Migrating Real-time Applications to the Web Browser

Technical Overview and Case Study

UMNAD 932/12

Benjamin Eriksson
Migrating Real-time Applications to the Web Browser
– Technical Overview and Case Study

av

Benjamin Eriksson

UMNAD 932/12

Examensarbete 30 hp för civilingenjörsexamen i Teknisk Datavetenskap

Fredrik Georgsson
Kursansvarig

Mikael Rännar
Handledare

Fredrik Georgsson
Examinator
Abstract

Today the web browser get a more central role in the arsenal of programs users use and at the same time installing programs is a bit of the past and every type of applications gets a web alternative. Users sees this as a good thing, however software companies which have put a lot of time, effort and money into their products might not be as happy in this platform paradigm shift.

The question is though, does it need to be a paradigm shift? Or is it possible to move an old application into this new, Operating System (OS) and architecture agnostic, multi purpose system the web has developed into?

This thesis tries to summarize which techniques there are, how they work, what their strengths and weaknesses are. As a bonus it also contains an independent case study of how it is to use the technology Native Client (NaCl) in a large scale project.

The conclusion, from the case study, is that NaCl is a very promising technology both for migrating old code and for creating high performance code for this new platform. Our tests shows a 12.9% speed up when we using the 64 bit NaCl version over the native x86 32 bit version of Algodoo.

However, even though the technology seems mature the Software Development Kit (SDK) aren’t which might be reason enough to idle for a while.

keywords: C/C++, web browser, plug-in, case study, NaCl, Emscripten, Algodoo
Acknowledgements

I would first like to thank my supervisor, Emil Ernefeldt, and other colleagues at Algoryx and my supervisor Mikael Rännar at the department of Computing Science at Umeå University for their support and valuable feedback. I would also like to thank to my family and friends, for their support, you know who you are.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Screen shot of Algodoo running on Mac OS X</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>The NaCl Environment Layers</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Interleaved process activity during the life time of a web site tab</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>Stalling plug-in causing other components to stop respond</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Execution flow for a blocking call using the nacl-mounts library</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>Simplified NaCl flow schematic, from src to running program</td>
<td>13</td>
</tr>
<tr>
<td>3.6</td>
<td>Simplified Portable Native Client (PNaCl) flow schematic, from src to running program</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Flowchart of the porting process</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Algodoo's basic program flow</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Algodoo running inside Chrome on GNU/Linux using Native Client</td>
<td>27</td>
</tr>
<tr>
<td>5.2</td>
<td>Screenshot of the test scene running inside of Algodoo</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Wall time of number of seconds it takes to simulate 10 time steps</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Box plot of the data found in Figure 5.3 on page 30</td>
<td>30</td>
</tr>
<tr>
<td>5.5</td>
<td>Wall time of number of seconds it takes to simulate 100 time steps</td>
<td>31</td>
</tr>
<tr>
<td>5.6</td>
<td>Box plot of the data found in Figure 5.5 on page 31</td>
<td>31</td>
</tr>
<tr>
<td>5.7</td>
<td>Average wall time for simulating 1 time step after overhead is removed</td>
<td>32</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Compatibility matrix for current browsers and plug-in systems . . . 8
3.2 Feature matrix of the different technologies . . . . . . . . . . . . 17

4.1 Availability depending on architecture . . . . . . . . . . . . . . . . 22
List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Desktop main function implementation</td>
<td>25</td>
</tr>
<tr>
<td>4.2 NaCl initialization implementation</td>
<td>25</td>
</tr>
<tr>
<td>5.1 Script used to time the physics simulation in the test scene</td>
<td>29</td>
</tr>
<tr>
<td>B.1 trace.h</td>
<td>47</td>
</tr>
<tr>
<td>B.2 example.cpp</td>
<td>47</td>
</tr>
<tr>
<td>B.3 output</td>
<td>48</td>
</tr>
</tbody>
</table>
Abbreviations

**ABI** Application Binary Interface. 7, 9

**API** Application Programming Interface. xvi, 7–9, 15, 21

**DMCA** Digital Millennium Copyright Act. 16

**GNU GPL** GNU General Public License. xv, 10

**GNU LGPL** GNU Lesser General Public License. 10

**IR** Intermediate Representation. 13, 15

**JIT** Just in Time. 15

**NaCl** Native Client. i, vi, vii, xi, 3, 6, 8–14, 16, 17, 19, 22–29, 33, 35, 45, 46

**NPAPI** Netscape Plugin API. xvi, 8, 9

**OS** Operating System. i, 13, 23, 28

**PNaCl** Portable Native Client. vii, 13, 14

**POSIX** Portable Operating System Interface. 11, 23, 26, 45

**PPAPI** Pepper Plugin API. 8–11, 23, 24, 28, 33

**SDK** Software Development Kit. i, 11, 22, 26, 46

**SDL** Simple Directmedia Library. 15

**SFI** Software Fault Isolation. 7
Glossary

Application Binary Interface (ABI) The interface which defines how to communicate between different parts of a software on a binary level. It defines how different data types are represented, how functions are called program start up etc. xiii, 7, 9

Application Programming Interface (API) The interface which a piece of software can communicate with another application or library. It defines which functions exist and their input/output. xiii, xvi, 7–9, 15, 21

CMake An open source cross platform build system. 22

Digital Millennium Copyright Act (DMCA) An United States copyright law. xiii, 16

GNU General Public License (GNU GPL) A copyleft license which adds restrictions which forces derivate works to comply to the same terms. xiii, xv, 10

GNU Lesser General Public License (GNU LGPL) A copyleft license which is based on the GNU General Public License (GNU GPL) but allows dynamical linking with software with other licenses. xiii, 10

Intermediate Representation (IR) A format or data structure which is created by an input of a program and can generate multiple outputs. It is used in LLVM in a step between the compilation and the instruction generation to make it possible to have multiple input language and output instruction generators. xiii, 13, 15

Just in Time (JIT) A compiler technique which means that the program is compiled to native machine instructions after it is started. Since the compiler is run during runtime, instead of before as traditionally, the compiler has access to runtime information and can hence optimize the program with it. xiii, 15

LLVM Compiler infrastructure. Formerly abbreviation of Low Level Virtual Machine. xv, 13, 15
Native Client (NaCl) /ˈnækl/ Google’s technology which makes it possible to run native code in the web browser in a secure environment independent of which operation system is used. i, vi, vii, xi, xiii, 3, 6, 8-14, 16, 17, 19, 22-29, 33, 35, 45, 46

Netscape Plugin API (NPAPI) Netscape’s Application Programming Interface (API) for plug-ins, created in 1996 and still used today. Makes it possible to interface web sites with native code to enrich their functionality and appearance. xiii, xvi, 8, 9

Pepper Plugin API (PPAPI) Google’s new plug-in API which was formerly based on Netscape Plugin API (NPAPI) but has since been redesigned. Makes it possible to create both portable and sand-boxed plug-ins. xiii, 8-11, 23, 24, 28, 33

Portable Native Client (PNaCl) /ˈpIn@kl/ Google’s technology to make it possible to run native code, generated from a portable binary, in a web browser independent of which operation system or architecture is used. vii, xiii, 13, 14

Portable Operating System Interface (POSIX) /ˈp6zIks/ Set of standards for software compatibility between operative systems. xiii, 11, 23, 26, 45

SCons An open source cross platform python based build system with automatic dependency analyze capabilities. 22

Simple Directmedia Library (SDL) Popular cross-platform library which abstracts low level access to, otherwise platform specific, audio, 2D and 3D graphics, input devices, and more. xiii, 15

Software Development Kit (SDK) Set of software used to create new software for a product. i, xiii, 11, 22, 26, 46

Software Fault Isolation (SFI) Software based technology which makes it possible to sandbox a program’s code and data segments so an untrusted native application can be run safely within a virtual memory space. xiii, 7

WebKit An open source web browser engine. The Project started as a fork of the web engine found in KDE’s web browser Konquror. Webkit’s web rendering engine is used in e.g. Safari, Chrome, and Qt. 21
Chapter 1

Introduction

As more and more people get connected to the Internet using both wired and mobile connections, people has started to use websites for more than just information gathering as they used to. People use websites for almost everything these days; from writing diaries and sending email to writing documents and streaming video, creating presentations and playing games. This transformation in use has been possible due to the scripting language JavaScript which has been transformed to fit the needs over the years.

Using JavaScript have not been the only choice in order to make interactive websites though, several other technologies such as; Java, Flash, Shockwave, and native code using plug-ins, to mention a few, has been used over the years.

In a world where more and more web applications pop up and becomes the new standard and where nothing needs to be installed and nothing can harm your computer, how do you catch the new wave with a legacy system programmed in classic programming language such as C and C++?

Is it possible to convert an old program and reuse the code base that you and your company already invested so many hours in onto the new world of install-less software on the web? And perhaps more importantly, how can conversion be done?

This thesis will introduce you to technology making this possible as well as presenting a case study of how it is to port an application with a legacy C++ code base to a web.

1.1 Background

In order to touch the present, we need to focus on the background. The following chapter presents the background information concerning this thesis project.

Algoryx AB  Algoryx AB started in 2007 as a spin off company from Umeå University’s VRlab. The company has today two products; AgX, which is a 3D physics engine used by the industry for simulating everything from movie visual effects to heavy machines and Algodoo, a physics learning tool.

Algodoo  In 2007 Emil Ernerfeldt created the 2D physics sandbox learning tool called Phun for his Master thesis in Computing Science[1]. Phun was later
taken over by Algoryx Simulation AB and renamed to Algodoo in 2009[2, 3]. Since the transfer, Algodoo has been upgraded to support a new improved graphical user interface[4], optics with lasers[5], and a direct solver which is used to improve simulation of ropes, bridges et cetera[6]. A screen shot of Algodoo can be found in Figure 1.1.

Figure 1.1: Screen shot of Algodoo running on Mac OS X.
1.2 Outline

Chapter 1: Introduction  An introduction of why this thesis was written and some background information which might be interesting for the reader. In fact this is the chapter you are looking at!

Chapter 2: Problem Description  This chapter describes the underlying problem which this thesis has tried to solved by both the practical work this report is based on and the report itself. The chapter also mentions a few prior and related works to this thesis.

Chapter 3: Moving legacy C/C++ code to the Web  This chapter tries to summarize which technologies currently exists for moving a legacy code base into the web browser.

Chapter 4: Case study  Contains a case study on the porting process of an application to Google’s NaCl environment.

Chapter 5: Results  Without results, why bother? - This is the chapter where the results are presented.

Chapter 6: Conclusions  In this chapter all the results and technologies discussed including a section on future work.

Chapter 7: Analysis  This chapter is where the analysis if all the goals, for this thesis, was met.
Chapter 2

Problem Description

The main objective of this thesis is to evaluate the different alternatives for moving real time programs onto the web. This is done by examining how each technology works, what their potentials and drawbacks are and how they perform in relation to each other.

In order to apply the knowledge we gain during the research we will also be testing how it is to port a real world application to the most suitable technology in a case study.

In order to call the project successful we will end up with:

• A comparison in features between the different technologies. Including information about performance and portability.
• A ported real world application to one of the technologies.
• Performance result between native and web version to compare with the results gained in the feasibility study.

2.1 Problem Statement

Evaluate current technologies for porting whole programs using C and C++ to the web and use the most fitting one to convert Algodoo to a web application. This as avoiding the installation process would make it more easy for end users to get started with Algodoo.

2.2 Goals

The goals can be summarized into the following items:

• Evaluating different technology approaches to convert C and C++ applications into web applications.
• Finding out which one suites Algodoo the best.
• Use the best fitting technology to convert Algodoo to a web app.
• Evaluating performance impact due to conversion.
2.3 Purposes

The main purpose of this thesis was to investigate if, how and how well Algodoo could be turned into a web application.

2.4 Related Works

Three NaCl porting case studies was found on the internet when we started writing this thesis, however neither of them used both OpenGL and blocking file access[7, 8, 9].

Tomas Brännström’s Master’s thesis is somewhat related in the sense of discussing how to move physic simulation to the web and incorporate in websites.[10] Another thesis worth mention is Tobias Widlund’s bachelor’s thesis which describes how to design dynamic HTML user interfaces for NaCl modules. Tobias Widlund’s work was however based on the implementation done during this thesis’ case study.[11]

Two more case studies appeared when writing the last chapter[12, 13]. Both these are good reading but appeared to late in this project to have an impact on the case study inside this work.

More information about all the case studies above can be found in Appendix A.1 on page 45.
Chapter 3

Moving legacy C/C++ code to the Web

The traditional way of running C and C++ code on a web site is by creating a plug-in which the user downloads and installs in the web browser. However, things have changed and today there are three technologies which make it possible to run programs, written in C and C++, in a web browser. These technologies are:

**Plug-in** Software that can be downloaded and installed to extend the functionality of the web browser. Common plug-ins are Flash Player, Java and PDF readers[14].

**Native Client** (NaCl) Google’s open source project to create a secure, portable, platform independent and more integrated type of plug-ins by creating a new plug-in API, platform independent Application Binary Interface (ABI) and by only allowing a subset of the processor instruction set in combination of using Software Fault Isolation (SFI)[15, 16].

**C/C++ to JavaScript compilation** Two projects are available: the open source project Emscripten[17] and Onan-Games’ Mandreel[18]. Both use similar technology and have similar performance[19, 20].

Mandreel by Onan-Games has been omitted in this thesis due to the limited documentation on how it works and the similarities to Emscripten.

In the following sections, within this chapter, the technologies mentioned above will be further explained, discussed, and compared to each other in respect to speed, security, portability, etc.

### 3.1 Plug-ins

Plug-ins in web browsers was first used by Netscape Communications for its web browser Netscape Navigator in version 2.0 released in February 1995. This change made it possible to install small programs which extended the functionality of the web browser in a way that was not possible with the prior helper
program approach. Early plug-ins was Real Audio and Acrobat Reader which enabled embedded files in web sites.\[21\]

Microsoft adopted the plug-in approach for making the web richer in 1996 with their release of Internet Explorer 3.0 and both NPAPI and Microsoft’s own plug-in system ActiveX was supported.\[22\] This was however changed in Internet Explorer 5.5 sp 2 when NPAPI support was dropped. After that only ActiveX plug-ins was supported which forced software companies to provide two different plug-ins in order to support both Netscape Navigator and Internet Explorer.\[23\]

Fast forwarding into today all major web browsers supports plug-ins but there are still no framework standard. The same, but extended, APIs that was used back in the 90’s, NPAPI and ActiveX are still used today.

In 2009 Google created a new API called Pepper Plugin API (PPAPI). This was created to address problems with NPAPI, however no other web browser manufacturer has adopted it.\[16, 24, 25\]

The three major plug-in types today are; NPAPI, PPAPI and ActiveX. Which type each browser support can be seen in Table 3.1.

<table>
<thead>
<tr>
<th>Web browser</th>
<th>PPAPI</th>
<th>NPAPI</th>
<th>Active X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Chrome/Chromium</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Safari</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Opera</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Internet Explorer</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3.1: Compatibility matrix for current browsers and plug-in systems.

There is a project called FireBreath which creates an unified API from NPAPI and ActiveX making it possible to have one code base and deploy to all major platforms and web browsers.\[26\]

3.2 Native Client

Native Client (NaCl) has roots back as far as 2008 when it was created to work as a NPAPI plug-in\[27\]. Since then both Opera and Mozilla has rejected the technology for their web browsers.\[28, 29\]

According to the team behind NaCl, two of the main problems with how traditional plug-ins works are in security and portability.\[30\]

A traditional plug-in can execute any code with the same permissions as the web browser, including spawning new processes and reading and writing local files on the file system.\[31\]

Google tries to address this problem in NaCl by introducing some constraints to which instructions and sequences of instructions a NaCl plug-in can execute. The NaCl runtime verifies that these constraints are met by using a validator. This validation is run before a module is executed by the web browser. In order to make it possible for modules to communicate with the world the modules has to call functions in the predefined API, PPAPI. More on this topic can be read in Section 3.2.5 on page 11. A side effect of a secure environment for plug-ins
Another problem NaCl tries to address is that traditional plug-ins are operating system specific, which means that you can not run the same plug-in on Windows and Mac OS X even though they share the same CPU architecture. Since NaCl enforces a predefined API that must be followed there will never be any code that is operating system dependent. The only thing that could cause problem is if different ABIs are used. This is why NaCl defines its own, which is used on all operating systems and architectures.

3.2.1 Application Programming Interface

The available supported APIs when programming for the NaCl platform, is besides a limited C Standard Library, the Pepper API, also known as PPAPI, which handles the interaction between the NaCl module and the web browser as shown in Figure 3.1.

![NaCl Environment Layers](image)

Figure 3.1: The NaCl Environment Layers

PPAPI is a plug-in API inspired from NPAPI but redesigned to make it more secure and portable by, but not limited to, moving the plug-ins out of process.

JavaScript and NaCl Module Communication

All communications between JavaScript and the NaCl Module are done by passing messages through PPAPI. These messages can both be normal strings, which can be useful for json messages, and other types such as array buffers. However, instead of transferring the data to the module, the data gets copied.

More about sending binary messages can be found in Appendix A.3 on page 46.

PPAPI Feature Set

The features supported by the PPAPI resembles the features in JavaScript in the HTML5 standard. The current version of PPAPI has support for, but not limited to: 2D Graphics, 3D Graphics using an API based on OpenGL ES 2.0, Audio, Gamepad input, Fullscreen, and Mouse Capture.

No support exists currently for using webcam and microphone input but this is said to change when the HTML5 standard for it get finalized.

3.2.2 Libraries

The only official standard set of libraries available are those which forms the PPAPI and the two libc libraries glibc and newlib. PPAPI consists of both C
and C++ bindings.

The sparse shared API between the web browser and the NaCl environment is due to security and portability. A larger API would either increase the attack surface of the browser or increase the web browser download size.\[38\]

**Dynamic Linking and Native Client**

Dynamic linking was added when the GNU C Library glibc was introduced as an alternative to newlib in the toolchain.\[39\] The introduction of dynamic linking also makes it possible to share common libraries between modules, which means that shared libraries do not need to be downloaded once for every module. The support of dynamic linking also makes it possible to use GNU Lesser General Public License (GNU LGPL) libraries with non-GNU LGPL/GNU GPL programs as the license otherwise forbids such abominations if neither the source or object files are available.\[40, 41\]

### 3.2.3 Thread Restrictions

Even though NaCl supports pthreads, there are some limitations. In order to explain why the limitations exist we need to start with explaining how NaCl operates within Chrome.

Just as Chrome separates each tab into separate processes due to security reasons, plug-ins are also treated with the same scepticism. This means that if a plug-in crashes the whole browser would not crash, not even the website, just the plug-in element.\[42\] However, even though the web rendering and NaCl lies in separate processes does not mean that they are running asynchronously.\[43\]

In order to render web pages plug-ins needs to be called during rendering for best result. This means that process activity will, at points, be interleaved.\[44\] See Figure 3.2.

![Figure 3.2: Interleaved process activity during the life time of a web site tab.](image)

In the event of a “busy wait” in the main thread of a NaCl plug-in while rendering the whole rendering process, which includes both web rendering and JavaScript execution, would stall.\[43\] See Figure 3.3 on the next page. Because of this, blocking calls are discouraged on the main thread.\[43\]

Another restriction is that all PPAPI calls needs to be done on the main thread. This restriction is planned to be removed in the future.\[43\]
3.2.4 Naclports

Besides the official supported SDK there is another, less supported, but still crucial piece of software: naclports. Naclports is a source repository of patches and build scripts for ports of open source software and libraries to the NaCl platform.\[45\]

There is one library in the set of libraries that is not a port of an existing project but instead created directly for the NaCl project called “nacl-mounts”.

Nacl-mounts

Due to the nature of the PPAPI, no synchronous file system functions are available, which means that Portable Operating System Interface (POSIX) functions such as `open`, `stat` and `read` is not available in the C library provided with NaCl SDK\[43\]. However, nacl-mounts exists to help us with our despair as this library provides a POSIX file system shim on top of local storage and loading files over HTTP.\[46\]

This library simulates blocking calls by spawning non-blocking calls from a background thread onto the main thread and waits for the reply. Since only PPAPI calls can be done on the main thread all these simulated POSIX calls needs to be done off main thread. For an overview of the execution flow for how nacl-mounts simulates blocking calls see Figure 3.4 on the following page.\[47\]

3.2.5 Security

To ensure that the module is safe for execution a validator assures that no potential hazardous side effects can be done using the requested binary\[32\].

The validator looks for the following four categories of problems when determining if the module is safe to execute: Data integrity, Reliable disassembly, Safe instructions, and Control flow integrity\[32\].

On the x86 architecture these restriction are verified in the following way:

Data integrity No data should be possible to neither load nor store outside the sandbox.\[32\] It should also not be possible to write data in segments that are executable, nor execute instructions in data segments.\[48\]

Reliable disassembly No overlapping instructions are allowed. This is ensured by not allowing any instructions to cross a 32 byte boundary and by
forcing jump and branch instructions to only target addresses at the beginning of these boundaries. These properties makes it easier to validate the instruction stream. [32]

**Safe instructions** Instructions such as `SYSENTER`, `SYSCALL`, and `INT` are disallowed to avoid system function invocations. `RET` is also an illegal instruction to avoid implicit memory addressing. [32] However there is no blacklist of rejected instructions, but instead a whitelist of accepted instructions. [15]

**Control flow integrity** For instance, `JMP` instructions are checked so they are using register addressing and that they are preceded by an `AND` instruction with the value of `0xfffffffff0` and that the target address lies inside of the memory region of the program instructions as mentioned in Reliable disassembly on the previous page. [32] [15]

For further reading, including how the x86, x86_64, and ARM implementations works, see Appendix A.2 on page 46.

**Exceptions**

NaCl does not handle hardware exception recovery, in other words, the ability recover from segmentation faults, floating point exceptions and external interrupts because different operative systems handles these differently. [32]

Software exceptions such as C++ exceptions, are however supported, since they can be implemented in a portable fashion in user space. [32]
3.2.6 Supported Architectures

Since NaCl’s sandbox implementation heavily relies on the underlying architectures hardware it has to be reimplemented rather than just ported when moved to a new architecture. However x86, x86-64 and ARM are already supported. [49]

3.2.7 Portable Native Client

As mentioned earlier NaCl makes it possible to use a plug-in in a web browser independently of OSs, as long as the architecture is the same. One problem NaCl struggles with is that the application needs to be compiled, and tested, for every architecture it wants to target. And those architectures which are not targeted would not be able to run the software. This can be a problem in the case where architecture specific code is used, for instance if assembly optimized routines are used, or if the developer drops support for an architecture. [50]

Another potential problem is if a new architecture is introduced in the future. Then all programs would have to be recompiled and tested for this new architecture, due to the lack of binary for it. [50] See Figure 3.5.

PNaCl makes plug-ins not just platform independent, it makes them architecture independent too. [50]

This portability can be achieved by using a slightly modified LLVM Intermediate Representation (IR) bitcode. By compiling to a portable format the module can be translated by the client to its own architecture. [50]

If a new architecture is introduced nothing other than the runtime needs to be ported to this architecture as the executable is generated from the platform independent LLVM bitcode IR. See Figure 3.6 on the following page. [50]
Figure 3.6: A simplified schematic overview of the steps from source code to running PNaCl program on multiple architectures and how introducing a new architecture is no longer a problem as it was with NaCl (Figure 3.5 on the preceding page).
3.3 Emscripten

The idea of Emscripten is to allow compiling of C and C++ code into JavaScript, which then can be run in a web browser\[51\]. Emscripten is however not a C/C++ to JavaScript compiler but rather a LLVM bitcode IR to JavaScript compiler.\[51\]. This makes it possible to support any language which can be compiled into LLVM bitcode, but for now only C and C++ is supported\[51, 17\]. This makes it possible to use your already available C/C++ code base and instead of using a GCC, Clang, or similar compiler suite to produce an executable binary, you use the drop-in replacement compiler wrapper commands emcc\[51, 52]\.

The emcc commands uses Clang internally to generate the LLVM bitcode from the C and C++ source files. The output of emcc is, depending of output mode, either a JavaScript file (default), a LLVM bitcode object file (if -c is used), or a HTML file (if the output name, specified with -o, has an .html postfix)\[51, 52]\.

In the final stage, when emcc finally outputs JavaScript code, it is not just the LLVM bitcode translated, but also the necessary runtime system and support libraries.

Since modern JavaScript environments already applies Just in Time (JIT) optimization to JavaScript programs both the runtime and program gets optimized for maximum performance while running\[51\].

Another interesting aspect of having both the runtime and compiled code in JavaScript is that there is no gap between them, everything lives inside the same scope. And because of that, it is possible to both call Emscripten generated code from normal JavaScript code and the other way around. An Emscripten generated function are just like any other JavaScript function.\[51, 53\]

This makes it possible to create a C library that is used in JavaScript code but it also makes it possible to use JavaScript libraries inside C code compiled into JavaScript.\[53\]

3.3.1 Libraries

Since it is possible to call JavaScript from translated C functions, as mentioned above, it is possible to create C and C++ wrappers/shims for JavaScript libraries. This has made it possible to add OpenGL functionality by using a shim on top of WebGL\[54\] and a limited Simple Directmedia Library (SDL) API by using JavaScript events, canvas operations etc.\[55\].

As soon as a new web technology emerges it is possible for a developer to create a wrapper to make use of these features in their C and C++ code base.

3.3.2 Thread Restrictions

Since everything is compiled into JavaScript all C and C++ libraries are limited to what features JavaScript supports. And since JavaScript does not have any support for threads with shared state, an language translated to JavaScript cannot either. Hence the lack of Pthreads or similar.\[51\]
3.3.3 Real World Applications

Emscripten has been used successfully to port C and C++ applications in a number of applications and libraries to the web. Some of the projects are:

**OpenTTD**  Open source transport strategy game modeled after the game Transport Tycoon Deluxe by Chris Sawyer.[56]

**Broadway**  A JavaScript version of Android’s H.264 decoder, compiled with Emscripten and optimized both with Google’s Javascript closure compiler and by hand.[57]

**Doom**  Classic first person shooter game by id Software. Shared on the Mozilla blog but later removed due to a Digital Millennium Copyright Act (DMCA) take down request due to original doom shareware files included.[58, 59]

**Ammo.js**  A port of the 3D physics engine library Bullet.[60]

3.4 JavaScript Rewrite

Another approach, if one wants to move an application, is also to rewrite it in JavaScript manually. Some projects that has been rewritten are;

**bullet.js**  Manually converted version of JBullet which is a Java Conversion of the open source 3d physics engine Bullet.[53]

**Emberwind**  Platform 2d game by Time Trap, written in C++ and available for desktop computers and iOS devices and now also HTML5 browsers. It has both a canvas2d and WebGL rendering backend[61, 62]. Available at: http://emberwind.com/

3.5 Performance

The performance between the technologies described in this thesis has previously been studied by Webber in December 2011 where he created a test suite around Box2D[20]. He describes in his results that NaCl was about 25% of the performance of the native version and that the compiled-to-JavaScript versions was about between 5% and 10% of the native performance.

3.6 Comparisons

All technologies has both advantages and drawbacks, Table 3.2 on the next page tries to summarize the feature scope and how they relate.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Plug-in</th>
<th>NaCl</th>
<th>Emscripten</th>
<th>Rewrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share code between JS and C/C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Secure, no worries for customer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>No installation need</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Share state threads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Works with all major browsers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Works with all major platforms</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Within 50% performance of native</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3.2: Feature matrix of the different technologies
Chapter 4

Case study

In order to try how it is, or rather; if it is, possible to port a large legacy real-time project to the web I got the opportunity to port the 2D physics sandbox and learning tool Algodoo.

There was a few constrains set by the company, Algoryx AB, for the end product:

- The user should not need to install anything other than a web browser.
- The end product should be able to share as much code base possible with the desktop version.
- The product should work on at least Windows and Mac OS X

Due to these limitations the only options left, was to either use NaCl or Emscripten. However since Emscripten does not have support for threads and would be significantly slower than the desktop version only NaCl was an option in the end.

The porting process we choose to work with was quite straight forward, some might even call it naïve, but it has been proven to work for me in previous projects. The process is described in Figure 4.1 on the following page.

4.1 Introduction

In a nutshell, Algodoo is:

- closed sourced proprietary piece of software
- 150k+ lines of C++ code\textsuperscript{1}
- 2D physics simulation, using both iterative and direct solvers.
- dependent of the following libraries:

\textbf{AgX Sabre} The direct solver in Algoryx’ industrial 3D physics simulator AgX.

\textsuperscript{1}Counted using David A. Wheeler’s ‘SLOCCount’
Start

Get a "Hello World" program to work to learn the basics about how the platform and toolchain works

Create a build file for the new platform

Try to compile

Did it work?

Yes

Try to link

Did it link?

Yes

Try to run the program

Any bugs found?

Yes

Fix the bugs

No

Any features left out?

Yes

Add one of the features

No

Fix whatever it failed on first. Remove any features which might not be needed at this time if it helps.

Port the missing library using the same scheme

Done

Figure 4.1: Flowchart of the porting process.
COLAMD Linear algebra library, which helps AgX direct solver to create sparser matrices.

OpenSceneGraph Header dependencies on the OpenThread library in OpenSceneGraph, which is used by AgX.

boost Library for extending C++, used by both Algodoo and AgX Sabre, mostly for its function object wrappers and file system access.

pango, cairo, glib Text rendering.

libzip Decompression of compressed scene files.

iconv Conversion between wide and non wide strings.

Qt Multi platform application framework, used for OpenGL context creation, event handling, file handling, image decoding.

4.2 Removed Features

Some of the original features where removed and other simplified in order to speed up the development of a working prototype.

HTML/CSS rendering GUI components Some of the UI elements are rendered using Qt’s QWebWidget, but since we tried to remove all Qt dependencies we had to choose; either port WebKit, rework the UI to make use of the browser’s own rendering engine, or ignore the components in this prototype. Since reworking the UI was out of scope for this project and bringing in WebKit into the project would be a bit awkward and redundant, we just took the easy way out and put the web UI components aside for the time.

Text rendering All text rendering was skipped to avoid dependency on the libraries cairo and pango. Both these libraries are however available ported and included in naclports.

Joystick Even though an API for gamepads input exist we choose not to implement joystick/gamepad support.

std::wstring to std::string conversion Instead of using iconv, a dummy implementation was created.

4.3 Determine Progress

One problem, when working with both an unknown platform and code base is that it is hard to both show and estimate the progress of the porting. However, even though nothing can be shown to work during the progress, it is possible to do educated guesses on the progress by looking at how many files that has been compiled without errors in relation to how many that failed. This can either be done by comparing number of C/C++ files with number of object files, or perhaps even better, if either gcc or clang is used, by passing the flag -DM to the compiler. This compiler flag tells the compiler to output dependencies for each compiled file. The output files can then be analyzed to show which files has been successfully compiled.
4.4 Porting issues

The porting process has not been without issues though, some of the issues are listed in the sections below.

4.4.1 Choosing Host Development Operating System

Apparently the SDK is not as platform agnostic as the plug-ins it creates. Although the NaCl SDK is available for Windows, Mac, and Linux does not mean that it is a breeze to work with it on all platforms.

Both Mac and Linux works quite well, when the right configuration environment variables had been found\textsuperscript{2}. Making it work on Windows was much more difficult and even though it would have yielded a setup which would have enabled me to build both a desktop version and a NaCl version side by side, it was abandoned.

The differences are shown available in Table 4.1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Windows x86_64</th>
<th>Linux x86_64</th>
<th>Mac x86_64</th>
<th>Mac x86_32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome Available</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NaCl binary support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Debugger</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Live view of stdout</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Got NaClports Working</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Algodoo Build Available</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hardware Available</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: Availability depending on architecture.

In the end, we ended up using a 64 bit Linux distribution as host operating system even though that choice meant that it would not be possible to build both desktop and NaCl version of Algodoo on the same computer.

4.4.2 Build configuration

Algodoo’s uses CMake as build system, however, no one had tried to build it for any other target architecture than the host architecture. This caused problems since CMake tried to look for libraries in the standard library search paths, causing it to identify the wrong architecture build for libraries. After a quick reviewing, the standard build system, for the desktop build, was scrapped in favor for the recommended by Google, build system SCons for building NaCl modules\textsuperscript{3}.

4.4.3 Detecting x86-64 Architecture

The library AgX which is used for the direct solver used the void pointer type size in order to detect if it is compiled for a 32 bit or 64 bit architecture. Since

\textsuperscript{2} Environment variable setup changed somewhere around pepper17 and is not the same when using naclports

\textsuperscript{3} The recommended build system has changed to use Makefiles since the project started.
NaCl only has a 32 bit address space even for 64 bit application this test fail to work.

4.4.4 Rendering

Algodoo is made to be backward compatible with older computer and graphic cards and was therefore using a traditional fixed function pipeline OpenGL 1.4 with extensions where available. This was however not compatible with the Graphics 3D API in NaCl as that is based on OpenGL ES 2.0, which have deprecated the old pipeline in favor for the new shader based one. The OpenGL API mismatch led to the need of an updated rendering system which was kindly contributed by the lead developer of Algodoo, Emil Ernerfeldt.

The updated rendering system was not made with performance in mind, but rather as a quick-and-dirty fix of the lack of renderer. A new rendering system is in the works though.

4.4.5 Qt

Algodoo has been utilizing Qt since version 2.0 for setting up OpenGL context, web GUI components, input events, string handling and for file IO. At the time when this project was started the Qt NaCl port was determined too experimental to use and it was therefore decided that Algodoo’s Qt portions would be back ported to SDL and boost as it was in pre-2.0.

4.4.6 Environment Detection

In order for Algodoo to support multiple OSs, each OS has their own implementation of a common OS class. Since no NaCl class existed, for obvious reasons, a new one had to be created. Due to the similarities between programming for NaCl and for GNU/Linux, the NaCl OS backend implementation was based on the old GNU/Linux implementation (pre-Qt). However this implementation assumed either `getenv("LANG")` or `getenv("LC_ALL")` to return a valid pointer to a string. This worked fine in GNU/Linux but crashed the application in Windows. A simple check of the return values for the `getenv` calls before they where used solved the issue.

4.4.7 File Handling

As mentioned earlier all functions are non-blocking in the PPAPI. This was not how Algodoo was designed. Instead of redesigning the resource management handling from the ground up we decided to use the POSIX shim nacl-mounts from naclports.

By using nacl-mounts it was possible to use standard blocking unix calls for file handling with the only implication of moving the program off the main thread as nacl-mounts requires.

The move off the main thread caused however some problems since all OpenGL calls needs to be called from the main thread due to the PPAPI thread restriction mentioned in Section 3.2.3 on page 10. To fix this we decided to preload all graphic resources to memory before starting the simulation.
4.4.8 Main Loop

As many other games and real-time applications Algodoo makes use of a main loop which is called from the main function which is the entry point after the program has loaded. For each iteration of the main loop the user inputs are polled, the simulation is stepped one time step, and one frame is rendered.

PPAPI however assumes that the program is event driven which means that when the program, or in NaCl lingo: module, loads or an input event occurs a function is called. The difference might seem to be subtle but there is a big difference, which part of the code is in charge.

In order to convert Algodoo into an event based application the main loop, which is visualized in Figure 4.2, had to be split up.

![Figure 4.2: Algodoo’s basic program flow.](image)

The whole main function got converted into four functions:

**InitializeAlgodoo** Initialization code to setup paths, load resources etc.

**PreRun** Setup timers, calibrate joysticks etc.

**Step** Pump events, update GUI, step simulation. Might load files using nacl-mounts, cannot be run in main thread.

**Draw** Draw simulation and GUI. Uses PPAPI’s 3D Graphics function calls, must be run in main thread.

**PostRun** Unload all resources. Uses nacl-mounts functions.

By refactoring the code into this structure both the NaCl version and desktop version could coexist in the same code base. See Figures 4.1 and 4.2.
Listing 4.1: Desktop main function implementation.

```plaintext
main:
call InitializeAlgodoo
call PreRun
  while ShouldContinue:
    call Step
    call Draw
    call PostRun
```

Listing 4.2: NaCl initialization implementation.

```plaintext
ModuleLoaded: // called from mainthread
  start thread(newthread)

newthread: // called from newthread
  call InitializeAlgodoo
  while ShouldContinue:
    call Step
    call drawframe on mainthread
    wait until Draw is finished
    call PostRun

drawframe: // called from mainthread
  call Draw

InputEventHandler: // called on mainthread
  push events on queue
```

### 4.4.9 In-line Assembly

One of the dependencies of Algodoo had in-line assembly which included non-white listed instructions for the x86-32 architecture. Instead of porting these routines to valid NaCl instructions we decided to only support 64 bit for now.

### 4.5 Some notes on the SDK

Even though it is entirely possible to port a game or application to NaCl the lack of tools and documentation makes it really hard.

#### 4.5.1 Documentation

A real concern regarding NaCl development is its documentation, or rather lack of documentation. As documentation traditional is the first place to look for when a problem arises at the horizon, it is more than frustrating to work with NaCl. I have found myself, more than once, reading NaCl’s source code, in order to understand how to use the Pepper API.

When I used Java back in the days I was amazed by how well it was documented. The same applies to Apples documentation. However this is not the
4.5.2 Debuggers and Profilers

By the 6th of March 2012 an experimental debugger for NaCl modules was released. This version is known to be unstable and only working on the 64 bit versions of Windows and Linux.

I had some issues stepping over printf which caused the debugger to crash. Another issue was that the debugger could not show where a segmentation fault or similar exception occurred.

These problems made the debugger unusable in my point of view and debugging was made using stdout tracing instead. A tracing routine for C++ (with an example) can be found in Appendix B.1 on page 47.

4.5.3 Naclports

Even though naclports is not officially a part of the SDK it is essential for the porting process as it adds libraries like boost, libz for compression and nacl-mounts to simulate POSIX file system functions.

As mentioned, nacl-mounts is essential when doing a port of a project which uses synchronous file system access, however there are no manual or example other than source code patches for existing projects using it. The recommended place to learn how to use the piece of software works are from reading the NaCl patch and for the open source project “Battle for Wesnoth”. However that patch is for an old version of nacl-mounts, but there is a patch to update the patch which can be studied to learn about how to use the library.[63].

4.5.4 Some Final Notes

The technology is incredible, but without documentation and working tools it is not for the moment worth taking the time. This makes the whole platform feeling just like a prototype that might be scrapped, rather then the future of web technology.
Chapter 5

Results

The results of this thesis is a working, but limited, prototype of Algodoo running in the Chrome web browser and the relative efficiency results between the desktop and the NaCl version of Algodoo.

5.1 Algodoo - Native Client Edition

Besides the lack of text rendering, encoding conversion, Web GUI components and a stable input handling routine, this version of Algodoo is working. A screenshot of the NaCl version of Algodoo can be found in Figure 5.1.

Figure 5.1: Algodoo running inside Chrome on GNU/Linux using Native Client.
5.2 Performance

Comparing the whole Algodoo, NaCl vs. native, performance would be irrelevant, due to the changes which had to be done for the rendering engine. Therefore only the physics simulation was profiled and compared.

All tests are run on a freshly booted computer in either Windows 7 or GNU/Linux Kubuntu 10.10. Chrome version 21 has been used for the NaCl test runs and a pre released version of Algodoo, 2.0.2, has been used for the native tests.

Algodoo is currently only available as a 32 bit application while the NaCl version only exist as a 64 bit application, as discussed in Section 4.4.9 on page 25. Even though it would be neat in the general case to compare native 32 bit with NaCl 32 bit, and native 64 bit with NaCl 64 bit we can not and we actually should not in this case. Most computers are 64 bit today, and comparing the old 32 bit native Algodoo with a 64 bit NaCl version gives us an insight of what would happen if such a move was done today.

5.2.1 NaCl Version Setup

Due to the current main thread dependence of PPAPI (as described in Section 3.2.3 on page 10) and that logic/simulation is done in another thread, due to blocking file IO (described in Section 3.2.4 on page 11), the profiling code and the simulation had to be done in different threads. This was solved by timing the execution from where the module just have rendered one frame to the next. This should give a biased approximation of the time needed for the simulator.

The bias will exist due to the time it takes to record the current time and what other events are occurring in the web browser and OS. The bias can be removed by sample the time it takes for 10 steps and calculate the difference and divide that result with the number of extra simulation step.

Both Windows and GNU/Linux is tested to see if there are any performance differences between the OSes.

5.2.2 Native Version Setup

Even though Algodoo already includes a profiler we choose to time the step function manually instead. This is done by using Algodoo’s internal scripting language Thyme to get the current time, step simulation, get new time, etc. just like what had to be done manually in the source code for the NaCl version. See Listing 5.1 on the next page for the Thyme script used for the data gathering.

5.2.3 Test Scene

A test scene was created in order to have something to simulate. This scene consists of a pair of motors, a stack blocks of glass which a laser is pointed at, some springs, and finally some water. See Figure 5.2 on the facing page.

No ropes or wires where included in the test scene as only a debug build of the direct solver library (AgX Sabre) was available when doing the tests.
Listing 5.1: Script used to time the physics simulation in the test scene.

```javascript
// need gui to update so don’t paste both into the console at once
res = []; for (50, (n) => { res = res ++ [timeSteps(100), "\n"] }); System.writeFile("native.dat", string.list2str(res)); Scene.loadScene("test.phz");
```

Figure 5.2: Screenshot of the test scene running inside of Algodoo.

5.2.4 Test Results

Plots of the test result data are available in Figures 5.3 on the next page and 5.5 on page 31 and box plots of the same data are available in Figures 5.4 on the next page and 5.4 on the following page.

In the test results we can see that running the NaCl version in both Windows and GNU/Linux does not yield the same performance. The NaCl version runs in Windows at 84.2% the speed of what the same version runs in GNU/Linux. See Figure 5.7 on page 32.

The test runs also shows indications that the NaCl version might be faster (12.9% faster) than the native version. Even though this might be surprising at first, keep in mind that the native version is a 32 bit program and that the NaCl version is 64 bit with 32 bit pointers.
Figure 5.3: Wall time of number of seconds it takes to simulate 10 time steps. The banding visible in the figures are due to the low precision default timer used in windows.

Figure 5.4: Box plot of the data found in Figure 5.3.
Figure 5.5: Wall time of number of seconds it takes to simulate 100 time steps. The banding visible in the figures are due to the low precision default timer used in windows.

Figure 5.6: Box plot of the data found in Figure 5.5.
Figure 5.7: Average wall time for simulating 1 time step after overhead is removed.
Chapter 6

Conclusions

The goal was to both gather information about how different technologies work, which makes it possible to port legacy programs into the web and which would be the most suited for porting Algodoo and write a case study about the porting process.

We have covered much of the basics and touched upon the different technologies strengths and weaknesses. Even though the conclusion for which technology to use in this project was NaCl does not mean it is somehow the best technology for every project. It just suited the needs even though limiting the user base to chrome users could be argued.

The resulting NaCl Algodoo prototype should not in any way be interpreted as a finished product but rather as a pre-alpha or tech preview used to determine if it is a possible direction in the future for such applications.

Even though the current NaCl Algodoo port is crippled in many ways, it does show us that it is not just possible, it is a rather interesting approach, both in terms of technology and performance.

The porting process could however have gone smoother. Earlier reports of porting a full product in a few days, might be true if you know your NaCl environment and code base to port, but if you do not, you could sit a few weeks just to make it compile.

If an iOS or Android version would have existed it would have taken a lot less time, since those platforms are more similar to the NaCl platform.

6.1 Future Work

Concerning the port, it would be of greatest interest to get pango-cairo and iconv working in order to enable text rendering and conversion and of course add support for 32 bit. Another problem is the input handling. It is somehow still not working as intended, due to some threading bug. If this could be ironed out, either by finding the bug or waiting until PPAPI calls can be made on background threads, then the program would not spontaneously crash in the middle of execution.

Concerning the NaCl environment; it would be really appreciated to have the PPAPI and naclmonuts documented with examples and better tools for debugging and profiling code.
Chapter 7

Analysis

As can be seen in Chapter 2 on page 5, there was some criteria which needed to be fulfilled in order for this project to be a success.

A feature comparison between technologies A feature comparison was made and is available in Section 3.6 on page 16.

A performance comparison between technologies A performance comparison is available in Section 3.6 on page 16.

A ported real world application Algodoo has been ported to NaCl. The process has been discussed in Chapter 3 on page 7.

A performance result for ported application In the tests that has been made it clearly shows that the technology is fast enough to handle Algodoo in our case where we compare the performance of the current 32 bit version against the 64 bit NaCl version.

Although the ported application is not ready for consumers yet, this thesis has shown that such a port is indeed possible. Since the ported application works, and all the goals has been met, the project can be determined as a success.
Bibliography


b.blogspot.se/2012/04/zombie-track-meat-post-mortem.html (visited
2012-09-10), April 2012.

view/feature/175210/the_ins_and_outs_of_native_client.php (vis-
ited 2012-09-10), August 2012.

chromium.org/2012/07/npapi-plug-ins-in-windows-8-metro-mode.
html (visited 2012-07-03), July 2012.

nacl-sfi-model-on-x86-64-systems (visited 2012-08-05), August
2012.

[16] ChromiumTeam. Native client - getting started: Background and
getting-started-background-and-basics (visited 2012-09-10), August
2012.

com/kripken/emscripten/wiki/ (visited 2012-09-10), August 2012.

Web_Store.pdf (visited 2012-09-10), December 2011.

b.blogspot.se/2012/02/box2djs-box2d-on-web-is-getting-faster.
html (visited 2012-09-10), February 2012.

j15r.com/2011/12/for-those-unfamiliar-with-it-box2d-is.html (vis-
ted 2012-09-10), December 2011.

se/books?id=SpFTAAAAAYAAJ.

2012-09-10), May 1996.

[23] Microsoft. Description of internet explorer support for netscape-style

com/articles/view/the-opera-plug-in-interface/ (visited 2012-09-
10), January 2008.

ted 2012-09-10), September 2012.


Appendices
Appendix A

Further Reading

A.1 Native Client (NaCl) - Case Studies

Listed below are a set of case studies for different ports projects to NaCl which might be of interest.

Multiple Arcade Machine Emulator (MAME) As the name suggests, the object of the project is to emulate multiple arcade machines. The emulator is open source and supports a wide range of classic and less known arcade games.

The case study highlights problems such as Portable Operating System Interface (POSIX) file operations, newlib-glibc incompatibilities, building software where target system ≠ host system, and problems with projects using self modifying code.

The case study can be found at: https://developers.google.com/native-client/community/porting/MAME

XaoS The XaoS project is an open source fractal viewer/zoomer and can display fractals such as Mandelbrot, Barnley, Newton fractals etc.

The case study highlights how to deal with a main-function, events, and screen updates.

The case study can be found at: https://developers.google.com/native-client/community/porting/xaos

AirMech AirMech is a game which merges both the action and the strategy genera into one fast pace 3D game.

The case study highlights how they migrated from a Direct3D renderer to the OpenGL ES 2.0 based renderer NaCl requires.

The case study can be found at: http://carbongames.com/2012/01/Native-Client/

Zombie Track Meat Zombie Track Meat is a track and field type zombie game.

The case study highlights the porting process from an iOS developer’s point of view, debugging capabilities, and file handling.
The case study can be found at:
http://fuzzycube.blogspot.se/2012/04/zombie-track-meat-post-mortem.html

**Go Home Dinosaurs**  Go Home Dinosaurs is a tower defence game where you, as a mammal, defend your juicy stakes from evil dinosaurs.

This article is not perhaps so much a case study as a guide on how to get started, including tips and trix, written by a developer after porting one of his games to NaCl.

Some of the topics highlighted by the author are: *how to get started*, *debugging*, and *threading*.

The article can be found at:

### A.2 NaCl - Security Models

The original implementations of the security model in native client are described in the following papers:
and:

However, the implementations might change slightly in the future which is why it is also recommended to read the articles on:
http://www.chromium.org/nativeclient

### A.3 NaCl - Posting Binary Messages

An example of how to post binary messages can be found in the example file _histogram_ in the NaCl Software Development Kit (SDK). The example is also available at:
Appendix B

Examples

B.1 Tracer

The following code demonstrates a tracing routine in C++:

```
Listing B.1: trace.h
1 #include <stdio.h>
2 #define TRACE fprintf(stdout, "Trace: %s: %d : %s\n",
3     __FILE__, __LINE__, __PRETTY_FUNCTION__); fflush(stdout)
```

```
Listing B.2: example.cpp
1 #include "trace.h"
2 #ifdef __native_client__
3 # include "ppapi/c/instance.h"
4 # include "ppapi/c/module.h"
5 #endif
6
7 class testclass {
8 public:
9     static void mfunc() {
10     TRACE;
11 }
12 }
13 void testfunc() {
14     TRACE;
15     testclass::mfunc();
16 }
17
18 #ifdef __native_client__
19 int main() {
20     testfunc();
21 }
22 #else
23 class ExampleInstance : public pp::Instance {
24     public:
25         explicit ExampleInstance(PP_Instance instance) : pp::
26             ::Instance(instance) {
```
```cpp
    testfunc();
    }
virtual ~ExampleInstance() {}

    class ExampleModule : public pp::Module {
      public:
        virtual pp::Instance* CreateInstance(PP_Instance ←
          instance) {
          return new ExampleInstance(instance);
        }
    }

    namespace pp {
      Module* CreateModule() {
        return new ExampleModule();
      }
    } #endif
```

Listing B.3: output

1 Trace: example.cpp: 15 : void testfunc()
2 Trace: example.cpp: 11 : static void testclass::mfunc()