but there is more.

1.2.1 Cloud Computing

Cloud Computing is a new technology that was introduced to mainly improve the back-end and network performance. Cloud computing is a very broad term but for web applications the focus is on Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Content Delivery Networks. These technologies can be used to serve web applications which then become Software as a Software (SaaS). Google App Engine (GAE) is a prime example of a PaaS. A developer only has to deploy his application to GAE and it will take care of the rest. GAE provides applications with a (limited) set of interfaces to communicate with external services. Applications on GAE scale automatically based on a Queries per Second (QPS) metric. QPS are the number of incoming requests per second. This automatic scaling is cost efficient and keeps the back-end of the applications fast regardless of the current load.

Amazon's Elastic Compute Cloud (EC2) platform is a nice example of an IaaS. An IaaS provides infrastructure rather than a platform. This means the developer has more control over his application and external services. This additional control comes at the cost of more management: IaaS does not scale automatically. The developer is responsible for providing the scaling logic. Both IaaS and PaaS have the same goal: keep the application back-end fast even when sudden traffic spikes occur.

Google and Amazon (among others) also have a public content delivery network. CDNs are designed to serve static content like JavaScript files or images efficiently to users. CDNs distribute content geographically so that the content is as close to the end-user as possible. This speeds up the loading of static resources because the number of hops the data has to pass through is very limited. CDNs are designed to speed up the network aspect of the web platform.

1.2.2 HTTP/2 and QUIC

Hypertext Transfer Protocol version 2 (HTTP/2)[47] and Quick UDP Internet Connection (QUIC)[26] are two new network protocols designed to speed up the networking aspect of the web platform. HTTP/2 is currently being standardized by the Internet Engineering Task Force (IETF). Its design is based on the SPDY protocol invented by Google and aims to replace the outdated HTTP/1.1 protocol. HTTP/2 is a major upgrade from HTTP/1.1 and features things like multiplexing, flow control, prioritization, improved security, interactive mode where the server can push resources that were not yet requested, and header compression. Although the use of encryption is not mandatory in HTTP/2, Google (Chrome) and Mozilla (Firefox) have decided to only support the encrypted version.

The QUIC protocol is being developed by Google but has no official specification yet. Google is already testing it in production and it has shown promising results. It builds further on the advances made with HTTP/2. QUIC for example does require the use of encryption and because it is built on top of the User Datagram Protocol (UDP) there is no head-of-line blocking between multiplexed streams. Head-of-line blocking occurs in HTTP/2 when a TCP fragment is lost during transit. The fragment has to be retransmitted before the rest of the stream can be processed. This will block all multiplexed streams in HTTP/2. QUIC does not have this problem because UDP is an unreliable protocol which means QUIC has to take care of data integrity and therefore can handle the streams independently on a higher level. Both these protocols speed up the networking aspect of the web platform.

1.2.3 Analytics

Measuring and monitoring web application performance is important because without performance data it is not possible to optimize properly. A lot of tools are available to measure, collect, and analyse performance data. Client and network performance for example can be measured with the new Navigation Timing API[77]. Advanced analytics platforms like Google Analytics can be used to collect, store, and compare performance data over time. These tools are very useful for detecting performance regressions. The same tools can also be used to identify bottlenecks so that those can be fixed. There are tools to monitor and analyse the performance of the back-end as well.

1.3 Why Chrome

A large part of this thesis consists of Chrome specific research. The decision to use Chrome as the main web browser is based on several factors:

- **Multi-process architecture**: in Chrome, each tab runs in a dedicated, sandboxed renderer process. This is good for performance, stability, and security;
- **Developer Tools**: Chrome ships with advanced debugging and analysis tools. Almost every aspect of the browser can be analysed;
- **Open-source**: the entire source code is available through the Chromium open-source project[60];
- **Web standards support**: Chrome has very good web standards support and is pushing new technologies, like WebRTC, that are required for some of the proof of concepts in this thesis.

Only the research is specifically focused on Chrome. All code written as part of this thesis will work in all modern, standard-compliant browsers.

1.4 Research Questions

The goal of this thesis is to look at some problems encountered when porting native applications to the web. Specifically, the goals are to identify problems, analyse them to identify the underlying causes, and where possible, propose solutions. The in-casu application is a visualization tool for detailed, high-resolution image data. The applications performs operations (translations, pixel manipulations, etc.) on the image data and visualizes the results from different viewpoints. The visualizations may not contain artifacts and the application has to be very responsive as well. While the focus here was on a specific type of application, most of the findings are true for all web applications. To achieve these goals we formulate the following research questions;

1. What tools, technologies, or other browser features are available to web developers today, and more specifically:
   (a) how do these tools, technologies, or features help developers write high-performance applications for the web;
   (b) how do these tools, technologies, or features compare to those available on most native application platforms;
   (c) are there new issues caused by these tools, technologies, or features, what is the impact of these issues, and can they be resolved.
2. Is it possible to run (high-performance) WebGL applications on low-end devices by emulating WebGL support. The emulation has to be:
   (a) transparent to the user;
   (b) perform well (fast enough to make it usable);
   (c) scalable.

1.5 Conclusion

Because we try to answer two distinct research questions, a double approach is called for. To answer the first research question, we will do a **case-study** of the available tools and technologies. This case-study encompasses chapters 2, 3, 4, and 5. Chapter 2 shortly introduces some new web standards and technologies. Chapters 3 and 4 study important architectural features of Google Chrome in both Blink (the rendering engine) and V8 (the JavaScript engine) to see how these improve the overall performance of the browser and web applications. Chapter 3 takes an in-depth look at memory management in Chrome, specifically the garbage collection for *WebGL rendering contexts*. Chapter 4 further investigates the performance of *getImageData* (part of the canvas API) calls on accelerated canvas. Chapters 5 proposes a solution where a 2D canvas is combined with a WebGL canvas to simplify the rendering of text and handwritten lines.

The second research question is answered in chapter 6 where we conduct a **feasibility study** to see whether it is possible to emulate WebGL on low-end devices. This feasibility study results in a proof of concept which describes a system that acts as an almost full drop-in replacement for the built-in WebGL rendering context on low-end devices. This system is described and analysed in chapter 6 as well.

Chapter 2

Introducing New Web Technology

This chapter provides a short introduction to new web standards and technologies that are used throughout the rest of this thesis. Understanding the basic concepts and ideas behind these technologies will help the reader understand the remaining chapters. Only very recent and not yet widely known technologies are introduced here.

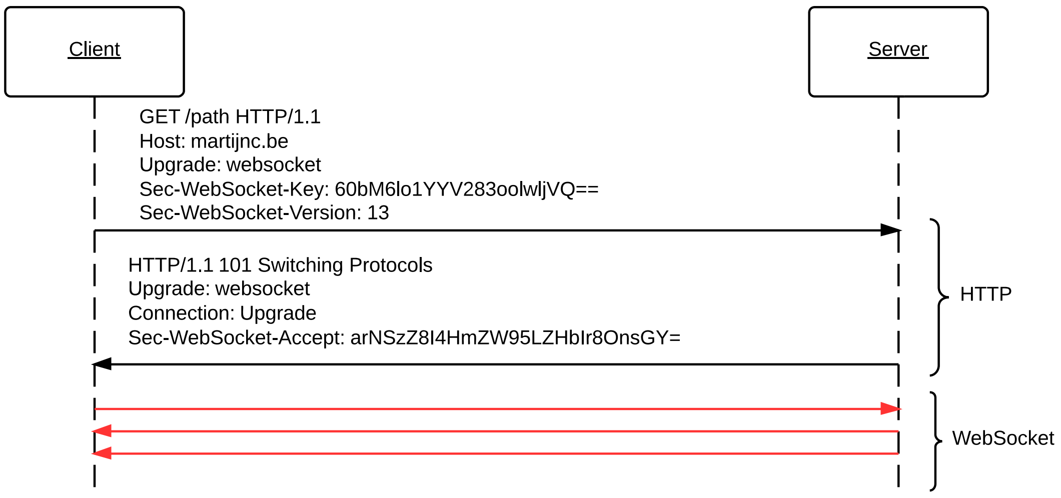
2.1 WebSockets

The WebSockets protocol provides direct bi-directional communication between a client and a server. A WebSocket connection runs over a *Transmission Control Protocol* (TCP) connection and provides a TCP like socket programming interface to web applications. TCP is a protocol that provides reliable bi-directional connections between systems and is one of the basic protocols of the internet. With WebSockets, a web application can send and receive binary data over a data channel, this eliminates the need for the less efficient *Asynchronous JavaScript and XML* (AJAX)[4] based long polling technique. Web applications can use AJAX to request web pages from JavaScript in the background without needing to navigate but the technology was never intended for bi-directional communication.

Long polling is a technique that uses long lasting AJAX requests between client and server so that the server can send data to the client immediately. The client connects to the server using an AJAX request and both parties keep this connection alive until the server has data it needs to send to the client. The server simply places this data in the response body of the AJAX request. At this point the client needs to re-connect to the server using a new AJAX request. In the pre-WebSockets era this was the only way to have real-time communication from server to client.

2.1.1 WebSocket handshake

A WebSocket connection starts its life as a traditional HTTP request. The normal HTTP request flow is used for the WebSocket handshake. The client starts a WebSocket handshake, displayed in figure 2.1, by sending a HTTP upgrade request to a HTTP/WebSocket server. A WebSocket capable HTTP server will process this request and reply with the *HTTP 101 Switching Protocols* status code indicating to the client that it accepts the WebSocket connection. The original HTTP request is now upgraded to a WebSocket connection and the handshake is complete. Both endpoints can start using the connection as a bi-directional WebSocket data channel. An HTTP server can reject the upgrade request from the client by replying with any other HTTP status code. By using an HTTP Upgrade request, one port can be used for running both



```
GET /path HTTP/1.1
Host: martijnc.be
Upgrade: websocket
Sec-WebSocket-Key: 60bM6lo1YYV283oolwljVQ==
Sec-WebSocket-Version: 13
```

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: arNSzZ8I4HmZW95LZHbIr8OnsGY=
```

Figure 2.1: The WebSocket handshake starts with the HTTP protocol and is upgraded once the handshake is complete.

the HTTP server and the WebSocket server. A WebSocket connection can be encrypted using TLS[36].

2.2 WebRTC

WebRTC is a technology that is being developed jointly by the W3C, WHATWG, and the IETF[37] and intents to enable real time, peer-to-peer communications in the browser. Google has been a major player in the development of WebRTC and is pushing for the adaption by open sourcing their own implementation. WebRTC has three main components:

- The **MediaStream API** provides access to synchronized media streams. JavaScript provides an interface to get a MediaStream object with synchronized video and audio streams from a user's webcam. The MediaStream objects take care of encoding, decoding, bandwidth management, encryption, signal processing, and codec handling. Both clients mutually agree on what codec they will use. For video, the VP8 codec is most widely used. The Opus codec is preferred for audio.[32, 31]
- The **RTCPeerConnection API** provides a WebRTC based peer-to-peer connection between two clients that can be used to send synchronized streams (through a MediaStream object) over a network. RTCPeerConnections are build on top of the Stream Control Transmission Protocol (SCTP)[55] (general data transport) or the Secure Real-time Transport Protocol (SRTP)[46] (streams). The ICE/STUN protocols are used to aid with connection setup.
- The **RTCDataChannel API**, like the RTCPeerConnection, provides a WebRTC based peer-to-peer connection between two clients but on a lower level. The RTCDataChannel can be used to send any type of data between two clients, not only MediaStream objects.

2.2.1 Connection Setup

Although the main principle behind WebRTC is to create peer-to-peer connections, a central server is required to handle the connection setup. WebRTC connections are setup by exchanging

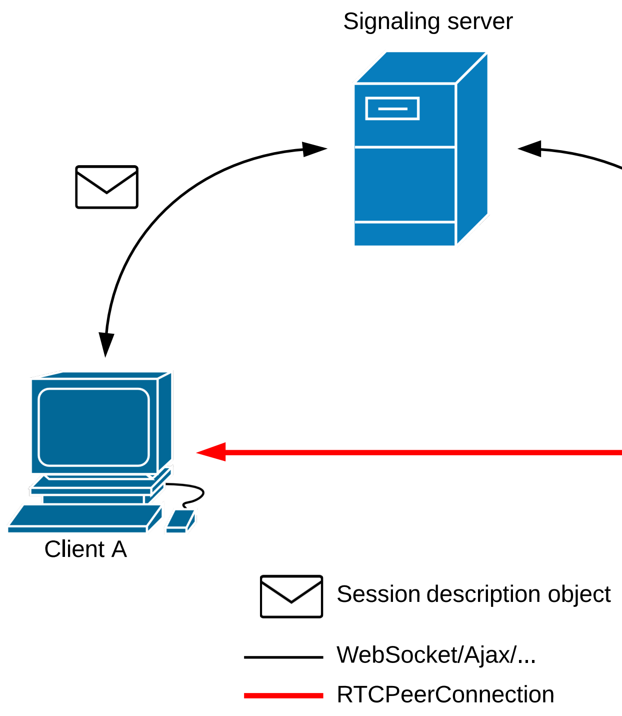


Figure 2.2: Connection setup and signaling in WebRTC. Both clients use a central signaling server to exchange session description objects. The clients use the information inside these objects to setup the actual RTCPeerConnection.

so-called *session description objects* between two clients. These objects contain the information both clients need to setup the RTCPeerConnection: address information, encryption keys, what codec to use, etc. Once these *session description objects* have been exchanged via a central signalling server, both clients can setup the RTCPeerConnection and the server is no longer required. The actual data is exchanged directly between two clients and not routed through a central server. The WebRTC specification does not specify how these *session description objects* should be exchanged; each application is free to use any method it prefers.

This process is visualized in figure 2.2. Both clients connect to the central signalling server. *Client A* initiates the connection by sending a *sessions description object* to the signalling server. The server will forward this object to *client B*, *client B* will then send a *session description object* back to the signalling server. The signalling server forward this object to *client A*. Both clients can now setup the actual RTCPeerConnection.

2.2.2 Routing RTCPeerConnections Through Firewalls and NAT

Due to the limited availability of IPv4 addresses most users are connected to the internet via Network Address Translation (NAT)[11]. By deploying NAT, multiple systems can share one public IP address. The NAT device (usually the router) handles the address translations for all systems transparently. When NAT is used, WebRTC needs a way to get the user's public IP address rather than the local one. WebRTC uses Session Traversal Utilities for NAT (STUN) to get a user's real IP address. Before clients can send their *session description object*, they have to connect to a STUN server and request their public IP address.

When a client is behind a more restrictive NAT or firewall, setting up the RTCPeerConnection could fail altogether. To setup a RTCPeerConnection under these conditions, a Traversal Using

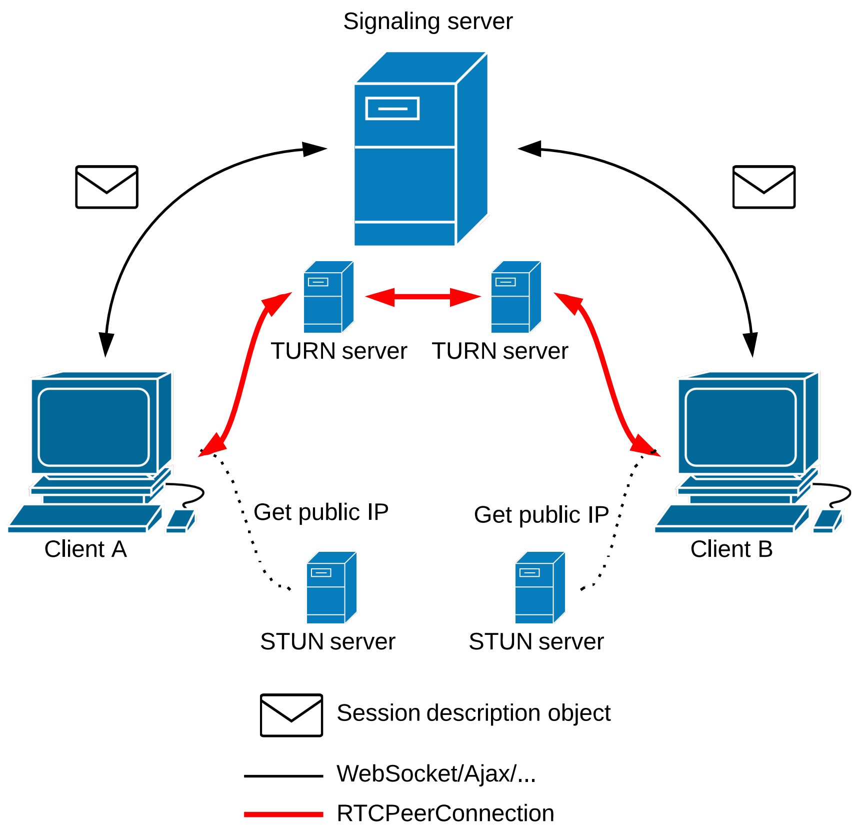


Figure 2.3: Connection setup and signaling in WebRTC behind firewalls and NAT

Relays around NAT (TURN) server needs to be used. When such a server is used, the RTCPeerConnection is no longer a direct connection between two clients but routed through the TURN server instead. Operating a TURN server costs resources so the other methods are preferred. To avoid the use of an expensive TURN server, clients exchange *Interactive Connectivity Establishment* (ICE) candidates[38]. These candidates contain information about what types of connections they can support. Client and server will prefer an ICE candidate that only requires a STUN server over an ICE candidate that requires a STUN and a TURN server. Figure 2.3 displays the connection setup when TURN and STUN servers are used together. Google found that 86% of all attempts could be completed using only a STUN server[20].

2.3 The Canvas Element

The canvas element represents a bitmap that can be embedded into an HTML page. The bitmap can be manipulated through a simple 2D canvas API or through the more advanced 3D WebGL API. The 2D API[75] provides a set of methods to facilitate simple operations like drawing lines, points, images and text. The 3D WebGL API[14] provides a more abstract but more powerful API for advanced graphics operations.

2.3.1 WebGL

The WebGL API is standardized by The Khronos Group, the same author as the OpenGL specification. WebGL is heavily based on OpenGL ES 2.0[67] and has a similar shader-based

API. The WebGL API can be seen as a JavaScript wrapper around OpenGL with some changes to make it work in a web environment. Providing direct access to the OpenGL API would cause major security problems.[70]

WebGL makes it possible to run heavy (3D) graphics applications, such as games, directly in the browser without the need for plugins. Although the WebGL specification is relatively new, it already has very good support across all major browsers[10].

Shaders

Just like OpenGL ES 2.0, WebGL uses a programmable pipeline. This pipeline is programmed with shaders written in the OpenGL Shading Language (GLSL)[68]. There are two types of shaders:

- **Vertex Shaders** take primitive shapes consisting of vertices and perform operations that will determine the position of the vertices. In the case of 3D textures, the vertex shaders has to interpolate values because WebGL does not support 3D textures. When the vertices are processed by the vertex shader, the shapes are rasterized and forwarded to the fragment shader.
- **Fragment Shaders** are responsible for calculating and storing the final color of each fragment in each of the vertices.

The different shaders can share values with each other and with the JavaScript code using three types of variables. *Uniforms* are sent to both the vertex shaders and the fragment shaders and don't change during the rendering of a frame. *Attributes* are only sent to vertex shaders and are used to sent values from JavaScript to the vertex shaders. *Varyings* are variables that can be used to exchange data between the vertex shaders and fragment shaders.

2.4 Conclusion

This chapter introduced new and not yet widely known technologies used extensively throughout the rest of this thesis. WebSockets are used to support bi-directional communication channels between browsers and servers. WebRTC provides API's for bi-directional communication channels between two browsers directly It contains additional objects and abstractions for video and audio streaming. Finally, it introduces the canvas element with the 2D API for simple rendering and the more advanced WebGL API for advanced, hardware accelerated graphics.

Chapter 3

Chrome's Architecture

This chapter explains the key architectural features of Google Chrome. The focus lies on Blink, the rendering engine, and V8, the JavaScript engine. Towards the end, an in-depth analysis of how memory is managed inside Chrome is given. The reasoning behind the decision to use Chrome as the main browser has been given in the introduction, some of those reasons are explained in more detail here. Understanding the basic architecture of Chrome is required to understand the other chapters.

3.1 Overview

Modern web browsers are complex pieces of software with a lot of modules and external dependencies. Each module inside a browser handles one or more of the steps required to display a web page. The main modules in Chrome are Blink (rendering engine), V8 (JavaScript engine), and Chrome itself.

The rendering engine in Chrome, called Blink, is a WebKit fork (minus JavaScriptCore which is replaced by V8). The diagram in figure 3.1 shows the process Blink goes through to turn HTML and CSS into a picture on the screen:

1. Parse the HTML and construct the Document Object Model (DOM) Tree. The DOM tree is a data structure that represents the HTML structure of the web page. Each DOM node in the DOM Tree represents a HTML element (see figure 3.1). This process is often referred to as *DOM construction*. For simplicity, the text nodes are omitted in this example.

2. Parse the CSS

3. Use the DOM Tree and the parsed CSS rules to construct the Render Tree. The nodes in the DOM Tree have a one-to-one mapping to the nodes (RenderObjects) in the Render Tree unless a DOM node should not be part of the actual layout.

4. Blink will use the Render Tree to perform a *layout*. In the layout (or Reflow in some browsers) step, Blink calculates the width and height properties for each RenderObject in the Render Tree recursively. Based on these dimensions, Blink can calculate the position for each RenderObject.

5. Each RenderObject in the Render Tree will now paint itself by writing paint commands to the *Paint Command Buffer*.

6. In the final step, Blink uses the commands in the *Paint Command Buffer* to actually draw the page to a canvas. This step is called *rasterization*.

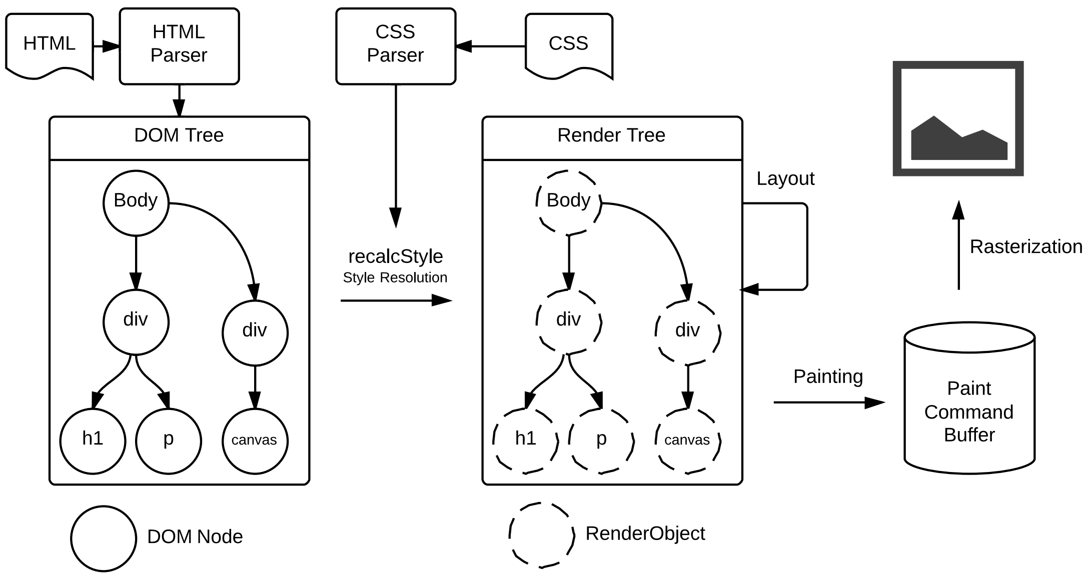


Figure 3.1: The data structures and operations Blink uses to turn HTML and CSS into a picture. The DOM tree contains all nodes in the HTML document while the Render Tree only contains the nodes that are actually part of the page. The Render Tree is ultimately painted and rasterized.

V8, the JavaScript engine in Chrome implements the ECMAScript language specification and compiles and executes JavaScript.[28] It can manipulate the DOM and respond to events (e.g. user input). Through several additional Web JavaScript API's it also provides functionality to perform network request or access device hardware.

The Chrome browser code contains non-web related features and provides the other modules with platform independent API's to perform network requests, use a cache, etc. It also contains a custom, cross platform UI toolkit and a compositor to composite the UI and the bitmaps it receives from Blink.

3.1.1 Module interaction

All the modules in a web browser have to work together to display a web page. The interactions are shown in the diagram in figure 3.2. Starting in the bottom right corner, Blink acts as the embedder for V8 and is the only module that can directly interact with V8. All HTML elements live in Blink's memory but can be manipulated through JavaScript. This means Blink and V8 need to interact with each other and share access to these elements. This is done through the V8 bindings layer. The interfaces for the elements that need to be accessible from V8 are defined in WebIDL[9] files, the actual C++ V8 wrapper objects are then automatically generated based on these WebIDL files.

Moving up in the diagram, Blink is wrapped by the content layer. Blink and the Content layer interact through *the Blink public API*. This API contains a *web API*, used by the embedder (content layer) to communicate with Blink and a *platform API*, used by Blink to request the embedder to do lower-level tasks (e.g.: fetching from cache or network). The content layer is a minimal multi-process sandboxed browser.

The other modules in the diagram are part of Chrome itself (V8 and Blink are separate projects). These modules implement Chrome's specific features like the UI toolkit, bookmarks,

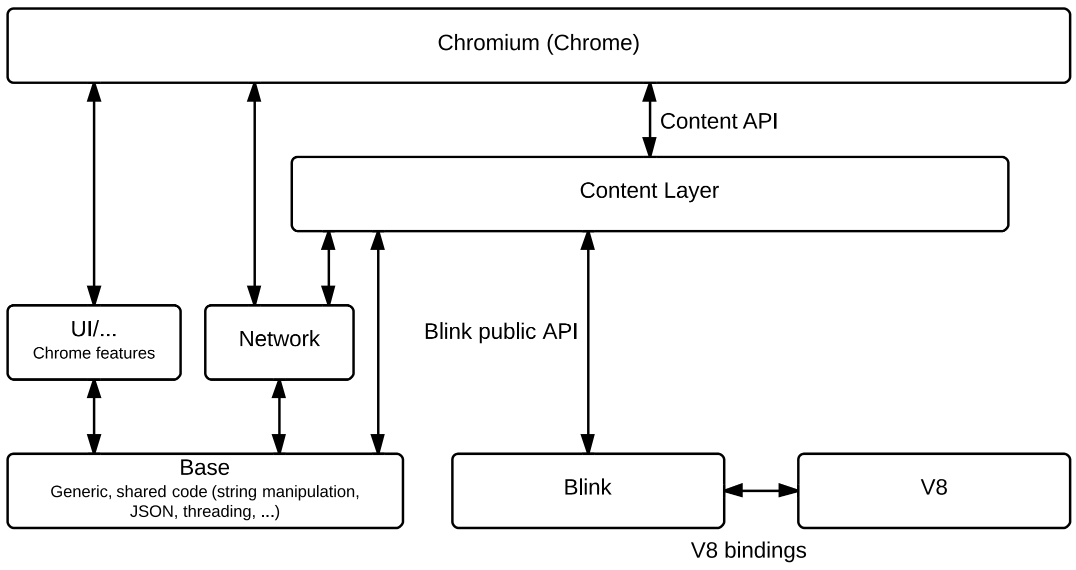


Figure 3.2: Architectural overview of the Chrome web browser showing how the three main components (Blink, V8, and the general browser code) interact with each other. The diagram is based on an outdated architectural diagram[62].

auto-complete, the password manager, etc. The *base* module provides shared, generic code for string manipulations, threading, etc. that is used by the other modules.

3.2 Process Model

Chrome uses a multi-process architecture to improve stability, performance, and security. Figure 3.3 shows the processes Chrome creates. Each tab runs its own sandboxed renderer process or renderer. The sandbox stops the renderers from directly accessing the network, file system, or other renderers. Each renderer process has its own Blink and V8 instances. Each active plugin also gets its own process. This way, a compromised renderer or plugin cannot damage the user's system or interfere with other websites. A hung tab (renderer) or plugin cannot block the rest of the browser either which improves stability. In some cases, Chrome might need to run multiple tabs in one renderer process (e.g. iframes and anchor tags with a *blank* target).

The browser process is the main process in Chrome. It provides network and file access to the other processes and handles the communication between the different renderers. The browser process also contains the compositor that instructs the GPU process on how the draw the UI and web page to the screen. The GPU process is the only process that communicates directly with the graphics card. On Linux and OSX the GPU process uses OpenGL. On Windows, Chrome uses the Almost Native Graphics Layer Engine (ANGLE)[23] to convert OpenGL calls to Direct3D calls because of poor OpenGL driver support. On systems with poor graphics hardware, Chrome will fall back to a software rasterizer called SwiftShader[73].

The number of simultaneously open tabs is not limited in Chrome but the maximum number of renderer processes is. The exact limit depends on the amount of installed memory. On a system with 1GB of ram, the number of renderers is limited to 12 while on a system with 4GB of ram, this limit is 51. This value is calculated so that Chrome does not use more than roughly half of the available memory. There is also a hard limit of 82 simultaneous renderers. Processes for plugins and the GPU are not included in this limit.

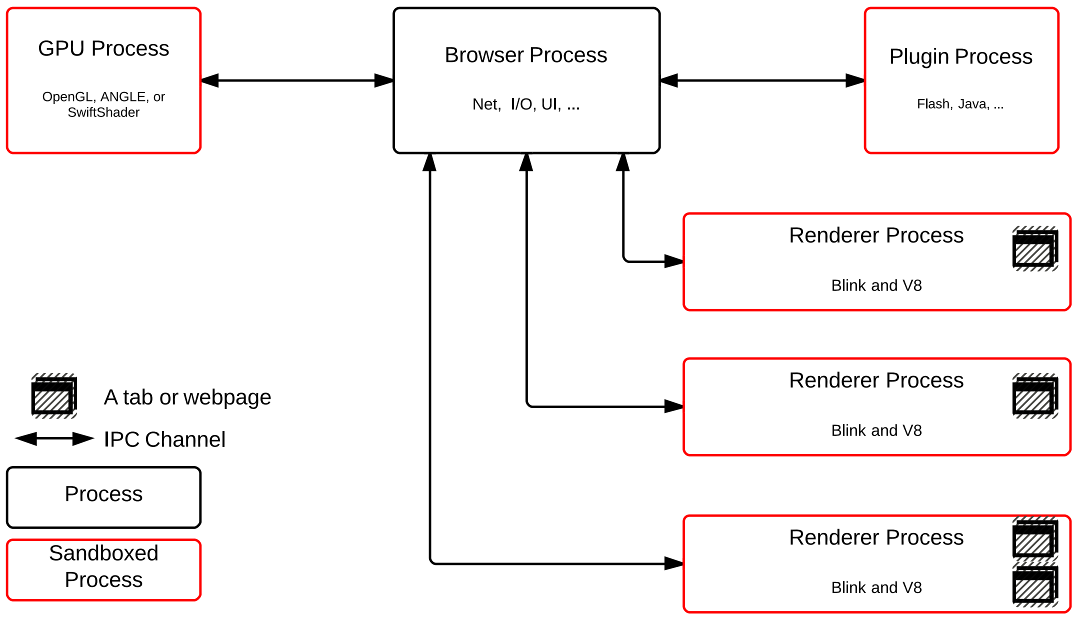


Figure 3.3: Chrome's process model. Each renderer, each plugin, and the GPU code runs in its own sandboxed process. They communicate with the browser process using IPC channels. Sandboxed processes cannot talk to each other directly.

When Chrome hits this limit, one renderer process might need to support multiple tabs. Because each renderer only runs one Blink and one V8 instance, Chrome needs a way to isolate the contents and objects of the different tabs. This isolation is done through the *RenderView* class in Blink and the *Context* class in V8. The different tabs in a renderer will share the Blink and V8 runtime but the actual content (the heap) is isolated by these classes. In this scenario, a compromised renderer could potentially leak information between the tabs running in the same renderer process. To minimize the risk of cross-origin information leaks, Chrome tries to be smart about grouping tabs together and first tries to group tabs from the same origin together.

3.2.1 Inter-process Communication

Because Chrome has a multi-process architecture, it also needs a way for these different processes to communicate with each other. The mechanism to facilitate this is called *Inter-process communication*. When a process needs to communicate with another process (e.g. renderers need to send bitmaps to the browser process, etc.), they do so by sending messages over the IPC channels. Each process is assigned a piece of memory in the browser process which it uses to write data associated with IPC messages. The actual messages are send through *named pipes*.

Communication with the GPU process is a special case. For performance reasons, the data and commands are not send to the browser process but written in a *GPU Command Buffer*. The process is visualized in figure 3.4. The GPU commands are similar to the actual OpenGL ES calls. When a renderer has written all commands to the command buffer, it sends a IPC signal, a *flush*, to the GPU process. The renderer includes a pointer to the commands in the command buffer and the GPU process will read, validate, and execute these commands on the GPU. On Windows, the GPU process uses ANGLE to convert OpenGL calls to Direct3D calls. On all other platforms, the GPU process can use OpenGL directly.

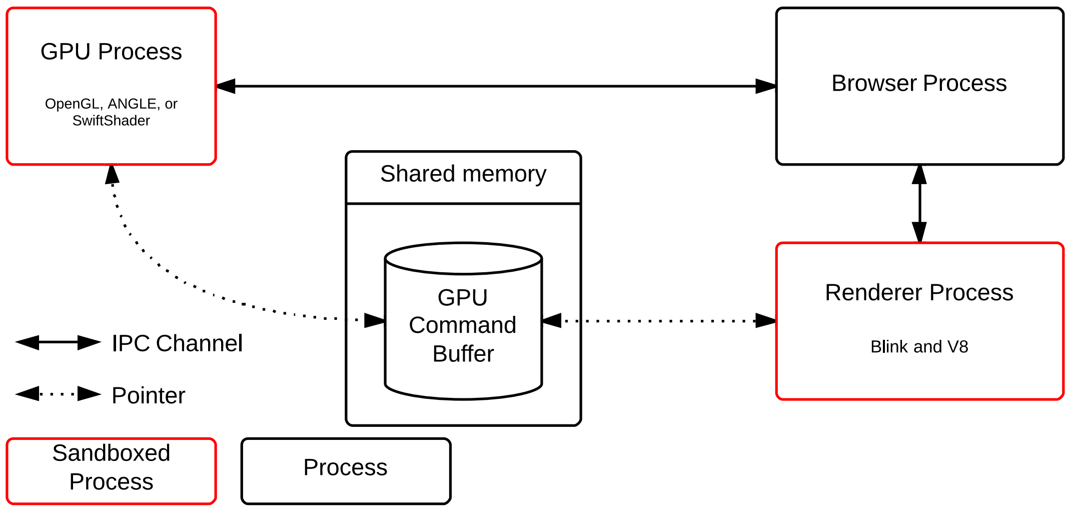


Figure 3.4: Chrome's processes communicate through IPC. Each process has an IPC channel to the browser process to communicate with the other process. Commands for the GPU process are not send over IPC but stored in shared memory (to which each renderer has a pointer).

3.3 Graphics Back-end

Chrome has a very complex graphics back-end for rendering. There are two distinct rendering paths, a software path and a hardware path. The traditional software path uses Skia to paint and rasterize pages while the hardware path uses Ganesh, an OpenGL back-end for Skia. The earlier claim that the *render tree* in Blink is painted and rasterized is not entirely correct. The render tree can be divided into several *render layers*. Each render layer is painted, rasterized and stored in a separate bitmap. The most important creators of layers for our purposes are the video elements and the canvas elements. These elements get their own layers because they can be hardware accelerated. Layer creation is needed to facilitate compositing in the hardware path. This will improve video decoding and WebGL performance.

3.3.1 The Software Path

In software mode, the render layers are traversed and painted from back to front. All render layers are drawn into the same *WebCore::GraphicsContext*. Compositing in the software path is therefore part of the paint process. The *GraphicsContext* in Chrome is a wrapper for Skia and is backed by a *SkBitmap*. This results in a single bitmap (this is abstracted away from the render layers by the *GraphicsContext*) that is stored in shared memory. When painting and rasterization is complete, Blink signals the browser process that it has finished. The browser process will use this bitmap to display the web page on the screen. Certain CSS properties like animations and 3D transforms can also cause layer creation. The *translate3d* layer creation hack is often used to force layer creation to speed up scrolling or animations on a web page. In modern versions of Chrome, the software path is only used on simple pages without accelerated canvasses or video elements.

3.3.2 The Hardware Path

In the hardware path, each GraphicsLayer gets its own GraphicsContext and subsequently its own bitmap. Render layers with only normal HTML elements (not video or canvas) are still

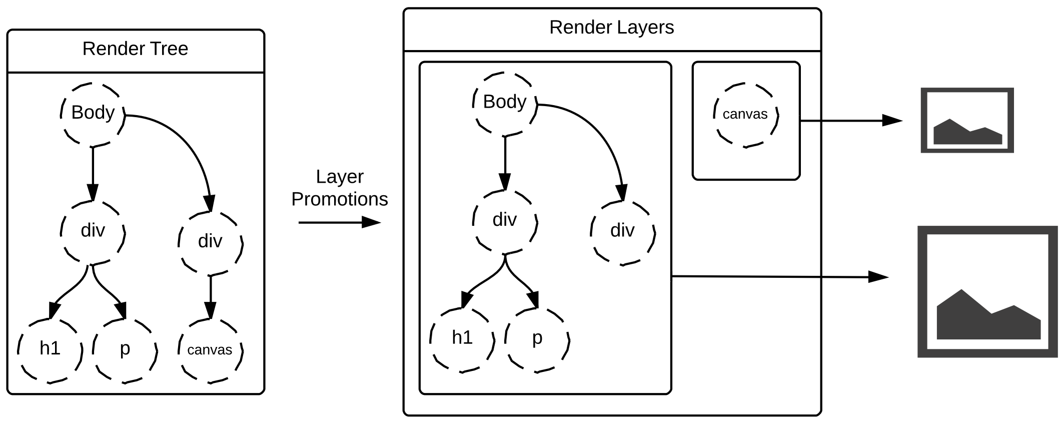


Figure 3.5: The concept of layer promotion. The RenderObjects in the render tree are placed into two separate layers. All objects, except the accelerated canvas element, live in one layer. The canvas element is assigned its own layer. This will aid compositing in the hardware path and avoid a lot of unnecessary GPU to CPU communication. Each layer draws to a separate canvas in the hardware path.

painted and rasterized by Skia on the CPU. These render layers are then divided into 256px by 256px tiles and uploaded as textures to the GPU via IPC.

This tiling is done to reduce video memory usage and bandwidth on the GPU bus. Most web pages are very long but the user only sees a small portion of the entire page. By dividing what normally would be a very large texture into smaller, more manageable tiles, the browser does not need to store one large texture in video memory that is only partly visible. It also saves bandwidth because user interactions can cause invalidations on the page. These invalidated regions are repainted and rasterized by the CPU and need to be uploaded again. Instead of uploading one large texture, Chrome can upload only the tiles that contain invalidated pixels.

Compositing in the Hardware Path

The layer with HTML elements is now painted, rasterized, and uploaded to the GPU as a texture. However, this layer does not contain the accelerated canvas element and has a hole where that canvas should be. The canvas element, like the accelerated video element, is a special case since it is not painted by the CPU but by the GPU. The renderer stores draw calls in the GPU Command Buffer and uses IPC to tell the GPU process to start processing the commands it stored in this buffer. The GPU executes these commands and draws directly to a bitmap on the GPU.

This means the page is now made up of two separate bitmaps instead of one. These two bitmaps need to be drawn together to get the resulting page. This step is called compositing. The renderer attaches position information to each render layer. The GPU reads this information and draws the bitmaps (actual textures in video memory at this point) and applies the appropriate translations to each layer to position it correctly. The render layers are drawn in a sorted order as to preserve z-index ordering of different elements.

Chrome traditionally used multiple compositors to composite the entire Chrome window. Each renderer had its own compositor that was responsible for compositing the web page. The browser process had another compositor that composites all UI elements and the bitmap it received from the compositor in the renderer. Very recent versions of Chrome replaced these double compositors with the *Ubercompositor*[64]. The Ubercompositor runs in the browser process and is responsible for compositing all visible layers, both those part of the page and those part of the browser UI.

3.4 Memory Management in Blink

During the development of a WebGL heavy application, Chrome regularly ran into a *WebGLRenderingContext* limit. The WebGLRenderingContext object exposes an interface to draw on a WebGL canvas and holds the actual drawing buffer. Chrome limits the number of active *WebGLRenderingContexts* to 16. Even though the application hit this limit several times, it never used more than 16 contexts simultaneously and cleared all references to old contexts correctly. There is however a problem with the garbage collection inside the engine.

All WebGLRenderingContext objects live in Blink's memory. Objects in Blink are allocated like any traditional C++ object but instead of working with the raw pointers, Blink provides the *RefPtr* and *PassRefPtr* class templates. These templates are a form of smart pointers. They implement reference counted garbage collection[1]. The smart pointers keep track of how many active references there are to an object, and when the last reference is removed it will free the allocated memory.

This does not happen with the WebGLRenderingContexts because they are accessed from JavaScript and V8 holds an active reference to each context through the V8 bindings layer. V8 creates a wrapper object for the context This wrapper object holds a reference to the corresponding Blink object but lives in V8's memory. V8 does not use reference counted garbage collection. To investigate further and find potential solutions or workarounds, we need to know how V8 manages memory, and specifically, how it cleans up old objects.

3.5 Memory Management in V8

V8 divides the heap into several spaces[1]. First of all there are *new space* (fixed size) and *old space* (can grow). New objects are initially placed in *new space*, except large objects (larger than *Page::*kMaxHeapObjectSize) which skip *new space* and are immediately placed in *large object space.* Objects in *new space* that have been alive for a while are moved to *old space.* V8's full heap layout is shown in figure 3.6.

3.5.1 New space

When V8 allocates a new object, it places this object in *new space.* Objects are promoted from *new space* to *old space* after having survived 2 minor garbage collections in *new space* without being collected.

Allocations in *new spac*e are fast. It contains a pointer to free space that is moved each time new objects are allocated. This space is cleaned up when a minor garbage collection is executed. During a minor garbage collection inactive objects are cleaned up and some objects are promoted to *old space.* A minor garbage collection of *new space* is very fast (∼2ms [48]). and generally happens more often than a major garbage collection. New space consists of two SemiSpaces called *to_ space* and *from_ space.* Their function is explained later.

[1]`https://code.google.com/p/chromium/codesearch#chromium/src/v8/src/v8globals.h&sq=package:chromium&type=cs&l=179`

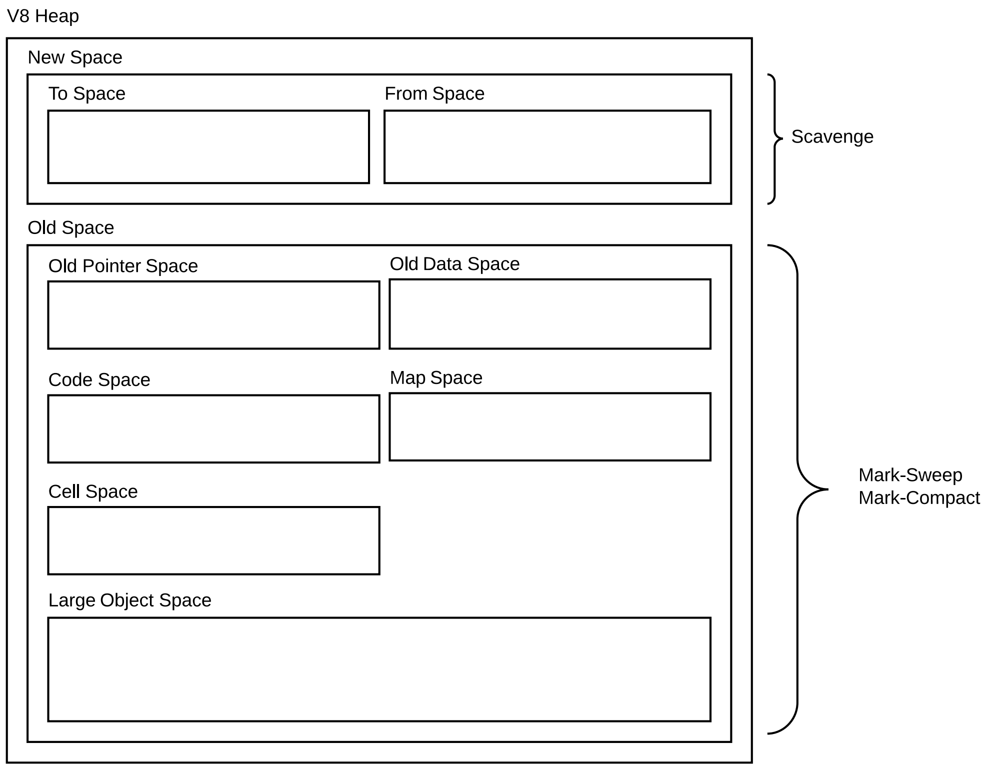


Figure 3.6: V8's heap layout. The heap is divided into several smaller spaces. Each space contains a specific type of data. This layout helps the garbage collectors shown on the right.

3.5.2 Old space

Old space is divided up further in several spaces. Each space holds a specific type of data;

- **Old pointer space** contains objects that can reference other objects;
- **Old data space** contains pure data objects (primitive Strings, Numbers, ...) that cannot reference other objects;
- **Code space** contains objects with executable code in them. Code space and large object space are the only spaces that can contain executable code;
- **Map space** contains Maps. The concept of Maps is internal to the engine and not of relevance here;
- **Cell Space** contains PropertyCell objects. These objects represent the properties of the global object;
- **Large object space** contains large objects, including large code objects.

3.6 Garbage Collection

V8 has two types of garbage collectors[2]; a minor garbage collector called *Scavenge* and a major garbage collector called *mark-compact*. The minor garbage collector cleans up *new space* while

[2]`https://code.google.com/p/chromium/codesearch#chromium/src/v8/src/v8globals.h&type=cs&sq=package:chromium&l=209`

the major garbage collector cleans up *old space* and *new space*. Both garbage collectors use a *stop-the-world* tactic and halt program execution. This can cause noticeable pauses during a major garbage collection round. To overcome this problem, the major garbage collector works incrementally and only processes a small part of the heap during each pause.

3.6.1 Minor Garbage Collector

A minor garbage collection round, called *Scavenge*, is used to clean up *new space*. The minor garbage collector in V8 is an implementation of Cheney's copying algorithm[8]. The algorithm divides up *new space* into two *SemiSpaces*; *to_ space* and *from_ space*. The algorithm first swaps the pointers to *to_ space* and *from_ space*. Next, the algorithm goes through all objects and checks whether they are still active, if so, they will be copied from *from_ space* to *to_ space* while objects without active references are cleaned up. Objects that have survived two minor garbage collections are added to a queue to be promoted to *old space*.

What triggers a minor garbage collection?

The minor garbage collector appears to be invoked when needed i.e, when JavaScript needs to allocate memory but there is not enough space left in *new space*. Since the size of *new space* is limited, garbage collecting it will be fast.

3.6.2 Major Garbage Collector

A major garbage collection round, called *mark_ sweep* or *mark_ compact*, cleans up *old space*. This major garbage collector consists of two phases; a marking phase and a sweeping phase. An optional third compacting phase can be used to rearrange active objects to minimize the empty space between objects. During each pause it will either;

- mark objects in a part of the heap as active or inactive or;
- sweep the heap by cleaning up objects marked as inactive or (pauses for up to 50ms when not incremental [48]);
- optionally compact the heap by moving objects together (pauses for up to 150ms when not incremental [48]).

What triggers a major garbage collection?

A major garbage collection is triggered by a call to *v8::IdleNotification(hint)*. The renderer calls this function when it is idle and uses the first parameter to signal to V8 how much work can be done during this pause. V8 will set the return value to *true* to indicate to the renderer that it cannot collect more garbage. The renderer should not call *v8::IdleNotification* again until *real work has been done*. *v8::IdleNotification* is called;

- periodically from the render thread;
- when the V8 context is disposed (i.e. when a tab is closed or when the browser navigates away from the current page);
- by WebWorkers.

The renderer can also issue a *low memory notification*. A *non-critical* low memory notification will call *v8::IdleNotification* twice, a *critical* low memory notification immediately triggers a full major garbage collection, and additionally, clears the Skia[27][3] font cache. This only happens

[3]Skia is the 2D graphics back-end used in Chrome

when Chrome is close to running out of memory and tries to avoid a fatal out-of-memory state which would result in a renderer crash.

For most web pages, the periodical calls to *v8::IdleNotification* from the renderer are the most important. These periodical calls are handled by a timer; the interval between two calls is determined by a damped function:

$$delay\_ms = delay\_ms + \frac{1000 * 1000}{delay\_ms + 2000}$$

Starting at 1000ms, the delay will grow each time this function is applied;

$$1000, 1333, 1633, 1908, 2163, 2403, 2630, 2845, 3051, ...$$

In the foreground tab, this delay is limited to 30 seconds and reset to 1 second once this limit is reached. The call to *v8::IdleNotification* is never made when the renderer is using more than 3% CPU. These limits do not exist for tabs running in the background.

The first call to *v8::IdleNotification* is made roughly 30 seconds after the renderer was created. After the first call the delay will exceed the 30 second threshold and the interval is reset to 1 second. The interval between subsequent calls will be determined by the damped function described above (until it reaches 30 seconds and is reset again).

The return value of *v8::IdleNotification* also influences the delay. When *v8::IdleNotification* returns *true*, V8 is signaling that it is done collecting garbage and there should be no calls to *v8::IdleNotification* until *real work has been done*. The renderer will set the interval to 30 seconds at this point. As can be seen, the invocation of the garbage collection is highly unpredictable.

3.6.3 Collection of old WebGLRenderingContext objects

A WebGLRenderingContext that is no longer used and has no active references to it should be destroyed by the garbage collector. Since Blink uses reference counted garbage collection, it cleans up the object immediately after the last reference is removed. When the associated canvas is removed from the DOM, Blink has to wait on the garbage collector in V8 to clean up the wrapper object first before it can clean up the context. V8 does clean up the wrapper object correctly but it can take a while before it actually does so.

Therefor, the problem of hitting the active WebGLRenderingContexts threshold is caused by a missing link between the limit imposed on the active contexts and the garbage collector. The garbage collector runs independently from this limit; it is triggered by other factors. This means that a lot of *WebGLRenderingContext*s could be ready to be garbage collected but the garbage collector feels no need to do that now. This is only a problem for WebGLRenderingContext because it is the only object that has such an artificial limit. This is a known issue in Chrome but unfortunately appears to be low priority[4].

The memory integration between Blink and V8 is a bit of a mess and has caused security problems and memory leaks in the past. It is difficult to link V8's tracing garbage collector with Blink's reference counted garbage collector. Because of these problems, Blink is moving towards a V8 like garbage collected heap for objects as part of project Oilpan[24]. Instead of allocating each object separately, objects are stored in the Oilpan Heap, which lives in Blink's memory. Each object inherits from *GarbageCollected<T>* or *GarbageCollectedFinalized<T>* which provide the tracing logic for the garbage collector. This will make the integration between V8 and Blink cleaner and more stable but it is likely to make our problem worse.

[4] `https://code.google.com/p/chromium/issues/detail?id=290482#c9`. Last checked on May 25, 2014.

3.6.4 Possible solutions

Now that the cause of the problem has been identified and the fact that is probably going to get worse we need to find a solution. We found three solutions that work around the problem. Some of these solutions have major disadvantages unfortunately.

Starting a Garbage Collection from JavaScript

JavaScript is a garbage collected language. A developer should not have to worry about memory management. It is the job of the JavaScript virtual machine. This is why there is no interface to clear memory from JavaScript. In Chrome, a *gc* object can be exposed to JavaScript by starting Chrome with the *–js-flags="–expose-gc"* command line flag.

A major garbage collection can be started by calling *gc.call().* This can be used to clear up old WebGLRenderingContexts explicitly on request. There are disadvantages to using this method though; *gc.call* performs a full garbage collection, not one incremental step. This causes long garbage collection pauses and can slow down the application. The method is not very feasible in practice either. Nobody starts their browser with a command line flag.

Modify the source code to invoke the garbage collector when the active WebGLRenderingContext threshold is reached

A more preferable method is coupling the active WebGLRenderingContext threshold with the garbage collector. When the modified browser code detects the threshold is being reached, invoke or hint the garbage collector to run from C++. This method has the same problems as the previous method. One incremental garbage collection step is not guaranteed to clear up the old WebGLRenderingContexts. A full garbage collection round is required resulting in the same long pause times. A small difference is that the browser knows how many active contexts there are and can control the garbage collector more efficiently. This method means all users have to use a modified Chrome build which will be difficult to achieve as well.

Recycling active WebGLRenderingContexts

Instead of clearing references to old contexts, the context can be stored and recycled later. This can be done by implementing a WebGLRenderingContext factory that handles all WebGLRenderingContext creations. Code that needs a WebGLRenderingContext, can request one from the factory and old WebGLRenderingContext should be returned to the WebGLRenderingContext factory. The factory will reset the state and store the instance in a buffer. The next time a WebGLRenderingContext is requested, it can return a existing instance from the buffer.

Correctly implemented, this should keep Chrome from hitting the 16 active WebGLRenderingContexts limit (under the assumption the code does not need more than 16 contexts simultaneously). Resetting the contexts is important because they could otherwise leak other WebGL objects like textures and buffers.

3.7 Conclusion

The focus of this chapter was on the architecture of Chrome, the interaction between the main components, and the memory architecture. Chrome uses Blink for rendering HTML and CSS; JavaScript is handled by the V8 JavaScript engine. For improved performance, stability, and security, Chrome employs a multi-process architecture. Communication between the renderers, the GPU process, and the plugin processes is handled by the main browser process via IPC.

There are two distinct code paths for graphics rendering; the software path and the hardware path. The software path uses software rasterization for all operations. The hardware path uses the GPU to accelerated the rendering of certain elements. Chrome supports hardware accelerated video decoding, accelerated canvas rendering, and accelerated 3D CSS animations. This hardware acceleration is supported through the concept of render layers in Blink: accelerated elements are drawn in their own separate layers and composited by the graphics card. This greatly improves rendering performance.

Memory in Chrome either belongs to Blink (HTML elements) or V8 (JavaScript objects). Blink and V8 share memory objects through the V8 bindings layer. Memory in V8 is divided into two spaces: new space and old space. Garbage collection in V8 is handled by two garbage collectors; the minor garbage collector that cleans up new space and the major garbage collector cleans up to entire heap. Both calculate the retaining path to the elements in the heap, elements without a retaining path are cleaned up. Blink's garbage collector uses reference counting to clean up old objects. These distinct types of garbage collectors cause problems in the bindings layer between Blink and V8 which is one of the reasons why Blink is moving to a V8 like tracing garbage collector.

The garbage collectors in V8 are only ran every so often This caused problems for WebGL heavy pages because the maximum number of active WebGL rendering contexts is independently limited to 16. The proposed solutions included exposing the *gc* object to JavaScript, modifying the source code, and the actual implemented WebGL rendering context factory solution.

The results of the research in this chapter will prove very useful when reading the remaining chapters. A good understanding of browser internals also aids developers in understanding the performance characteristics of their applications. The following chapter builds upon this chapter but the focus shifts to the available developer tools. The architectural aspects discussed in this chapter become apparent through these developer tools.

Chapter 4

Chrome Performance

Performance is a key objective for all solutions created in the context of this thesis. To evaluate the performance of any code, good performance tools are important. The built-in Developer Tools in Chrome contain some advanced tools to monitor and evaluate the performance of important aspects like network requests, memory usage, and code run time from the JavaScript world. Chrome also contains a full-featured JavaScript debugger.

With the built-in *tracing tool* (`about://tracing`), it is possible to dive even deeper and see how JavaScript calls are handled inside Chrome's core, where the code is spending most of its time, and how the memory is used.

4.1 Performance Tooling

A short introduction to some of the performance tools that will be used later on seems appropriate. The *Timeline* view gives a breakdown of what is happening inside the renderer, the *Heap Inspector* breaks down memory usage and is a useful tool for detecting memory leaks, and the *benchmarking JavaScript API's* exposes high-resolution timers to JavaScript.

4.1.1 Timeline

The timeline view works with recordings: a developer can start a recording, perform the actions he is interested in, and stop the recording. The timeline view displays all actions that occurred during the recording These actions can be broken down further. Specifying an interval in the timeline will give a further breakdown of what the renderer is doing.

These breakdowns contain the most important actions in a web browser:

- **Loading**: Fetching resources from the network or cache.
- **Scripting**: Execution of JavaScript on the page.
- **Rendering**: Rendering of the page, often referred to as *Layouts* or *Reflow*s.
- **Painting**: Painting of (part of) the page (includes the decoding of images).

It is possible to add custom actions to a recordings by calling *console.timeStamp('My Action')* from any location in the code (see section 4.1.4*).*

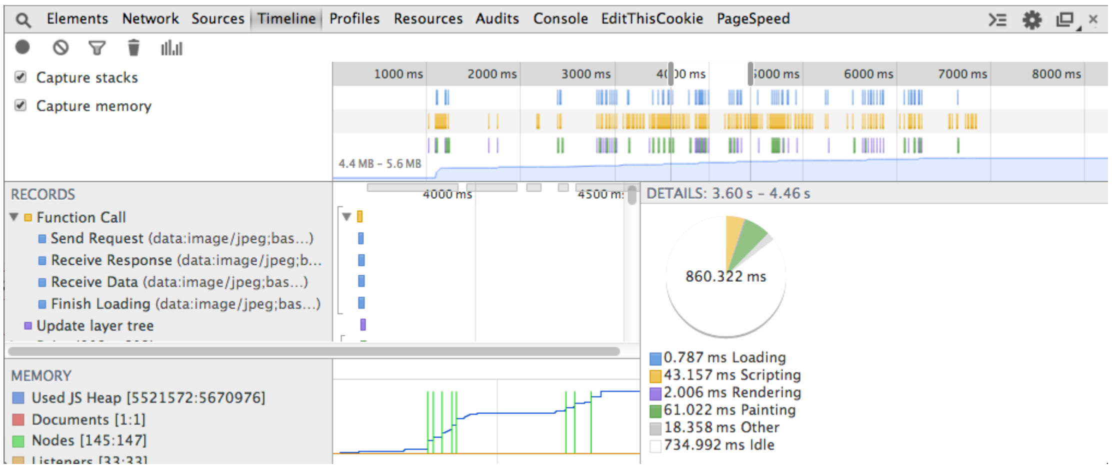


Figure 4.1: Screenshot of the Timeline view in Google Chrome showing a breakdown of the run-time of the different actions for the interval from 3.6s to 4.46s.

4.1.2 Profiling

While it is a very useful tool for getting a good understanding of what is going on JavaScript code, the timeline view does not show where the code is spending most of its time or how the memory is used. The Developer Tools contain extra tools to profile JavaScript execution and memory usage.

JavaScript CPU Profile

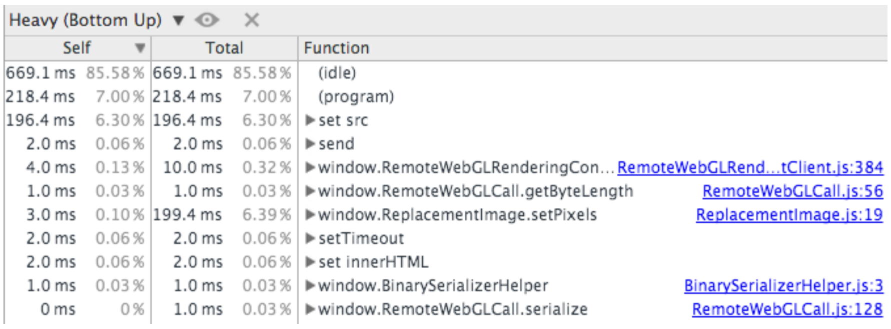
Like the Timeline view, the JavaScript CPU profiler works with recordings. A developer can record only the actions he is interested in and analyze those. The JavaScript CPU profiler shows a bottom up list of JavaScript calls sorted by run-time as shown in figure 4.2 (a). Alternatively, figure 4.2 (b) shows the same recording in a Tree (Top Down) view. The first view is used to get an immediate view of which functions are taking up most of the time.

For example: from the view in figure 4.2 (a), it can be concluded that the *set src* action takes up most of the running time. It is possible to climb up the tree to find out from where this function is called. The view in 4.2 (b) shows the same thing but in reverse; the *WebsocketSocket.message* call is responsible for most of the running time. It is possible to decent into this function call and see where it spends most time. If we were to do this here, we would end up at the *set src* action because it is called by *WebsocketSocket.message* eventually.

Heap Snapshots

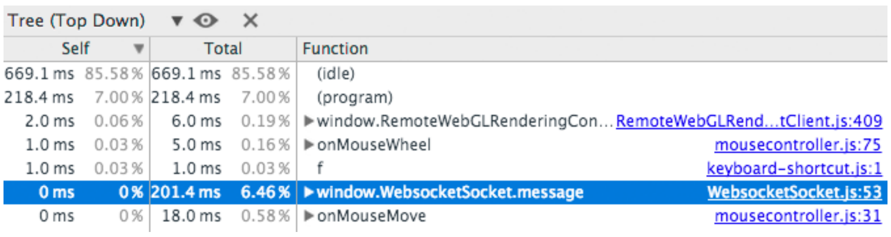
The profiling section of the Developer Tools also contains a feature to record heap snapshots. A recorded heap snapshot first opens in the *Summary* view. Here all objects are grouped together based on the constructor that was used to create them. For each active object in the heap, the following additional information is shown:

- **Distance**: The length of the shortest path to a garbage collector root. Figure 4.3 shows an example heap with several objects. To calculate the distance for node D, first determine what the retaining path to node D is. The retaining path for an object is the shortest path from any GC root to the object. Objects with a retaining path are active, objects without a retaining path have no active references and will be collected by the garbage collector.


Heavy (Bottom Up)

Self		Total		Function	
669.1 ms	85.58 %	669.1 ms	85.58 %	(idle)	
218.4 ms	7.00 %	218.4 ms	7.00 %	(program)	
196.4 ms	6.30 %	196.4 ms	6.30 %	▶set src	
2.0 ms	0.06 %	2.0 ms	0.06 %	▶send	
4.0 ms	0.13 %	10.0 ms	0.32 %	▶window.RemoteWebGLRenderingCon…	RemoteWebGLRend…tClient.js:384
1.0 ms	0.03 %	1.0 ms	0.03 %	▶window.RemoteWebGLCall.getByteLength	RemoteWebGLCall.js:56
3.0 ms	0.10 %	199.4 ms	6.39 %	▶window.ReplacementImage.setPixels	ReplacementImage.js:19
2.0 ms	0.06 %	2.0 ms	0.06 %	▶setTimeout	
2.0 ms	0.06 %	2.0 ms	0.06 %	▶set innerHTML	
1.0 ms	0.03 %	1.0 ms	0.03 %	▶window.BinarySerializerHelper	BinarySerializerHelper.js:3
0 ms	0 %	1.0 ms	0.03 %	▶window.RemoteWebGLCall.serialize	RemoteWebGLCall.js:128

(a) Bottom Up View


Tree (Top Down)

Self		Total		Function	
669.1 ms	85.58 %	669.1 ms	85.58 %	(idle)	
218.4 ms	7.00 %	218.4 ms	7.00 %	(program)	
2.0 ms	0.06 %	6.0 ms	0.19 %	▶window.RemoteWebGLRenderingCon…	RemoteWebGLRend…tClient.js:409
1.0 ms	0.03 %	5.0 ms	0.16 %	▶onMouseWheel	mousecontroller.js:75
1.0 ms	0.03 %	1.0 ms	0.03 %	f	keyboard-shortcut.js:1
0 ms	0 %	201.4 ms	6.46 %	▶window.WebsocketSocket.message	WebsocketSocket.js:53
0 ms	0 %	18.0 ms	0.58 %	▶onMouseMove	mousecontroller.js:31

(b) Tree View

Figure 4.2: Screenshots of the CPU Profiler in Chrome's Developer Tools.

For node D, the retaining path is R →A → B → D. This makes that the distance for node D is 3.

- **Objects Count**: How many objects exist that were created using this constructor.
- **Shallow Size**: The size of the object itself and only the object itself. With the exception of Arrays and Strings, the shallow size for objects is generally small.
- **Retaining Size**: The amount of memory that would be freed when this objects is garbage collected. This includes the shallow size (and retaining size) of all objects that can only be reached from this object. The retaining size for object A would be to sum of the shallow sizes of objects A, B, C, and D.

The *Comparison* view compares two heap snapshots and is a convenient tool for detecting memory leaks. By taking a snapshot before doing an action that should clean-up itself (e.g.:

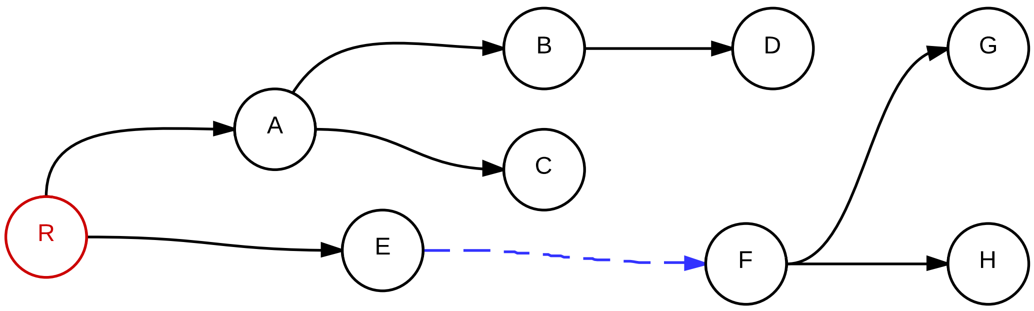
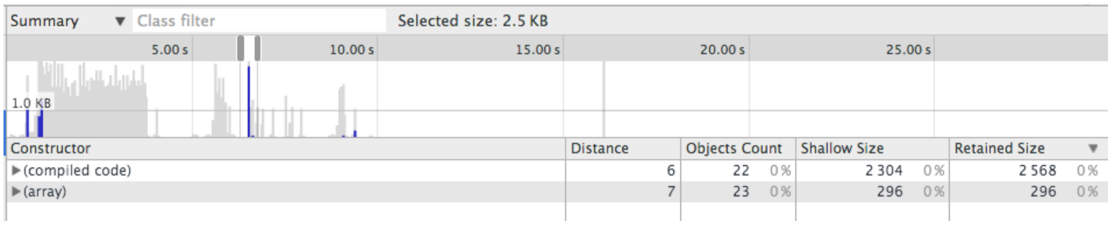


Figure 4.3: Example of a heap structure with retaining paths. The red node (R) is a GC root. The blue edge between nodes E and F is forgotten reference causing nodes F, G, and H to leak.



Selected size: 2.5 KB

Constructor	Distance	Objects Count		Shallow Size		Retained Size ▼	
▶(compiled code)	6	22	0%	2 304	0%	2 568	0%
▶(array)	7	23	0%	296	0%	296	0%

Figure 4.4: Screenshot of the Heap Allocations tracker. A gray color means the objects have been cleared while blue means the objects are still alive. Bars can be partially Grey and partially blue.

creating a canvas, drawing on it, and removing it again) and comparing that snapshot with a snapshot taken after the action should show no new objects. A different number of active objects could indicate a memory leak.

Heap Allocations

The last tool found in the profiling section of the Developer Tools, records the heap allocations and is the best tool to detect memory leaks. All allocations are recorded and displayed on a chart over time. The height of each bar indicates the total size of all objects allocated at that point in time. The color of each bar indicates whether those objects have been cleaned up since being allocated.

Figure 4.4 shows a heap allocation recording. It shows that most allocated objects already been cleaned up. The blue bars indicate that some objects remain active, by selecting any of the blue bars, Chrome displays what those objects are. Clicking on any of these objects will display its retaining path. Blue bars can indicate a memory leak but don't necessarily imply a leak. It is important to check what the active objects are and whether they are expected to be still alive.

Leaking Memory

The heap structure in figure 4.3 is an example of a memory leak. Nodes F, G, and H were created after the user performed an action (e.g. creating a canvas and drawing on it). The objects have a retaining path to the GC root through node E. When the user performs an action that removes the canvas, the retaining path should be broken (the edge between node E and node F). Because of a logical error in the code however, this reference is never destroyed and the objects keep their retaining path and are never destroyed. This is a memory leak and can be fixed by destroying the reference between node E to node F.

4.1.3 Trace Event Profiling Tool

Previously discussed tools, while very useful to analyze JavaScript, do not allow developers to see what happens inside the browser itself. During development we encountered several cases where we needed to see what was happening inside Chrome's core. Chrome ships with a less user-friendly but more powerful version of the previously discussed *Timeline* view called the *Trace Event Profiling Tool.* This records events that have been compiled into the C++ code. Internally, the *timeline* view uses the same traces but shows them in a different way.

Figure 4.5 shows a small section of a recording, specifically, it shows how Chrome handles a *MouseMoveEvent* by calling the attached event handler (v8.callfunction). This is a very simple

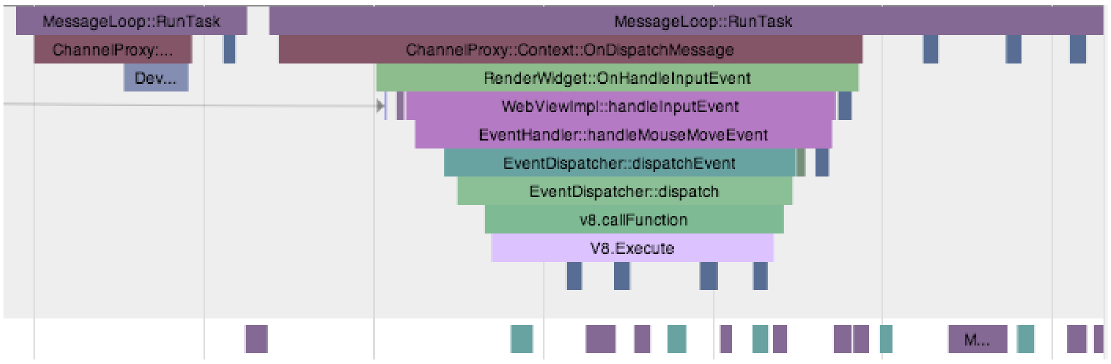

MessageLoop::RunTask
ChannelProxy:...
Dev...
MessageLoop::RunTask
ChannelProxy::Context::OnDispatchMessage
RenderWidget::OnHandleInputEvent
WebViewImpl::handleInputEvent
EventHandler::handleMouseMoveEvent
EventDispatcher::dispatchEvent
EventDispatcher::dispatch
v8.callFunction
V8.Execute
M...

Figure 4.5: Screenshot of the Trace Event Profiling Tool

example but this tool is used throughout the rest of this document so understanding it is important. Horizontally, each bar corresponds to an event inside Chrome. An event usually maps directly to a C++ function call inside Chrome's core, the name of this function is shown as text on the bar. The length of each bar corresponds to the total execution time (time on the call-stack). Vertically, the bars can be seen as forming a (partial) calling stack in function of time. The first bar, *MessageLooop::RunTask* calls *ChannelProxy::Context::OnDispatchMessage,* which in turn calls *RenderWidget::OnHandleInputEvent* and so on. *ChannelProxy::Context::OnDispatchMessage* does not call *RenderWidget::OnHandleInputEvent* immediately, it first performs another action which is not instrumented (or traced). Empty spaces do not imply the browser is sitting around doing nothing; not the entire source-code is traced.

Adding Tracing Events

When a developer is interested in a particular piece of the code that is not instrumented in enough detail, it is possible to add custom instrumentation to that part of the codebase[59]. Trace events can be added by calling any of the provided tracing macro's. The example in algorithm 4.1 will record trace events called *someInterestingAction* and *doPartialInterestingAction* but not *doNextPartialInterestingAction.* Adding the tracing macro is the only requirement, it will show up in the traces automatically after that.

4.1.4 JavaScript Benchmarking API's

Traditionally, JavaScript only exposed time related data through the built-in *Date* object. With this object, it is possible to get time information accurate to one millisecond. While this is accurate enough for most applications, some benchmarks require a high-resolution timer, especially those involving fast, high-performance code. The W3C recently introduced the User Timing[41] and High Resolution Time[40] specifications. Together, these specifications define a *window.performance.now* method that returns a *DOMHighResTimeStamp* instance. This object represent the number of milliseconds, accurate to one microsecond since navigation started. Algorithm 4.2 demonstrates the usage of this API.

Chrome also ships with a *chrome.Interval* object that implements a high-resolution timer with an accuracy of one microsecond. This object is only exposed to JavaScript when Chrome is started in benchmarking mode. Benchmarking mode can be enabled by starting Chrome with the *–enable-benchmarking* command line flag. Algorithm 4.3 demonstrates the usage of this API.

Algorithm 4.1 A Chrome tracing code snippet. Traces for the doPartialInterestingAction and someInterestingAction methods will be recorded.

```
#include <base/debug/trace_event.h>
void doPartialInterestingAction() {
  TRACE_EVENT0("MY_COOL_COMPONENT", "doPartialInterestingAction")
     ;
  // ...
}

void doNextPartialInterestingAction() {
  // ...
}

void someInterestingAction() {
  TRACE_EVENT0("MY_COOL_COMPONENT", "someInterestingAction");
  doPartialInterestingAction();
  doNextPartialInterestingAction();
}
```

Algorithm 4.2 Recording high-resolution timing information using the window.performance object.

```
for (var i = 0; i < n; i++) {
     window.performance.mark('start_operation');
     operation(i);
     window.performance.mark('end_operation');

     window.performance.measure('operation_N_' + i + '_measure', '
        start_operation', 'end_operation');
}
var measurements = window.performance.getEntriesByType('measure')
    ;
for (var i = 0; i < measurements.length(); i++) {
     var measurement = measurements[i];
     // measurement.duration is a double
     console.log(measurement.name + ' took ' + measurement.
        duration + 'ms');
}
```

Algorithm 4.3 Recording high-resolution timing information using the chrome.Interval object.

```
var timer = new chrome.Interval();

timer.start();
operation();
timer.stop();

console.log('Took ' + timer.microseconds() + 'microseconds');
```

Algorithm 4.4 Recording timing information using the console.time* functions.

```
console.time('SomeAction');
operation();
console.timeEnd('SomeAction');
```

Recording Timing Information Through the Console

When the high-resolution time is not required, it is easier to use the *console* object to get timing information. The *time('action')* and *timeEnd('action')* methods record the time, accurate to a millisecond, between both calls and print this in the console. The event also shows up in the *Timeline* view of the Developer Tools and in the traces recorded with the *Trace Event Profiling Tool.* Algorithm 4.4 demonstrates the usage of this API.

4.2 V8 Optimizations

Modern web browsers optimize the different aspects that come into play when rendering web pages and executing JavaScript. The focus here is on optimizations in the V8 JavaScript engine because for our use-cases, JavaScript heavy WebGL applications with simple layouts, JavaScript and canvas performance are the most important aspects. It is important to know how these optimizations work, JavaScript code that works with the engine rather than fight against it will perform much better.

4.2.1 Hidden Classes

One of the most important architectural concepts in the V8 engine are the *hidden classes.* JavaScript is a dynamically typed language which means types and type information can change and properties can be added and removed from objects at run-time. Dynamically typed languages traditionally used hash-maps to store and retrieve dynamic properties. This makes property access in dynamically typed languages slower than property access in typed languages where the offset of each property is determined by the compiler. V8 solves this problem by introducing type information through hidden classes. These hidden classes (or maps) are internal to the engine and not exposed through JavaScript. Hidden classes are stored in *MapSpace* inside the V8 heap.

Hidden Class Creation

Algorithm 4.5 defines a *MySocket* function that is used to create an object twice. Running this code in V8 will result in the creation of 3 hidden classes for the *MySocket* function. Figure 4.6 shows the different steps through which each object goes before it is fully created. When line 6 is executed, V8 will allocate a new object called *dataSocket* and allocate small initial backing stores for the *properties* and *elements* properties. At this point only the initial hidden class for the *MySocket* function (class 0) exists and the new objects *Hidden Class* property will point to this initial *class 0* (step 1.1).

V8 will now start creating the new object by jumping to line 2. This line adds a new property to our *MySocket* object, this will result in the creation of a new hidden class: class 1. V8 tracks the transition from class 0 to class 1 as adding a *host* property and updates the hidden class pointer in our *dataSocket* object (step 1.2). This new hidden class is created by copying class 0 and adding a new property called *host* and storing the offset. The actual value for this property

Algorithm 4.5 Example of Hidden Class creation for two objects that will share the same hidden class

```
1 window.MySocket = function(host, port) {
2     this.host = host;
3     this.port = port;
4 }
5
6 var dataSocket = new MySocket('martijnc.be', 1234);
7 var controlSocket = new MySocket('martijnc.be', 1235);
```

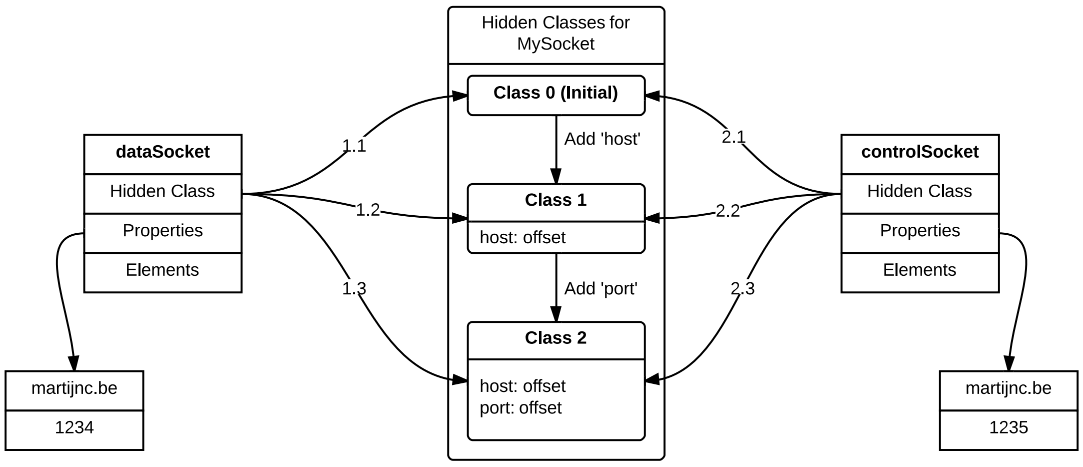


Figure 4.6: Visualization of the hidden class creation in Algorithm 4.5

is stored in the *properties or elements* backing store of the new object at the offset stored in the hidden class.

On line 3, the *po*rt property is added to the new object resulting in the creation of a new hidden class (class 2) and a transition: from class 1 to class 2. V8 tracks this hidden class transition as adding a *port* property to hidden class 1. The hidden class pointer in our *dataSocket* object is updated (step 1.2) and the value is stored in the appropriate backing store. The *dataSocket* object is fully created at this point.

V8 will now execute line 7 where a new object, called *controlSocket*, is created with the same *MySocket* function. V8 will allocate room for this new object and the two initial backing stores. The initial class for this object at this point is class 0 (step 2.1). To create this object fully, V8 jumps to line 2 again where a new *host* property is added. Previously, V8 created a new hidden class at this points and tracked it with adding a new *host* property. This time round, V8 already knows of a transition from class 0 that adds a *host* property: the transition to class 1. Instead of creating a new hidden class, the new object will just point to class 1 as its new hidden class and the actual value is stored in the appropriate backing store at the offset found in the hidden class (step 2.2). The use of hidden classes makes the creation of the first object slow but very fast after that.

On line 3, the *po*rt property is added to the new object and V8 goes through the same process: instead of creating a new hidden class, it reuses the existing transition from class 1 to class 2 and updates the hidden class of the new object to point to class 2. The actual value is stored in the backing store at the offset in the hidden class. Both objects are now sharing the some hidden class. This speeds up property access and uses less memory (less hidden classes). Adding the following additional line to algorithm 4.5 would result in a new hidden class for the *controlSocket* object:

```
controlSocket.debug = true;
```

The *dataSocket* and *controlSocket* objects now have different hidden classes, this will worsen performance and result in extra memory usage.

Property Access With Hidden Classes

The introduction of hidden classes into V8 was motivated by fast property access. Instead of doing a look-up in a hash-map to find the value of a specific property, V8 finds to offset for the requested property in its hidden class and uses this offset to get the value in the *properties* or *elements* backing store. This eliminates the need for slow dictionary look-ups.

4.2.2 Inline Caching

V8 employs a technique called *inline caching* to further speed up property access. Each call site initially uses a generic look-up function to access the property, the inline cache for this call site is in the *uninitialized* state. The first time this call site is executed, V8 goes through these steps:

1. Jump to the generic look up function
2. In this function:
   (a) Find the offset in the object's hidden class
   (b) Find the location of the backing store
   (c) Use the offset to find the exact position in the backing store
   (d) Return the exact location
3. Access the property

At this point, V8 will replace the call to the generic look up function with a code stub that directly points to the object. The inline cache for this call site is now in a *monomorphic* state.

The next time this call site is executed, V8 will use this new and faster code stub to locate the variable. V8 assumes that the property is at the same offset it was last time, this assumption is only guaranteed to be true if both objects use the same hidden class. The new code stub first tries to validate this assumption, if the assumption fails, V8 will bail out and return to using the generic look up function. When the location of the property is found, the location will be stored in the inline cache as well. The inline cache is now a *polymorphic* state and contains cached fast property access code stubs for two hidden classes.

V8 will cache property access code stubs for a maximum of four different hidden classes at any call site[1]. If V8 needs to bail out more than 4 times and fall back to the generic look up code, the inline cache will transition into a *megamorphic* state. Inline caches in a *megamorphic* state always use the slower generic look up code and never return to a *polymorphic or monomorphic* state.

During a major garbage collection, all inline caches are cleared so that the garbage collector can clear up old code stubs but it gives the inline caches a chance to return to a faster *monomorphic* state as well. Sharing hidden classes appropriately has huge performance advantages.

[1]`https://code.google.com/p/chromium/codesearch#chromium/src/v8/src/ic.cc&sq=package:chromium&type=cs&l=696`

4.2.3 V8 Compiler

Unlike most scripting language engines, V8 does not use a interpreter to execute JavaScript code but compiles the code using a JIT compiler. JavaScript is compiled Just-in-Time when it is executed for the first time. This initial compilation is by a generic compiler that does very little optimizations; the key here is fast compilation. The compiler returns an assembly code stub for the current architecture (V8 supports the ARM, x86 and MIPS architectures). These code stubs are stored in CodeSpace inside the V8 heap.

The Optimization Compiler

V8 also has an optimization compiler called *CrankShaft*. Crankshaft profiles code execution and uses this data to determine which functions are *hot* and would benefit most from optimization. The CrankShaft profiler assigns a counter to each function that is decremented each time the function is called. The value that is subtracted from the counter depends on the size of the function; larger functions have larger decrement values. Creating very large functions to trigger optimization faster however is not a good idea because V8 never optimizes very large functions.

When CrankShaft decides to optimize a function, it will use type information it collected during profiling to generate a optimized code stub. V8 will do on-stack replacement of the old code stub and replace it with new and faster optimized code stub. This means CrankShaft makes assumptions about type information (parameters and return type). These assumptions are not guaranteed to be true every time the function is called and need to be checked. If these assumptions fail, V8 needs to bail out and replace the optimized call stub with the slower code stub that was generated by the generic JIT compiler. This process is called *deoptimization*.

While the dynamic nature of JavaScript doesn't impose type restrictions, writing code that does stick to some type restrictions will improve performance. Code optimizations and deoptimizations can be tracked by passing the *–trace-opt* and *–trace-deopt* flags to V8.

4.3 Benchmarking getImageData

In Google Chrome, 2D canvases can be rendered in accelerated and non-accelerated mode. For 2D canvases with fewer than 65792 (257*256)[2] pixels, software rendering is used regardless of whether hardware acceleration is available[3]. When available, hardware acceleration is always used for larger canvases. During development, a noticeable drop in performance of *CanvasRenderingContext2D::getImageData* was observed when the 2D canvas was hardware accelerated. This phenomenon is investigated in this section.

Hardware acceleration can be disabled in Chrome by enabling the *Disable accelerated 2D canvas* flag on the `chrome://flags` page.

4.3.1 Benchmarking

CanvasRenderingContext2D::getImageData will be benchmarked using a small test-case. The test-case, getimagedata.html, initializes itself by loading a 1.5MB JPEG image (3000px by 2400px) onto a 2D canvas. This 2D canvas measures 1000px by 800px meaning that the original image has to be resized. Loading the image before running the actual test is important because resizing is an expensive operation and could otherwise influence the test results.

[2]`https://code.google.com/p/chromium/codesearch/#chromium/src/third_party/WebKit/Source/core/frame/Settings.in&l=51&sq=package:chromium`

[3]`https://code.google.com/p/chromium/codesearch#chromium/src/third_party/WebKit/Source/core/html/HTMLCanvasElement.cpp&l=433&sq=package:chromium`

Google Chrome 31.0.1650.4 dev

	NVIDIA GeForce GT 650M	Intel HD Graphics 4000
Accelerated Mode	4.2642814 seconds	9.67104464 seconds
Non-Accelerated mode	3.23748142 seconds	3.24130776 seconds
Difference	**25% slower**	**66% slower**

Chromium 32.0.1657.0 (225974)

	NVIDIA GeForce GT 650M	Intel HD Graphics 4000
Accelerated Mode	4.32544265 seconds	9.67260493 seconds
Non-Accelerated mode	3.25453929 seconds	3.25660825 seconds
Difference	**25% slower**	**66% slower**

Table 4.1: Results of the getImageData benchmark

Each time the test-case is run, *CanvasRenderingContext2D::getImageData* is called 1000 times. To eliminate interference from other applications running in the background, the test-case will be run 100 times after which the average is calculated. Other applications on the system running the benchmark will be closed (as far as possible). Chrome will run with only one tab open (running the test-case) and with all plugins disabled.

Accurate timing information is recorded using *chrome.Interval* (see section 4.1.4). This object is only available when Chrome is started with the *–enable-benchmarking* command line flag. This flag gives JavaScript access to a high-resolution timer within Chrome (provided by the kernel).

4.3.2 Test Conditions

The test-case described in the previous section was run 8 times under different conditions. Two different versions of Chrome were used. The test-case was run 4 times in each version, twice with hardware acceleration enabled but on different GPU's and twice with hardware acceleration disabled, again on different GPU's. The GPU can be selected using the gfxCardStatus tool[12]. Chrome was restarted completely after each run.

The versions of Chrome that were used are a build from Google's dev channel (31.0.1650.4 dev) and a local build of the Chromium SVN (32.0.1657.0 (225974)).

The GPU's that were used are an NVIDIA GeForce GT 650M with 512MB of video memory and a Intel HD Graphics 4000 with 384MB of video memory.

Other specifications of the system that was used:

- **CPU:** 2,3 GHz Intel Core i7
- **RAM:** 16 GB 1600 MHz DDR3
- **OS:** OS X 10.8.5 (12F37)

4.3.3 Results

The results displayed in table 4.1 are the averages over 100 runs (one run consists of 1000 *getImageData* calls) for each condition. The last rows compare hardware rendering to software rendering.

From the timing results in table 4.1, it becomes clear that the type of GPU greatly impacts the performance of *CanvasRenderingContext2D.getImageData* in Chrome when hardware acceleration is used. It has no impact on the performance when hardware acceleration is disabled. Using hardware acceleration was slower under all conditions.

4.3.4 Identifying causes

In the previous section it was determined that the type of GPU has a big impact on the performance of *CanvasRenderingContext2D.getImageData* in accelerated mode. This observation can easily be explained because when hardware acceleration is used, most work is done by the GPU and the NVIDIA GeForce's memory interface (80GB/s - GDDR5) is a lot faster than the memory interface on the Intel HD Graphics 4000 (25.6GB/s - Shared DDR3 Memory). While the video memory buffer of the Intel Graphics 4000 resides in main memory, it is not possible for the CPU to read it directly, it has to go through the slow memory interface. In this section, a dive into the internals of Chrome hopefully will yield more indications why *CanvasRendering-Context2D.getImageData* runs slower in accelerated mode.

4.3.5 Analysis of getImageData

The graphics back-end in Chrome was explained in much more detail earlier (Section 3.3) but to shortly reiterate: there are two rendering paths, a software path and a hardware path. The software path uses a simple *SkBitmap* for software rasterization. This bitmap is stored in main memory. The hardware path (accelerated mode), uses Ganesh, an OpenGL back-end for Skia. The actual bitmap is stored in video memory on the GPU instead of main memory.

This difference is abstracted in *WebCore::GraphicsContext*. In software mode, the GraphicsContext uses Skia directly to draw or read from the skBitmap in main memory. In hardware mode, the GraphicsContext sends OpenGL calls to the GPU process through IPC. The GPU executes these (OpenGL) commands and draws or reads from a buffer in video memory.

Analysis of getImageData in non-accelerated mode

Google Chrome has some performance analysis tools build-in as part of the Developer Tools which can be used to analyze most aspects of the browser like JavaScript execution, memory usage, layout and rendering. Unfortunately, these tools do not provide us with enough detail.

To get a good insight into what is happening inside Chrome (specifically Skia for software rendering) while a *CanvasRenderingContext2D.getImageData* call is being executed, a trace was recorded using `chrome://tracing/`. The tracing tool is integrated into the actual C++ code and can be extended to record almost everything.

The default traces that are recorded inside Chrome were not sufficient either so extra traces were added to the source code, mostly inside Skia. The recorded traces contain one test-run each (1000 calls). Figure 4.7 shows one *CanvasRenderingContext2D.getImageData* call in detail.

The length of each bar indicates how long the function was on the calling stack (total execution time), this includes the time spent in functions called by this function. The stacking indicates the calling order but is not a complete calling stack. Only functions which explicitly use one of the tracing macros[4] are displayed here.

As you can see in figure 4.7, most time is spent converting the pixels from one representation to another (source to destination bitmap). The *CanvasRenderingContext2D.getImageData* call in this sample took 3.282 ms, the *SkConvertConfig8888Pixels* call took 3.265 ms (99% of the entire *getImageData* call). Copying the bitmap from main memory to main memory is a very cheap operation compared to converting the pixels. This all happens inside the renderer process which runs on the CPU.

[4]`https://code.google.com/p/chromium/codesearch#chromium/src/base/debug/trace_event.h&sq=package:chromium`

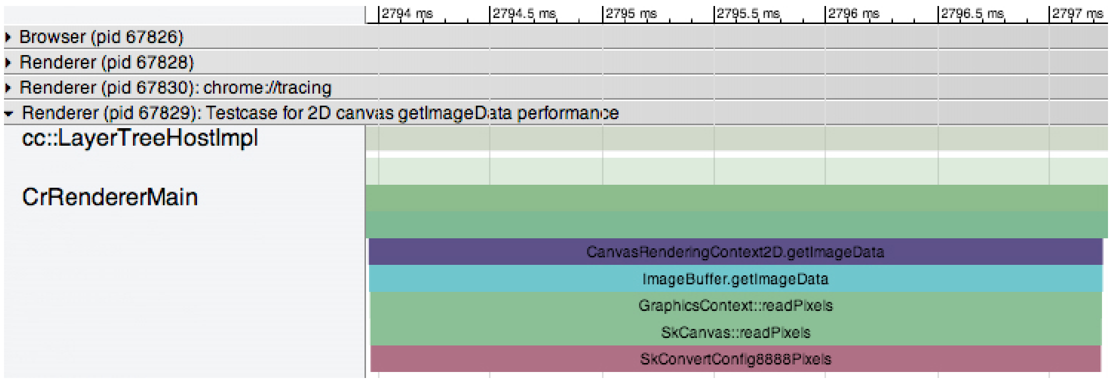

2794 ms
2794.5 ms
2795 ms
2795.5 ms
2796 ms
2796.5 ms
2797 ms
Browser (pid 67826)
Renderer (pid 67828)
Renderer (pid 67830): chrome://tracing
Renderer (pid 67829): Testcase for 2D canvas getImageData performance
cc::LayerTreeHostImpl
CrRendererMain
CanvasRenderingContext2D.getImageData
ImageBuffer.getImageData
GraphicsContext::readPixels
SkCanvas::readPixels
SkConvertConfig8888Pixels

Figure 4.7: Detailed trace of one getImageData call in non-accelerated mode. The GraphicsContext sends calls to a skCanvas directly (backed by a skBitmap in main memory).

Analysis of getImageData in accelerated mode

Due to the multi-process architecture of Chrome, the execution of a *CanvasRenderingContext2D.getImageData* call in accelerated mode (hardware rendering) is very different. The GPU is instructed from a GPU process (this is not true for all platforms, on Android for example, this is a GPU thread). The renderer has no direct access to the GPU process and has to send instruction through IPC (usually through the browser process). This is likely to create additional overhead.

To get a good insight into the workings of *CanvasRenderingContext2D.getImageData* in accelerated mode, a new trace was recorded. Additional traces were again added to the source, mostly inside Skia and the GLES2 implementation (which runs inside the GPU process).

The trace in figure 4.8 shows the renderer process and the GPU process. The renderer process executes *CanvasRenderingContext2D.getImageData* and sends a message to the GPU process through IPC requesting the pixel data for the canvas. It takes ∼0.19 ms before *glReadPixels* is called by the GPU process. This call takes 3.341 ms to execute after which it take another 0.7 ms before the initial *getImageData* call is finished in the renderer process.

Execution of *CanvasRenderingContext2D.getImageData* in this sample took 4.235 ms. ∼78% of that is used by *glReadPixels*. The other 22% is mostly communication overhead between processes. When using a 2D canvas in accelerated mode, *glReadPixels* is the most expensive call. *glReadPixels* copies data from the frame-buffer (in video memory) to main memory.

The cost of *glReadPixels* is very dependent on the type of GPU. The trace displayed above was recorded while using the NVIDIA GeForce GT 650M. When a trace is recorded when the Intel HD Graphics 4000 is in use, *glReadPixels* takes 8.152 ms to execute. This is 89% of the total time needed.

4.4 Benchmarking getImageData with different canvas sizes

The test-case used for this benchmark, canvassizebenchmark.html, performs 250 *CanvasRenderingContext2D.getImageData* calls during each run. The test-case was run 25 times (6250 calls in total) for each size after which the average is calculated. Other applications on the system running the benchmark were closed (as far as possible). Chrome was run with only one tab open (running the test-case) and with all plugins disabled.

The test-case benchmarks 22 canvases of increasing size. The initial canvas has a surface area of 10000 pixels (100px by 100px). The surface area is increased by 50% each time. The largest

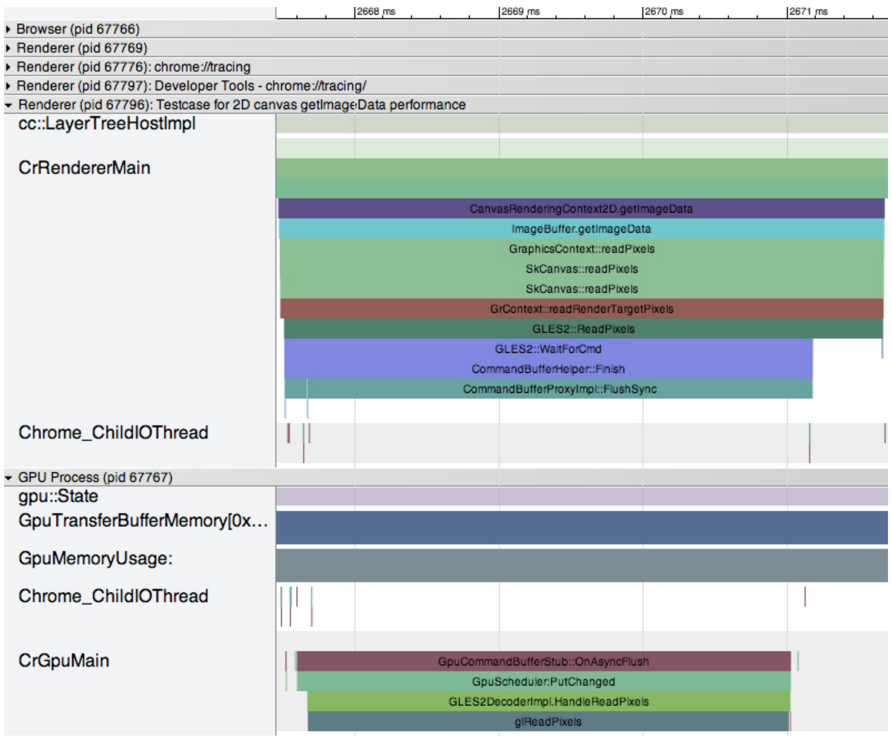

2668 ms
2669 ms
2670 ms
2671 ms
Browser (pid 67766)
Renderer (pid 67769)
Renderer (pid 67776): chrome://tracing
Renderer (pid 67797): Developer Tools - chrome://tracing/
Renderer (pid 67796): Testcase for 2D canvas getImageData performance
cc::LayerTreeHostImpl
CrRendererMain
CanvasRenderingContext2D.getImageData
ImageBuffer.getImageData
GraphicsContext::readPixels
SkCanvas::readPixels
SkCanvas::readPixels
GrContext::readRenderTargetPixels
GLES2::ReadPixels
GLES2::WaitForCmd
CommandBufferHelper::Finish
CommandBufferProxyImpl::FlushSync
Chrome_ChildIOThread
GPU Process (pid 67767)
gpu::State
GpuTransferBufferMemory[0x...
GpuMemoryUsage:
Chrome_ChildIOThread
CrGpuMain
GpuCommandBufferStub::OnAsyncFlush
GpuScheduler:PutChanged
GLES2DecoderImpl.HandleReadPixels
glReadPixels

Figure 4.8: Detailed trace of one getImageData call in accelerated mode. The GraphicsContext sends OpenGL calls to the GPU process via IPC.

canvas in this test measures 7062px by 7062px giving a surface area of roughly 50 million pixels. One would expect the time a *getImageData* call takes to be linear with the surface area which is why the surface area is increased in steps instead of the width and height.

The same image is placed on all canvases. This 4.2MB JPEG image measures 3264px by 3264px and is resized when it is placed on the canvas before the benchmark is started. The actual content of the canvas is not relevant as it is stored as a bitmap without compression. The image is decoded once and lives as a bitmap from that point onwards, the actual image that is used does not influence this benchmark.

Accurate timing information is recorded using *chrome.Interval*. This object is only available when Chrome is started with the *–enable-benchmarking* command line flag. This flag gives JavaScript access to a high-resolution timer within Chrome (provided by the kernel).

4.4.1 Test Conditions

From the previous section (results in table 4.1) we can conclude that the version of Chrome has no impact on the performance of *CanvasRenderingContext2D.getImageData*. Because of this, only one version of Chrome was used. The benchmark was ran under the following conditions:

- With hardware acceleration disabled
- With hardware accelerated enabled using the Intel HD Graphics 4000
- With hardware accelerated enabled using the NVIDIA GeForce GT 650M

The GPU was selected using the gfxCardStatus[12] tool. Chrome was restarted completely after each run. The version of Chrome that was used is 32.0.1671.3 dev.

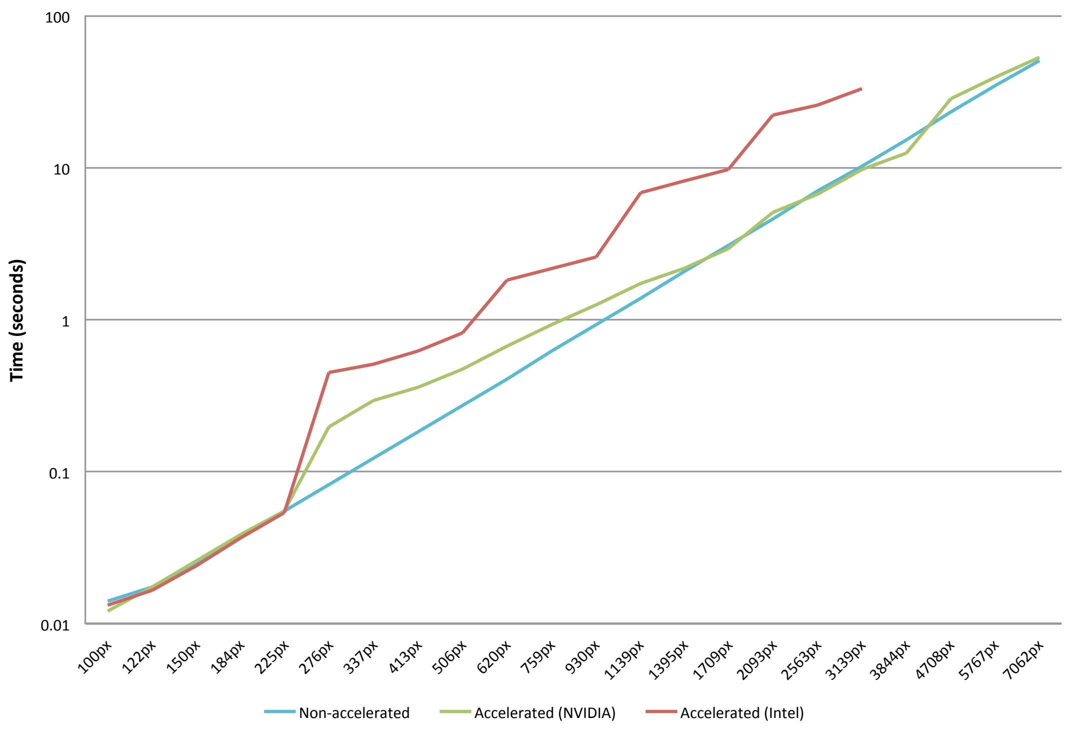


Figure 4.9: Performance comparison of getImageData calls on canvasses of different size. The performance drop is clearly visible here when the canvas becomes hardware accelerated.

The GPU's that were used are a NVIDIA GeForce GT 650M with 512MB of video memory and a Intel HD Graphics 4000 with 384MB of video memory. The test system is the same that was used for the previous tests.

4.4.2 Results

The results displayed in figure 4.9 are the averages of 25 runs (one run is 250 getImageData calls) for each condition. Google Chrome does not accelerate canvases with a surface area of less than 65792 pixels (257px by 256px). This can clearly be seen in the chart above. The lines drift away once the canvas size exceeds this limit and becomes hardware accelerated. At this point, hardware accelerated mode is slower. The NVIDIA GeForce recovers as the canvases increase in size but the Intel HD Graphics remains noticeably slower. When running on the Intel HD Graphics GPU, the four largest canvases were not tested because the test system ran out of video memory and became unstable and crashed. Note that the chart in figure 4.9 has a logarithmic x axis.

To compare the performance of accelerated and non-accelerated mode, a chart with relative times is more useful. The time in non-accelerated mode is used as the baseline.

The chart in figure 4.10 shows the same phenomena, once the canvas size exceeds the acceleration limit, hardware acceleration becomes noticeably slower with the NVIDIA GeForce recovering from this as the canvas size increases further. The Intel HD Graphics remains slow.

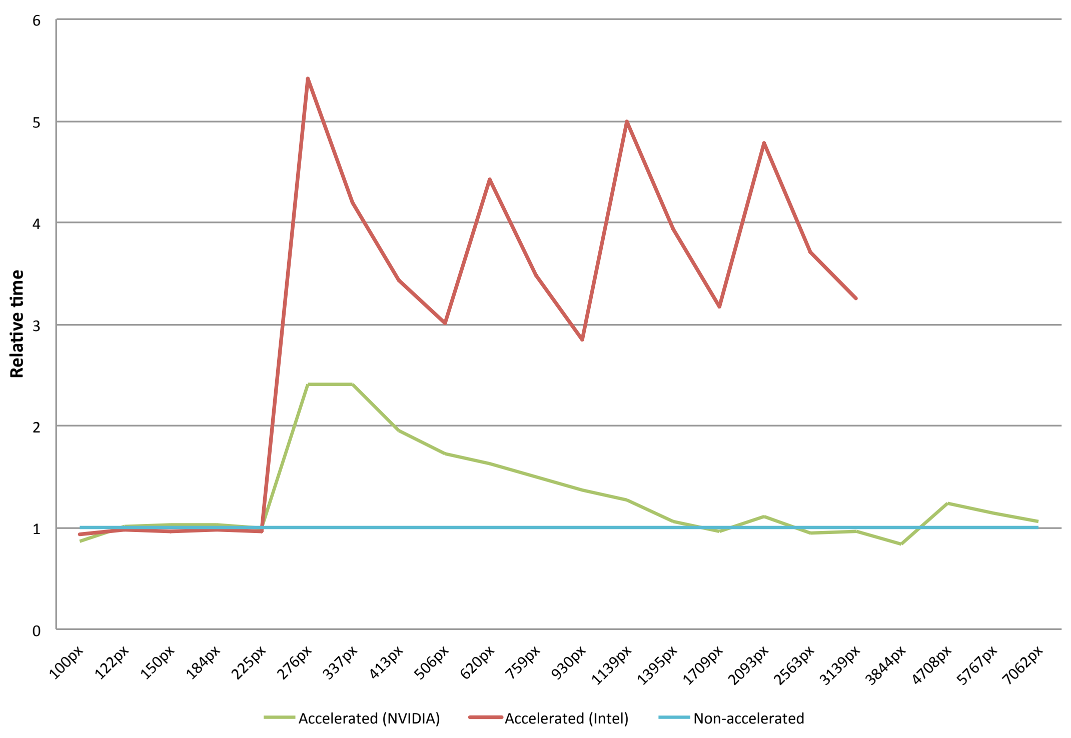


Figure 4.10: Relative performance comparison of getImageData calls on canvasses of different size. The performance drop is clearly visible here when the canvas becomes hardware accelerated.

4.4.3 Analysis of getImageData

To investigate whether *CanvasRenderingContext2D::getImageData* behaves the same internally, the traces that were done for the previous sections were recorded again but this time to see if the behavior of the function changes as the canvas size increases.

Non-accelerated mode

The recorded trace shows that most of the time *CanvasRenderingContext2D::getImageData* is still spend inside *SkConvertConfig8888Pixels*. The exact percentage changes as the canvas size increases. For the smallest canvas, *SkConvertConfig8888Pixels* takes up 83% of the total execution time. This percentage rises to over 99% as the canvas increases in size.

Accelerated mode

In accelerated mode, the behavior of *CanvasRenderingContext2D::getImageData* changes as the canvas size increases. When the canvas exceeds a certain size, the *CanvasRenderingContext2D::getImageData* is handled by multiple calls to *glReadPixels*. The sum of the calls to *glReadPixels* still takes up most of the time. The percentage of time spend inside *glReadPixels* varies between 57% and 97%.

This percentage increases as the canvas grows in size because the overhead caused by IPC remains steady while the execution time of *glReadPixels* increases. The percentage slightly decreases again when multiple *glReadPixels* calls are used to get the pixel data. When this happens, the IPC overhead increases because each call is send over IPC.

4.5 Decoding images

Decoding images is a slow process, especially when the images are large. Browsers give web authors little to no control over how or when images are decoded. In Google Chrome, images are downloaded or fetched from the cache immediately (as required by the HTML specification) but decoded only when they are actually needed. For most images this means decoding is done during rasterization (painting) of the page. An image that is not part of the page or not scrolled into view[5] will not be decoded to save memory and CPU cycles. When the image will be used as the source of WebGLTexture, different methods can be used to load the image and to some extend control when the image is decoded.

This section looks at the decoding performance of several image types and discusses some techniques and new technologies that can be used to decode images and load them onto WebGL-Textures through JavaScript.

4.5.1 Tested image formats

Google Chrome has build-in support for the JPEG, PNG, GIF(, ICO), BMP and WEBP image formats. We will focus on the lossless image types; PNG, BMP and WEBP. We will run a benchmark to see which image formats are decoded the fastest. The benchmark also includes lossy JPEG. GIF was not included in the test because it has very limited color support.

The same source image will be used for all benchmarks. This source image is a 18000px by 18000px TIFF image made by the Hubble Space Telescope. The image was selected because it has a lot of detail. The image was resized and converted to the correct sizes and formats. The benchmark starts with an image containing 10000 surface pixels (100px by 100px), the number of surface pixels will be increased by 50% each run until we reach an image measuring 7062px by 7062px.

The times were recorded using a custom build of the Chromium SVN. The build included extra code that records the time spend inside the image decoders.

JPEG

The JPEG image format is the only lossy image format tested in this benchmark. The JPEG test-images are created with PhotoshopAdobe [3] and saved with maximum quality.

PNG

The PNG images, like the JPEG images, are created with Photoshop. Unlike the JPEG images, the PNG images are lossless which results in larger files. This could be a problem when the images are served over a network connection.

WEBP

The WEBP image format was designed specifically to decrease the file size. WEBP images can be lossy or lossless. The test-images used here are lossless. The file sizes of the test-images are indeed smaller than the equivalent PNG images. The WEBP test-images are created using the *cwebp* command line tool.

[5]This is not always true, Google Chrome prerenders parts of the page that are not in view but are expected to be soon.

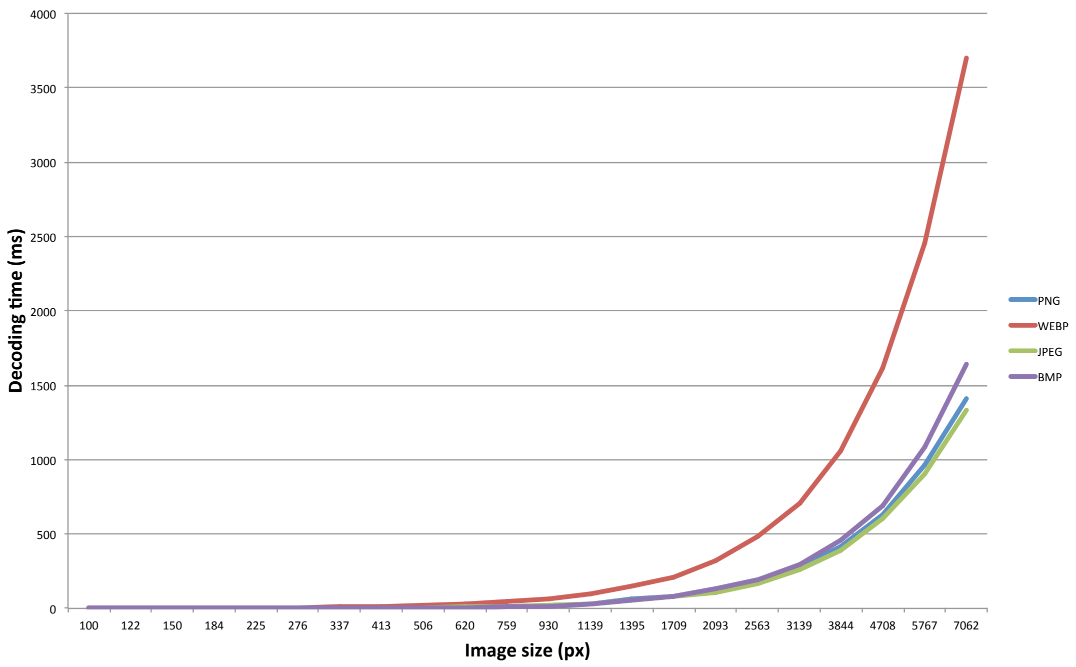


Figure 4.11: Comparison of the decoding times of images of different size and different type. WEBP is the slowest encoding because it compresses more aggressively. The other image formats are very close in terms of decoding time.

BMP

The last image format used in this test is the BMP image format. This lossless image is similar to the PNG image format. The BMP test images are created with Photoshop as well.

There are many image compressors available for the different image formats. Most compressors can be configured as desired. It could be interesting to further research and test different configurations to see which yield smaller files, preserve image quality and have the least impact on the decoding time.

4.5.2 Decoding times

During the benchmark each test image was decoded 100 times. The averages are displayed on the graph below.

The cost incurred for the smaller file size of WEBP images becomes clear, the decoding time is a lot worse than that of the other lossless image types. While preparing the test images, encoding the WEBP was noticeably slower as well. The other image formats perform pretty similar with the PNG format being a little faster than the BMP format for the larger images. When decoding smaller images, the BMP format tends to be faster.

The PNG format on average only needed 40% of the time the WEBP images needed to decode the equivalent image.

This benchmark only looks at the actual decoding times. When the images are served over a network connection the file size could be important as well.

4.6 Loading images

The only way to instruct a web browser to create a new image resource is to create an *Image* object. As mentioned before, a web browser will only fetch the resource from the network or cache at this time.

4.6.1 The HTML specification on images

The HTML specification requires the image *load* event to fire when the image is *Completely available.* This term is defined in the HTML specification[34] as follows:

> The user agent has obtained all of the image data and at least the image dimensions are available.

This means that the image is not yet decoded when the *load* event fires, there are no requirements regarding decoding in the HTML specification. There is also no means to trigger an image decode through JavaScript or be informed of image decoding progress. New technologies that give web authors some control over this process are discussed later.

In Google Chrome, images are decoded when they are needed. This means most images are decoded during rasterization (painting), this can introduce noticeable lag to the user. Since painting by default still happens on the renderer thread, image decoding will block JavaScript as well.

4.6.2 WebGL and images

Images that are to be displayed on a WebGL canvas are usually created through JavaScript and are not part of the visual DOM. These images are decoded when they are loaded onto the canvas. Loading large images onto a canvas is subject to the same decoding lag caused by image decoding during rasterization and can be noticed by the user or make the application unresponsive for a few seconds. This section looks at different techniques that can be used to load images onto a canvas and compares the performance.

To WebGL, an image is just a 2D texture and WebGL provides several ways to load pixel data onto such a WebGL texture;

- by providing an *ArrayBufferView* object;
- by providing an *ImageData* object;
- by providing an *HTMLImageElement;*
- by providing an *HTMLCanvasElement*;
- it by providing an *HTMLVideoElement* (not investigated further).

Loading with an ImageData Object

The *ImageData* Object represents pixel data from a 2D canvas. The objects has the following properties; width, height, resolution and data. The data property is a *Uint8ClampedArray* holding the actual pixel data. The only way to get an *ImageData* object is through a *CanvasRenderingContext2D* meaning the image has to be loaded onto a 2D canvas first.

This technique, shown in figure 4.12, creates additional overhead compared to loading the *HTMLCanvasElement* directly; it includes a *CanvasRenderingContext2D::getImageData* call which from previous research can be considered slow (Section 4.3). This technique can be used to cache the pre-decoded image for later use, this idea is described later.

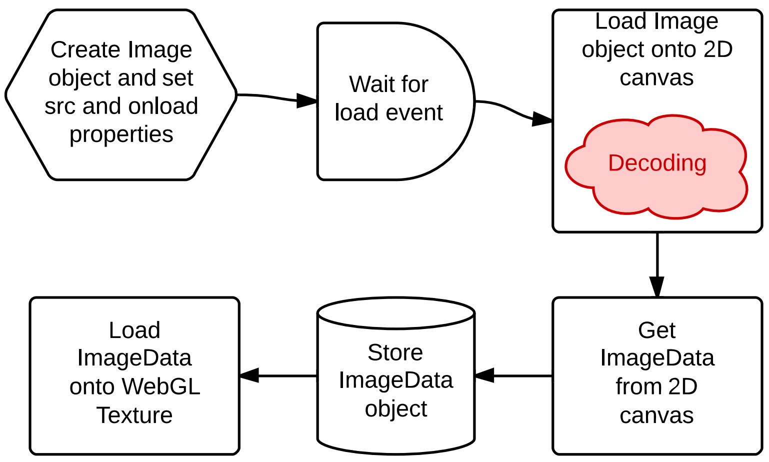

Create Image object and set src and onload properties
Wait for load event
Load Image object onto 2D canvas
Decoding
Get ImageData from 2D canvas
Store ImageData object
Load ImageData onto WebGL Texture

Figure 4.12: Visualization of loading cached ImageData onto a WebGL Texture. The browser will decode the image when it is being loaded onto the 2D canvas.

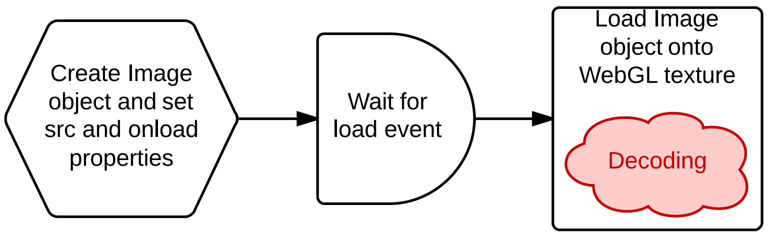

Create Image object and set src and onload properties
Wait for load event
Load Image object onto WebGL texture
Decoding

Figure 4.13: Visualization of loading a HTMLImageElement onto a WebGL Texture. The browser will decode the image when it is loaded onto the WebGL texture.

Loading with an HTMLImageElement

A *HTMLImageElement* or *Image* object in JavaScript can be loaded onto a WebGL texture directly. This technique, shown in figure 4.13, is the simplest way to get an image onto a WebGL canvas but it also gives the author the least control over decoding and loading. When this technique is used, the image is decoded when it is loaded onto the WebGL texture for the first time.

Loading with an HTMLCanvasElement

The last technique discussed here is loading a *HTMLCanvasElement* onto a WebGL texture as shown in figure 4.14. To get the image onto the WebGL texture, the image is loaded onto the 2D canvas first. This 2D canvas is then passed to the WebGL texture. The image is decoded when it is first loaded onto the 2D canvas.

Loading with an ArrayBufferView Object

Instead of using an actual image object, it is also possible to load an already decoded image onto a WebGL texture. The decoded images is a bitmap, each pixel is represented by 4 bytes (RGBA) and stored in an ArrayBufferView object. This object can be used as the source of a WebGL texture. The advantage of this is that the image is already decoded, the disadvantage is that the decoded image uses more memory and bandwidth if it is send over a network.

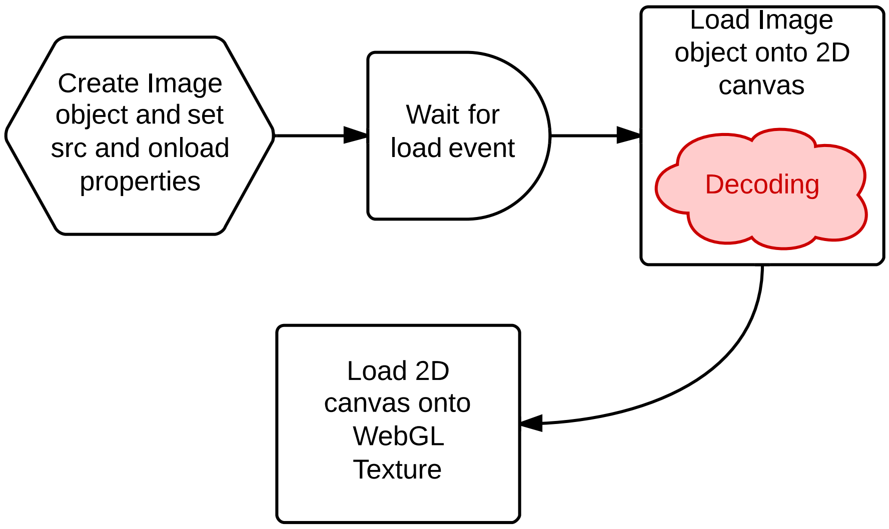

Create Image object and set src and onload properties
Wait for load event
Load Image object onto 2D canvas
Decoding
Load 2D canvas onto WebGL Texture

Figure 4.14: Visualization of loading a HTMLCanvasElement onto a WebGL Texture. The browser will decode the image when it is being loaded onto the 2D canvas.

4.6.3 Performance

The displayed *total time* for all techniques is the fastest time in which it is possible to go from a new *Image* object[6] to the image being loaded onto the WebGL texture. Timing information for relevant sub-steps are provided as well.

The image that was used during the benchmarks measured 4096px by 4096px, all canvases had the same dimensions.

Figure 4.15 displays the recorded timing information for the different image loading techniques. These times are JavaScript execution times, the time that JavaScript execution was blocked/busy. The total time does not imply time-to-screen.

The most obvious difference between the four techniques is the lack of a *Loading onto texture* step when using a 2D canvas as the source for the WebGL texture. This is because unlike the two other techniques, the *Loading onto texture* step is *asynchronously* handled by the by GPU process. When the other techniques are used, this step is done **synchronously**, blocking JavaScript execution. The time before the image is actually displayed on the screen is very similar for all techniques[7]. The ArrayBufferView technique does not need to decode the image, making it the fastest technique. It comes with higher memory and bandwidth usage though. Discarding the alpha channel partly solves this but cannot be used for images that need opacity.

Image decoding only takes up a small part of the total time that is required to get the image displayed on the screen, most time is lost on the GPU thread uploading the decoded image. The technique where a 2D canvas is passed to the WebGL texture just blocks JavaScript execution the least amount of time.

[6] Just before setting the src property
[7] Actual times were not recorded since this is not possible with JavaScript

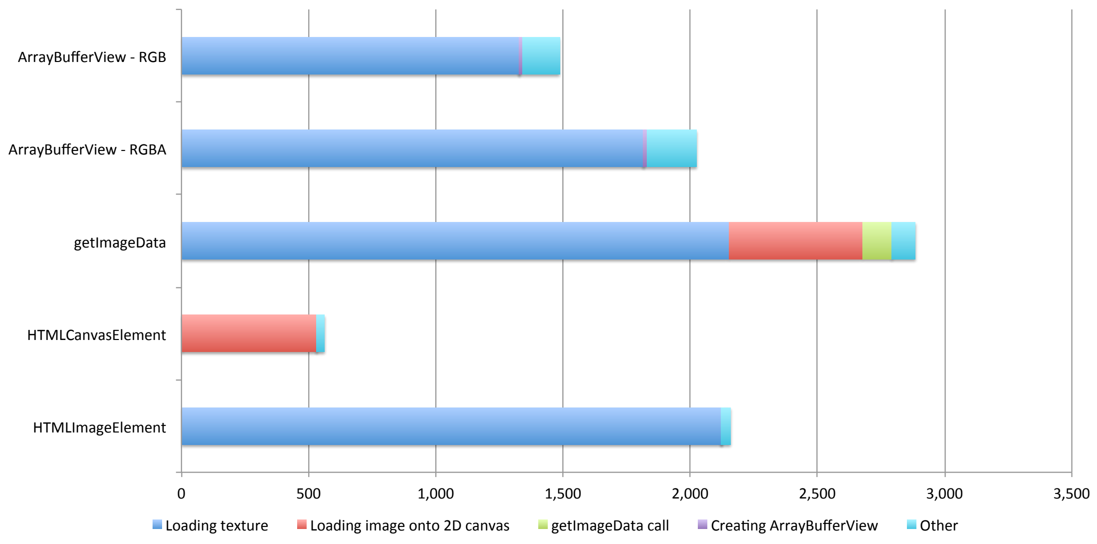


Figure 4.15: Comparison of the different image loading techniques

4.6.4 Off-thread decoding

It is possible to move image-decoding of the renderer thread of the current page by using other web technologies. Three techniques are briefly described here but unless image decoding is a real problem it is best not to use them as they introduce a lot of overhead.

Using an iframe

A web page can off-load image decoding to an iframe, by sending image decoding request to the iframe and send back the resulting pixel data. This technique has a lot of communication overhead because the data has to be copied multiple times. Communication between the two documents can be done using *Window.postMessage*. The performance gains would be minimal because loading the image onto a WebGL texture would still take up a lot of time. The only advantage is that image decoding does not block JavaScript execution.

Using multiple tabs and a Shared Web-Worker

Similar to using an iframe to do image decoding, a separate tab can be used to achieve the same results. Both tabs can communicate through a Shared WebWorker and MessagePorts[34]. This technique suffers from the same problems as the previous one.

Using a Web-Worker and a JavaScript decoder

One or multiple Web-Workers can be used to handle image decoding. Each web-worker can be instructed to decode a JPEG image by using a JavaScript decoder like jpgjs[52]. The actual decoding will be slower than the native C++ implementation used by Chrome. Security and quality could be issues as well.

These techniques can be used to move image decoding of the renderer thread or to do decoding on a as-needed basis but they all have major drawbacks and should be used with caution.

4.7 Conclusion

The built-in Developer Tools in Chrome form a full-featured IDE providing developers with essential debugging and analysis tools. The timeline view visualizes the internal operations performed by Chrome for rendering, painting, and JavaScript execution and the associated costs. The tools in the profiling section include a CPU profiler for identifying costly operations, a memory allocation profiler, and a heap snapshot tool for detecting memory leaks. The advanced tracing tool gives developers an in-depth look into the internal working of Chrome and their applications. JavaScript objects like *window.performance* and *chrome.Interval* expose high resolution timers to JavaScript developers. Combined, these tools help developers understand their applications and potential optimizations. It makes developing for the web a lot easier.

In V8, the concept of hidden classes, inline caching, and the optimizing compiler were introduced to make the JavaScript engine faster. The hidden classes introduce a notion of type in an otherwise dynamically typed language. This makes that V8 can employ techniques used in static typed language like inline caching to improve performance. Inline caching is were the generic look-up code for object properties is replaced by the direct reference of the object at the call sites to speed up property access. Further performance improvements can be found in the optimizing compiler, it re-compiles *hot code* and applies optimization techniques to generate high-performing code stubs.

This chapter also took an in-depth look at the performance difference between *getImageData* calls on accelerated and non-accelerated canvases. The underlying cause was found in the differences between the software and hardware paths. The pixel data for a non-accelerated canvas is stored in main memory and can be accessed and copied by the CPU quickly. The pixel data for an accelerated canvas is stored in video memory on the GPU, a slow readback is required to get it to the CPU which caused the observed difference in performance. There is some IPC communication overhead caused by Chrome's multi-process architecture as well.

The rest of the chapter is focused on image loading and decoding. The different methods for loading images onto a WebGL texture have very similar performance characteristics. The 2D canvas to WebGL canvas method has the advantage of an asynchronous texture upload which does not block JavaScript execution. The decoders for the different image showed very similar performance, the only exception was the noticeable slower WEBP format. The advantage of the WEBP format is the greatly reduced file size.

Chapter 5

Drawing Text and Handwritten Lines with WebGL

WebGL does not have native support for rendering text, handwritten text, or other shapes. This means text and handwritten lines have to be rendered separately using a different technique and loaded onto the WebGL canvas afterward. There are different techniques for drawing text and lines and loading the result onto a WebGL canvas.

5.1 Text

This section looks at two techniques for drawing text onto a WebGL canvas; using a 2D canvas to render the text and using Scalable Vector Graphics[17] to render the text. The resulting images (text quality) will be compared.

Another possibility is to use HTML elements and position them above the canvas. When using this technique, the text is not actually part of the canvas. During development, this technique also caused performance problems.

5.1.1 Drawing text using a 2D canvas

While a *WebGLRenderingContext* has no native text-rendering support, a *CanvasRendering-Context2D* exposes two methods for text-rendering; *strokeText* and *fillText*. These methods can be used to work around the WebGL limitation. The idea behind this technique is very simple, first draw the text onto the 2D canvas and convert it to a WebGL Texture afterward.

The text-rendering capabilities of a 2D canvas are very limited, especially compared to standard HTML but should suffice for most applications.

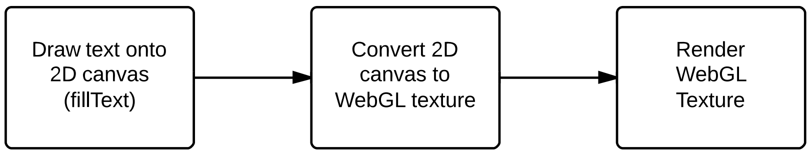

Draw text onto 2D canvas (fillText)
Convert 2D canvas to WebGL texture
Render WebGL Texture

Figure 5.1: Process of drawing text onto a WebGL canvas using a 2D canvas

Algorithm 5.1 WebGLTextTexture example

```
var canvas = document.createElement('canvas');
var context = canvas.getContext('webgl');
var textTextureHelper = new WebGLTextTextureHelper(context, '
   canvas');
var textElement = new WebGLTextElement('Example text', 60, 60);
textTextureHelper.addTextElement('example', textElement);
...
context.bindTexture(context.TEXTURE_2D, textTextureHelper.
   getTexture());
```

A WebGLTextTexture example

The implementation contains a *WebGLTextTextureHelperA.1* object. This object encapsulates the implementation details of the technique described above (and the SVG technique explained later). A *WebGLTextTextureHelper* objects takes the WebGLRenderingContext that will render the texture. A second optional parameter can be used to chose the technique to use (a 2D canvas or SVG). The *WebGLTextElementA.2* represent one line of text. Each *WebGLTextElement* can have a different font, color and size. The *WebGLTextElement* objects are passed to the *WebGLTextTextureHelper* that will render the text onto a *WebGLtexture* that can be retrieved by calling *WebGLTextTextureHelper::getTexture.*

To optimize performance, the texture is only redrawn and reloaded when needed. The *WebGL-TextTextureHelper* tracks changes made to all *WebGLTextElement*s it has to render, when a change is recorded the texture is marked invalid and will be redrawn when the texture is requested with *WebGLTextTextureHelper::getTexture.* Multiple changes between two *WebGLTextTextureHelper::getTexture* calls will only trigger one redraw so changes can be batched.

Quality of the resulting text

Figure 5.2 shows a sentence containing all characters in the alphabet rendered onto a WebGL canvas using the technique described above. The second image shows some characters in more detail.

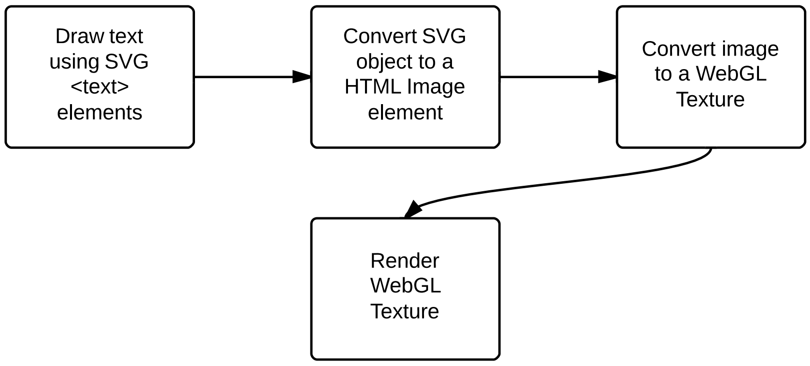

Draw text using SVG <text> elements
Convert SVG object to a HTML Image element
Convert image to a WebGL Texture
Render WebGL Texture

Figure 5.3: Process of drawing text onto a WebGL canvas using SVG

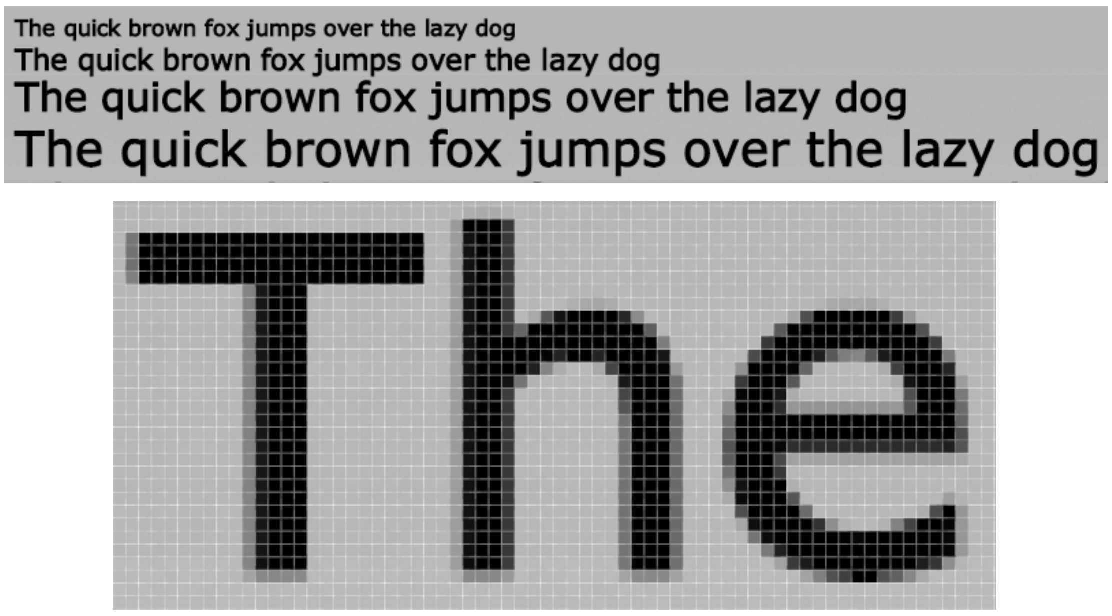

The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The

Figure 5.2: Text rendered in different sizes in Verdana using a 2D canvas

5.1.2 Drawing text with SVG

Another technique, shown in figure 5.3, to work around the WebGL text-rendering limitation is to use SVG. The text-rendering capabilities of SVG are better than those of a 2D canvas but still behind those of standard HTML. This extra functionality comes at a cost. The process is more resource intensive. The idea behind this technique is to first draw the text using SVG's *text* element. The entire SVG element can be converted to a XML string which can be used as the source of a *HTMLImageElement*. This *HTMLImageElement* is then loaded onto a WebGL texture.

This approach works in Firefox but the WebGL implementation in Google Chrome is bit more picky about what image resources it accepts. To make this work in Google Chrome an extra step is needed. Instead of converting the image to a WebGL texture directly, it first has to be loaded onto a 2D canvas that can be converted to a WebGL texture as displayed in figure 5.4.

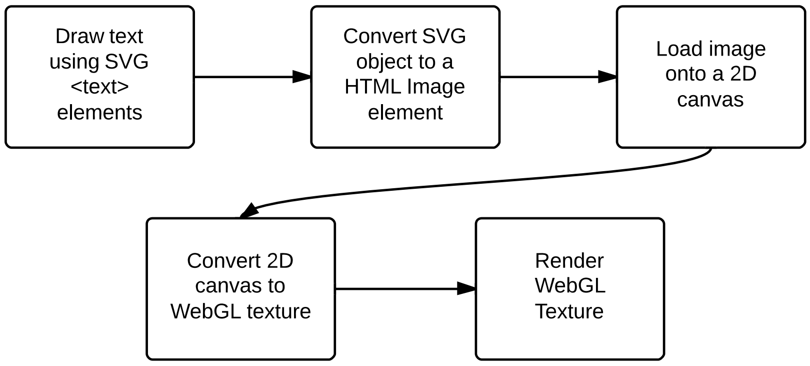

Draw text using SVG <text> elements
Convert SVG object to a HTML Image element
Load image onto a 2D canvas
Convert 2D canvas to WebGL texture
Render WebGL Texture

Figure 5.4: Process of drawing text onto a WebGL canvas using SVG with an explicit rasterizing step

Quality of the resulting text

Figure 5.5 shows the same sentence used previously rendered onto a WebGL canvas using SVG followed by a detail view of the pixel rendering.

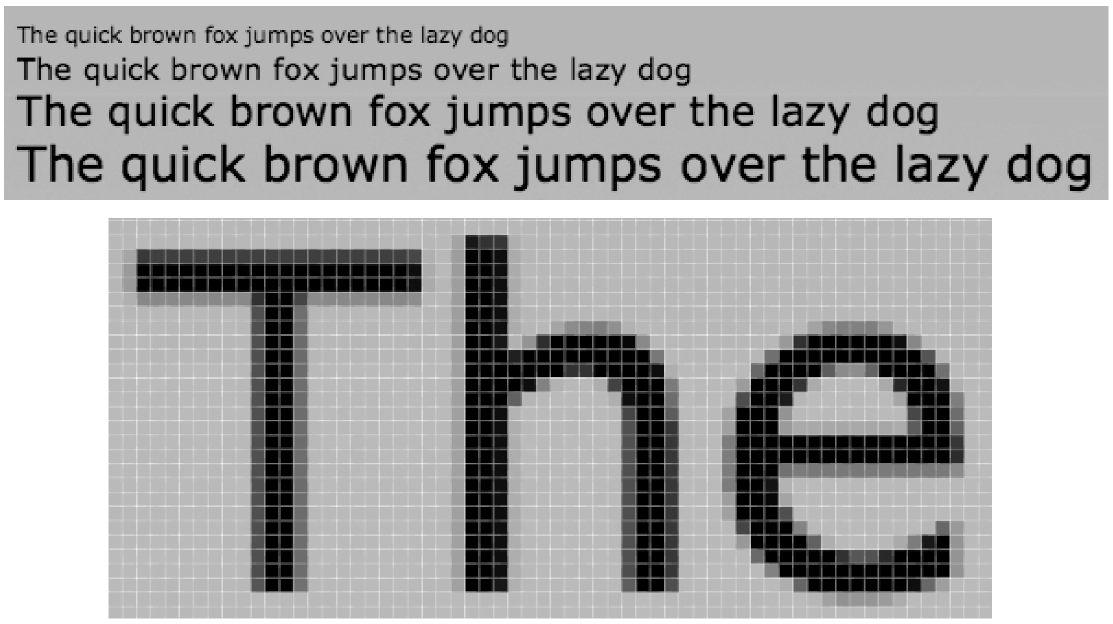

The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The

Figure 5.5: Text rendered in different sizes in Verdana using SVG

5.1.3 Comparison

The image below shows text rendered in different colors and font sizes using both techniques. The technique using the 2D canvas font-rendering techniques *strokeText* and *fillText* is displayed on the left, the technique based on SVG's *text* tag on the right.

The SVG technique renders the font sharper than the 2D canvas technique. This could cause legibility problems for small font-sizes. Other than that, the SVG technique tends to give slightly

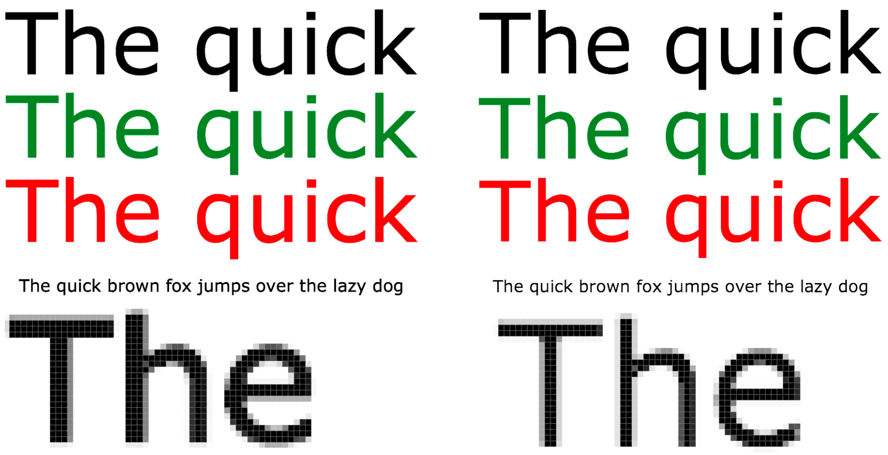

The quick
The quick
The quick
The quick brown fox jumps over the lazy dog
The
The quick
The quick
The quick
The quick brown fox jumps over the lazy dog
The

Figure 5.6: Comparison of both techniques with 2D canvas rendering on the left and SVG rendering on the right.

better overall results but has the disadvantage of the additional *img* to *canvas* conversion which impacts performance.

The screen-shots shown here were made on OSX. Font-rendering in Google Chrome running on OSX or Linux is done by the HarfBuzz font shaping engine[66]. Google Chrome on Windows uses Uniscribe[49] but work is underway to move to DirectWrite[50][1]. DirectWrite support is expected to ship in Chrome 37[7]. This causes differences in font-rendering between platforms. It is best to test which technique yields the best results for the current application.

5.2 Handwritten Lines/Annotations

This section looks at the same 2D canvas to WebGL texture techniques that was used earlier to draw text but this time to draw lines and annotations instead. The SVG technique described earlier can be used here as well. Instead of placing text inside the SVG element, SVG paths can be used to draw lines or more complex shapes.

5.2.1 Drawing lines using a 2D canvas

A *WebGLRenderingContext* has no native support for drawing simple lines or annotations (handwritten text or shapes). It is possible to draw these shapes onto a WebGL canvas using the same techniques used for the text rendering. The *CanvasRenderingContext2D* exposes several methods for drawing paths. When the paths are drawn, the state of the 2D canvas can be uploaded to a WebGL texture. The updated process is shown in figure 5.7.

Figure 5.8 shows two examples of handwritten annotations. The first annotation is drawn on top of an image, the second is handwritten text on a white background. In both cases the annotations blend together with the background very well. This simple technique gives very good and readable results on almost all backgrounds (when you look past my horrible handwriting).

5.2.2 Synchronizing Annotations

Handwritten annotations are a great tool to support collaboration. Multiple clients can analyse one image by joining a conference call and adding annotations to the image. These annotations

[1] `http://crbug.com/25541`

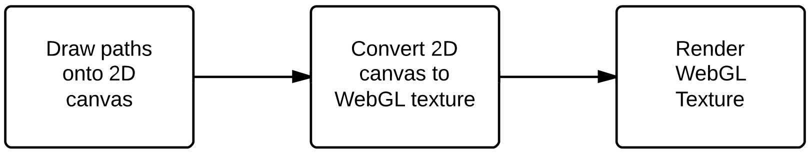


Figure 5.7: Process of drawing lines and annotations onto a WebGL canvas using a 2D canvas

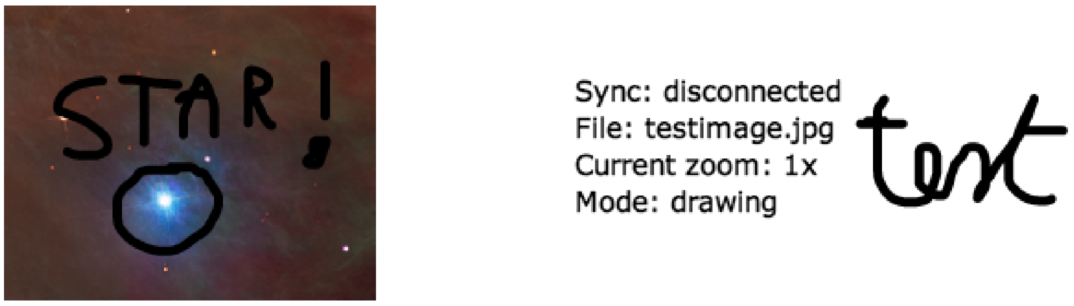


Figure 5.8: Example of two handwritten annotations. One on a white background and one on top of an image.

should then be synchronized between the clients. To synchronize the annotations, all clients connect to a WebSocket server and each WebGLAnnotation (Section B.2) is assigned a globally unique identifier. The WebSocket server is a simple broadcasting server, it forwards a received message to all connected clients except the sender. When a client needs to send out an update, it will serialize the annotation object and send it to the WebSocket server. The server will forward the message to all connected clients who will deserialize the message and update the annotation based on the points and the identifier in the deserialized annotation object.

5.3 Conclusion

While WebGL doesn't provide text-rendering out of the box, it can easily be added by combining WebGL with other web technologies. This chapter looked at text-rendering using a 2D canvas and Scalable Vector Graphics. Both techniques deliver good overall results but lack some of the capabilities that normal HTML text has like selection, searching, flow control, or advanced styling. Similar techniques can be used to draw handwritten lines to a WebGL canvas as well.

The results show no clear winner, SVG tends to give slightly better results in the tests but has worse performance. SVG also gives authors slightly more control over how text is rendered. Future versions of SVG are likely to include flow control. What technique is best depends on the application and platform. Older versions of Chrome contain a bug in the Cross-Origin Resource Sharing (CORS) implementation. Because of this bug which taints a canvas element when an SVG image is loaded, the SVG technique can not be used directly.

Chapter 6

Remote WebGL Rendering

WebGL relies heavily on the graphics stack of the system it is run on. Some systems, especially mobile devices, lack advanced graphics hardware. This can result in poor WebGL performance, low frame rates or a complete lack of WebGL support. This chapter describes a possible solution where the actual rendering is done by an external system and the resulting image is send back to the client. In this scenario, the system responsible for the rendering acts as a server and can handle multiple clients simultaneously.

The solution here describes a WebGL rendering server with multiple client support. The solution includes a JavaScript object that acts as the default WebGLRenderingContext[14] object but actually communicates with the rendering server to execute WebGL commands. The communication between client and server can be done using WebSockets[33] or WebRTC's RTCPeerConnection's DataChannels[2].

6.1 Emulating a WebGLRenderingContext

The WebGLRenderingContext object is only available in browsers with WebGL support. An instance can be created by calling the getContext method on any HTMLCanvasElement[34]. In browsers without WebGL support, this call will fail and you won't be able to obtain an instance of the WebGLRenderingContext object. Our solution extends the HTMLCanvasElement and the HTMLImageElement[34] with a *getRemoteContext* method that should be used to create an instance of the RemoteWebGLRenderingClient object. This object emulates the interface of the built-in WebGL context object.

6.1.1 RemoteWebGLRenderingContextClient

The RemoteWebGLRenderingContextClient object is the client part of the setup. When an instance is requested, this object will connect to a RemoteWebGLRenderingServer instance based on the connection information passed to the *getRemoteContext* method. The RemoteWebGLRenderingServer object runs on the device acting as the server and listens for incoming connections (either via WebSockets or PeerConnections) and creates a RemoteWebGLRenderingContextServer for each client.

6.1.2 RemoteWebGLRenderingContextServer

A RemoteWebGLRenderingContextServer instance only handles request from one client but multiple instances can be run simultaneously. This object receives serialized RemoteWebGLCall objects from the client. These RemoteWebGLCall objects are deserialized and executed on a real WebGLRenderingContext. When necessary, the server sends the state (the image data in the framebuffer) of the context back to the client. This image data can be send as a raw bitmap, as a BASE64 encoded JPEG, or as a BASE64 encoded PNG. For performance reasons, support for the raw bitmap has been dropped, the PNG format can send the pixeldata in a lossless manner to the client and requires less bandwidth.

6.1.3 RemoteWebGLCall

The RemoteWebGLCall object is used to represent WebGL calls that need to be send of the network. A RemoteWebGLCall can have (up to 128) parameters. These parameters can be one of the ECMAScript[16] primitive types (except Undefined) or any[1] other type encapsulated in a RemoteWebGLObject. A RemoteWebGLCall can have an optional return value. The return value should always be a RemoteWebGLObject.

6.1.4 RemoteWebGLObject

A RemoteWebGLObject should be used to store objects (all types except the ECMAScript primitives). The object can be serialized, send over the network and be deserialized again. The object can contain data (set with *setData*) or it can be associated with an object (using *linkObject*). Data contained in the object is serialized together with the object and send over the network while associated objects are not. This distinction is needed because some data has to be send over the network (e.g. data to initialize WebGLBuffers) while WebGL* objects cannot be send over the network.

6.1.5 Object referencing

The solution never creates an instance of the built-in WebGLRenderingContext object on the client. Since other WebGL objects (e.g. WebGLTexture) can only be created through a WebGLRenderingContext, another way of referencing objects is needed. From a JavaScript perspective, these WebGL objects are empty which means they cannot be send over the network and still be uniquely identified.

To overcome this problem, this solution replaces all these objects with instances of the RemoteWebGLObject object. Each RemoteWebGLObject has a random, unique identifier.

The working of this system is explained in 6.1. When a client attempts to create a WebGLTexture, the RemoteWebGLRenderingContextClient will actually create a RemoteWebGLObject (1.1), forward the call to the server using a RemoteWebGLCall object (1.2) and return the RemoteWebGLObject. The return object of the RemoteWebGLCall is set to the newly created RemoteWebGLObject.

During serialization, only the identifier of this object is send to the server (1.2) which will create a RemoteWebGLObject (1.3) with the same unique identifier and create a real WebGLTexture (the server can do this because it holds a real WebGLRenderingContext instance). The server will then associate this WebGLTexture with the RemoteWebGLObject (1.3). When the client wants the reference this texture in the future to execute a bindTexture call (2.1), it can use the RemoteWebGLObject it obtained when trying to create the texture (2.1). Because of the identifier, the server knows which WebGL object it needs to use (2.3).

[1]Some types may require additional serialization methods

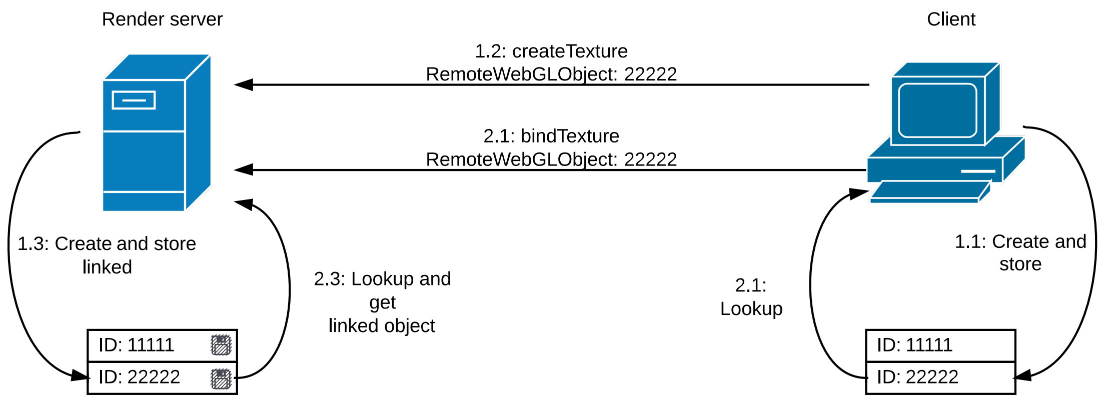

Render server
Client
1.2: createTexture RemoteWebGLObject: 22222
2.1: bindTexture RemoteWebGLObject: 22222
1.3: Create and store linked
2.3: Lookup and get linked object
2.1: Lookup
1.1: Create and store
ID: 11111
ID: 22222
ID: 11111
ID: 22222

Figure 6.1: Object referencing

6.1.6 Replacement element

The RemoteWebGLRenderingContextClient can use the default HTMLCanvasElement backed by a 2D rendering context to display the image data it receives from the server. Alternatively, the current solution can also use a HTMLImageElement to display the image data.

6.2 Communication

The network link between client and server handles traffic that normally only occurs between the CPU and GPU internally. The latency and available bandwidth on this link will heavily influence the frame-rate of the system because every frame has to be send from the server to the client. To avoid link congestion, the client throttles expensive calls based on how many frames are already in transit. To facilitate this throttling, client and server have to exchange information about their state. This requires an additional connection, otherwise the control packets would be delayed by the much larger and heavier data packets (e.g. pixel data). When a client connects to the server, it will actually open two connections to the server: one for data and one for control packets.

6.2.1 Connection setup

There currently are two standardized methods to facilitate bi-directional communication between a server and a client. The client here, being a web browser, does not allow the creation of a regular socket like you would find in other more low-level languages. To allow bi-directional communication, modern browsers expose the WebSocket API and the RTCPeerConnection API. A WebSocket can only connect to a WebSocket server while a RTCPeerConnection can connect to another web browser directly. Figure 6.2 shows how connections are setup between the clients and the server. Two client; client A and client B are connected to the same render server. Client A connects using only WebSockets, client B connects using both technologies. The control connection always runs over a WebSocket because it needs to be reliable and stable.

The WebSocket server in this setup acts a dumb proxy that only channels packets sort of like a switch. Ideally this server would run on the system responsible for rendering to avoid having additional network delays.

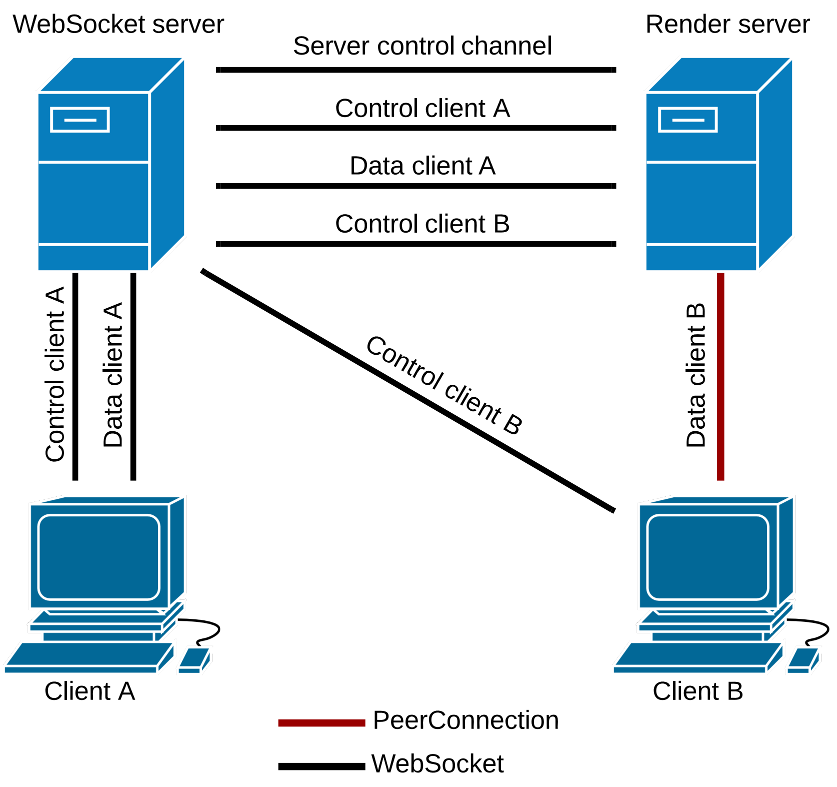


Figure 6.2: Connection setup

RTCPeerConnection

The RTCPeerConnection interface is part of the WebRTC specification and can be used to facilitate communication. A RTCPeerConnection sets up a direct, real-time communication channel between client and server. The current solution relies on PeerJS[72], a JavaScript library to abstract implementation details and browser quirks. The PeerJS library includes a server that is used to identify and setup the direct communication channels. Only the data connection can make use of a RTCPeerConnection. Only image data is send over the data connection.

WebSocket

A WebSocket connection is an alternative way of communication but requires all data to pass through a WebSocket server because web browsers only support the 'client' part of the system. The control connection always uses a WebSocket because it is reliable. The serialized RemoteWebGLCall objects are send over the control connection. Client and server will also use this connection the exchange information about their state.

6.2.2 Connection usage

The communication between client and server is handled by two separate connections; a data connection and a control connection. Figure 6.3 visualizes this concept, the control connection is black and the data connection is red.

- The control connection handles all RemoteWebGLCall objects except the calls containing image data because their size would block other much smaller objects and delay the entire system. Data is send in both directions. This connection also handles state information like performance numbers that are used to throttle the system to avoid congestion of the data connection.
- The data connection only handles BASE64 encoded image data. Frames containing image data are larger than other frames and take longer to arrive at the client. When the data connection is busy sending image data, calls and data for the next frame can be send over the control connection. This setup removes a bottleneck in the communication layer of the system and results in more optimal use of the server because it can start work on the next frame while the result of the previous frame might still be traveling on the network.

6.2.3 Framing

RemoteWebGLCall objects are serialized to byte streams (UInt8Array) before being send over the network. Binary serialization is preferred over JSON because it can represent the same data with less bytes and can be (de)serialized faster. These byte streams created by the custom WebGL objects conform to the frame design explained in this section.

RemoteWebGLCall

A complete RemoteWebGLCall frame is visualized in figure 6.4 and contains the following values:

- **Timestamp**: The timestamp of the system at the time of serialization. Since serialization is done just before passing the frame to the socket, the timestamp can be used to determine

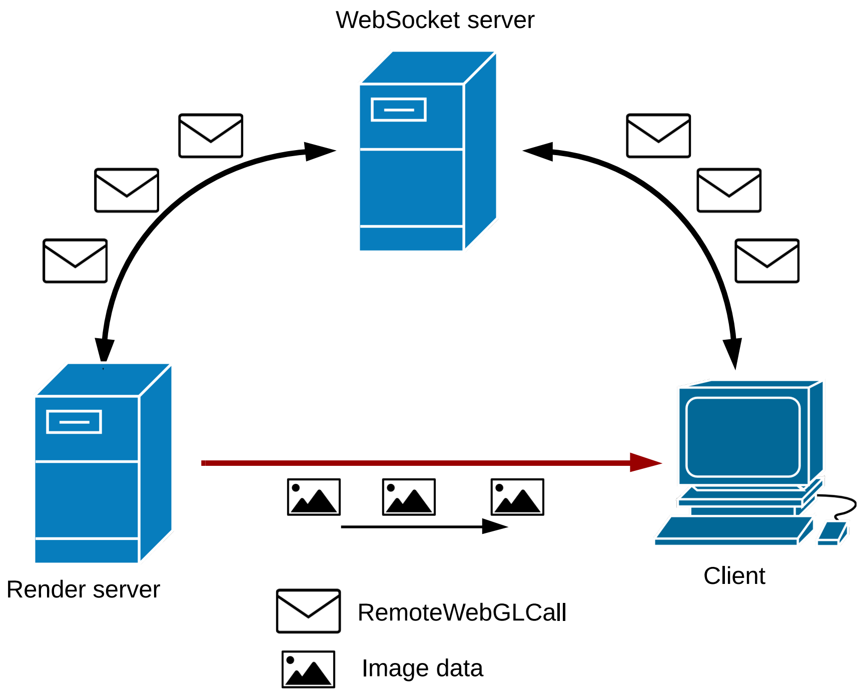


Figure 6.3: Message exchange in the Remote WebGL Rendering solution. The client sends RemoteWebGLCalls the the server. The server responds with new frames, transported as BASE64 encoded images.

Timestamp	**Length**	**Type**	**Return**	**# parameters**
64 bits	32 bits	8 bits	1 bit	7 bits

[Parameters]	**Return object**
Variable	Variable

Figure 6.4: A RemoteWebGLCall frame

how long it took the frame to be send. The timestamp is in microseconds and only used for benchmarking. The timestamp has a accuracy of 1 millisecond and resulting timing data is only useful when both the client and server are running on the same device.

- **Length**: A 32 bit unsigned integer representing the total length of the frame excluding the sequence.
- **Type**: A 8 bit unsigned integer representing the type of the call. Each function defined in WebGLRenderingContext has its own identifier. In addition to the default WebGL calls, extra calls can be represented by a RemoteWebGLCall to facilitate calls from the server back to the client (e.g. Returning pixel data).
- **Return**: A boolean (1 bit) value which is set to true (1) when the RemoteWebGLCall contains a return object. This return object is an instance of RemoteWebGLCall.
- **Number of parameters**: A 7 bit unsigned integer representing the number of parameters in the RemoteWebGLObject.
- **Parameters (Optional)**: A RemoteWebGLCall can contain a maximum of 128 parameters. A parameter can be a ECMAScript primitive or a RemoteWebGLObject. The primitives are preferred as they can be represented with fewer bytes. Detailed information on the different parameter types and the bytes needed to represent them can be found in Table 6.1 on page 61. Note that every parameter requires an additional 8 bits to represent its type.
- **Return object (Optional)**: A serialized RemoteWebGLObject.

Type	Length (+ 8 bits)	Notes
Null	0 bits	Only the parameter type is used to represent a Null value
Number	64 bits	An ECMAScript number is represented by a double-precision 64-bit binary format IEEE 754 value
String	(16 bits * strlen(String))	Each character in an ECMAScript String is represented by a 16 bit unsigned integer
Boolean	8 bits	The smallest component of an UInt8Array
RemoteWebGLObject	Variable	Framing of a RemoteWebGLObject is explained in more detail later

Table 6.1: Supported parameter types and their size. The sizes are based on the internal ECMAScript representations and padded to fit in 8 bits.

RemoteWebGLObject

A complete RemoteWebGLObject frame is visualized in figure 6.5 and contains the following values:

- **Length**: A 32 bit unsigned integer representing the full frame length.
- **Identifier**: A 64 bit double-precision float used to uniquely identify this object.
- **Type**: A 8 bit unsigned integer representing the type of data contained in the data field
- **Data (Optional)**: The data set using setData() which is send over the network. Currently only Float32Array and UInt8Array objects are supported.

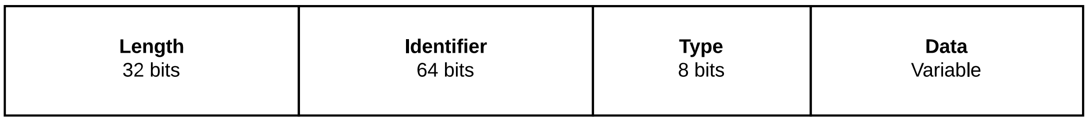


Figure 6.5: A RemoteWebGLObject frame

6.3 Performance

The major components of the system are rendering, image encoding, image decoding, frame serialization, frame deserialization, latency, and bandwidth. The current solution contains code to actively monitor the performance of the major components and will use this data to adjust itself to give the best result to the end-user. Table 6.2 contains all timing information that is currently being collected.

Name	Description
Image deserialization	The time required to deserialize RemoteWebGLCall objects containing image data
Data connection delay	The time it took a frame containing image data to arrive at the client
JPEG data size	The size of the frames containing JPEG encoded image data
PNG data size	The size of the frames containing PNG encoded image data
JPEG encoding	The time it takes to encode the canvas to a JPEG image
PNG encoding	The time it takes to encode the canvas to a PNG image

Table 6.2: Recorded timing information during the benchmarks.

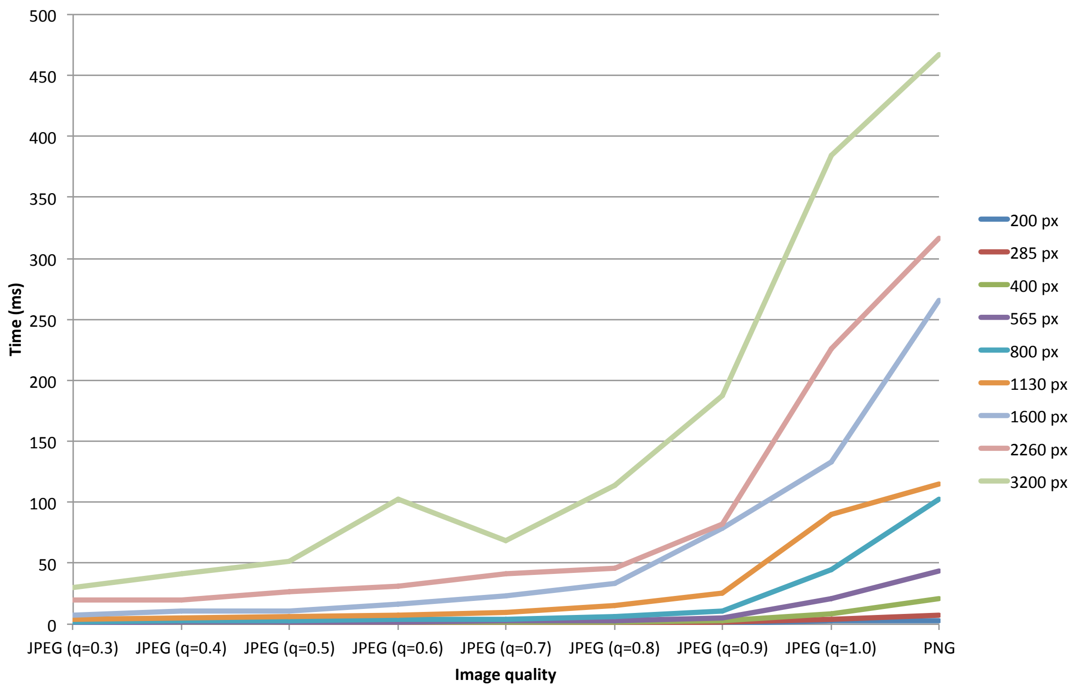
6.3.1 Throttling

The implementation takes away most work from the clients and moves it to the server. Because the client now has almost no work to do, it can request frames at a very high rate. The server and data connection however, cannot process requests at the same high rate. The client will have to throttle expensive calls as to not overload other components of the system. To do this, the client will never request a new frame when another request has not been fulfilled. It will wait for the previous frame to arrive before requesting the next frame.

6.3.2 Image quality

The quality of an image has a major impact on both the time it takes to decode and encode and the resulting file size as can be seen in figure 6.6. The preferred lossless PNG format results in larger packets which take longer to (de)serialize and send over the network. The lossy JPEG format results in smaller frames. The JPEG encoder takes a quality parameter that can be used to further speed up the system but since we need lossless imagery, we are in theory bound to the slower PNG solution.

The implementation uses both formats; when the clients is interacting with the system and needs frames fast, the system falls back to the JPEG format. The user is unlikely to notice the drop in quality because the image is moving. When the user stops interacting, the system automatically sends the current canvas state in PNG format to the client. This makes the system 'smooth' when the user needs it to and it makes the system lossless when it needs to.



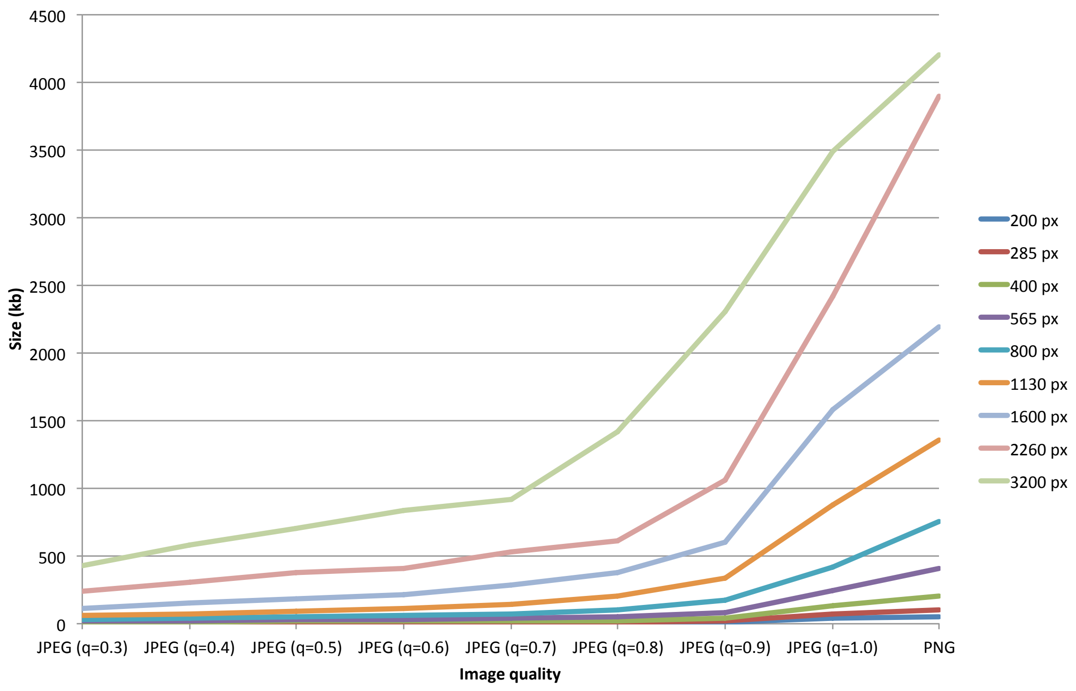


Figure 6.6: Image deserialization time and file size. The graphs show a steady increase in both encoding time and file size. When the JPEG quality is 1.0, both graphs shoot up. The size and encoding time for PNG images are the worst.

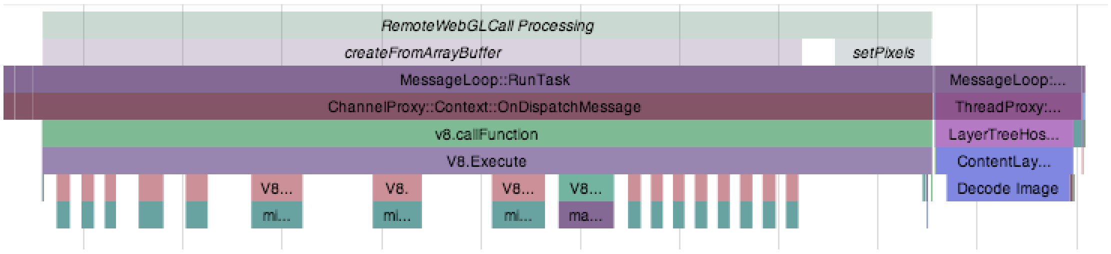


Figure 6.7: Processing of one RemoteWebGLCall with image data (client). The topmost bar shows covers the full processing of a RemoteWebGLCall. The createFromArrayBuffer bar shows that most time is spend deserializing the frame. The binary-to-text conversion appears to be very slow. It also shows a lot of garbage collector pauses.

To further optimize performance, client and server exchange performance metrics and the system uses this data to increase or decrease the JPEG quality setting. When the system becomes slow, the quality will be decreased to speed things up. The current implementation aims for at least 15 frames per second. When it drops below this threshold, the quality decreases. The resulting PNG image will always be lossless. This metric requires some tweaking based on what devices need to be supported, available bandwidth, and other requirements.

Figure 6.6 also reveals a considerable jump in both time needed to deserialize and file size at the highest JPEG quality. It is best to avoid using JPEG at the highest quality, the extra resource usage does not match the gain in quality.

6.3.3 Function call analysis

To further optimize the code it is necessary to know how it is behaving and where the CPU is spending most of its time. Figure 6.7 is a screenshot of a trace recording using the internal tracing functionality. It shows which function are being called and how much time they take up. The length of top bar is the total time it takes to process the RemoteWebGLCall. It immediately becomes clear *createFromArrayBuffer* takes up most of the time. This function handles the deserialization of the RemoteWebGLCall object. Further investigation revealed that the binary to string conversion for the BASE64 encoded image takes up all this time.

The same trace also reveals a lot of garbage collection work. Remember from previous chapters that the garbage collector in V8 uses a *stop-the-world* tactic; when the garbage collector is working, our code is paused. Take all the smaller chunks together and it adds up. The pause times also increase as the length of the processed string increases until a full garbage collection is performed. This clearly is an area where the code can be improved.

A Closer Look at String Deserialization

The original algorithm for converting the binary serialized string to a normal string representation is very slow because it is fighting with the garbage collector. It aggressively allocates memory for new string objects that need to be cleaned up. The garbage collection pauses make the algorithm slow. We devise two new algorithms that could potentially improve on this aggressively memory allocation and compare them to the original.

The testcase, binarytostring.html, contains three algorithms to convert serialized strings back to their original representation. Each algorithm was run on strings of increasing lengths, starting with strings containing 50000 characters up to strings containing 3200000 characters. These sizes correspond to BASE64 encoded PNG images from 200px by 200px to 2200px by 2200px.

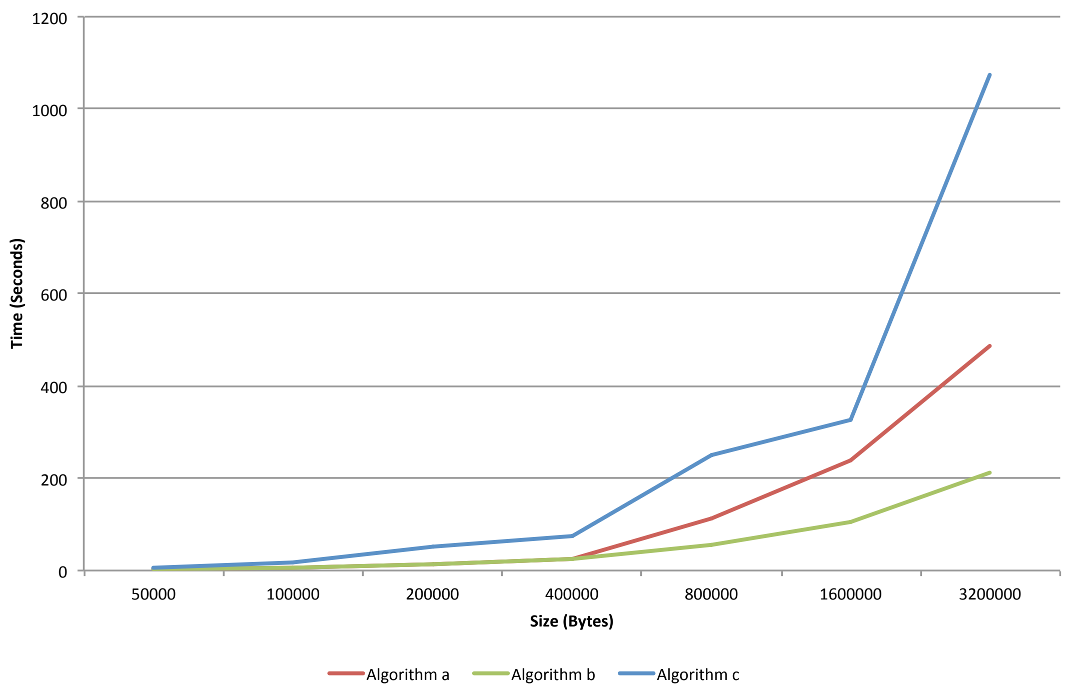


Figure 6.8: Timing results for the two fastest binary-to-string conversion algorithms. Algorithm a (original) starts out as the fastest but as the string length increases, algorithm b becomes faster.

Each algorithm was run 1000 times for each string length. The results or displayed on the graph in figure 6.8. *Algorithm a* is the original algorithm, *algorithm b* uses a technique that allocates memory less aggressively by batching the conversion together in shards of 65000[2] characters. *Algorithm a* creates a new string objects for each character (strings are immutable), *algorithm b* for every 65000 characters. This way, the garbage collector has a lot less work to do which results in less and shorter GC pauses.

Algorithm c converted each character to a string, stored them in an array and joined them together in one go using the *Array.join* method. This first creates a string object for each character which slows it down.

Algorithm b is the clear winner, for the largest tested string, it is twice as fast as the original algorithm. *Algorithm a* was slightly faster for the smaller strings (below 400000 characters).

Triggering the Optimization Compiler

The V8 JavaScript engine tries to optimize hot code (see section 4.2.3). To confirm that the code is being optimized, start Chrome with the *–js-flags="–trace-opt –trace-deopt"* command line flag. This will output below optimization and de-optimization information to the console. The output confirms that all the functions are being optimized because they are *hot and stable*. (Note: *shiftUint16* wraps *getUint16*)

```
[marking 0x467b02e1 <JS Function charCodeAt (SharedFunctionInfo 0
    x4331a8ad)> for recompilation, reason: hot and stable, ICs with
    typeinfo: 1/4 (25%)]
```

[2] to avoid exceeding the maximum call stack size

```
[optimizing 0x467b02e1 <JS Function charCodeAt (SharedFunctionInfo 0
   x4331a8ad)> - took 0.060, 0.234, 0.059 ms]
[marking 0x249cb215 <JS Function window.BinarySerializerHelper.
   shiftUint16 (SharedFunctionInfo 0x2493bfb1)> for recompilation,
   reason: small function, ICs with typeinfo: 6/6 (100%)]
[marking 0x467afa89 <JS Function getUint16 (SharedFunctionInfo 0
   x4332bc85)> for recompilation, reason: hot and stable, ICs with
   typeinfo: 4/6 (66%)] [optimizing 0x249cb215 <JS Function window.
   BinarySerializerHelper.shiftUint16 (SharedFunctionInfo 0x2493bfb1
   )> - took 0.038, 0.091, 0.030 ms]
[optimizing 0x467afa89 <JS Function getUint16 (SharedFunctionInfo 0
   x4332bc85)> - took 0.060, 0.159, 0.059 ms]
```

6.4 Legacy Browser Support

The solution described up to this points relies on new technologies like WebSockets and WebRTC PeerConnections to transfer data between client and server. WebSockets has been around for a couple of years and has good support in modern web browsers, WebRTC on the other hand is very new and only supported in very recent versions of a limited set of web browsers. Since our solution relies on these technologies, it does not support older, legacy web browsers. In an attempt to improve browser support, we replaced or polyfilled unsupported technologies and managed to get support for Internet Explorer 8 and up.

6.4.1 Long Polling AJAX

The legacy version of the Remote WebGL Rendering solution replaces WebSockets and WebRTC PeerConnections with Long Polling AJAX between the client and the WebSocket server. The WebSocket server handles the conversion from a single Long Polling AJAX call to the control and data WebSocket connections between WebSocket server and the render server.

Figure 6.9 shows the connection setup for client A using WebSockets and client B using the new legacy method. The connection setup for client A remains the same, it connects to the WebSocket server which still acts as a dumb proxy. The connection setup for the legacy client B is very different; the client buffers all RemoteWebGLCalls until it needs to send a drawArrays call at which point it will flush the buffer by serializing all buffered calls and sending them to the WebSocket server as part of an AJAX request. The WebSocket server forwards these commands to the actual render server and keeps the AJAX request alive while it waits for the image data from the render server. When it receives this image data, it will send it to legacy client B as part of the response body. This process is repeated the next time client B needs a new frame; each AJAX request can only be used once.

6.4.2 JavaScript Typed Arrays

WebGL uses special typed arrays (e.g. *UInt8Array*, *Float32Array*, ...) to initialize buffers. These typed arrays were recently added to JavaScript for WebGL and are not supported in older browsers. Instead of replacing the use of these types, we used a simple polyfill the simulate support for these types. A polyfill is a piece of code that provides support for technologies when there is no native support for them yet by emulating its behavior. A polyfill is generally slower than a native implementation but since the legacy version does not rely heavily on these types that should not be a problem.

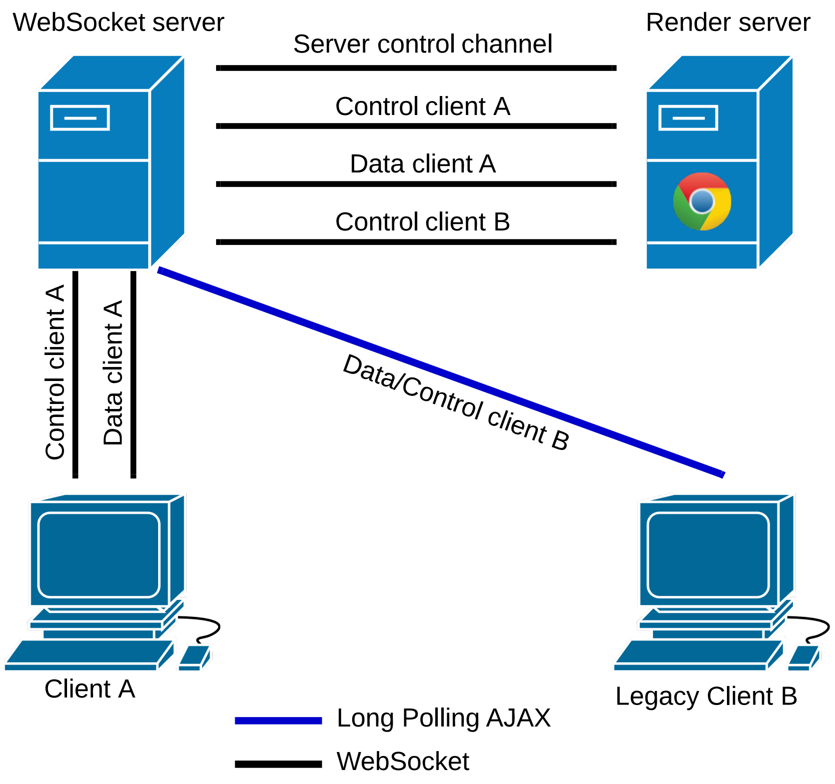


Figure 6.9: Connection setup with legacy support. Legacy client B communicates with the WebSocket server using long polling. The WebSocket server translates the long polling requests from the client to the two WebSockets connections between the servers. Client A uses the normal connection setup.

6.4.3 Data-URI limitations in Internet Explorer 8

Internet Explorer 8 limits the length of data-URI's to 32kb. Tests proved that it is not possible to store an images larger than 100px by 100px (roughly) in a 32kb data-URI. These dimensions are to small to have a usable canvas. A tilled approach were to image is divided up into smaller images, each represented as a different data-URI can be used to overcome this limitation. The disadvantage of such a solution is that not all the images that form a full frame are decoded and displayed at the same time, this will cause tearing. Another solution would be to have the WebSocket server decode the images, store it locally, and return a URL to the full image instead of the actual image data. The URL can be set as the source of an image element.

6.5 Canvas Video Streaming

Up to this point our solution only used images to communicate the canvas state back to the client. This meant that for each frame the server had to get the canvas state from the framebuffer, encode it to the JPEG or PNG formats, and serialize the result. The client had to deserialize the message, decode the image and display the result. This entire process consumes a lot of resources on both the client and the server. It would be better if these resources are used to actually process WebGL calls. The images also take up a lot of bandwidth.

To reduce overhead and unnecessary image operations, the solution was partially rewritten to support WebRTC. Instead of sending full images over the network, the new solution creates a WebRTC connection between client and server. This WebRTC connection is used to stream the state of the canvas to the client. This method does not only avoid costly image operations but also has built-in bandwidth management (Section 2.2). This eliminates the need for custom throttling code.

6.5.1 Connection Setup

The new solutions moves the rendering server from a Chrome tab to a native application. The native application uses a QWebPage[54] to process WebGL commands. Because the rendering server no longer runs in a Chrome tab, the rest of the setup can be simplified as well. The native applications also acts as a WebSocket server, this eliminates the need for an external, proxying WebSocket server; clients can connect directly to the WebSocket server inside the native application. The simplified connection setup is shown in figure 6.10.

The native application will request the current canvas state from the embedded QWebPage and use that image as the source for a video stream. This video stream is continuously streamed to the client using WebRTC's RTCPeerConnection (Data client A). The WebSocket connection (Control client A) is still used to reliably transport serialized RemoteWebGLCall objects. To avoid having three connections between client and server, the lossless PNG images are send over this connection as well. The lossless image are still required because the video stream, like the JPEG images, is lossy.

6.6 Render Server Scalability

During the development of the remote WebGL rendering solution, the performance has been analysed several times. This identified several performance issues for which fixes have been implemented. The analyses so far was mostly focused on the performance from the perspective of the client. One of the goals was to have one render server for multiple clients. Now it is time to look how well the render servers can handle multiple clients simultaneously.

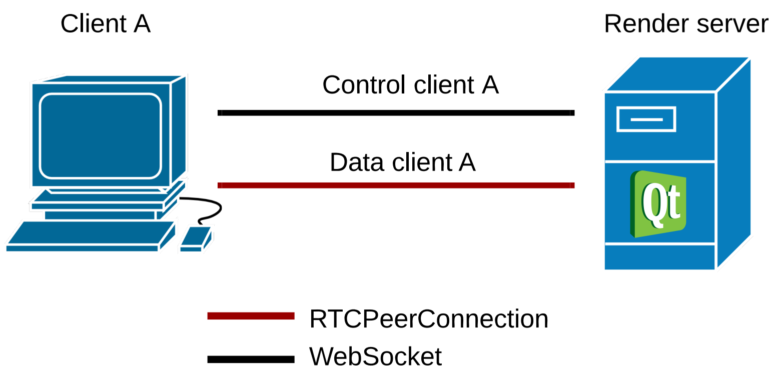


Figure 6.10: Connection setup for the Remote WebGL Rendering solution with WebRTC streaming. The central WebSocket server has been eliminated and built into the native rendering server.

To measure how well the render servers scale, we will run benchmarks with canvasses of varying sizes and with multiple clients connected simultaneously. The client code has been modified slightly so that it continuously request new frames. The throttling code remains unchanged; a client can only have one draw request open at any given time. This scenario gives the worst-case scenario for the render server, the results in the real-world will always be better.

The clients and server will run on separate systems and since we want to focus on the server, the clients will not decode and display the image data they receive from the server. The system running the clients could get congested with image decodes slowing down the system and skewing the test results.

6.6.1 Test Conditions

The render servers will run on a MacBook Pro and they will use the NVIDIA GeForce GT 650M with 512MB of video memory. Other specifications of the system:

- **CPU:** 2,3 GHz Intel Core i7
- **RAM:** 16 GB 1600 MHz DDR3
- **OS:** OS X 10.8.5 (12F37)

The server was directly connected to the system running the clients via a 1000BASE-T (Gigabit) connection. The client system has a 2.4 Ghz Intel Core i7 CPU, 4GB of 1333 MHz DDR3 memory, and an Intel HD-video 3000 GPU using 384MB of shared memory as video memory. The system runs MAC OSX Lion 10.7.5. (11G63b).

The different test resolutions are shown in table 6.3. The resolutions are based on real device resolutions. The test includes resolutions for a variety of devices, ranging from the resolution of a non-retina smartphone screen to that of a 15” retina laptop screen. Table 6.3 shows how the different test resolutions map to real devices. During the test, we will scale up the number of connected clients gradually and monitor how this impacts the frame rates for the clients. The iPhone 5 resolution is not tested because it is very similar to the iPad 2 resolution.

6.6.2 Standard Version

The results of the benchmark with the original image based version are shown on the graph in figure 6.11. The biggest problem with the original version is that it runs in a single Chrome

Device Type	Resolution	Screen Pixels	Example
Mobile (Non-Retina)	480 x 320	153,600	iPhone 4
Mobile (Retina)	1136 x 640	727,040	iPhone 5
	1920 x 1080	2,073,600	Nexus 5
Tablet (Non-Retina)	1024 x 768	786,432	iPad 2
Tablet (Retina)	2048 x 1536	3,145,728	iPad 3
	2560 x 1600	4,096,000	Galaxy Tab Pro 10.1
Desktop (Non-Retina)	1440 x 900	1,296,000	15' Laptop
Desktop (Retina)	2880 x 1800	5,184,000	15' Retina Laptop

Table 6.3: The test resolutions used in the benchmark. The resolutions include different types of real devices, from a normal non-retina smartphone to a 15' retina laptop and everything in between.

tab. This means all clients share one renderer and block each other. Because of this, the server has difficulty handling multiple clients. The server also has difficulty handling large canvasses because of the expensive image encodes. Once the server runs out of video memory, Chrome has to constantly re-upload textures which causes a huge drop in performance.

6.6.3 WebRTC Version

The WebRTC based version of the remote rendering server scales better than the original version. It benefits a lot from its multi-threaded design and can use the available resources of the server system (a quad-core) much more effectively. The results of the benchmark with the WebRTC version are shown on the graph in figure 6.12. The frame rate is capped to 30 frames per second because this is the upper limit of the WebRTC video stream. The server can generate frames faster in some cases but doesn't because the frames would be useless. Just like the original version, the WebRTC version has similar problems when the server runs out of video memory. During the benchmark it was also observed that the WebRTC server stops accepting new clients when it is under high load.

6.7 Conclusion

The remote WebGL rendering proof of concept described in this chapter extends the reach of WebGL to low performance devices and older web browsers. The solution emulates WebGL support via a RPC like system: WebGL calls are serialized and send to a central rendering server. The rendering server does the actual WebGL rendering and returns the result to the client. The initial versions performed poorly but throughout development the performance improved gradually. Supported browsers include old browsers like Internet Explorer 8 and mobile browsers on Android and iOS.

The proof of concept includes a version with adaptive image streaming (JPEG and PNG) and a WebRTC based video streaming alternative. The image based variant runs a rendering server in Chrome while the WebRTC variant uses a small Qt application to support WebRTC streaming. The client-side shows good performance, the current bottlenecks appear to be on the server side. The original image based version suffers from its single threaded nature. The clients block each other which drastically limits its scalability. The WebRTC version is multi-threaded which results in better scalability. Both version suffer hugely from video memory starvation.

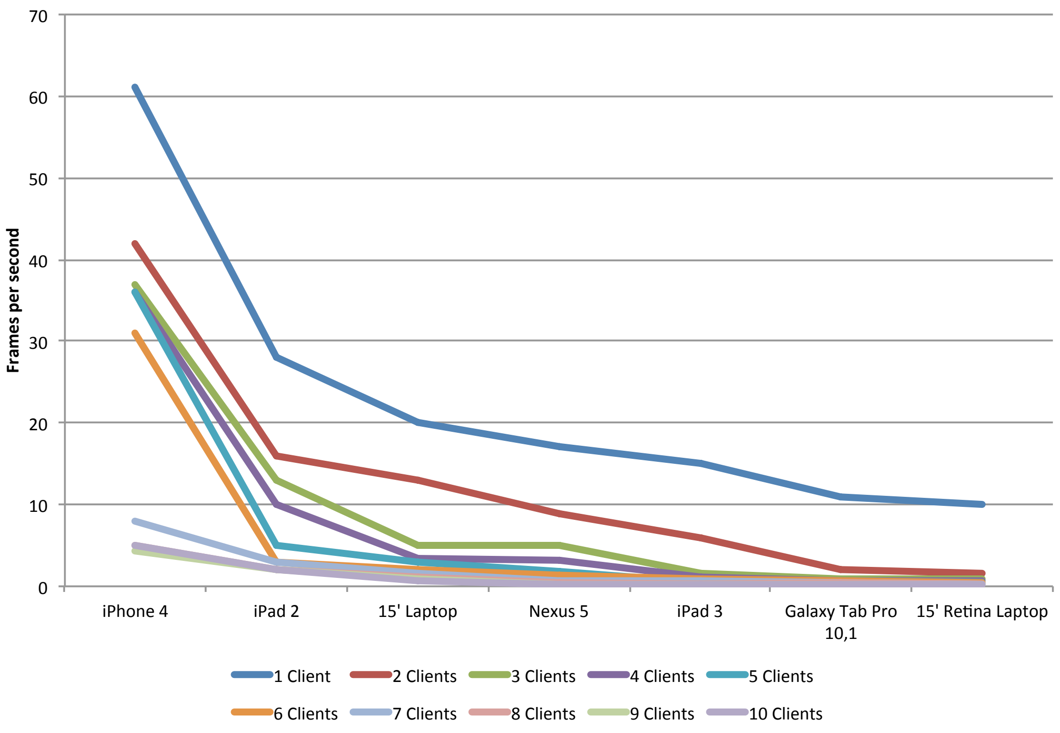


Figure 6.11: Results of the server scalability tests for the original (image based) version.

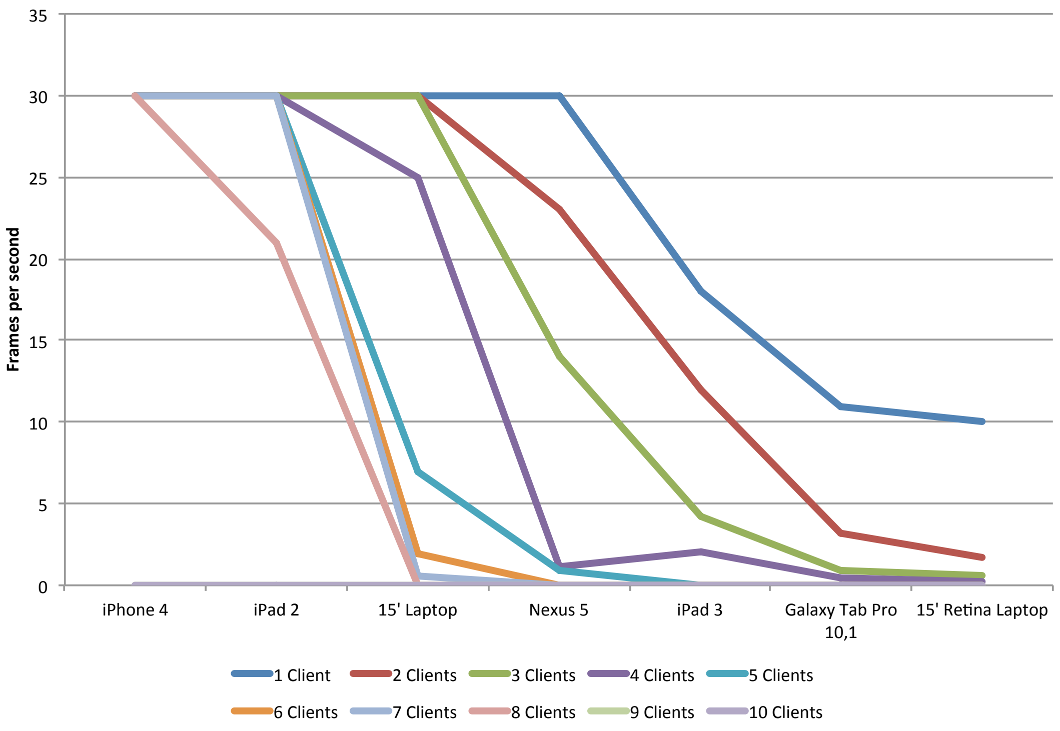


Figure 6.12: Results of the server scalability tests for the WebRTC based version. The video stream has a maximum frame-rate of 30fps.

Chapter 7

Conclusions and Future Work

The goal of this thesis was to investigate the state of the web as an application platform and to do this we defined two research questions in the introduction. Chapters 2, 3, 4, and 5 focused on answering the first research questions. Chapter 6 answered the second research question by proposing, implementing, and evaluating a remote WebGL solution that emulates WebGL support on low-end devices. In this chapter we will shortly summarize how the research questions were satisfied and what future technologies can further improve on our work.

7.1 Tools, Web Technologies, and Browser Features

1. What tools, technologies, or other browser features are available to web developers today, and more specifically:
   (a) how do these tools, technologies, or features help developers write high-performance applications for the web;
   (b) how do these tools, technologies, or features compare to those available on most native application platforms;
   (c) are there new issues caused by these tools, technologies, or features, what is the impact of these issues, and can they be resolved.

Chapter 2 started of with a short introduction into new technologies like WebSockets and WebRTC which provide bi-directional communication channels (sockets) to web applications. These technologies replaced long polling, the slower and less efficient, and at the time only alternative. WebRTC also facilitates peer-to-peer video and audio streaming turning the implementation of a video conferencing application into a trivial operation. WebRTC handles the complex things such as codec handling, bandwidth management, and encryption. The WebGL canvas API gives web applications access to a OpenGL like shader-based render pipeline on the GPU. This gives developers a tool to move complex and high-performance graphics application to the web.

In chapter 3 we looked at the architecture of the Chrome web browser itself. The multi-process architecture for example makes the browser fast, stable, and secure. Other techniques like rendering and compositing in the hardware path greatly improved the rendering performance of web applications. Without the hardware path, technologies like WebGL and hardware accelerated video decoding would be slow or not possible at all. In this chapter we also take an in-depth look at memory management in Chrome to analyse a problem introduced by one of these new technologies: the artificial limit of the number of WebGLRenderingContext and the slow garbage collection. Several possible solutions were proposed and evaluated and the solution which recycles previously used WebGLRenderingContext was deemed to be the best. The other solutions either required changes to Chrome's source-code or the use of command line flags making them impractical.

The built-in Developer Tools are evaluated in chapter 4. The Developer Tools are a collection of tools that all aim to help developers write better and faster code. There are analysis tools for to evaluate CPU usage, memory usage, and allocations. The Developer Tools also contains a full featured JavaScript debugger with breakpoints and watches. We found these tool be more user-friendly than those available on most native platforms. Chapter 4 also investigates the poor performance of *getImageData* calls on an accelerated canvas. This turned out to be caused be the new hardware path where to bitmap is stored in video memory rather than main memory. This means a slow *glReadPixels* call is needed to get the actual canvas content and transfer it over the slow GPU $\leftrightarrow$ CPU bus. This is a disadvantage of one of the new techniques and unfortunately not an easy fix.

Finally, chapter 5 described two solutions to extend WebGL with support for text rendering and handwritten lines (annotations). The first solution combined WebGL with the 2D canvas API (which already has text and path rendering support), the second solution used SVG together with the 2D canvas API to get the same result.

We can conclude that the web has become a stable application platform with loads of new features, tools, and browser features that enable the porting of complex applications. Unfortunately, some of these improvements also introduced new problems that could not always be resolved. The new technologies we looked at are only a small subset of all the new technologies over the last years[65].

7.2 Remote WebGL Rendering

2. Is it possible to run (high-performance) WebGL applications on low-end devices by emulating WebGL support. The emulation has to be:

    (a) transparent to the user;

    (b) perform well (fast enough to make it usable);

    (c) scalable.

The second research question was answered in chapter 6 where the **feasibility study** resulted in a working proof of concept that emulates WebGL on low-end devices and in legacy browsers. The proof of concept proposes a drop-in replacement for the built-in WebGLRenderingContext. This replacement acts as a proxy, it connects to a render server and uses an *remote procedure call* (RPC) like system to execute WebGL commands on a remote WebGL canvas. The result is streamed back to the client in real-time, either via images or via a WebRTC videostream. The chapter provides performance figures for several aspect of the system, both on the client and the server side. The performance drops off as the canvas size increases, which was expected. The solution performs relatively well but the server could potentially be made more scalable.

7.3 Future work

The evolution of the web platform does not stop here, new web standards are still being developed and implemented daily. As closing notes we will talk about some of these upcoming standards and technologies that can be used to further improve the topics discussed in this thesis.

- **New multi-threaded server**: The current server implementations do not take full advantage of multi-core systems. Writing a more advanced rendering server, for example using Chrome's content module, will likely yield much better scalability.
- **ImageBitmap object**: The ImageBitmap object is currently part of the HTML specification and allows for asynchronous image decoding. The source for an *ImageBitmap* can be a HTMLImageElement, a HTMLVideoElement, a HTMLCanvasElement, a *Blob*, an ImageData object , a *CanvasRenderingContext2D* or another ImageBitmap. This object has not been implemented yet but gives web developers a lot more control over the decoding process for images. The interface is available in Web-Workers making multi-threaded image decoding possible as well.
- **Canvases in Web-Workers**: Support for proxying canvases to workers was recently added to the HTML specification. By calling *canvas::transferControlToProxy*, the rendering context of a canvas can be transferred to a Web-Worker. This makes it possible to build a multi-threaded render server in Chrome. Unfortunately, this is currently unavailable in Chrome.
- **Hardware accelerated JPEG decoding**: People at Opera Software are experimenting with moving the rasterization (decoding) of JPEG images to the GPU in Chrome[18]. The hardware accelerated decoding paths on modern graphics card will greatly improve the decoding times of JPEG images. This will result in speed-ups for the remote WebGL rendering solution.
- **WebGL Texture sharing and compression**: The Khronos group proposed a new API that would allow texture sharing between WebGL rendering contexts[69]. This would save considerable amounts of memory in the remote WebGL rendering, currently the same data is often stored multiple times. Decreasing video memory usage will make the server more scalable.
- **Encoding API**: A new encoding specification has been written that defines the *TextDecoder* and *TextEncoder* objects[5]. These objects provide encoding and decoding support for converting strings from one representation to another. These new objects can replace the custom string deserialization code. These object are only supported in Firefox at the moment.

Nederlandse Samenvatting

Sinds de originele introductie van het internet eind twintigste eeuw is het web enorm gegroeid: van een platform voor het delen van voornamelijk (gelinkte) statische content met weinig interactie mogelijkheden naar een platform dat complexe applicaties ondersteunt. Deze transitie begon met eenvoudige applicaties zoals e-mail clients (Hotmail, Gmail) tot complexe applicaties zoals volledige game-engines (Unreal Engine 3[51]) en wetenschappelijke applicaties zoals Google's Quantum Computing Playground[21]. Dit werd mogelijk dankzij nieuwe technologie zoals WebGL en nieuwe JIT JavaScript Compilers die het web in compleet applicatie platform gemaakt hebben.

Context

Door de recente verbeteringen op vlak van functionaliteit en performantie op het web, is het een mooi moment om te kijken hoe het nu staat met al deze nieuwe technologie, zeker nu het web enorm aan populariteit wint als applicatie platform. De recente stijging in het mobiele (web) gebruik maakt van performantie een belangrijk aspect bij het ontwikkelen van web applicaties. De focus voor het eerste deel van het onderzoek ligt op Chrome. We hebben voor Chrome gekozen omdat Chrome een goede ondersteuning biedt voor nieuwe technologie (bv. WebRTC), de broncode volledige open-source is en er tijdens de ontwikkeling een grote focus was op snelheid.

Onderzoeksvragen

Het doel van deze thesis is onderzoeken in hoeverre het mogelijk is om complexe applicaties naar het web te porten en welke tools daarvoor beschikbaar zijn. Het onderzoek gebeurt door het ontwikkelen van een web versie van een bestaande native applicatie. De in-casu applicatie is een complexe applicatie waarbij bestaande gedetailleerde visualisaties weergegeven en gemanipuleerd kunnen worden. De data achter de visualisaties hebben een hoge resolutie en veel detail, de visualisaties moeten correct zijn (lossless) en moeten vanuit meerdere viewpoints bekeken kunnen worden. Voor dit onderzoek formuleren we volgende onderzoeksvragen:

1. Welke tools, technologieën en browser functionaliteit zijn er beschikbaar die web ontwikkelaars ondersteunen bij het ontwikkelen van web applicaties, hierbij kijken we specifiek naar
   (a) hoe deze tools, technologieën en functionaliteit ontwikkelaars helpen bij het schrijven van performante code;
   (b) hoe tools, technologieën en functionaliteit vergelijkbaar zijn met deze beschikbaar op native platformen;
   (c) of deze tools, technologieën en functionaliteit nieuwe problemen als gevolg hebben, hoe groot de eventuele impact hiervan is en of hiervoor oplossingen zijn.

2. Is het mogelijk om complexe WebGL applicaties te gebruiken op low-end toestellen door WebGL te simuleren. Deze simulatie moet

   (a) transparant werken vanuit het oogpunt van de gebruiker;

   (b) performant genoeg zijn zodat de applicatie werkbaar is;

   (c) schaalbaar zijn.

Aanpak (type masterproef)

De eerste onderzoeksvragen proberen we te beantwoorden aan de hand van een **gevalsanalyse** of **case-study** van de beschikbare tools en technologieën. Concreet kijken we naar de technologie binnen Chrome die zorgt voor verbeterde snelheid en naar nieuwe technologieën die nieuwe features introduceerden of tragere alternatieven vervingen. We doen ook een uitgebreide analyse van twee opmerkelijke nadelen van deze nieuwe features die naar boven kwamen tijdens de ontwikkeling van de eerder omschreven applicatie. De tweede onderzoeksvraag beantwoorden aan de hand van een **haalbaarheidsstudie** waarin we een remote WebGL rendering proof of concept beschrijven en implementeren.

Introductie recente web technologie

In dit onderdeel beschrijven we kort enkele recent ontwikkelde web technologieën. We focussen ons op technologie die later van pas zal komen bij het verdere onderzoek; deze introductie dient als achtergrond voor de rest het document. Dit onderdeel beantwoordt ook reeds deels de eerste onderzoeksvraag.

WebSockets

WebSockets is een technologie die is ontwikkeld ter vervanging van de long-polling techniek die gebruik maakt van AJAX om zo bi-directionele communicatie te faciliteren. Deze techniek was niet gebruiksvriendelijk en niet efficiënt omdat de AJAX hiervoor niet ontwikkeld is. WebSockets geeft web applicaties toegang tot een API waarmee ze een permanente bi-directionele verbinding kunnen opzetten tussen een server en een client. De API is vergelijkbaar met een socket API zoals die gevonden kunnen worden in andere talen. De handshake vertrekt vanuit HTTP request.

WebRTC

WebRTC biedt net zoals WebSockets de mogelijkheid aan web applicaties om een bi-directionele verbinding op te zetten maar dit keer direct tussen twee clients op basis van SRTP of SCTP[55, 46]. Het hoofddoel van WebRTC is het aanbieden van peer-to-peer video en audio applicaties maar kan ook gebruikt worden voor andere types data. Voor peer-to-peer video en audio biedt WebRTC automatische afhandeling voor het coderen en decoderen van video en audio, beheer van de bandbreedte en synchronisatie. Ondanks de directe peer-to-peer communicatie is er nog steeds een centrale server nodig die ondersteuning biedt voor het opzetten van de verbinding (om firewalls en NAT te omzeilen) aan de hand van protocollen zoals ICE, STUN en TURN[38].

Canvas

Het HTML canvas element bestaat uit twee API's: de eenvoudige 2D canvas API is beperkt tot een aantal basisbewerkingen, de krachtige WebGL API geeft web applicaties toegang tot de GPU. De WebGL API is op OpenGL gebaseerd en maakt net zoals OpenGL gebruik van shaders geschreven in GLSL[68]. Met WebGL is het mogelijk om complexe en krachtige 3D animaties te bouwen op het web.

De architectuur van Chrome

In dit onderdeel wordt de architectuur van Chrome bekeken, voornamelijk hoe bepaalde keuzes hierin de performantie van zowel de browser als de web applicaties verbeteren. Er wordt gekeken naar de 3 belangrijkste onderdelen, de multi-process architectuur en de grafische pipeline. Het laatste deel bestaat uit een uitgebreide analyse van het geheugenbeheer in de browser.

Modules

Chrome is opgebouwd uit verschillende modules, de belangrijkste zijn de render engine Blink, de JavaScript engine V8 en Chrome zelf. Blink is verantwoordelijk voor het parsen van de HTML en CSS op een webpagina. Blink start met het opbouwen van de *DOM Tree* door de HTML te parsen in het *DOM construction proces*, tegelijk wordt ook de CSS verwerkt. De DOM tree en de CSS regels worden dan samengevoegd in de *render tree*, deze *render tree* bevat hierna alle elementen die zichtbaar zijn op de pagina. De *render tree* dient als basis voor het opbouwen van de pagina layout zodat de pagina uiteindelijk op het scherm getekend kan worden. Dit proces is gevisualiseerd in figuur 3.1.

De V8 JavaScript engine is het tweede belangrijke onderdeel. De JavaScript engine leest alle JavaScript code op een pagina, compileert de code om deze later uit te voeren. De communicatie tussen Blink en V8 verloopt via de *V8 bindings* laag zodat de structuur van de pagina (de DOM tree) vanuit JavaScript gemanipuleerd kan worden. Chrome zelf biedt als laatste onderdeel functionaliteit voor de andere modules, een volledige custom UI toolkit en andere Chrome specifieke functies zoals favorieten en de download manager.

Multi-process architectuur

In tegenstelling tot de meeste web browsers gebruikt Chrome een multi-process architectuur. In deze architectuur krijgt iedere pagina net zoals elke plugin een eigen proces waarin Blink en V8 instanties draaien (render process). Dit verhoogt de veiligheid en stabiliteit van de browser omdat een vastgelopen tab geen invloed heeft op de rest van browser. Ook het aansturen van de grafische kaart verloopt vanuit een afgesloten proces. Om de impact van een mogelijk besmette pagina te beperken draaien alle processen in een restrictieve sandbox waarin er geen directe communicatie met besturingssysteem mogelijk is. De communicatie tussen de processen verloopt via *inter-process commnication* (IPC), dit zijn *named pipes* die beheerd worden door de browser.

Grafische pipeline

Er zijn twee gescheiden grafische pipelines beschikbaar in Chrome, het *software path* en het *hardware path.* Het software path voert alle grafische operaties uit op de processor, alle elementen op de pagina worden rechtstreeks in een bitmap getekend en naar de grafische kaart verstuurd.

Het hardware path gebruikt zowel de processor als de grafische kaart om grafische operaties uit te voeren, dit maakt technologieën zoals WebGL en het decoderen van video streams sneller. In plaats van rechtstreeks naar een bitmap te tekenen en deze naar de grafische kaart te sturen zal de browser grafische operaties naar het GPU proces sturen. Het GPU proces is een gescheiden proces dat de grafische kaart aanstuurt. Complexe operaties zoals tekenen op een WebGL canvas gebeurt veel sneller wanneer het wordt uitgevoerd door de grafische kaart.

Wanneer een pagina via het hardware path getekend wordt bestaat de pagina ook uit meerdere lagen. Deze lagen worden later door de grafische kaart samengevoegd. Dit heeft als voordeel dat de klassieke HTML elementen nog steeds door de processor getekend kunnen worden terwijl elementen met WebGL of video rechtstreeks door de grafische kaart afgehandeld kunnen worden. De communicatie tussen het render process en het GPU process gebeurt hier via een *GPU command buffer*. Blink schrijft commando's naar deze buffer en stuurt via IPC een bericht naar het GPU process wanneer de commando's in de buffer uitgevoerd moeten worden.

Geheugenbeheer

Door de integratie van Blink en V8 is het geheugenbeheer binnen Chrome een complex gebeuren. Objecten kunnen beheerd worden door Blink of V8, sommige objecten moeten echter vanuit beide modules beheerd kunnen worden. Het geheugen van beide modules moet dus onderling gedeeld worden, dit gebeurt via de eerder genoemde *V8 bindings layer* maar veroorzaakt soms problemen met het opruimen van WebGLRenderingContext objecten. Deze objecten worden opgeslagen in het geheugen dat bij Blink hoort, Blink gebruikt *reference counted garbage collection* waarbij objecten opgeruimd worden wanneer er geen actieve verwijzingen meer zijn. Men zou dus verwachten dat deze WebGLRenderingContext objecten ook opgeruimd worden op deze manier, dit blijkt niet het geval te zijn omdat deze objecten ook beheerd worden vanuit V8.

V8 gebruikt een *tracing garbage collector* om oude objecten op te ruimen, deze garbage collector ruimt objecten slechts op na een bepaald, variabel interval. Dit veroorzaakt problemen omdat objecten nog enige tijd actief blijven nadat alle verwijzingen naar het object opgeruimd zijn. De garbage collector in V8 gebruikt een *stop-the-world* tactiek waarbij alle operaties gepauzeerd worden, om deze pauzes te beperken werkt de garbage collector in V8 incrementeel. De garbage collector verwerkt in iedere stap slechts een deel van de volledige heap (het V8 geheugen). De V8 heap bestaat uit een aantal kleinere delen (zie figuur 3.6), elk deel bevat een specifiek type data zodat het verloop van de garbage collector verder geoptimaliseerd kan worden.

Een van deze optimalisaties is het gebruik van een dubbele garbage collector. De eerste garbage collector ("*scavange*") verwerkt alleen recent aangemaakte objecten in een deel van de heap ("*new space*"). De tweede gabarge collector verwerkt de rest van de heap, dit deel bevat objecten die langer bestaan. Dit deel wordt minder vaak verwerkt omdat het verwerken hiervan langer duurt. Tijdens elke volledige garbage collector ronde wordt voor elke object gecontroleerd of er nog actieve verwijzingen zijn, als blijkt dat er geen actieve verwijzingen meer zijn zal het object aan het eind van de ronde opgeruimd worden. De rest van de objecten blijven gewoon bestaan. Door de korte vertraging bij het opruimen van objecten binnen het V8 geheugen treden er soms problemen op voor WebGLRenderingContext objecten omdat het aantal hiervan expliciet beperkt is. Een mogelijk oplossing voor dit probleem is het recyclen van reeds actieve WebGLRenderingContext objecten in plaats van telkens een nieuw object aan te maken.

Performantie binnen Chrome

In het vorige deel werd de architectuur van Chrome toegelicht en hoe deze invloed had op de performantie. De focus binnen dit deel ligt op de Developer Tools in Chrome die door web ontwikkelaars gebruikt kunnen worden tijdens het ontwikkelen en op andere functies die specifiek gebouwd zijn om de browser te versnellen.

Developer Tools

Alle tools die beschikbaar zijn voor web ontwikkelaars om hun eigen code te analyseren zijn onderdeel van de ingebouwde Developer Tools. Naast een complete JavaScript debugger bevatten de Developer Tools ook de zogenaamde *Timeline View*. Hiermee is het mogelijk om een overzicht te krijgen van de acties die de browser intern uitvoert tijdens het renderen van een pagina. De 4 belangrijkste acties worden aangegeven: de laadtijd van resources, de tijd die door JavaScript nodig is, de tijd die nodig is om de pagina op te bouwen en de tijd die nodig is om de pagina te tekenen.

Naast de Timeline View zijn er ook verschillende profilers, een voor de processor en twee voor het geheugen. De CPU profiler dient om een profiel te maken van de opdrachten die de processor uitvoert en geeft een overzicht in tree. Deze tree kan verder gebruikt worden om trage onderdelen in de code op te sporen en te optimaliseren, er kan ook gecontroleerd worden van waaruit deze functies opgeroepen worden. De geheugen profilers geven een overzicht van de actieve objecten in het geheugen of de allocaties in functie van de tijd om zo het geheugengebruik van een applicatie te inspecteren.

Naast deze tools bevat Chrome ook nog de tracing functie waarbij er in de kern van Chrome gekeken kan worden, deze functie is niet zo gebruiksvriendelijk en vereist achtergrond informatie over de interne werking. Als laatste zijn er ook een aantal JavaScript API's die een *high-resolution* timer ter beschikking stellen. Deze timers zijn nauwkeurig tot op 1 microseconde, uiterst handig voor het maken van nauwkeurige benchmarks.

V8 optimalisaties

JavaScript is een erg dynamische taal waarbij de structuur van objecten tijdens het uitvoeren van een programma gewijzigd kan worden. Traditioneel betekende dit dat de elementen van elk object in een hashmap opgeslagen moesten worden. Hierdoor was het opzoeken van elementen in een object traag vergeleken met talen zoals C waar de structuur van objecten tijdens het compileren vast ligt. Om deze reden gebruikt V8 het concept *hidden classes* om zo toch voor snelle lookups te zorgen. Chrome doet dit door op de achtergrond objecten met een vaste structuur aan te maken, de *hidden classes*. Wanneer de structuur van een object wordt aangepast, wordt er een nieuwe *hidden class* aangemaakt. Met deze techniek kunnen de trager hashmap lookups vervangen worden door vast offsets voor elk element.

Het opzoeken van elementen in een object is nog verder geoptimaliseerd d.m.v. *inline caches*. De standaard lookup code voor elementen wordt op de verschillende aanroepplaatsen in de code vervangen door een inline cache. De inline cache bevat de directe offset voor het gevraagde element, zodat het element rechtstreeks opgevraagd kan worden. Deze optimalisatie is alleen mogelijk door de vaste structuur van elementen in de hidden classes.

Daarnaast bevat V8 ook *Crankshaft*, een extra compiler naast de standaard compiler die veel gebruikte delen code opnieuw compileert en hierop verdere optimalisaties toepast. Het detecteren van veel gebruikte code gebeurt met een teller in iedere functie die verminderd wordt bij iedere call. De geoptimaliseerde code is veel sneller maar het genereren van deze code is trager waardoor er een goede afweging gemaakt moet worden tussen welke code geoptimaliseerd wordt en welke code niet.

getImageData

Tijdens de ontwikkeling van de in-casu applicatie werd een nadeel van het renderen via het hardware path opgemerkt. Het uitlezen van de pixels op een canvas was merkbaar trager wanneer dit via het hardware path getekend werd dan wanneer dit via het software path verliep.

Verder onderzoek wees uit dat er een vrij groot verschil was: 25% tot 66% trager afhankelijk van de gebruikte grafische kaart (zie tabel 4.1 en figuur 4.10). De oorzaak van dit verschil in performantie komt door een trage readback van de grafische kaart in het hardware path. In het software path kan de pixel data eenvoudig van uit het werkgeheugen gekopieerd worden zonder over de eerder trage GPU - CPU bus te gaan.

Werken met afbeeldingen

Dit onderdeel bevat ook onderzoek naar de beste methode om afbeeldingen te gebruiken in combinatie met WebGL. Dit onderzoek start met een vergelijkende benchmark tussen de decodeer tijden voor verschillende afbeeldingsformaten (JPEG, PNG, WEBP en BMP). Alle geteste formaten hebben erg vergelijkbare tijden behalve WEBP, dit formaat is merkbaar trager dan de andere drie. De oorzaak hier ligt bij de hogere compressie bij WEBP met kleinere bestanden als gevolg. Ook het onderzoek naar de beste methode voor het laden van van afbeeldingen naar een WebGL canvas leverde weinig verschillen op. Van alle geteste methodes is het uploaden a.d.h.v. een *ArrayBufferView* het snelst omdat hier de afbeelding reeds op voorhand is gedecodeerd. De andere methodes uploaden de afbeelding ongeveer even snel, het uploaden via een 2D canvas gebeurt wel asynchroon waardoor deze methode uiteindelijk de voorkeur geniet.

Tekst en handschrift voor WebGL

WebGL heeft standaard geen ondersteuning voor het tekenen van tekst of handschrift (tekst of annotaties). Door WebGL te combineren met een 2D canvas of SVG is dit wel mogelijk. De 2D canvas API biedt wel functionaliteit om tekst of lijnen te tekenen, de inhoud van dit canvas kan dan als texture geladen worden. Ook voor het tekenen van handschrift kan teruggevallen worden op de 2D canvas API of op SVG paths. De resultaten blenden goed samen op het WebGL canvas en geven een degelijk resultaat.

Remote WebGL rendering

Het laatste onderdeel van deze thesis beschrijft de implementatie van een remote WebGL rendering oplossing waarbij het renderen door een centrale server gebeurt. Aan de client zijde wordt het WebGLRenderingContext object gesimuleerd door een drop-in vervanging. Dit nieuwe object serialiseert alle WebGL functie aanroepen en verstuurt de calls naar de render server, vergelijkbaar met RPC. De render server kan meerdere clients tegelijk afhandelen, de verbinding tussen client en server verloopt via WebSockets of WebRTC. Het serializeren van objecten en functie aanroepen is binair en gebruikt een speciaal frame ontwerp. De structuur van beide frames is weergeven in afbeelding 6.4 en afbeelding 6.5.

De communicatie verloopt altijd via twee aparte verbindingen. Het versturen van de RPC calls loopt via een stabiele WebSocket verbinding, het resultaat (de huidige staat van het canvas) wordt teruggestuurd over een tweede verbinding. Deze verbinding kan een WebSocket of een WebRTC verbinding zijn. In het geval van WebRTC kan het canvas als een videostream doorgestuurd worden, in het andere geval zullen afbeeldingen gebruikt worden. Om de impact van deze afbeeldingen op zowel de processor als het netwerk te beperken is de kwaliteit afhankelijk van de performantie van het volledige systeem.

Het verwijzingen naar objecten moeten gesynchroniseerd blijven tussen de client en de server. De RemoteWebGLObject objecten zorgen hiervoor. Ieder object krijgt een globaal unieke identifier die wanneer het object over het netwerk verstuurd moet worden meegestuurd wordt. Zowel client als server kunnen zo naar de juiste objecten verwijzen door de juiste identifiers mee te geven.

Conclusie

Omdat het doel van deze thesis breed is hebben we besloten hiervoor twee onderzoeksvragen te formuleren en deze individueel te beantwoorden. Het beantwoorden van de eerste onderzoeksvraag gebeurde aan de hand van een gevalsanlyse van de beschikbare tools, technologieën en browser features. De eerste hoofdstukken hebben deze onderzoekvraag uitgebreid beantwoord en onderzocht. Als definitieve conclusie kunnen we stellen dat er op het web zeker voldoende tools beschikbaar zijn ter ondersteuning van ontwikkelaars. Ook de nieuwe web standaarden helpen ontwikkelaars bij het ontwikkelen en porten van hun applicaties naar het web. Als laatste punt kwam ook de interne werking van web browsers, specifiek Chrome, aan bod. Hieruit blijkt dat ook de browser vendors verscheidene features hebben ontwikkeld om het web in het algemeen te verbeteren.

In hoofdstuk 6 werd uitgezocht of het mogelijk is om WebGL te gebruiken op toestellen met beperkte hardware of oude web browsers. Het resultaat hiervan is een werkende implementatie die uitvoerig getest en verbeterd is. De implementatie gebruikt een centrale render server die WebGL functies uitvoert voor alle verbonden clients via een RPC systeem. Het resultaat kan door middel van afbeeldingen doorgestuurd worden of in moderne web browsers met een WebRTC video stream. De implementatie werkt ook in oudere browsers zoals Internet Explorer 8. Uit de benchmarks kwam wel de beperkte schaalbaarheid van de server implementaties naar boven. Hier kan in de toekomst nog verder aan gebouwd worden.

In het algemeen kunnen we besluiten dat beide onderzoeksvragen op een positieve manier beantwoord zijn en dat het web een volwassen applicatie platform is geworden met uitgebreide ondersteuning voor ontwikkelaars.

Bibliography

[1] Abhinaba Basu: 2009, Back to basics: Reference counting garbage collection.
URL: *http://blogs.msdn.com/b/abhinaba/archive/2009/01/27/back-to-basics-reference-counting-garbage-collection.aspx*

[2] Adam Bergkvist (Ericsson), Daniel C. Burnett (Voxeo), C. A.: 2013, Webrtc 1.0: Real-time communication between browsers.
URL: *http://www.w3.org/TR/webrtc/*

[3] Adobe: n.d., Photoshop cc.
URL: *http://www.adobe.com/products/photoshop.html?*

[4] Anne van Kesteren: n.d.a, Xmlhttprequest - living standard.
URL: *http://xhr.spec.whatwg.org/*

[5] Anne van Kesteren, J.: n.d.b, Encoding - living standard.
URL: *http://encoding.spec.whatwg.org/*

[6] Ben Vanik: n.d., Webgl inspector.
URL: *https://chrome.google.com/webstore/detail/webgl-inspector/ogkcjmbhnfmlnielkjhedpcjomeaghda/*

[7] blink dev: n.d., Intent to ship: Directwrite on windows.
URL: *https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/gjzjDTlSENI*

[8] C. J. Cheney: 1970, A nonrecursive list compacting algorithm.
URL: *http://dl.acm.org/citation.cfm?doid=362790.362798*

[9] Cameron McCormack: n.d., Web idl.
URL: *http://www.w3.org/TR/WebIDL/*

[10] caniuse: n.d., Can i use - webgl.
URL: *http://caniuse.com/#search=webgl*

[11] Cisco Systems: n.d., Ipv4 address depletion.
URL: *http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_10-3/103_addr-dep.html*

[12] Cody Krieger: n.d., gfxcardstatus.
URL: *http://gfx.io/*

[13] Daniel Clifford: n.d., Google i/o 2012 - breaking the javascript speed limit with v8.
URL: *https://www.youtube.com/watch?v=UJPdhx5zTaw*

[14] Dean Jackson (Apple Inc.): n.d., Webgl specification.
URL: *https://www.khronos.org/registry/webgl/specs/1.0/, Year = 2013*

[15] Developers, T. X. T.: n.d., The x toolkit: Webgl for scientific visualization.
URL: *https://github.com/xtk/X#readme*

[16] Ecma International: n.d., Ecmascript® language specification.
URL: *http://www.ecma-international.org/ecma-262/5.1/*

[17] Erik Dahlstrom, Opera Software et al.: 2011, Scalable vector graphics (svg) 1.1 (second edition).
URL: *http://www.w3.org/TR/SVG11/*

[18] Erik Moller, P.: n.d., Chasing pixels, blinkon 2.0.
URL: *http://people.opera.com/emoller/blinkon2/*

[19] GitHub: n.d., Deploying at github.
URL: *https://github.com/blog/1241-deploying-at-github*

[20] Google Developers: n.d., Real-time communication with webrtc: Google i/o 2013.
URL: *http://www.youtube.com/watch?v=p2HzZkd2A40*

[21] Google, I.: n.d.a, Quantum computing playground.
URL: *http://qcplayground.withgoogle.com/*

[22] Google, I.: n.d.b, Web metrics: Size and number of resources.
URL: *https://developers.google.com/speed/articles/web-metrics*

[23] Google, Inc.: n.d.a, Angle: Almost native graphics layer engine.
URL: *https://code.google.com/p/angleproject/*

[24] Google Inc.: n.d.b, Garbage collection for blink c++ objects.
URL: *http://www.chromium.org/blink/blink-gc*

[25] Google, Inc.: n.d.c, Javascript memory profiling.
URL: *https://developers.google.com/chrome-developer-tools/docs/javascript-memory-profiling*

[26] Google, Inc.: n.d.d, Quic geek faq.
URL: *https://docs.google.com/a/martijnc.be/document/d/1lmL9EF6qKrk7gbazY8bIdvq3Pno2Xj_l_YShP40GLQE/edit*

[27] Google Inc.: n.d.e, Skia, a 2d graphics library.
URL: *https://code.google.com/p/skia/*

[28] Google, Inc.: n.d.f, V8 javascript engine.
URL: *https://code.google.com/p/v8/*

[29] Google Inc.: n.d.g, Webrtc.
URL: *http://tools.ietf.org/html/rfc5245*

[30] Google, Inc.: n.d.h, Webrtc architecture.
URL: *http://www.webrtc.org/reference/architecture*

[31] http://caniuse.com/: n.d.a, Can i use - opus codec.
URL: *http://caniuse.com/#search=opus*

[32] http://caniuse.com/: n.d.b, Can i use - webm video format.
URL: *http://caniuse.com/#search=vp8*

[33] I. Fette (Google Inc.), A.: n.d., The websocket protocol.
URL: *https://tools.ietf.org/html/rfc6455, Year = 2011*

[34] Ian Hickson (Google Inc.): n.d., Html, living standard.
URL: *http://www.whatwg.org/specs/web-apps/current-work/multipage/*

[35] IETF: 1999, Hypertext transfer protocol – http/1.1.
URL: *http://www.ietf.org/rfc/rfc2616.txt*

[36] IETF: 2008, The transport layer security (tls) protocol.
URL: *http://tools.ietf.org/html/rfc5246/*

[37] IETF: n.d., The internet engineering task force.
URL: *http://www.ietf.org/*

[38] J. Rosenberg: n.d., Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols.
URL: *http://tools.ietf.org/html/rfc5245*

[39] Jake Brutlag: n.d., Speed matters.
URL: *http://googleresearch.blogspot.be/2009/06/speed-matters.html*

[40] Jatinder Mann, Microsoft Corp.: n.d.a, High resolution time.
URL: *http://www.w3.org/TR/hr-time/*

[41] Jatinder Mann, Microsoft Corp., Zhiheng Wang, Google Inc., A.: n.d.b, User timing.
URL: *http://www.w3.org/TR/user-timing/*

[42] Kentaro Hara: n.d.a, Javascript bindings.
URL: *https://docs.google.com/a/google.com/presentation/d/1M8OFkFXg6bDURncZMw01if2A89YHYl94XmWT1pjvvJI*

[43] Kentaro Hara: n.d.b, Multi-process architecture.
URL: *http://www.chromium.org/developers/design-documents/multi-process-architecture*

[44] Kevin Millikin, F.: n.d., A new crankshaft for v8.
URL: *http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html*

[45] Lilli Thompson: n.d., Console to chrome.
URL: *http://console-to-chrome.appspot.com/*

[46] M. Baugher, D. McGrew, M. E. K.: n.d., The secure real-time transport protocol (srtp).
URL: *http://tools.ietf.org/html/rfc3711*

[47] M. Belshe, R. Peon, M.: n.d., Hypertext transfer protocol version 2.
URL: *http://tools.ietf.org/html/draft-ietf-httpbis-http2-12*

[48] Mads Ager: n.d., Google i/o 2009 - v8: High performance javascript engine.
URL: *https://www.youtube.com/watch?v=FrufJFBSoQY*

[49] Microsoft: n.d., Uniscribe.
URL: *http://msdn.microsoft.com/en-us/library/windows/desktop/dd374091(v=vs.85).aspx*

[50] Micrsoft: n.d., Directwrite.
URL: *http://msdn.microsoft.com/en-us/library/windows/desktop/dd368038(v=vs.85).aspx*

[51] Mozilla, E.: 2013, Epic citadel demo shows the power of the web as a platform for gaming.
URL: *https://blog.mozilla.org/futurereleases/2013/05/02/epic-citadel-demo-shows-the-power-of-the-web-as-a-platform-for-gaming/*

[52] notmasteryet: n.d., jpgjs.
URL: *https://github.com/notmasteryet/jpgjs*

[53] Paul Bakaus: n.d., The illusion of motion.
URL:
http://paulbakaus.com/tutorials/performance/the-illusion-of-motion/

[54] Qt Project Hosting: n.d., Qwebpage class reference.
URL: *http://qt-project.org/doc/qt-4.8/qwebpage.html*

[55] R. Stewart, Q. Xie, K. C. E.: n.d., Stream control transmission protocol.
URL: *http://www.ietf.org/rfc/rfc2960*

[56] Rovio: n.d., Angry birds.
URL: *http://chrome.angrybirds.com/*

[57] StatCounter: n.d., Statcounter global stats.
URL: *http://gs.statcounter.com/#all-os-ww-yearly-2008-2014*

[58] Technologies, A.: n.d., The akamai state of the internet report.
URL: *http://www.akamai.com/stateoftheinternet/*

[59] The Chromium Authors: n.d.a, Adding traces to chromium/webkit/javascript.
URL: *http://dev.chromium.org/developers/how-tos/trace-event-profiling-tool/tracing-event-instrumentation*

[60] The Chromium Authors: n.d.b, The chromium projects.
URL: *http://www.chromium.org/*

[61] The Chromium Authors: n.d.c, The chromium source code.
URL: *https://code.google.com/p/chromium/codesearch*

[62] The Chromium Authors: n.d.d, Content module.
URL: *http://www.chromium.org/developers/content-module*

[63] The Chromium Authors: n.d.e, Gpu command buffer.
URL:
http://www.chromium.org/developers/design-documents/gpu-command-buffer

[64] The Chromium Authors: n.d.f, Ubercompositor - chrome rendering model.
URL: *https://docs.google.com/a/chromium.org/document/d/1ziMZtS5Hf8azogi2VjSE6XPaMwivZSyXAIIp0GgInNA/*

[65] The Google Chrome Team: n.d., The evolution of the web.
URL: *http://www.evolutionoftheweb.com/*

[66] The HarfBuzz Authors: n.d., Harfbuzz opentype text shaping engine.
URL: *https://github.com/behdad/harfbuzz*

[67] The Khronos Group: n.d.a, Opengl es 2.0 for the web.
URL: *http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html*

[68] The Khronos Group: n.d.b, Opengl shading language.
URL: *http://www.opengl.org/documentation/glsl/*

[69] The Khronos Group: n.d.c, Sharing resources across webglrendering contexts.
URL: *http://www.khronos.org/webgl/wiki/SharedResouces*

[70] The Khronos Group: n.d.d, Webgl security.
URL: *http://www.khronos.org/webgl/security/*

[71] The Nielsen Company: n.d., An era of growth: the cross-platform report.
URL: *http://www.nielsen.com/us/en/reports/2014/an-era-of-growth-the-cross-platform-report.html*

[72] The peering team: n.d., The peerjs library.
URL: *http://peerjs.com/*

[73] TransGaming Inc.: n.d., Swiftshader.
URL: *http://transgaming.com/swiftshader*

[74] W3C: n.d., The world wide web consortium.
URL: *http://www.w3.org/*

[75] WHATWG: n.d.a, The canvas element.
URL: *http://www.khronos.org/webgl/*

[76] WHATWG: n.d.b, The web hypertext application technology working group.
URL: *http://www.whatwg.org/*

[77] Zhiheng Wang: n.d., Navigation timing.
URL: *http://www.w3.org/TR/navigation-timing/*

Appendix A

TextureHelpers Interfaces

A.1 WebGLTextTextureHelper

```
WebGLTextTextureHelper WebGLTextTextureHelper::WebGLTextTextureHelper(
    WebGLRenderingContext webglcontext, string backing);
undefined WebGLTextTextureHelper::addTextElement(string identifier,
    WebGLTextElement textelement);
undefined WebGLTextTextureHelper::removeTextElement(string identifier);
WebGLTextElement WebGLTextTextureHelper::getTextElement(string identifier
    );
WebGLTexture WebGLTextTextureHelper::getTexture();
undefined WebGLTextTextureHelper::clear();
undefined WebGLTextTextureHelper::reset();
```

A.2 WebGLTextElement

```
WebGLTextElement WebGLTextElement::WebGLTextElement(string text, number x
    , number y);
undefined WebGLTextElement::setText(string text);
undefined WebGLTextElement::setPosition(number x, number y);
undefined WebGLTextElement::setFont(string font);
undefined WebGLTextElement::setFontSize(number fontsize);
undefined WebGLTextElement::setTextAlign(string fontsize);
undefined WebGLTextElement::setColor(string color);
string WebGLTextElement::getText();
number WebGLTextElement::getX();
number WebGLTextElement::getY();
string WebGLTextElement::getFont();
number WebGLTextElement::getFontSize();
string WebGLTextElement::getTextAlign();
string WebGLTextElement::getColor();
```

Appendix B

AnnotationHelpers Interfaces

B.1 WebGLAnnotationHelper

```
WebGLTextTextureHelper WebGLTextTextureHelper::WebGLTextTextureHelper(
    WebGLRenderingContext webglcontext, function onchanged);
undefined WebGLTextTextureHelper::addAnnotation(WebGLAnnotation
    annotation);
undefined WebGLTextTextureHelper::removeAnnotation(string identifier);
WebGLTextElement WebGLTextTextureHelper::getAnnotation(string identifier)
    ;
WebGLTexture WebGLTextTextureHelper::getTexture();
undefined WebGLTextTextureHelper::clear();
undefined WebGLTextTextureHelper::reset();
```

B.2 WebGLAnnotation

```
WebGLTextElement WebGLTextElement::WebGLAnnotation(string identifier, string
    color);
undefined WebGLTextElement::setColor(string color);
undefined WebGLTextElement::setLineWidth(number width);
undefined WebGLTextElement::setLineCap(string capType);
undefined WebGLTextElement::setLineJoin(string joinType);
undefined WebGLTextElement::addPoint(number x, number y);
string WebGLTextElement::getColor();
string WebGLTextElement::getLineWidth();
string WebGLTextElement::getLineCap();
string WebGLTextElement::getLineJoin();
string WebGLTextElement::getIdentifier();
```

Appendix C

Testcases

C.1 getimagedata.html

```
<html>
  <head>
    <title>Testcase for 2D canvas getImageData performance</title>
  </head>
  <body>
    <h1>Testcase for 2D canvas getImageData performance</h1>
    <p>This page contains a testcase designed to validate the performance
        difference of <em>CanvasRenderingContext2D.getImageData()</em> when
        hardware acceleration is enabled and when hardware acceleration is
        disabled. In Google Chrome this can behaviour can be controlled through
        the <em>Disable accelerated 2D canvas</em> flag on <em>chrome://flags</em
        >.</p>
    <p>All output (errors and timing information) is send to the console.</p>
    <h2>Canvas</h2>
    <canvas id="canvas"></canvas>
    <div>
      <label for="run-test">Run tests:</label>
      <button id="run-test" name="run-test">click here</button>
    </div>
    <h2>Original image</h2>
    <img id="original-image" src="./assets/image-large.jpg" alt="Example image"
        width="1000">
    <script>
      (function(){
        var canvas = document.querySelector('#canvas');
        var original_image = new Image();
        var context = canvas.getContext('2d');
        var total = 0;

        // We will use chrome.Interval for high resolution timing information
        if (!chrome.benchmarking) {
          console.error('Run Chrom{e|ium} with the --enable-benchmarking command
              line flag for accurate timing information.');
          document.querySelector('#run-test').disabled = 'disabled';
          document.querySelector('#run-test').innerText = 'Check the console for
              errors';
        }
        canvas.width = 1000;
        canvas.height = 800;
        original_image.src = "./assets/image-large.jpg";
        function testcase() {
          var timer = new chrome.Interval();
          var iterations = 1000;

          timer.start();
          while (iterations--) {
            context.getImageData(0 ,0, canvas.width, canvas.height);
          }
          timer.stop();

          total += timer.microseconds();

          console.log('Ran getImageData 1000 times in ' + timer.microseconds() +
              ' microseconds (' + timer.microseconds()/1000000 + ' seconds).');
        }

        function runTests() {
          // Detect CORS violations when using file://
          try {
            context.getImageData(0 ,0, 1, 1);
          } catch (e) {
            console.error('Run Chrom{e|ium} with the --allow-file-access-from-
                files command line flag when opening files locally.');
            return;
          }

          console.log('Running testcase 100 times');

          var timer = new chrome.Interval();
          var iterations = 1;
          timer.start();

          while (iterations--) {
            testcase();
          }

          timer.stop();

          console.log('Ran test 100 times in ' + timer.microseconds() + '
              microseconds (' + timer.microseconds()/1000000 + ' seconds).');
          console.log('Average time per run: ' + (total / 100) + ' microseconds
              (' + (total / 100000000) + ' seconds).' );
        }

        function drawImage() {
          context.drawImage(original_image, 0, 0, original_image.width,
              original_image.height, 0, 0, canvas.width, canvas.height);
        }

        window.onload = drawImage;
        document.querySelector('#run-test').onclick = runTests;
      }());
    </script>
  </body>
</html>
```

C.2 canvassizebenchmark.html

```
<html>
  <head>
    <title>Test-case for 2D canvas getImageData performance</title>
  </head>
  <body>
    <h1>Test-case for 2D canvas getImageData performance with different canvas
        sizes</h1>
    <p>This page contains a test-case designed to investigate the performance of
        <em>CanvasRenderingContext2D.getImageData()</em> on canvases of different
         sizes when hardware acceleration is enabled and when hardware
        acceleration is disabled. In Google Chrome this can behaviour can be
        controlled through the <em>Disable accelerated 2D canvas</em> flag on <em
        >chrome://flags</em>.</p>
    <p>All output (errors and timing information) is send to the console.</p>
    <h2>Canvas</h2>
    <div id="container"></div>

    <div>
      <label for="run-test">Run tests:</label>
      <button id="run-test" name="run-test">click here</button>
    </div>

    <script>
      (function(){
        // var c = 100;
        var i = -1;
        var result = [];
        while(i++ <= 17) {
      result[i] = Math.round(c);
      console.log(c);
      c = Math.sqrt((Math.pow(c, 2))*1.5);
        }
       var sizes = [100, 122, 150, 184, 225, 276, 337, 413, 506, 620, 759, 930,
           1139, 1395, 1709, 2093, 2563, 3139, 3844, 4708, 5767, 7062];
        var container = document.querySelector('#container');
        var original_image = new Image();
        var results = [];
        var current = 0;

        // We will use chrome.Interval for high resolution timing information
        if (!chrome.benchmarking) {
          console.error('Run Chrom{e|ium} with the --enable-benchmarking command
              line flag for accurate timing information.');
          document.querySelector('#run-test').disabled = 'disabled';
          document.querySelector('#run-test').innerText = 'Check the console for
              errors';
        }

        original_image.src = "./assets/benchmark-image-large.jpg";

        function testcase(index, size, context) {
          var timer = new chrome.Interval();
          var iterations = 250;

          timer.start();
          while (iterations--) {
            context.getImageData(0 ,0, size, size);
          }
          timer.stop();

          return timer.microseconds();
        }

        function setUpCanvas(size) {
          // Cleanup
          container.innerHTML = '';

          var canvas = document.createElement('canvas');
          canvas.width = size;
          canvas.height = size;
          context = canvas.getContext('2d');

          context.drawImage(original_image, 0, 0, original_image.width,
              original_image.height, 0, 0, canvas.width, canvas.height);
          container.appendChild(canvas);

          return context;
        }

        function callTest(index, size, context) {
          var iterations = 25;
          var total = 0;
          console.log('Running testcase ' + index + ' (' + size + 'px) ' +
              iterations + ' times');

          while (iterations--) {
            total += testcase(index, size, context);
          }

          console.log('Average time per run: ' + (total / 25) + ' microseconds ('
               + (total / 25000000) + ' seconds).');
          results[index] = total / 25;

          current++;
          setTimeout(runTests.bind(this), 10);
        }

        function runTests() {
          if (current < sizes.length) {
            // Setup canvas
            var context = setUpCanvas(sizes[current]);

            // Detect CORS violations when using file://
            try {
              context.getImageData(0 ,0, 1, 1);
            } catch (e) {
              console.error('Run Chrom{e|ium} with the --allow-file-access-from-
                  files command line flag when opening files locally.');
              return;
            }

            // Do nothing for a second so the browser has time to do its own
                thing (GC and Layout)
            setTimeout(callTest.bind(this, current, sizes[current], context),
                1000);
          } else {
            console.log(JSON.stringify(results));
          }
        }
        document.querySelector('#run-test').onclick = runTests;
      }());
    </script>
  </body>
</html>
```

C.3 binarytostring.html

```
<html>
  <head>
    <title>Testcase for different binary to text conversion methods</title>
  </head>
  <body>
    <p>All output (errors and timing information) is send to the console.</p>

    <div>
      <button id="run-test">Run tests</button>
    </div>

    <script>
      (function(){
        var length = 3200000;
        var buffer = new ArrayBuffer(length * 2);
        var view = new DataView(buffer, 0);
        var iterations = 1000;

        // We will use chrome.Interval for high resolution timing information
        if (!chrome.benchmarking) {
          console.error('Run Chrom{e|ium} with the --enable-benchmarking command line flag for accurate timing information.');
          document.querySelector('#run-test').disabled = 'disabled';
          document.querySelector('#run-test').innerText = 'Check the console for errors';
        }

        function setupArrayBuffer(length) {
          var chars = "abcdefghijklmnopqrstuvwxyz0123456789=";

          for (var i = 0; i < length; i++) {
            view.setUint16(i * 2, chars.charCodeAt(chars.charAt(Math.floor(Math.random() * chars.length))));
          }
        }

       function testcase1() {
          var text = '';
          for (var i = 0; i < length; i++) {
            text += String.fromCharCode(view.getUint16(i * 2));
          }
        }

        function testcase2() {
          var characters = new Array(Math.min(length, 65000));
          var shard = 0;
          var text = '';

          for (var i = 0; i < length; i++) {
            characters[i - (shard * 65000)] = view.getUint16(i * 2);

            if (i - (shard * 65000) == 65000 - 1) {
              shard++;
              text += String.fromCharCode.apply(null, characters);
              characters = new Array(Math.min(65000, length - (shard * 65000)));
            }
          }

          text += String.fromCharCode.apply(null, characters);
        }

        function testcase3() {
          var text = '';
          var tempArray = new Array(length);

          for (var i = 0; i < length; i++) {
            tempArray[i] = String.fromCharCode(view.getUint16(i * 2));
          }

          text = tempArray.join('');
        }

        function runTests() {
          console.log('Running testcase 1000 times');

          var timer = new chrome.Interval();
          var iterations = 1000;

          timer.start();
          while (iterations--) {
            testcase2();
          }
          timer.stop();

          console.log('Ran test 1000 times in ' + timer.microseconds() + '
              microseconds (' + timer.microseconds()/1000000 + ' seconds).');
        }

        document.querySelector('#run-test').onclick = runTests;
        setupArrayBuffer(length);
      }());
    </script>
  </body>
</html>
```

Auteursrechtelijke overeenkomst