Lodz University of Technology

Faculty of Technical Physics, Information Technology and Applied Mathematics

Institute of Information Technology

Dariusz Jędrzejczak, 173095

Optimizing Links, a functional language that compiles to JavaScript, for computer games

Bachelor of Engineering Thesis
written under supervision of
dr inż. Jan Stolarek

Łódź 2015

Abstract

This thesis explores the subjects of web game programming in the functional paradigm and programming language optimization. A few simple web games are implemented in the the experimental functional programming language Links. Optimizations are introduced to the language to improve performance. A benchmark application is implemented to quantify their effectiveness.

Dedication

Rodzicom.

Acknowledgements

Sincere thank you for help, advice, support, patience and understanding to everybody who was on this ride with me.

In particular: James Cheney, Sam Lindley, and Jan Stolarek.

Also my family, friends and girlfriend, all the gamedev and functional programming communities of the Internet and everybody whom I forgot to mention.

Contents

Contents **vii**

1 Introduction **1**
1.1 Scope . . . 1
1.2 Choice of topic . . . 2
1.3 Existing solutions and literature . . . 2
1.4 The future of functional programming in game development . . . 3
1.5 Challenges . . . 3
1.6 Thesis goals and contributions . . . 4

2 Tools and methods **5**
2.1 Programming languages . . . 5
2.1.1 Links . . . 5
2.1.2 JavaScript . . . 7
2.1.3 OCaml . . . 7
2.1.4 Haskell . . . 8
2.2 Software . . . 8
2.3 Programming paradigms and methods . . . 9

3 Background **11**
3.1 Functional programming in game development . . . 11
3.1.1 Potential advantages . . . 11
3.1.2 Issues and disadvantages . . . 13
3.1.3 Adoption of the paradigm . . . 13
3.1.4 Examples . . . 14
3.2 JavaScript games and garbage collector . . . 15
3.3 Links' compiler . . . 15
3.3.1 The `setTimeout` function . . . 15
3.3.2 Significant source files . . . 16
3.4 Links' runtime system . . . 16
3.4.1 Concurrency in Links . . . 16
3.4.2 Passing messages to processes . . . 17
3.4.3 Lists . . . 20
3.4.4 Equality . . . 20
3.4.5 Debugging . . . 21

4 Web game design and implementation 23
4.1 Implemented web games . . . 23
4.1.1 Benchmark . . . 27
4.2 Requirements and design . . . 28
4.3 Anatomy of a game . . . 30
4.3.1 Auxiliary functions . . . 31
4.3.2 Type definitions . . . 31
4.3.3 The `main` function . . . 32
4.3.4 Updating game state . . . 32
4.3.5 Restarting the game . . . 34
4.3.6 Main game loop . . . 35
4.3.7 Game logic . . . 37
4.3.8 Rendering . . . 39
4.3.9 Web page structure . . . 45

5 Optimizations and benchmarking 47
5.1 Initial notes . . . 47
5.2 The benchmark application . . . 47
5.3 The unoptimized version . . . 52
5.4 Basic optimizations . . . 53
5.4.1 Optimized `_yield` and `_yieldCont` . . . 53
5.4.2 Faster `setTimeout` . . . 54
5.4.3 Increased `_yieldGranularity` . . . 56
5.4.4 Turning off double buffering . . . 57
5.4.5 First two optimizations combined . . . 58
5.4.6 First three optimizations combined . . . 59
5.4.7 First two optimizations without double buffering . . . 60
5.4.8 Other simple optimizations . . . 61
5.4.9 Debugging . . . 62
5.4.10 The garbage problem . . . 63
5.5 Advanced optimizations and profiling . . . 65
5.5.1 Profiling . . . 65
5.5.2 Baseline frame rate for remaining optimizations . . . 69
5.5.3 JavaScript optimizer . . . 70
5.5.4 Comparison with the native version . . . 72
5.5.5 Execution time profiling . . . 75
5.5.6 Linked list type . . . 76
5.5.7 Linked list type with native `take` and `drop` . . . 78
5.5.8 JavaScript linked lists . . . 82
5.5.9 JS lists with `null` . . . 84
5.5.10 Equality . . . 85
5.5.11 `_yield` . . . 88
5.6 Observations and summary . . . 94
5.7 Performance in games . . . 96

6 Summary and conclusions **101**
6.1 Conclusions 103
6.2 Future work 104

Bibliography **105**

Glossary **107**

Acronyms **109**

A Attached files **111**
A.1 DVD 111
A.2 Files used in optimizations and benchmarking 112

List of Figures **115**

Chapter 1

Introduction

1.1 Scope

The topics discussed in this thesis are:

- The application of functional programming in game development and related performance issues.
- Trends and predictions pertaining to the adoption of the functional paradigm in game development.
- Web game development in JavaScript and HTML5 and related performance considerations.
- Languages compiled to JavaScript.

In practice I combine these topics by using Links[1] – an experimental functional language compiled to JavaScript – to write several web games. In the process, I benchmark the performance of the language and introduce optimizations to its runtime system to improve it.

The kind of optimizations that I will be talking about are mostly at the level of the programming language's runtime system [2].

This thesis investigates the practical application of functional programming languages and performance challenges that come with the choice of the functional paradigm for developing computer games.

I examine performance by writing computer games of increasing complexity and looking at the frame rate that can be achieved. I then introduce and test the effectiveness of various optimizations to the language's runtime system and compiler. The goal is to achieve a satisfactory frame rate (ideally about 60 frames per second) in studied cases.

My specialization is Computer Simulation and Games Technology and I look at the problem from a game developer's standpoint, but the conclusions of this thesis may apply to any performance-intensive applications of functional programming languages.

[1] `http://groups.inf.ed.ac.uk/links/`

1.2 Choice of topic

I picked this topic, because I see big potential in functional languages and I predict that the languages used in video game industry will evolve towards being more functional as they incorporate more and more elements characteristic of this paradigm. I also think that it is worth to invest time and resources into working with a language that compiles to JavaScript – and also with JavaScript – as this area is rapidly developing and has a promising future.

I was interested in the functional paradigm, design and implementation of programming languages and obviously game development for a long time, so doing this project was a good opportunity to combine together and learn more about these topics.

I intend to explore the subjects of this thesis further and consider the choices I made a good starting point. An experimental and self-contained language like Links allowed me to observe closely the process of creating a programming language.

1.3 Existing solutions and literature

There are many languages that compile to JavaScript[2] and new ones are constantly being created. Also compilers, translators and similar tools for existing languages are being developed.

Some of the languages feature the idea of tierlessness (code in them is compiled for both client and server) and are functional like Links – Opa[3], Ur[4], Haste[5] (which is a dialect of Haskell).

Elm[6] is a functional language intended for web programming that supports functional reactive programming (FRP), which is a paradigm that could potentially be useful for developing games. The use of FRP in games has been explored in [8].

Other functional languages compiled to JavaScript include OCaml, Haskell, LiveScript, Khepri, Roy, Agda, Idris.

Worth mention also are other interesting languages that compile or translate to JavaScript: Haxe[7] (which can also be compiled to many other languages, such as ActionScript, C++, Java or Python), ClojureScript[8] (from Clojure[9]), CoffeeScript[10] (JavaScript with prettier syntax) and a large family of languages derived from it[11].

[2] https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js

[3] http://opalang.org/

[4] http://impredicative.com/ur/

[5] http://haste-lang.org/

[6] http://elm-lang.org/

[7] http://haxe.org/

[8] https://github.com/clojure/clojurescript

[9] http://clojure.org/

[10] http://coffeescript.org/

[11] https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS#coffeescript-family--friends

It is also possible to generate JavaScript from existing languages, like C/C++ (via Emscripten[12]), Ruby (via Opal[13]) or PHP (via Uniter[14]).

Functional game programming is a developing field and the interest in creating games with this paradigm is increasing[15], but it is still a niche area. My sources are mostly internet-based as there are not (yet) many books on topics I touch on in this thesis, which are largely of experimental and research nature.

1.4 The future of functional programming in game development

A few prominent figures in the video game industry have discussed the use of functional programming in the field.

Tim Sweeney, the founder of Epic Games[16], describes "the next mainstream language" – a potential future programming language for game development with functional features – in [18]. Co-founder of id Software[17] John Carmack talks about the benefits of functional programming in game development [5, 6, 7].

Many other game programmers and authors have expressed their interest and in the subject as well [3, 11].

From these discussions and observations we can predict that mainstream programming languages (not only those used in game development) will move towards being more functional as more and more functional features will be gradually introduced into them.

In the nearest future we should see a more widespread adoption of the functional paradigm in existing languages [5] both in terms of style of writing code and in language features as new versions of languages emerge.

1.5 Challenges

The work that I did in the scope of this thesis certainly exercised my engineering skills. I had to put together a few broad theoretical subjects, some new to me. I had to add whatever was missing to the knowledge acquired with formal and informal learning in order to complete the goals I set for myself. Applying this knowledge in practice and achieving tangible results was quite a task.

Approaching the subject of the use of functional programming in game creation was particularly challenging as, compared to other game development-related topics, there is not very much material (especially practical) available about this one.

[12] `http://kripken.github.io/emscripten-site/`
[13] `http://opalrb.org/`
[14] `http://asmblah.github.io/uniter/`
[15] See Chapter 3, 3.1.4 for examples of games written in functional languages.
[16] `http://epicgames.com/`
[17] `http://www.idsoftware.com`

Despite of some research being done in the area for a long time, it is still in an early stage of development (see Chapter 3, 3.1.3).

A more concrete challenge was working with an experimental language (Links) with almost no documentation available, which required me to reverse engineer whatever I needed to know in order to implement optimizations.

1.6 Thesis goals and contributions

The goals of this thesis are as follows:

- Optimize the runtime system and possibly the compiler of the Links language in order to achieve a satisfactory frame rate of at least 30 FPS in simple web games.
- Decrease the performance gap between applications written in Links and their native JavaScript versions.
- Create several computer games in Links using the functional paradigm; each of greater complexity than the previous.
- Analyze and describe the process and challenges of game development using the functional approach.

My contribution to game development in functional languages is exploring its use and effectiveness – specifically in web-based games – by trying to assess and improve performance of a functional language compiled to JavaScript.

In Chapter 3 I provide the necessary theoretical background, describing game development in functional languages and the significant elements of the Links language.

The practical side of this contribution is writing web games – described in Chapter 4 – and improving the performance of the language in order to make them playable (Chapter 5[18]). I summarize the results and conclude in Chapter 6. The technologies, techniques and tools that I used are described in Chapter 2.

Appendix A includes the description of all significant files that were used for performance benchmarking and optimization as well as describes the rest of the contents of the DVD attached to this thesis.

[18]The basis of this chapter is the documentation that I created while working on my traineeship at the University of Edinburgh. Available for download here:
`https://github.com/links-lang/links/blob/dariusz/documentation/performance3.pdf`.

Chapter 2

Tools and methods

This Chapter briefly describes the tools that I used in my work: programming languages, software and the functional paradigm.

2.1 Programming languages

2.1.1 Links

I used the Links programming language[1], which is an experimental language for web programming. It was created and is being developed by a team of researchers led by Philip Wadler at the University of Edinburgh [9].

My goals in choosing this language:

- Explore in practice how functional languages and compilers work internally.
- Contribute to developing the language.
- Research functional programming in game development.
- Learn more about JavaScript – another language I am interested in – as this the main target language of the Links' compiler.

Links is being actively developed and some of its features changed as I was working on my project. I learned how to use it from the basic documentation [1] and the help of researchers who develop it. It has a curly brace-based syntax and features typical for functional languages as well as some features characteristic of itself. I describe these below.

The core concept of Links is that it combines the functionality of languages used for programming three main components of a web application: server side, client side, and database. This is typically done using different languages, such as Java, JavaScript and SQL.

The Links compiler takes care of generating code for each of the three tiers producing a complete Asynchronous JavaScript and XML (AJAX) application.

[1] `http://groups.inf.ed.ac.uk/links/`

The client side part of an application in Links is compiled to JavaScript. This part was my focus.

Links features strong static typing and strict (eager) evaluation. It provides mechanisms for concurrency based somewhat on those found in the Erlang language[2]: we can create threads (called *processes*) and communicate with them via message passing.

The syntax of the language can be seen on the listing below. It is a source code of a simple client-side application that adds and removes items from a todo list[3]:

```
fun remove(item, items) {
  switch (items) {
     case []     -> []
     case x::xs -> if (item == x) xs
                   else x::remove(item, xs)
  }
}
fun todo(items) client {
   <html>
    <body>
     <form l:onsubmit=
        "{replaceDocument(todo(item::items))}">
        <input l:name="item"/>
        <button type="submit">Add item</button>
     </form>
     <table>
      {for (item <- items)
        <tr><td>{stringToXml(item)}</td>
            <td><form l:onsubmit=
                  "{replaceDocument(
                      todo(remove(item,items)))}">
                 <button type="submit">Completed</button>
                </form>
            </td>
        </tr>}
      </table>
     </body>
   </html>
}

page
 <#>{todo(["add items to todo list"])}</#>
```

[2]http://www.erlang.org/doc/getting_started/conc_prog.html
https://github.com/links-lang/links/blob/sessions/README

[3]Taken from one of the official examples:
http://groups.inf.ed.ac.uk/links/examplessrc/todo.links

The syntax is a little C-like, with curly braces and familiar looking function definitions and invocations. Some typical functional constructs look more familiar to an imperative programmer as they are hidden under syntax sugar (`for` loops are list comprehensions, `switch-case` blocks are pattern matching). We see that we can easily use XML (HTML) in the source code, thanks to XML quasiquotes[4].

2.1.2 JavaScript

JavaScript is an integral part of the Open Web Platform[5]. It is the "lingua franca of the web"[6]. It is one of the most popular – if not the most popular[7] – programming languages on the web. It has been even dubbed the "Assembly of the Web"[8]. It is a part of all modern web browsers, which means that nearly every Internet user has it available. It is used, among other things, for writing client-side asynchronous applications.

It is multi-paradigm, having object-oriented (prototype-based), imperative and functional features.

The development and adoption of the HTML5 standard makes JavaScript or languages that compile to it the perfect choice for developing computer games on the web[9].

My optimizations to Links' runtime involved working a lot with JavaScript. The following were written by me in JavaScript:

- Optimized versions of existing functions and methods.
- Interfaces for JavaScript functions that were called from Links. These include functions for manipulating HTML5 canvas element, functions for manipulating linked lists, specialized equality functions and some debugging functions.
- A native version of the benchmark application for comparison with the Links' version.

2.1.3 OCaml

The Links' compiler is written in the OCaml language. OCaml[10] supports multiple programming paradigms – primarily functional and object-oriented. The language

[4] `http://groups.inf.ed.ac.uk/links/quick-help.html#xml_quasiquotes`
Also see Chapter 4, 4.3.9 for a brief description.
[5] `http://www.w3.org/wiki/Open_Web_Platform`
`http://www.webplatform.org/`
`http://en.wikipedia.org/wiki/Open_Web_Platform`
[6] `http://blog.codinghorror.com/javascript-the-lingua-franca-of-the-web/`
[7] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
January Headline: JavaScript programming language of 2014!
[8] `http://www.hanselman.com/blog/JavaScriptIsWebAssemblyLanguageAndThatsOK.aspx`
[9] `https://developer.mozilla.org/en-US/docs/Games/Introduction_to_HTML5_Game_Gevelopment_%28summary%29`
[10] `http://ocaml.org/`

has a static type system with advanced type inference. It is used in many applications[11]. Its features make it well-suited for writing compilers[12].

One of my optimizations (see Chapter 5, 5.5.11) required a slight change in the JavaScript generated by the compiler. This required me to learn the basics of the OCaml language.

2.1.4 Haskell

Haskell[13] is one of the most popular functional programming languages. It was my choice for learning the basics of the functional paradigm [14]. It is a purely functional language with lazy evaluation and static strong typing.

I ported a lot of code for the first three games that I developed from Haskell. I also ported some functions from its standard library (see Chapter 5, 4.3.1, 4.1).

2.2 Software

For writing applications in Links I used Geany[14] – a text editor with basic Integrated Development Environment (IDE) capabilities. No existing IDEs have support (like syntax coloring) for the Links programming language.

To run Links applications on my computer, which runs the Arch Linux operating system, I set up an environment according to official install instructions[15]. I built Links' interpreter from source code using the recommended version (4.01) of the OCaml compiler.

In order to test the applications I installed the XAMPP[16] package, which provides an Apache HTTP Server[17].

The Links project uses GitHub[18] for hosting source code. I worked on my own branch[19] using the Git[20] command line tool.

I used three different web browsers to compare performance and garbage collectors' behaviour between them: Chromium 36.0.1985.143, Opera 25.0.1583.1 and Firefox 30.0.

No external libraries were used in my applications as none are available for the Links language.

[11] https://ocaml.org/learn/companies.html
[12] http://flint.cs.yale.edu/cs421/case-for-ml.html
[13] https://www.haskell.org/haskellwiki/Haskell
[14] http://www.geany.org/
[15] https://github.com/links-lang/links/blob/sessions/INSTALL
[16] https://www.apachefriends.org/index.html
[17] http://httpd.apache.org/
[18] https://github.com/
[19] https://github.com/links-lang/links/tree/dariusz
[20] http://git-scm.com/

2.3 Programming paradigms and methods

Aside from exploring optimization possibilities for the Links language, the main focus of this thesis was to assess the viability of developing games using the functional paradigm[21].

Functions in the functional paradigm are defined in the same way as in mathematics – their output values depend only on the arguments that are input to the them. The other main point of functional programming is avoiding the use of mutable data and state changes. The paradigm is declarative, focusing on specifying the *what* rather than the *how*, and its principal constructs are expressions rather than statements as it tries to minimize or eliminate side effects. These properties allow for easier formal analysis of programs, which means reasoning about their behavior.

In terms of abstraction, functional languages belong to the category of (very) high level languages. They manage memory automatically, with the help of some sort of a garbage collector.

There are dynamically as well as statically typed functional languages. In statically typed languages such as Haskell, OCaml or Links an important feature is type inference, which facilitates writing code and makes it much less cluttered.

Static strong typing, like in Links, means that for the price of sometimes harder to write code, essentially all type errors are caught by the compiler at compile time. This feature helped me immensely with debugging the applications that I wrote in Links as there is no debugging tools available for this language.

Programs written in the functional style are usually shorter than imperative ones. Functional programming, due to its high level of abstraction and declarativeness, allows for writing more expressive and shorter (than imperative) code.

[21] http://en.wikipedia.org/wiki/Functional_programming
http://c2.com/cgi/wiki?FunctionalProgramming

Chapter 3

Background

3.1 Functional programming in game development

Functional programming is not widely used in game development, but it has potential to become significant in this field. In recent years it has been gaining interest and popularity in the game development industry.

I will attempt to describe the functional paradigm from a game developer's perspective, heavily referencing John Carmack's keynote speech [6, 7] as I think that in it he presents a very good summary of arguments for and against functional programming in game development. And does so from a very practical, experience-based perspective.

He did research on the use of Haskell in game development by porting the game Wolfenstein 3D[1] to the language. In the keynote he describes his findings and conclusions that he came to whilst developing this project, as well as functional paradigm and its pros and cons in general.

3.1.1 Potential advantages

Carmack states that the "brutal purity" of the Haskell language and static strong typing enforce writing code that is shorter, cleaner, less bug-prone and better to maintain in the long term compared to imperative code. He says that using a functional language like Haskell in game development could potentially make programming much easier.

If we define a spectrum of functional purity, then a typical imperative language (such as C) would be considered almost completely impure and a typical purely functional language, such as Haskell would be considered almost completely pure. The end on this spectrum that mainstream languages appear to be moving towards is purity. Some proponents have stronger positions on purity (Carmack), some weaker [12].

This adoption of functional features and then move towards purity should increase with changes and improvements in hardware, most of which is currently

[1] `http://en.wikipedia.org/wiki/Wolfenstein_3D`

much better suited to work with imperative languages [4].

According to Tim Sweeney, a major advantage of functional languages over imperative ones in the future is related to them being better in handling heavy parallelism and multithreading (see [18], slide 61).

On the issue of static vs dynamic typing Carmack's stance is very much leaning towards the strong static approach. He argues that even though sometimes writing statically typed code can be not particularly comfortable as the programmer needs to "build up a type scaffolding for something that should be really easy", it provides very significant benefits.

Carmack, referencing his experience in supervising and programming big game projects, states:

> Everything that is syntactically legal, that the compiler will accept, will eventually wind up in the code base.

This is one of the reasons why static typing and static analysis of code can be very valuable as they restrict programmers, preventing and picking up on a lot of bugs at the compiler-level.

In [5] Carmack talks more about functional programming in game development. He argues for the use of functional paradigm even in non-functional languages (precisely C++). A few excerpts from this article also provide a good overview of the advantages and disadvantages of FP in game programming:

> A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are greatly amplified, almost to the point of panic if you are paying attention. Programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible.
>
> I do believe that there is real value in pursuing functional programming, but it would be irresponsible to exhort everyone to abandon their C++ compilers and start coding in Lisp, Haskell, or, to be blunt, any other fringe language...

He states:

> No matter what language you work in, programming in a functional style provides benefits.

3.1.2 Issues and disadvantages

Because of the fact that functional data structures are immutable and the paradigm avoids changing state, a typical functional-style application does a lot more copying and is much more memory-intensive than a typical imperative application. This and other principles of the functional paradigm (like avoiding side effects) can carry a significant performance overhead compared to imperative solutions.

In the aforementioned keynote speech [6, 7] Carmack also mentions the downsides and issues of FP and discusses possible solutions to them. One of the issues is the performance impact of the garbage collector, which, he says, is manageable provided that the overhead introduced by the GC is fixed. And from developer's perspective, automatic memory management makes programming easier.

He describes a way of dealing with state changes and side effects in a pure functional language, which is based on an event-passing mechanism. Carmack states that in general it is preferable to minimize the use of such mechanisms in imperative languages as they "decouple the flow of control". But in case of functional programming "that's the only way to do effects". And in the Haskell language such approach turns out, as he describes, "really clean", thanks to the feature of partial function application (which is fairly common in functional languages).

Besides the above, because of the paradigm still being largely experimental and not very popular in game development, there is a problem with lack of good libraries for making games. This means that it is harder for a beginner programmer to start programming.

3.1.3 Adoption of the paradigm

When it comes to video games, the reasons for slow adoption of new, higher-level concepts and languages have to do a lot with performance. In game development, especially mainstream (complex, AAA games), one of the main points of focus is trying to stay on the cutting edge of performance, which means writing games so that they meet strict performance requirements. (Video games can be considered to be soft-realtime systems [13].)

This explains why the assembly language was used in game development for a long time [17] and higher-level languages were adopted more slowly than in less performance-intensive applications.

As the computational capabilities and complexity increases, more abstract and higher-level solutions are preferred. Developing a modern AAA game in Assembly would take an unreasonable amount of time and effort. So in exchange for expressive power and ability to operate on higher levels of abstraction some performance sacrifices are necessary and accepted.

Functional languages are on a very high level of abstraction, so extrapolating the aforementioned trend it should be the direction we will be going in. And indeed, as we can observe and conclude from what I stated so far in this chapter, it seems to be.

3.1.4 Examples

One of the most notable functional languages that was used for game development is Haskell. Games that were written in Haskell include:

- Nikki and the Robots, that was sold on Steam[2],
- Frag[3], a first-person shooter, implemented to investigate the use of functional reactive programming in games [8],
- Raincat[4], a game developed by Carnegie Mellon[5] students,
- Super Nario Bros.[6], a clone of Nintendo's classic

as well as various bigger and smaller games or game-related projects created by Haskell community[7].

Creating games with Haskell is becoming easier as wrappers, frameworks and engines are being developed. For example:

- Bullet[8], a wrapper for the Bullet[9] physics engine,
- LambdaCubeEngine[10], a Domain Specific Language for 3D graphics,
- IrrHaskell[11], a wrapper for the Irrlicht[12] game engine,
- haskell-game[13], which provides "a suite of libraries for covering all sorts of aspects of game development",
- Helm engine[14] (inspired by the Elm[15] language), which allows programming using functional reactive programming. Yampa[16], used in [8], is a similar project.

[2] http://steamcommunity.com/sharedfiles/filedetails/?id=107105028
[3] https://www.haskell.org/haskellwiki/Frag
[4] http://bysusanlin.com/raincat
[5] http://www.cmu.edu/index.shtml
[6] https://www.youtube.com/watch?v=gVLFGQGRsDw
[7] http://hackage.haskell.org/packages/#cat:Game
https://www.haskell.org/haskellwiki/Applications_and_libraries/Games
http://keera.co.uk/blog/2014/11/24/haskell-android-games-adventure-engine-beta-testing/
[8] http://hackage.haskell.org/package/bullet
[9] http://bulletphysics.org/wordpress/
[10] https://www.haskell.org/haskellwiki/LambdaCubeEngine
[11] http://hackage.haskell.org/package/IrrHaskell
[12] http://irrlicht.sourceforge.net/
[13] https://github.com/haskell-game
[14] http://helm-engine.org/
[15] http://elm-lang.org/
[16] https://www.haskell.org/haskellwiki/Yampa

Other functional languages, such as F#[17], or the languages from Lisp family[18] are also being used in game development.

3.2 JavaScript games and garbage collector

The main issue impacting performance in JavaScript game development is the slowdowns caused by the garbage collector[19].

In order to alleviate this problem, games written in JavaScript have to manage the memory usage in a way that generates as little garbage as possible.

In case of a language that compiles to JavaScript, like Links, it is also important that the runtime system generates as little overhead (including garbage) as possible. This is what my optimizations were focused on.

3.3 Links' compiler

Before talking about potential performance bottlenecks in Links' runtime system, I will introduce some background and describe briefly all the elements of Links that were relevant in my optimizations.

Because of Links being an active research project, experimental in nature – there is no documentation available for its compiler or runtime[20]. My descriptions come from reverse engineering of the system.

3.3.1 The `setTimeout` function

The client side part of an application in Links is compiled to JavaScript. The compiler generates JavaScript in continuation-passing style. It uses the `setTimeout` function to support concurrency (see 3.4.1).

`setTimeout` is one of a few timer functions[21] used in JavaScript for asynchronous programming. These are native JavaScript functions that allow delaying of execution of arbitrary instructions. `setTimeout(func, delay)` is usually supplied with two arguments: a callback `func`tion to execute and a `delay` in milliseconds, after which this function will be executed.

[17] `http://fsharp.org/guides/apps-and-games/index.html`

[18] `http://lispgames.org/`
`http://c2.com/cgi/wiki?LispInJakAndDaxter`

[19] `http://www.gamedev.net/page/resources/_/technical/game-programming/writing-fast-javascript-for-games-interactive-applications-r3516`

[20] In fact, there is only basic documentation available for the language [1], which describes its syntax, type system, some features, library functions, and how to run scripts in Links.

[21] `https://developer.mozilla.org/en-US/Add-ons/Code_snippets/Timers`

3.3.2 Significant source files

These two source files of the Links compiler are important for my optimizations:

- *lib.ml* – this file among other things provides the Links compiler with declarations of functions defined in `jslib.js` library (described the next section). I modified it each time I wanted to add an interface to a JavaScript function or define some function natively in JavaScript.
- *irtojs.ml* – this is a source file of the Links compiler that contains the part of it responsible for generating JavaScript code from an intermediate representation. I modified it to implement the optimization described in Chapter 5, 5.5.11.

3.4 Links' runtime system

The main part of Links' runtime system that cooperates with compiled JavaScript applications is contained within the *jslib.js* library. Below I describe briefly some of its significant, from the point of view of my optimizations, components[22].

3.4.1 Concurrency in Links

Links supports concurrency by providing a means of forking the main thread of control into more threads (called processes in Links). This is done using the `spawn` language primitive:

```
var processId = spawn { expression };
```

From Links' documentation [1]:

> This starts a new process which begins by evaluating `expression`. The value of the expression is discarded if evaluation ever completes. `spawn` returns an identifier of the new process to the calling process.

The work of process scheduling is essentially handled by two functions in the Links runtime: `_yield` and `_yieldCont`[23].

To see the JavaScript implementation of `_yield` (`_yieldCount` is almost identical), please refer to Chapter 5, 5.5.11 or to the file *jslib.js* attached to this thesis.

[22]Note: while describing or enumerating various JavaScript or Links functions, depending on whether it is relevant in the context, I use just their name, name and list of arguments, a fragment of definition or a full definition. Sometimes I change the original names of arguments (like `x`, `c` and the like) to more descriptive to improve clarity.

[23]I will later sometimes refer to both of these as `_yield*`.

Here is a brief description of these functions and global variables relevant to them:

- `_yield(func, arguments, continuation)` – has 3 arguments: a `function` in continuation-passing style, a list of `arguments` for this function and the `continuation` that this function will "return" into. It uses `setTimeout` to invoke the `function` asynchronously, immediately when control is available – the second argument to `setTimeout` is 0. The important effect that this causes is clearing of the JavaScript engine's call stack, which prevents an overflow. A stack overflow would be inevitable as continuation-passing style functions instead of returning call continuations, quickly filling the call stack.
- `_yieldCont(continuation, argument)` – takes a `continuation` and an `argument` that will be "returned into" it. Works in the same way as `_yield`, except that this `continuation` is passed to `setTimeout` instead of a function.
- `_yieldCount` – a counter variable that counts calls to `_yield*`. It is incremented every time any of the two `_yield*` functions is invoked.
- `_yieldGranularity` – determines at what value of `_yieldCount` should the call stack be cleared.
- `_current_pid` – stores the identifier of the currently active process.

The concurrency is enabled at the runtime system level thanks to the use of continuations. Upon compilation, invocations of Links functions and returns from them are translated to JavaScript as follows: `_yield` is used to wrap invocations of continuation-passing style functions and `_yieldCont` wraps invocations of continuations in these functions (a continuation in CPS is invoked when a function "returns"). Both `_yield*` functions take care of periodically clearing the call stack and giving up control to any process that might need it by calling `setTimeout`.

The fact that invocations and returns from functions are the majority of a typical Links application means that `_yield` and `_yieldCont` are the most frequently called functions in the compiled (JavaScript) code and they take up the most of the execution time of the application (see Chapter 5, 5.5.5). Because of this it is crucial that their implementation is as optimal as possible – even a slight optimization will have a great impact on the overall performance.

3.4.2 Passing messages to processes

We create a new Links process using the `spawn` primitive, which returns its identifier. We can then use this identifier to pass messages to the process. From Links' documentation [1]:

> This identifier can be used to address messages to the new process, with the `!` primitive (pronounced "send"):
>
> ```
> procId ! msg
> ```

> This appends the value msg to the mailbox for the process identified by `procID`. The return value is just `()`. The mailbox is FIFO, so if you know that some message is sent before some other message, you know they will be received in that order.
>
> Each process' mailbox is given a static type according to the messages it expects to receive.
>
> *[...]*
>
> A process can receive messages using the `recv` function, which returns the next message in the current process' mailbox.
>
> ```
> var nextMsg = recv()
> ```

Before I introduced any optimizations the `spawn` primitive was implemented as follows:

```
function _spawn(f) {
  _maxPid++;
  var childPid = _maxPid;
  _makeMailbox(childPid);
  setTimeout(function () {
               _debug("launched process #" + childPid);
               _current_pid = childPid;
               f(function () {
                 // delete the mailbox when finished
                 delete _mailboxes[childPid];
               })
             }, 0);
  return childPid;
}
```

Where `_maxPid` is the highest process identifier allocated so far. `_makeMailbox` is defined like so:

```
var _mailboxes = {0:[]};

function _makeMailbox(pid) {
  if (!_mailboxes[pid])
    _mailboxes[pid] = [];
}
```

`f` is the expression (zero-argument function) passed to `spawn`. It is scheduled to run as a new "process" using `setTimeout` with second argument of 0 (meaning immediately when the control is available). When it finishes, it calls a continuation that deletes its mailbox.

Before I introduced any optimizations the `!` primitive was defined as follows:

```
function _Send(pid, msg) {
  _debug("sending message '" + msg._label +
         "' to pid " + pid);
  if (!_mailboxes[pid])
    _makeMailbox(pid);
  _mailboxes[pid].unshift(msg);
  _wakeup(pid);
  _debug(pid + ' now has ' + _mailboxes[pid].length +
         ' message(s)');
  _dumpSchedStatus();
  return {};
}
```

The definition of `_wakeup`:

```
var _blocked_procs = {};

function _wakeup(pid) {
  if (_blocked_procs[pid]) {
    _debug("Waking up " + pid);
    var proc = _blocked_procs[pid];
    delete _blocked_procs[pid];
    setTimeout(proc, 0);
  }
}
```

Processes use the `recv` function for receiving messages:

```
function recv(kappa) {
  // (1)
  if (_mailboxes[_current_pid].length > 0) {
    msg = _mailboxes[_current_pid].pop();
    kappa(msg);
  } else {
    var current_pid = _current_pid;
    _block_proc(current_pid,
                function () {
                    _current_pid = current_pid;
                    recv(kappa);
                });
  }
}
```

I removed a few calls to `DEBUG.assert` from the beginning of the definition (at line marked with the `(1)` comment) for brevity. `kappa` is the continuation, which receives the next message in current active process' mailbox.

If there are no messages in the mailbox, the process is blocked until there are:

```
function _block_proc(pid, its_cont) {
  _blocked_procs[pid] = its_cont;
}
```

3.4.3 Lists

jslib.js defines the usual basic list manipulating functions, such as:

```
Concat(xs, ys),
Cons(x, xs),
empty(list),
hd(list), // head
tl(list), // tail
length(list),
take(n, list),
drop(n, list),
max(list),
min(list),
// etc.
```

These are implemented using regular JavaScript arrays as representation for lists. For example:

```
function _hd(list) { return list[0]; }
function _take(n, list) { return list.slice(0, n); }
```

This is certainly not an optimal representation as manipulating JavaScript arrays involves a lot of copying. Using a simple linked list implementation improves performance significantly and gives a better memory footprint (see Chapter 5, 5.5.6, 5.5.8).

3.4.4 Equality

Checking for equality of two values is done using the `LINKS.eq(left, right)` function defined in jslib.js. It compares `left` and `right` side, using dynamic type checking functions like:

```
DEBUG.is_object(value),
DEBUG.is_number(value),
DEBUG.is_string(value),
DEBUG.is_array(value)
```

Most of them utilize the `is_instance` function, defined in the following way:

```
function is_instance(value, type, constructor) {
    return value != undefined
        && (typeof value == type ||
            value instanceof Object &&
            value.constructor == constructor);
}
```

For example, `DEBUG.is_number`, which is defined like so:

```
return is_instance(value, 'number', Number);
```

So `LINKS.eq` checks the type of the compared entities and uses an appropriate method of comparison for these types. For example for an array:

```
// this is a fragment of LINKS.eq definition:
if (DEBUG.is_array(l) && l != null &&
    DEBUG.is_array(r) && r != null) {
    if (l.length != r.length)
        return false;

    for (var i = 0; i < l.length; ++i) {
        if (!LINKS.eq(l[i], r[i]))
            return false;
    }

    return true;
}
```

This way of doing comparison can certainly be optimized at compiler level, taking advantage of all the available static type information and perhaps using specialized equality functions in the compiled JavaScript, which I simulated in one of my optimizations – see Chapter 5, 5.5.10.

3.4.5 Debugging

The main debugging functions are as follows:

- `DEBUG.assert(predicate, message)` – just throws an exception and displays a simple error `message` informing which assertion failed (`predicate` is false).
- `DEBUG.assert_noisy(predicate, message)` – throws an exception and displays more detailed error `message`, prints a stack trace.
- `_debug(message)` – prints `_current_pid` and an error `message`.

All debugging functions will display error messages if the `DEBUGGING` flag is set to true. Otherwise, they won't.

Chapter 4

Web game design and implementation

In order to achieve the goals of this thesis, I developed four web games and a benchmark application in Links, the purpose of which was to test the effectiveness of the optimizations that I introduced to the language's runtime system.

4.1 Implemented web games

I approached writing games in Links by developing projects of increasing complexity and implementational difficulty. The basis for the first three games were their versions written in Haskell[1][2][3]. I started by porting the code from that language, modifying and adjusting it as I went. The last game (based on Pac-Man[4]) was designed and written from scratch.

[1] https://github.com/gregorulm/h2048
[2] https://github.com/RudolfVonKrugstein/jshaskell-blog
[3] https://github.com/isomorphism/lazy-tetrominoes
[4] http://en.wikipedia.org/wiki/Pac-Man

The first game I wrote was a clone of the popular puzzle game 2048[5].

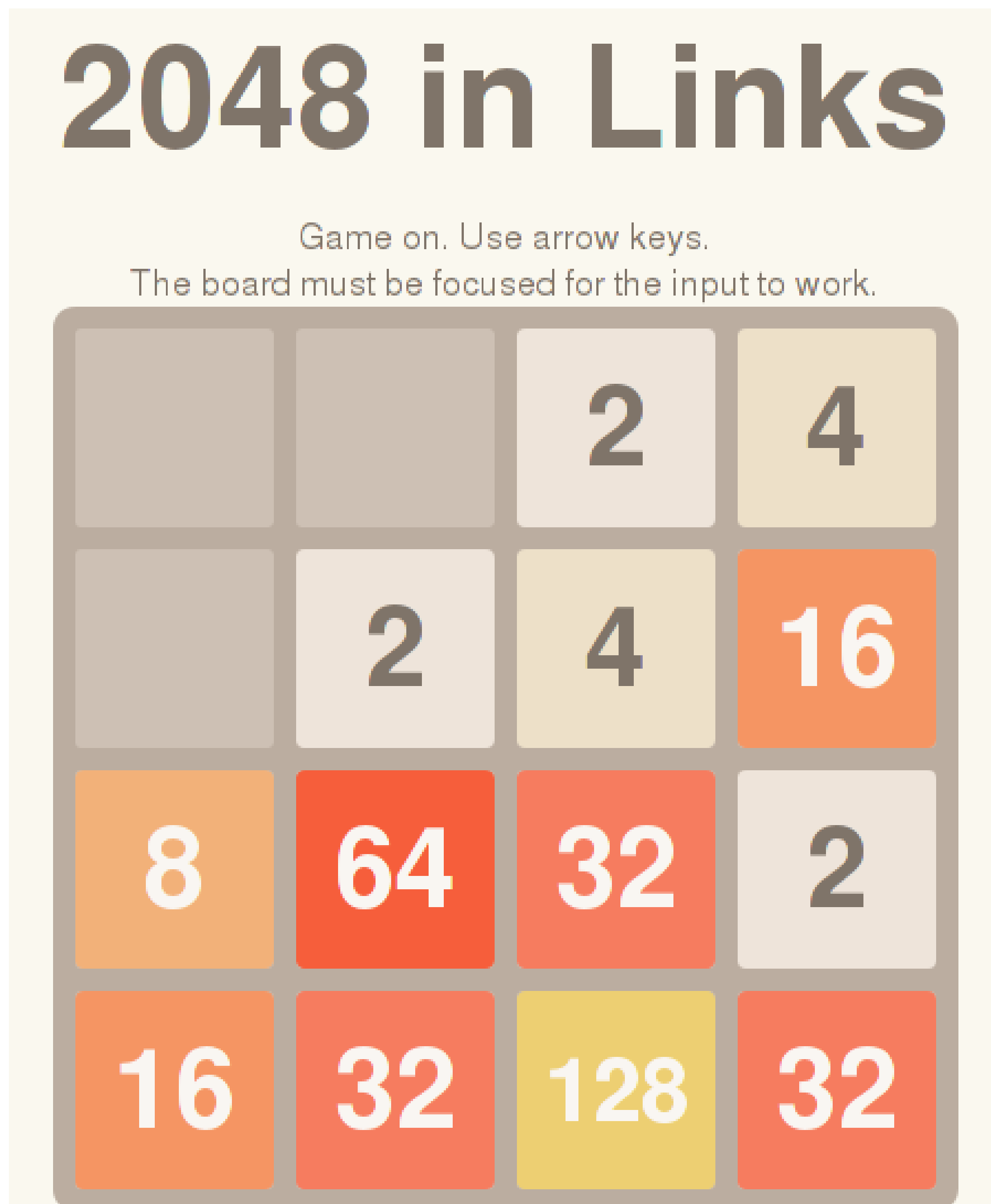


Figure 4.1: The puzzle game 2048 implemented in Links

[5] `http://en.wikipedia.org/wiki/2048_%28video_game%29`

The second was a clone of the classic Breakout[6].

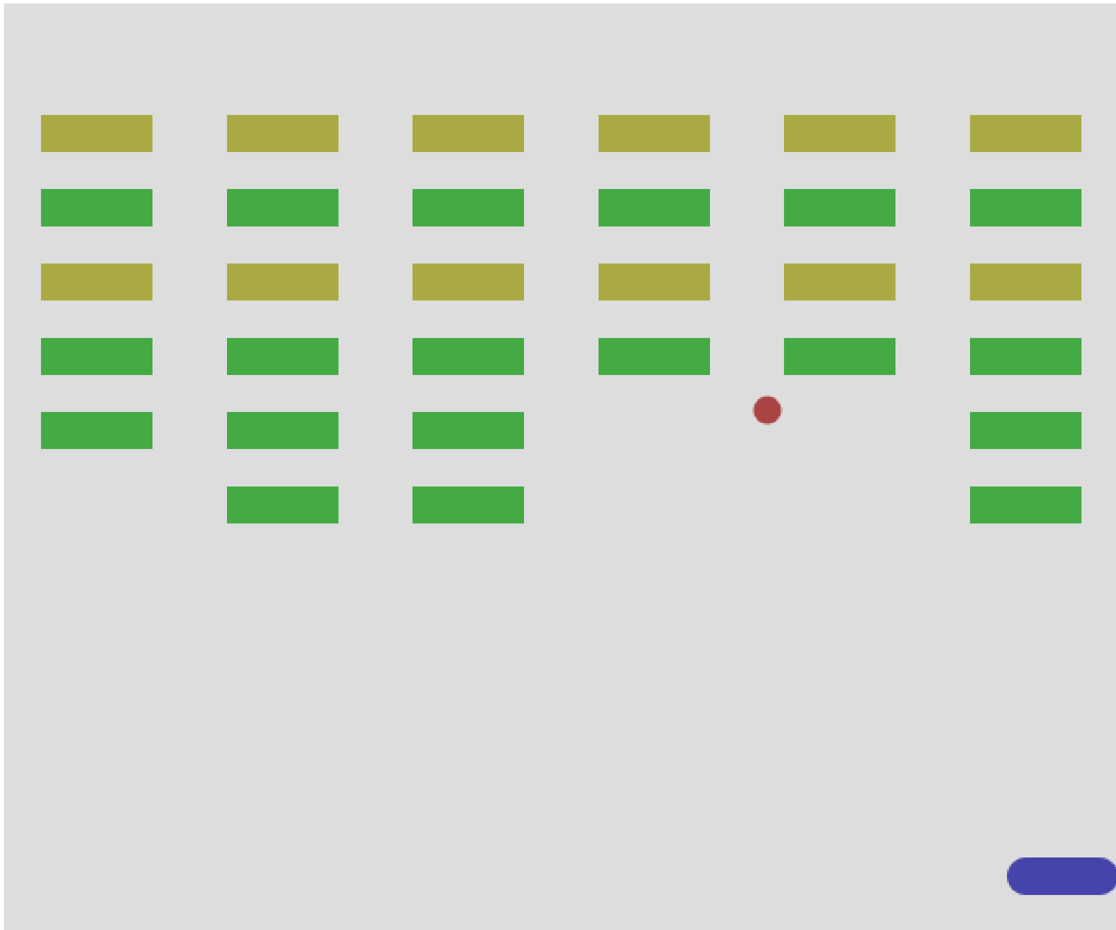


Figure 4.2: A Breakout clone in Links

[6] http://en.wikipedia.org/wiki/Breakout_%28video_game%29

Next I implemented a clone of Tetris[7].

Figure 4.3: Links version of the classic Tetris

[7] http://en.wikipedia.org/wiki/Tetris

Finally I designed and wrote a variation of Pac-Man.

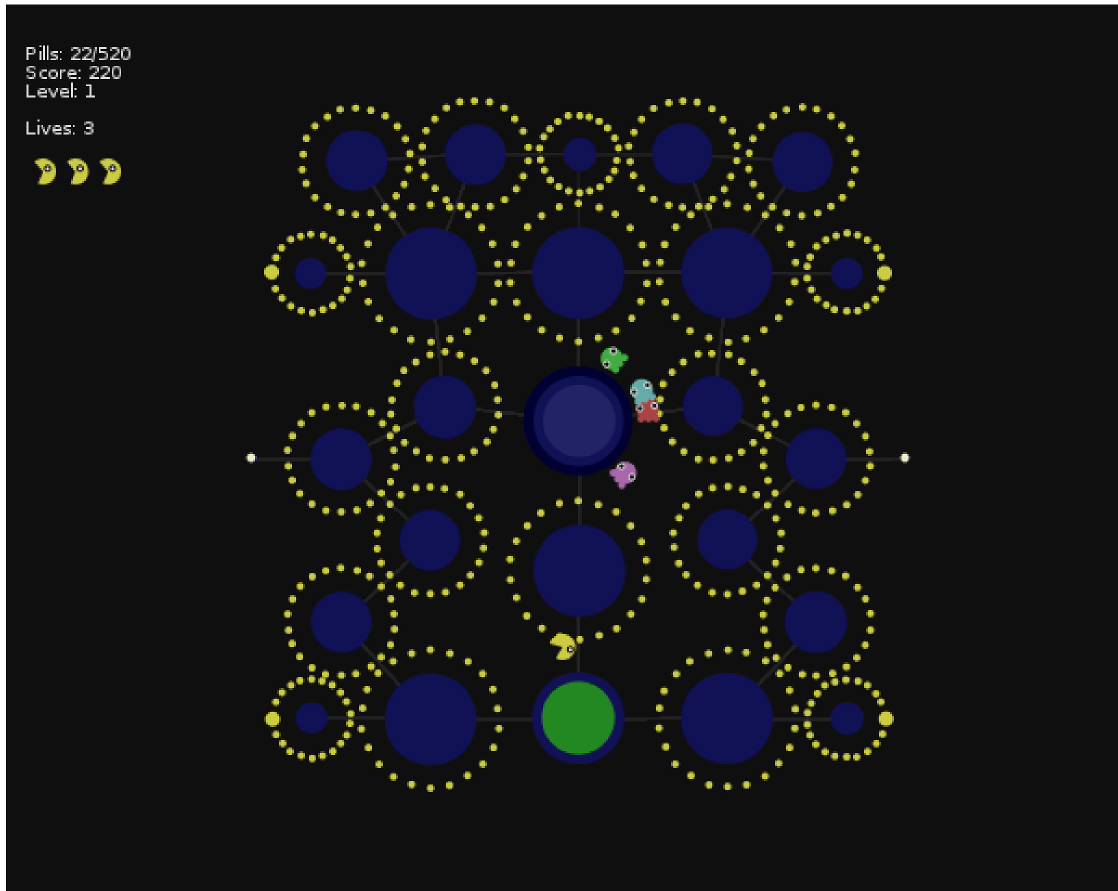


Figure 4.4: My variation of Pac-Man written in Links

In the next section I describe the design and implementation of the game.

4.1.1 Benchmark

While implementing the Breakout clone I wrote a benchmark application[8] and started introducing optimizations to Links' runtime system to improve its performance.

The benchmark is implemented similarly to all of the games. Its purpose is measuring and collecting data about changes of its own frame rate in time and presenting this data on charts.

[8]Described in detail in Chapter 5, 5.2

4.2 Requirements and design

The standard frame rates in modern computer games are 30 or 60 Frames Per Second (FPS)[9].

Two important reasons for such frame rate requirements are[10]:

1. They allow for acceptable responsiveness of player's input.
2. The perceived smoothness of animation is satisfactory for a typical player.

I assumed 60 FPS as my target ideal frame rate and 30 FPS as the absolute minimum. This would be in an average simple web game on average hardware.

The most complex of the games I wrote is a variation of the classic Pac-Man. The computational complexity of Pac-Man in my view represents well the complexity of a typical simple HTML5 web game.

The dimensions of the game area were set to 600x480, which are about the average in web-based (HTML5 or other) games[11].

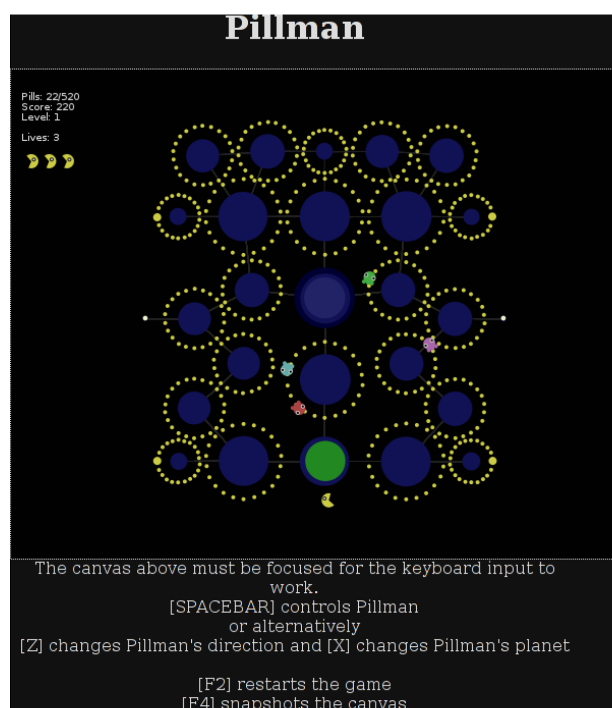


Figure 4.5: A screenshot showing all of the contents of a web page that contains the game

[9] http://www.polygon.com/2014/6/5/5761780/frame-rate-resolution-graphics-primer-ps4-xbox-one

[10] http://gaming.stackexchange.com/questions/25465/why-do-video-game-framerates-need-to-be-so-much-higher-than-tv-and-cinema-framer

[11] http://www.emanueleferonato.com/2009/04/20/the-perfect-size-for-a-flash-game/

The user interface for my games was kept very simple: I placed the title of the game at the very top of the web page, below it the game area and a few simple instructions for the player (explaining controls). The game is initialized when the user clicks on the game area (the canvas).

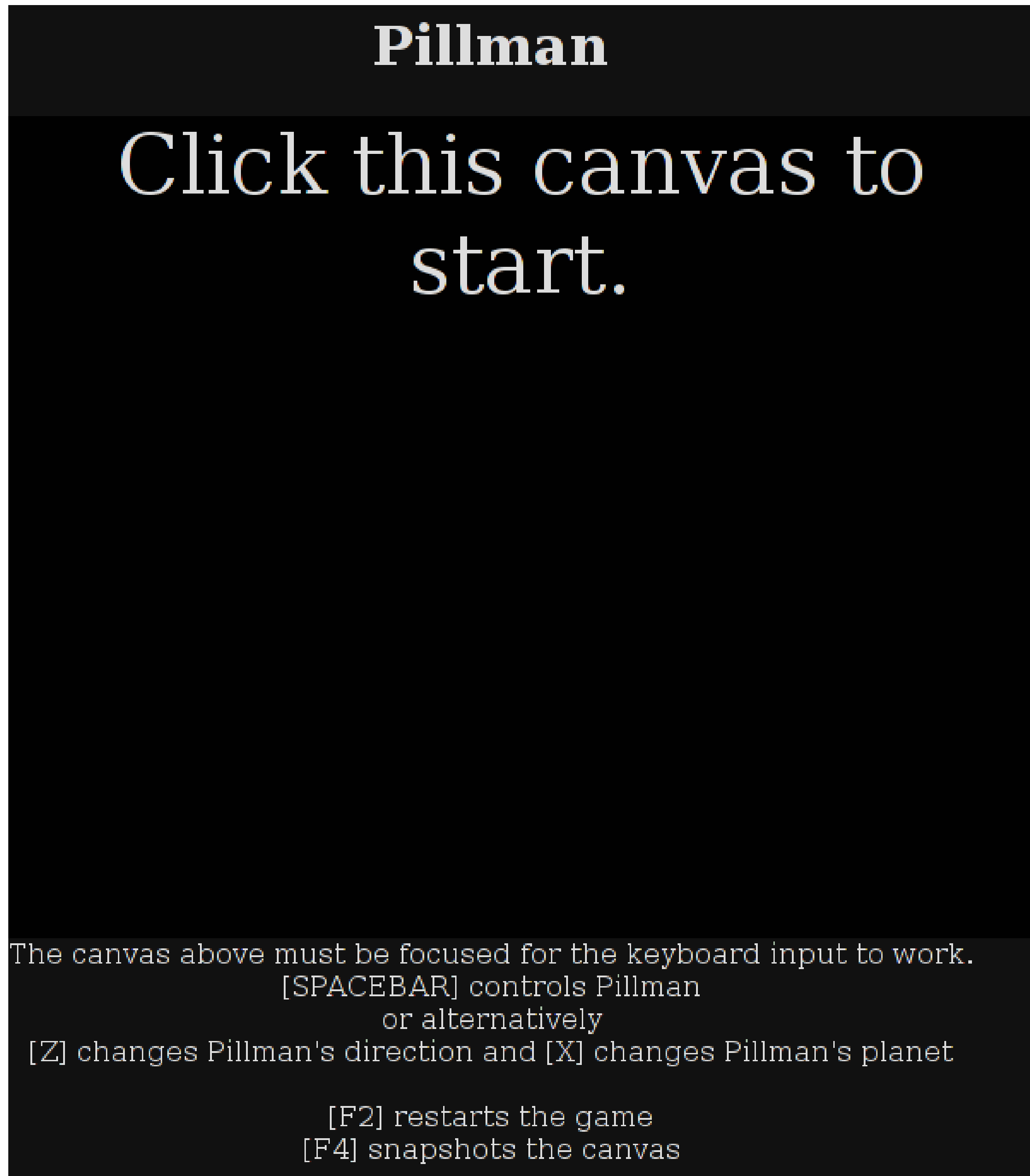


Figure 4.6: What the player sees after launching the game (entering a web page that contains it)

4.3 Anatomy of a game

Implementation-wise all of the applications – including the benchmark – follow similar structure[12]:

1. Auxiliary function definitions.
2. Type definitions.
3. `main` function definition, which includes definitions of:
   (a) values (describing dimensions and appearance of game area and objects, duration times of animations, properties of game objects, game logic, key codes, simulation parameters, etc.),
   (b) game-specific auxiliary functions,
   (c) the main and auxiliary drawing functions,
   (d) main game logic functions (for collision detection, game object manipulation, etc.),
   (e) input handling functions,
   (f) functions for game processes/threads,
   (g) and finally the structure and appearance of the the web page (HTML and CSS) on which the interface of the game is displayed.
4. Invocation of `main()`.

I will now briefly describe elements of this structure[13].

[12]Links so far has no support for modules or a built-in way of including source files, so I decided to use a single source file per application and copy-paste the common parts. I experimented with using a Perl script or sed (`www.gnu.org/software/sed/`) and C preprocessor to do the copy-paste for me, but decided to stick to the manual method. For small size projects this was not particularly inconvenient.

[13]In these descriptions, I will be quoting some Links code – usually with a brief explanation of how it works. In places where syntax or semantics of the language proves not intuitive enough, please refer to the official documentation [1]

4.3.1 Auxiliary functions

While developing applications in Links I defined a small library of general-purpose functions, which includes:

1. Mathematical functions, particularly for manipulating 2D vectors (addition, subtraction, multiplication). These had to have separate versions for operating on arguments of integer and floating point types as Links does not do conversion between those and even has a separate set of operators for floats.

2. Functions ported from Haskell's standard library. As I mentioned, the first games I developed were based on code written in Haskell which used some functions with no direct equivalents available in Links. This required using a different approach or porting the functions from Haskell. The ports were approached pragmatically and sometimes not exact, with the main point being that the application using the functions works as it should. In one case – while implementing the Tetris clone – I wrote a version of `IntMap`[14] that emulated the original's functionality, but did not preserve all of it – my version of `elems`[15] does not sort the elements by key as it was not necessary in the particular case.

3. Functions for manipulating my custom linked list type. One of my optimizations involved implementing a linked list type in JavaScript to replace the list type (based on JavaScript arrays) used by default in Links (see 5.5.6). Most of these were implemented directly in JavaScript, except for a few, which – for simplicity of implementation – were written in Links. These include a faster version of `map`, which only applies a function to all elements of a list and does not return a new list as it is concerned only with the side-effects of the function. This is its definition in Links:

```
fun lsMapIgnore(f, l) {
        if (lsEmpty(l)) ()
        else { var _ = f(lsHead(l));
               lsMapIgnore(f, lsTail(l)) }
}
```

   The names of the list manipulating functions that I implemented all have an `ls-` prefix.

4.3.2 Type definitions

An example of a type or, more precisely, type alias definition in Links[16] is:

```
typename Vector = (Float, Float);
```

[14] https://hackage.haskell.org/package/containers-0.1.0.1/docs/Data-IntMap.html
[15] https://hackage.haskell.org/package/containers-0.1.0.1/docs/Data-IntMap.html#v:elems
[16] http://groups.inf.ed.ac.uk/links/quick-help.html#type_aliases

Here we define a `Vector` type, which is a tuple[17] of two floating point values. After defining a type alias, we can use it instead of writing out the whole type. I describe some type alias definitions in following sections.

4.3.3 The `main` function

In the following sections, I describe in detail how the main loop [10] of my applications works. Most of the Links code that follows is taken from the source of my Pac-Man clone.

When reading the code note that we operate on the game state very explicitly, passing it around and modifying it in various functions.

This is characteristic of the functional approach and in opposition to the imperative way, where the game state is usually implicit and defined by many variables, which are subject to side-effects.

4.3.4 Updating game state

I describe the code in this section in order in which it appears in the original source. It is important, because the code uses processes, which send messages to each other. In such case, Links' compiler requires that the processes that are referred to at some point in the code (by their identifier) are defined before that point. The order of definition of functions that do not refer to processes does not matter.

In order to facilitate reading the descriptions, I marked some lines of code with comments containing an identifier in square brackets (e.g. `[updateState.1]`). I will refer to those identifiers (printed in bold type) in my explanations.

The main function responsible for handling input and updating game state is `updateState`, defined as follows:

```
fun updateState() {
        fun mainLoop(gameState: Game, dt,
                     lastTime, fpsInfo, ii) {
            # [updateState.1]
            # this is the main game loop...
        }

        var _ = recv(); # [updateState.2]
                        # wait for initialize()
        mainLoop(getInitialGameState(), 0.0, clientTime(),
                 initialFpsInfo, []); # [updateState.3]
        if (not(haveMail())) self() ! CarryOn else ();
            # [updateState.4] restart
        updateState() # [updateState.5]
}
```

[17] http://groups.inf.ed.ac.uk/links/quick-help.html#pairs__tuples__and_records

For now I omit the definition of `mainLoop` (**[updateState.1]**). I will describe this function later.

Right after the definition, the `updateState` function is run in a separate process:

```
var updateProcId = spawn { updateState() };
```

I will refer to it as the *update process*. The first instruction it executes is `recv` (**[updateState.2]**), which makes it wait until there is a message in its mailbox. This is because it should not start executing until the user started (initialized) the game by clicking on the canvas element that it uses.

Next, I define two functions for handling input:

```
fun onKeyDown(e) {
        updateProcId ! (KeyDown(getCharCode(e)): Input);
}

fun onKeyUp(e) {
        updateProcId ! (KeyUp(getCharCode(e)): Input);
}
```

They send messages of type `Input` – all messages in Links are statically typed – to the update process. The `Input` type is defined as follows:

```
typename Input = [| KeyUp: Int | KeyDown: Int | CarryOn |];
```

This means that it is a variant type[18], which can be any of the three things: the constant `CarryOn`, an `Int` with a `KeyDown` label, or an `Int` with a `KeyUp` label. The update process can only receive messages of this type.

And then the initialization function, which is invoked when the user wishes to start the game:

```
fun initialize() {
        var _ = recv(); # wait until a message is received

        # [initialize.1]:
        jsSetOnEvent(getNodeById(containerId),
                     "keydown", onKeyDown, true);
        jsSetOnEvent(getNodeById(containerId),
                     "keyup", onKeyUp, true);

        var _ = domSetStyleAttrFromRef(
                    getNodeById("info"),
                    "display", "none");

        updateProcId ! CarryOn
}
```

[18] `http://groups.inf.ed.ac.uk/links/quick-help.html#polymorphic_variants`

Immediately after definition I spawn it as a separate process (the *initialize process*),

```
var initializeProcId = spawn { initialize() };
```

which will receive a message when the user clicks the main canvas element.

`jsSetOnEvent` (**[initialize.1]**) is a wrapper for `addEventListener`[19], which is a JavaScript method for registering event handlers. It is defined as follows:

```
function _jsSetOnEvent(node, event, fn, capture) {
    node.addEventListener(
        event,
        function(e) { fn(e, _idy) }, // [js.1]
        capture
    );
    return;
}
```

The second argument of `addEventListener` must be a callback that takes one argument (the event), but `fn` is a function of two arguments – it is a Links function, and as such uses continuation-passing style – its second argument is its continuation. This is why it is wrapped in an anonymous function (**[js.1]**) in which we pass it an identity continuation, which does nothing:

```
function _idy(x) { return; }
```

Once we have added the event listeners, every time there is a `keyup` or `keydown` event (user presses or releases a key), our `onKeyUp` or `onKeyDown` functions will be called with the event object. They in turn will send a message with the integer character code (obtained from the event using `getCharCode`) wrapped in proper type to the update process, which is responsible for handling input.

The call to `domSetStyleAttrFromRef` causes the message that encourages the user to click on the canvas and start the game to disappear. It will be replaced by the game screen.

The last initialization step is waking up the update process, by sending it the `CarryOn` message. This will start the `mainLoop` (**[updateState.3]**), which will run until user requests to restart the game.

4.3.5 Restarting the game

When `mainLoop` terminates, the next line (**[updateState.4]**) will cause the update process to send a `CarryOn` message to itself if no messages are waiting in its mailbox. Then (**[updateState.5]**) the update process will recurse, going back to **[updateState.2]**. Since it has at least one message waiting in the mailbox, it will carry on to **[updateState.3]**, running the `mainLoop` with its initial arguments again, effectively restarting the game.

[19]`https://developer.mozilla.org/pl/docs/DOM/element.addEventListener`

4.3.6 Main game loop

The `mainLoop` function of the update process (**[updateState.1]**) handles user input (**[mainLoop.1]**), game logic (**[mainLoop.2]**) and rendering (**[mainLoop.3]**).

It is defined as follows:

```
fun mainLoop(gameState: Game, dt, lastTime,
             fpsInfo, inputState) {
    var now = clientTime();
    var dt = dt +.
             fmin(1.0,
                  intToFloat(now - lastTime) /. 1000.0);

    fun receiveInput(inputSoFar: [Input]) {
        if (haveMail()) {
            receiveInput(recv()::inputSoFar)
        } else inputSoFar
    }

    var i = receiveInput(inputState); # [mainLoop.1]

    var (gameStatePrim, dtPrim) =
        updateLogic(dt, gameState, i); # [mainLoop.2]

    if (gameStatePrim.metaState == Restart)
        ()
    else if (floatEq(dtPrim, dt)) {
        # don't redraw if there were no logic updates
        mainLoop(gameStatePrim, dtPrim, now, fpsInfo, i)
    } else {
        mainLoop(
            if (gameStatePrim.metaState == Download)
                (gameStatePrim with metaState = Run)
            else gameStatePrim,
            dtPrim,
            now,
            draw(gameStatePrim, lastTime, now, fpsInfo),
                # [mainLoop.3] draw & get new fpsInfo
            [] # reset input
        )
    }
}
```

Its arguments are:

- `gameState` of type `Game`, which is defined more or less as:

```
typename Game = (pillman: EntityDescription,
                 roger: EntityDescription,
                 # ...
                 pillCount: Int, score: Int,
                 lives: Int, level: Int,
                 state: GameState,
                 metaState: GameMetaState);
```

  This type describes the whole game state – including the state of game objects, which have the type `EntityDescription`, which in turn describes the state of a particular game entity (the player or an enemy) – its position, speed, direction, etc. The `GameState` and `GameMetaState` types are defined as:

```
typename GameMetaState = [| Run | Download |
                            Restart |];
typename GameState = [| On | Eaten | Eaten2 |
                        Ate | Over | NextLevel |
                        NextLevel2 | Won |];
```

  They represent all possible game states.

  `GameMetaState` is more general and it pertains to the state of the whole application. `Run` means that the main loop is being executed normally, in which case the `GameState` is relevant – it represents the state of a singular game play.

- `dt` of type `Float` represents the time (in milliseconds) since the last update of the game logic.

- `lastTime` of type `Float` represents the time (in milliseconds) of the last invocation of `mainLoop`.

- `fpsInfo` is a record that stores data related to calculating frame rate information.

- `inputState` of type `[Input]` is a list of player's inputs waiting to be processed by the game logic.

The game logic updates (**[mainLoop.2]**) with a fixed time step [10]. This is handled in the `updateLogic` function:

```
fun updateLogic(dt, gameState: Game, i) {
        if (dt > step) {
                var gameState =
                    mainGameLogic(gameState, i);

                updateLogic(dt -. step, gameState, [])
        } else (gameState, dt)
}
```

It updates the game state (by calling `mainGameLogic` with the current game state and input) the number of times determined by the time since the last update. `step` represents the value of $\frac{1}{60}$ of a second, enforcing the desired frame rate (60 FPS):

```
var step = 1.0 /. 60.0;
```

4.3.7 Game logic

This is how the `mainGameLogic` function is defined:

```
fun mainGameLogic(gameState: Game, i) {
    var gameState = handleKeys(i, gameState);
        # [mainGameLogic.1]

    # [mainGameLogic.2] helper functions
    fun ghostCollision(ghost) {
        circleCircleCollision(
            (gameState.pillman.position, pillmanR),
            (ghost.position, ghostR))
    }
    # ...

    switch (gameState.state) { # [mainGameLogic.3]
        case On ->
            # all depending on state:

            # ai updating functions...
            # planet changing conditions...
            # direction changing conditions...

            # update entities...
            # handle pill collisions...
            # handle ghost collisions...
            # check level up conditions:
```

```
            var gameState =
                if (gameState.pillCount == pillCount)
                    (gameState with
                        state = NextLevel,
                        timeout = 60)
                else gameState;

            gameState

        case Over ->
            # on game over
            gameState

        # other states...

        case NextLevel2 -> # animate
            # update game state accordingly...

            if (gameState.level > 255)
                # Victory condition
                (gameState with state = Won)
            else gameState

        case _ -> # any other state means do nothing
            gameState
    }
}
```

It takes the current `gameState` and the current `input` state as its arguments and returns a new game state.

Apart from defining some helper functions (**[mainGameLogic.2]**), it processes the user input (**[mainGameLogic.1]**) and is responsible for transitioning the game from one state to the next (**[mainGameLogic.3]**). The main state is `On`, in which the normal game logic is executed: updating game objects (the player and the Artificial Intelligence (AI)) handling collisions, and other transformations of the game state dependent on specific conditions. Other states include the transition between one game level to the next or the state of the game being over.

4.3.8 Rendering

All the graphics in the games I made is rendered on a HTML5 canvas element[20]. I use the basic interface available in JavaScript for manipulating it. In order for it to work in Links, I added all the necessary function "declarations" at the compiler level as for now the language has no working foreign function interface[21] for JavaScript.

Below I present the list of functions mentioned in the previous paragraph:

```
jsSave,
jsRestore,
jsGetContext2D,
jsFillText,
jsCanvasFont,
jsDrawImage,
jsFillRect,
jsFillCircle,
jsBeginPath,
jsClosePath,
jsFill,
jsArc,
jsMoveTo,
jsLineTo,
jsLineWidth,
jsScale,
jsTranslate,
jsStrokeStyle,
jsStroke,
jsSetFillColor,
jsClearRect,
jsCanvasWidth,
jsCanvasHeight,
jsSaveCanvas
```

I will call this my *canvas library* for Links. All these functions are wrappers for JavaScript canvas manipulating methods that operate on context objects. Their names are analogous to those from JavaScript; each has a `js-` prefix.

For example, the function `jsFillRect` is wraps the `fillRect` method of a context object from JavaScript:

```
function _jsFillRect(ctx, x, y, width, height) {
        ctx.fillRect(x, y, width, height);
}
```

[20] http://www.w3schools.com/tags/ref_canvas.asp
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

[21] http://www.c2.com/cgi/wiki?ForeignFunctionInterface

This function is then used in Links as follows:

- First we get the context object for our canvas element (the value of its id attribute is stored in `canvasNodeId` in this example):

  ```
  var ctx = jsGetContext2D(getNodeById(canvasNodeId));
  # ...
  ```

- Then, if we want to draw a rectangle (for example in a rendering function of our game), we pass this object as an argument to our function:

  ```
  jsFillRect(ctx, 0.0, 0.0, 10.0, 10.0);
  ```

So the difference is that we input the context object to the function as the first argument. The rest of the arguments are the same as in the original `fillRect` method[22].

Implementation of all the wrappers in the canvas library can be found in *jslib.js* file (attached to this thesis – see Appendix A).

In the main loop the drawing is done with the `draw` function (**[mainLoop.3]**):

```
fun draw(gameState: Game, lastTime, now, fpsInfo) {
    # prepare canvas
    var (mainCanvas, dispCanvas) =
        if (stringEq(domGetStyleAttrFromRef(
                getNodeById(canvasId), "display"),
                "none") ||
            not(doubleBuffer)) # [draw.5]
            (canvasId, canvas2Id)
        else (canvas2Id, canvasId);
            # [draw.1] double bufferning 1
        var ctx = jsGetContext2D(getNodeById(mainCanvas));

    # define colors
    var pillmanColor = "#cc4";
    var pupilColor = "#111";
    # ...

    # ...

    # drawing functions, depend on state
    fun drawPillman(pillman, gameState) {
        # [draw.3]
    }
    # ...
```

[22] http://www.w3schools.com/tags/canvas_fillrect.asp

```
    #
    # DRAWING PART
    #

    var drawOffset = (x = 108.0, y = 24.0);
        jsSave(ctx);
        jsTranslate(ctx, drawOffset.x, drawOffset.y);
    # draw the game area...
    # draw the connections between planets...
        # lsMapIgnore over connections...
    # highlight the connection
        # which pillman collides with...
    # draw ghost planet bottom layer...
    # draw the planets...
    # draw ghost planet top layer...
    # draw pills...
    # don't draw the rest if game over...
        # highlight the current planet...
        # draw Pillman...
        # draw ghosts...
        # draw portals...
    jsRestore(ctx);

    # draw the HUD
    jsSetFillColor(ctx, textColor);
    # ...
    jsFillText(ctx, "Score: " ^^
               intToString(gameState.score), 10.0, 40.0);
    # ...

    # calculate and draw new fpsInfo:
    var dFps = 1000.0 /.
               (intToFloat(now - lastTime) +. 1.0);
    jsSetFillColor(ctx, textColor);
    var fpsInfo = drawFps(ctx, fpsInfo, dFps); # debug

    # double buffering
    if (doubleBuffer) # [draw.2] double buffering 2
        swapBuffers(mainCanvas, dispCanvas)
    else ();

    # [draw.4] save canvas to file...
    fpsInfo # return
}
```

It inputs the following arguments:

- `gameState` of type `Game` – current game state.
- `lastTime` of type `Float` represents the time (in milliseconds) of the last invocation of `mainLoop`.
- `now` of type `Float` represents the time (in milliseconds) of the current invocation of `mainLoop`.
- `fpsInfo` is a record that stores data related to calculating frame rate information.

Double buffering

Double buffering is a technique commonly used in computer (game) graphics which, among other things, reduces or eliminates flickering and other drawing artifacts [16].

In my applications it is implemented with two canvas elements serving as buffers. Both canvases are displayed at the same position on the screen, but only one is visible at a time. The drawing is done on the invisible canvas. After it is completed, the visibility of the canvases is swapped – the visible one is hidden and the invisible one is displayed[23].

At the beginning of the `draw` function, the canvas to draw on is determined (**[draw.1]**) by checking its `display` attribute. The swapping of the buffers is done after all drawing by the `swapBuffers` function (**[draw.2]**):

```
fun swapBuffers(mainCanvas, dispCanvas) {
    var ctx = jsGetContext2D(getNodeById(dispCanvas));
    jsDrawImage(ctx, getNodeById(mainCanvas), 0.0, 0.0);
    var _ = domSetStyleAttrFromRef(getNodeById(mainCanvas),
                                   "display", "block");
    var _ = domSetStyleAttrFromRef(getNodeById(dispCanvas),
                                   "display", "none");
    clear(ctx)
}
```

The `doubleBuffer` flag (**[draw.2]**, **[draw.5]**) controls whether double buffering is on or off.

`draw` defines some auxiliary functions for drawing the game objects and their elements as well as some values (like colors) that describe their appearance. What is drawn at the moment depends on current game state.

[23]Normally browsers try to do double buffering automatically – as explained in [23] – but it won't work for Links, because the JavaScript generated for the drawing makes asynchronous calls all the time, so to avoid flickering of the canvas I had to implement the mechanism "manually".

For example `drawPillman`, a function (**[draw.3]**) that draws the player character, has in its definition the following code:

```
if (gameState.state == Eaten2) {
    # draw the final frame of the eaten animation...
    jsLineWidth(ctx, 1.0);
    jsStrokeStyle(ctx, pillmanColor);
    # use jsBeginPath, jsMoveTo, jsLineTo, jsClosePath...
    jsStroke(ctx);
}
```

I use only the basic canvas primitives (circles, rectangles, etc.) and drawing functions and no external graphic files in my games' rendering (which means that it is essentially procedural vector graphics)[24].

The game area is drawn first, with various elements and game objects are drawn in a specific order. Drawing ends with the HUD (heads-up display) and some debugging information about frame rate. The latter is handled by the `drawFps` function:

```
fun drawFps(ctx, fpsInfo, dFps) {
    var fpsInfo =
        (fpsInfo with
            frameCount = fpsInfo.frameCount + 1,
            dFps = dFps);

    # update lowest and highest registered frame rate
    # calculate the average frame rate
    # ...

    fpsInfo
}
```

I also added a function that saves the image data from a canvas to a file. This allows the player to make "screenshots" of the game area while playing.

The function is defined in JavaScript as follows:

```
function _jsSaveCanvas(canvas, node, mime) {
        var imageData = canvas.toDataURL(mime);
        node.href = imageData;
}
```

[24] The use of only the basic canvas manipulating functions could allow for integration with a WebGL renderer with canvas fallback, like Pixi.js:
`http://www.pixijs.com/`

It is used in the `draw` function like so (**[draw.4]**):

```
var gameState = if (gameState.metaState == Download) {
    # screenshot
    var downloadNode = getNodeById("download");
    var imageName = gameName ^^ "-" ^^
                    intToString(clientTime()) ^^ ".png";
    var _ = domSetAttributeFromRef(downloadNode,
                                   "download", imageName);
    replaceChildren(
        <#>
            {stringToXml(
                "Click to download the snapshot as")}
            <br />
            {stringToXml(imageName)}
        </#>,
        downloadNode);
    jsSaveCanvas(getNodeById(mainCanvas), downloadNode,
                 "image/png");

    (gameState with state = On)
}
else gameState;
```

If the player presses a key, the `metaState` of the game is set to `Download`, which causes the current frame to be saved to file. Actually the image data is encoded with base64 encoding into an URL using the `toDataURL` method[25]. This URL is then written to a `href` attribute of a HTML link node. This link can be clicked by the user to download the image file. The Links code generates the name for the file and writes it to the value of the node (**[page.1]** – see 4.3.9).

[25] `https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement.toDataURL`

4.3.9 Web page structure

The output of the `main` function is the web page, which contains the user interface:

```
page
    <html>
        <head>
        <style>
            /* ... */
            #{stringToXml(canvasId)} {{
                display: block;
                position: absolute;
                top: 0px;
                left: 0px;
            }}
        </style>
        </head>

        <body>
            <div id="container">
                <h1>Pillman</h1>

                <div id="{containerId}" tabindex="1"
                    l:onfocus="{initializeProcId ! 0; }">
                    <canvas id="{canvas2Id}"
                            width="{floatToString(
                                    canvasWidth)}"
                            height="{floatToString(
                                    canvasHeight)}">
                    </canvas>
                    <canvas id="{canvasId}"
                            width="{floatToString(
                                    canvasWidth)}"
                            height="{floatToString(
                                    canvasHeight)}">
                    </canvas>
                    <div id="info">
                        Click this canvas to start.
                    </div>
                </div>
                <div id="msg"></div>
                <span>The canvas above must be focused
                    for the keyboard input to work.
                </span><br />
                <span>
                    [SPACEBAR] controls Pillman
```

```
                </span><br />
                <!-- ... -->
                <a id="download"
                   tabindex="2" download="">
                </a> <!-- [page.1] -->
            </div>
        </body>
    </html>
```

Its structure and appearance is defined with HTML and CSS. The HTML is directly embedded into Links source with XML quasiquotes[26]. Inside these, we may evaluate Links expressions, by surrounding them with curly braces. In the above case, we see that values of some identifiers and attributes are determined this way. User interaction with XML elements is handled with l-event attributes[27]:

```
<div id="{containerId}" tabindex="1"
      l:onfocus="{initializeProcId ! 0; }">
```

Here, the `l:onfocus` l-event attribute is used to send a message to the initialize process when the user clicks in the area that contains the canvas element, on which the game screen is displayed. This effectively starts the game.

[26] http://groups.inf.ed.ac.uk/links/quick-help.html#xml_quasiquotes
[27] http://groups.inf.ed.ac.uk/links/quick-help.html#handling_user_actions

Chapter 5

Optimizations and benchmarking

Running the games described in Chapter 4 with acceptable frame rate required optimizations.

After analyzing the performance of the Links language system, I concluded that a good starting point for optimizations was the *jslib.js* library – which is the main part of the runtime of the language (see Chapter 3 for description). I began introducing optimizations to the library, until the desired frame rate was achieved.

This chapter describes the optimizations and the ways of carrying out measurements and obtaining performance data.

5.1 Initial notes

Links' debug mode was off during all tests (`debug=off` in the config file[1]). I made sure not to have any extra applications running in the background while testing (aside from a text editor, file manager and a terminal emulator, which were running constantly). All tests were performed using Chromium 36.0.1985.143, unless specified otherwise.

I generated hundreds of charts, from which I selected a representative one for each optimization.

5.2 The benchmark application

In order to quantify the effectiveness of my optimizations I wrote an application[2] (based on the game "framework" that I developed while implementing the games – see Chapter 4) in Links, which displays a chart of instantaneous frame rate for every frame. Numbers of frames are on the X axis and the instantaneous frame rate is on the Y axis.

[1]For description of the config file, please refer to Links' INSTALL document:
`https://github.com/links-lang/links/blob/sessions/INSTALL`
Section: `RUNNING WEB APPLICATIONS` et seq.

[2]see the file *performance-frozen.links* attached to this document – described in Appendix A

The application is itself very resource consuming, though all it does is processing 600 samples of frame rate data and drawing on the screen. I did not make attempts at optimizing it, though, because that is irrelevant – it is enough that the same application is used unchanged for every measurement.

The benchmark's implementation follows a very similar structure to the one described in Chapter 4, 4.3, except that the "logic" is obviously simpler as the application is not very interactive. Most of the computation performed by the application is essentially contained within its `draw` function:

```
fun draw(datapoints: Datapoints, lastTime,
         now, fpsInfo, chartParams) {
    # prepare canvas...

    # HELPER FUNCTIONS
        # fun scalePoint((x, y))
        # fun drawChartLine(ctx, color, y, msg)...
        # fun markYAxis (fraction)...
        # fun plotPoint(p)...

        # prepare datapoints
        var offset = fpsInfo.frameCount;
        var dFps = 1000.0 /.
            (intToFloat(now - lastTime) +. 1.0);

        var leftPoints = take(offset, datapoints); # [1]

        var middlePoint = [(offset + 1, dFps)];

        var datapointsLength = length(datapoints);
    var diff =
        floatToInt(chartParams.xScale) - datapointsLength;

        var rightPoints =
            drop(offset + 1,
                 take(floatToInt(chartParams.xScale),
                      datapoints)); # [2]

        # display debug info
        # jsFillText...

        # draw chart reference lines
        # drawChartLine...

        # draw x axis (frame numbers)...
```

```
        # draw datapoints
        jsBeginPath(ctx);
        var firstPoint = scalePoint(hd(datapoints));

        jsMoveTo(ctx,
                 intToFloat(firstPoint.1), firstPoint.2);

        var (midPointx, midPointy) =
            scalePoint(middlePoint !! 0);

        jsSetFillColor(ctx, "#222");
        var plottedLeftPoints =
            map(plotPoint, leftPoints); # [3]

        # calculate additional data for saving to file
        # if (chartParams.snap)...

        jsSetFillColor(ctx, "#2a2");
        ignore(map(plotPoint, middlePoint));

        jsSetFillColor(ctx, "#888");
        ignore(map(plotPoint, rightPoints)); # [4]
        jsStroke(ctx);

        # draw y axis (FPS)
    # markYAxis...

        # calculate and draw new fpsInfo
        var fpsInfo =
            drawFps(ctx, fpsInfo, dFps, chartParams);

        # new datapoints
        var datapoints =
            leftPoints ++ middlePoint ++ rightPoints; # [5]

        # double buffering
        if (doubleBuffer)
            swapBuffers(mainCanvas, dispCanvas)
        else ();

        # save canvas to file
        # if (chartParams.snap)...

        (fpsInfo, datapoints) # return
}
```

The most computationally or memory-intensive parts are marked with comments: `[1]`, `[2]`, `[3]`, `[4]`, `[5]`. The first argument of the function is the list of 600 `datapoints`. This list is processed with functions such as `take`, `drop`, `map` and `++` (concatenate), which causes a lot of copying (see Chapter 3, 3.4.3). `map` also means that a large number of function calls (one for each data point) is performed.

The benchmark measures its own frame rate, but the changes of it caused by the optimizations are reflected in changes of frame rate in the implemented games. This is because both the games and the benchmark essentially work in the same way. The computations that have the biggest impact on performance in the benchmark have also the biggest performance impact in games. See 5.7 for demonstration of the in-game performance after optimizations.

With this application, I determined the frame rate after each optimization and compared it to the baseline frame rate, which was measured before these optimizations.

The chart-generating application works like this: every frame the highest and the lowest registered frame rate is updated if needed. Every 600 frames, which I call an *iteration*, the average frame rate is calculated and collecting samples starts over. The samples from the previous iteration are marked with gray dots and the samples from the current iteration are marked with black dots. This is an example chart illustrating this:

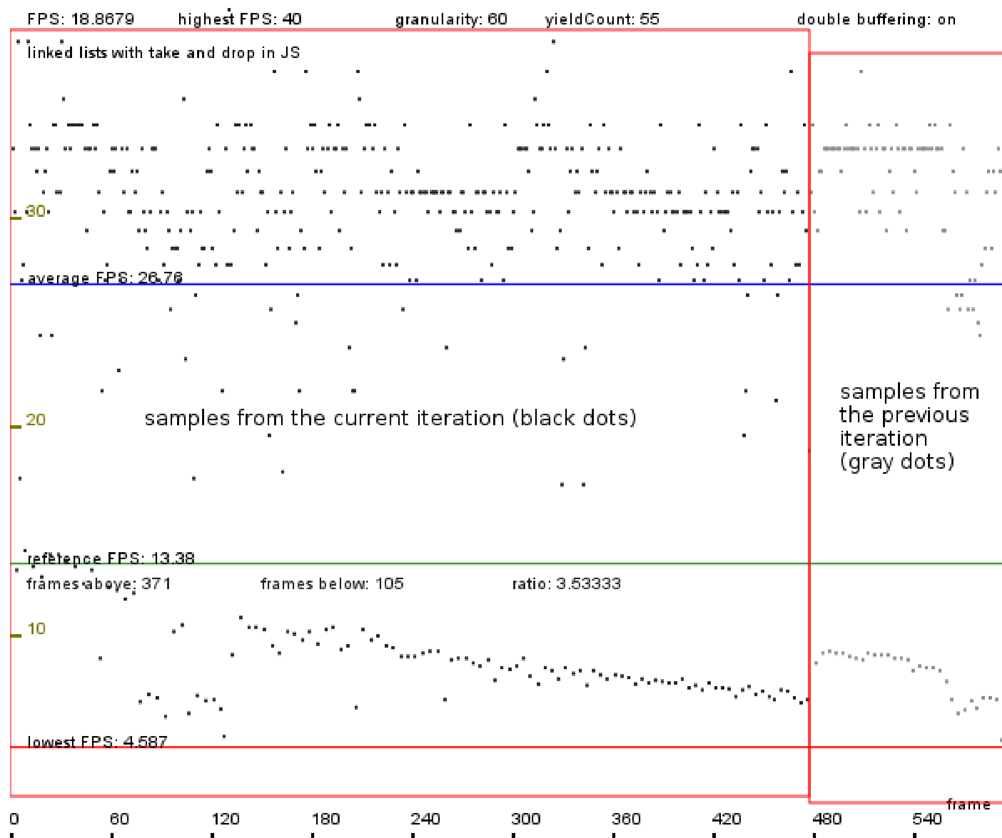


Figure 5.1: An example chart

Every chart consists of:

- X axis (frames): from 0 to 600, a mark every 60 frames
- Y axis (instantaneous frame rate): from 0 to the highest registered frame rate, marks at 25, 50 and 75% of the highest frame rate
- Blue line[3] ("average FPS") – indicating the average frame rate (calculated over 600 frames from the previous iteration)
- Green line ("reference FPS") – user-defined reference frame rate. Can be moved with up and down arrow keys. Below this line are three values dependent on its position:
  - Frames above – how many samples calculated so far in this iteration lie above the reference line
  - Frames below – how many samples lie below the line
  - Ratio – the number of frames above the line divided by the number of frames below
- Red line ("lowest FPS") – indicates the lowest instantaneous frame rate
- Text at the top:
  - FPS – current instantaneous frame rate
  - highest FPS – highest instantaneous frame rate
  - granularity – the value of `_yieldGranularity` (constant)
  - yieldCount – current value of `_yieldCount`; in the charts without the first optimization this value is very high as it is incremented every time `_yield` or `_yieldCont` is called (eventually it overflows, which may cause an unplanned stack clear); after the optimization the value is reset when it reaches the value of `_yieldGranularity`
  - double buffering – indicates whether double buffering[4] is on or off
  - description – the second line from the top describes the chart

[4] `http://en.wikipedia.org/wiki/Multiple_buffering#Double_buffering_in_computer_graphics`

[4] Note: the colors of the lines are irrelevant and are there only to aid in reading of the charts. When viewing the black and white version of this thesis, please look at the captions above the lines – for example the blue line has the "average FPS" caption above it.

5.3 The unoptimized version

Before any optimizations, the average frame rate in the benchmark application was about 1.6 FPS. This is illustrated on the following chart:

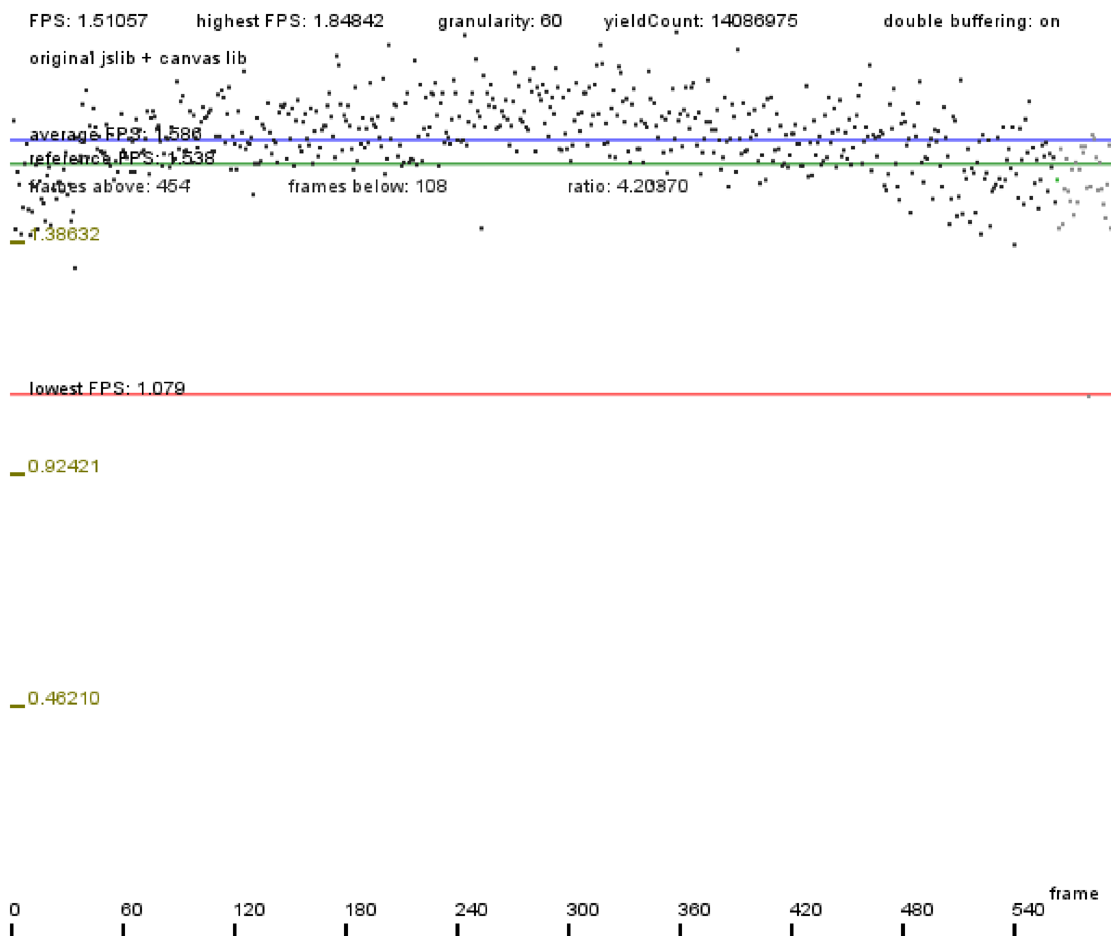


Figure 5.2: The unoptimized version

In the next section, when calculating absolute improvement I assume 1.6 FPS as the baseline.

5.4 Basic optimizations

This section describes four basic optimizations that I attempted before moving on to a little more sophisticated ones (see 5.5).

5.4.1 Optimized `_yield` and `_yieldCont`

Removing some calls to debugging functions and getting rid of one modulo operation and one negation in the bodies of `_yield` and `_yieldCont` (see also 5.5.11) improved the performance a bit. In fact the improvement is much more significant than what comparing the next chart to the previous may indicate as we will see when we combine this optimization with the next one.

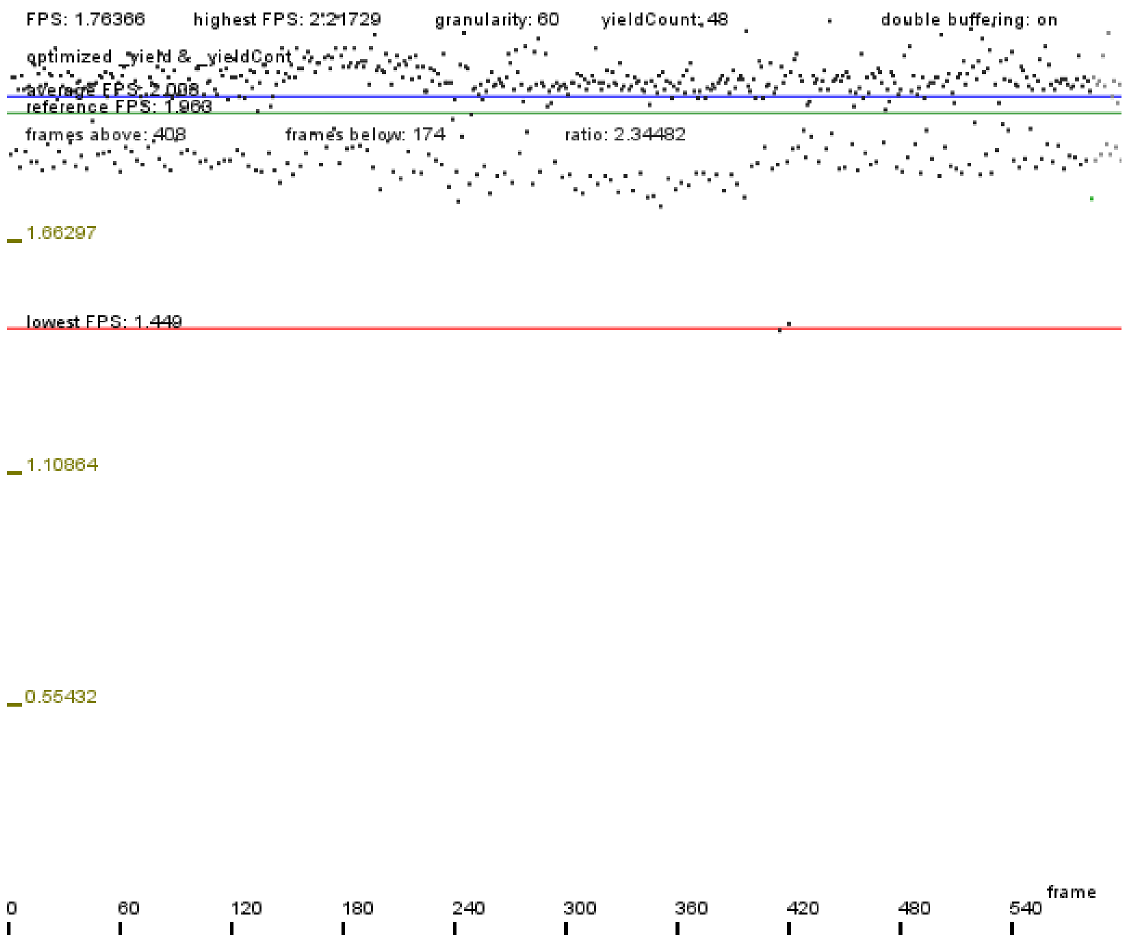


Figure 5.3: First optimization – faster `_yield*`

Average frame rate is 2 FPS. Absolute improvement: 25 %[5].

We can start to see the pattern described in section 5.4.10: the frame rate oscillates between some higher and lower value. It drops almost every third frame.

[5]Note that the these percentages have to be interpreted with caution as frame rate is not a particularly stable parameter to measure. We can tell that the optimization was effective only if the increase is very significant and/or when comparing and combining with other optimizations.

5.4.2 Faster `setTimeout`

All calls to `setTimeout` with the second argument of 0 were replaced by a call to `setZeroTimeout`[6].

`setTimeout` effectively has a minimum delay of about 4 ms[7]. `setZeroTimeout` does not have that limitation.

It is defined as follows:

```
(function() {
        var timeouts = [];

        var messageName = "0TMsg";

        function setZeroTimeout(fn) {
                timeouts.push(fn);
                window.postMessage(messageName, "*");
        }

        function handleMessage(event) {
                if (event.source == window &&
                    event.data == messageName) {
                        event.stopPropagation();
                        if (timeouts.length > 0) {
                                timeouts.shift()();
                        }
                }
        }

        window.addEventListener("message",
            handleMessage, true);

        window.setZeroTimeout = setZeroTimeout;
})();
```

It uses the `window.postMessage`[8] method, which makes it possible to execute the `fn` callback immediately[9].

[6]Implementation from:
http://dbaron.org/log/20100309-faster-timeouts

[7]https://developer.mozilla.org/en/docs/Web/API/window.setTimeout#Minimum.2F_maximum_delay_and_timeout_nesting

[8]https://developer.mozilla.org/en-US/docs/Web/API/window.postMessage

[9]https://developer.mozilla.org/en-US/docs/Web/API/window.setImmediate#Notes

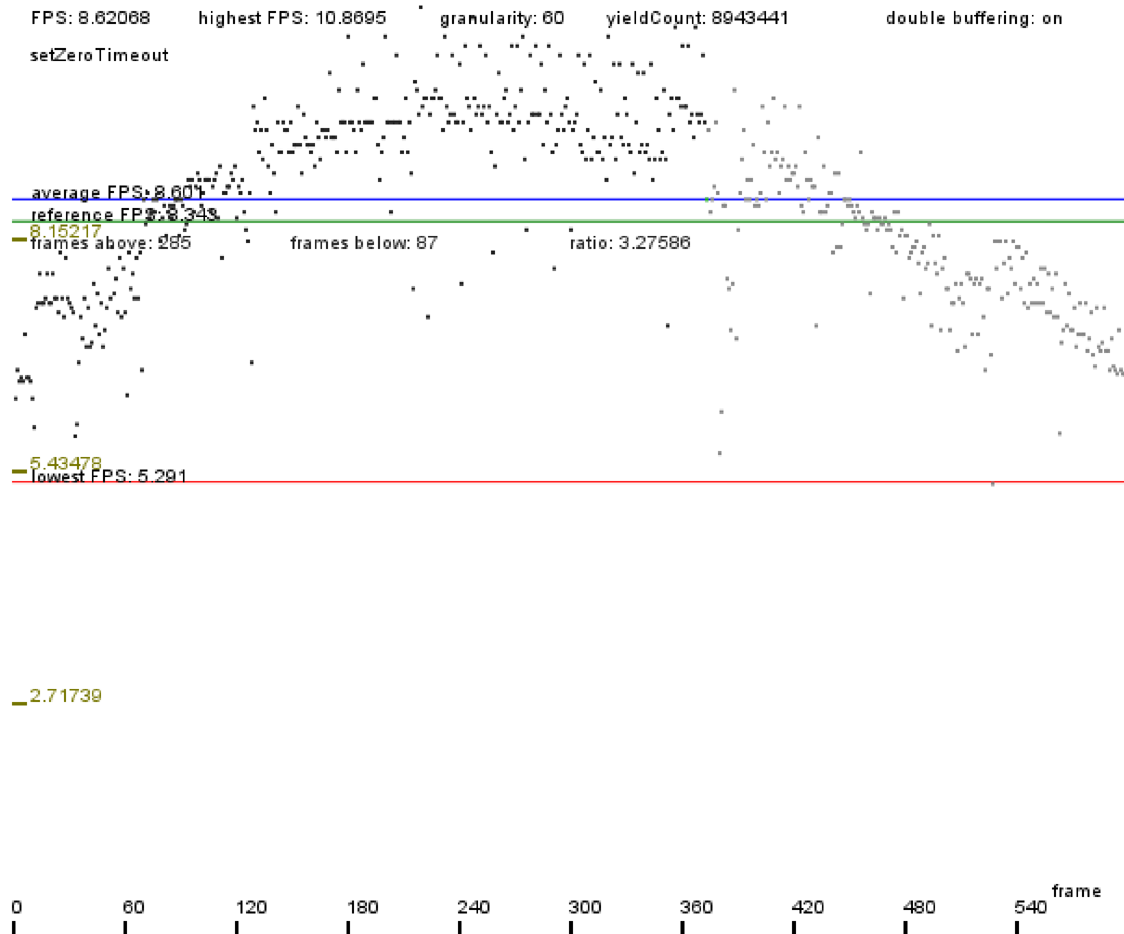


Figure 5.4: Second optimization – `setZeroTimeout`

Average frame rate is 8.6 FPS. Absolute improvement: 437 %.

A major increase. This is one of the most significant optimizations. Potentially saving up to 4 ms on each call to `_yield*` significantly boosted the overall performance.

The oscillation of the frame rate is more apparent when we combine this optimization with the previous – so not on this chart. This is most likely because the amount of time spent yielding each frame is much longer without the `_yield*` optimization and garbage collecting time stays roughly the same.

This is probably also the reason for the slightly curved shape of the outline of the data points on the chart.

5.4.3 Increased `_yieldGranularity`

I changed the value of `_yieldGranularity` from the original 60 to 260.

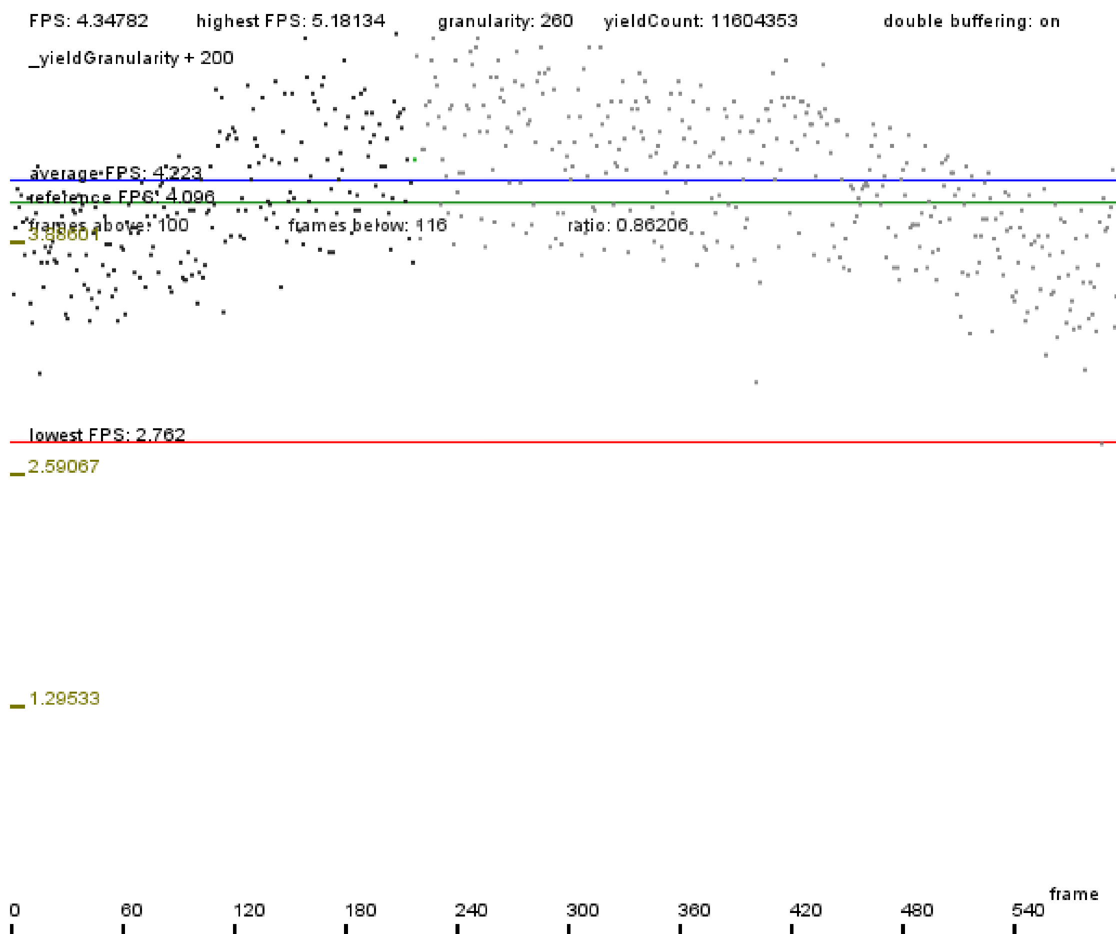


Figure 5.5: Third optimization – calling `setTimeout` less often

Average frame rate is 4.2 FPS. Absolute improvement: 163 %.

Increasing `_yieldGranularity` obviously has an impact on performance as `setTimeout` is not called so often, which means less frequent clearing of the JavaScript call stack. But this works up to a point[10] and the maximum value of `_yieldGranularity` differs between applications and browsers.

[10]Because as `_yieldGranularity` grows the amount of garbage being accumulated also grows (see Figure 5.11)

5.4.4 Turning off double buffering

As described in Chapter 4, 4.3.8 the rendering in my applications is done with double buffering.

Turning the mechanism off caused virtually no change in frame rate:

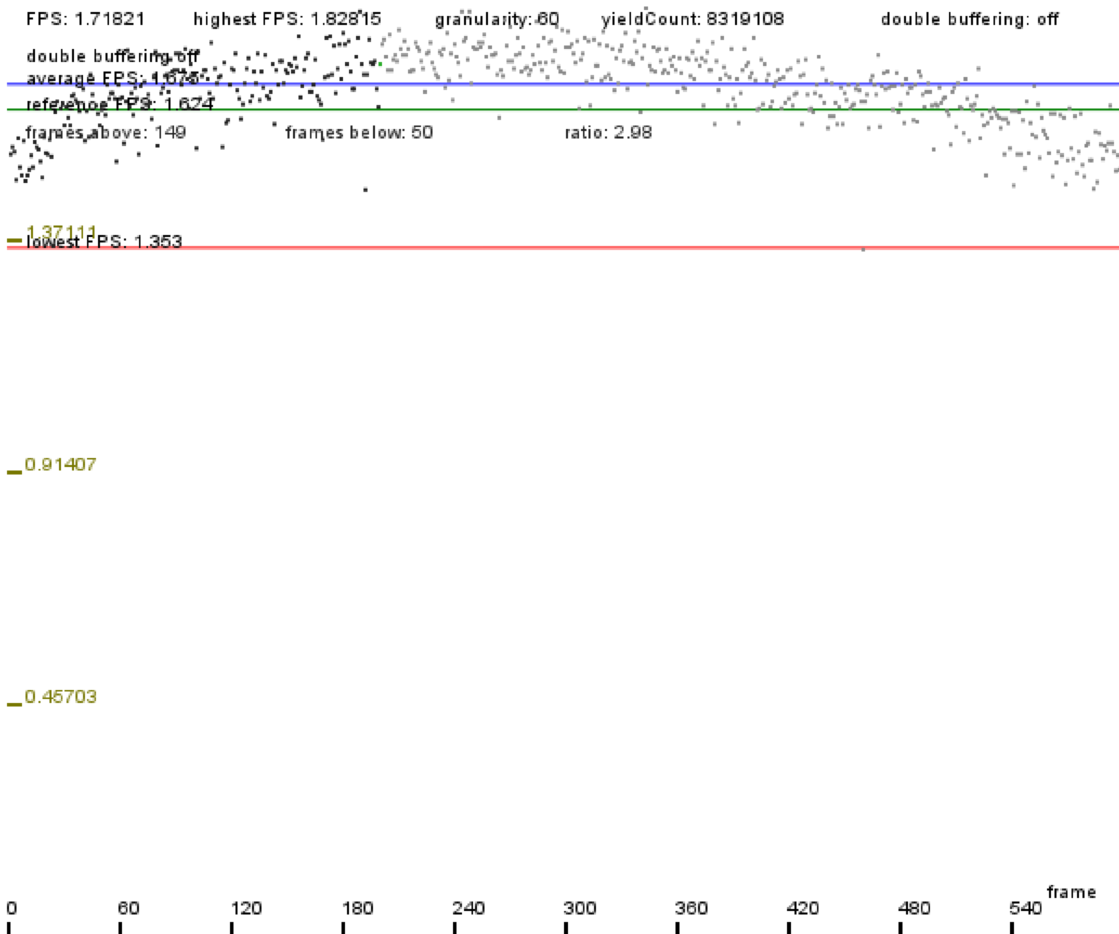


Figure 5.6: Fourth optimization – double buffering off

Average frame rate is 1.7. Absolute improvement: 6 %.

Almost no effect compared to the original. On the other hand doing similar tests with the Breakout clone shows that turning off double buffering has a more significant effect.

This is likely because in the game, there is much less calls to canvas drawing functions. There is at least 600 rectangles being drawn in the benchmark each frame – one for every data point. So in this case swapping buffers and redrawing the ready-made bitmap is a smaller fraction of the drawing time.

5.4.5 First two optimizations combined

When I applied the `_yield` (5.4.1) and `setZeroTimeout` (5.4.2) optimizations together I obtained the following result:

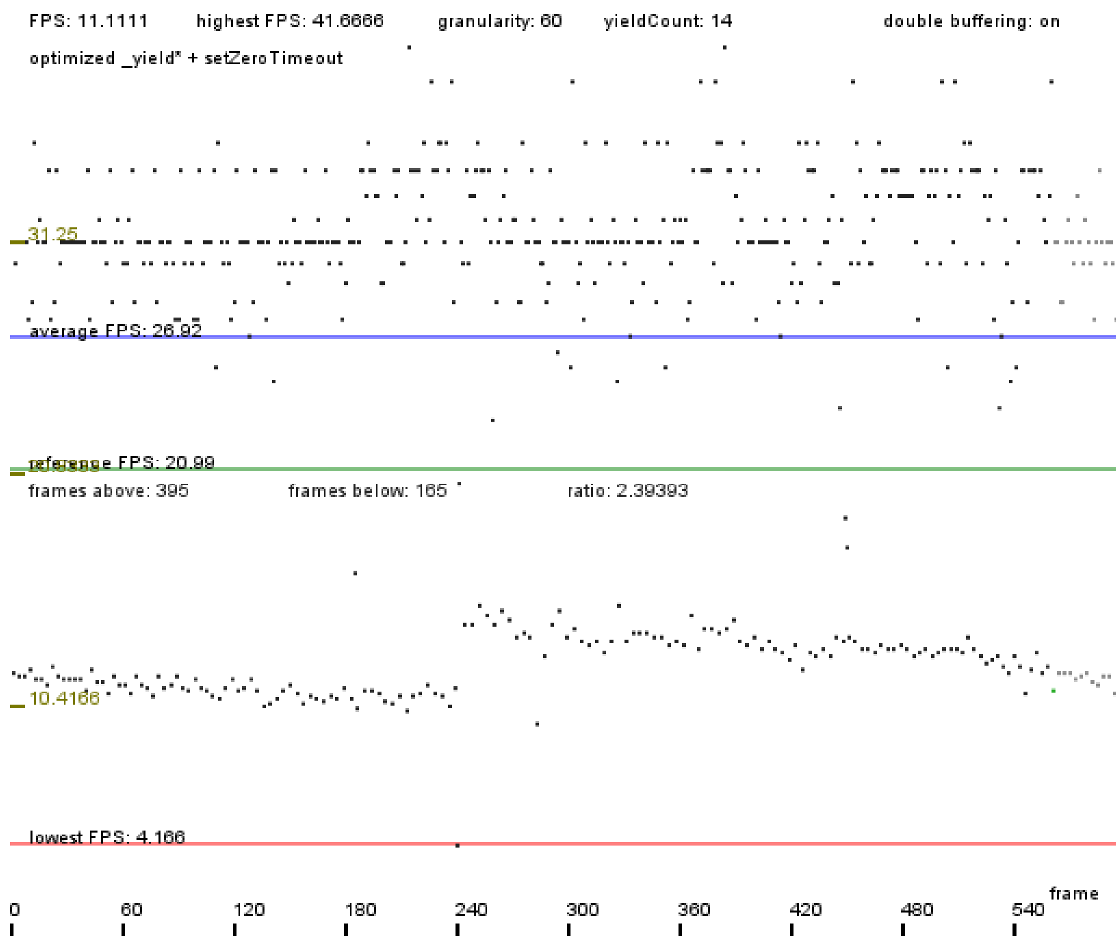


Figure 5.7: First two optimizations combined

Average frame rate is 27 FPS. Absolute improvement: 1588 %.
Great improvement. The oscillation of the frame rate is clearly apparent.

5.4.6 First three optimizations combined

I combined the first two optimizations (5.4.5) and increased `_yieldGranularity` as in 5.4.3.

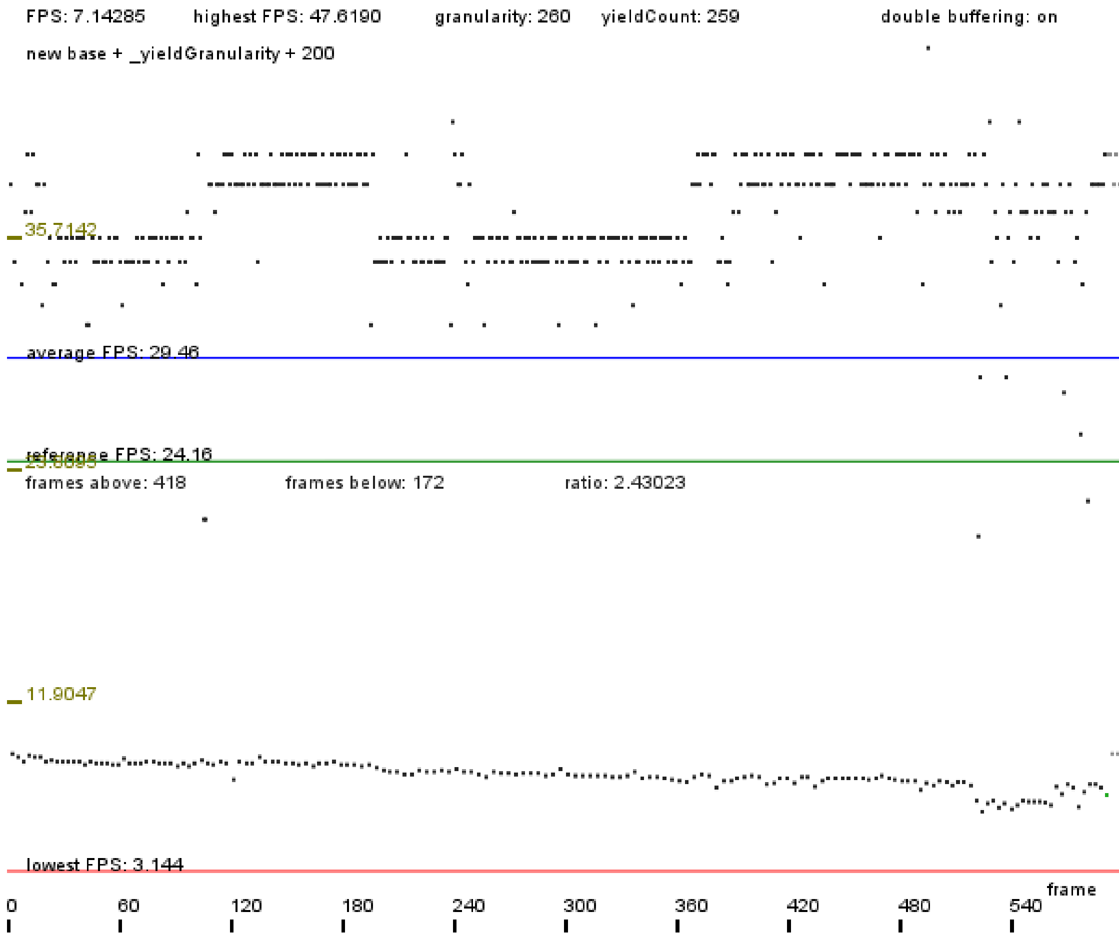


Figure 5.8: First three optimizations combined

Average frame rate is 29 FPS. Absolute improvement: 1713 %.

Improvement over the first two optimizations: 7 %.

An increase in `_yieldGranularity` bumps up the frame rate a bit, but not very significantly.

Note: we can observe that besides oscillating, the frame rate is also slowly going down with time. See 5.4.10 for explanation.

5.4.7 First two optimizations without double buffering

The `_yield` and `setZeroTimeout` (5.4.5) optimizations were also tested with double buffering turned off.

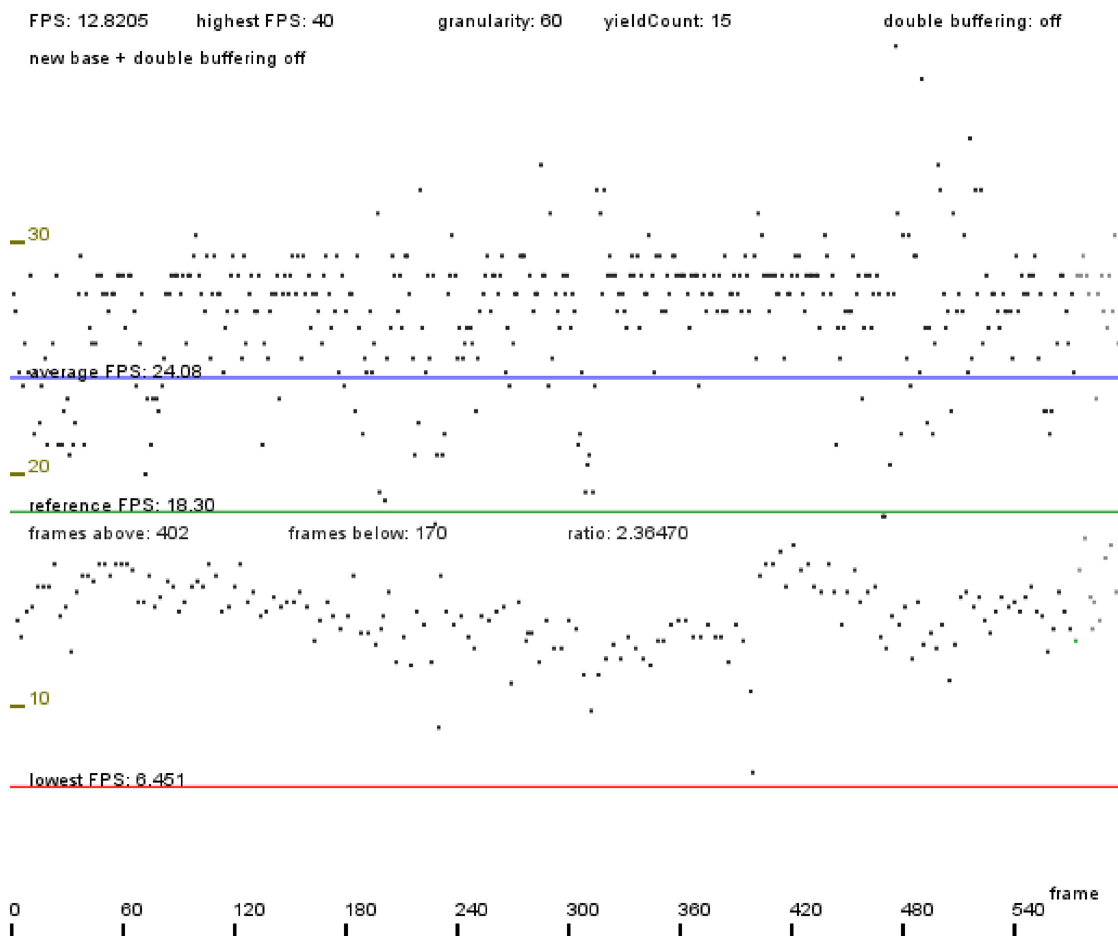


Figure 5.9: First two and the fourth optimization combined

Average frame rate is 24 FPS. Absolute improvement: 1400 %.
No improvement over the two optimizations alone.

5.4.8 Other simple optimizations

Other, less significant basic optimizations I tried were:

- Using a much (up to 10x) faster implementation of queues[11] – I tested that with `setZeroTimeout` (which uses a queue for storing functions to be called) and with `_send` (`!` in Links' syntax) and `recv` (which use queues for process mailboxes), but it turned out not to be a significant improvement, because of the generally small size of the queues. This faster implementation shows its advantages when used with bigger queues; for small ones the gain is cancelled out by the overhead.

- As some benchmarks show[12], the currently used implementation of queues based on lists and `unshift-pop` methods can be up to 2 times slower than the implementation based on `push-shift` in Firefox and Opera. On the other hand in Chromium the `unshift-pop` seems to generally be a bit faster. So in this case I decided not to optimize.

- I replaced the use of `new Date().getTime()` with `Date.now()` as the former way of getting the current time unnecessarily creates a `Date` object.

[11] http://code.stephenmorley.org/javascript/queues/
[12] http://jsperf.com/queuing-push-shift-vs-unshift-pop

5.4.9 Debugging

A thing that turned out to be significant for better performance was making sure that the debug functions were not called when the debug mode was off. The main debugging functions in Links are:

```
DEBUG.assert,
DEBUG.assert_noisy,
_debug,
_dumpSchedStatus
```

I found that `_dumpSchedStatus` was causing a very significant slowdown, unnecessarily executing code containing loops every time `_send` was called. This happened even if Links was not in debug mode as the function did not take `DEBUGGING` flag into account:

```
function _dumpSchedStatus() {
  _debug("---------\nMailbox status:");
  for (var i in _mailboxes) {
    if (_mailboxes[i].length > 0)
      _debug("  pid " + i + ": " +
             _mailboxes[i].length + " msgs waiting");
  }
  var blockedPids = "";
  for (var i in _blocked_procs) {
    if (blockedPids != "") blockedPids += ", ";
    blockedPids += i
  }
  if (blockedPids != "")
    _debug("  blocked process IDs: " +
           blockedPids + ".");
}
```

It is important to remember that to ensure better performance in release versions of not just Links, but any applications, debugging should be turned off.

Implementing debug functions so that they have as small an impact on performance as possible is also reasonable. Debug functions should not be called unnecessarily in performance-critical code.

5.4.10 The garbage problem

There is a clear pattern in the optimized versions: every n frames the frame rate drops significantly and it is slowly going down with time – this is caused by the garbage collector collecting large amounts of accumulating garbage periodically. The green line on every chart may be helpful in estimating the n.

The next two charts were made using a modified version of the benchmark application.[13].

To confirm that the GC is the cause of drops in frame rate, I extended Links with a function that allows invoking Chromium's garbage collector on demand. I put calls to this function after code that I thought was responsible for generating a lot of garbage – calling `map` on a big list. I forced 2 GC invocations per frame. This indeed stabilized the frame rate:

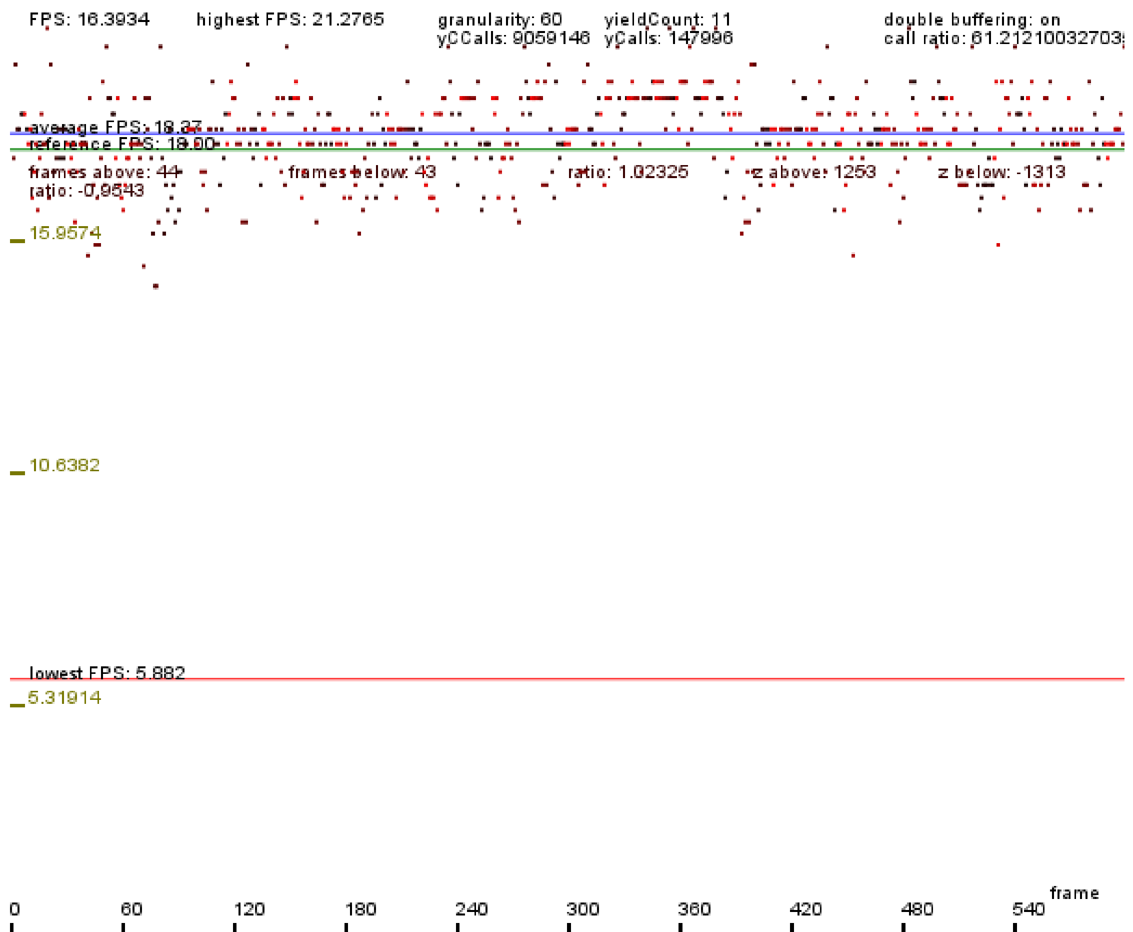


Figure 5.10: Invoking GC after mapping a function over a big list to clean up stabilizes the frame rate; optimizing the implementation of `map` may significantly boost performance

[13]Defined in *performance2.links* file. See Appendix A for the list and description of attached files.

When I increased `_yieldGranularity` too much, the pattern on the chart changed – GC was invoked more often. Probably because the bigger the call stack grows, the more garbage related to the data on the stack accumulates as for example the memory referenced from the stack cannot be reclaimed until the stack is cleared.

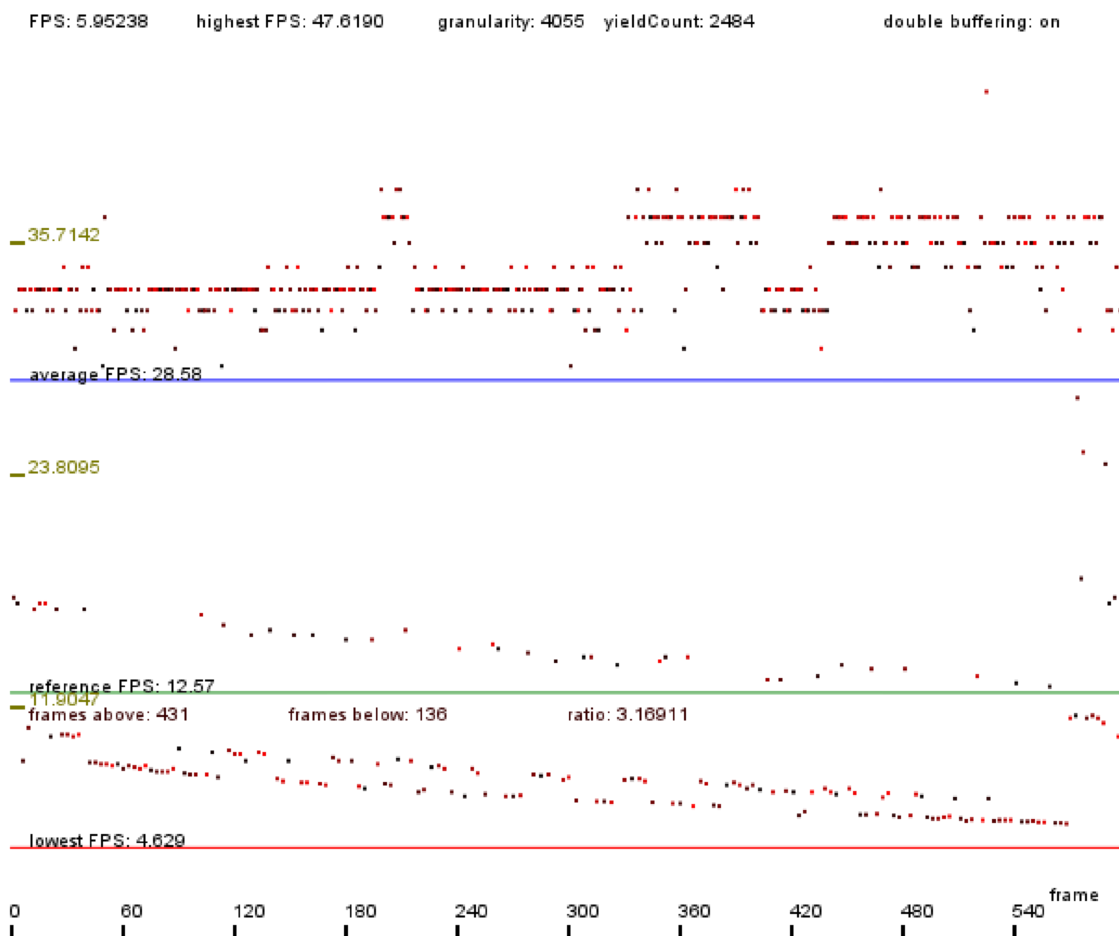


Figure 5.11: Too high `_yieldGranularity` results in more GC slowdowns

5.5 Advanced optimizations and profiling

This section describes various other optimizations that I attempted. It also includes a comparison of the benchmark with a native JavaScript implementation.

5.5.1 Profiling

I used profiling tools available in the Chromium browser (Chrome Dev Tools[14]) to examine memory usage in my applications as well as the Firebug extension in Firefox[15] to profile function execution times.

In the remaining part of this Chapter I am referring to profiling data to describe my optimizations. Below I provide a brief instruction on how to read the data.

***

This is an example heap timeline chart:

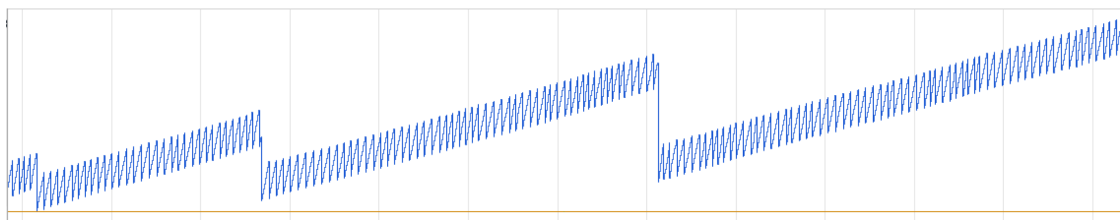


Figure 5.12: Example timeline chart: 1 minute, 10.8-90.2 MB

The above example timeline chart shows how heap size changes in time. The caption below contains its title and information describing the chart: the time span of the measurement (here 1 minute), the lowest (here 10.8 MB) and the highest (here 90.2 MB) registered heap size. Every timeline chart has this information in the caption.

[14] https://developer.chrome.com/devtools/docs/timeline
https://developer.chrome.com/devtools/docs/heap-profiling
https://developer.chrome.com/devtools/docs/javascript-memory-profiling

[15] http://getfirebug.com/whatisfirebug

This is an example heap allocation record with important elements highlighted and described:

Figure 5.13: Example heap allocation record; note that this was recorded over about 30 seconds

The "reference" line and size are important as they are good for quick comparisons.

***

Along with the heap allocation record I will also show some heap object statistics:

Constructor	Objects Count		Shallow Size		Retained Size	
▶(array)	4 459	11 %	844 152	22 %	1 009 568	26 %
▶(compiled code)	1 921	5 %	553 824	15 %	855 376	22 %
▶Object	1 843	5 %	69 896	2 %	710 624	19 %
▶(system)	11 824	29 %	383 960	10 %	625 504	16 %
▶(closure)	2 703	7 %	194 616	5 %	524 688	14 %
▶(string)	3 007	7 %	152 592	4 %	152 592	4 %
▶Window / http://loc	1	0 %	88	0 %	72 808	2 %
▶Array	521	1 %	16 672	0 %	61 168	2 %
▶system / Context	123	0 %	9 056	0 %	52 936	1 %
▶InternalArray	12	0 %	384	0 %	42 672	1 %
▶(number)	656	2 %	10 496	0 %	10 496	0 %

Figure 5.14: Example heap object statistics

The meaning of the columns is[16]:

- Constructor represents all objects created using this constructor.
- Objects Count displays the number of object instances.
- Shallow Size displays the sum of shallow sizes of all objects created by a certain constructor function. Shallow size means the memory that is held directly by the object itself.
- Retained Size displays the maximum retained size among the same set of objects. Retained size means the size of memory held by an object and its dependent objects (which are deleted along with the object when the memory is freed).

Shallow and retained sizes are in bytes.

***

This is an example execution time profile:

	Calls	Percent ▼	Own Time	Time	Avg	Min	Max
_yield2	165185	13.91%	5205.393ms	573939.758ms	3.475ms	0.07ms	77.147ms
_yieldCont	135920	11.58%	4332.607ms	489277.252ms	3.6ms	0.071ms	77.227ms
putPixel	38465	4.86%	1819.33ms	133761.883ms	3.477ms	0.177ms	46.064ms
_lsCons	82977	4.24%	1586.679ms	1586.679ms	0.019ms	0.017ms	2.337ms
putPixel/<	38465	3.45%	1289.77ms	130851.235ms	3.402ms	0.106ms	45.986ms
scalePoint_1687	35663	3.4%	1273.085ms	118482.167ms	3.322ms	0.107ms	46.235ms
lsMapIgnore	24616	3.23%	1206.915ms	88030.293ms	3.576ms	0.139ms	37.123ms

Figure 5.15: Example execution time profile

[16] From https://developer.chrome.com/devtools/docs/javascript-memory-profiling#summary-view

The meaning of the columns is[17]:

- **Function** – name of the called function.
- Calls – number of calls to the function.
- **Percent** – time that all calls to this function took as percentage of the total time of the profiling session.
- Own Time – time spent within the function (time spent within functions called by that function is not taken into account).
- Time – total time spent for all calls to this function.
- **Avg** – average time for one call of the function.
- Min – minimum time spent within the function.
- Max – maximum time spent within the function.

The most important columns are marked with bold type.

[17] From `https://getfirebug.com/wiki/index.php/Profiler`

5.5.2 Baseline frame rate for remaining optimizations

As a starting point for the more advanced optimizations I used a version of *jslib.js* that includes the first two of the basic optimizations (5.4.5) as well as has the debugging (5.4.9) and other minor issues cleaned up[18].

Thus, the baseline frame rate for all charts that follow is 28 FPS:

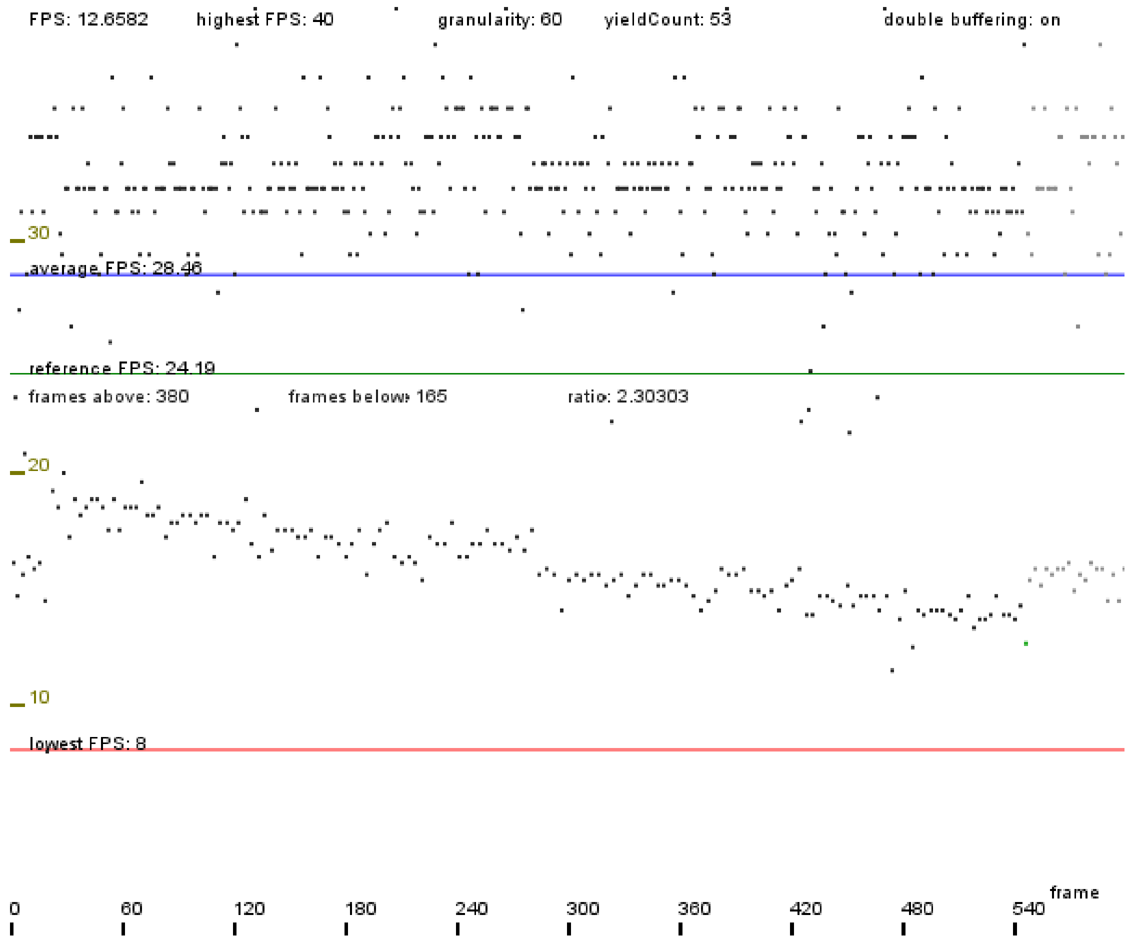


Figure 5.16: This is the baseline for all charts that follow

[18]It is attached to the thesis as *new base jslib.js*. See Appendix A, A.2.

5.5.3 JavaScript optimizer

I tried running the runtime (*jslib.js*) and the JavaScript generated by the Links compiler through the Google Closure Compiler[19].

I supplied the following parameters to the compiler:

```
// @output_file_name default.js
// @compilation_level ADVANCED_OPTIMIZATIONS
// @language ECMASCRIPT5
```

`ADVANCED_OPTIMIZATIONS` means that I want the most aggressive optimizations to be performed[20].

In order to get the application to run, I had to introduce some modifications to *jslib.js*[21], so that it complied with requirements of the compiler[22].

The effect of this optimization is presented below:

Figure 5.17: Performance of the code optimized with the Closure Compiler

Average frame rate is 29 FPS. Improvement in relation to the baseline: 1 %.

[19] `https://developers.google.com/closure/`

[20] `https://developers.google.com/closure/compiler/docs/compilation_levels`

[21] The final input to the Closure Compiler is attached to this document as *google closure input.js*. See Appendix A, A.2.

[22] `https://developers.google.com/closure/compiler/docs/api-tutorial3`

I tested the code multiple times, also on Firefox. The effect of bumping up the frame rate by about 1 FPS was consistent. Nonetheless, such an improvement is practically negligible.

This means that the optimizations that the Closure Compiler performs, namely:

- removal of comments, line breaks, unnecessary spaces, extraneous punctuation, and other whitespace,
- optimizations within expressions and functions, including renaming local variables and function parameters to shorter names,
- renaming of global variables, function names, and properties,
- dead code removal,
- global inlining

have very little significance in this case and we should look for optimization possibilities in other places.

Although this optimization did not improve the performance significantly, running the output of the compiler through an optimizer could be beneficial for bigger applications. Adding this sort of optimization (and obfuscation, which can also be desirable) to the Links compiler could be taken into consideration.

The next two charts present the comparison of timeline plots generated by the built-in Chromium profiler between the original (used for the baseline) and the optimized code.

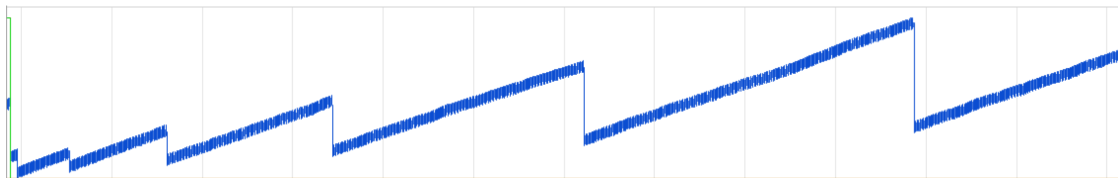


Figure 5.18: Chromium profiler's heap timeline plot **before** Closure Compiler's optimizations (the frame rate was 28 FPS); 4 minutes, 8.3-220 MB

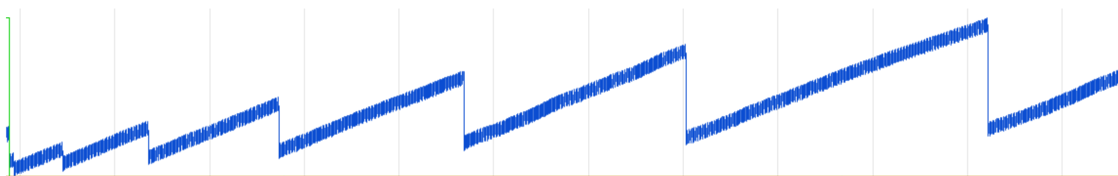


Figure 5.19: Chromium profiler's heap timeline **after** the code was optimized with the Closure Compiler (the frame rate was 29 FPS); 4 minutes, 6.7-174 MB

We can observe that the maximum heap size dropped by 46 MB. This is a significant improvement and another reason to consider extending the compiler with this optimization.

5.5.4 Comparison with the native version

It is also interesting to compare the previous timelines with a timeline for the native JavaScript version of the benchmark[23]:

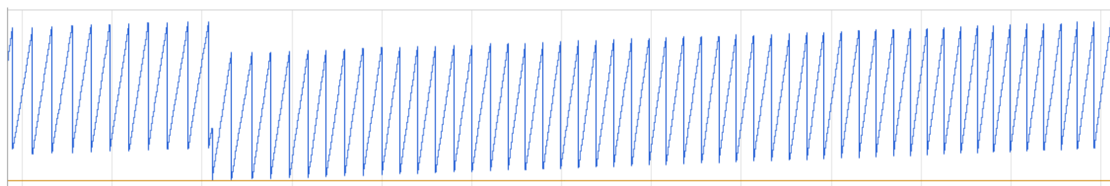


Figure 5.20: Timeline for the native JavaScript version; 2 minutes, 4-23.8 MB

We see that much less garbage is being generated. About 4 times less garbage is being collected per GC event. The garbage collector slowdowns have no significant impact on performance in this case.

***

Compare also the heap allocation record:

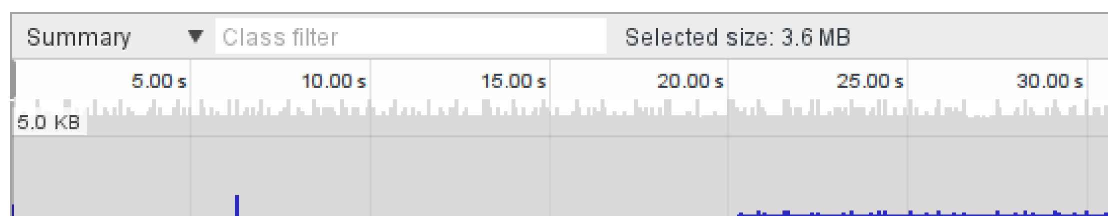


Figure 5.21: Native JavaScript version, heap allocations

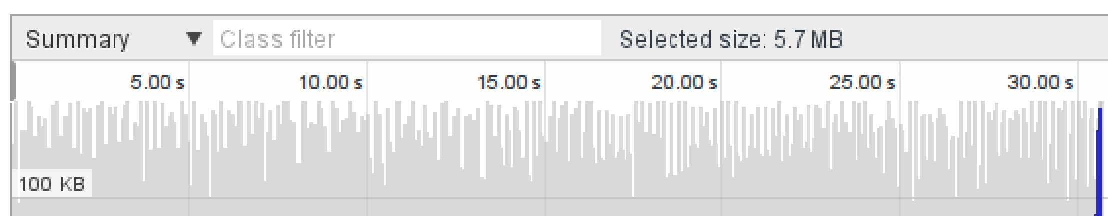


Figure 5.22: Links version (the one used to generate the baseline – see 5.5.2), heap allocation record

In the Links version of the application very large amounts of memory are being allocated and collected. The reference line is at 100 KB and the heap size goes up to 5 times higher than that. In case of the JavaScript version the size of the heap is much smaller and there are no spikes. The reference line here is at 5 KB,

[23] Attached in file *performance.html* – see Appendix A, A.2

which means that the amount of memory allocated is up to 100 times greater in the Links version.

***

And heap object statistics:

Constructor	Objects Count		Shallow Size		Retained Size	
▶ (array)	4 459	11 %	844 152	22 %	1 009 568	26 %
▶ (compiled code)	1 921	5 %	553 824	15 %	855 376	22 %
▶ Object	1 843	5 %	69 896	2 %	710 624	19 %
▶ (system)	11 824	29 %	383 960	10 %	625 504	16 %
▶ (closure)	2 703	7 %	194 616	5 %	524 688	14 %
▶ (string)	3 007	7 %	152 592	4 %	152 592	4 %
▶ Window / http://loc	1	0 %	88	0 %	72 808	2 %
▶ Array	521	1 %	16 672	0 %	61 168	2 %
▶ system / Context	123	0 %	9 056	0 %	52 936	1 %
▶ InternalArray	12	0 %	384	0 %	42 672	1 %
▶ (number)	656	2 %	10 496	0 %	10 496	0 %

Figure 5.23: Native JavaScript version, heap objects

Constructor	Objects Count		Shallow Size		Retained Size	
▶ (array)	11 867	21 %	2 365 008	39 %	2 747 632	46 %
▶ (closure)	6 341	11 %	456 552	8 %	2 164 744	36 %
▶ (compiled code)	4 170	7 %	1 123 816	19 %	1 710 632	28 %
▶ Object	2 735	5 %	88 184	1 %	1 393 472	23 %
▶ system / Context	1 703	3 %	104 824	2 %	1 372 856	23 %
▶ Array	1 840	3 %	58 880	1 %	1 269 424	21 %
▶ (system)	18 414	32 %	623 392	10 %	1 025 696	17 %
▶ Window / http://localho	1	0 %	88	0 %	400 304	7 %
▶ (string)	5 420	9 %	241 616	4 %	241 688	4 %
▶ c	20	0 %	480	0 %	65 664	1 %
▶ (concatenated string)	688	1 %	27 520	0 %	37 616	1 %

Figure 5.24: Links version (the one used to generate the baseline – see 5.5.2), heap objects

In the Links version there are large amounts of objects on the heap with arrays and closures having the biggest retained sizes. In the native version there is much less memory objects being generated. Function closures take up much less memory.

The chart produced by the JavaScript version:

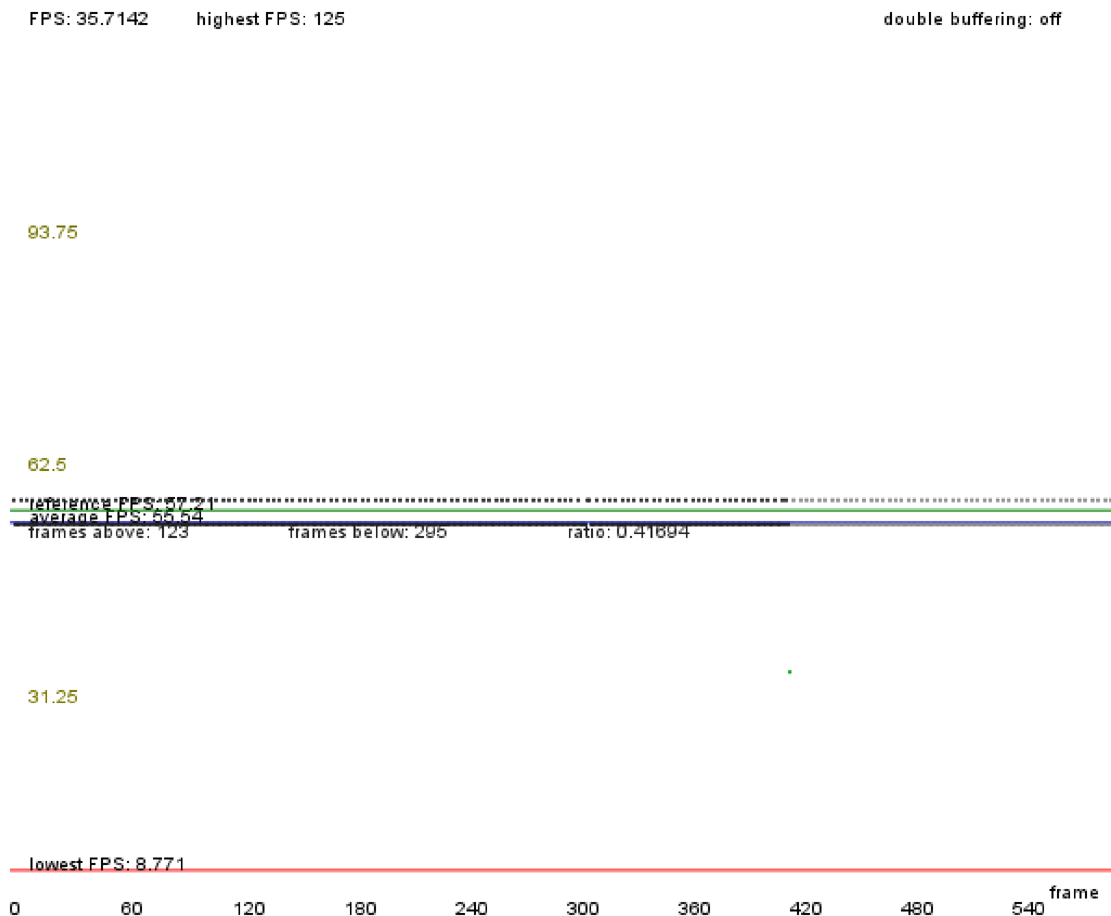


Figure 5.25: A chart generated by the JavaScript version of the benchmark

Note that double buffering is off, because it is unnecessary. Also the frame rate is limited to 60 FPS – frames are drawn using `requestAnimationFrame`[24]. Compared to the Links version, the frame rate here is stable (as expected – around 60 FPS).

[24] `https://developer.mozilla.org/en/docs/Web/API/window.requestAnimationFrame`

5.5.5 Execution time profiling

Profiling the execution time of any of my Links applications gives results similar to this:

	Calls	Percent ▼	Own Time	Time	Avg	Min	Max
_yield	72173	12.57%	3520.344ms	441966.667ms	6.124ms	0.069ms	64.442ms
LINKS</LINKS.eq	42105	8.31%	2327.104ms	6600.942ms	0.157ms	0.016ms	29.873ms
DEBUG</<.is_array	65154	7.52%	2104.892ms	3433.801ms	0.053ms	0.049ms	2.932ms
_yieldCont	59419	7.03%	1968.201ms	326011.922ms	5.487ms	0.049ms	64.531ms
___append	72172	5.42%	1516.437ms	1516.437ms	0.021ms	0.018ms	3.805ms
is_instance	65154	4.75%	1328.909ms	1328.909ms	0.02ms	0.019ms	0.531ms
_Concat	38344	4.43%	1239.774ms	2089.944ms	0.055ms	0.049ms	8.437ms
LINKS</LINKS.concat	38344	3.04%	850.171ms	850.171ms	0.022ms	0.019ms	8.385ms
_Cons	23690	2.72%	760.763ms	2051.792ms	0.087ms	0.08ms	8.471ms
concatMap	9767	2.36%	661.824ms	59392.895ms	6.081ms	0.311ms	57.352ms
DEBUG</<.is_unit	34230	2.26%	631.935ms	631.935ms	0.018ms	0.016ms	0.087ms
_tl	28668	2.04%	571.488ms	571.488ms	0.02ms	0.018ms	1.526ms
fold_left	9543	1.9%	531.448ms	50745.051ms	5.318ms	0.137ms	63.218ms
_hd	28668	1.89%	529.918ms	529.918ms	0.018ms	0.017ms	1.017ms
concatMap/<	9385	1.62%	452.956ms	54224.969ms	5.778ms	0.172ms	26.463ms
concatMap/</</</<	9385	1.56%	435.489ms	47863.003ms	5.1ms	0.172ms	34.252ms
map	5051	1.14%	318.988ms	29698.037ms	5.88ms	0.319ms	64.784ms
concatMap/</<	9385	1.14%	318.981ms	53585.898ms	5.71ms	0.106ms	26.394ms
concatMap/</</<	9385	1.13%	315.353ms	56259.066ms	5.995ms	0.106ms	57.473ms
zip	4628	1.11%	311.201ms	64499.496ms	13.937ms	0.527ms	62.428ms
foldStep_1873/</<	4501	1.04%	290.569ms	21938.159ms	4.874ms	0.272ms	17.764ms
zip/</<	4501	1.02%	285.234ms	62251.253ms	13.831ms	0.432ms	62.071ms
map/<	4788	0.83%	232.334ms	25953.619ms	5.421ms	0.173ms	23.582ms

Figure 5.26: Profiling execution time of my Breakout clone

Obviously `_yield` and `_yieldCont` take the most overall time and are called the most often (see Chapter 3, 3.4.1 for details). A lot of list operations means calling a lot of functions that manipulate them, which are also quite costly.

Note that the average execution time of `_yield` is over 6 ms.

From this we see that aside from `_yield*` the following functions are candidates for optimization:

- `LINKS.eq` – the compiler should make use of type information and generate specialized code for comparing different types of objects, instead of just using a general runtime function.
- `map` and other list manipulating functions (`take`, `drop`, `zip`, `hd`, `tl`...) – the way lists are implemented (right now they are just JavaScript arrays) should be changed and all functions operating on lists should be adjusted to that more efficient implementation; that would be a major improvement in performance.

These optimizations were implemented and are described in sections 5.5.6 to 5.5.10.

5.5.6 Linked list type

I replaced all Links lists with a custom list type, defined in Links like so:

```
typename Ls(a) = mu l . [| Nil | Cn: (a, l) |];
```

`mu` means it is a recursive definition.

I have defined all basic list manipulating functions for this type – also in Links. These include:

```
lsLength,
lsHead,
lsTail,
lsLast,
lsEmpty,
lsZip,
lsAppend,
lsConcatMap,
lsFilter,
lsMap,
lsTakeWhile,
lsTake,
lsDrop
```

The names follow the common convention, except that they are prefixed with `ls-`.

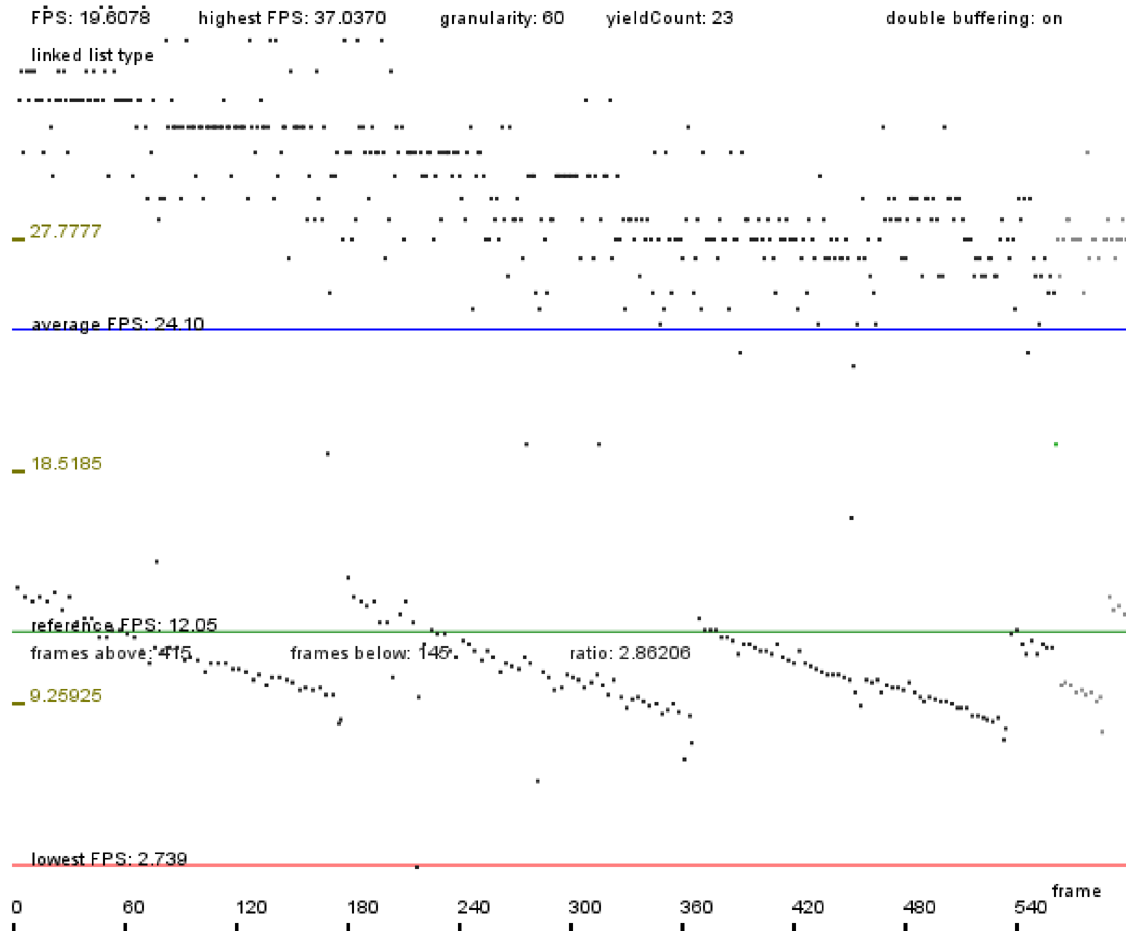


Figure 5.27: Linked list type implemented in Links

We can see a drop in frame rate and a lot more garbage being generated. This is the opposite of what we would expect. There seems to be much more copying.

Before I tried investigating how the generated JavaScript looks like and what is the exact reason for this copying, I checked the hypothesis that functions like `take` and `drop`, which might do some unnecessary copying, are problematic here. Then I encountered strange behaviour (see 5.5.7) and moved on to a different approach.

5.5.7 Linked list type with native `take` and `drop`

The most copying in the benchmark application seems to be caused by `take` and `drop` functions. For my custom list implementation I replaced them with optimized JavaScript versions, called `lsTake` and `lsDrop` respectively.

Before this optimization, they were defined in Links:

```
fun lsTake(n, l) {
        fun lsTakeHelper(m, n, l) {
                switch (l) {
                        case Nil -> Nil
                        case Cn(x, xs) ->
                                if (m < n)
                                    Cn(x,
                                       lsTakeHelper(
                                           m + 1,
                                           n, xs))
                                else Nil
                }
        }
        lsTakeHelper(0, n, l)
}

fun lsDrop(n, l) {
        fun lsDropHelper(m, n, l) {
                switch (l) {
                        case Nil -> Nil
                        case Cn(x, xs) ->
                                if (m < n)
                                    lsDropHelper(
                                        m + 1,
                                        n, xs)
                                else Cn(x, xs)
                }
        }
        lsDropHelper(0, n, l)
}
```

I use helper functions and recursion here. This compiles to continuation-passing style, which adds to the overhead.

I implemented the optimized versions in JavaScript:

```
function _lsTake(n, xs) {
        var arr = [];
        while (xs._label !== 'Nil' && n > 0) {
                arr.push(xs._value['1']);
                xs = xs._value['2'];
                --n;
        }
        return _lsFromArray(arr);
}

function _lsDrop(n, xs) {
        while (xs._label !== 'Nil' && n > 0) {
                xs = xs._value["2"];
                --n;
        }
        return xs;
}
```

Recursion is replaced with loops. Functions take advantage of the internal representation of the list type used by Links' compiler. `_lsDrop` performs no copying.

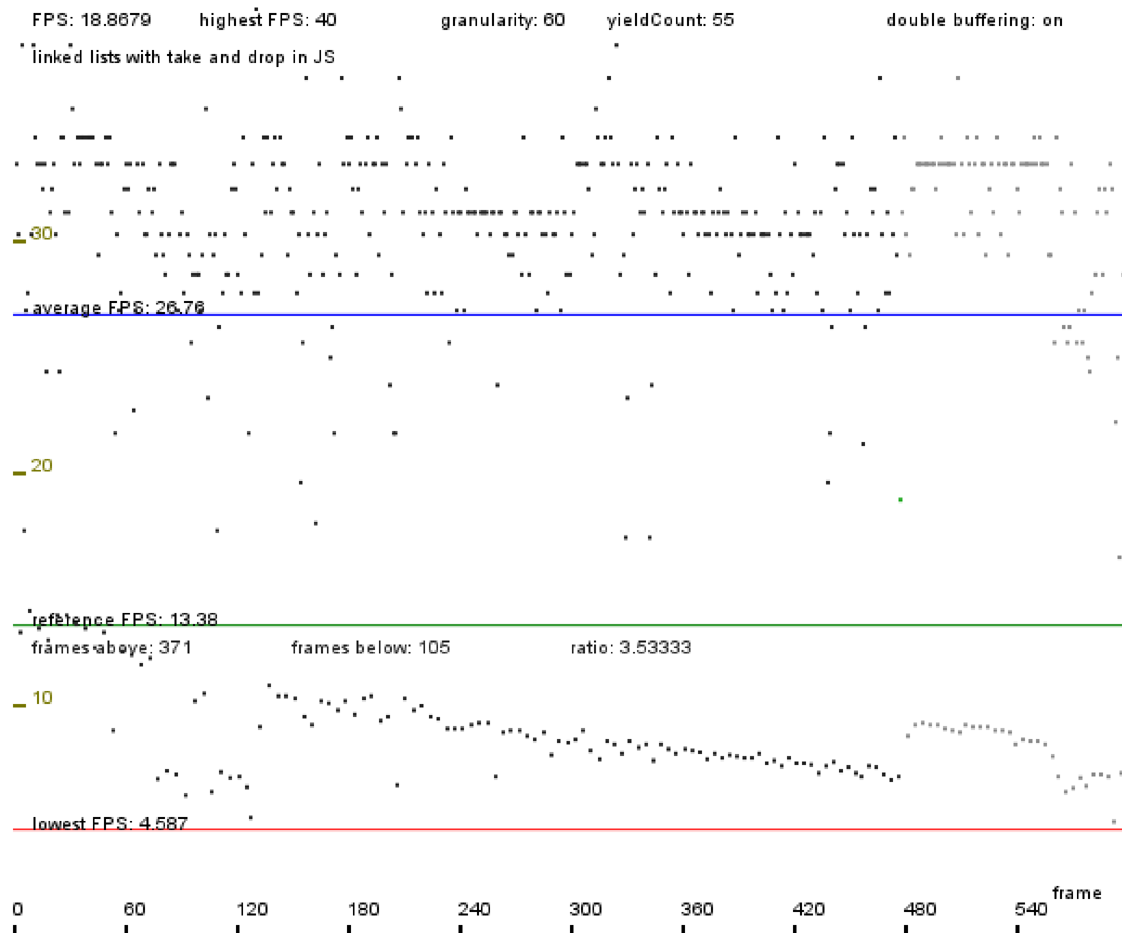


Figure 5.28: Linked lists with `take` and `drop` in JS

There indeed is some improvement – frame rate went up from 24 to almost 27 FPS – but still overall the performance is lower than the baseline (28 FPS).

After over 8 iterations of the benchmark I also noticed this strange behaviour:

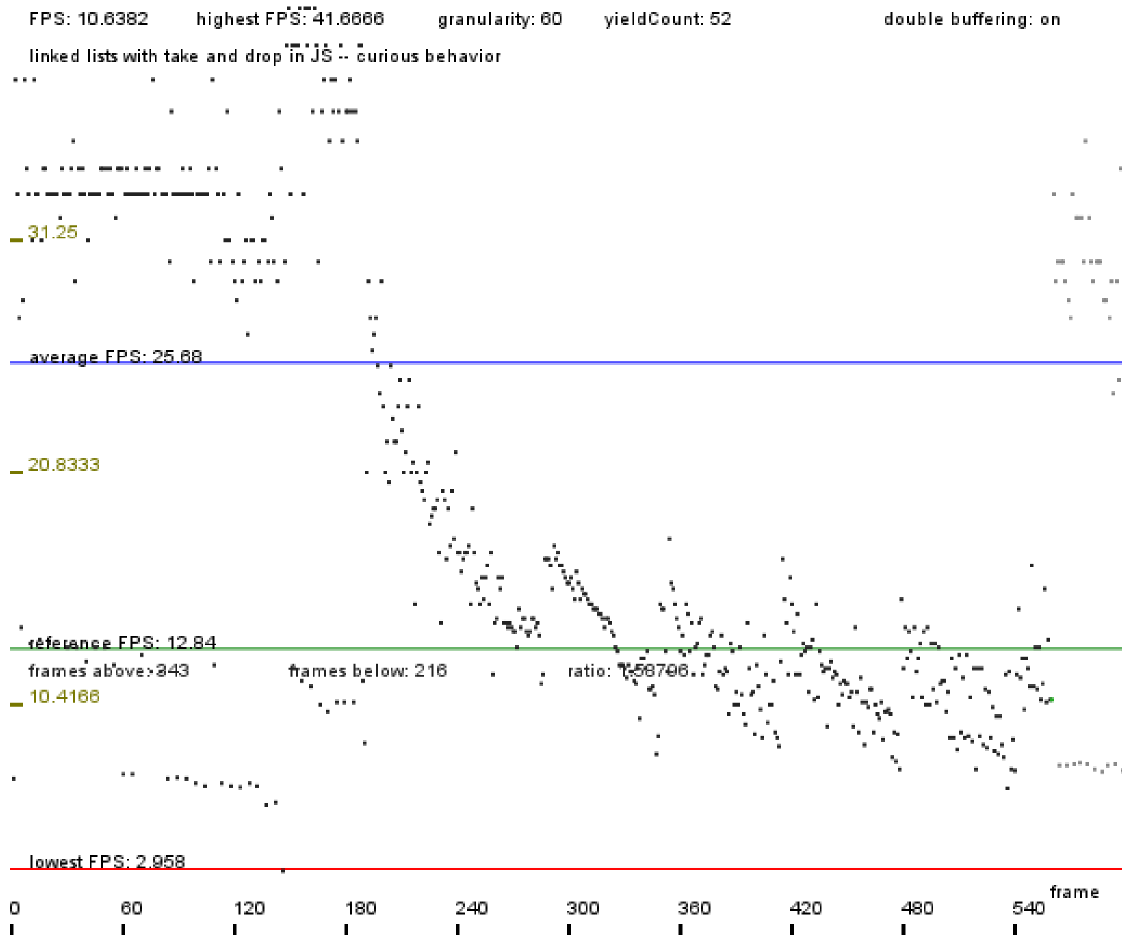


Figure 5.29: Linked lists with take and drop in JS – curious case

I was not able to track down the cause, but I concluded that I will not rely on the list type defined in Links, but instead I will reimplement it entirely in JavaScript.

5.5.8 JavaScript linked lists

I replaced the original list type with a JavaScript implementation. I implemented all relevant list manipulating functions in JavaScript as well, except `map*`, which was implemented in Links as `lsMap*`.

The cons[25] function that constructs a linked list from a `head` and a `tail` was defined in JavaScript as:

```
function _lsCons(head, tail) {
    return { _label: '::', _head: head, _tail: tail };
}
```

An empty list was defined as:

```
var lsNil = { _label: '[]' };
```

This implementation was later further optimized (see 5.5.9).

The effect of this optimization:

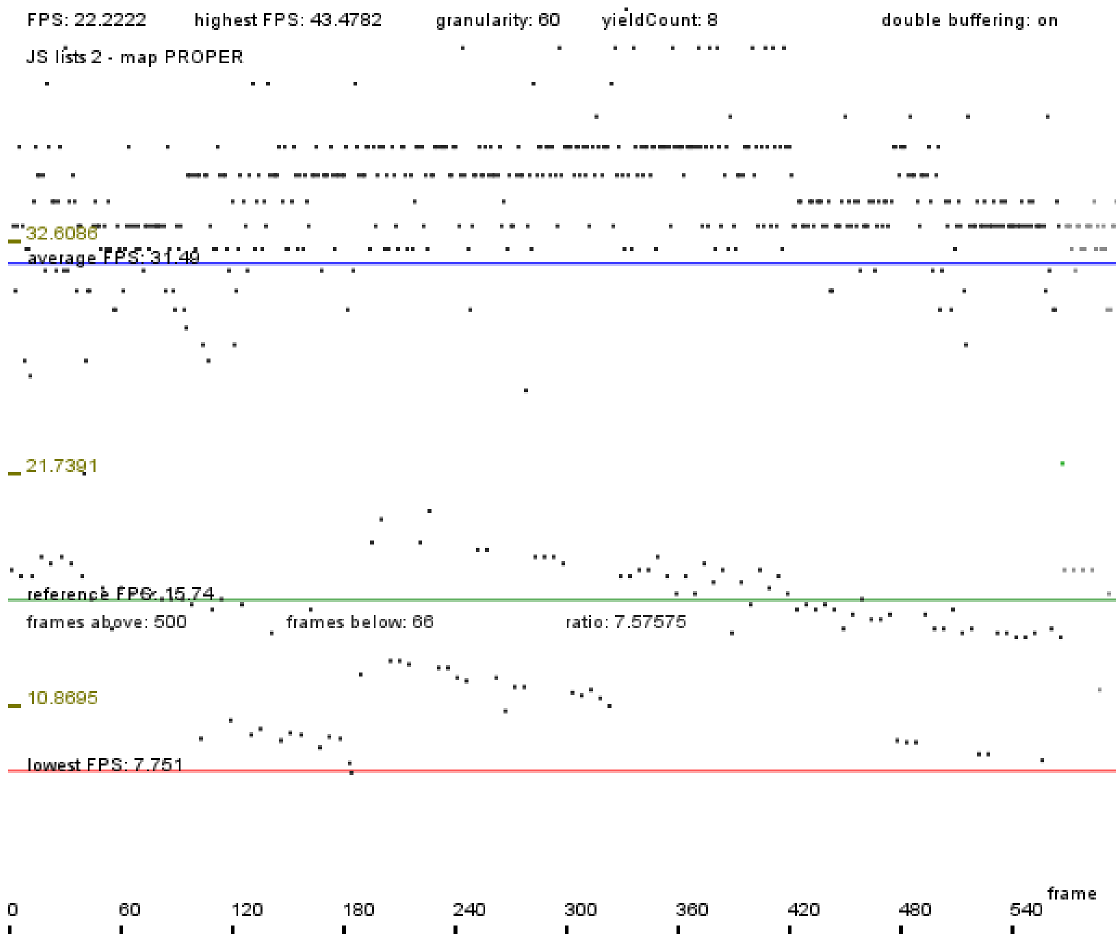


Figure 5.30: Linked lists implemented entirely in JS (except `map*`)

Average frame rate is 31 FPS. 12 % more than the baseline.

[25] `http://en.wikipedia.org/wiki/Cons`

We see a small improvement. The frame rate goes up, the garbage collection count decreases a bit.

***

We can also see the improvement in the heap allocation records:

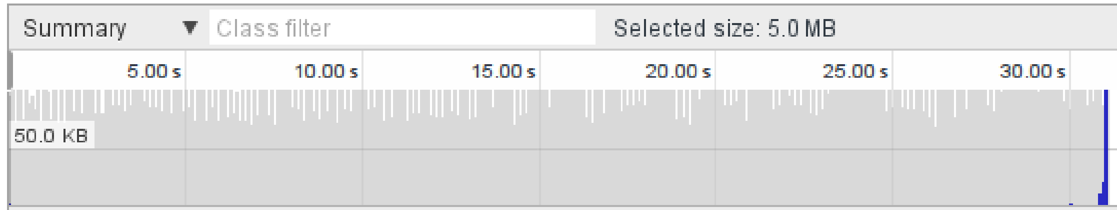


Figure 5.31: Native JavaScript linked lists – less heap allocations

We notice that the reference line dropped from 100 KB (see 5.22) to 50 KB and the heap size is a bit more stable in time. Less garbage is being generated.

Constructor	Objects Count		Shallow Size		Retained Size ▼	
▶(array)	10 277	18 %	1 585 728	30 %	1 941 552	37 %
▶(closure)	5 652	10 %	406 944	8 %	1 522 592	29 %
▶Object	5 205	9 %	206 744	4 %	1 500 760	29 %
▶(compiled code)	4 076	7 %	967 672	19 %	1 453 720	28 %
▶Window / http://localhos	1	0 %	88	0 %	1 054 784	20 %
▶(system)	17 939	32 %	603 816	12 %	995 440	19 %
▶system / Context	1 000	2 %	65 360	1 %	687 416	13 %
▶Array	1 143	2 %	36 576	1 %	399 872	8 %
▶(string)	5 323	10 %	238 072	5 %	238 144	5 %
▶c	20	0 %	480	0 %	65 664	1 %
▶(concatenated string)	685	1 %	27 400	1 %	37 544	1 %
▶InternalArray	16	0 %	512	0 %	34 728	1 %

Figure 5.32: Native JavaScript linked lists – heap objects

Compare the retained size of `Array` objects before (5.22) and after optimization – 1,269,424 bytes vs 399,872 bytes, which is over 3 times less. The reason for this is that before the optimization lists were represented with `Array` objects. The amount of these objects before and after is 1,840 and 1,143 respectively.

The new linked list representation also caused the change in the amount of `Object` objects – 2,735 vs 5,205. The retained size did not increase proportionally (1,393,472 bytes vs 1,500,760 bytes), showing that our list type takes up relatively little space.

5.5.9 JS lists with `null`

The implementation of `_lsCons` was simplified like so (compare to the implementation from 5.5.8):

```
function _lsCons(head, tail) {
    return { _head: head, _tail: tail };
}
```

And the empty list was simply defined as:

```
var lsNil = null;
```

The lack of the `_label` property made list objects smaller, thus further reducing the amount of generated garbage.

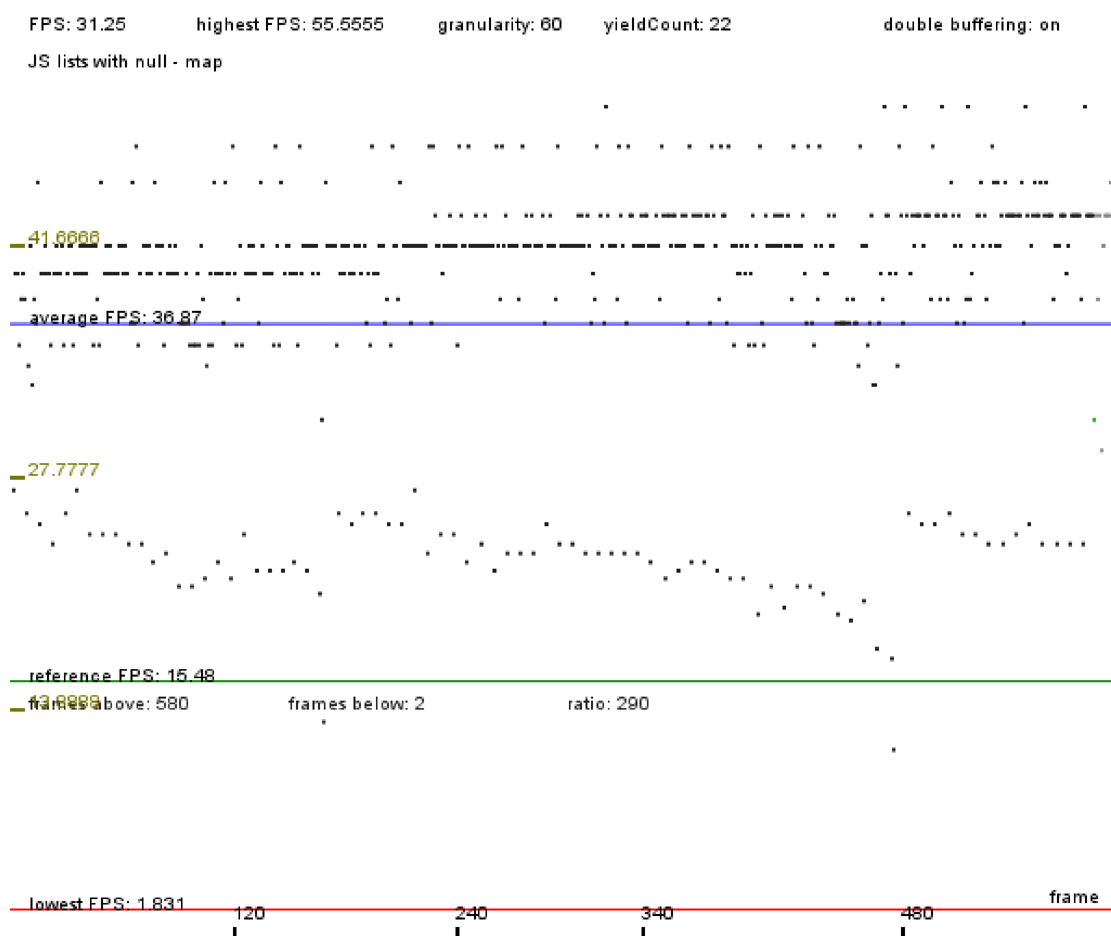


Figure 5.33: Further optimized native JavaScript linked lists

Simplifying and optimizing the native linked list bumped up the frame rate a bit more. The final average frame rate for this optimization is 37 FPS (9 FPS more than the baseline). There is less garbage collection and the low point of frame rate oscillation went up a bit.

As illustrated below, this optimization did not influence the heap size significantly:

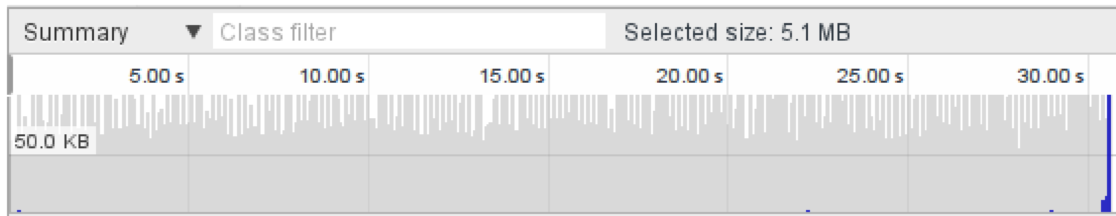


Figure 5.34: Heap allocations after linked list optimizations

5.5.10 Equality

Another optimization, which was applied on top of the previous one was replacing all the comparisons (`==`, etc.) in the code of the benchmark application with calls to specialized equality functions – which assume the type of the things being compared – implemented in JavaScript. This was supposed to reduce the overhead of `LINKS.eq` (see Chapter 3, 3.4.4).

For example if we know that we are comparing simple integers at compile time – and we do since Links is statically typed – instead of compiling this comparison to an invocation of `LINKS.eq`, which will again check the type at runtime, we may compile to:

```
function _intEq(l, r) {
        return l === r;
}
```

or even inline the comparison. I implemented a few functions analogous to `intEq` just to test what performance impact such an optimization could have.

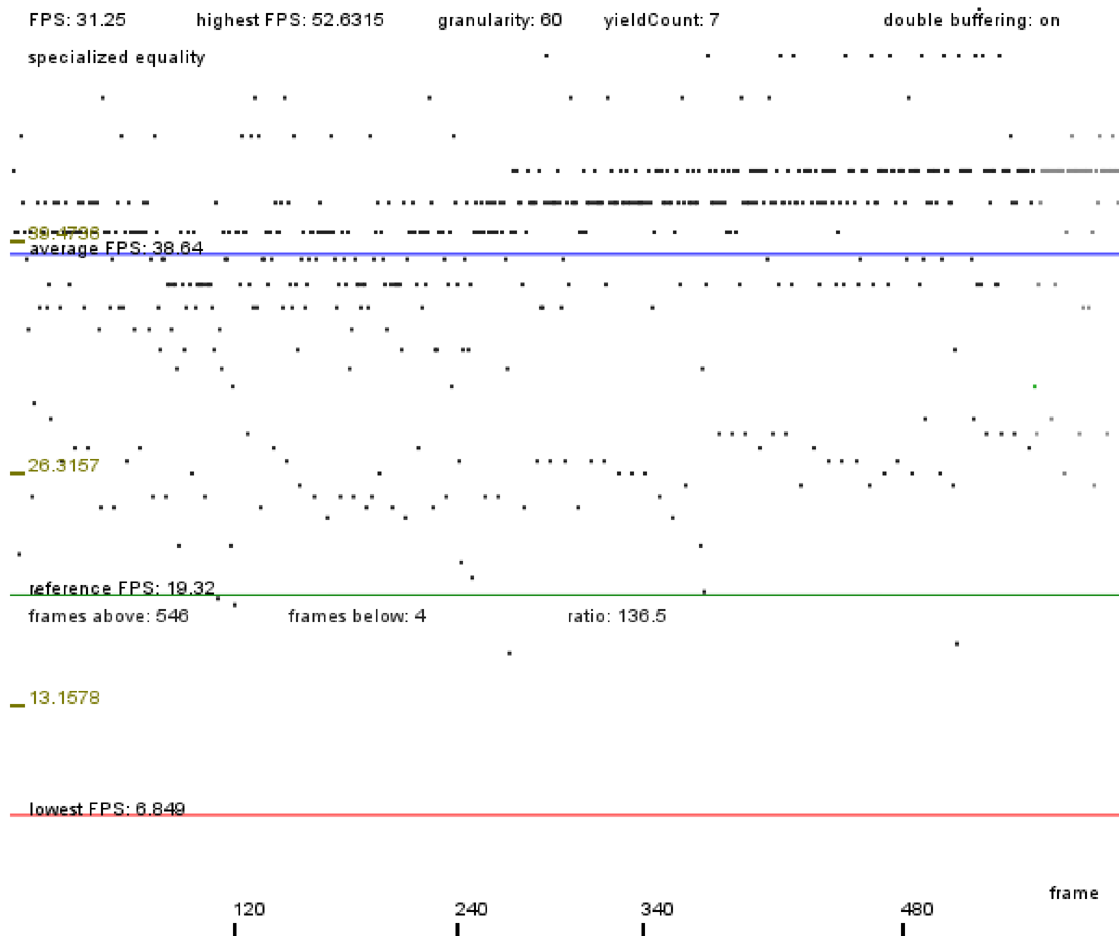


Figure 5.35: Using specialized functions for equality adds some more frames per second

Indeed, it turned out to be a slight improvement.

Looking at the Firebug plot showing function execution time:

	Calls	Percent ▼	Own Time	Time	Avg	Min	Max
_yield	137180	22.97%	7085.719ms	477872.583ms	3.484ms	0.076ms	118.495ms
_yieldCont	109960	12.76%	3936.668ms	393977.801ms	3.583ms	0.075ms	118.682ms
___append	137180	10.98%	3387.833ms	3387.833ms	0.025ms	0.021ms	110.323ms
lsMapIgnore	23521	4.87%	1502.319ms	83213.99ms	3.538ms	0.147ms	91.952ms
putPixel	26101	4.02%	1239.515ms	89203.568ms	3.418ms	0.17ms	118.784ms
lsMapIgnore/</<	23081	3.65%	1125.141ms	82204.579ms	3.562ms	0.143ms	92.127ms
scalePoint_1690	24181	3.01%	929.178ms	82979.916ms	3.432ms	0.113ms	118.122ms
putPixel/<	26101	2.75%	848.897ms	87478.673ms	3.352ms	0.107ms	118.717ms
plotPoint_1741/<	23861	2.67%	825.009ms	82418.115ms	3.454ms	0.112ms	118.042ms
plotPoint_1741	23861	2.65%	818.001ms	79416.593ms	3.328ms	0.111ms	118.235ms
lsMapIgnore/<	23081	2.59%	800.367ms	80934.866ms	3.507ms	0.111ms	91.115ms
plotPoint_1741/</<	23861	2.51%	775.149ms	81955.139ms	3.435ms	0.108ms	118.645ms
ignore	23161	2.48%	766.367ms	78996.921ms	3.411ms	0.108ms	90.987ms
_lsCons	35980	2.45%	754.891ms	754.891ms	0.021ms	0.018ms	43.048ms
_lsTail	26301	1.81%	559.628ms	559.628ms	0.021ms	0.017ms	81.366ms
_lsEmpty	28760	1.6%	495.133ms	495.133ms	0.017ms	0.016ms	0.082ms
_jsFillRect	26101	1.57%	485.38ms	485.38ms	0.019ms	0.017ms	0.67ms
_lsHead	26381	1.53%	473.475ms	473.475ms	0.018ms	0.016ms	0.593ms
_lsFromArray	80	1.28%	396.103ms	975.939ms	12.199ms	2.14ms	64.732ms
handleMessage	4535	0.73%	224.342ms	30851.288ms	6.803ms	0.179ms	118.923ms
lsMap	3260	0.7%	217.486ms	12019.063ms	3.687ms	0.175ms	118.415ms
_objectEq	12034	0.7%	214.571ms	214.571ms	0.018ms	0.016ms	0.07ms
setZeroTimeout	4535	0.67%	207.151ms	207.151ms	0.046ms	0.034ms	0.538ms

Figure 5.36: The time that all the calls to `LINKS.eq` took was slightly reduced by using specialized functions for comparison – here only `objectEq` (at the bottom) took any significant time – compare that to `LINKS.eq` execution time (at the top on 5.26)

We see that obviously comparing objects (`_objectEq`) took the most time. The rest of the comparison functions, like `_intEq` are not in the picture as they took much less time.

We can also observe that `lsMapIgnore`, which is my custom native JavaScript function that works similarly to `map` – but is only interested in side effects, so it does not have to construct and return a list – was a significant optimization. Compare its average time of 3.5 ms with `map`'s almost 6 ms (seen on 5.26).

Similarly the other list manipulating functions (`_ls*`) achieved better or at least as good performance as the original versions.

5.5.11 _yield

The last and the most significant optimization was to the `_yield` function. I removed calls to `__append` and `apply` and replaced the three arguments to `_yield` with a single lambda argument. This optimization required modifying the code generated by the compiler (a few lines in *irtojs.ml*[26] had to be added or adjusted). This optimization was applied on top of all the previous ones.

Before any optimizations, `_yield` was defined more or less like so:

```
function _yield(f, args, k) {
  DEBUG.assert_noisy(DEBUG.is_function(f),
    "_yield: 1st arg expected a function, got: " + f);
  DEBUG.assert_noisy(DEBUG.is_function(k),
    "_yield: 3rd arg expected a function, got: " + k);
  DEBUG.assert(DEBUG.is_array(args),
    "_yield: 2nd arg expected an array, got: " + args);

  var argslen = args.length;
  var arguments = ___append(args, k);

  ++_yieldCount;

  if ((_yieldCount \% _yieldGranularity) == 0) {
    if (!_handlingEvent) {
      var current_pid = _current_pid;
      setTimeout((function() {
                      _current_pid = current_pid;
                      return f.apply(f,
                                ___append(args, k))}),
                 0);
    }
    else {
      throw new _Continuation(
                  function () {
                      return f.apply(f,
                                ___append(args, k));
                  }
                );
    }
  }
  else {
    return f.apply(f, ___append(args, k));
  }
}
```

[26]See Chapter 3, 3.3.2 for the description of the file

After all optimizations:

```
function _yield(f) {
        ++_yieldCount;
        if (_yieldCount == _yieldGranularity) {
                _yieldCount = 0;
                if (_handlingEvent) {
                        _theContinuation.v = f;
                        throw _theContinuation;
                } else {
                        var current_pid = _current_pid;
                        window.setZeroTimeout(
                          function() {
                            _current_pid = current_pid;
                            return f()
                          }
                        );
                }
        } else {
                return f();
        }
}
```

We can see that the signature of the function changed – it now takes only one argument. I removed calls to the `DEBUG` functions, got rid of a modulo operation and a negation. I optimized away all calls to `apply` and `___append` and all unnecessary local variables. I also replaced the call to `setTimeout` with the call to the faster `setZeroTimeout` and I am no longer creating a new `Continuation` object each time the function is called, but reusing a global object (`_theContinuation`) instead.

`___append` was defined as follows:

```
function ___append(xs, x) {
  var out = [];
  for (var i = 0; i < xs.length; i++) {
    out[i] = xs[i];
  }
  out[i] = x;
  return out;
}
```

Removing calls to it means that there should be much less unnecessary temporary arrays, which effectively means less garbage.

Below I present charts generated in Chromium and Firefox after this optimization:

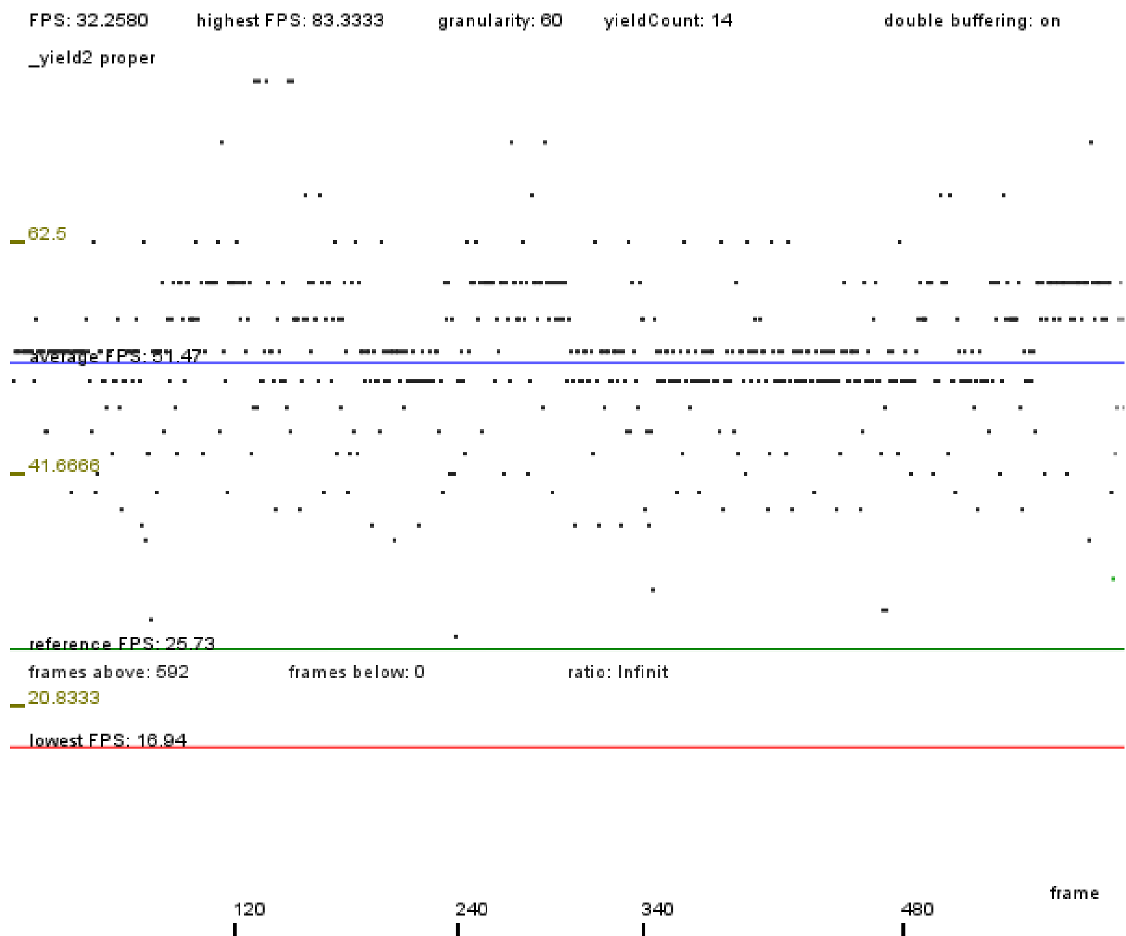


Figure 5.37: `_yield` optimization in Chromium

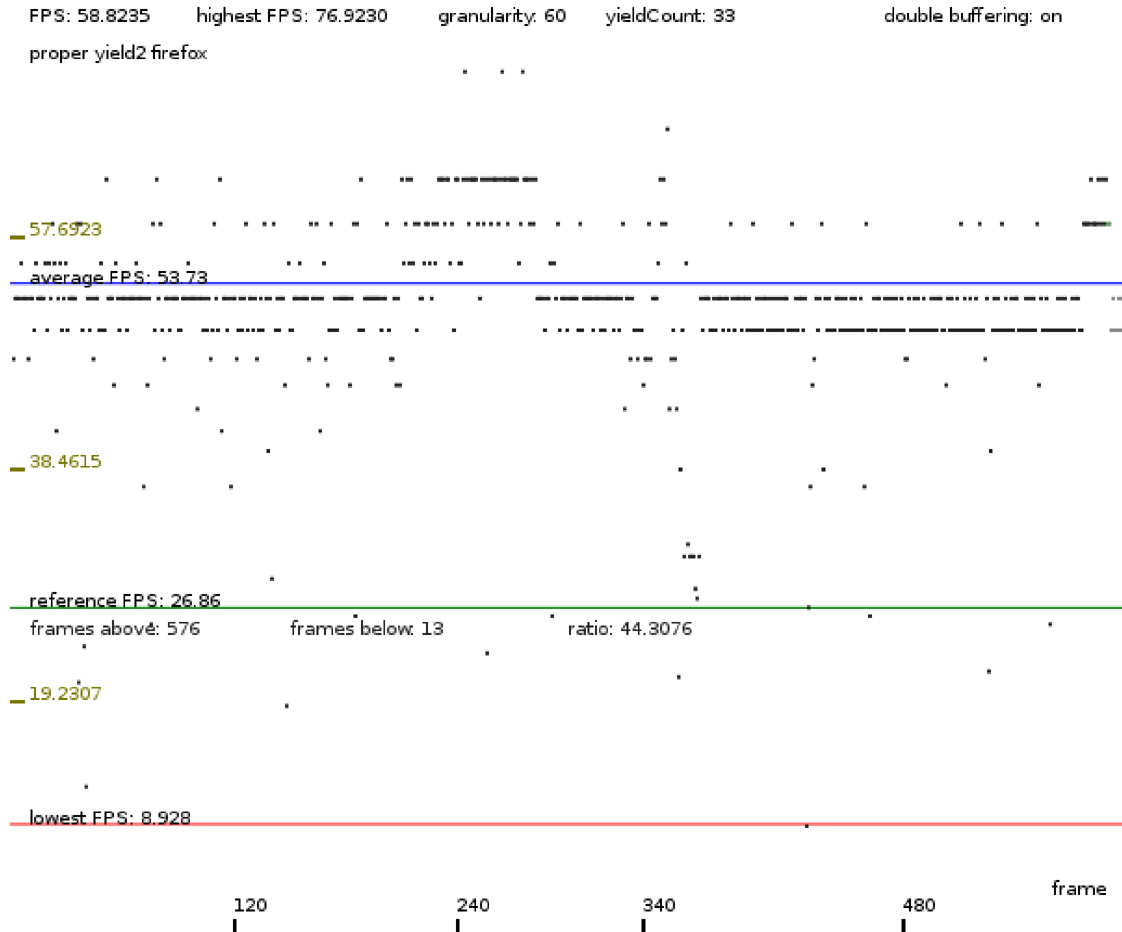


Figure 5.38: `_yield` optimization in Firefox

The oscillation gap shrank significantly and an average frame rate of over 50 FPS was achieved – almost double the baseline frame rate.

It is interesting to compare those charts as we can see slight differences between garbage collectors in these browsers. The data point distribution is different – particularly the more abrupt drops in frame rate reveal when the garbage collector is run. In Chromium there is generally less "big" drops, which signify periodical collections of large amounts of garbage. Both browsers use the same type of GC[27].

[27] `https://blog.mozilla.org/nnethercote/2014/03/31/generational-gc-has-landed/`

Comparing timeline plots before and after the optimization:

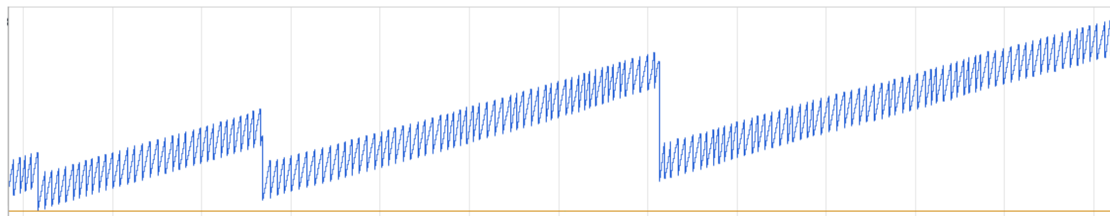


Figure 5.39: Timeline plot **before** the `_yield` optimization; 1 minute, 10.8-90.2 MB

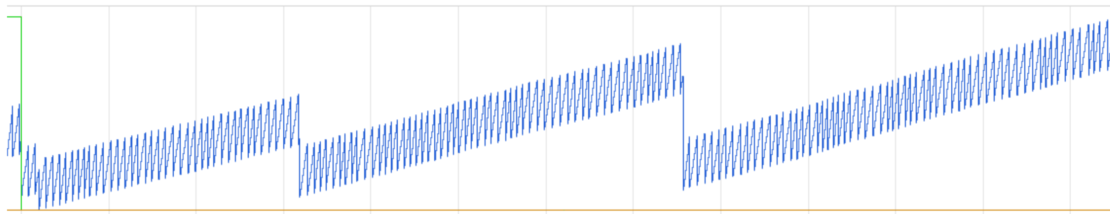


Figure 5.40: Timeline plot **after** the `_yield` optimization; 1 minute, 6.4-65 MB

we see that the size of the heap in time decreased significantly. The maximum registered size was 65 MB, which is almost 30 MB less than before.

***

Heap allocation record:

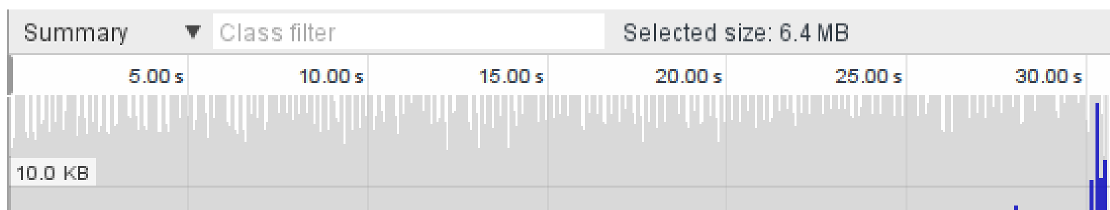


Figure 5.41: Heap allocations after the `_yield` optimization

We see that the reference size dropped further, from 50 KB (5.34) to 10 KB. Otherwise no significant changes.

And execution time:

	Calls	Percent ▼	Own Time	Time	Avg	Min	Max
_yield2	165185	13.91%	5205.393ms	573939.758ms	3.475ms	0.07ms	77.147ms
_yieldCont	135920	11.58%	4332.607ms	489277.252ms	3.6ms	0.071ms	77.227ms
putPixel	38465	4.86%	1819.33ms	133761.883ms	3.477ms	0.177ms	46.064ms
_lsCons	82977	4.24%	1586.679ms	1586.679ms	0.019ms	0.017ms	2.337ms
putPixel/<	38465	3.45%	1289.77ms	130851.235ms	3.402ms	0.106ms	45.986ms
scalePoint_1687	35663	3.4%	1273.085ms	118482.167ms	3.322ms	0.107ms	46.235ms
lsMapIgnore	24616	3.23%	1206.915ms	88030.293ms	3.576ms	0.139ms	37.123ms
plotPoint_1738/</<	35196	3.15%	1178.661ms	117385.763ms	3.335ms	0.104ms	46.163ms
plotPoint_1738/</</<	35196	3.13%	1170.9ms	118977.242ms	3.38ms	0.209ms	46.097ms
plotPoint_1738	35196	3.12%	1168.553ms	115544.688ms	3.283ms	0.106ms	46.334ms
plotPoint_1738/</</</<	35196	3.07%	1150.254ms	116714.379ms	3.316ms	0.105ms	45.918ms
lsMapIgnore/<	23973	3.05%	1139.371ms	85225.882ms	3.555ms	0.17ms	37.017ms
lsMapIgnore/</</<	23973	3.01%	1124.624ms	86736.398ms	3.618ms	0.201ms	37.192ms
plotPoint_1738/<	35196	2.95%	1103.322ms	117097.326ms	3.327ms	0.138ms	46.267ms
_jsFillRect	38465	2.92%	1091.319ms	1091.319ms	0.028ms	0.022ms	0.574ms
_lsEmpty	41804	2.1%	786.086ms	786.086ms	0.019ms	0.016ms	46.079ms
lsMapIgnore/</<	23973	2.07%	773.786ms	85397.617ms	3.562ms	0.104ms	37.26ms
_lsFromArray	116	2%	749.553ms	1695.019ms	14.612ms	7.602ms	51.254ms
lsMap/<	14782	1.97%	736.468ms	49826.756ms	3.371ms	0.169ms	46.4ms
lsMap	14841	1.93%	723.106ms	49896.956ms	3.362ms	0.136ms	53.874ms
lsMap/</</<	14783	1.9%	711.75ms	50680.895ms	3.428ms	0.2ms	53.956ms
lsMap/</</</<	14986	1.86%	694.956ms	46046.824ms	3.073ms	0.136ms	8.649ms
_lsTail	38756	1.83%	685.066ms	685.066ms	0.018ms	0.016ms	0.533ms

Figure 5.42: Execution time after `_yield` optimization

Here `_yield2` is the optimized version of `_yield`. Its average execution time is 3.475 ms, compared to 6.124 ms before (see 5.26) – almost twofold improvement, which is reflected in the frame rate. Note again that `___append` was optimized away from `_yield`.

The final frame rate, after all optimizations, is 51 FPS. Almost 32 times more than in the beginning, before I introduced any optimizations (5.3).

And there is still room for a lot more (and more advanced) optimizations. I discuss some of them in Chapter 6.

5.6 Observations and summary

Simple optimizations, notably those presented in sections 5.4.1, 5.4.2, 5.4.3, 5.5.9, 5.5.10 and 5.5.11 of this Chapter, increased performance significantly.

The biggest performance issues were caused by inefficiencies in the scheduler mechanism of the Links language and garbage collector slowdowns.

Large amounts of garbage were produced because of the inefficient implementation of the basic data structure in the language – the list – which caused a lot of unnecessary copying of JavaScript arrays (used to represent the data structure).

These problems were largely improved by optimizations of `setTimeout` and `_yield*` functions and by implementing a custom linked list structure and using it instead of the original.

***

Figure 5.43 on the next page presents comparison of heap object statistics between the native JavaScript version of the benchmark (see 5.5.4), the baseline Links version (see 5.5.2) and the Links version after all optimizations.

We can see that overall the Links versions produce a lot more objects and take up much more memory than the native one.

Comparing the Links versions before and after optimizations, the amount of `Array` objects decreased and the amount of `Object` objects increased. This is because I replaced the `Array`-based list representation with a faster linked list implementation (5.5.9).

Also, the retained size of `(closure)` objects dropped while their amount stayed the same. Large amount of closures is created by `_yield*` (5.5.11). The last optimization got rid of the need to call `___append`, which produced temporary arrays in the closures, thus decreasing the retained size.

	native		baseline		optimized	
objects count						
(array)	4 459	11 %	11 867	21 %	10 839	15 %
(closure)	2 703	7 %	6 341	11 %	6 831	10 %
(compiled code)	1 921	5 %	4 170	7 %	4 434	6 %
Object	1 843	5 %	2 735	5 %	5 875	8 %
system / Context	123	0 %	1 703	3 %	582	1 %
Array	521	1 %	1 840	3 %	1 118	2 %
(system)	11 824	29 %	18 414	32 %	21 715	31 %
Window	1	0 %	1	0 %	3	0 %
(string)	3 007	7 %	5 420	9 %	4 885	7 %
shallow size						
(array)	844 152	22 %	2 365 008	39 %	2 088 688	31 %
(closure)	194 616	5 %	456 552	8 %	491 832	7 %
(compiled code)	553 824	15 %	1 123 816	19 %	1 157 960	17 %
Object	69 896	2 %	88 184	1 %	216 496	3 %
system / Context	9 056	0 %	104 824	2 %	39 120	1 %
Array	16 672	0 %	58 880	1 %	35 776	1 %
(system)	383 960	10 %	623 392	10 %	705 336	11 %
Window	88	0 %	88	0 %	264	0 %
(string)	152 592	4 %	241 616	4 %	237 784	4 %
retained size						
(array)	1 009 568	26 %	2 747 632	46 %	2 487 240	37 %
(closure)	524 688	14 %	2 164 744	36 %	1 479 424	22 %
(compiled code)	855 376	22 %	1 710 632	28 %	1 715 912	26 %
Object	710 624	19 %	1 393 472	23 %	1 800 576	27 %
system / Context	52 936	1 %	1 372 856	23 %	560 648	8 %
Array	61 168	2 %	1 269 424	21 %	252 792	4 %
(system)	625 504	16 %	1 025 696	17 %	1 244 608	19 %
Window	72 808	2 %	400 304	7 %	444 448	7 %
(string)	152 592	4 %	241 688	4 %	237 856	4 %

Figure 5.43: Comparison of heap object statistics between JavaScript, baseline and optimized versions. Significant information is underlined.

This is the comparison of heap allocation records:

Figure 5.44: The optimizations decreased the amount of memory used by the benchmark (note the reference size: 100, 50, and 10 KB)

5.7 Performance in games

After all the optimizations I was able to achieve satisfactory frame rate in all of the implemented applications.

The performance of Links before my optimizations was insufficient for even the simplest games (like a Breakout clone) to be playable.

The optimizations allowed to increase the frame rate, which provided the necessary smoothness of animations and responsiveness of user's input. The frame rate above the assumed minimum (30 FPS) was consistently achieved in the most complex of the implemented games.

This means that all of the practical goals of this thesis were completed.

Here is a comparison of the frame rate before:

Figure 5.45: Frame rate in my Tetris clone – before any optimizations – 3 FPS on average

And after optimizations:

Figure 5.46: Frame rate in my Tetris clone – after all optimizations – the average frame rate is 330-430 FPS, over 100 times more

For a simple game like Tetris the frame rate improved tremendously. From a few to a few hundred FPS.

Playability was achieved also in the most complex of the implemented games – the Pac-Man clone:

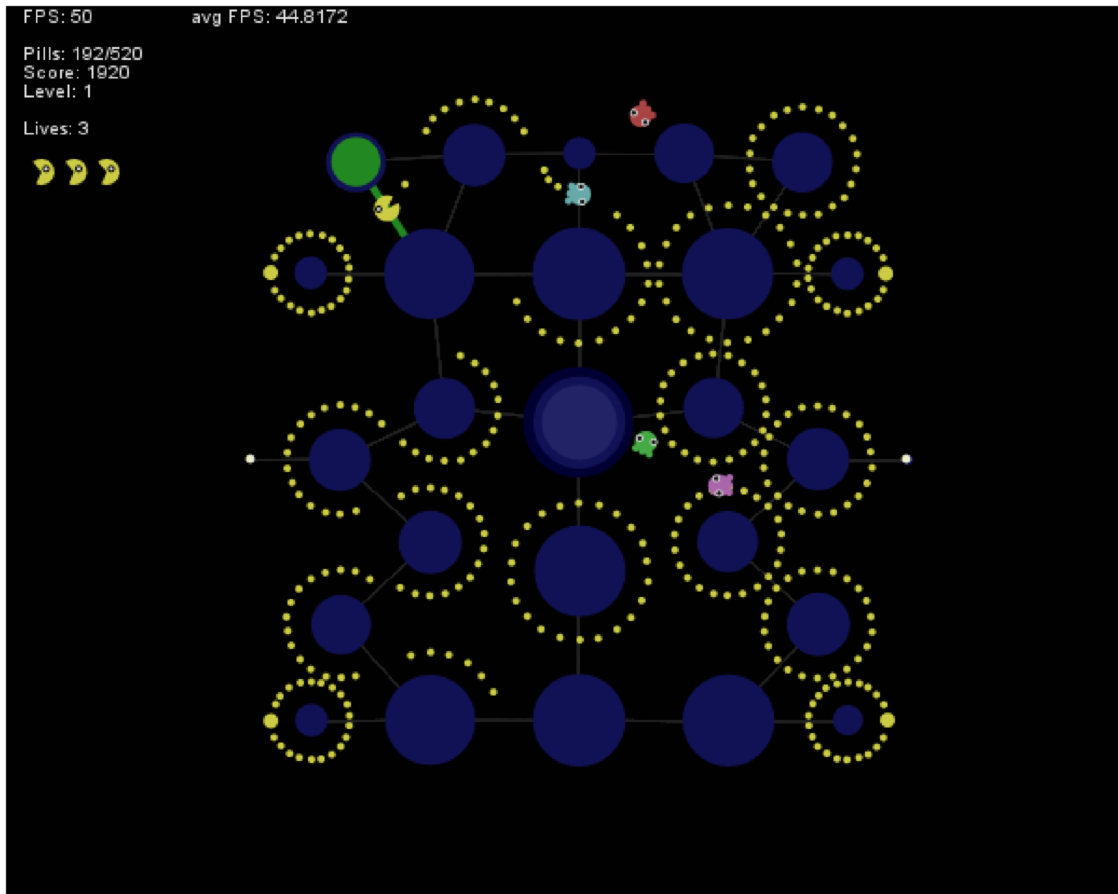


Figure 5.47: Running my Pac-Man clone after optimizations; the average frame rate is 40-50 FPS

Chapter 6

Summary and conclusions

In this thesis I combined the topics of functional programming, web game development and programming language optimization.

I developed four web games (see Chapter 4) in Links – an experimental functional web programming language – and by optimizing its runtime system achieved playability (which was not possible before), quantified by satisfactory frame rate. I also wrote a benchmark application to measure the effects of each of my optimizations (described in Chapter 5)[1].

In the process of making this thesis I learned a lot about functional programming – the paradigm in general – as well as familiarized myself on various levels with different functional languages, particularly Haskell, OCaml, and obviously Links. I also broadened my knowledge of JavaScript and web application development.

Learning about all these subjects and using my knowledge in practice was very challenging and exciting and I am sure that all of this was a valuable step towards becoming a better engineer.

All of the thesis goals (stated in Chapter 1, 1.6) were fulfilled.

[1] Both the games and the benchmark application are available online:
`https://rawgit.com/slindley/links/dariusz/examples/index.html`
To run precompiled versions straight in the browser click on *compiled version* links in the *Game examples* and *Performance* sections. Links to compiled versions of the games are also available here:
`https://rawgit.com/slindley/links/dariusz/examples/games.html`.

The following charts illustrate the effectiveness of introduced optimizations:

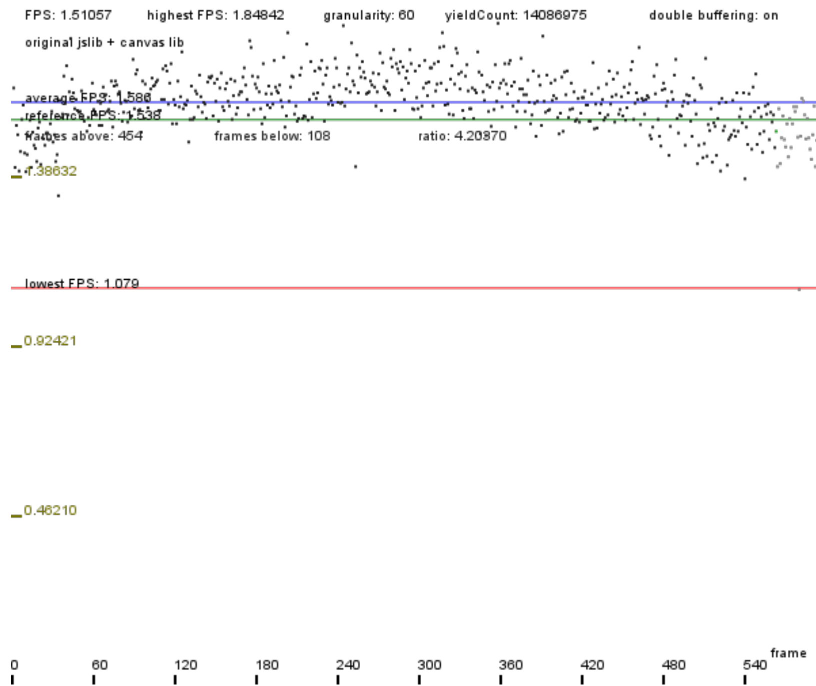


Figure 6.1: Before: average frame rate is 1.6 FPS

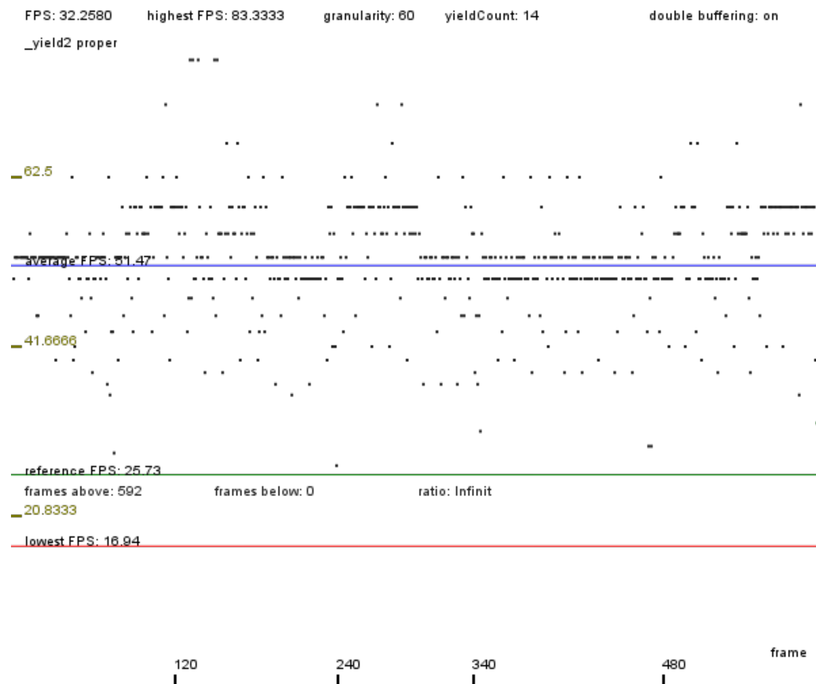


Figure 6.2: After: average frame rate is 51 FPS

The frame rate in the benchmark application increased from 1.6 FPS to 51 FPS (3188 %). Similar improvement was observed in games – see Chapter 5, 5.7.

6.1 Conclusions

The work that I did in the scope of this thesis allowed me to confirm in practice a lot of engineering principles [21] related to software optimization:

- When optimizing it is crucial to analyze performance and detect bottlenecks.
- We should focus our optimization efforts on the most significant issues, where we can get the biggest performance gains.
- Profiling is an immensely valuable tool for performance analysis.
- We can optimize on many levels: design, algorithms, data structures, source code, compiler, runtime system.
- Sometimes more complex algorithms perform well on large amounts of data, but the cost of their initialization, setup, etc. can cancel out or outweigh the benefits for small amounts of data – in such case simple algorithms prove to be more suitable.

From working with different languages and paradigms (functional and imperative) I conclude that each has its upsides and downsides and a pragmatic approach – combining multiple solutions and picking the best-suited tool – is the most effective.

In theory, all Turing-complete [19] programming languages have effectively the same capabilities. But in each one, these capabilities are shaped into a different tool, suited to do certain tasks better than other.

If we are focusing on practical results, the most effective way is to use languages and paradigms not conservatively, but pragmatically. Mixing together different approaches and features, picking that, which does the job that needs to get done best.

6.2 Future work

In terms of optimizations to the Links language these are possible directions that one might consider to further improve the performance of the language:

- Write applications in a way that avoids generating garbage. For Links it means enabling (more) ways to do it as well as optimizing the compiler and the runtime, so that less garbage is being generated.
- Implement a custom garbage collector, give the user more control over memory.
- Make the compiler generate code for a language that compiles to LLVM bytecode, then generate fast JavaScript from it using Emscripten[2] (or something similar). This would most likely also mean implementing a custom garbage collector.
- Optimize JavaScript generated by the compiler.
- Further optimize some Links' data structures and functions. The optimized linked list type should be polished and adapted to the language.
- Introduce more advanced optimizations to lists, such as list fusion [22] and deforestation [20].
- Implementing a few levels of debugging could be considered.
- Specialized equality functions seem to improve the performance, so they should replace `LINKS.eq`.
- The continuation-passing style JavaScript generated by the compiler could be optimized by adapting a more optimal CPS representation [15].
- Good place to look for ready-made solutions are other languages, similar to Links, like Elm and Opa.

In terms of programming language development I believe that the trend for incorporating the functional paradigm and features into the mainstream will continue.

In the area of functional game development there is definitely a lot of future work to be done and I believe (as outlined in Chapters 1 and 3) that the combination of computationally intensive applications – such as computer games – and functional programming will produce a lot of innovation in the future.

I intend to continue my work and research in these subjects.

[2] `http://en.wikipedia.org/wiki/Emscripten`

Bibliography

[1] Links basic documentation. http://groups.inf.ed.ac.uk/links/quick-help.html.

[2] Andrew W. Appel. A Runtime System. https://users-cs.au.dk/hosc/local/LaSC-3-4-pp343-380.pdf, may 1989.

[3] Davide Aversa. In search of the "Philosopher's Code". http://www.davideaversa.it/2014/08/in-search-of-the-philosophers-code/, August 2014.

[4] Arjan Boeijink, Philip K.F. Hölzenspies, and Jan Kuper. Introducing the pilgrim: A processor for executing lazy functional languages. In Jurriaan Hagen and Marco T. Morazán, editors, Implementation and Application of Functional Languages, volume 6647 of Lecture Notes in Computer Science, pages 54–71, Berlin, Germany, 2011. Springer Verlag.

[5] John Carmack. In-depth: Functional programming in C++. http://gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php, 2012.

[6] John Carmack. QuakeCon 2013 Keynote, part 4. https://www.youtube.com/watch?v=1PhArSujR_A, 2013.

[7] John Carmack. QuakeCon 2013 Keynote, part 5. https://www.youtube.com/watch?v=cWA_9L70moE, 2013.

[8] Mun Hon Cheong. Functional Programming and 3D Games. Master's thesis, University of New South Wales, Sydney, Australia, November 2005.

[9] Philip Wadler Ezra Cooper, Sam Lindley and Jeremy Yallop. Links: Web Programming Without Tiers. volume 4709 of Lecture Notes in Computer Science, November 2006.

[10] Jake Gordon. Javascript Game Foundations – The Game Loop. http://codeincomplete.com/posts/2013/12/4/javascript_game_foundations_the_game_loop/, December 2013.

[11] James Hague. Purely Functional Retrogames. http://prog21.dadgum.com/23.html, http://prog21.dadgum.com/24.html, http://prog21.dadgum.com/25.html, http://prog21.dadgum.com/26.html, http://prog21.dadgum.com/37.html, 2008, 2009. [Series of articles].

[12] James Hague. Functional Programming Doesn't Work. http://prog21.dadgum.com/54.html, http://prog21.dadgum.com/55.html, 2009, 2010. [Series of articles].

[13] Pieter J. Mosterman Katalin Popovici. Real-Time Simulation Technologies: Principles, Methodologies, and Applications, 2012.

[14] Miran Lipovača. Learn You a Haskell for Great Good. http://learnyouahaskell.com/, 2011.

[15] Florian Loitsch. Exceptional Continuations in JavaScript. In 2007 Workshop on Scheme and Functional Programming, September 2007.

[16] Robert Nystrom. Game Programming Patterns. http://gameprogrammingpatterns.com, November 2014.

[17] Mark Overmars. A Brief History of Computer Games. http://www.cs.uu.nl/docs/vakken/b2go/literature/history_of_games.pdf, 2012.

[18] Tim Sweeney. The Next Mainstream Programming Language: A Game Developer's Perspective. https://www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/docs/sweeny.pdf, 2006.

[19] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, s2-42(1):230–265, 1937.

[20] Philip Wadler. Deforestation: transforming programs to eliminate trees. pages 231–248, 1990.

[21] Bob Wescott. Every Computer Performance Book: How to Avoid and Solve Performance Problems on The Computers You Work With. CreateSpace Independent Publishing Platform, USA, 1st edition, 2013.

[22] H. Iwasaki M. Takeichi Y. Onoue, Z. Hu. A Calculational Fusion System HYLO. pages 76–106, Le Bischenberg, France, February 1997.

[23] Boris Zbarsky. [whatwg] Canvas size and double buffering. http://www.mail-archive.com/whatwg@lists.whatwg.org/msg19969.html, Feb 2010.

Glossary

AAA AAA video games are the ones with highest budgets for development and promotion, which usually means very high quality.
http://en.wikipedia.org/wiki/AAA_%28game_industry%29. 13

continuation-passing style It is a style of programming. A function in CPS always takes as one of its arguments a function of one argument – called a continuation. After performing its computation, when it has its output value ready, instead of returning it to the caller, it calls the continuation, passing the output value to it. The continuation represents computations that are to be performed after the function has completed. CPS is often used by compilers as an intermediate representation.
http://en.wikipedia.org/wiki/Continuation-passing_style
http://en.wikisource.org/wiki/Scheme:_An_Interpreter_for_Extended_Lambda_Calculus/Section_3#Continuation_Passing_Recursion . 15, 17, 34, 78, 104

first-person shooter FPS is a video game genre in which the player explores the game world from a first-person perspective, through the eyes of the character he or she is controlling. "Shooter" means that the main focus of the game mechanics is projectile weapon-based combat.
http://en.wikipedia.org/wiki/First-person_shooter. 14

frame rate The number of frames (images) displayed every second. Measured in frames per second (FPS). Producing higher frame rates requires more processing power and means smoother animation.
http://en.wikipedia.org/wiki/Frame_rate. 1

functional reactive programming FRP is a paradigm that incorporates ideas of time flow and compositional events into functional programming. This facilitates writing interactive animations, user interfaces, simulations, games and other interactive programs.
http://en.wikipedia.org/wiki/Functional_reactive_programming
https://www.haskell.org/haskellwiki/Functional_Reactive_Programming
http://elm-lang.org/learn/What-is-FRP.elm. 2, 14

heads-up display HUD is the area of the game screen, which contains information relevant to the player, like current game level, score, health, etc. It is a part of the user interface.
http://en.wikipedia.org/wiki/HUD_(video_gaming). 43

instantaneous frame rate Instantaneous means that it indicates how many frames could be processed in a second, if all frames in that second took as much time to process as the current frame.. 47, 51

purely functional language A language that has variables defined in a mathematical sense, where identifiers refer to immutable values. Computations performing side effects are treated specially in such language – they can be represented using structures called monads.
http://en.wikipedia.org/wiki/Purely_functional. 11

web game (Browser game) is a type of computer game that is implemented on top of a web browser, usually using web technologies such as JavaScript, HTML and CSS.
http://en.wikipedia.org/wiki/Browser_game. 1

Acronyms

AI Artificial Intelligence. 38

AJAX Asynchronous JavaScript and XML. 5

FP Functional Programming. 12, 13

FPS Frames Per Second. 28, 37

FRP Functional Reactive Programming. 2

GC Garbage Collector. 13, 63

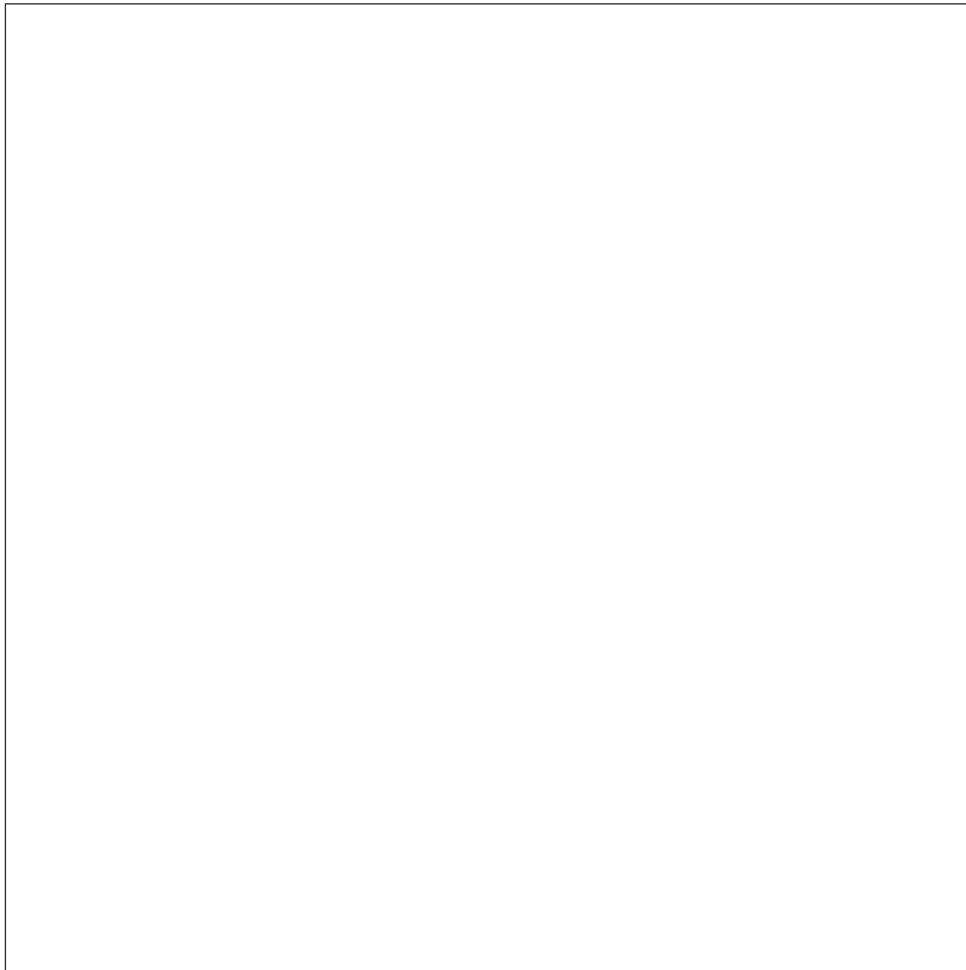
HUD Heads-Up Display. 43

IDE Integrated Development Environment. 8

Appendix A

Attached files

A.1 DVD



The attached DVD contains the following directories:

dist – compiled versions of all applications written for this thesis; these can be run in a web browser. This directory includes a copy of Links' runtime libraries (in the **lib** subdirectory).

doc – electronic version of this thesis (in PDF format) and a presentation from diploma seminar (in Polish).

src – source files of all applications written for this thesis.

files – all the files described in A.2

The next section describes all relevant files that were used during performance measurements and optimizations: the benchmark application as well as the runtime.

All of these files are contained within the `files` directory on the attached DVD.

Different versions of the runtime were produced by modifying the original. This allowed me to easily test different optimizations in isolation as well as in combination. For inspecting and comparing, I suggest using a `diff` tool on different versions of the files.

The names of the attached files approximately correspond to descriptions (if present) found on charts in Chapter 5.

A.2 Files used in optimizations and benchmarking

Various versions of the benchmark application are attached to this document as the following files:

- *performance-frozen.links* – the original benchmark application in Links (most charts in this document were generated by it)
- *performance-frozen-optimized.html* – a version of the original benchmark optimized by the Google Closure Compiler[1]
- *performance.html* – native JavaScript version of the benchmark application
- *BASE performance-frozen-lists.links* – *performance-frozen.links* with a custom list type defined in Links. This is the base for all **performance-frozen-lists* files.
- *TAKE-DROP performance-frozen-lists.links* – uses JavaScript versions of `lsTake` and `lsDrop` (which work like `take` and `drop`) defined in *JS lists 2 - map jslib.js*
- *JS LISTS 2 - MAP performance-frozen-lists.links* – uses a custom JavaScript linked list type (defined in *JS lists 3 - map jslib.js*)

[1] `https://developers.google.com/closure/`

- *JS LISTS 3 - MAP performance-frozen-lists.links* – uses a custom optimized JavaScript linked list type (defined in *JS lists with null - map jslib.js*)
- *specialized equality performance-frozen-lists.links* – uses specialized functions for testing equality (defined in specialized equality *jslib.js*)

Various versions of the Links runtime (jslib.js)[2]:

- *original jslib + canvas lib.js* – the original (unoptimized) version of *jslib.js* which was used as a reference – I added only the interface for canvas manipulating functions to it. I used the *jslib.js* file from GitHub – from the version of sessions branch (last commit July 30), which my branch (dariusz[3]) was derived from.
- *optimized `_yield` and `_yieldCont` jslib.js* – the original with optimized versions of `_yield` and `_yieldCont` functions – the optimization removed any references to functions in the `DEBUG` namespace from the body of `_yield` and `_yieldCont` and made some other minor changes
- *setZeroTimeout jslib.js* – the original with all calls to `setTimeout` with the second argument of 0 replaced by a call to `setZeroTimeout`[4]
- *`_yieldGranularity` + 200 jslib.js* – the original with `_yieldGranularity` increased from 60 to 260
- *new base jslib.js* – the original with optimizations from *optimized `_yield` and `_yieldCont` jslib.js* and *setZeroTimeout jslib.js* combined
- *optimized yield + setZeroTimeout jslib.js* – same as previous
- *new base + `_yieldGranularity` + 200 jslib.js – new base jslib.js* with `_yieldGranularity` increased from 60 to 260
- *google closure jslib.js* – modified new *base jslib.js* with a function for invoking Chromium debugger; this file was used as part of the input to Google Closure Compiler; the whole input is attached in the file *google closure input.js*
- *JS lists 2 - map jslib.js* – adds a few functions for manipulating the linked list type defined in *JS LISTS 2 - MAP performance-frozen-lists.links*
- *JS lists 3 - map jslib.js* – defines a linked list type (based on the one in the Elm language[5]) and functions for manipulating it entirely in JavaScript. Used with *JS LISTS 3 - MAP performance-frozen-lists.links*

[2]The names of the attached files approximately correspond to descriptions (if present) found on charts in Chapter 5

[3]`https://github.com/slindley/links/compare/dariusz`

[4]Implementation from:
`http://dbaron.org/log/20100309-faster-timeouts`

[5]`http://elm-lang.org/elm-runtime.js`

- *JS lists with null - map jslib.js* – *JS lists 3 - map jslib.js* with further optimizations of the linked list type
- *specialized equality jslib.js* – adds specialized JavaScript functions for testing for equality as (theoretically faster) an alternative to `LINKS.eq`
- *proper yield2 jslib.js* – adds an optimized version of `_yield` that required a slight change in the JavaScript generated by the compiler (*irtojs.ml* was adjusted for this optimization – it is attached to this document)

Significant Links' compiler source files:

- *lib.ml* – the original *lib.ml* + interface for canvas manipulation
- *lib 2.ml* – the above *lib.ml* + interface for manipulating linked lists and specialized equality functions
- *irtojs.ml* – the original *irtojs.ml* adjusted for an optimized version of `_yield`

List of Figures

4.1 The puzzle game 2048 implemented in Links 24
4.2 A Breakout clone in Links 25
4.3 Links version of the classic Tetris 26
4.4 My variation of Pac-Man written in Links 27
4.5 A screenshot showing all of the contents of a web page that contains the game 28
4.6 What the player sees after launching the game (entering a web page that contains it) 29

5.1 An example chart 50
5.2 The unoptimized version 52
5.3 First optimization – faster `_yield*` 53
5.4 Second optimization – `setZeroTimeout` 55
5.5 Third optimization – calling `setTimeout` less often 56
5.6 Fourth optimization – double buffering off 57
5.7 First two optimizations combined 58
5.8 First three optimizations combined 59
5.9 First two and the fourth optimization combined 60
5.10 Invoking GC after mapping a function over a big list to clean up stabilizes the frame rate; optimizing the implementation of `map` may significantly boost performance 63
5.11 Too high `_yieldGranularity` results in more GC slowdowns 64
5.12 Example timeline chart: 1 minute, 10.8-90.2 MB 65
5.13 Example heap allocation record; note that this was recorded over about 30 seconds 66
5.14 Example heap object statistics 66
5.15 Example execution time profile 67
5.16 This is the baseline for all charts that follow 69
5.17 Performance of the code optimized with the Closure Compiler . . . 70
5.18 Chromium profiler's heap timeline plot **before** Closure Compiler's optimizations (the frame rate was 28 FPS); 4 minutes, 8.3-220 MB . 71
5.19 Chromium profiler's heap timeline **after** the code was optimized with the Closure Compiler (the frame rate was 29 FPS); 4 minutes, 6.7-174 MB 71
5.20 Timeline for the native JavaScript version; 2 minutes, 4-23.8 MB . . 72

5.21 Native JavaScript version, heap allocations 72
5.22 Links version (the one used to generate the baseline – see 5.5.2), heap allocation record 72
5.23 Native JavaScript version, heap objects 73
5.24 Links version (the one used to generate the baseline – see 5.5.2), heap objects 73
5.25 A chart generated by the JavaScript version of the benchmark . . . 74
5.26 Profiling execution time of my Breakout clone 75
5.27 Linked list type implemented in Links 77
5.28 Linked lists with `take` and `drop` in JS 80
5.29 Linked lists with `take` and `drop` in JS – curious case 81
5.30 Linked lists implemented entirely in JS (except `map*`) 82
5.31 Native JavaScript linked lists – less heap allocations 83
5.32 Native JavaScript linked lists – heap objects 83
5.33 Further optimized native JavaScript linked lists 84
5.34 Heap allocations after linked list optimizations 85
5.35 Using specialized functions for equality adds some more frames per second 86
5.36 The time that all the calls to `LINKS.eq` took was slightly reduced by using specialized functions for comparison – here only `objectEq` (at the bottom) took any significant time – compare that to `LINKS.eq` execution time (at the top on 5.26) 87
5.37 `_yield` optimization in Chromium 90
5.38 `_yield` optimization in Firefox 91
5.39 Timeline plot **before** the `_yield` optimization; 1 minute, 10.8-90.2 MB 92
5.40 Timeline plot **after** the `_yield` optimization; 1 minute, 6.4-65 MB . 92
5.41 Heap allocations after the `_yield` optimization 92
5.42 Execution time after `_yield` optimization 93
5.43 Comparison of heap object statistics between JavaScript, baseline and optimized versions. Significant information is underlined. 95
5.44 The optimizations decreased the amount of memory used by the benchmark (note the reference size: 100, 50, and 10 KB) 96
5.45 Frame rate in my Tetris clone – before any optimizations – 3 FPS on average 97
5.46 Frame rate in my Tetris clone – after all optimizations – the average frame rate is 330-430 FPS, over 100 times more 98
5.47 Running my Pac-Man clone after optimizations; the average frame rate is 40-50 FPS 99

6.1 Before: average frame rate is 1.6 FPS 102
6.2 After: average frame rate is 51 FPS 102