CS407 April 2013

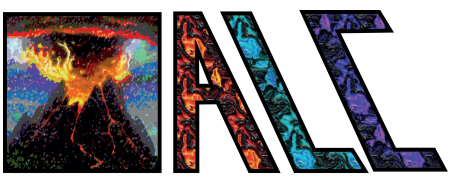
Game Development in Haskell — A Practical Investigation

Group Report

Authors: Laith Alissa — 0918520
Jon Cave — 0907931
Joseph Siddall — 0935112
Vic Smith — 0931968

Supervisor: Sara Kalvala

Client: Matthew Leeke



Authors Laith Alissa, Jon Cave, Joseph Siddall, Vic Smith

Supervisor Sara Kalvala

Client Matthew Leeke

Abstract A full multiplayer strategy game — designed to be competitive in the current independent game market — was developed and implemented exclusively using the Haskell language, with the goal of evaluating the efficacy of Haskell for development of this nature. Haskell was shown to be effective in this space, despite the pure functional nature of the language and the comparative lack of library support in the domain of games. Full details of the strengths and challenges of this approach are reported on herein, and a set of best practices for the use of Haskell in game development is suggested.

Keywords Functional Programming, Haskell, Game Development, Graphical User Interface, Networking, Design Patterns, Development as Research, Artificial Intelligence

PUBLISHED BY THE DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF WARWICK

First printing, April 2013

Contents

List of Figures

List of Tables

List of Listings

1 *Introduction: On Games and Functional Programming*

ADVANCES IN TECHNOLOGY WON'T BE AS SIGNIFICANT AS THEY HAVE BEEN IN THE PAST — MOST GAMES WON'T BE MATERIALLY IMPROVED BY SIMULATING EVERY DROP OF WATER IN THE POND YOU ARE WADING THROUGH. MORE RESOURCES CAN BE PROFITABLY SPENT TO MAKE THE CREATION PROCESS EASIER.
— John Carmack

Functional programming (FP) has a long history, with its roots in the λ-calculus of Alonzo Church.[1] One of the first functional programming languages was Lisp, invented by John McCarthy in 1958, which is still used today, over 50 years later.[2] Various languages have refined and extended the functional paradigm over the years — probably the most notable as of now being Haskell, Scala, OCaml, F#, and Erlang.

Despite the amount of time such languages have been available, use in industry has typically been far less than that of imperative languages such as C, C++, and Java.[3] That being said, in recent years there has been increasing use of functional techniques and languages in certain areas. Erlang was designed for the development of highly fault tolerant telecommunication systems.[4] OCaml is used extensively by some organisations in the financial sector to create trading algorithms and other similar applications.[5] Scala is also increasingly popular in various domains, helped in part by its compatibility with the Java Virtual Machine (JVM) and object oriented (OO) design. And finally Haskell has gone from strength to strength in recent years, is used in diverse problem domains, and has a very active development community.[6]

One of the most often cited reasons against the use of functional programming in some domains is that of performance. This is due in part to mutable data structures generally being easier to represent on machine hardware; and it therefore being harder for functional compilers to convert the code into an efficient representation.[7] However, it is not a given that any program would

[1] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33(2):346–366, 1932

[2] E.D. Reilly. *Milestones in computer science and information technology*. Greenwood Publishing Group, 2003, pages 156–157

[3] M. Odersky, L. Spoon, and B. Venners. Programming in scala: a comprehensive step-by-step guide. *Artima Inc.*, 2010, page 11

[4] J. Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007

[5] Y. Minsky. OCaml for the masses. *Communications of the ACM*, 54(11): 53–58, 2011

[6] For example: T. Hawkins. Controlling hybrid vehicles with haskell. presentation. commercial users of functional programming, CUFP (2008). For other instances see the *Commercial Users of Functional Programming* (CUFP) Workshop, `cufp.org`, and the *Industrial Haskell Group* (IHG), `industry.haskell.org`.

[7] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996

run slower if written in a functional language: in some cases lazy-evaluation or compiler optimisations made possible by immutability can mean a program runs faster, plus advanced compiler techniques such as array fusion can lead to programs nearing the efficiency of hand-crafted C.[8] And with modern machines getting ever faster, the domain of problems that require high levels of efficiency is getting smaller.

Performance problems alone cannot account for the fringe position of functional programming. The efficacy of the functional approach has been touted for many years,[9] yet it is still rare for mainstream projects to make any use of functional languages. A possible reason for this is familiarity: because most large scale and enterprise software projects are written in languages such as Java and C++, knowledge about functional techniques and design patterns among professionals is low. And, of course, this means that new projects are unlikely to take up Haskell or other functional languages; and in absence of direct experience outdated beliefs can remain.

Rather than researching and discussing the theoretical advantages of the functional paradigm, this project instead attempts to demonstrate the value of functional programming by utilising it in a problem domain that should pose a significant challenge, and not normally considered a 'good' domain for FP. The chosen domain is the development of a computer game. By developing the software entirely in a functional language and documenting the process thoroughly, it is hoped to provide valuable insight for future projects that could benefit from a functional approach, and also to dispel myths that functional code is slow or hard to maintain.

This choice was made for several reasons. Game programming brings together a diverse range of computing areas — for example human interaction in real time, detailed graphics and animation, artificial intelligence / planning, networking, and various other dynamic elements.[10] For these reasons games are often at the forefront of technological and theoretical advances in computing.[11] However, a game is also a tangible, sizeable piece of software, yet achievable for a four person group in seven months.

As well as demonstrating FP over a wide range of areas, a game also represents a serious business venture.[12] Computer games have been a huge industry for almost as long as personal computers have existed. Demand is high, and a vast amount of games are being continually developed — from triple-A ventures and large companies, to independent ("indie") companies, startups, and hobby groups.

The game developed as part of this project is certainly not the first game ever to be developed in a functional language, nor is it likely to be the last. But by fully documenting and explaining the process, it is hoped to provide a valuable insight into the effectiveness of FP for both game development and other professional computing applications.

[8] Geoffrey Mainland, Roman Leshchinskiy, Simon Peyton Jones, and Simon Marlow. Haskell beats C using generalized stream fusion. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/haskell-beats-C.pdf`

[9] For example see J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[10] See C. Crawford. The art of computer game design. 1984.

[11] L. Haddon. Electronic and computer games: The history of an interactive medium. *Screen*, 29(2):52–73, 1988

[12] Entertainment Software Association. Essential facts about the computer and video game industry. URL `www.theesa.com/facts/pdfs/ESA_EF_2012.pdf`

1.1 Working Title for the Game: Project Serenity

At the outset of the research it was not clear exactly what form the specification of the game would take. For this reason a working title was used to refer to it, and as of now no further title has been substituted. The working title is useful to disambiguate the game itself from the overall project, and is used throughout this document. The title is *Project Serenity*.

1.2 Independence of Game Design from Programming Language

In order for the conclusions of this project to be meaningful, it is important that the game is not designed with the aim of demonstrating FP in mind. If it were, then the apparent performance of the language would be inflated by the nature of the design. It is crucial that the game resembles a standard game project as much as is possible in order for the test to be fair.

In order to enable this as much as is practically possible without having a separate team design the game, several measures were taken. The primary method used was to ban mention of the implementation language during design sessions. The priorities of game design were strictly confined to those independent of the code, and it was agreed that even if features had to be developed separately in a more mainstream language in order to include them, they should stay.

This concern had to be balanced with the requirement to be very aware of the language during actual development in order to document both strengths and weaknesses met in over the course of the project.

1.3 On the Choice of Language for the Project

The following factors were used to judge suitability between languages for use in the project.

Strength of Type System It was desired to use a strong, advanced type system to enable the full benefits of advanced FP techniques.

Purity Many of the advantages of FP come only when true separation between pure functional code, and impure 'effectful' code, is available.[13]

Concision Concise but readable code syntax was preferred to verbose or obscure.[14]

Speed Due to the domain, a reasonable degree of performance was required of the target code.

[13] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989

[14] F.P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley, 1995, also W.M. Taliaferro. Modularity the key to system growth potential. *Software: Practice and Experience*, 1(3):245–257, 1971, and R.W. Wolverton. The cost of developing large-scale software. *Computers, IEEE Transactions on*, 100(6): 615–636, 1974.

Testing Good testing support was desirable both from a software development point of view, and the ability to show test-driven techniques in an FP context.[15]

Community and Library Support Good availability of libraries and a thriving community all vastly aid in development.

Familiarity Languages the team already had some experience of would allow for more effort to be spent on the actual project than learning a completely new language, but also any language that was *too* familiar might make some results less applicable.

Several different functional programming languages were reviewed and discussed in regards to these factors.

[15] See K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10), 1999.

1.3.1 Lisp / Scheme / Clojure

Lisp and its many dialects include some of the earliest examples of functional programming languages. These strong roots in the history of the development of functional programming could make one of the members from this family of languages a good choice for demonstrating the utility of FP.

Clojure, a modern take on the concept of Lisp, has some properties that make it particularly appealing. Firstly, it runs on the JVM which means that programs developed in Clojure are highly portable since Java platform runs on most popular operating systems. However, with this comes the ability to make use of the Java standard library. This may be a great benefit in most software development projects, but it could be construed as cheating when attempting to develop a game using only FP (within reason). A second benefit to Clojure is its focus on concurrency through software transactional memory.[16] This could come in useful for a game development project since it may be desirable to run worker tasks, such as networking, in separate threads from the graphics display to prevent lag.

[16] See `clojure.org/concurrent_programming`

Lisp syntax is unlike that of many modern C-style programming languages, and can be hard to read for a number of reasons, most notably the excessive number of parentheses that can sometimes be required. That said, the project team has some familiarity with using Lisp from a university module on artificial intelligence and having some members (and a supervisor) who are avid Emacs users.

1.3.2 Scala

Scala is relatively recent introduction to the world of programming languages, design was only started in 2001. However, it has quickly grown in popularity and is currently used by the likes of Twitter, Foursquare, The Guardian, and UBS. Just like Clojure, Scala is compiled into Java bytecode that is executed using the JVM. This has the benefits of portability, but the drawback (for this project at least) of the temptation of non-functional Java libraries.

However, Scala is really a multi-paradigm language and it is possible to write Scala code that is essentially identical to its Java counterpart. Although Scala supports a fully functional approach, its multi-paradigm nature is the main reason that Scala was not chosen for this project — it would be hard to objectively test the functional paradigm itself when using a multi-paradigm language. The aim is to demonstrate the usage of the functional approach rather than a mostly functional approach.

It must be noted that this is not intended as a criticism of Scala, just a reason for it being less suitable for this particular project.

1.3.3 Other ML variants (SML, OCaml, F#, etc)

There are several other languages that are broadly similar to Scala which can be considered as the ML family of languages; these include Microsoft's functional language based on the Common Language Runtime (CLR) called F#, and of course ML itself and object oriented extensions of it such as OCAML. To a greater or lesser degree most of the observations about Scala in the context of this project apply to all of these languages.

1.3.4 Erlang

Erlang was a strong contender in the language selection process. It is a mature language, having been initially developed in the 1980s, that has active support. Its huge focus on concurrent programming is what makes it so special and attractive. As mentioned in the earlier section on Clojure, this could be a great boon for the development of a game. Erlang was also intended to be used for rapid development via prototyping.[17] The time constraints on the project made it likely that we would be using an agile development methodology based on the iterative development of prototypes (this turned out to be true, see Section 5.2) and so the support that Erlang provides for this approach could have been useful.

[17] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009, page 14

However, Erlang has a dynamic typing system that is weaker than those provided by other functional languages, such as Haskell. Of the languages considered, it was also one of the least familiar to the project team.

1.3.5 Haskell

Haskell really fitted all of the requirements considered. It is notable for its strong, static type system and its innovative approach to ad-hoc polymorphism through type classes.[18] Haskell is also a purely functional language, but does have support for side-effects using a special construct. Therefore there is a full syntactic and semantic distinction between pure and impure code.

[18] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM

The community of developers and researchers that surrounds Haskell is very large. A dedicated website, called Hackage, is used to host over 5,000 libraries to aid development.[19] This includes

[19] See `hackage.haskell.org/cgi-bin/hackage-scripts/stats`

approximately 79 packages that are focused on testing.

Another benefit to Haskell is that the project team were more familiar with it compared to other functional programming languages, having studied it as part of a second year module on FP. This would reduce the amount of ramp up time required to get started on the project. However, none of the team had developed a large scale project with Haskell before, so it could not be classified as too familiar.

Haskell could be described as the archetype of purely functional languages and it covers all of the factors that were desired in the language for this project. For these reasons Haskell was chosen for the development of Project Serenity. However, it is important to note that this does not imply that Haskell is necessarily 'better' than any of the other languages considered, only that it is probably the most appropriate language for meeting the goals of this project.

1.4 Presentation of Results and the Overall Structure of this Document

At the conception of this project it was known only that some kind of game was to be developed with a functional language. As discussed previously, Haskell was identified as the language to be used for this project. The next phase of the project was to conduct research into existing work, build some pilot games to ensure that the desired outcomes were possible, and to combine this work into a full specification for the future work. This was undertaken mostly during the summer before the first university term (in 2012), and in the first few weeks. The specification was delivered on the 25th October. This work is presented in Chapter 2.

The next, and main stage of the project was the actual development of Project Serenity. This work is examined from two different angles in this report. Firstly, the end software at the time of publication is considered in Chapter 3 from the point of view of a software product, and how the process of the design and development proceeded. Also considered are future directions work into Project Serenity might take. Separately from this is a detailed report on insights into the use of FP for game development: best practices, areas that FP excels in, and areas in which problems were encountered, presented in the form of a 'guide' for functional game programming. This forms the contents of Chapter 4.

Chapter 5 is a full evaluation of the project management techniques used throughout, and the success of the project itself. Business value and other professional issues are also addressed.

Finally, the full results are summarised and their implications are considered in Chapter 6.

2 Research and Development: Games in Space and Time

A GAME IS A SERIES OF INTERESTING CHOICES.
— Sid Meier

Prior to undertaking the main planning and development phases of the project the team embarked on a series of pilot projects, and research into the games industry and functional programming. This initial research and development was done to explore some different concepts for interesting games as well as building some familiarity with game development in Haskell. Two literature reviews were also performed. The first was aimed and looking into existing literature on the suitability of FP for practical software development in the real world with an eye for game development in particular. The second review took a look at the current state of the games industry and the processes involved in designing and creating a game. Finally the project team drew on this research to design a game concept to develop. A full specification of the functional and non-functional requirements was drawn up to help guide the development of the game. This chapter presents all of this research, development, and specification work in detail.

2.1 Pilot Projects: Moon Survival, Conway's Game of Life, and SpaceTime

To make an initial assessment on the feasibility of the project and to gain some insight into both technical design and the style of game to be made, several individual pilot projects were undertaken. Reported on here is *Moon Survival*, which was written in Java; a Haskell implementation of Conway's Game of Life;[20] and *SpaceTime*, a game based on time travel and written in Haskell.

[20] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970

2.1.1 Moon Survival Prototype

Moon Survival is a sandbox game based on the moon. The player controls a character stranded on the moon whose objective is to survive. The moon has no atmosphere, but the player's character requires air to survive. Therefore, he must construct a base and add

oxygen generators inside of it. The basic physics of the game will cause the breathable air that is generated to diffuse throughout the base allowing the player to survive. The style of game is a creative open world in which to experiment, similar to Minecraft.[21] The player can build whatever and wherever they want to, and can experiment with oxygen diffusion physics.

There are two items available for the player to build:

air generator creates breathable air and releases it into its immediate surroundings.

wall solid block that halts the diffusion of air as well as preventing the player from walking through that tile.

Using only these two components the player must build their base in which their character is able to survive.

The physics of diffusion is highly simplified. The game is able to mimic the movement of air through the world without having to perform too many complex calculations. Figure 2.1 shows how air diffuses throughout a three by three area in one iteration of the game loop. The top diagram in the figure shows the state of the game prior to starting, and the bottom one shows how diffusion has occurred after one step. As shown, the air diffuses from an area of high concentration to areas of lower concentration.

The diffusion equation uses the following terms:

c function from block to air concentration at that block

t diffusion rate, constant between 0 and 1 representing transfer rate, 0 is no transfer, 1 is transfer until equilibrium between blocks

For two adjacent blocks, b_1 and b_2, the volume of air that diffuses from b_1 to b_2 is:

$$transfer = \frac{c(b_1) - c(b_2)}{2} * t$$

Although this equation for diffusion allows for incredibly quick calculation, it has a problem due to its simplicity. The amount of air transferred from the centre block in Figure 2.1 reduces for each consecutive neighbour, i.e. the first neighbour gets more air than the second neighbour, and the second neighbour gets more air than the third neighbour, and so on, instead of all of the neighbours receiving the same degree of air transfer.

The game controls are extremely simple and will be familiar to most computer gamers. To move the character the player uses 'w', 'a', 's', and 'd' to move North, West, South, and East respectively. Figure 2.2 shows a screenshot of these movement controls within the game. If there is an obstacle in the way, such as a wall block, the player will be unable to move in that direction. A slight limitation of this movement system is that holding down a key does not cause

[21] Minecraft is a game about breaking and placing blocks. At first, people built structures to protect against nocturnal monsters, but as the game grew players worked together to create wonderful, imaginative things. See `minecraft.net`

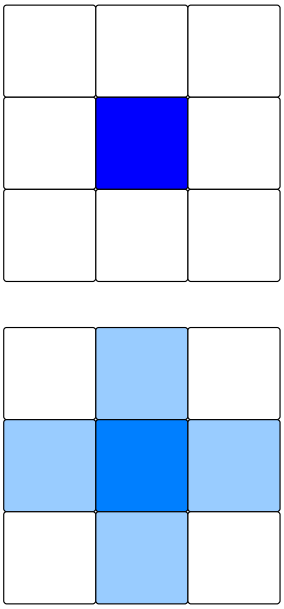


Figure 2.1: Diffusion of air through a 3x3 block layout. The darker the blue, the greater the concentration of air in the block. Air diffuses from the area of high concentration in the centre block to its non-diagonal neighbours.

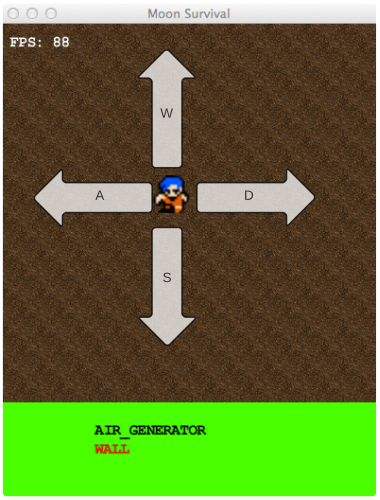


Figure 2.2: Overview of Moon Survival controls.

the player to continue to move in that direction. Instead the key must be repeatedly tapped to have this effect.

The player is equipped with one item (either 'wall' or 'air generator') at a time, and can switch between the items with the tab key. The currently selected item is displayed in the command panel at the bottom of the screen. Pressing the space bar places the currently selected item directly in front of the player. If the block directly in front of the player is already filled with the currently selected item then it will be removed. For example, if the player presses the space bar, and the wall item is selected, then the wall block in front of the player is removed.

The GUI is described in the annotated screenshot shown in Figure 2.3. The window is broken down into two panes: the game panel and the command panel. The game panel displays the player and their immediate surroundings. The command panel displays the items that the player has access to. The currently selected item is highlighted in red.

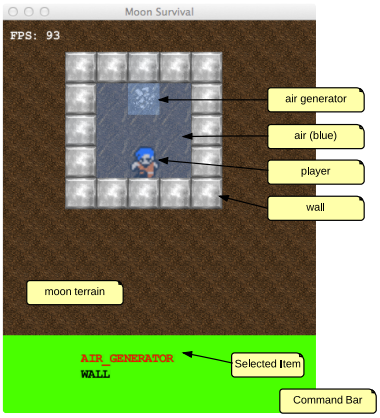


Figure 2.3: Moon Survival GUI.

Although this is a relatively simplistic game it helped get in to the mindset of game design and development. It showed that the open world sandbox style game is entertaining even when it is so basic. Playing the game highlighted a key aspect, danger. The player must maintain a supply of oxygen in order to survive as they expand their base. The greater sense of danger and urgency to constructing and maintaining your base made the game more challenging and fun. The game had a paucity of available items that limited gameplay somewhat. Despite this limitation, this prototype gave insight into how to make entertaining games and how missions are not a necessary element of fun games.

2.1.2 *Conway's Game of Life*

Invented by British mathematician John Horton Conway in 1970, Conway's Game of Life (or just "Life"), has become iconic both within academia and elsewhere, and was thought to be a fitting demonstration of real-time graphics and game logic using Haskell.

Life is a basic example of a cellular automaton. The game field is a rectangular grid of cells that can be either dead or alive. Each time step the state of the cells is updated. The standard rules are a cell dies if it has fewer than 2 neighbours or more than 3, a dead cell comes to life if it has exactly three neighbours, otherwise the cells stay in their previous state.[22]


Figure 2.4: Glider gun in Conway's Game of Life.

[22] For more details on the Game of Life, see Gardner 1970, and `pentadecathlon.com/lifeNews/index.php`.

The implementation starts with a paused grid of dead cells. The user can toggle the state of a cell by clicking it, and stop and start the simulation by pressing the space bar. Array data types from DATA.ARRAY are used to store the state of the game, and the topology is made to "wrap around" so events on the leftmost edge affect the right, for example.

The logic and rules of the game of life were coded using pure code, and then the library *Gloss* was used to draw the game board,

update the board by calling the pure logical functions, and process mouse and keyboard events. The complete program is only 90 lines of Haskell.

It was this experiment, and the ease of the implementation that led to the decision to use Gloss for Project Serenity, as its simplicity would allow rapid development of an early version.

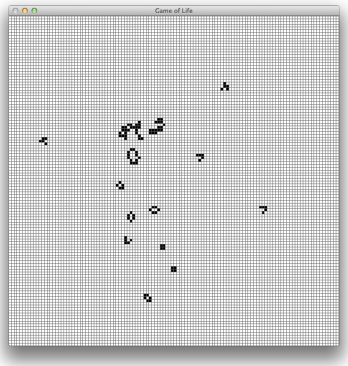


Figure 2.5: Full screen from Haskell implementation of life, showing gridlines.

2.1.3 *SpaceTime*

SpaceTime (or Serenity Mark I) was the title given to the most advanced Haskell prototype developed. The concept, inspired by the RTS game *Achron*,[23] explores the concept of a game where past actions can be changed, affecting the present and future. This concept formed the early ideas that grew into the full Project Serenity design, although the time travel aspect was eventually dropped.

SpaceTime prototyped several concepts that were later refined in Project Serenity, including a basic framework for AI and a system for button widgets for giving commands. Player actions were modelled as happening at the specific moment selected in the timeline (see screenshot in Figure 2.6), which could be dragged to a different position or played at a constant rate and the current version of events would animate. If a new command is given at a specific point then history is recalculated.

[23] http://www.achrongame.com/site/

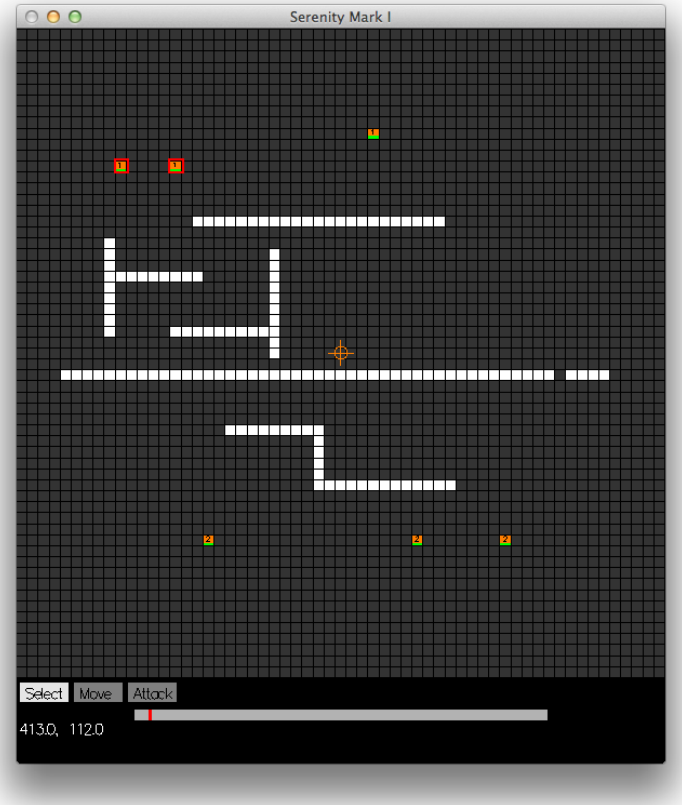


Figure 2.6: A screen capture from *SpaceTime*.

A problem encountered during the development of SpaceTime

was the effect of lazy evaluation on frame rate. The lazy semantics of Haskell are highly beneficial in many ways, but can in some cases lead to problems. Evaluation can be forced, but the best way to do so is not always obvious. The prelude provides a function `seq`, which fully evaluates its first argument before returning its second. However, there are many pitfalls for the unwary. For example, the expression `seq a a` is actually equivalent to just `a`, and so evaluation is not forced. Also, the forced evaluation is only to *weak head normal form* (WHNF). This is a technical term and the details are out of the scope of this discussion,[24] but the main repercussion of this is that if `seq` is used to force a list, only the first element will actually be evaluated. Further tricks are required to force each element of the list to WHNF (this is referred to as forcing the *spine* of the list). Forcing evaluation of the length of the list will force the spine for example. There is a package called *deepseq* to help address these issues, but they can still be hard to identify and fix.

[24] For more details on WHNF see `en.wikibooks.org/wiki/Haskell/Graph_reduction`.

The SpaceTime prototype was invaluable in many ways to the later development of the full Project Serenity. Most notably it taught that good structuring of Haskell modules is not as obvious as first impressions might suggest, and that some thought at the beginning of a project can go a long way. Given the problems with laziness, it was suspected that similar issues might crop up in the full game, but this didn't actually happen. SpaceTime also highlighted the complexities of user interface programming with Gloss, that led eventually to the development of the Sheen framework (see Section 4.5).

2.2 *Literature Review: Functional Programming for Games*

This section will discuss the suitability of the functional approach in the real world, and the use of Haskell in particular. It will look at past projects and research into the use of functional programming languages in industry in an attempt to discover how functional programming helped or hindered development.

John Hughes' paper "Why functional programming matters" aims to demonstrate how "vitally important" functional programming is to the real world by exploring and demonstrating its advantages.[25] Hughes argues that modularity is the key to designing and implementing successful programs for three main reasons. Firstly, small modules are much easier to code quickly because the requirements for a small component are much easier to reason about, design, and implement. Second, the more generic modules that are constructed can be reused. This leads to faster development during subsequent projects. Thirdly, the independence of modules allows them to be developed and tested separately, helping to parallelise the work that needs to be done and reducing the amount of time required for debugging. These advantages of modular design combine to bring great improvements to productivity. These benefits of modularisation are also espoused by Parnas who agrees that modular programming shortens development time, improves comprehensibility of the resulting programs, and increases flexibility — it is possible to make large changes to one module without affecting any of the others.[26]

[25] J. Hughes. Why functional programming matters. *The Computer Journal*, 32 (2):98–107, 1989

[26] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972

However, the ability of the programmer to modularise their code is reliant on the ways in which they can glue the solutions of subproblems together. This glue must often be provided by the programming language itself. Hughes argues that functional programming provides two very important kinds of glue: higher order functions and lazy evaluation. These two aspects of functional programming are very powerful and allow greatly improved modularisation.

General higher order functions, such as `map` and `reduce`[27], can be used as glue for simpler, specialised functions to make more complex ones. Higher order functions are great examples of code reuse as they can be used to create many other functions with minimal effort. Hughes gives examples of operations over lists and trees, such as summing up the elements of a list, whose implementation is greatly simplified by the use of higher order functions. Lazy evaluation, on the other hand, allows whole programs to be glued together. When composing two programs it might be infeasible to store the entirety of the output of the first function in memory to pass on to the second. Lazy evaluation is a solution to this problem. The output function is only started when the input to the second function is required, and only runs for long enough to provide the required amount of input. If the consuming function terminates

[27] The `reduce` function is called `foldl` in Haskell

early then the producing function can also quit. This even allows the producer to create an infinite amount of output. This allows modularisation by constructing a generator that outputs a large set of potential answers and a separate selector that chooses the correct one.

Hughes finishes with an example from the field of artificial intelligence, a field of computer science that is very relevant to game development. He shows how the alpha-beta pruning algorithm can be constructed relatively simply using modularisation through higher order functions and lazy evaluation. The algorithm works by generating the entire set of possible game states that are reachable from the current position. This list can then be lazily evaluated to find the optimal move, but without actually constructing the entire, possibly infinite, game tree. Higher order functions are used throughout to build up complex functions from simpler ones. Hughes also shows that due to the modularisation of the example it is much easier to understand and make modifications to the program.

This paper is a great example of the power that is available from the functional approach. Giving real examples Hughes is able to make a strong case for the effectiveness of modularisation through laziness and higher order functions. The demonstration of a highly modular version of the alpha-beta pruning algorithm, in particular, is of great interest due to its applicability to game development. The conclusion that the functional approach leads to more general, reusable modules is supported by John Backus' ACM Turing award lecture from 1977. Backus gives the example of a program to calculate the inner product and finds that "the functional version is nonrepetitive, ... is more hierarchically constructed, is completely general, and creates 'housekeeping' operations by composing high-level housekeeping operators."[28]

In his 1987 paper "No Silver Bullet", Brooks identified four difficulties that are inherent in the nature of software development: complexity, conformity, changeability, and invisibility.[29] Brooks believes that these four essential difficulties make it very unlikely that there will be a "single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity." However, Moseley and Marks argue that complexity is the only significant problem; that "complexity is *the* root cause of the vast majority of problems with software today."[30] Other problems can either be classified as complexity, or derive from unmanageable complexity. They argue that simplicity is vital to successful software development, and that functional programming can help to deliver this simplicity.

Moseley and Marks indentify several causes of complexity in real software systems. The first cause is mutable state. Brooks also mentions the problem of state, saying that from "the difficulty of enumerating, much less understanding, all the possible states of the

[28] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978

[29] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987

[30] B. Moseley and P. Marks. Out of the tar pit. In *Software Practice Advancement (SPA)*, 2006

program, ... comes the unreliability".[31] State hinders understanding of software through testing and reasoning about the code. This is because testing a program in one state does not guarantee anything about how the program will behave when in a different state. The vast number of different possible states also makes it infeasible to understand them all. The functional solution to the complexity of state is to discard state and side effects. Programming with a pure functional language, such as Haskell, creates referentially transparent programs. Referential transparency means that given the same set of arguments a function will *always* return the same result. Removing state and side effects eases understanding of programs because they are easier to reason about and test: "avoiding side effects has serendipitous effects on testing."[32]

[31] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987

[32] N. Smallbone. Property-based testing for functional programs. Licentiate thesis, Chalmers University of Technology, 2011

However, Moseley and Marks suggest that "the main weakness of functional programming is the flip side of its main strength — namely that problems arise when (as is often the case) the system to be built must maintain state of some kind." Games are an example of programs that must keep some kind of state, such as a score or the positions of entities in the world. Is it possible to simulate the necessary state in a functional language that has removed mutable state? One possibility would be to add a new parameter and change the return type of functions to allow them to accept and output state. In this way the state can be threaded through the entire program. Moseley and Marks point out that this would recreate a pool of global variables and, although referential transparency is maintained, the ease of understanding is lost. The all important concept of modularity is raised again by Wadler who notes that "it is with regard to modularity that explicit data flow becomes both a blessing and a curse."[33] He describes explicit data flow as "the ultimate in modularity" since all data in and out is seen clearly and is accessible. On the other hand, "the essence of an algorithm can become buried under the plumbing required to carry data from its point of creation to its point of use." The other approach, applicable to Haskell, is to use monads. Wadler explains that monads can be used to "mimic the effect of impure features such as exceptions, state, and continuations."[34] The use of monads in functional programming allows a developer to work with state without drowning under the huge amount of explicit data flow required in the former approach. Although Moseley and Marks are still concerned that "despite their huge strengths monads have as yet been insufficient to give rise to widespread adoption of functional techniques."

[33] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995

[34] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992

The second cause of complexity identified by Moseley and Marks is complexity from control. Control is the order in which things happen within a program. In most programming languages the developer is concerned with control because often the ordering is controlled by the order in which code appears in a program and because this order is further modified by branching and looping instructions. The problem with control is that it hinders informal reasoning about a program. A reviewer must assume that the or-

dering of statements is significant until proven otherwise. If a mistake is made in this process then subtle bugs can be introduced into a program. Functional programming helps slightly with this problem since the approach encourages more abstract control with functions such as `map` instead of explicit loops. Also, due to the referentially transparent nature of functional programming, the order of execution of function calls is irrelevant.[35,36]

The final major cause of complexity is code volume. Large, bloated programs require much more effort to fully understand. Brooks believed that the complexity of a software project increases nonlinearly with its size.[37] For this reason it is "vital to reduce the amount of code to an absolute minimum".[38] The functional approach to programming has been noted to produce much more concise programs. For example, Hughes states that "functional programs are an order of magnitude shorter" than their conventional counterparts.[39] Moseley and Marks also argue that by reducing the complexity caused by state and control it is much less likely that complexity will grow with code volume in a nonlinear fashion, citing Dijkstra:[40]

> It has been suggested that there is some kind of law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. And this is because it need not be true... As a result I tend to the assumption — up till now not disproved by experience — that by suitable application of our powers of abstraction, the intellectual effort needed to conceive or to understand a program need not grow more than proportional to program length.

It has been shown in the explorations of the previous two causes of complexity that functional programming can help to reduce the complexity of state and control. Therefore, the issue of code volume may be less of a cause for complexity than in other programming paradigms.

Common misonceptions surround the use of functional languages for practical software projects. Many seem to believe that functional programming restricts a developer; that it is too hard to build graphical programs, work with input and output, or perform other stateful computation, such as networking. The author of the darcs version control system laments that a common reaction from people hearing about darcs is to say that "it is a shame that it is written in Haskell".[41] They believe that, because it is written in Haskell, darcs will be inefficient, hogging memory and running slowly. Roundy then goes on to discuss the problems and successes he encountered whilst developing darcs in Haskell to show how Haskell can be used to build useful, real world programs.

Roundy talks about testing with Haskell and the power of the QuickCheck library. QuickCheck is a property based testing library that requires the developer to create specifications for their code. QuickCheck will then automatically generate test cases for these expected properties.[42] Testing is extremely important aspect of

[35] J. Hughes. Why functional programming matters. *The Computer Journal*, 32 (2):98–107, 1989

[36] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995

[37] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987

[38] B. Moseley and P. Marks. Out of the tar pit. In *Software Practice Advancement (SPA)*, 2006

[39] J. Hughes. Why functional programming matters. *The Computer Journal*, 32 (2):98–107, 1989

[40] E.W. Dijkstra. The humble programmer. *Communications of the ACM*, 15 (10):859–866, 1972

[41] D. Roundy. Darcs: distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4. ACM, 2005

[42] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279, 2000

good software development. Therefore, any programming language that is going to be used for real software projects requires good support for testing. The availability of testing libraries for Haskell that have been used successfully in existing projects is a good sign for the suitability of Haskell for developing real world applications. The same day that Roundy started making use of QuickCheck he was able to discover and fix a bug. However, he found that it was sometimes hard to develop custom data generators which worked correctly. Often it was found that test cases failed because of invalid patches being generated instead of bugs in the darcs code itself.

Roundy goes on to talk about how essential the foreign function interface (FFI) was for the development of darcs. The FFI is used to links Haskell programs to other programs written in a different language, such as C. In darcs, for example, the FFI was used to interface with `libcurl` for HTTP support. The necessity for the FFI suggests that functional programming may not be suitable for all problems and that complex, functional projects might have to 'resort' to making use of non-functional libraries. On the other hand, this paper was written in 2005, so the number of Haskell libraries for common problems will have increased. So, resorting to non-functional libraries is less likely to be required for common problems.

Roundy also talks about the difficulty of optimisation in Haskell. He states that increasing laziness at a high level often helps to improve memory usage, whilst increasing strictness at lower levels usually makes functions faster. However, the difficulty is in determining which approach to take to optimise a given function and it is almost never obvious how a change will affect the laziness of a function. Efficiency is an important requirement for real time games, so difficulties with optimisation may have a negative impact on the quality of a game. On the other hand, Roundy praises the utility of the profiling tools that are available for Haskell. Using these tools it is much easier to pinpoint the areas of code to focus optimisation efforts on.

Deciding how to optimise a function in Haskell is not the only difficultly. It may require dropping into another language. Roundy states that for the lowest level functions "optimisation has consisted of rewriting a key function in C or calling a C library function". Again, this is not a good sign for the performance of functional programming. However, in the eight years since this paper was written, a large number of performance improvements have been made to Haskell compilers. This means that functional programs written today are more likely to perform well. It may also be the case that the optimisations that were made to darcs could have been made in a different manner whilst still making use of Haskell.

Roundy concludes that darcs has been a highly successful project written in Haskell. His comments support the ideas of modularity proposed by Hughes stating that "Haskell itself allows the creation of clean internal interfaces in the code". These clearly separate

modules allowing contributors to focus on certain areas instead of having to learn the entire code base and all of its iteractions. And, although there have been efficiency problems in the past, they have mostly been fixed.

2.3 *Literature Review: Designing a Game for 2013*

The most popular games over recent years have been of the first person shooter, hack and slash, FIFA and Fitness/Dance games genres. Unsurprisingly these genres are among those most frequently published by the publishers shown in Tables 2.1 and 2.2, implying that they are significantly more profitable than other genres, which indicates that such genres are also the most popular. This makes designing a new game challenging, since there is an immediate bias towards a genre which shows promise for sustained development and sequel games, and has the largest player base.

Title	*Publisher*
Call of Duty: Modern Warfare 3	Infinity Ward
FIFA 12	Electronic Arts
Battlefield 3	Electronic Arts, Sega
Zumba Fitness	Majesco Entertainment
The Elder Scrolls V: Skyrim	Bethesda Softwork
Just Dance 3	Ubisoft
Assassin's Creed: Revelations	Ubisoft
LA Noire	Rockstar Games
Saints Row: The Third	THQ, CyberFront, MicroByte
Batman: Arkham City	Warner Bros. Interactive Entertainment

Table 2.1: Best selling games of 2011.

Title	*Publisher*
Call of Duty: Black Ops II	Activision Blizzard
FIFA 13	Electronic Arts
Assassin's Creed III	Ubisoft
Halo 4	Microsoft
Hitman Absolution	Square Enix
Just Dance 4	Ubisoft
Far Cry 3	Ubisoft
FIFA 12	Electronic Arts
The Elder Scrolls V: Skyrim	Bethesda Softworks
Borderlands 2	2k Games

Table 2.2: Best selling games of 2012.

The highest priority when designing a game is to make it fun for the target audience, however, the reception of a game is heavily dependant on the ever-changing gaming culture, so it's extremely difficult to formalise the art of game development. Consequently academic references are extremely rare, so the project's literature review aims to identify current trends by investigating a range of publications, from leading game designers such as *Valve corporation EA Games*, *Ubisoft*, as well as the leading game critics *GameSpot*, *IGN*, and independent game reviewers *Tom Francis*, *Anton Temba*, *Wolfgang Kramer* and *Peter Collier*.[43]

[43] B. Chavanu. The top 5 youtube video game reviewers. 2011. URL `makeuseof.com/tag/top-youtube-video-game-reviewers/`; and Top 15 most popular video game websites, 2013. URL `ebizmba.com/articles/video-game-websites`

Originality Francis and Kramer both assert that a successful game should have some original content such as a unique storyline, an unusual physics engine, new weapon concepts, or anything which pulls away from conventional games and adds a new dimension to gaming. If a game has limited originality then it risks becoming dull and uninteresting since it will appear to be a combination of existing game features instead of offering a new concept to the gaming community.[44] [45]

Adaptive Pacing and Replayability Temba highlights the need for the gaming experience to be fuelled by lots of active events, where the player can make choices to influence the outcome. Kramer echoes this view by expressing the need for a unique gaming experience developed by carefully architecting the game. [46] [47] Valve addresses the issue of replayability through "algorithmically adjusting the game pacing on the fly to maximise "drama" (player excitement/intensity)".[48] and spends a lot of design time investigating a procedurally populated environment. The purpose of these adjustments is to fit the gameplay to the individual player so the game is always challenging and exciting, which can be particularly difficult for cooperative multiplayer games.

Many approachable solutions for the project to create a unique gaming experience include procedurally generated environment, intelligently offering *supplies* or bonus upgrades (temporary invulnerability, invisibility etc) to losing players, to ensure the game is balanced and enjoyable for players of all abilities.

Surprises Game journal *Gameranx* highlights the need for surprises in both the plot and hidden features, which "extend the length and replay value of a game, and more often than not give gamers something to talk about".[49] A typical feature of a well structured game is to have different possible outcomes for every possible decision the player is allowed to make. This is a rather typical pattern found in games such as Unreal Tournament, where a player's weapon arsenal might allow him to fend off foes competing for an objective.

Equal opportunity Francis and Wolfgang both consider equal opportunity is vital for multiplayer games, and so it's important that no player has a distinct advantage over any other human player; such imbalances could be caused by advantageous starting positions, unequal starting budgets or anything which gives one player a better chance of winning. Clearly, different games have different requirements for fulfilling equal opportunity, but one of the most important requirements of a multiplayer game is to give every player an equal chance to win the game from the starting point. [50] [51]

No early elimination No player should lose hope of winning early in the game. In the event of poor playing in the start of the game, a player should be able to redeem themselves to regain a chance of winning the game, in this context it's acceptable for the game

[44] T. Francis. What makes games good. 2011. URL pentadact.com/2011-05-27-what-makes-games-good/

[45] W. Kramer. What makes a game good? *The Games Journal*, 2000. URL thegamesjournal.com/articles/WhatMakesaGame.shtml

[46] A. Temba. The infinite game. 2013. URL gamedev.net/page/resources/_/creative/game-design/the-infinite-game-r3045

[47] W. Kramer. What makes a game good? *The Games Journal*, 2000. URL thegamesjournal.com/articles/WhatMakesaGame.shtml

[48] M. Booth. The ai systems of left 4 dead. Valve, 2009. URL valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf

[49] Shocking surprises and secrets in video games, 2012. URL gameranx.com/features/id/5634/article/top-20-shocking-surprises-and-secrets-in-video-g

[50] T. Francis. What makes games good. 2011. URL pentadact.com/2011-05-27-what-makes-games-good/

[51] W. Kramer. What makes a game good? *The Games Journal*, 2000. URL thegamesjournal.com/articles/WhatMakesaGame.shtml

to *help* a player (e.g. spawning health boosts near the player, or having the AI teams primarily target other players. Valve introduced AI assisted power-ups in *Left 4 Dead 2*, where the game designer specifies available supplies and the game's AI decides when to spawn them based on the player's needs, which reduces the chance of the player being unable to progress due to earlier difficulties. [52]

Low waiting times Francis and Wolfgang agree that a player's interest lies in being involved with a game and don't want to endure period of low activity. This was a common problem in early real-time-strategy games such as Age of Empires, where a player would spend significant time in the "setting up" phases. It's important to balance the focus of the game so there's focus on areas of interest without risking the compromising the story or atmosphere. [53] [54]

Environmental Art and Sound Valve argues that much of the game's atmosphere stems from the player's environment, so atmospheric artwork and sound should be considered an important factor in making the players feel immersed in the game.[55]

This project intends to challenge the conventional genre space by aiming to produce a real time strategy game. The RTS genre has given rise to some extremely popular productions in the early day of gaming corporations such as Total Annihilation, Supreme Commander and the Command and Conquer franchise. Early games lacked sophisticated hardware and game engines, so there was a large focus on developing the gameplay experience, which is likely why these games are so iconic. Since time frame and budget of the project is limited by time and resources, it seems that a simple RTS game would be suitable choice of genre for the game provided it does not interfere with the criteria previously outlined in this section.

[52] M. Booth. The ai systems of left 4 dead. Valve, 2009. URL `valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf`, slide 75

[53] T. Francis. What makes games good. 2011. URL `pentadact.com/2011-05-27-what-makes-games-good/`

[54] W. Kramer. What makes a game good? *The Games Journal*, 2000. URL `thegamesjournal.com/articles/WhatMakesaGame.shtml`

[55] B. Jacobson. Sound practices of game business and design. Valve Corporation, 2007. URL `valvesoftware.com/publications/2005/AGDC2005_SoundPractices.pdf`, Slide 8

2.4 Formal Project Specification

After completing the research and pilot projects the first major task was to deliver a project specification. Part of the specification was to detail the list of requirements and functionalities that are expected in the final deliverable. Determining the objectives and requirements is an important step to make sure that the project does not fall into a 'ready, fire, aim' situation in which a project is started with no clear goals.[56] This section gives the requirements for Project Serenity that were envisioned at the beginning of term one.

[56] H. Maylor. *Project Management*. Pearson Education Limited, 4th edition, 2010, pages 14–15

2.4.1 Functional Requirements

The game will be a real-time strategy game set in space. Opposing fleets of spacecraft will battle each other. A player's goal is to ensure the survival of their fleet and the destruction of the enemy.

FR1 Multiplayer As a multiplayer game, two players must be able to participate in a battle against each other. These players will be on different computers on the same network. This is a more attainable requirement than supporting gameplay across the internet, as latency and packet loss issues will be less intrusive.

FR2 Fleets Each player will control a fleet of ships. Ships have two health stats: hull and shields. Once hull health has been reduced to zero then the ship is destroyed. However, functioning shields prevent hull damage. It is intended that shields be relatively fast to recharge but hull damage is slow and expensive to repair.

FR3 Ship Customisation The ships that make up a player's fleet will each have a number of pluggable slots. Prior to a game each player will be able to customise the ships in their fleet by filling these slots with different pieces of equipment. There will be two types of slot: system slots and weapons slots. System slots allow for extra internal systems to be added to the ship, for example extra shields or long range scanners. There are three types of weapon slots found in different areas of the ship, they are named front, side, and turret. Weapons slots allow the player to choose which types of weapons their ships will use; however, a fleet budget will prevent a user from choosing the best weapons on every ship. See figure 2.7.

FR4 Resource System Three resources exist within the game: fuel, metal, and anti-matter. These resources are only used by the ships. Fuel maintains a ship's shields, without a shield all damage will be done to the ship's hull. Metal is used to slowly repair a ship's hull after it has been damaged. Anti-matter is a rare resource that is required by the more powerful weapons available.

Planets generate a constant supply of resources, so players can gain extra resources via planetary capture. The resources gen-

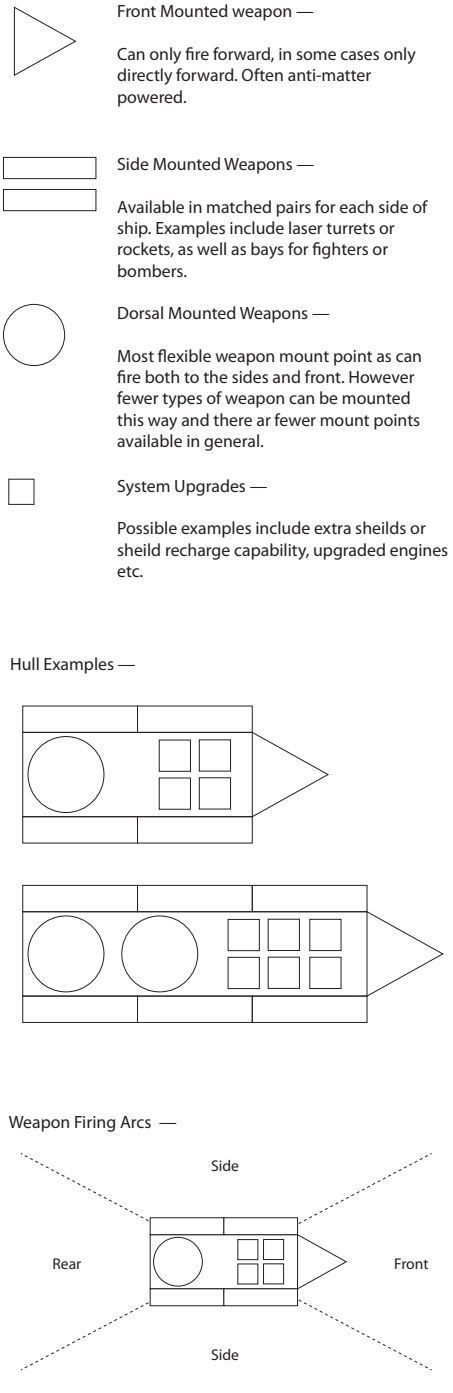


Figure 2.7: Diagrams showing the initial concept for ship customisation and weapon configurations.

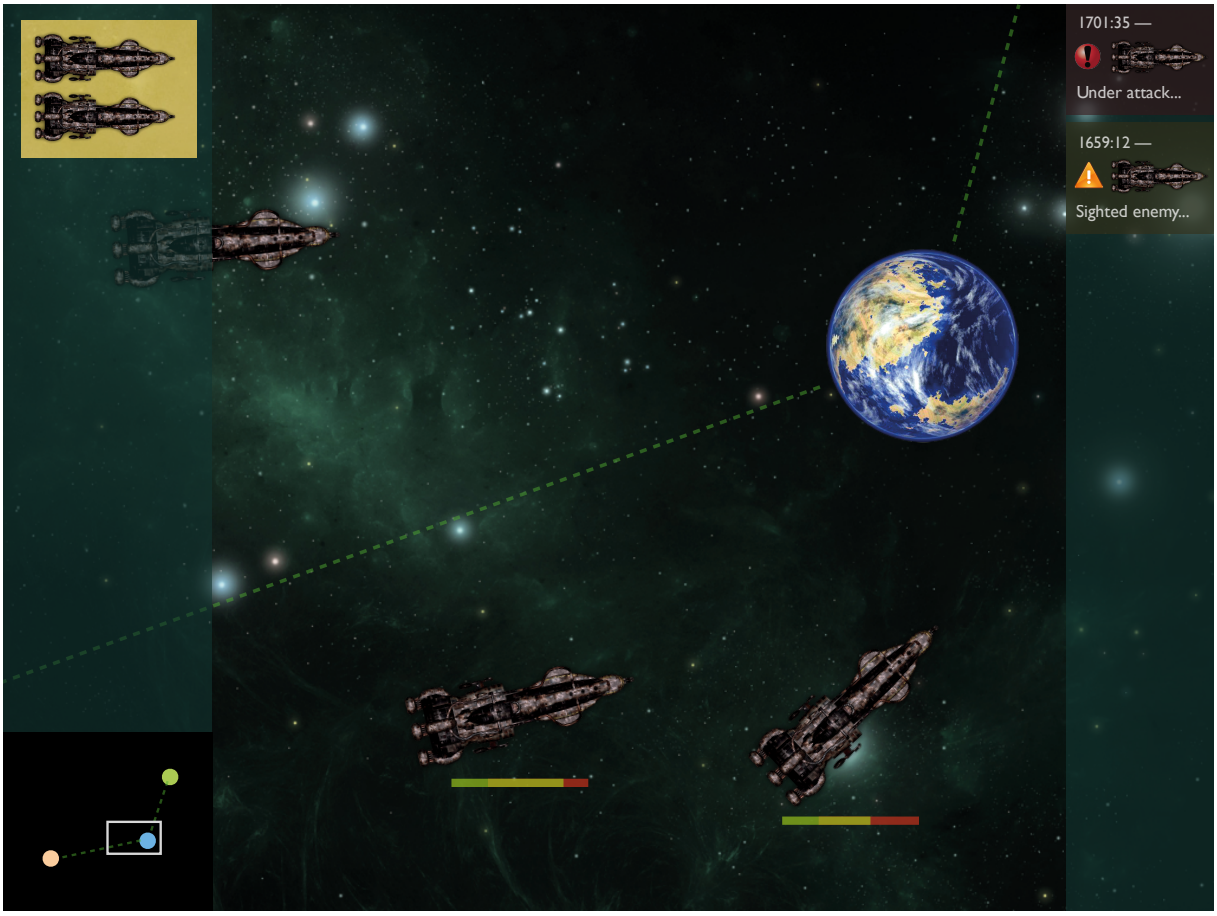


Figure 2.8: Initial mockup of a screen during gameplay, showing a planet, spacelanes, selected ships, minimap, and reports from other ships.

erated by the planets owned by a player feed into that player's global stockpile. Individual ships then draw resources from this stockpile.

FR5 Planetary Capture Planet ownership is the only method of generating resources. Planets can either be unoccupied or they can be owned by a single player. To capture a planet, a player must have their ships in range of the planet for a certain period of time. If the planet belongs to the enemy then it will take twice as long for it to be captured than an unoccupied planet.

FR6 Fog of War Players must not be able to see the state of the entire map unless they control it all. There will be three levels to the fog of war: unknown, visited, and visible. Any locations on the map that a player's ships have not visited will be 'unknown' and the player will not be able to see anything that is at that location. Any locations that have been explored at least once will be 'visited'; the player will be able to see the general layout of the area, but not any details such as enemy ships. Finally, any locations covered by planets or ships owned by the player will be 'visible' and all aspects of the map in the area will be revealed.

FR7 AI A sophisticated artificial intelligence system will be an important component in the game. Each player's fleet will be

controlled through an AI system. It will use a planning algorithm that takes a high level objective given by the player and generates a series of steps to achieve that objective. The AI must control the individual ships that make up the fleet; it is up to the human player to decide on the overall strategy of the fleet.

A successful AI system must receive orders and quickly convert them into a sensible plan which it performs and reviews autonomously. If an impossible goal is set, or an existing goal is invalidated by changes to the world, then the AI must detect this and act accordingly.

FR8 Campaign A campaign mode will be available which consists of a fixed number of battles between the two players. The final battle will be the 'showdown' that determines the overall winner. The victor of each of the earlier battles will be granted bonuses toward the final battle, giving them an advantage against their opponent.

Between every battle each player will have an opportunity to perform minor customisations on the ships in their fleet.

FR9 Operating System Requirements The game must be playable on recent versions of Mac OS X and Linux. The nature of the libraries that are likely to be used for development make it possible that also supporting Windows will be relatively easy, but this is not guaranteed and no commitment on it is being made at this stage.

2.4.2 *Non-Functional Requirements*

NFR1 Fun One of the most important requirements is that the game should be fun to play. Players should enjoy the game and want to play it multiple times.

NFR2 Short game sessions An individual battle should not last too long. If a game is likely to take a number of hours it is a barrier to entry for players as they must schedule large amounts of their time if they are to play at all. The aim should be for an individual battle to last between 20 and 35 minutes. Tournaments or campaigns are an optional feature that could provide inbuilt support for longer playing sessions.

NFR3 Approachable An end user should be able to play a networked game with minimal configuration (no more configuration required than a typical installation and networking configuration).

NFR4 Reliable Both the client and server should be stable programs that are not prone to crashing. If either were to crash regularly then it would ruin the experience and cause people to stop playing the game.

The networking component should also be reliable. Minor network disruption should not cause a huge loss in communication between the clients and server.

NFR5 Secure Although the game server will initially be intended for LAN usage it is important that it should not cause a computer running it on the Internet to be exploitable. It should not be vulnerable to attacks such as denial of service, which would stop the machine from performing any other tasks whilst under attack, or remote code execution, which could allow an attacker to take control of the target machine.

Furthermore, the game system should not be vulnerable to cheating by modification of the client code, packet injection attacks, or other similar methods of gaining an unfair advantage.

3 *Results: From Specification to End Product*

FINISHING RACES IS IMPORTANT, BUT RACING IS MORE IMPORTANT.
— Dale Earnhardt

A large part of this project was purely a software development project to deliver a fun real time strategy game set in space. A lot of the design and development of this game is irrelevant to the research directed portion of the project which aimed to investigate the suitability of Haskell and FP for game development. Instead it rather generic work that is applicable to game development in any programming language. For example, designing the gameplay, implementing a reliable networking architecture, and creating graphical assets to display. This chapter describes the more general part of the project and presents the finished product itself.

The process of designing a game involves taking the basic concept of the game and the system requirements that were created for the specification, and producing a plan for implementing these features. This design is then implemented step by step to iteratively create a working game that fits the original requirements. The major aspects of this design and implementation process will be covered including the server-client architecture, the models and mechanisms that underpin the game concept, the artificial intelligence and path finding systems, and the creation of the assets and graphics that make up the GUI.

The project proceeded in two major phases: the alpha phase that took place during term one, and the beta phase that started in the Christmas holidays and continued through to the end of second term and the Easter holidays. The aim of the alpha stage was to lay the foundations of the game and create a prototype release that comprised of the minimal set of features to be called a 'working game'. This foundation would be the infrastructure on which the actual gameplay features could be built. This included a networking architecture and library, basic model of the game components, the basic game loops that would update these models, and finally some assets to be used to represent game entities and a rendering process to display them. The alpha stage produced what was desired, a networked program with a usable GUI. Two

or more players could launch the program such that they would communicate with each other. Each player was in charge of their own ship and could control it by clicking on a location for the ship to fly to.

The beta stage started by fixing some bugs and introducing some enhancements to this groundwork. However, the main aim of this stage was to flesh out the game mechanics on top of the foundations. This work introduced the major features of the game including an artificial intelligence framework to handle invidual ship behaviour, a proper GUI menu for launching games, improved graphics and GUI elements to match the mockup drawn up for the specification, and the gameplay mechanisms such as attacking and planetary capture. All of this development culminated in the release of a fully working game in which players have access to an elegant GUI which enables them to easily play a real time strategy game set in space. The following sections will explore the work from both of these stages in greater detail.

3.1 *Client-Server Architecture*

One of the first technical design decisions to be discussed was that of the client-server architecture and how the different clients would keep synchronised with each other. This was a very important decision to make as a poor networking set up can severely impact the playability of the game. For example, Patrick Wyatt talks about how the first ever multiplayer game of *Warcraft* experienced a serious game-state synchronisation problem; both computers involved ended up "showing entirely different battles that, while they started identically, diverged into two entirely different universes".[57] Three separate strategies for solving this problem were discussed: lockstep, simple server-client, and server-client with simulation.

[57] P. Wyatt. The making of warcraft part 3, 2012. URL `codeofhonor.com/blog/the-making-of-warcraft-part-3`

Lockstep is a peer-to-peer strategy in which each computer has a full model of the game state. Whenever a client's game state changes they must send this update to all of the other clients to apply to their copies of the game state too. There is no centralised server in this networking model, so disconnection of a single client does not cause the game to stop. It also allows users to play games on networks with high latency without actually experiencing any perceived lag. This is because clients apply delayed packets as they arrive. The robustness afforded by this technique is highly desirable for games with long playing times, such as Sid Meier's Civilisation.[58]

[58] Sid Meier's Civilization is a turn-based strategy game, for more information see `civilization.com`.

However, maintaining synchronisation when using a peer-to-peer protocol like this can be very challenging. The clients have to be able to collectively handle messages arriving in different orders for each individual. If this happens the game can start to desynchronise and the games will ultimately diverge. For example, Figure 3.1 shows how two clients can easily diverge. Each pair of grids in the diagram represents the state of the game for each client, with client

one on the left and client two on the right, at the given time step. Client one sees the blue character successfully hit the red character, but client two sees the red character step out of the way and the blue player's shot misses.

In the simple server-client model the server and each of the clients have a copy of the game state, but instead of sharing the updates in a peer-to-peer fashion the server holds the master copy and it distributes canonical updates to the clients. When a player performs an action, the client sends a message to the server informing it of what happened. The server then applies the action to the game state to create an update which it sends to all of the clients. It must be ensured that all update messages are applied in the correct order, but this is much simpler to solve compared to the peer-to-peer version of the same problem since one machine is in charge of the decisions instead of a collective of machines.

The drawback to this approach is that clients must wait for the server to provide updates before they can show the new game state to the users. This can cause considerable lag when the network has high latency or is experiencing loss.

The final strategy was server-client with simulation. This is similar to simple server-client in that each node has a copy of the game state and the server's is deemed to be the master copy. However, clients also perform some simulation using client side prediction of the events about to occur in the very near future. If the client side prediction is incorrect then the authoratitive version of the game state held by the server will eventually correct the client's local state. This helps compensate for any latency on the network. The downside is that there can be noticeable changes in the game state if a correction is not received until a long time after the initial client side prediction failure occured. This strategy is popular with modern first person shooters such as *Counter-Strike* and *Team Fortress*.[59]

The simple server-client strategy was chosen for Project Serenity because of how much simpler it is to implement than the other two approaches. Also, since the game would be developed for LAN usage, latency would be less likely to be an issue. In the future, this strategy could also be extended to include some client side prediction and simulation.

One concern that was raised during these discussion was that of bandwidth. Would the volume of messages being passed between the clients and the server be too great? To answer this a basic design for the update messages was created and calculations were performed to estimate how large and numerous they would be. An example of a simple update message generated by the server would be to change the location of a ship in the world. Such a message would include: an identification number for the ship, new location coordinates, and new direction vector. The use of absolute values in this message instead of deltas reduces the chances of desynchronisation due to packet loss.

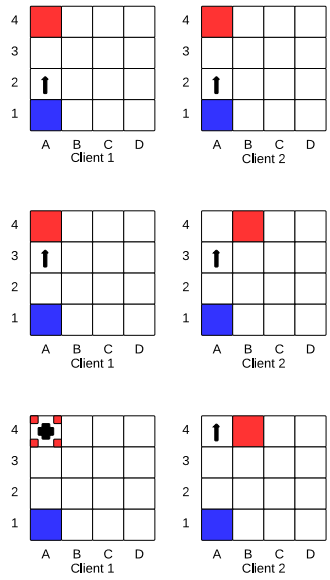


Figure 3.1: Example of desynchronisation when using a lockstep strategy.

[59] Y.W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization, 2001. URL `developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization`

Update Type	*Description*	*Fields*
position	Used when both the ship's location and direction has changed	entityID, location, direction
order	When a ship's order changes	entityID, order
goal	When a ship's goal changes	entityID, goal
plan	When a ship's plan changes	entityID, plan
health	When a ship's hull or shields health changes	entityID, hull health, shield health

Table 3.1: Description of update messages.

The original design of the messaging system only included three update messages:

addEntity Used to add a new entity to the game. It contains the entire details of the new entity.

deleteEntity Used to remove an existing entity from the game. It only needs to contain the entityID.

updateEntity The most common update. It is used to update existing entities. It contains an entire entity definition that is used to replace the existing one.

Having only three types of update was a very simple design and was easy to reason about. However, two of the three updates, addEntity and updateEntity, contained entire entity objects which leads to large messages containing redundant information. Sending an entire entity definition across the network like this would be very slow and should only be done when absolutely necessary.

The addEntity message does need to contain an entire entity since it is adding something new to the game. Since this message will primarily be used at the start of the game this is not a concern.

However, the updateEntity message is used extremely frequently since it would be sent every time an entity changes location, direction, is damaged, receives new orders, and so on. Evey time a single piece of data about the entity changes an entire entity is sent across the network. This would take up a large amount of bandwidth sending a great deal of information that has not actually changed. The solution to this problem was to split the updateEntity message into multiple different update types, one for each common usage of update. Table 3.1 shows the set of different update types that were devised.

With this change to the design, when an ship's field changes only that field is sent across the network. For example, if a ship's

position is changed then only the position data is transmitted. This is significantly cheaper with regards to network usage compared to the monolithic updateEntity.

An analysis on the amount of traffic that would be sent using the new design was then undertaken to ensure that enough savings had been made. Some assumptions had to be made about the number of ships involved in a game, the number of game steps per second, and so on, but in these cases a worst case scenario was assumed to make sure nothing was underestimated. To perform the analysis, the size of each update message and their frequency needed to be estimated.

First, the sizes of various common data types, such as INT and FLOAT, were collated. These figures are shown in Table 3.2. With these figures available it was possible to determine the sizes of different fields in the update messages by working out the type of each field. This data is shown in Table 3.3. For example, a location is known to be a pair of FLOATs and so it can be assumed to consume eight bytes.

Secondly, the size of individual fields was used to calculate the size of whole messages which include multiple fields. Then the frequency at which different messages would be sent by the server was estimated. This data is shown in Table 3.4.

The number of bytes sent to a single client in a single game step was calculated from the data in Table 3.4:

$$\begin{aligned} bytes &= 20 \times 1 + 19 \times 0.2 + 14 \times 0.3 + 104 \times 0.3 + 12 \times 0.1 \\ &= 60.4 \end{aligned}$$

Let S be the number of game steps per second, and N be the number of clients connected to the server. Then the number of bytes the server sends per second is $S \times N \times 60.4$. This formula to calculate network traffic per second was then applied to various likely values of S and N. The results are shown in Table 3.5.

For the greatest expected values of S and N, 60 and 16 respectively, the network usage per second is only 56.62 kilobytes. This is easily acceptable for a multiplayer game running on a LAN.

Type	*Size (bytes)*
INT	4
FLOAT	4
DOUBLE	8

Table 3.2: Byte sizes of numeric fields.

Field Name	*Size (bytes)*
entityID	4
location	8
direction	8
order	15
goal	9
plan	100
hull	4
shield	4

Table 3.3: Byte sizes of update fields.

Update Type	*Size (bytes)*	*Frequency*
position	20	1
order	19	0.2
goal	14	0.3
plan	104	0.3
health	12	0.1

Table 3.4: Size of each update message and its average frequency per game step.

Steps Per Second	*Number of Clients*	*kB/s*
15	2	1.77
15	4	3.54
15	8	7.08
15	16	14.16
30	2	3.54
30	4	7.08
30	8	14.16
30	16	28.31
45	2	5.31
45	4	10.62
45	8	21.23
45	16	42.47
60	2	7.08
60	4	14.16
60	8	28.31
60	16	56.62

Table 3.5: Estimate of traffic generated for various game setups.

3.2 Modelling the Game

The model that makes up the game consists of the internal representation of the various game entities, such as ships and planets, as well as the business logic that is used to manage and control them and the relationships between them. During the alpha phase, this model was rather simple as it only existed to store the locations of the entities. As development progressed, it grew in complexity and feature completeness. Currently there are two sets of models: one for the fleets of ships, and the other for the map of planets and spacelanes.[60] This section will describe and discuss these models and the reasonings behind their design.

[60] Spacelanes are connections between some planets that allow ships travelling along or nearby them to move more quickly. See Section 3.6 for details.

3.2.1 Sectors and Planets

The different game maps are known as sectors. A sector models all of the planets in a battle, how they are connected with spacelanes, and where the different spawn points are located.[61] The actual fields that make up the sector model are shown in Table 3.6. Sectors have different sizes because it allows the players to customise the play time required for a battle to some degree. Choosing a larger sector will typically result in a longer battle than smaller ones. The specification has a requirement for short battles, but it is good to add some flexibility to allow players the freedom to decide how to enjoy the game. It could have been possible to have static spawn points that are enforced for all sectors, for example spawning players in the corners of the sector so that they are as far apart as possible. However, the actual design has per sector spawn points so that the spawn points can be customised to be as fair as possible relative to the layout of the planets within the sector. This decision was made to make the game as balanced as possible to maximise the enjoyment of all players.

Field	*Description*
Name	Sector name
Size	Sector dimensions
Spawn points	List of locations where different players' fleets are spawned
Planets	List of planets in the sector
Spacelanes	List of spacelanes that connect the planets

Table 3.6: Fields of the Sector model.

[61] Spawn points are locations where a player's fleets will be created

The list of planets that make up the sector are specified using the planetary model. Planets are modelled in such a way as to have a number of different fields that make the planet unique. Table 3.7 gives a breakdown of the properties that comprise the model of a planet. Just like sectors, planets have a name to allow players to easily reference a specific planet instead of having to use location information or a numerical identifier. One of the most interesting pieces of the planetary model is the *ecotype*. The ecotype of a planet specifies the type of ecological system that the planet represents, for example the ocean ecotype is used for planets that are almost entirely covered by water. This ecotype has a big effect on the planet because it determines the quantities of the different resources found in the planet.[62] For example, some ecotypes will have a greater proportion of metal (e.g. the metal ecotype). However, the quantity of resources stored by a planet is actually represented by a separate field instead of being statically determined by the ecotype. This was done to make it possible to add some more variety to the different

Field	*Description*
Name	Planet name
Ecotype	Type of planet, e.g. primordial, ocean, or desert
Location	Location of planet within the sector
Resources	Quantity of resources generated by the planet if captured
Capture status	Current owner of the planet, if any, and by how much they have captured it

Table 3.7: Fields of the Planet model.

[62] The use of resources is detailed in Section 3.3

planets. For example, it could be possible to use a random number generator to set the quantities of resources in a planet within the range specified by its ecotype. Currently all resource quantities are specified statically, but this potential enhancement would add some variation to the battles to keep the game interesting for regular players.

3.2.2 *Ships, Classes and Configurations*

The second set of models deal with the fleets of ships that are controlled by each player. A fleet is simply a list of ships which a client sends to the server having chosen which ships to include and having made their customisations to these ships. The representation of a ship is actually broken down into a generic ship model and its specific configuration. The general ship model contains the configuration as well as the attributes that are updated during game play, such as the ship's orders and plans, and its current location and direction. The full list of fields for this model is shown in Table 3.8. The use of location, direction, and damage should hopefully be obvious. They can be updated every tick by the server's main loop as the ship moves around the sector and engages with the enemy. Orders, goals, and plans are used by the artificial intelligence that controls the ships, see Section 3.5 for more details. The list of targets that a ship is currently attacking is also controlled by the artificial intelligence. The most interesting part of the ship is its configuration which specifies what ship class it is and the customisations that have been applied by the user.

Field	*Description*
Location	Current location of the ship within the sector
Direction	The direction the ship is facing
Damage	How much damage the ship has taken
Order	Current order for the ship
Goal	Current goal the ship is trying to achieve
Plan	Current plan to execute in pursuit of the goal
Targets	List of enemy ships that the ship is attacking
Config.	Class of ship and its customisations

Table 3.8: Fields of the Ship model.

The ship configuration contains the chosen ship class along with the lists of weapons and support systems that the player selected to add to the ship. The ship class represents the type, or blueprint, of ship specifying the basic statistics of the ship. The set of statistics defined by the class is: speed, hull health, shield health, details of each of the weapon slots, details of the support system slots, and the cost of the ship. These statistics are used by the simulation code in the server to determine how the ship moves, how much more damage it can sustain before being destroyed, how much damage it does to the enemy ships it is attacking, and so on. A player is able to select the set of ship classes that they wish to include in their fleet, but they are restricted by a fixed budget. There are three ship classes that come with the game by default: corvette, destroyer, and dreadnought. The corvette is the smallest and quickest ship that can only be given one weapon. The destroyer is of middling strength and speed, it allows for five weapons to be added. Finally, the dreadnought is the largest and slowest of the three, but it packs the largest punch by supporting up to nine weapons.

This layered model approach to representing the ships was used to separate the different types of data that have to be stored. The ship model deals with the dynamic data that is updated during gameplay, the ship class is concerned with the statistics that define

the ships behaviour, and the configuration stores the user's customisations. This is useful because it allows a more efficient method of storing custom fleets on disk. Only the list of ship configurations needs to be kept since the class data is the same for all ships of the same type so they can be loaded at run time and the rest of the ship data is only relevant during a battle.

3.2.3 Weapons and Support Systems

Before a battle is started, a user is able to customise the ships that make up their fleet. They do this by first selecting the types of ships to use and then filling the pluggable slots with extra parts. There are two types of part: weapons and support systems. Weapon slots are arranged in three different locations around the ship. Side weapons are used to fire to the port and starboard of the ship. The front weapon slot, if present on a ship, can hold a weapon which fires on the front firing arc. Finally, there are turret slots that allow a weapon to shoot in all directions apart from backwards. Support system slots are used to hold extra internal systems to enhance the ships abilities, such as shield boosting or cloaking devices. This ability to add different parts to a ship was one of the functional requirements laid out in the project specification, so the availability of some parts to use is of great importance to the game.

The ship configuration holds two lists that describe the weapons and support systems that have been added to the ship by the user. These lists represent the filling of the pluggable slots that are part of a ship. The game pre-defines two types of weapons: lasers, which are relatively short-range weapons that inflict a medium amount of damage on enemy ships; and rockets which are much longer range weapons that cause a great deal of damage. Just like ships, the weapon stats are stored in a model. The weapon model contains the range and damage information of each weapon type so that it can be referenced by the server to determine which ships are eligible for targeting and how much damage they should receive.

3.2.4 Defining Model Instances

All of the different ship, weapon, and system types that have been discussed so far are not actually hardcoded into the source code of the game. Instead they are defined in external configuration files that are loaded by the game code at run time — therefore, these types are collectively known as addons in the parlance of the game. This route was chosen because it would allow easy modification of the attributes of these addons without having to recompile the source code. So, players who are not able to program still have the option of changing some of the statistics to customise gameplay. It also possible for players to add their own custom addons to enhance the game with more options for fleet customisation. Allowing the game to be modified in this way is hugely popular in

the gaming community. For example, *Counter-Strike* began life as a community built mod of the game *Half-Life* before becoming the most popular game in the world.[63]

The addon definitions are stored in YAML files in the resources directory. YAML was chosen as the file format due to its focus on human readability. Other serialisation formats, such as XML and JSON, were considered, but were discarded in favour of the combination of brevity and readability afforded by YAML. When the game is launched, the resources directory is scanned for YAML configuration files and the data is loaded into memory. Adding new instances of models only requires adding a new YAML file to the appropriate directory, and configuring existing ones is as simple as editing a text file. Figure 3.2 shows the corvette ship type defined in YAML.

[63] Vic. A history of counter-strike, 2012. URL `lambdageneration.com/posts/a-history-of-counter-strike/`

corvette.yml

```
shipName: Corvette
fileName: Corvette
centerOfRotation: (512, 128)
speed: 2.0
cost: 1000
damageStrength:
  hull: 50
  shields: 75
weaponSlots:
  -
    location: (1, 2)
    direction: (10, -5)
    type: Side

systemSlots:
  -
    location: (18, 12)
    direction: (10, 15)
  -
    location: (18, 12)
    direction: (10, 15)
```

Figure 3.2: Example of a YAML configuration file.

3.3 *Resources: The Currency of the Game*

The game was designed to include a resource system where players would collect resources to be used to power their fleets. Resources represent the currency of the game and a player's access to resources has a dramatic affect on their chances of winning a battle. There are three different types of resource: fuel, metal, and anti-matter. Each resource has a unique purpose in the game, requiring the player to modify their strategy according to the quantity of resources they have access to.

Fuel Fuel is used to regenerate a ship's shields. As discussed previously, shields are vitally important to the protection of a ship since they prevent or reduce the amount of damage that is done to the ship's hull. Without fuel a ship would be much more vulnerable to attack since it is unable to replenish its shields. An extension to this concept would be to require ships to have fuel to power their shields at all, however, this potential dynamic has not been explored fully.

Metal If a ship's hull has been damaged then the supply of metal can be used to slowly repair it. Access to a greater quantity of metal allows a player to adopt a bolder strategy since their damaged ships can be brought back to full health if they are not destroyed completely. Without metal any damaged ships become easy prey for the enemy since they are unable to recover from attacks and will quickly be destroyed.

Anti-matter There are certain very powerful weapons that a ship can be equipped with. However, these weapons require a supply of anti-matter, the rarest of the three resources, to power them. Having access to anti-matter allows a player to unleash devastating attacks from these special weapons and could possibly turn the tide of battle in their favour.

A player is able to mine resources from the planets that are under their control.[64] As discussed in Section 3.2, each planet in the sector has a supply of resources that varies depending on its ecotype. Planets that are captured by a player will provide a constant trickle of resources to the owner.

[64] See Section 3.4 on how to gain control of planets

Although the framework for the resource system is in place, the actual utilisation of resources has not been implemented yet. This is because discussions about the mechanics of resource consumption never really came to a concrete conclusion. So, it was decided that it would be better to focus on the other planned features and to return to resources if there was time. The remainder of this section discusses the two potential resource systems that were debated.

The first possibility is that each ship in a player's fleet has an individual supply of resources. When the ship is stationed near to a friendly planet then it will be able to resupply from the player's global stockpile. The quantity of resources that an individual ship

can store for use would be dependent upon its type, larger ships being able to carry more supplies than smaller ones. Once a ship leaves the safety of home territory and starts receiving damage or using special weapons it will start to drain its internal resource cache. The ship's supplies will eventually empty out and it would have to return to a home planet to reconnect to the resource supply. If a ship continues to battle without resources then it will eventually sustain damage that it is unable to repair and will be more likely to get destroyed.

The other option would be to only have a single global supply of resources. Ships would then drain that global stockpile as needed no matter where they are located in the sector. This mechanism would probably encourage more aggressive playing styles since ships will be receiving resources to help repair themselves so long as there are resources available. To ensure there are enough resources to supply this rampaging fleet the player's strategy is likely to focus on capturing as many planets as possible, a attacking style of play.

There are advantages to both options which is why a relatively long time was spent discussing and deliberating the design. The former design is more realistic since ships are not magically furnished with extra supplies whenever they require them, instead they must travel back to a friendly planet to restock. It also would give rise to a more tactical and positional style of play since the players must keep their ships within reach of fresh supplies. However, this requirement to continously return to base could lengthen the battles which are supposed to be short. The second option is simpler and might be easier to implement since only one resource count needs to be stored for each player instead of storing counts for each ship in play.

Another question about the resource system that needs answering is whether or not planets have a finite supply of resources. If planets have an infinite resource supply then they will continue to produce resources for their respective owners every game tick until the game ends. On the other hand, a finite supply of resources per planet would mean that planet owners can mine resources from their planets as before, but after a while that supply of resources will dry out. The benefits of this finite design would be that it is more realistic since planets cannot actually hold an infinite amount of material, and that it could be a good way of enforcing shorter battle times — a player would want to gain as many resources as possible which means attacking as many planets as possible and taking the fight to the other player as quickly as possible.

It is a shame that the full resource system was not added to the game before the end of term two. However, it was important that its design had a chance to be carefully thought out and correctly specified otherwise it could have ended up detracting from the entertainment instead of enhancing it.

3.4 *Planetary Capture*

Planetary capture is another of the major functional requirements captured by the game specification. It is the method by which players expand their territory and gain access to more resources. The basic idea of planetary capture is that if a player controls the area surrounding a planet, i.e. there are no enemy ships in the vicinity, then they are able to start capturing. After a certain period of time the capture is complete and the planet is now owned by the capturing player.

Every planet stores its current owner and the percentage by which it has been captured. If the percentage captured is less than one hundred percent then the planet is said to be partially captured. A planet's owner is the only player who is able to exploit its resources. Figure 3.3 shows a planet that is approximately seventy-five percent captured by a ship owned by the red player.

Figure 3.3: The planet "Castillon" being captured by a red team ship

$n \leftarrow$ number of different players with ships in vicinity of planet p
if $n = 1$ **then**
  $o \leftarrow$ current owner of planet p
  $a \leftarrow$ player with ships in the vicinity of planet p
  $c \leftarrow$ percentage completion of planetary capture
  if $o = a$ **and** $c < 100$ **then**
    Increment the percentage captured by two
  elsif $o \neq a$ **and** $c > 0$ **then**
    Decrement the percentage captured by two
  elsif $o \neq a$ **then**
    Change planet ownership from o to a
  fi
else
  $c \leftarrow$ percentage completion of planetary capture
  if $c \leq 0$ **then**
    Reset the ownership of planet p to nobody
  elsif $c < 100$ **then**
    Decrement the percentage captured by one
  fi
fi

Listing 1: Pseudocode for the mechanism of planetary capture

The full mechanism by which a planet, p, is shown in Listing 1. If only a single player is in control of the area surrounding a planet then they either increase the percentage by which it is captured if they are the current owner. If another player is listed as the current owner then the percentage captured is decreased until it reaches zero and ownership is transferred. However, if there are no players or multiple players in the vicinity of a planet then the ownership of partially captured planets starts to decay until it reaches zero and the planet owner is reset.

3.5 *Artificial Intelligence: Orders, Goals, and Plans*

A sophisticated artificial intelligence (AI) system was highlighted as one of the major functional requirements for the project. The plan was to use the AI to take high level orders from the player and convert them into a detailed plan for individual ships to execute. In this way the player would direct the overall strategy of their fleet, but without actually captaining the individual ships. This would reduce the amount of micromanagement a player has to undertake and increase the sense of realism — the admiral of a real fleet is unable to directly control individual ships under their command. This plan for a high level AI was seen as one of the larger challenges that the project would face.

Artificial intelligence can often be a make-or-break factor in determining the success of a game.[65] Without a convincing intelligence system, a game can quickly become infuriating to play. This is because a human player expects any computer controlled components to behave sensibly. In some cases well known algorithms exist that enable 'intelligent' behaviour to be implemented relatively easily, for example the use of the A* search algorithm for pathfinding. However, higher level intelligence systems are much more challenging. A system capable of creating and executing quality plans from abstract orders is going to be one of the hardest components to implement.

[65] S. Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002, page 3

As well as providing an entertaining experience an AI system must also be efficient. There cannot be large delays between the user giving an order and it being carried out. Any planning algorithms have to run quickly otherwise the lag in feedback will detract from the realism of the game. An inefficient AI system could also stop the game from running smoothly — which is of great importance for a real-time strategy game. This would lead to a poor user experience causing people to stop playing the game. For these reasons, careful thought was put into designing a system that could fulfill the important requirements for a successful AI system.

The AI framework that was implemented in Project Serenity is based on a planning hierarchy. At highest level are orders, such as move to a given location or capture the specified planet. These orders are under the control of the player, but it is the job of the AI system to plan a series of actions to achieve them. The first step is to convert the order into a goal. There is mostly a one-to-one mapping between orders and goals, for example OrderMove maps to GoalBeAt, however this layer of abstraction allows the AI to choose unrelated goals if the current order is impossible or would result in the needless loss of a ship. The goal is then decomposed into a plan which is a series of actions that are performed to complete the current goal (and hopefully fulfill the current order too).

The current planning cycle for a ship is as follows:

```
if the current plan is empty then
  if the current order is complete then
```

Listing 2: Pseudocode for the AI framework's planning cycle.

```
    Reset the ship's order to ORDERNONE
  else
    g ← create goal from the current order
    p ← create a plan from the new goal, g
    Update the ship's goal and plan with g and p
  fi
else
  if the first action in the current plan is complete then
    Remove the action from the head of the ship's plan
  else
    Perform the action at the head of the ship's plan
  fi
fi
```

This planning cycle runs every server tick so that all ships in the game can continuously formulate and update their goals and plans.

An improvement to the planning cycle would be to add in periodic replanning steps. This would allow a ship to change its plan if the world around it changes, for example to disregard an order to capture a planet if it is guarded by multiple enemy ships and there is no friendly support in the area. In the current implementation this is done on a bit of an ad-hoc basis instead of being formalised into the planning loop. For example, the code that implements the action to move to another ship will check if the target ship is still near to its location when the plan was originally formulated. If the target ship has moved too far then the current plan is scrapped so that a new plan can be constructed.

3.5.1 Targeting Enemy Ships

As well as constructing a planning framework for executing player given orders, it was necessary to build some systems to carry out intelligent actions outside of the usual planning loop. One such example is that of targeting enemy ships to attack. As a ship moves about the sector it will fire upon any enemy ship that it encounters. However, if there are multiple ships in the vicinity then it is necessary to select which to target as it may not be possible to attack all of them at once. A clever targeting system had to be devised to solve this problem in an effective manner.

The first step is to generate a list of all potential enemy targets. This is a list of all enemy ships that are in range of any of the ship's weapons. A list of enemy ships to attack is selected for each weapon by taking the enemy ships that would be attacked by the least number of other weapons as well. This method of target selection means that a ship will spread its attacks out to hit as many enemy ships as possible.

This pattern of list generation and selection fits the one described by John Hughes very well.[66] This is one of the reasons why the Project Serenity team found Haskell to be highly adept for programming AI algorithms. The separation between the potential

[66] J. Hughes. Why functional programming matters. *The Computer Journal*, 32 (2):98–107, 1989

target list generation and the selection from the list also means that it would be very simple to modify the selection strategy. For example, a possible enhancement to the game would be to allow players to configure a targeting strategy for their fleet which would focus attacks on the weakest enemies.

3.6 *Ships, Spacelanes, and Path Finding*

The specification made it clear — by the way weapons and navigation between planets was to be structured — that ship motion needed to be relatively detailed. It is no use having a weapon that can be only used in one arc if ships could instantly pivot on the spot. Instead it was desired for ships to move as large objects with a great deal of momentum, with large turning circles and ponderous movements. However it would be undesirable for the motion of ships to be too realistic, as it would make the game needlessly difficult and confusing. A real object undergoing acceleration in space could of course, accelerate essentially indefinitely (until it was nearing the speed of light) but would need to take an equal amount of time to decelerate to stop again.

Intimately related to these questions is the precise nature of the *spacelane* mechanics. Spacelanes between planets were introduced into the game in order for the otherwise empty space between planets to have some strategic topology, the idea being that the spacelanes can be traversed much more quickly than empty space. and so control of the lanes are vital to a successful strategy. Many different models were discussed for this mechanic during design meetings, both from the point of view of the in universe explanation, and the precise way in which the game would implement it. Initial ideas on the physical mechanism of the spacelane was that they conferred an additional optional acceleration: i.e. that if a ship is approximate to a lane it could opt to undergo greater acceleration than a ship far from a lane. But as mentioned, accurately modelling acceleration in space is somewhat contrary to the gameplay aspirations of the design.

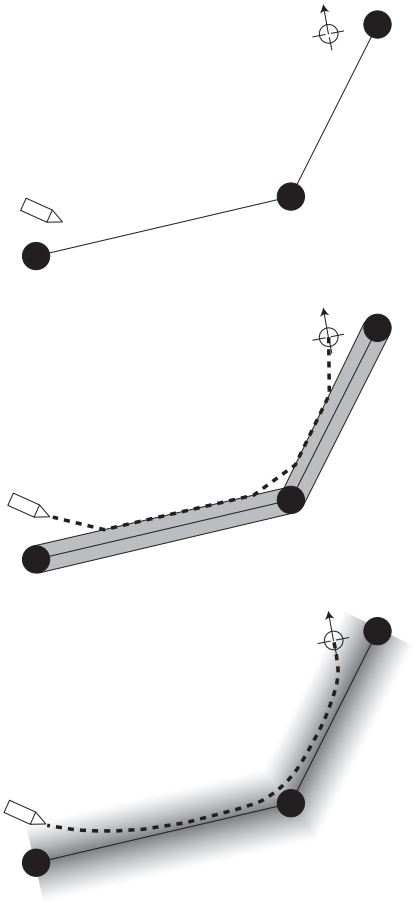


Figure 3.4: Illustrations of different models for a path between current location and target utilising a spacelane.

A simpler mechanic for ships that was considered to be more suitable, is for them to have a top speed that they could reach relatively quickly; but for them to still have large turning circles. This makes their motion more similar to naval ships engaged in warfare. The interpretation of the spacelane in this model cannot be simply a gain in acceleration as this confers little benefit. Instead it is clear that the spacelane must confer an addition to top speed. A possible physical interpretation of these behaviours is that the ships are moving in some kind of ether with friction, and so will reach a terminal velocity. The spacelane is then an area with an artificially induced lower density of ether.

Another fundamental question about the nature of the spacelane is whether its effects are immediately felt at full strength at a certain distance from it, or whether they fall off related to distance. The former is most definitely simpler, but the latter feels more natural as a mechanism. Figure 3.4 shows the difference in the shortest paths that each of these approaches may generate, discounting the additional issue of ship turning.

3.6.1 *Algorithms for Pathfinding*

Physical simulation of a large mass moving and turning in space was the initially preferred implementation method, but it became increasingly clear that this was going to be difficult both from an implementation and a user perspective. And while modelling the spacelane as having a continuous, field like nature, it was not at all clear how this would be achieved. It was decided to instead merely approximate the desired behaviour using bezier curves.

Over the Christmas break the code for a ship moving between two arbitrary points and headings was developed. Coding bezier paths themselves was not difficult, but moving an object continuously along one at a constant speed was much more complicated than expected. The accurate solution of this problem is called reparameterisation by arc length, and requires a solving a differential equation, but the method eventually used in Serenity was a numerical interpolation. Results are shown in Figure 3.5.

Figure 3.5: Bezier path at regular intervals (small circles) and the same path after having been reparameterisated by arc length (large circles).

Even with the ability to move a ship smoothly along a given path, the selection of the path still required some considerable thought. An example of the control points generated for a move is shown in Figure 3.6.

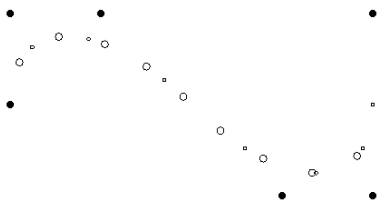


Figure 3.6: Bezier path at regular intervals (small circles) and the same path after having been reparameterisated by arc length (large circles).

3.6.2 *Applying A* to a Sector*

Applying a pathfinding algorithm to a sector has two stages: generating a graph which represents the sector, and running the algorithm that traverses the graph to find the shortest route to the destination. The algorithm chosen for the second stage is A* because it will always find the optimal solution (assuming the heuristic is admissible) and, if an efficient heuristic is used, then it is very fast too. However, the difficulty of using A* is that it requires a discrete graph to navigate, but the sector is not discrete as ships can move anywhere within it. The planets and spacelanes provide some points of reference, but they do not restrict the movement of the ships. Therefore, it is necessary to build a discrete graph from the sector model during stage one.

Just using the planets as nodes connected by edges of spacelanes does not work since the ships can go into other areas of space not covered by such a graph. Advanced techniques such as constructing a navigation mesh or a waypoint graph do not really apply in this situation since there are no obstacles for graphs to be built from. So, the solution was to take the simplistic graph generated only from the planets and spacelanes, and then add extra nodes and edges where appropriate. Extending the graph with these extra nodes was the difficult problem that had to be solved.

First a typical ingame situation was taken to be analysed for the possible routes that a ship could take. Figure 3.7 shows such a scenario. Figure 3.7 (a) shows the location of two ships, P_1 and P_2, and three planets, A, B, and C, connected by spacelanes. If both ships want to reach planet C then it is obvious which route P_2 will

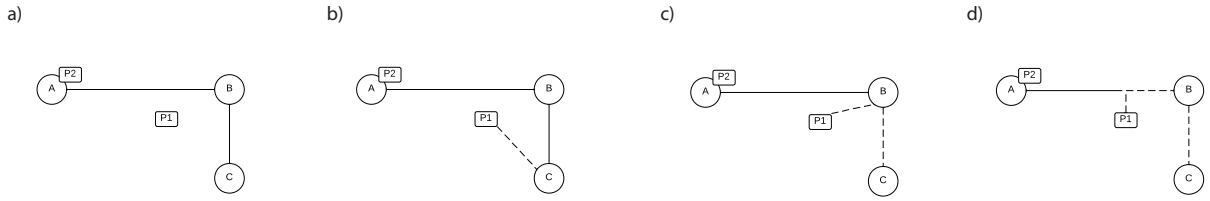


Figure 3.7: Potential routes a ship could take to reach its destination.

take, it can travel along spacelanes from A to B to C. However, there are three different possibilities for ship P_1:

1. Go directly to planet C, shown in (b).

2. Go to the closet planet, B, and then follow the spacelane to C, shown in (c).

3. Travel to the nearest spacelane, between A and B, and then follow the spacelanes to C, shown in (d).

Travelling across open space is always going to be considerably slower than using spacelanes, so option one can be immediately be discarded as suboptimal. However, how fast are spacelanes? Is it best to greedily use spacelanes as much as possible or is it sometimes faster to travel a little longer through open space to reach a spacelane closer to the destination. Spacelanes have a multiplier effect on the speed of a ship so that a ship travelling along a spacelane speeds up x times compared to its usual speed, where x is a parameter defined by the sector model.

Suppose that x is very large or infinite then travel along spacelanes with be almost instantaneous. Under these circumstances option three will be the optimal route since the least time is spent travelling off of spacelanes. However, for a lower value of x, option two will be optimal since the reduced distance to reach planet B outweighs the benefits of the spacelane. Therefore, it is obvious that the heuristic cost function used by A* must take the multiplier effect of the spacelanes into account:

$$cost = \frac{distance}{speed \times multiplier}$$

Where the *distance* between two nodes is simply the Euclidean distance. Using this cost function it becomes easier to find a good route by adding 'virtual nodes' to the graph on nearby spacelanes and then using the A* algorithm to compare the costs of this extended graph. For example, in the scenario shown in Figure 3.7 a virtual node is added to the spacelane between A and B. This suggests a simple set of rules for adding new nodes:

1. If the start point is in open space then add a node at the closest point on the closest spacelane

2. If the destination is in open space then add a node at the closest point on the closest spacelane

For example, Figure 3.8 shows how two nodes are added at N_1 and N_2 if a ship wants to go from P_1 to P_2. The graph is then completed to make all of the sensible routes connected. Using the same example from Figure 3.8 it means that edges will be added between the P_1 and N_1, P_2 and N_2, P_1 to B, B to P_2, and so on. The A* algorithm, using the simple cost function, can then be applied to this fully connected graph to find the best possible route to the destination.

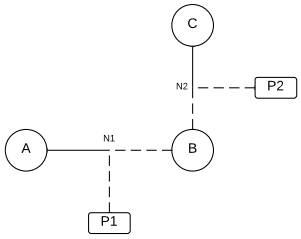


Figure 3.8: Adding nodes to the nearest spacelanes.

However, these two rules are too simplistic and will not provide the best route in all cases. Figure 3.9 shows an example in which a ship wants to travel from P_1 to P_2. The current ruleset will add nodes N_1 and N_2 since they are the closest points on the nearest spacelanes. However, it is actually quicker to travel via N_3 than it would be to go straight from N_1 to P_2. There are just too many possibilities for quicker routes as the edge cases keep on building up. Therefore, it was decided to use a brute force solution that involves adding nodes at regular intervals along the nearest spacelanes. The best added node may not match the optimal solution exactly, but it will be better than the majority of other options. Using brute force is also not a highly desirable technique, but it works fast enough for this use case for now as further designing would have been required to reach a better solution.

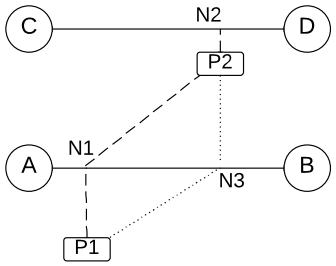


Figure 3.9: The closest point on the closest spacelane is not always your friend.

Now that a discrete graph has been generated to represent the sector it can be fed to the A* algorithm which will find an optimal solution for the graph it has been presented.

3.7 *Rendering the Game*

One very important aspect of the client program is rendering the current game state to a display for the user to interact with. Creating a picture of the game to display requires a number of graphical assets to visually represent the various entities and scenery that make up the game world. For Project Serenity all of these graphics were created from scratch using Photoshop and Illustrator. Figure 3.10 shows a selection of some of the assets that were created for different entities.

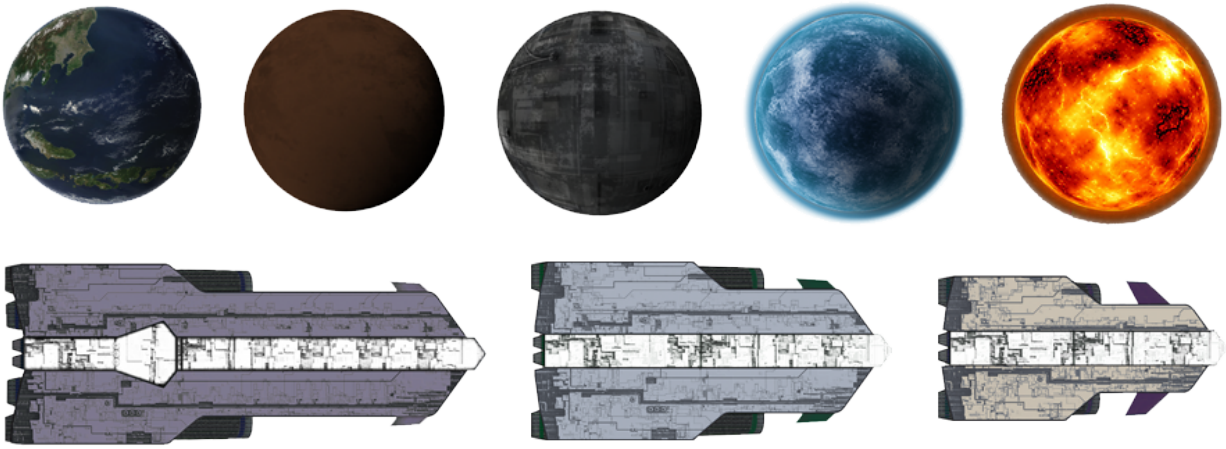


Figure 3.10: Planet and ship textures created for Project Serenity.

The client uses these assets to draw the sector and place each player's ships in the correct locations. This is done by placing the assets using world coordinates in the 2D space defined by the dimensions of the sector. The picture that has been built up so far is then translated and scaled according to the current panning and zoom settings stored in the client. This allows the user to smoothly zoom in and out as well as scroll around the sector. There are optimisations that could have been performed, such as working out which entities would be displayed and only drawing them instead of drawing them and scaling them out of view, but in the end there were not required.

Along with the actual world there is some extra GUI elements to help the player keep track of the current state of the game. Firstly there is a minimap in the bottom left corner. This displays a schematic representation of the sector with the locations of friendly ships. The minimap allows the user to quickly check on the current status of their fleet. It was originally hoped that the minimap could be used to show extra events such as known locations of enemies and current attacks. It could also be enhanced by allowing the user's view to be transported to the location clicked on the minimap. However, core features were prioritised over this extra functionality, and so they are not implemented yet.

The second piece of GUI is the ship and planet selection indicators. When a user left clicks on an entity, or drags a selection

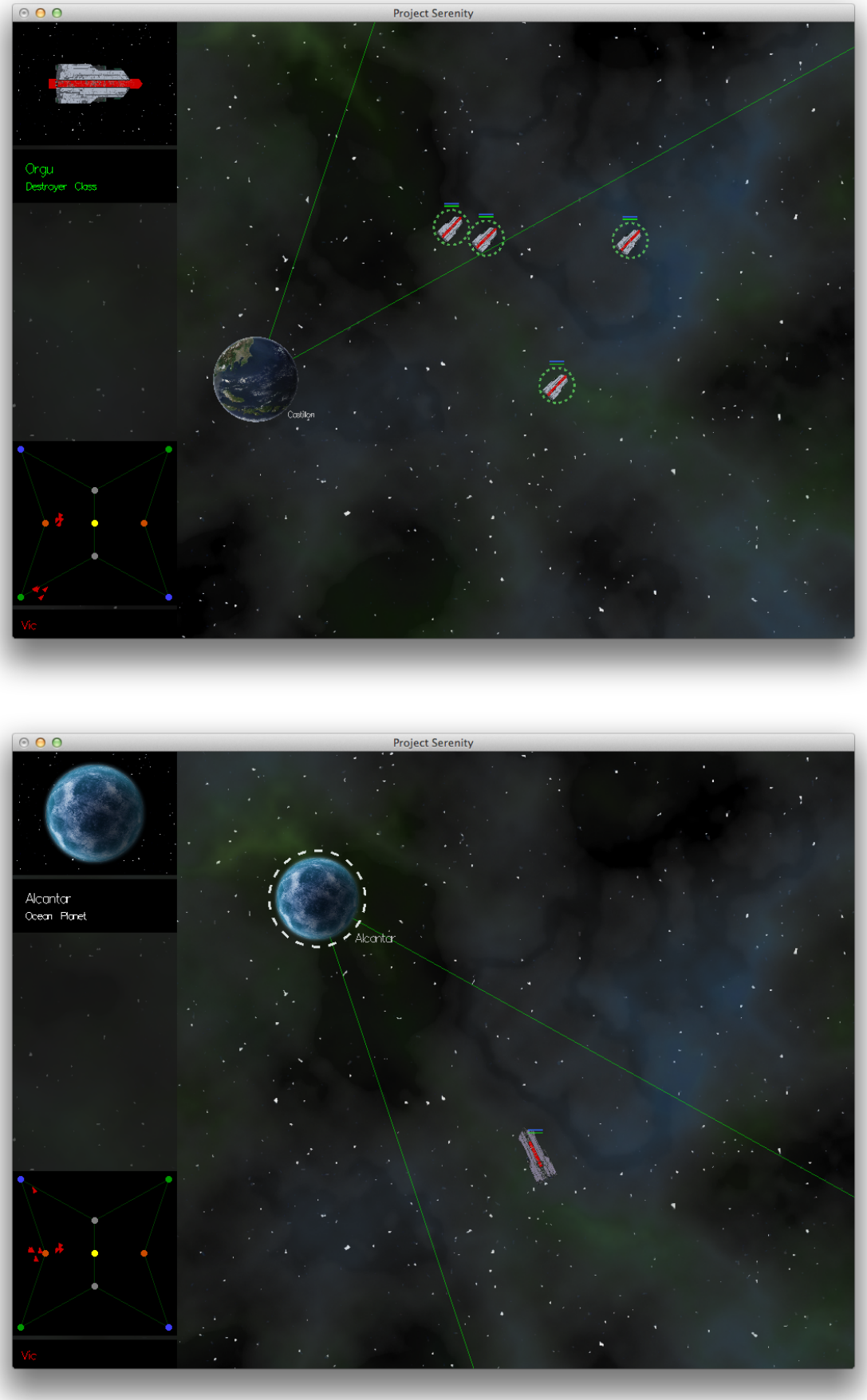


Figure 3.11: Screenshot during gameplay showing a planet, spacelanes, selected ships, and minimap.

box over a group of entities, they are selected and a dashed selection circle surrounds the entity. To make this feature easier to use a group selection does not select all entities within the drag area. Instead some are prioritised over others: ships are ranked above planets, and friendly ships are ranked above enemies. For example, if a selection is made over the entire sector then only the friendly ships will be selected. If a single entity is selected then details about it will be shown in the top left of the screen.

Figure 3.11 shows how similar the resulting game was to the initial mockup made during the specification stage (see Figure 2.8).

Another interesting piece of the rendering puzzle was to choose a colour as a unique visual representation for each player. This would be used to help players differentiate their ships from the enemy and to identify the planets that they own. A clever algorithm was used that would take a numerical player identifier and return a unique colour:[67]

$$colour(i) = HSV(i \times \phi^{-1} \mod 1, 1.0, 0.8)$$

This function uses the HSV colour space using a fixed saturation and value, but modifying the hue to create equally bright and vibrant colours. The player's identifier, i, is used to step into the possible values for hue by multiplying it by the reciprocal of the golden ratio, ϕ, modulo one. Due to the equidistribution theorem[68] this creates a sequence of colours that are evenly spread across the colour space. Using this method to generate team colours created visually pleasing colours that are suitably distinct from each other to be used for recognition.

[67] M. Ankerl. How to generate random colors programmatically, 2009. URL `martin.ankerl.com/2009/12/09/how-to-create-random-colors/`

[68] The equidistribution theorem states that the sequence $a, 2a, 3a, \ldots \mod 1$ is uniformly distributed when a is an irrational number.

3.8 *Menus and a Splash Screen: The Front End*

During the alpha stages of the development, the command line was the only way to specify whether a Serenity process should run as a client or a server, and what address and port to use, and so on. A significant part of the work in the beta stage (after Christmas) was to add the front end menus and UI. A whole framework was developed to build the menus, links between the different modes, and animate a splash screen.

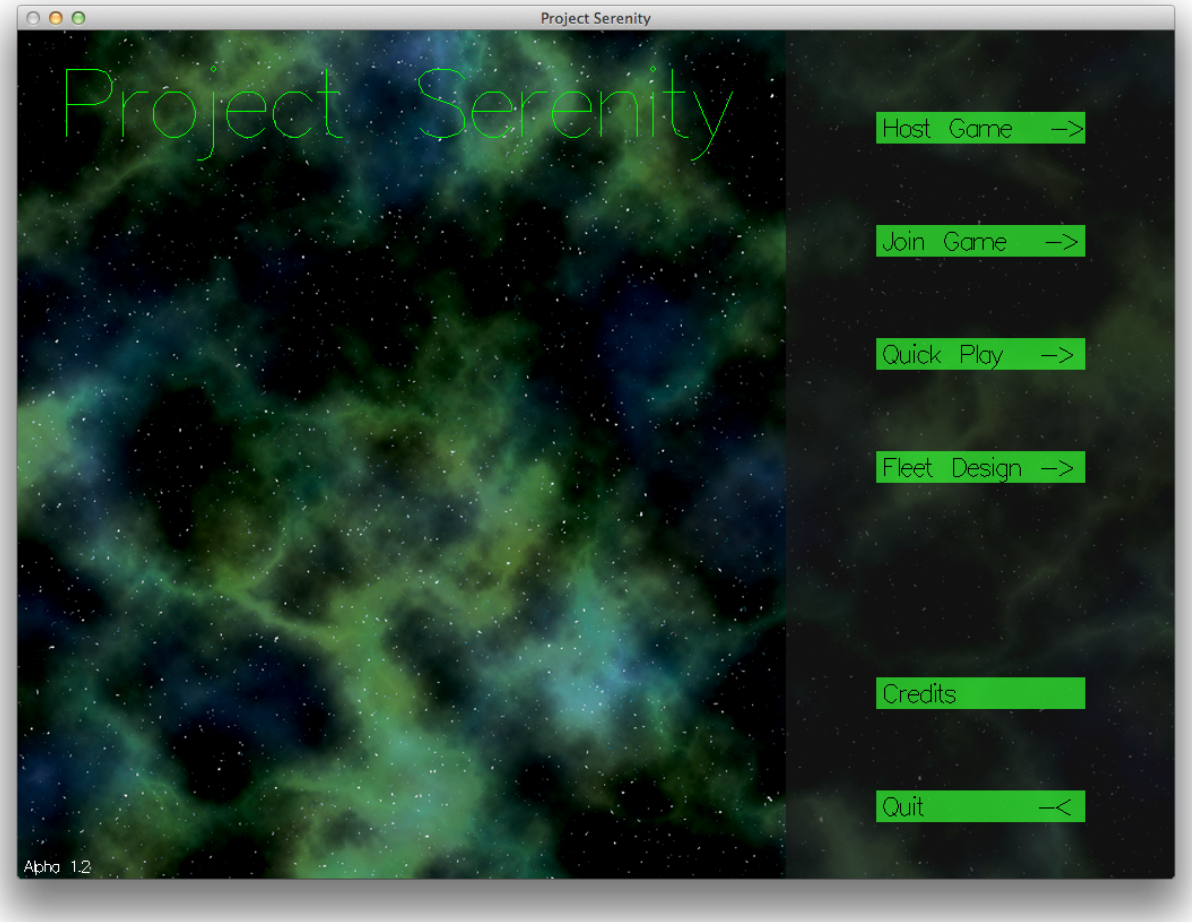


Figure 3.12: Screenshot of the main menu.

Figure 3.12 shows the main menu design. The font is provided by Gloss, and while it is far from perfect it wasn't worth spending development time on improving it. The Splash screen animation was added mostly to help test the burgeoning framework, but it certainly adds a feeling of professionalism to the application. Two screenshots from the splash screen are shown in Figure 3.13.

Considerable thought went into the design of the UI for hosting and joining a game. Controls enable and disable in a natural way that is clear in its method of use and no invalid state cannot be reached. These screens are shown in Figure 3.14.

A credits screen was also added to provide some information

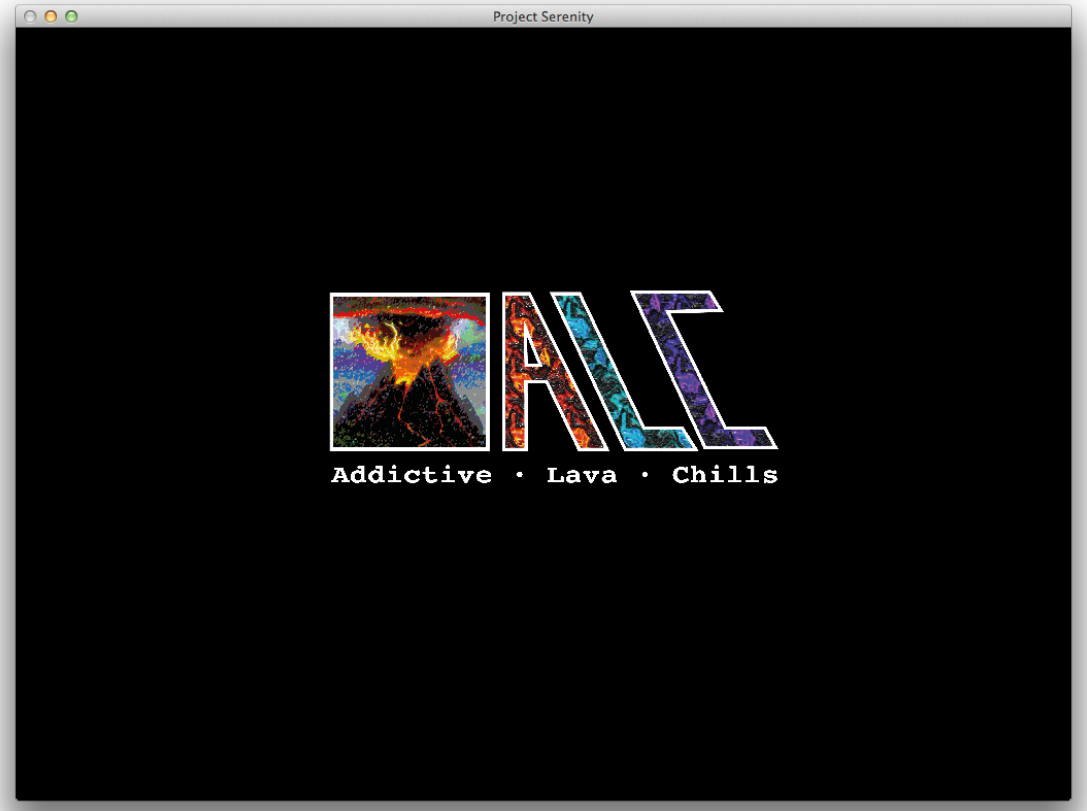


Figure 3.13: Screenshot of the splash screens shown when the game is launched.

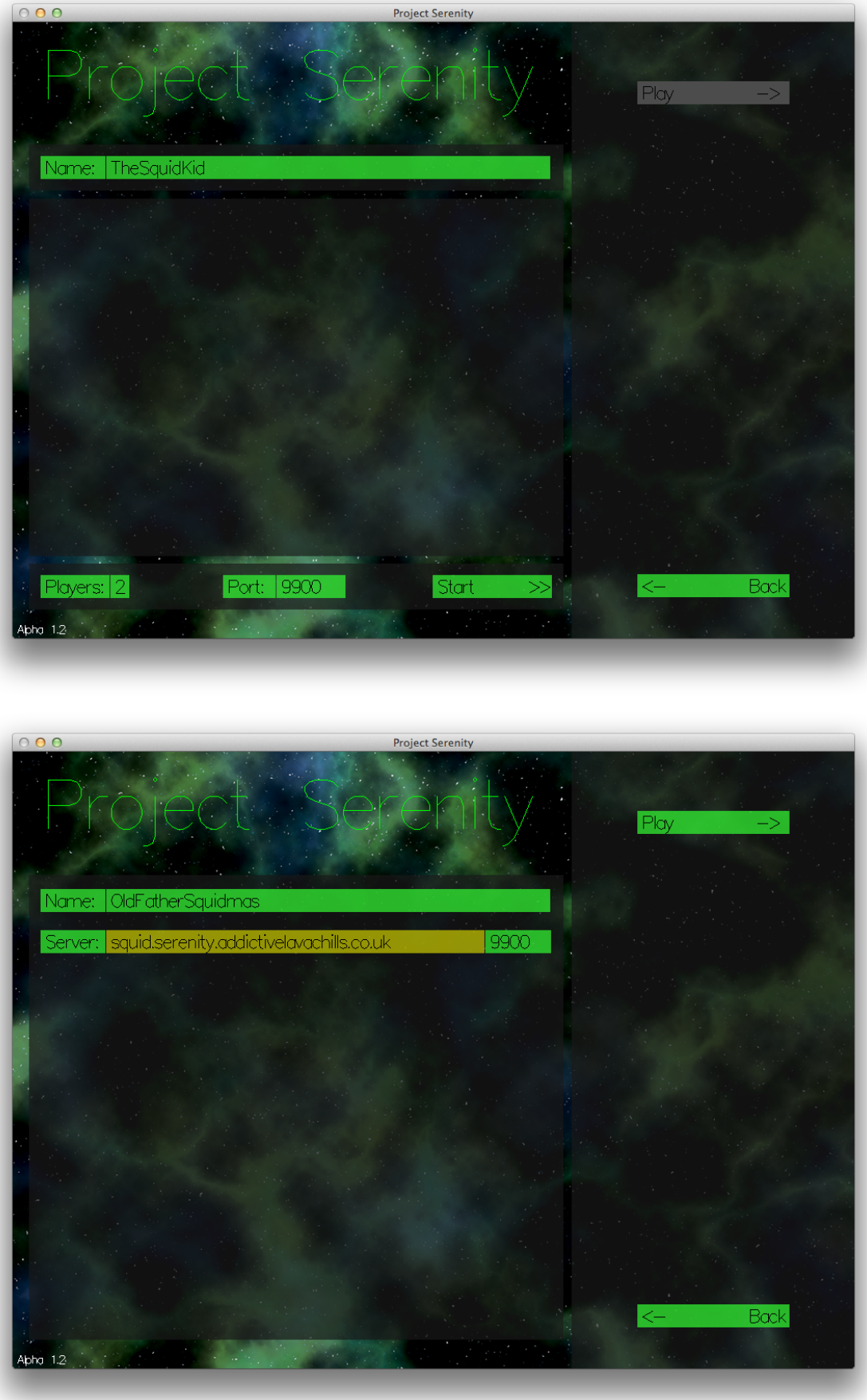


Figure 3.14: Screenshot of the menus used to host and join battles.

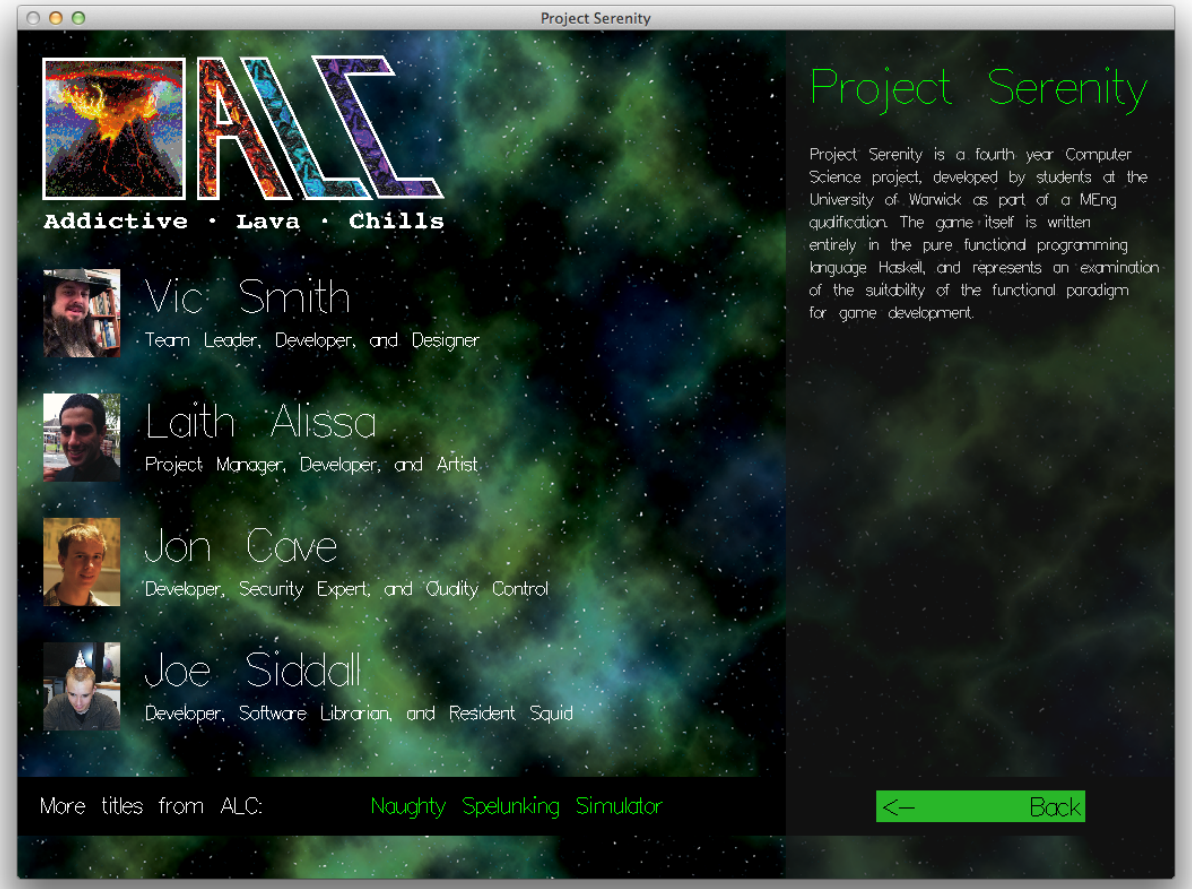


Figure 3.15: Screenshot of the credits screen, showing the development team and the description of the project.

about the project and the developers. A random game name generator sneaked in as an easter egg. The credits screen is shown in Figure 3.15.

The logo featured in the splash and credits screen is for the name given to the project group: *Addictive Lava Chills* (formed from an anagram of the member's names). Both the green starscape used in the menus, and the layered nebula and stars used in the main game graphics were created from scratch using Guassian noise filters and cloud effects in Adobe®Photoshop.

4 Findings: A Practical Guide to Haskell Game Development

THE DESIGNER OF A NEW SYSTEM MUST NOT ONLY BE THE IMPLEMENTER AND THE FIRST LARGE-SCALE USER; THE DESIGNER SHOULD ALSO WRITE THE FIRST USER MANUAL. . . IF I HAD NOT PARTICIPATED FULLY IN ALL THESE ACTIVITIES, LITERALLY HUNDREDS OF IMPROVEMENTS WOULD NEVER HAVE BEEN MADE, BECAUSE I WOULD NEVER HAVE THOUGHT OF THEM OR PERCEIVED WHY THEY WERE IMPORTANT.
— Donald E. Knuth

The previous chapter considered the finished game itself, in this chapter the technical details of the development are considered, primarily in terms of Haskell and FP. The aim of this chapter is to show that, while some of the techniques used in Haskell programming are different from the more traditional approaches, the overall performance of the language is comparable, and even sometimes favourable, to other wider used platforms.

The idea that target code produced from functional languages is slow and performs badly, and also that functional code is either hard to understand or bad from a software development point of view, persists among some practitioners despite various evidence to the contrary.[69] It is likely that one of the main reasons for this is unfamiliarity, as mentioned in the main introduction.

It is hoped that the review of techniques used for Project Serenity as reported on in this chapter, can in some way help to reduce this problem. For this reason the findings are given somewhat in the style of a guide: a guide on what to do (and what not to do) when embarking on a large scale Haskell project. Because Project Serenity was a game, the focus is mainly on aspects relevant to game development; but, as mentioned in the introduction, game programming incorporates a great many areas of software development.

[69] D. Roundy. Darcs: distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4. ACM, 2005

4.1 *Architectural Design — Code Organisation and Separation of Concerns*

When coming from an OO background, one of the first problems encountered when embarking on a large FP project is how to organise and architect the code. There is a certain paucity of literature pertaining to this issue; and while there are many projects on Hackage that provide good examples, the principles behind them are not always clear.[70]

[70] cf. Section 2.2 on page 24.

There are two related issues here. First is simply what files should have what code in them; and second, how to separate concerns effectively. These can be seen as quite separate issues, but it is helpful to consider them simultaneously here, as should become apparent.

Common practice in OO languages is to have one class per file[71] but it is not immediately clear what principle in FP should determine the contents of a file. Instead, the approach taken in the Serenity project is to allow *form* to follow *function* (no pun intended); that is, to allow the desired separation of concerns to drive the module layout — adjusting as becomes necessary — rather than a specific type of code concept.

[71] See `www.oracle.com/technetwork/java/codeconv-138413.html` (rev. 1999) and `www.possibility.com/Cpp/CppCodingStandard.html#cflayout` for example. (Accessed April 2013).

The main separation of concern that every Haskell project is going to contain to some degree or another is between pure and impure code, and so the first thing to be considered in designing an architecture for a project should be how the connection and communication between these areas is going to be managed. At first sight it would appear that almost all of what happens in a game is IO of some sort or another, but this is not the case. Code can be pure precisely when its output can be defined in terms of its arguments *only*.[72] Motion of a particle in space experiencing a force due to gravity, for example, can be defined in terms of pure code. The process of writing log entries into a file will not be, (although the code that designated the actual content of the entries could be).

[72] This is a simplification, but will suffice for the purpose of this text. See K. Menger. The ideas of variable and function. *Proceedings of the National Academy of Sciences of the United States of America*, 39(9):956, 1953.

This is all very well, but does not make it immediately clear how to break down a project's code. The approach taken during the design of Serenity's module structure, and the approach recommended by this guide, is to let the structure of the main loop of each runnable component guide the nature and interface of each module.

To demonstrate how this works, consider the architecture of the Serenity project. As discussed elsewhere,[73] the game uses a server-client model, but with none or very limited simulation clientside. There are therefore two runnable components, which will share some logic. Each main loop will therefore have two main impure parts: the receiving of IO over the network from the server or client, and local IO (be it output to the screen, logging, or input from the user).

[73] Section 2.4 page 33.

It is clear that some data structure must exist that models the state of the game. Because no simulation takes place on the client,

all the updates to this structure will be due to messages from the server. User interactions also need to generate messages *to* the server. So the client loop will consist of reading input from the network and updating the game state, rendering the screen, then reading input from the user and sending output to the network.

Conversely the server loop must read input from the clients and store it, simulate the game world and send the result of this simulation to all clients. Figure 4.2 is a diagram of the interaction between these two loops.

From this simple analysis of the required basic functionality, several concepts have already been defined. On top of the loops themselves, there is:

- the network exchange (which comes in two flavours, client to server and server to client)
- the game state itself
- the rules for updating the game state in reaction to updates from the server
- the rules for updating the game state in reaction to time passing (i.e. the simulation)
- The rules for rendering the screen

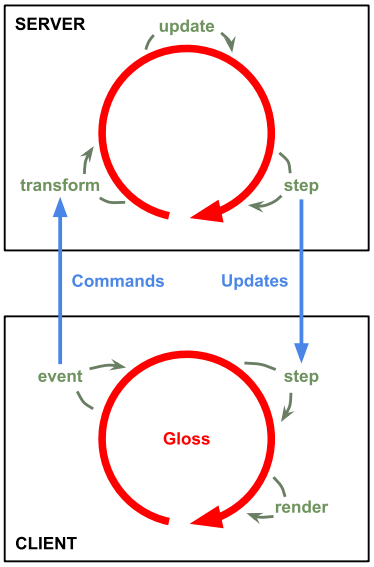


Figure 4.1: Depiction of server and client main loops. The main loops are in red, the network exchanges in blue, and pure function calls in green.

It should be clear that this has immediately imparted a basic way of breaking up the project, and if we examine the top level module namespaces in the early versions of the Serenity source it can be seen that they correspond largely with the above list (the main difference being that the graphics code was contained within the Client modules). But it has also provided a guide to the IO/pure separation. The various parts of the game state, and the different ways in which it can be update (collectively referred to in the source as the *model*) can be entirely pure. The network layer will clearly be impure, and can communicate representations of various aspects of the (pure) model.[74]

- Game
  - Server
  - Client
- Model
  - Game state
  - Time
- Network

Figure 4.2: The top level names in the module hierarchy in early versions of Serenity.

Surprisingly, the graphics logic is also entirely pure. This is because the code only has to define 'what' to draw, in terms of an abstract PICTURE type. In the case of Serenity the actual impure drawing took place within the external library Gloss, but in projects where OpenGL (or similar) is used directly, it would still be wise to maintain this separation between a pure logic layer, and an impure 'work' layer. This is an important and widely used concept, and is the subject of Section 4.3.

Summary of this section Two important patterns have emerged during this discussion. Firstly there is the approach to setting out the top level breakdown of a large project: by considering units of functionality from the point of view of the main loop of the program and how these parts interface. This leads to both a decent module breakdown and a insight into the appropriate separation of pure and impure code.

[74] Bear in mind each of these headings does not need to — and probably shouldn't — represent one Haskell file. Instead they each represent a top level module (rather like a package in Java) that is imported as one module but can be made up of several smaller internal modules.

4.2 *Haskell Modules, Encapsulation, and Connascence*

Connected with separation of code and concerns are the subjects of *encapsulation* and *connascence*. Encapsulation (sometimes referred to as *information hiding*) is the maintenance of a public interface that allows for implementation changes to be made without breaking another component that imports it, whereas connascence is a more general term for when modifying one piece of code must lead to modifying another in order to maintain correctness. Two pieces of code are connascent when a change in one necessitates a change in the other.[75]

[75] M. Page-Jones. Comparing techniques by means of encapsulation and connascence. *Communications of the ACM*, 35(9):147–151, 1992

Compared with Java and other similar OO languages, more effort has to be made in Haskell to have good encapsulation between modules. No formal effort was made to address encapsulation issues at the beginning of the Serenity project, and this has led to some (small) problems in a few places. In response to these, the conventions discussed below were adopted.

Most of the issues with encapsulation in Haskell are due to user defined types using the **data** keyword. Exporting constructors directly is bad encapsulation, because changes to the structure of the type can lead to dependant implementations failing. For example consider the simple datatype in a module shown in Listing 3.

```
module MyApp.Test where

data Example = Example Int Int
```

Listing 3: An example of a badly encapsulated module interface.

This leads to strong connascence between this module and any other importing it, because if an additional field were to be added to the Example type, every other module using the Example constructor would no longer compile. It also exposes the internals of the type and is such is bad encapsulation — another module could come to rely on the way information is represented in the type and then break when changes to the implementation are made.

The way to manage this is to use the Haskell record syntax, and to only export the field functions, and an abstract constructor (or constructors). This approach is shown in Listing 4 below.

```
module MyApp.Test (example, exampleOne, exampleTwo) where

data Example = Example { exampleOne :: Int, exampleTwo :: Int }

example :: Int → Int → Example
example = Example
```

Listing 4: A well encapsulated module interface.

Now the exported functions — example, exampleOne, exampleTwo — are the only available public interface to the module. If a field is added, or the internals of the type representation change, these three functions can be maintained (even if they are no longer defined through the record syntax). The implementation details of the module are now hidden and other modules importing it do not become connascent to it.

There are various further improvements to this. Exporting lenses, rather than the basic deconstruction functions provided by the record syntax, is desirable.[76] Also there are language extensions such as GADTs (Generalised Algebraic Datatypes) that can take this kind of method further.[77]

Another desirable feature is to have a single point of entry for other modules to import a set of functionality. For this reason it is a desirable pattern to have a single module App.X for example, that re-exports the public interface of every module underneath it (App.X.One, App.X.Two etc). Another module App.Y should only import App.X and not the inner parts of its implementation. This way module App.X can take care of the overall public interface, whereas modules underneath it in the hierarchy can share some implementation details if needed.

As well as providing encapsulation and avoiding connascence, this pattern leads to cleanly separated interfaces and can avoid problems with circular imports.

Summary of this section Some more care must be taken with encapsulation in Haskell than in OO languages like Java. However, it can be achieved by sticking to a couple of simple rules: don't export constructors directly, and keep inner module implementations cleanly separated.

[76] Lenses are the functional equivalent of setters and getters and key value coding. See Section 4.5 below.

[77] For more information on GADTs see `http://en.wikibooks.org/wiki/Haskell/GADT` (accessed April 2013) or F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *ACM SIGPLAN Notices*, volume 41, pages 232–244. ACM, 2006.

4.3 *On the Separation of Interface from Implementation and Cause from Effect*

There has already been some discussion about separating pure code and code that must interact with the outside world in Section 4.1, and on reducing dependency between components in Section 4.2. Here we discuss more specific design patterns that can address these and similar issues, in the more general setting of separation between interface and implementation.

4.3.1 *Classes*

Of primary importance in Haskell coding is the concept of *classes*, distinct from classes in an OO context. In Haskell, a type is an instance of a class if a given set of functions are provided. For example, Listing 5 shows the class declaration for a *functor*.

```
class FUNCTOR f where
    fmap :: (a → b) → f a → f b
```

Listing 5: The FUNCTOR typeclass.

Here, f, a, and b are type variables; f representing the type belonging to the class, and a and b able to be any types. For example, if the type f was [] (i.e. the list type) then it is easy to see that fmap is the same type as the normal map, and a therefore possible FUNCTOR instance (which is actually already the default instance) for a list is as shown in Listing 6.

```
instance FUNCTOR [] where
    fmap = map
```

Listing 6: A FUNCTOR instance for lists.

The advantage that this gives is that it is now possible to write functions that know nothing about the type of their inputs other than that they are an instance of some class. For example, we could define a version of fmap that works on a pair of different functors, like so:

```
fmap₂ :: (FUNCTOR f₁, FUNCTOR f₂) ⇒
    (a → b) → (c → d) → (f₁ a, f₂ c) → (f₁ b, f₂ d)
fmap₂ g₁ g₂ (x,y) = (fmap g₁ x, fmap g₂ y)
```

Listing 7: Example of writing a function using only the knowledge that the argument is a FUNCTOR.

This function can now be used whenever FUNCTOR instances are available on both types in a pair. This is a contrived example, but hopefully illustrative of the advantages of this approach — designing a module so that it works on any instance of a given class, be it an inbuilt one like FUNCTOR, or a new class provided by the module, is an excellent way to avoid coupling and allow for code reuse.

This pattern is used in several places in the Serenity code, most notably to provide a clean interface between the implementation of the model and the code that updates state during the game. Listing 8 shows an extract from SERENITY.MODEL.TIME showing the classes used to provide this interface. There are three classes, UPDATABLE, COMMANDABLE, and EVOLVABLE, to represent to concepts of reacting to

updates from the server, commands received from the client, and to the passing of time. Class inheritance (similar to inheritance in OO) is used so that an instance of COMMANDABLE or EVOLVABLE must already be an instance of UPDATEABLE.

Listing 8: Classes from SERENITY.MODEL.TIME.

```
class UPDATEABLE a where
    update :: UPDATE → a → a
    updates :: [UPDATE] → a → a
    updates = flip (foldr update)

class (UPDATEABLE a) ⇒ COMMANDABLE a where
    command :: COMMAND → a → [UPDATE]
    commands :: [COMMAND] → a → [UPDATE]
    command _ _ = []
    commands cs a = concatMap (flip command a) cs

class (UPDATEABLE a) ⇒ EVOLVABLE a where
    evolve :: UPDATEWIRE (a, GAME)
    evolve = pure []
```

←The type UPDATEWIRE is an object that gives logic for providing updates given the passing of time.

After this, all the server and client logic uses only these interfaces. This means that any types providing instances for these classes could take advantage of the server-client code — this could be a completely different game! The proper usage of classes allows for a great deal of reusability.

4.3.2 *Separating Syntax and Semantics Part I: Interim Types*

A very fundamental design pattern is to distinguish as much as possible between the form of a representation and its semantics. A classic example of this is the organisation of compilers into a front and back end. The front end is responsible for building a representation of the input string, usually in the form of an abstract syntax tree (AST), and the back end is responsible for converting the AST into code for the target system. This modularity confers a number of advantages, the foremost being that n input languages and m target architectures require only $n + m$ implementations rather than nm.[78]

[78] D. Grune, K. van Reeuwijk, and H.E. Bal. *Modern compiler design*. Springer, 2012, page 6, section 1.1.1.

This technique can be found throughout the Haskell runtime and in many of the popular Haskell libraries. The IO monad that allows a running Haskell program to interact with the real world is a perfect example. During compilation, actions in the IO monad are simply descriptions of operations to be performed, that can only be semantically interpreted to yield actual values at runtime.[79]

[79] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993

A motivating example will help to demonstrate the efficacy and need for this design pattern. Consider an entity within the state of a game. At various points this entity will need to be updated in various ways to react to different things. For the purposes of the running example we shall model: time passing, new orders being given from the player, and damage being inflicted. (This is

clearly in the context of a strategy game similar to that made in the Serenity project).

First some basic types are defined, without specific implementation details for brevity, for an entity, a command, and an amount of damage.

```
data ENTITY = ENTITY {...}
data ORDER = ORDER {...}
data DAMAGE = DAMAGE {...}
```

Listing 9: Some basic types for the entity update example.

Now the three types of update are to be implemented. Although these updates must eventually be applied "in the real world" by an IO routine, their behaviour can still be defined by pure functions. So an initial implementation might look like this:

```
updateTime :: DOUBLE → ENTITY → ENTITY
updateTime time entity = ...

updateOrder :: ORDER → ENTITY → ENTITY
updateOrder order entity = ...

updateDamage :: DAMAGE → ENTITY → ENTITY
updateDamage damage entity = ...
```

Listing 10: A naive approach to entity semantics.

With these three functions implemented, a basic interface for updating an entity can be exported. But it is not very flexible. To see why, consider the code applying the updates to the game state. The updates can be applied as they come in, or buffered in some structure, which for simplicity can be assumed to be a list. What type is the list? Without any further structure it can only be of type [ENTITY → ENTITY] and each element is one of the update functions partially applied, i.e. [updateTime 4.3, updateDamage DAMAGE {...}, ...]. Now say the writer of this code wants to add some additional behaviour as the ENTITY gets updated, such as logging, or notifying some other part of the state. He can't, as there is no way to introspect what is contained in each update, so any changes must be made to the update functions themselves, including their type signatures.

The solution is to use an interim type to separate the two concerns. This type will stand for the concept of *some kind of update that could be applied to an Entity*.

```
data ENTITYUPDATE =
    UPDATETIME DOUBLE
  | UPDATEORDER ORDER
  | UPDATEDAMAGE DAMAGE
```

Listing 11: Entity update interim type.

This type provides a separation between the *form* of, and the actual *semantics* of, updating an entity. An instance of the type can be introspected (either via pattern matching or a proper provided interface — the latter isn't detailed here to keep it simple), and other code is free to interpret the type in any way without making changes to it.

A semantics can of course be provided alongside this type as a simple function without removing the benefits gained by the separation. Listing 12 illustrates such a function. An implementation can now introspect and store updates, call this function, and add any additional behaviour required.

```
update :: EntityUpdate → Entity → Entity
update (UpdateTime t) = ...
update (UpdateOrder o) = ...
update (UpdateDamage d) = ...
```

Listing 12: Providing a default semantics.

This example illustrates a somewhat trivial case, but the general pattern is extremely advantageous, especially when it is not obvious how to limit the use of impure code. For example, this implementation could be extended by adding an IO routine to be called on each update, without needing to change the existing pure functions. The Picture type from the Gloss graphics library is an excellent example of this pattern in a real application. Instead of insisting that the caller uses IO drawing routines directly, the Picture type forms a pure interface to describe what *should* be drawn.

4.3.3 *Modelling Effects with Monads and Monad Transformers*

In the discussion thus far the state of the game or program has not been considered in detail. At first sight one might assume that impure code must be responsible, at least at some level, for the update of this state, but this is not actually the case. It is possible to pass state entirely functionally, as the following small example of a stack machine demonstrates:

```
data Operation a = Push a | Pop | Sum | Diff | Mult | Dup

data State a = State {stack :: [a]}

eval :: Num a ⇒ Operation a → State a → State a
eval (Push a) (State s) = State (a:s)
eval Pop (State s) = State (tail s)
eval Sum (State (a:b:s)) = State (a+b:s)
eval Diff (State (a:b:s)) = State (a−b:s)
eval Mult (State (a:b:s)) = State (a∗b:s)
eval Dup (State (a:s)) = State (a:a:s)

run :: Num a ⇒ [Operation a] → State a → State a
run = flip $ foldl $ flip eval
```

Listing 13: Modelling a simple stack machine using pure code only.

But this approach has a number of weaknesses. Firstly, passing state has to be dealt with manually, both in the type signatures and in the implementations, and there is the difficulty of extending the implementation to deal with errors or other concerns.

An effective pattern for modelling objects that have this kind of imperative structure is to use a *monad*,[80] and indeed monads have

[80] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992

become quite ubiquitous in Haskell programming despite having a somewhat ill-founded reputation for being hard to understand.[81]

Existing familiarity with monads will not be assumed here, but the details are covered only very briefly.[82] Listing 14 shows the MONAD typeclass.

A monad *m* can be thought of as a computational environment, where objects or routines can be combined using ≫= (pronounced 'bind') and return provides a way 'into' the context. Notably here, the implementation of ≫= can thread a state through a sequence of computations, and cause this state to be updated in any number of ways at each stage, reacting to errors etc. For this reason a monadic value *m a* is sometimes referred to as a monadic *action*.

[81] S. Peyton Jones. Tackling the awkward squad. Technical report, Technical report, Microsoft Research, Cambridge, 2001

[82] There are numerous online resources containing explanations of monads. M. Lipovača. *Learn you a Haskell for great good!* No Starch Press, 2011 contains an excellent introduction, and can be read online at `learnyouahaskell.com/`.

```
class MONAD m where
    return :: a → m a
    (≫=) :: m a → (a → m b) → m b
```

Listing 14: The MONAD typeclass.

The standard Haskell distribution comes with a number of monads built in, including the STATE monad, which provides an environment that keeps track of a generic state. Inbuilt functions get, put, and modify are provided to put values into the state, get them out, and call a function over the contents of the state respectively. Listing 15 shows the state machine example re-written to use the STATE monad.

```
import CONTROL.MONAD.STATE

type STACKMACHINE a = STATE [a]

popM :: STACKMACHINE a a
popM = do
    (x:xs) ← get
    put xs
    return x

pushM :: a → STACKMACHINE a ()
pushM x = do
    stack ← get
    put (x:stack)

eval′ :: NUM a ⇒ OPERATION a → STACKMACHINE a ()
eval′ (PUSH a) = pushM a
eval′ POP = do popM; return ()
eval′ SUM = do a ← popM; b ← popM; pushM (a+b)
eval′ DIFF = do a ← popM; b ← popM; pushM (a−b)
eval′ MULT = do a ← popM; b ← popM; pushM (a∗b)
eval′ DUP = do a ← popM; pushM a; pushM a

run′ :: NUM a ⇒ [OPERATION a] → STACKMACHINE a ()
run′ = foldM (λ_ → eval′) ()
```

Listing 15: Modelling a simple stack machine using pure code only, this time using the STATE monad.

The function execState can be used to find the value of the state after a computation, so the expression

```
execState (run' [PUSH 10, PUSH 5, MULT]) []
```

will yield [50].

Note that no state is passed manually, and the implementation of eval′ is dependent only on the interface to the monad STACKMACHINE *a*. From now on, extra functionality can be added by changing the implementation of the monad, without effecting the implementation of eval′.

To further illustrate this point, consider the following. Currently this implementation will crash if operations are made when the stack does not have the required number of elements:

```
> execState (run' [Push 10, Mult]) []
*** Exception: Pattern match failure in do expression
```

To improve this, an error mechanism can be added to the monad by using a *monad transformer*. A monad transformer is a monad formed by 'wrapping' an existing monad, so that return and (≫=) perform both the original monad's implementation and some additional functionality provided by the transformer. Monadic actions can be performed on the inner monad by using a lifting function. All that needs to be changed is the type synonym STACKMACHINE to wrap a MAYBE value[83] in a state monad transformer (using the STATET constructor), and the implementation of popM, as shown in Listing 16 below.

[83] MAYBE is used here, which models the computation either succeeding with a value or failing, but does no error reporting. However, if error reporting is required, the EITHER monad or the ERROR monad can be used.

```
type STACKMACHINE a = STATET [a] MAYBE

import CONTROL.MONAD.STATE

popM :: STACKMACHINE a a
popM = do
    stack ← get
    case stack of
        (x:xs) → do put xs; return x
        _ → lift NOTHING
```

Listing 16: Adding error handling using MAYBE and the state monad transformer.

←The lift function is used to access the inner monad, here causing the computation to fail by passing NOTHING into the MAYBE monad.

Without any further changes, the following results are obtained:

```
> execStateT (run' [pushM 10, pushM 5, Mult]) []
Just [50]
> execStateT (run' [pushM 10, Mult]) []
Nothing
```

Extending this to include error reporting or different modes of failure is achievable just as easily.

It should be apparent that modelling effects using monads provides a good number of advantages, from separation of concerns to elegance of code. Writing new monads can be tricky, but the desired effects can usually be easily achieved by combining existing monad transformers. It is these advantages that make the monad such a widely used design pattern in Haskell code.

4.3.4 *Separating Syntax and Semantics Part II: Free Monads*

A further technique for improving the separation between interface and implementation is the usage of *free monads*, a relatively recent research area in the Haskell community.[84] The use of free monads as a design pattern was only discovered and experimented with after the main bulk of Project Serenity was already implemented, and there are many parts of the design that could benefit a great deal by being refactored to make use of it. The use of free monads and why they can be so useful is the subject of the rest of this section.

[84] In the examples below, the free monad implementations come from Edward Kemmet's package *free* (`hackage.haskell.org/package/free`), which also provides the functions liftF, retract, and hoistFree. Various other free monad implementations are also available on Hackage.

The precise definition of a *free* structure is a concept from Category Theory and out of the scope of any discussion here; but like monads, the technical details are not required to make use of the pattern as a programmer.

In simple terms, then, a free monad is a monad that can be generated directly from a functor "for free" — i.e. without any additional information being given or imposed. This is achieved by maintaining the structure of the operations (as a list or stream) without actually interpreting them. A monadic value in this free monad can then be interpreted, i.e. converted to a monadic value in another monad by imposing a certain semantics, at any time at a later date, and even in multiple ways at different times.

Consider the stack machine example. After run (or run′) has been called, the computation is complete and the components of it cannot be introspected in any way. So, while there is a compile time separation provided between the OPERATION type and the STACKMACHINE monad, at runtime a STACKMACHINE monadic value is essentially opaque.

To improve upon this situation, a free monad can be formed from the OPERATION type. A wrapper is used to form a functor, with an additional type variable that represents "the rest of the computation" or continuation (the functor instance can be automatically derived). From this functor the free monad is formed. The code for this is shown below in Listing 17.

Listing 17: Forming a free monad from the OPERATION type.

```
data FreeFunctor a cont = FreeFunctor a cont
    deriving (Functor, Show)

type AbstractMachine a = Free (FreeFunctor (Operation a))

liftFF :: a → Free (FreeFunctor a) ()
liftFF x = liftF $ FreeFunctor x ()

push a = liftFF $ Push a
pop = liftFF Pop
sum = liftFF Summ
diff = liftFF Diff
mult = liftFF Mult
dup = liftFF Dup

example :: AbstractMachine Int ()
example = do push 9; push 5; mult; dup; mult
```

```
interpret :: (Functor m, Monad m) ⇒ (∀x. f x → m x) → Free f a → m a
interpret f = retract . hoistFree f

using :: Monad m ⇒ (t → m a) → FreeFunctor t b → m b
using actionFor (FreeFunctor operation c) = do actionFor operation; return c
```

The interpret function can now be used to coerce a value of type AbstractMachine *a* (the free monad) into a full monadic type with a specific semantics. using is a shortcut to lift a function between operations and actions into the type required by interpret.

Three example semantics are shown in Listing 18, firstly the normal operation of StackMachine with failure as implemented before, secondly some basic logging as a writer monad transformer, and lastly both of these simultaneously.

Listing 18: Three example semantics for the AbstractMachine free monad, the stack machine operation, basic logging, and both simultaneously.

```
log :: (Show a, MonadWriter [Char] m) ⇒ Operation a → m ()
log (Push a) = tell $ "Pushing " ++ show a ++ "\n"
log Pop = tell "Popping"
log Summ = tell "Summing\n"
log Diff = tell "Subtracting\n"
log Mult = tell "Multiplying\n"
log Dup = tell "Duplicating\n"

asStackMachine :: Num a ⇒ FreeFunctor (Operation a) c → StackMachine a c
asStackMachine = using eval'

asLog :: (Monad m, Show a) ⇒ FreeFunctor (Operation a) c → WriterT String m c
asLog = using log

type StackMachineLog c = WriterT String (StackMachine a) c
asStackMachineWithLog :: (Show a, Num a) ⇒ FreeFunctor (Operation a) c → StackMachineLog c
asStackMachineWithLog ff = do lift $ asStackMachine ff; asLog ff
```

These results can now be obtained in GHCI:

```
> execStateT (interpret asStackMachine test) []
Just [2025]

> putStr $ runIdentity $ execWriterT (interpret asLog test)
Pushing 9
Pushing 5
Multiplying
Duplicating
Multiplying

> fmap snd $ runStateT (runWriterT (interpret asStackMachineWithLog test)) []
Just [2025]

> fmap (snd.fst) $ runStateT (runWriterT (interpret asStackMachineWithLog test)) []
Just "Pushing 9\nPushing 5\nMultiplying\nDuplicating\nMultiplying\nPushing 9\nSubtracting\n"
```

It is clear that this approach is a very effective and powerful one, and can be expanded to allow for easy combination of different layers of concern, each with their own semantic interpretation, loosely coupled and swappable. There are several packages that build on this idea and make working with it easy, the most notable being *operational*,[85] a package for building custom monads, and *pipes*,[86] which models separate interpretation layers as a pipeline, allowing for filtering at each stage, and passing information in both directions. These kind of design patterns are invaluable to complex systems like games, that require multiple states to be maintained at the same time as networking, journalling, and various other activities; and their use is highly recommended in any similar domain.

[85] Heinrich Apfelmus 2011, `hackage.haskell.org/package/operational`

[86] Gabriel Gonzalez 2013 `hackage.haskell.org/package/pipes`

4.3.5 *Summary of this section*

High level abstractions enabled by features of the Haskell language and type system can enable an extremely flexible coding style that maintains clear separation between concerns, and implementation from interface. The design patterns identified in this section give methods toward achieving these ends in a reliable way.

Typeclasses are a basic language feature of Haskell, and provide a simple yet highly effective was to write reusable code components. A very powerful idiom for flexible code is to use interim types to separate concerns. Monads provide a very flexible approach to modelling effects, and this can be improved even further with the use of free monads.

4.4 Graphics Programming in Haskell

Any game is clearly going to involve graphics in some capacity or another, but graphics are not something that at first sight seem suited to a functional language. There are, however, various graphical frameworks and bindings available for Haskell.

For the purposes of the Serenity project, the choice basically boiled down to three options. Firstly, there are bindings directly to OpenGL available; which would provide the most flexibility but also likely the most development time. Secondly there are the bindings to the SDL engine, along with all the various tools provided with its framework. Many of the existing games written in Haskell use SDL. Lastly there is the use of the simple but effective layer over GLUT and OpenGL provided by the Gloss library.

This was not an easy decision, and some time went into making it. The direction taken in the Serenity project was to use the Gloss library, mostly because of its simple interface and ease of use, given the limited time and large scope of the rest of the project. The advantages of Gloss can be appreciated by considering the two screens in Figure 4.3. Both of these examples involve relatively simple code, which is almost entirely pure.

The decision to use Gloss has largely been held up, but there has been some problems due to features it lacks, the most notable being clipping. In the future it would probably be beneficial to replace Gloss with an in house framework providing a layer between the pure graphics code and impure bindings to OpenGL.

Some details of the OpenGL and Gloss approaches are given below to illustrate the differences and tradeoffs involved.

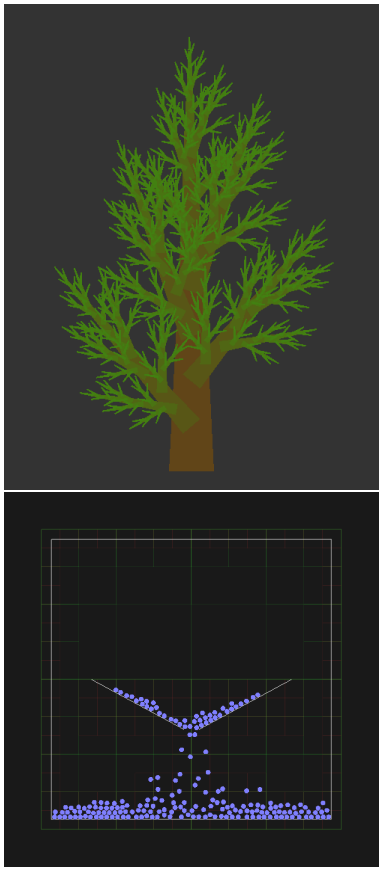


Figure 4.3: Gloss example screens from `gloss.ouroborus.net/`.

4.4.1 Using OpenGL Directly

Writing OpenGL code in Haskell is in many ways similar to writing it in C++ or any similar language. OpenGL calls are simply functions in the IO monad, (i.e. functions of type IO *a*), and there are some special primitive types such as GLFLOAT.

Listing 19 shows a very simple example of drawing an empty circle and a filled circle, and the output is shown in Figure 4.6. Normal Haskell style is used to create a circle: using the map function and some trigonometry. These are then converted into OpenGL actions and rendered in the display callback.

NB — The OpenGL code in this section is based on the tutorial at `haskell.org/haskellwiki/OpenGLTutorial1` (retrieved April 2013).

```
import GRAPHICS.RENDERING.OPENGL
import GRAPHICS.UI.GLUT

myPoints :: GLFLOAT → [(GLFLOAT,GLFLOAT,GLFLOAT)]
myPoints r = map (λk → (r∗sin(2∗π∗k/n),r∗cos(2∗π∗k/n),0.0)) [1..n]
    where n = 100

pointToVertex :: VERTEXCOMPONENT a ⇒ (a, a, a) → IO ()
pointToVertex (x,y,z) = vertex $ VERTEX3 x y z
```

Listing 19: Simple OpenGL example, drawing an empty circle and a filled circle (output shown in Figure 4.6).

```
renderMyPoints :: GLfloat → IO ()
renderMyPoints r = mapM_ pointToVertex (myPoints r)

main = do
    (progname, _) ← getArgsAndInitialize
    createWindow "OpenGL Example"
    displayCallback $= display
    mainLoop

display = do
    clear [ColorBuffer]
    renderPrimitive LineLoop (renderMyPoints 1)
    renderPrimitive TriangleFan (renderMyPoints 0.5)
    flush
```

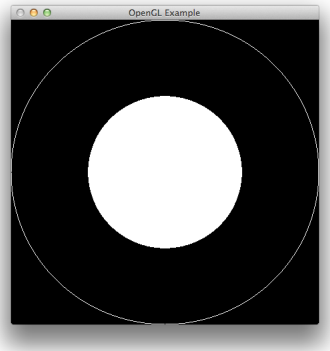


Figure 4.4: Output of example OpenGL code in Listing 19.

The challenge of programming graphics this way is far less an issue of language paradigm than it is of coding style: maintaining a proper separation between the concerns of basic rendering, specific entity models, the game logic, and so on, is the where main engineering effort is required — and this is no different in Haskell than in other languages.

As discussed in the previous section, there are various ways that such separation can be achieved, and that primary among these is the concept of an interim type. It is exactly this approach that is taken by the Gloss library, and this is the main reason Gloss was used in Project Serenity, to avoid the additional time it would take to build the infrastructure to work effectively with OpenGL.

4.4.2 *Using Gloss*

Gloss is a layer that sits on top of OpenGL and GLUT providing a much simpler and cleaner API for drawing vector graphics. The Gloss project website claims that "Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions", and that using Gloss allows you to "get something cool on the screen in under 10 minutes".[87] The simplicity of using Gloss compared to raw OpenGL code is shown in Listing 20 which recreates the simple OpenGL example in Gloss (although the colour of the circles has been changed to make the difference in outputs obvious).

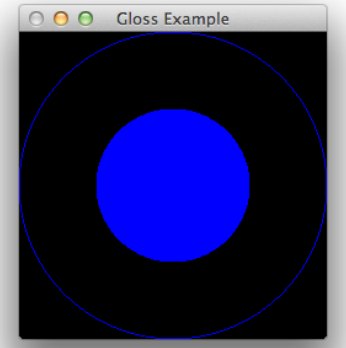


Figure 4.5: Output of example Gloss code in Listing 20.

[87] See `hackage.haskell.org/package/gloss`

```
import Graphics.Gloss

main = display (InWindow "Gloss Example" (250, 250) (0, 0)) black circles

circles = Pictures $ map (color blue) [circleSolid 125, circle 250]
```

Listing 20: Simple Gloss example, drawing an empty circle and a filled circle (output shown in Figure 4.5).

This example simply creates a window named "Gloss Example" with a black background and adds the specified Picture to it. A Picture is the Gloss abstraction of the OpenGL primitives. In the example the picture to display is defined by the circles function.

The example in Listing 20 uses the display mode to draw a static picture to the screen. The game mode, started with play or playIO, is obviously more useful when developing a game. It keeps track of a game world, the current state of the game, for which the developer provides callbacks for updating the world every tick and for converting the world to a PICTURE. It also allows the programmer to add callbacks for handling input events such as mouse movement and clicks. The Gloss documentation and `gloss-examples` package are great sources of further information on getting started with game development with Gloss.[88]

[88] See `hackage.haskell.org/package/gloss` and `hackage.haskell.org/package/gloss-examples`

4.4.3 *Summary of this section*

Given the limited time available for the Serenity project, the prebuilt pure interface onto OpenGL provided by the Gloss library was the best option, and this has been born out by the results. A custom made layer that is more appropriate for the needs of a complex game, and more readily adapted to changing requirements, would be beneficial to develop in the future.

4.5 *Implementing a Graphical User Interface (GUI) Framework, and the Power of Lenses*

Figure 4.6: Part of the fleet builder UI built using Sheen.

Gloss provided an interim layer to abstract away the complexities of OpenGL and impure drawing code, but as it turned out further abstractions on top of this were desirable. Gloss provides no "widgets" (buttons, scroll bars, etc), nor does it facilitate separable concerns for event handling.

Some time was spent in developing a framework on top of Gloss to enable easier construction of graphical interfaces, which has been glibly titled *Sheen*. Provided by Sheen is a mechanism for event handling, recursively nested views, and various pre-built widgets such as labels, buttons, and text boxes. The best approach to the design of Sheen was not obvious, and its structure, and use of *lenses* (see below) is considered to be quite novel and a demonstration of good Haskell style. This section considers various aspects of the Sheen design, but first it is worth examining lenses as significant use of them are made in the framework.

4.5.1 *On Lenses*

A lens is the functional equivalent of a setter and a getter at the same time. One way of thinking of a lens is it having the type

```
data Lens a b = {get :: a → b, set :: b → a → a}
```

Normally lenses are not actually implemented this way, but in such a way to be isomorphic to this. The most popular implementation of lenses is currently in the *lens* package by Edward Kmett,[89] which uses a generalised form of a so called van Laarhoven lenses. A van Laarhoven lens has type

```
type SimpleLens a b = ∀f. Functor f ⇒ (b → f b) → a → f a
```

and the generalised notion of a lens family has type

```
type Lens a b c d = ∀f. Functor f ⇒ (c → f d) → a → f b
```

The advantage of this definition of lenses is that they can be composed normally as functions using (.) and id from the Haskell prelude. This, and the large number of combinators provided in the *lens* package, is partly what makes lenses so vastly useful. For a full technical explanation of this implementation of lenses see `comonad.com/reader/2012/mirrored-lenses`; here the way they are used is all that need be considered.

[89] Edward A. Kmett, Copyright 2012-2013 `https://github.com/ekmett/lens/`.

There are two ways of creating lenses. First is to use the lens function to build the lens from a getting and setting function directly, for example

```
sndLens = lens snd (λ(a,_) b → (a,b))
```

This leads to a lot of boilerplate however. The alternative is to use the Template Haskell routines provided in the library, makeLenses or makeClassy. These do require Template Haskell (and hence GHC) but are very useful indeed. In the expression

```
data FOO a = FOO
    { _bar :: a
    , _baz :: [a]
    }
makeLenses ''FOO
```

the last line will create two addition functions

```
bar :: FUNCTOR f ⇒ (a → f a) → FOO a → f (FOO a)
baz :: FUNCTOR f ⇒ ([a] → f [a]) → FOO a → f (FOO a)
```

Or the same thing expressed in the LENS type synonym:

```
bar :: SIMPLE LENS (FOO a) a
baz :: SIMPLE LENS (FOO a) [a]
```

This demonstrates how easy it is create lenses with Template Haskell, but why is it worth doing? Lenses are invaluable for providing good interfaces between modules and hence separation of concerns. For example, say one module contains the overall state of an application, and another module is responsible for logic based on only a part of that state. By making the functions in this latter module depend, rather than the specific data type that contains the application state, but rather any type *a* and lenses from *a* onto the parts of the data that module is responsible for, the modules remain nicely separated. This technique is leveraged extensively when using Sheen.

4.5.2 *A Recursive View Hierarchy*

In the prototype projects written with Gloss, widgets were modelled quite simply as having an area of the screen that would trigger actions when an event fell within the bounds. This was unsatisfactory for a number of reasons. First, and especially as Gloss has no support for clipping,[90] was that care had to be taken to ensure that the picture of the widget ended up in the same place as its target area. And also there is the problem that as the layouts become more complicated, connascence between all the different widgets builds up. If one wanted to move an entire sidebar, for example, everything within it would require its position updating. Hardly ideal.

[90] Clipping is masking out parts of an image so that only one area of the screen can be drawn to.

Sheen was first created to solve this sole issue, by supporting the concept of a recursive structure. The definition of VIEW in the latest sheen code is

```
data VIEW a = VIEW
    { _viewSubviews :: [VIEW a]
    , _viewFrame :: EXTENT
    , _viewZIndex :: INT
    , _viewBackground :: MAYBE COLOR
    , _viewDepict :: MAYBE PICTURE
    , _viewEventHandler :: MAYBE (UIEVENT → a)
    }
```

Listing 21: Definition of a VIEW in Sheen. Some fields non-essential to the discussion are not shown.

Note that the subviews field makes this a recursive structure. The (optional) event handler field is a function from an event to a new overall state. The VIEWCONTROLLER class which is the required top level of the hierarchy, provides the function that actually lays out the view: getView :: VIEW *a*. This is why the event handler only returns an *a* and not an $a \to a$. Also in order to support concepts like focus of text boxes and other stateful behaviour, the view system needs somewhere to store state. The VIEWCONTROLLER class is shown in Listing 22.

Listing 22: VIEWCONTROLLER class.

```
class VIEWCONTROLLER a where
    globals :: SIMPLE LENS a (VIEWGLOBALS a)
    getView :: a → VIEW a
    updateTime :: FLOAT → a → a
    updateTime _ = id
```

Note that because the VIEWGLOBALS are provided by a lens, they can be both easily viewed or modified.

Sheen provides all the logic that handles clicks and key presses and calls the handler at the correct view based on where the click was and what view is currently in focus. In order to illustrate how useful the lens library functions for state monads are, a snippet from this code is shown in Listing 23.

Listing 23: Snippet from Sheen core logic.

```
handleViewEvents
    :: VIEWCONTROLLER a ⇒ ((FLOAT, FLOAT), UIEVENTS) → a → a
handleViewEvents (point, events) = execState $ do
    a ← get
    indexPathUnderMouse ← return $ indexPathAtPoint point (getView a)
    indexPathLastMouseDown ← use (globals.globalMouseDown)
    indexPathFocus ← use (globals.globalFocus)
    when (events^.uieventsUpdateFocus) $ updateFocus indexPathUnderMouse
    when (events^.uieventsUpdateMouseOver) $ updateMouseOver indexPathUnderMouse
    forM (events^.uieventsEventUnderMouse) $ sendEvent indexPathUnderMouse
    forM (events^.uieventsEventToFocus) $ sendEvent indexPathFocus
    when (indexPathUnderMouse ≠ indexPathLastMouseDown) $ do
        forM (events^.uieventsEventLastMouseDown) $ sendEvent indexPathLastMouseDown; return ()
    when (events^.uieventsUpdateMouseDown) $ globals.globalMouseDown .= indexPathUnderMouse
```

It is interesting just how much like imperative code this is, and yet this is a pure Haskell function. execState is used to run the computation in a state monad, which provides the when combinator that acts like *if* from C like languages. forM is simply mapM with the arguments reversed, and runs an effect for each member of a list, and so is like a *for-each* loop. (.^) is an infix synonym of view and used to get values from lenses; this can also be done as a state monad action with the use function. Finally, .= is a state action that performs assignment using a lens.

4.5.3 Widgets

Sheen also includes many widgets which are built using the basic view system. These can then be recursively used in larger view hierarchies. Widgets included in Sheen at the time of writing are a text label, a picture view, a button, a text box, and a table. All of these are built up from the views and sometimes using other widgets. To illustrate this Figure 24 shows the code for the table widget.

Listing 24: Table widget implementation.

```
data TABLE a = TABLE
    { _tableCellSep :: INT
    , _tableBackground :: MAYBE COLOR
    }
makeLenses ''TABLE

initTable :: INT → MAYBE COLOR → TABLE a
initTable sep backg = TABLE
    { _tableCellSep = sep
    , _tableBackground = backg
    }

table :: a → SIMPLE LENS a (TABLE a) → GETTER a [b] → (b → VIEW a) → ((INT, INT), (INT, INT)) → VIEW a
table a table bsLens bView2 bounds =
    (initView bounds) & (viewBackground .~ (a^.table.tableBackground))
    <++ map (λ(b, i) → (viewOrigin._2 .~ top − sep∗i) $ bView2 b) ((a^.bsLens) `zip` [1..])
    where
        sep = a^.table.tableCellSep
        top = (bounds^._1._2) + (bounds^._2._2)
```

Note how all of the Sheen views and widgets assume nothing about the type that they are drawing, all that is needed is a VIEWCONTROLLER instance at the top of the hierarchy, and lenses to pass to the various functions.

4.5.4 Use in the Main Project

Sheen is used to build all of the UI in Project Serenity, from the menus to the in-game sidebar. Each screen is a separate Haskell module, and provides a class specifying the lenses needed to draw the views for that screen. There is a main module that imports all of the different screens, builds all the required instances and calls the view function based on what mode the application is currently in.

An interesting observation of coding with Sheen is the difference between the functional and imperative styles of the interface. Most of the widgets require a space to store some internal state (for example the TABLE type for the table widget in Listing 24). A screen can therefore be animated in an "imperative style" by updating this state in reaction to events. But animation can also be achieved in the building of the view hierarchy itself. Views can be drawn

based on the interface to the underlaying state, and so the animation logic is in a "functional style". The meeting point between pure and impure, side effect and logic, is very close together when doing UI programming, and Sheen represents an approach that allows a much more functional and pure approach than most GUI systems available for Haskell.

4.5.5 Summary of this Section

The Sheen framework designed during the implementation of Project Serenity is a significant step towards an advanced event driven GUI system. Although it currently depends on Gloss this could be replaced with a different back end in the future. Sheen makes significant use of classes, lenses, state monads, and represents a full example of how powerful some of the methods in this chapter can be.

4.6 Network Programming in Haskell

Fast paced multiplayer games require efficient networking and real time packet delivery. If this is not provided then the network can become a bottleneck causing the game to lag or halt. This requirement means that the transmission control protocol (TCP) cannot be used for game network development because of the implementation of reliability in TCP. If a packet is lost when using TCP then the receiver stops and waits for that data to be resent and any new data that is sent is held in a queue until the lost packet arrives. Therefore, any packet loss on the network causes relatively large pauses in communication which will cause objects in the game world to stop receiving updates and the game hangs too.

So, networked multiplayer games that rely on real time network communications must use the user datagram protocol (UDP) since it does not enforce a stop-and-wait style reliability system. However, this means that a layer on top of UDP must also be used to implement an efficient form of reliability, deal with duplicate packets and out of order packets, and create virtual connections. Unfortunately, whilst Haskell provides a low level networking library, no suitable library for game networking could be found, so one had to be developed.

4.6.1 Sending and Receiving Packets

The `network` package is a low level networking interface that can be used to send and receive packets using UDP.[91] It provides an easy to use API for creating and sending data over sockets.

[91] `http://hackage.haskell.org/package/network`

Listing 25: Example of receiving UDP packets

```
import NETWORK.SOCKET

port = 9900

main :: IO ()
main = withSocketsDo $ do
    sock ← socket AF_INET DATAGRAM defaultProtocol
    bind sock (SOCKADDRINET port iNADDR_ANY)
    socketPrint sock

socketPrint :: SOCKET → IO ()
socketPrint sock = do
    (msg, _, _) ← recvFrom sock 512
    putStrLn msg
    socketPrint sock
```

Listing 25 shows a simple UDP server which receives packets sent to port 9900 and prints the data that was received. First, this code initialises the networking subsystem using `withSocketsDo`. This is only necessary on machines running the Windows operating system, but it is best practice to include this for portability. Then a UDP socket is created: `AF_INET` indicates the use of IPv4, the `DATAGRAM` socket type sets UDP, and a default protocol number

is set. The socket is bound to the specified listening port so that the operating system knows to forward incoming packets on port 9900 to this socket. The socket is then passed to a loop which reads incoming data with recvFrom and then prints it.

Listing 26: Example of sending UDP packets

```
import NETWORK.SOCKET

port = 9900

main :: IO ()
main = withSocketsDo $ do
    sock ← socket AF_INET DATAGRAM defaultProtocol
    addr ← inet_addr "127.0.0.1"
    sendTo sock "Hello world!" (SOCKADDRINET port addr)
    close sock
```

Sending a packet, as shown in Listing 26, is just as simple. This is a very basic UDP client that creates a socket and sends a string to port 9900 on the local machine. inet_addr is used to convert a STRING into a network address in host byte order. This address can then be used as part of an argument that specifies the destination for the sendTo function. Finally the program cleans up after itself by closing the socket.

These two examples show how simple it is to develop basic networking functionality in Haskell. However, UDP alone, as mentioned previously, is not robust enough to be used for games on real world networks which experience packet loss and packets arriving out of sequence. So, the next step is to develop a virtual connection over UDP and build a layer of reliability on top of that.

4.6.2 *Packets, Connections and Reliability*

Firstly, a simple PACKET data type is required to represent game packets to be sent out on the network. This is useful as it allows us to easily add headers to the packet and to pass around data that is simpler to manipulate and only convert it to a single set of bytes at the last minute.

Listing 27: Simple packet representation

```
import DATA.WORD (WORD32)
import DATA.BYTESTRING.CHAR8 (BYTESTRING)

data PACKET = PACKET
    { packetProtocol :: WORD32
    , packetData :: BYTESTRING
    }
```

The first version of the PACKET representation is show in Listing 27. This packet only contains a 32 bit protocol identifier and the actual packet data as a string of bytes. The use of a protocol identifier allows the networking code to ignore any incoming packets that do not have a matching identifier. This is necessary because client addresses are not known in advance, so another piece of information is required to distinguish legitimate packets from any other traffic.

Now these packets must be used, or extended further, to initialise connections. One approach would be to just start sending packets and assume a connection. When a listening server receives a valid packet it can recognise it as the establishment of a connection. However, with multiple computers this might not be robust enough. Therefore, for Project Serenity a brief handshake sequence, similar to the one employed by TCP, was used. Partially copying the TCP connection establishment handshake requires adding a flag bit field to the packet header. Each bit represents a flag which can be on, bit set to one, or off, bit set to zero. Flags can then be specified to identify connection initialisation.

Listing 28: Reliable packet data structure

```
import DATA.WORD (WORD32)
import DATA.BYTESTRING.CHAR8 (BYTESTRING)

data PACKET = PACKET
    { packetProtocol :: WORD32
    , packetSeq :: WORD32
    , packetAck :: WORD32
    , packetAckBits :: WORD32
    , packetFlags :: WORD8
    , packetData :: BYTESTRING
    }
```

The problem with TCP was that it forces packets to be delivered in an ordered stream. Instead of this games require packets to be delivered at a steady rate, but detect any loss and make a decision at the application level as to whether or not the packet should be resent. The first step to reliability is knowing which packets were received at the destination; this requires a method of identifying individual packets. Just like the establishment handshake, this can be stolen from TCP: each packet contains a sequence number. As packets are sent the sequence number is incremented monotonically. Listing 28 shows the packetSeq field in the PACKET data structure that holds the sequence number. The receiving end can now identify which packets it has received, but it still needs a method of relaying this information back to the sender.

Letting the other end know about packets received is done using acks, short for acknowledgements. The idea is to record the sequence numbers of incoming packets as the remote sequence number for the connection. A list of recently received packets is also kept. Then when a packet is sent the packetAck is set to the remote sequence number, i.e. the sequence number of the most recently received packet, and the packetAckBits bitfield is set so as to identify which of the 32 packets prior have also been received. Bit n in the packetAckBits bitfield is set to one if the nth most recent packet has been received. Using this method allows 33 acknowledgements per packet.

When a packet is received its ack and ack bitfield can be checked to detect any lost packets. If an ack for a packet is not received

within a certain timeframe then it can be deemed to be lost. This timeframe is dependent on the rate of network traffic, for example if approximately 30 packets are sent per second then an acknowledgement is expected within one second. Notifications of packet loss can be passed up to the application layer where a decision can be made on whether or not to bother resending the packet. This suggests that the packet data should include application level identifiers to ensure that any resent packets can be detected as duplicates if necessary.

4.6.3 Networking threads

Now a fully fledged networking library is available it needs to be put to use. How should it be used without impacting the performance of the game animations. One option would be to ensure that all networking options are non-blocking so that the networking code can be called during the main game loop. However, for Project Serenity it was decided to use Haskell's powerful and easy to use support for parallel threads.

The idea was to run two threads: one for incoming messages and the other for outgoing ones. To send a message it would be put into a shared queue by the main loop. The outgoing thread, consuming this queue, would read in the message and write it out onto the network. Similarly, as a message arrives the incoming thread would put it into a separate queue for the main loop to read from. Writing safe producers and consumers that operate like this is notoriously difficult to do correctly. Fortunately, however, the software transactional memory library, `stm`, makes this very simple to do in Haskell.

Software transactional memory is a technique for performing groups of memory operation atomically.[92] The idea is similar to that of transactions in databases. A block of code that is called *atomically* is guaranteed to be isolated from the memory operations of any other thread. Upon completion of the atomic transaction its transaction log is validated before the changes are committed to memory. If validation fails, e.g. because memory read was altered by another thread during execution, then the changes are discarded and the block is restarted. The use of transactions removes the complexity of traditional locks and allows concurrent code to be written with ease.

[92] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM

The `stm` library provides a transactional first-in-first-out channel, TCHAN, that is perfect for this situation. One thread can write values into the channel whilst another reads them out in the order they were put in. When a connection is established a TRANSPORTINTERFACE is created. This contains two TCHANs that are unique for the connection, one is the inbox and the other the outbox. The interface is placed into a map that maps from IP addresses to these channels. Using the connection map the networking workflow is as follows:

1. Create a socket and bind it to a known port.

2. Accept incoming requests placing the new connections into the connection map.

3. Spawn two threads which run indefinitely. One deals with the inbox channels; it receives incoming messages from the socket and puts them into the inbox TCHAN created for that IP address. The other reads in messages from all of the outbox channels and relays them onto the network.

Listing 29: Spawning threads to deal with network traffic. This is simplified version of the code in Project Serenity.

```
type CONNECTIONMAP = MAP SOCKADDR TRANSPORTINTERFACE

data TRANSPORTINTERFACE = TRANSPORTINTERFACE
    { channelInbox :: TCHAN MESSAGE
    , channelOutbox :: TCHAN MESSAGE
    }

sendAndReceive :: CONNECTIONMAP → SOCKET → IO ()
sendAndReceive connections sock = do
    forkIO $ forever $ inboxLoop connections sock
    forkIO $ forever $ outboxLoop connections sock
    return ()
    where
        inboxLoop connections sock = do
            (message, channels) ← receive connections sock
            atomically $ writeTChan (channelInbox channels) message

        outboxLoop transport = mapM_ (readAndSend socket) (M.toList connections)

        readAndSend sock (addr, channels) = do
            message ← atomically $ tryReadTChan (channelOutbox channels)
            case message of
                JUST m → send sock m addr
                NOTHING → return ()
```

Listing 29 shows a simplified version of the code that performs step three of this workflow in Project Serenity. Two infinite loops are spawned on two new threads using forkIO to spark a new lightweight thread and forever to repeat monadic actions indefinitely. inboxLoop calls a function that reads an incoming packet and returns the message it contained with the TRANSPORTINTERFACE associated with the sending address. The message is then atomically written into the inbox channel using writeTChan. The outboxLoop does the reverse by attempting to read messages from all of the outbox channels — this is done with the non-blocking tryReadTChan — and firing off them off onto the network. This is a great example of how easy it can be write concurrent code using Haskell. It only takes one line of code to fork a new thread and a couple more to write data into shared memory free from deadlock or corruption.

Summary of this section Haskell and its libraries have amazing support for networking and concurrency. forkIO is makes it very easy to create concurrent programs and the incredible power of the

`stm` library makes it manageable to do so too. However, creating a networking framework suitable for a multiplayer game was still a huge undertaking and it is unfortunate that the library created for Project Serenity is not yet suitable for release and use in other games.

4.7 *Effective Unit Testing with Quickcheck and HUnit*

The HUnit library is the Haskell implementation of the standard xUnit unit testing framework. The basic idea of the HUnit library is to provide the functions under test with some example data and to compare the actual result with the expected result.

```
import TEST.HUNIT

testEmpty = TESTCASE $ assertEqual "Empty list is zero" (len []) 0
testTwo = TESTCASE $ assertEqual "List length two" (len [0, 1]) 2
tests = TESTLIST [testEmpty, testTwo]
```

Listing 30: Example usage of HUnit. Testing a function that returns the length of a list.

An example of a small set of tests is shown in Listing 30. This example shows two simple tests for a function, len, that returns the length of a given list. A test, or list of tests, can be run with the use of the runTestTT function:

```
ghci> runTestTT tests
Cases: 2  Tried: 2  Errors: 0  Failures: 0
```

Property based testing is a higher level approach to testing in which the programmer develops a specification for the code to be testing. QuickCheck is a type-based property testing library that generates test cases automatically from the developer defined expected properties.[93] These properties need to be true for all inputs to the function (i.e. invariants). This is an extremely powerful approach to testing that allows the developer to write short testable specifications that are used to verify the code with thousands of test cases which would be infeasible to write by hand.

[93] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279, 2000

```
import TEST.QUICKCHECK

prop_NonNegativity x = absolute x ≥ 0
prop_Multiplicativeness x y = absolute (x ∗ y) == (absolute x) ∗ (absolute y)
prop_Subadditivity x y = absolute (x + y) ≤ (absolute x) + (absolute y)
prop_Idempotence x = absolute (absolute x) == absolute x
prop_Symmetry x = absolute (−x) == absolute x
```

Listing 31: Example usage of QuickCheck. Properties of a function that returns the absolute value of a number.

Listing 31 shows an example use case of QuickCheck to define five properties of a function, absolute, that returns the absolute value of a given number. The tests can then be run by invoking the quickCheck function on one of the properties, for example:

```
ghci> quickCheck (prop_NonNegativity :: Integer -> Bool)
+++ OK, passed 100 tests.
```

This means that for one hundred randomly generated test cases the property held. It is possible to get QuickCheck to run a different number of tests by using the quickCheckWith function and specifying a different number for the maxSuccess argument. The actual test cases that were generated can be viewed by using verboseCheck

instead of quickCheck — be warned that, as the name implies, this is very noisy!

Testing is another area where the separation of pure and impure code becomes very useful. By using the techniques laid out in the previous sections it is possible to have the majority of game logic in pure functional code. This is of great benefit when it comes to testing because, as Claessen and Hughes state, "functional programs are well suited to automatic testing".[94] Referentially transparent functions are much easier to test than those that produce side-effects because program state before and after execution is a nonissue.

[94] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279, 2000

During the development of the Serenity project an effective testing method combining the use of QuickCheck and HUnit was found. QuickCheck was used to generate large numbers of test cases for individual pure functions, and HUnit for impure functions, such as network code, and units of code comprised of several functions used together. This approach allows using the powerful property based testing where applicable in conjunction with more specific HUnit test cases to maximise test coverage and confidence that the expected results are being produced.

4.7.1 Organising an automated test suite

When creating a test suite for your software project it is useful to organise it in such a way as to allow quick automated testing, i.e. being able to run the entire test suite with a single command. The easiest way to do this in a Haskell project is to use the `test-framework` library. This library enables the user to group lists of tests into cohesive units and then run a list of tests and test groups with a single function call.

Listing 32: Running tests with test-framework

```
import TEST.FRAMEWORK

main = defaultMain allTests

allTests =
    [ PACKET.tests
    , TRANSPORT.tests
    , MESSAGE.tests
      — ...
    , MATHUTIL.tests
    ]
```

Listing 32 is an extract from the main entry point to the Serenity test suite. It creates a list of tests from the test groups exported by individual test modules. These tests are run by the defaultMain function provided by TEST.FRAMEWORK.

Grouping HUnit and QuickCheck tests with `test-framework` requires the use of two further libraries, `test-framework-hunit` and `test-framework-quickcheck2`. These two libraries provide functions to convert HUnit assertions and QuickCheck properties

into the TEST type used by `test-framework`. A brief example of their use in Serenity to build a group of tests to export is shown in Listing 33.

```
import TEST.FRAMEWORK (testGroup)
import TEST.FRAMEWORK.HUNIT (testCase)
import TEST.FRAMEWORK.QUICKCHECK2 (testProperty)

tests = testGroup "Network utility tests"
    [ testCase "Test readNTChan" testReadNTChan
    , testCase "Test readTChanUntilEmpty" testReadTChanUntilEmpty
    , testProperty "Empty TChan" prop_EmptyTChan
    ]
```

Listing 33: Grouping tests with test-framework

A test suite organised with the `test-framework` module can be easily integrated into the Cabal build system.[95] For example, here is the additional configuration added to the `Serenity.cabal` file to enable a Cabal backed test suite:

[95] See `http://www.haskell.org/cabal/users-guide/developing-packages.html#test-suites`

```
Test-Suite test-serenity
  type: exitcode-stdio-1.0
  hs-source-dirs: tests, src
  main-is: Main.hs
  build-depends:
    -- ... list of dependencies
```

The cabal-install program will look in `Main.hs` to find the main function which runs the entire suite of tests. With this test suite definition in place only a few commands a required to configure the project with the tests enabled, build it, and run the entire suite:

```
$ cabal configure --enable-tests
$ cabal build
$ cabal test
```

This set up allows easy use of a build server to run the test suite whenever a change is pushed to a master version control repository. For the Serenity project an instance of the Jenkins continuous integration server was set up for this purpose.[96] Jenkins allows the user to specify a shell script to execute when building the project. During the development of Serenity the following script was used to automatically run the test suite:

[96] See Section 5.8

```
set -e
cd '$WORKSPACE/Serenity/'
cabal update
cabal install --only-dependencies --avoid-reinstalls --
 enable-tests
cabal clean
cabal configure --enable-tests
cabal build
cabal test
```

Listing 34: Test suite script for Serenity.

Using this script the build server installs any new dependencies and then runs the entire test suite. Using a build server to run the test suite on every change is a good method of running the tests regularly to help find any errors early in the development cycle.

4.7.2 Test driven development

Test driven development (TDD) is a practice in software development that promotes testing by writing tests before implementing the functionality that it tests. The TDD cycle proposed by Kent Beck has five steps:[97]

[97] K. Beck. *Test-Driven Development: By Example*. Addison Wesley Professional, 2003

1. Add a test that defines the new functionality. By writing the test before starting on the implementation the developer is forced to clearly understand the requirements of the new feature and think about some design aspects before rushing into coding, such as the API of a new function.
2. Run all tests to watch the new test fail. Since the implementation has not been written yet the new test must fail, but running the test suite now has two benefits. Firstly it checks that the new test is not worthless by always passing. Secondly it ensures that the test suite is run frequently causing the code to be exercised often.[98]
3. Write the minimal amount of code required to make the new test pass. This code is not supposed to be perfect, but the simplest implementation to pass the test.
4. Run the tests to watch the new test succeed; the naive implementation passes the tests. This is a good baseline to start improving the code from.
5. Refactor the new code. The implementation can now be improved to make sure that it is of production quality. The test suite can be used to prove that the refactor is not changing the functionality of the code.

[98] Note that the code might not even compile now since the new function under test is not defined. To remedy this the developer can do the least that is required to get the system compiling again, e.g.`newFunction=⊥`

This cycle can then be repeated with a new test for a new piece of functionality. TDD also has good support for regression testing. If a bug is discovered then the developer tasked with fixing it would write a test to reproduce the bug before fixing the current implementation. In this way the set of test cases is broadened to cover even more possible code paths and to ensure that previous bugs are not reintroduced by future changes.

This is an approach to development advocated by the Serenity project team, and a core development technique that was attempted during the project,[99] for a number of reasons. Firstly, following TDD ensures that a project has a large test suite with a good coverage of the code base because functions should not be implemented without a test being written first. This is highly beneficial because it shows that the software is reliable and it gives developers confidence that their changes are not damaging the functionality of

[99] See Section 5.2

existing code. The 2005 study by Erdogmus et al. supports this as they found that when adhering to the test-first nature of TDD "programmers write more tests per unit of programming effort" and that, in turn, more tests lead to an increase in productivity.[100] A second benefit for TDD is suggested by Sommerville who states that TDD "helps programmers clarify their ideas of what a code segment is actually supposed to do".[101] This is because constructing the test for a new piece of functionality involves thinking about its requirements and design.

[100] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005

[101] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011, page 222

Summary of this section Testing is an important technique in software development. Haskell has great support for testing, including some powerful libraries that are able to test at a level that is not possible in other programming languages. However, when writing a test suite for your application it is important to keep Dijkstra's warning in mind: "testing is hopelessly inadequate. . . [it] can be used very effectively to show the presence of bugs but never to show their absence."[102]

[102] E.W. Dijkstra. On the reliability of programs. circulated privately, 1971. URL `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF`

4.8 The Good, the Bad, and the Ugly

So far this chapter has largely considered ways in which FP is advantageous in development, and some design patterns to this end. This section considers various issues that are likely to occur in Haskell development for games but can be challenging to overcome. The severity of these issues are considered and some mitigating strategies are suggested.

4.8.1 Efficiency Problems

A game, a real time strategy game in particular, must run efficiently. However, reasoning about space and time usage in Haskell programs can be difficult due to the nature of lazy evaluation and its interaction with garbage collectors.[103] This difficulty can make it harder to develop an efficient game.

[103] R. Cheplyaka. Reasoning about space usage in haskell, 2012. URL http://ro-che.info/articles/2012-04-08-space-usage-reasoning.html

A common efficiency problem encountered by Haskell developers is that of thunk leaks. A thunk leak is caused by a chain of dependent thunks stored in the heap waiting to be evaluated. Fortunately, once the cause of the problem has been located it can often be relatively simple to fix.[104] However, in other cases it may not be as easy to fix without more work going into redesigning and re-architecting large portions of code.

[104] E.Z. Yang. Anatomy of a thunk leak, 2011. URL http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/

During the early phases of the development of Serenity a memory leak was discovered in the Gloss graphics library. Even a very simple Gloss program to display the amount of time run so far would gradually consume more and more memory as it ran. Tests discovered that the program would acquire about twenty-five kilobytes of memory per second until it ran out of memory and crashed. With a much more complex program with a larger 'world' to keep track of this behaviour would have been much worse. Thankfully, upon reporting this issue to the developer of Gloss he was able to quickly identify the problem as "a typical laziness-induced space leak". This example not only illustrates how common memory leaks can be, but also the benefit of working with open source libraries provided by a very helpful community.

4.8.2 Dependency Hell

The Haskell Common Architecture for Building Applications and Libraries (Cabal)[105] was used to ease building and packaging of the game. Using Cabal it was possible to create a package that could be distributed to users for them to install on their systems. Cabal allows a developer to provide a list of dependencies that must be installed before the package can be used. A user can then use the cabal-install program to install all of these dependencies automatically along with the package itself.

[105] http://www.haskell.org/cabal/

Unfortunately dealing with dependencies did not work very smoothly and was a major source of frustration and time wasting.

The first large indicator of this problem came when trying to update some dependencies on Mac OS X. The quickest and easiest way to start developing with Haskell on Mac OS X, the operating system used by three members of the team, is to install the Haskell platform[106]. The Haskell platform provides a compiler and runtime, a number of useful developer tools (including Cabal and cabal-install), and a number of commonly used packages. However, the version of the Haskell platform available during the development of this project, version 2012.4, included a version of the Glasgow Haskell Compiler (GHC) which was incompatible with some of the newer versions of the packages that the project depended upon. These newer dependencies were required for the bug fixes and features that they contained, but it was impossible to compile and install these dependencies using the components of the Haskell platform at the time. In order to solve this issue the Haskell platform had to be removed in its entirety and replaced with the latest version of GHC compiled from source. Doing this also meant downloading and installing cabal-install from source as well. Working out the set of steps required to reliably fix the problem took a couple of hours. Replicating these steps on the other Macbooks took several more hours.

[106] http://www.haskell.org/platform/

This was not the end of all troubles with dependency management. On several occasions team members were forced to reinstall all dependencies. Compiling the Haskell code for almost thirty libraries and their dependencies is rather slow and time consuming, especially due to the size of some the requirements such as the Haskell bindings for the OpenGL graphics system. Complete reinstalls were required when cabal-install would seem to inexplicably become unable to install a new package due to conflicts. There appeared to be no easy solution to these problems other than to remove all previously installed packages and start again. It was also discovered that in order to install a library with profiling enabled all of its dependencies had to have been installed with profiling enabled as well. So, the Linux user also had to undergo the long and tedious reinstall cycle at the beginning of the project as the distribution had disabled library profiling by default.

4.9 Conclusions

Just as the SpaceTime prototype taught many lessons that proved useful when starting the full game (see Section‘2.1.3), coding Project Serenity itself has pointed out many things that could be improved in the future. Like any language, there are many small tricks and subtleties that separate satisfactory Haskell code from exemplary Haskell code. Also new methods are being discovered at a fairly rapid pace in the Haskell world: lenses are a relatively new discovery, as are free monads.

Among the key decisions made in the coding of Project Serenity, probably the one which had the most effect on what was able to be eventually produced was the decision to use Gloss for the graphics. Gloss allowed for very rapid development of the main game graphics, but some effort was still required in the area of UI elements. However, it is safe to say that the effect of this decision on a project of this timeframe was overall very positive. In the future it would definitely be very beneficial to develop a replacement or extension of Gloss that would allow for more flexibility and more provision for adding features (such as clipping).

Another key decision was to build a networking framework on UDP. The separation of concerns between the networking and the rest of the code was well handled, but it is possible that TCP would have worked just as well for the implementation, and that the development time could have been spent elsewhere.

Methods based on free monads, such as that provided by the *pipes* library (see **??** above) could have been used to enable a greater modularity to the implementation, making adding of new concerns, such as journalling, easier in the future.

Throughout the various design patterns identified in this chapter, a common theme can be discerned: that of higher and higher levels of abstraction. One of the greatest strengths of functional languages is the ability to use abstractions to closely model domain specific processes. Monads are an excellent example of this — the bind function can be thought of as a kind of “programmable semicolon”, with the do notation acting to allow easy creation of domain specific languages.[107] The lens combinators are also examples of this, and can be used to model many of the stateful styles of coding using operators like += all within pure computations.

Abstraction and separation of concerns are two of the most important concepts in programming and software development, followed closely by maintainability and testability. Much of the power of Haskell lies in its unique provisions to these ends.

[107] Monadic parsers are a great example of this; see G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(04): 437–444, 1998.

5 Project Management

PLANNING IS AN UNNATURAL PROCESS. IT IS MUCH MORE SATISFYING TO DO SOMETHING AND THE NICEST THING ABOUT NOT PLANNING IS THAT FAILURE COMES AS A COMPLETE SURPRISE RATHER THAN BEING PRECEDED BY A LONG PERIOD OF WORRY AND DEPRESSION.
— Sir John Harvey, c.1800

Project management is an essential piece of the software engineering process. Projects must be managed carefully and correctly because of the budgetary and scheduling constraints they are subject to. Whilst good management does not guarantee the success of a project, badly managed projects will usually result in failure.

This chapter will discuss and review the project management techniques used during this project. It aims to lay out the plans that were made to help the project be completed successfully, why these methods where chosen, and how useful they were.

5.1 Time Planning

Overall the project was timed reasonably well, every deliverable was completed to the expected quality by the required deadline. The specification was revised to the customer's satisfaction prior to the deadline for its finalisation, the poster was delivered on time despite issues with the printing contractor, and the final report and software deliverable are also expected to be completed and delivered on schedule at the time of writing. The only time management issues were related to soft deadlines which were internal to the project and had little impact on the project as a whole.

Despite this the time management of the project cannot be said to be entirely flawless. The project was scheduled to take advantage of the time mutually available in term one, and avoid infringing study time for other modules in term 2 and 3, but this aim was not really acheived. By the end of the first term the project should have been at the alpha stage, but the volume of the work required was drastically underestimated, as was the number of work hours available. It was important to ensure the group spent a lot of time together to avoid miscommunications and to ease collaborative

efforts, so a lot of effort went into scheduling coding session each week and emergency development weekends to mitigate any delays that occurred. The term 2 scheduling focused mainly on fixing minor bugs, strengthening weak points and developing advanced features beyond the project's requirements, but upon reflection, many of the features scheduled for completion in term 1 bled over all the way to the end of term 2.

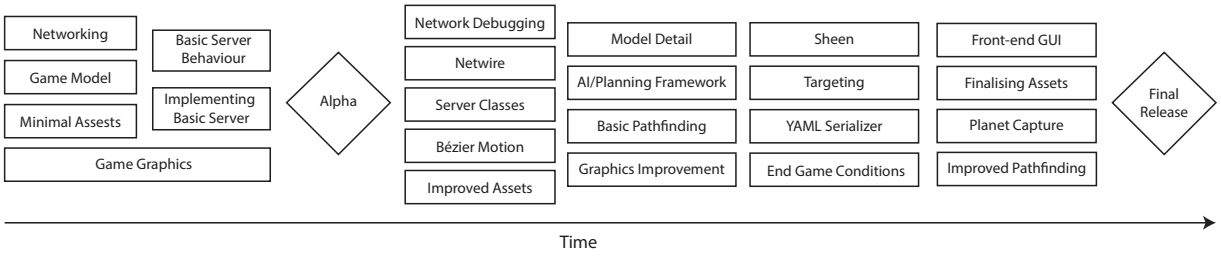


Figure 5.1: Actual workflow of project.

Casual conversation was an issue throughout the course of the project, very often a concise meeting deteriorated into off-topic conversation and cost the project valuable time, the informal atmosphere in the project group exasperated the problem since the management roles couldn't failed as a part-time occupation, and there was no "bad guy" to keep the group focused on project work all the time.

Not only was it difficult to maintain a good working pace, but it became difficult to assess the appropriate amount of time to ask of each team member since the number of university contact hours per week fluctuated as modules began/ended and as other assignments were set.

5.1.1 Unavoidable issues

One of the earliest issues arising with the schedule was attempting to arrange collaboration sessions which didn't conflict with any group member's other arrangements (particularly university assignments and contact hours). Rather than working asynchronously, the group decided to meet later in the evening and on weekends, when group members were tired and less productive, this made making milestones harder to reach on schedule as the project progressed.

An unavoidable issue in most projects is what's known as student syndrome, where people only start to fully apply themselves to a task in the last possible moment before a deadline.[108] Although lack of motivation was never a problem, it was often the case that the team spent too much time perfecting something that was already of an acceptable quality, when there were other project components which desperately needed work.

The most severe problem was slightly underestimating the time required for each project component. It's very difficult to accurately gauge how much work will be required to complete a particular

[108] H. Maylor. *Project Management*. Pearson Education Limited, 4th edition, 2010

task, and a schedule is almost always an underestimate. Even under the most efficient working conditions, the project's critical chain gradually slipped further and further behind schedule until many unfinished components were under threat of being unfinished by the delivery date.[109]

For future projects conducted in this atmosphere it would be well to be aware of these issues in advance, and take greater steps to alleviate them. Many good strategies for dealing with these type of problems are outlined in Martin 2011.[110]

[109] G.K. Rand. Critical chain. *The Journal of the Operational Research Society*, 49(2):181, 1998

[110] R.C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall, 2011, Chapter 10, page 135.

5.1.2 *Conclusions drawn from time management*

The initial scheduling of the project was overambitious, and it took quite a few emergency work sessions to bring the project back on schedule. Time could have been managed more effectively if the estimates were more realistic, however, an overestimate of time might have made the project team prone to Parkinson's law.[111] Time estimation can is a double edged sword, since it takes a delicate balance to not damage the progress of the project by underestimating or overestimating, but in practice it's impossible to accurately predict how much time certain tasks will take and the time required for debugging varies massively between bugs.

The problem with scheduling in software projects is that any time allocated for fixing unknown bugs and fine tuning features can be used as a means for a developer to defer fixing detected bugs until later. The problem is that deferred bugs will then cost the project even more time because important issues were not dealt with as they arose. Not only will there be a context switching overhead for any deferred bugs, but there may be extremely serious, yet undetected bugs that then cost the developers debugging time which hasn't been accounted for in the project schedule.[112] One possible solution to such delays is to keep the entire team motivated to meet soft deadlines throughout the project so the team is kept forced to deal with delays promptly, that way the effect of delays for the overall project is largely mitigated.

Time management is a serious problem for any software project, and requires a strong will and good practice to get right. Many of the issues with time management are exacerbated by the conditions student projects are subject to, and so a great deal of effort must be put into keeping projects focused and running to schedule.

[111] J.F. Bryan and E.A. Locke. Parkinson's law as a goal-setting phenomenon. *Organizational Behavior and Human Performance*, 2(3):258–275, 1967

[112] J.F. Bryan and E.A. Locke. Parkinson's law as a goal-setting phenomenon. *Organizational Behavior and Human Performance*, 2(3):258–275, 1967

5.2 *Development Model*

An agile approach to software development was chosen for this project. This methodology was chosen because of its focus on rapid development and handling change. When working in a small team, a heavy weight plan-driven development approach can dominate the actual process of development due to the overhead. Sommerville states that using such an approach means that "more time is spent on how the system should be developed than on program development and testing."[113] In contrast, an agile approach is designed to deliver working software quickly so that changes can be suggested and implemented in future iterations.

[113] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011, page 58

The agile method that was used was that of extreme programming developed by Kent Beck.[114,115] Extreme programming is an 'extreme' approach to iterative development. Small releases are made as quickly as possible. These releases are then evaluated and iterated on until a final release that meets all requirements is delivered. Instead of planning and designing for the far future, extreme programming advocates doing both of these activities — bits and pieces at a time — throughout the entire project development lifecycle.

[114] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10), 1999

[115] K. Beck. *Extreme Programming Explained*. Addison-Wesley Professional, 2000

One core practice at the centre of extreme programming is testing and test driven development. Test driven development, as described previously in section 4.7, involves developing test cases before coding the actual implementation. The benefits of test driven development are a system that is thoroughly tested, reduced ambiguities in specification before implementation begins, and avoidance of 'test-lag'. However, Sommerville notes a few problems that can be encountered when using test driven development:

1. Programmers prefer programming to testing. It is very tempting for a programmer to write incomplete tests, or skip test writing altogether, before moving onto the more rewarding task of implementation.

2. In some cases tests can be difficult to write. For example, testing user interfaces and display logic.

3. It is hard to judge the completeness of a test suite. There may be a large number of tests, but do they actually cover all of the code and all possible program execution.

Although the Serenity project does include a test suite the team failed to stick to the test driven development tenet of extreme programming. This was mainly caused by the first issue pointed out by Sommerville. The team preferred to go straight into implementing a feature or enhancement and then, maybe, write tests after the fact.

Another important practice in extreme programming is that of pair programming. Two developers work in tandem at the same computer. One programmer, the driver, actively writes the implementation of the program. The other, the observer, continously

reviews each line of code as it is typed and thinks about the direction of the work.[116] There are a couple of major advantages to pair programming:

1. It is an informal code review process that can be very effective at discovering errors as the code is written.
2. It promotes collective ownership and responsibility for the component being worked on. Code is not 'owned' by an individual who may dislike others working in the same area or be demotivated by critism during code reviews, similarly one individual is not held responsible for any problems.

[116] L. Williams. Integrating pair programming into a software development process. In *14th Conference on Software Engineering Education and Training*, pages 27–36, 2001

Studies have shown that pair programming may have little effect on overall productivity, but creates a substantial reduction in errors in the code.[117] This is prescribed to the continuous code review that occurs, and a decrease in false starts and redoing work.

[117] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *First International Conference on Extreme Programming*, pages 223–247, 2000

Pair programming was an effective technique that was used throughout the development of Serenity. The experience lead to the conclusion that pair programming is a good method for development for the advantages mentioned above as well as the following reasons:

Problem solving When a problem is encountered two people are able to discuss it together which often helps either or both of them coming up with a solution quicker than they would individually.

Learning Knowledge is constantly exchanged within the pair. So after a component has finished and the pair move on to other work they have both become more effective programmers in some way.

Team building Working together lead to better communication and enhanced teamwork. This made the project team a more effective work group.

However, due to such a small team it was felt that it would be impossible to work on all tasks in pairs and still finish the work within the time constraints. Therefore, some work was done individually, but still always trying to work in close proximity to enable teamwork where necessary.

5.3 *Project Control*

Projects that have run late or overbudget often find that the steps to failure happened gradually. The small extra costs or wasted days gradually added up to form a very large and significant problem. Project control is about detecting these small deviations and then implementing corrective actions that prevents a build up of small problems.[118]

[118] H. Maylor. *Project Management*. Pearson Education Limited, 4th edition, 2010, page 291

Any large software system is going to undergo change as it is developed. Requirements may change, bugs have to be fixed as they appear, and original design decisions may turn out to be insufficient. Therefore, a set of change management processes are required to ensure that the evolution of a system is controlled and does not succumb to scope creep or other problems that would prevent successful delivery of the software.[119]

[119] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011, page 685

For the Serenity project the change management process shown in Figure 5.2 was devised. This protocol was designed to allow stakeholders outside of the development team to submit change requests for assessment. When a change request is received it would be evaluated in terms of its cost and impact on the project requirements. If the change is accepted then a notification is relayed to the whole development team and relevant stakeholders. All change requests would be archived so that reasons for rejection or approval could be reviewed again in the future.

In the end no change requests were submitted by any stakeholder not a part of the development team. However, it is believed that this well documented and rigorous change management workflow would have enabled the project to deal with any such change requests in an efficient manner. Many minor changes, such as bug fixes, were discussed more informally within the development as they had no effect on the schedule or requirements. This worked effectively whereas using the formal change management procedure for such small changes would be overkill and lead to a lot of time being wasted in communications overhead.

A formalised change management workflow is necessary for a successful project, but it must only be used where appropriate to ensure that it does not get in the way of smaller changes that can be dealt with more informally.

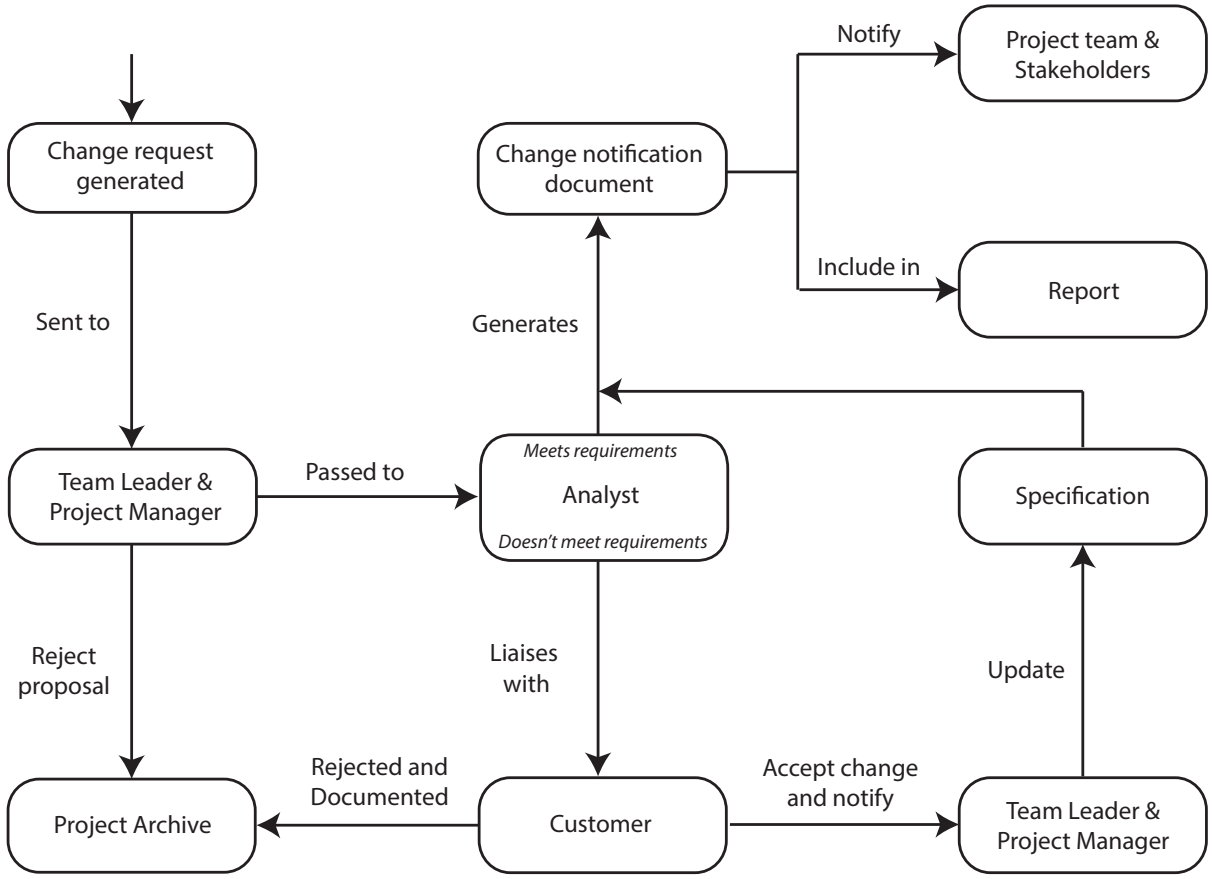


Figure 5.2: Change management workflow.

5.4 Communication

The various stakeholders that were identified during the specification stage are shown in Table 5.1. The relationship between the stakeholder and the project is shown, along with a rough estimate of their power and interest.[120] This grid formed a reference for making sure that all interested parties were communicated with appropriately throughout the duration of the project.

[120] See A. Mendelow. Stakeholder mapping. In *Proceedings of the 2nd International Conference on Information Systems, Cambridge, MA*, 1991

This communication plan was developed early during the project specification to enable communication to be effectively prioritised and maintained. Here the efficacy of the communication plan is reviewed.

5.4.1 Supervisor Meetings

Prior to starting the project it was thought that regular communication with the project supervisor was likely to be a critical factor in success of the project. Therefore the possibility of weekly meetings with at least one member of the group if not more were planned. During the project the feedback from these meetings from the supervisor was very helpful in ensuring that the project was kept on track and that the goals were always kept in mind. Although the meetings were not as frequent as planned there were several per term. Also, the team made their calendar available to the supervisor so that she could drop by during the planned coding sessions that occurred twice weekly.

Stakeholder	*Relationship*	*Power*	*Interest*	*Requirements*	*Measurements*	*Communication Strategy*
Project Team	Internal	High	High	Good working environment, creative input.	Meeting project spec, good grades!	Various, detailed elsewhere.
Supervisor — Sara Kalvala	Internal	High	High	Good communication.	Adherence to spec, good PM, high quality write-up.	Weekly meetings.
Client — Matt Leeke	Core External	High	High	Good communication, creative input, hard work	Strength of software, strength of report	Weekly meetings.
Second Assessor	Core External	High	Low	None	Marking scheme	Deliverables only.
Projects Organiser — Steve Matthews	External	High	Low	Cooperation when required.	Deliverables on time.	Email or meeting if required.
Playtesters	External	Low	High	Able to report issues / feature requests.	Strength of game, input considered.	Email.
Other future users	Rest of World	Low	High	Game works and is reliable.	Strength of game, re-playability.	Website, forums, blog.
The Haskell and FP Communities	Rest of World	Low	High	None	Interest in / strength of results and tools released.	Online as above, and via the final report.

Table 5.1: Stakeholders for the project.

5.4.2 *Client Meetings*

The client is clearly vital to the success of the project, and so it was planned for there to be weekly feature releases to the client along with meetings to discuss the changes. It was thought that continual feedback on each release would allow for early identification of any problems in the game. In reality these meetings were not actually this frequent, due to time constraints on the client as much as on the project team. However, feedback from the client was always positive and no large changes needed to be made to the project requirements. The approach taken by the client was commendable, providing a valuable source of feedback and helping to maintain focus, without attempting to micromanage or control the nature or creative freedom of the development. Indeed, the project team could have made more effort in keeping up communication with the client and taking advantage of his input.

5.4.3 *Projects Organiser and Second Assessor*

The projects organiser could exert a strong influence over the project if they wished, but as there are many projects and it would be inappropriate for them to demonstrate partiality, therefore extended levels of communication were not necessary. Brief updates pertaining to deliverables were all that were required. However, the team was aware that if the project organiser initiated communication then they should be made a high priority.

It was felt that communication with the second accessor was, for the most part, not appropriate, excepting when delivering this final report and various presentations. This approach seems to have been correct.

5.4.4 *Playtesters and End Users*

End users are clearly important to the goals of the project, but they had little interest or influence during the early stages. Therefore, infrequent updates via email and a mailing list for any events that are organised were sufficient communication with this group.

There was not time in the end schedule for nearly as much playtesting as was desired, and so this communication plan has remained largely untested. However, it is clear that good communication would be required to maintain interest and give feedback on bug reports etc.

5.4.5 *The Haskell and Functional Programming Communities*

The overall end goal of this project is not just the game that was developed, but an examination of Haskell and Functional Programming as a game development environment. However, the Haskell community at large is unlikely to have much interest in the project during its development. Communication back to the community

should therefore be largely via this final report, as well as the methods for end users above.

5.5 Team Structure

Before starting the development phase of the project the roles and responsibilities for each team member were formalised. Each person was given a set of roles and responsiblities for which they would be primarily in charge of. It was envisaged that these roles would be flexible and that responsibilities would be shared, but having a formal specification of the member ultimately responsible for an aspect of the project's development and management would be useful for ensuring that everything was done correctly. The roles that were given to each team member as laid out in the specification are as follows.

5.5.1 Laith Alissa

Project Manager Responsible for overseeing the human aspects of the project in general, including managing a schedule, organising meetings and collaborative development sessions. Makes decisions involving tradeoffs between project time, cost, and quality.

Customer Liaison Meets with the customer at regular intervals to discuss the project progress, outlook, and any issues which are require customer input.

Analyst Responsible for ensuring the customer requirements are addressed during planning and development stages of the project, and ensures that the solution will sufficiently address the customers needs.

User Manager Responsible for communicating with play-testers and users. Finds end users to test the product in the later stages of development, and provides feedback to the project team. Prime responsibility is to identify issues which are not clearly visible from the project development perspective, but are more apparent to end users.

Graphic Designer Responsible for prototyping and developing graphical design elements, such as ship sprites, terrain, maps and user interface.

5.5.2 Jon Cave

Code Reviewer Responsible for interpreting other developer's code, checking for logical inconsistencies and familiarising themselves with the project as a whole.

Chairman Responsible for coordinating meetings, ensuring all issues are resolved or at the least discussed, and that all meeting participants have a chance to voice concerns and contributions.

Testing and Integration Officer Ensures the code is thoroughly tested for bugs, and discovered bugs are flagged and dealt with in

reasonable time. Responsible for managing integration testing to prevent bugs occurring on the master branch.

Security Officer Checks for security flaws in the product, performs security evaluations (such as penetration testing and code review) to ensure the product is sufficiently secure.

5.5.3 Joseph Siddall

Software Librarian Ensures team completes documentation to a sufficient standard for long term maintenance.

Line Manager Oversees day to day development, intervenes if a developer is off track, ensuring minimal time is wasted perusing low priority work.

Testing and Integration Officer Ensures the code is thoroughly tested for bugs, and discovered bugs are flagged and dealt with in reasonable time. Responsible for managing integration testing to prevent bugs occurring on the master branch.

5.5.4 Victor Smith

Team Leader Leads the project. Responsible for coordinating the project team, ensuring team members are working to the best of their ability, responsible for making decisions when the there is no clear solution to a particular problem.

Lead Developer Responsible overall for technical areas of the project. Consults other developers when there is development difficulty, also responsible for directing programming style and technique.

Composer Composes soundtrack for the game, and produces required sound effects, voice overs, soundscapes, and other related resources.

5.5.5 Common Roles

Some of the roles were shared by the whole project team. This is because all of the team members were involved in day to day development and game play design.

Programmer Performs the day-to-day programming specified by the line manager.

Tester Performs general code testing (e.g. unit tests, component tests).

Gameplay Designer Critically analyse gameplay design and experience, giving feedback to the team leader on how to improve the game's appeal.

5.6 Risk Management

A proactive approach to risk management was taken. This technique was chosen in order to maximise the probability of avoiding risks instead of having to move into 'fire-fighting mode' if something went wrong.[121]

[121] R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 7th edition, 2010, page 745

As part of this proactive risk management strategy, a number of potential risks were identified. These risks are shown in table 5.2 along with their estimated probabilities of occurring and impact if they were to occur.

Risk	*Description*	*Probability*	*Impact*
Length underestimate	The time required to develop the software is underestimated	Medium	High
Size underestimate	The quantity of work required to complete deliverable on time has been underestimated	Medium	High
Team member illness	One or more team members unable to work due to illness	Medium	High
Hardware failure	Damage to critical hardware causing loss of data	Medium	Medium
Requirements change	Large number of changes to requirements during development	Medium	Medium
Ambiguous requirements	Requirements are not fully understood or misinterpreted leading to loss of development time as the specification is recreated	Medium	Medium

Table 5.2: Risk identification and analysis.

With the risks identified, and their likelihood and consequences estimated it is necessary to draw up plans to mitigate their effects. There are three types of management strategies for individual risks: avoidance strategies to reduce the probability of the risk occurring; minimisation strategies to reduce the impact of the risk; and contingency plans to deal with the risk if it does arise.[122] It is best to avoid the risk, but if this is not possible then minimisation of the effects and, finally, contingency plans should reduce the overall impact of a risk on the project. The mitigation and management strategies for each risk previously identified are listed in table 5.3.

[122] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011, page 601

Risk	*Mitigation / Management*
Length underestimate	Detailed work breakdown with weekly releases to ensure that schedule slippage can be caught early
Team member illness	Well documented code (enforced by the software librarian) so that other members can quickly start work on less familiar sections of the codebase
Hardware failure	Backups and distributed source control, see section 5.8
Size underestimate	Detailed work breakdown structure
Requirements change	Thorough change management system, see section 5.3
Ambiguous requirements	Thorough planning phase

Table 5.3: Risk mitigation and management.

The final stage of the risk management process is monitoring. Throughout the duration of the project each identified risk was reassessed for changes to its probability and impact. This allowed the mitigation and management strategies to be revisited to ensure that they were as effective as possible.

The two previously identified risks that actually occurred were team member illness and length underestimate. On a couple of occasions a team member was ill and unable to attend group work sessions or work to their full capacity. Fortunately, the team was able to reduce the impact of this by ensuring that the work each individual was performing was not hindered by an absence. This was done by allocating work tasks to be as separate as possible to allow more parallel development to occur. Also, an illness was never so severe as to stop a team member from working for longer than a day or two.

The proactive approach to risk management was a good choice. By reviewing the potential risks before starting the development phase of the project it was much easier to avoid risks that could have had disastrous consequences for the project. For example, by implementing a thorough backup strategy prior to any data loss actually taking place it was ensured that no work would have been lost if a hardware failure had occurred. Continuously monitoring and reassessing these risks was also helpful in preventing any risks becoming more probable or having a greater impact.

5.7 Legal, Ethical, and Social Issues

Prior to embarking on this project several potential professional issues were identified and discussed to ensure that they were addressed appropriately.

Any software project faces issues surrounding copyright and intellectual property. During development the team were aware that all third party libraries used in the project must be licensed appropriately. This meant only using software with a permissive license (e.g. Apache, BSD, or MIT licenses) and no unlicensed proprietary software. It was also important to avoid infringing on the intellectual property of established game publishers and their existing games. Obviously it was infeasible to review all previously published games to check if Serenity bears great similarities to any of them. However, the project team was careful not to directly copy any designs or unique game play characteristics of existing games that they are familiar with.

The games industry is faced with several ethical and social issues that were thought about carefully when designing the Serenity game. Highly controversial issues that surrounds current games, such as racism and violence, were thought about carefully and it was ensured that they would not be issues with the Serenity game. This is because the game is based in a fictional world with a very unrealistic graphical representation. Another social dilemma for game creators to consider is that of the addictiveness of their creations. Players often invest many hours, often consecutively, in playing games and can even spend large sums of money on pay-to-play games. The game that was developed was designed to have short playing periods that would allow convenient times for the player to quit.

5.8 Evaluation of Tools and Techniques

A number of different tools were used in the day to day running of the project. These tools were used to ease the collaboration between members by ensuring that everyone knew what they were supposed to be doing and how to share it with the others.

5.8.1 Source Control: Git

Source control systems are an essential tool for software projects, especially those with multiple developers. Using source control to easily track the changes to source code is helpful because, as McConnell explains, having a history of changes helps a developer to identify the origin of bugs quickly.[123]

The Git source control system[124] was chosen for this project. Git was chosen for a number of reasons. Firstly, it has a lightweight branching model which allows for quick creation of new development branches for experimental features independent of any other development. Chacon notes this as Git's "killer feature".[125] Another reason for choosing Git is its distributed nature. Every user has a full clone of the entire repository that can act as a replacement for any other instance of the repository; this means that there is no single, centralised point of failure.

A centralised master repository was hosted on Github, a popular code host.[126] This made it easier for team members to share their changes to the code base. Once a change has been made it can be 'pushed' to the Github repository; other team members can then 'pull' it to their local repositories.

Git was an ideal source control system for the project. Not only does it allow the users to see the entire history of changes made to the project it makes collaborative development between multiple team members incredibly easy. Instead of having to resolve conflicting changes to files it is possible to use the built in merge tool to combine the work of two or more people on the same set of files with minimal effort. Git is recommended for all future development projects.

[123] S. McConnell. *Code Complete: A practical handbook of software construction*. Microsoft Press, Redmond, 2nd edition, 2004, page 667

[124] `http://git-scm.com/`

[125] S. Chacon. *Pro Git*. Apress, New York, 2009, page 38

[126] `https://github.com/`

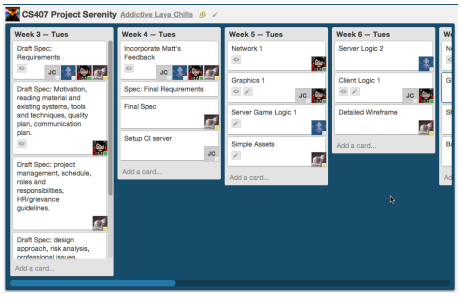


Figure 5.3: Project Serenity's Trello page

5.8.2 Tracking and Managing Releases: Trello

Trello is a modern, online take of an age old concept — a wall of post-its. Trello allows for overall planning of tasks and communication between members in a highly dynamic, fluid way. Cards are arranged into a list-of lists, each list being some kind of functional decomposition or 'stovepipe'. Trello has been used in the project to track the requirements for each weeks releases, but still allowing rapid changes to be made and communicated.

Trello acted as a useful reference to the current state of the project: what tasks are planned and who is doing them. It is a good tool for keeping a small group of people organised, but requires

careful management to keep it up to date and useful.

5.8.3 Bug Tracking: Github Issues

Trello does not entirely alleviate the need for a full bug tracker. When working of development of individual features and components it was necessary to have a method of tracking more transient issues and tasks. The initial plan was to use the service provided by Fogbugz, a stand alone bug tracking web application. However, in the end the Issues feature built into Github hosted repositories was used. This was decided because it reduced the number of third party services that were relied upon and it provided better integration with the hosted repository itself. Although it is much more minimal than Fogbuz, it provided enough features, such as milestones, labels, email notifications, and issue assignment, for this project.

An issue tracker was very useful for noting down smaller tasks and bugs that cannot be fixed immediately, but must not be accidentally forgotten about. If a project is using Github to host code then the Issues component is recommended if only simple issue tracking features are required. However, for more complex requirements a more fully featured service may be a better option.

5.8.4 Backups

Backups of the source code and other project assets are essential. In the event of a disaster, such as losing a computer to a fire, it must be possible to recreate the entire project in its latest state quickly and without any repetition of work.[127] The use of Git and Github for source control made it simple to ensure that all source code was located on multiple computers. By 'pushing' commits to the Github repository all code was stored in the cloud. When other team members 'pulled' the code later on it was then mirrored on their computers as well. Extra precautions were taken by also 'pushing' the Git repository to private servers owned by the team members. Also, the free service provided by Dropbox[128] was used to share and backup any other files, such as assets and planning notes.

[127] S. McConnell. *Code Complete: A practical handbook of software construction*. Microsoft Press, Redmond, 2nd edition, 2004, page 669

[128] `https://www.dropbox.com/`

Fortunately a situation which required recovery from backups was never encountered, but the strategies put it place should have been more than adequate if the need ever arises in the future. Since the main backup strategy was part of normal development workflow it was impossible to forget to implement it. However, it does have to be ensured that all members are sharing their changes as frequently as possible (even if a feature is not fully finished) to prevent any work in progress from being lost in case an individual's computer dies. The extension to this core backup strategy, storing the repositories on other personal servers, was easy because of Git's feature set which was designed to be distributed in nature and so

allows tracking and updating of multiple remote repositories.

5.8.5 Continuous Integration: Jenkins

Section 4.7 introduced the idea of continuous integration as a method of regularly running the test suite and performing a full build of the software. The Jenkins continuous integration server[129] was used for this project. Jenkins was chosen because it is a widely used open source project.[130] Jenkins also works well with projects hosted on Github because of its Github plugin[131] and Github's Jenkins specific post-receive hook.

[129] http://jenkins-ci.org/

[130] http://stats.jenkins-ci.org/jenkins-stats/

[131] https://wiki.jenkins-ci.org/display/JENKINS/GitHub+Plugin

Although Jenkins does not have a very good user interface it suited the needs of this project. It was a good tool for running a test suite with every change to the code base and ensuring that the game would build correctly. Email notifications on failure were useful, but also relatively noisy at times. Continuous integration is probably more useful for more mature projects that need to ensure that changes do not introduce any regressions than it is for a project in the early stages of development when a lot of features are not fully working.

5.9 Review of Requirements

The project was successful in fulfilling most of the requirements but, due to delays during development, not all the features could be completed in time. Rather than risk not having a playable game, some of the minor features were deferred until the end of the project, and focus was shifted to the project's critical chain.

5.9.1 Functional Requirements

FR1 Multiplayer The specification required two player games to be supported, at present the game will theoretically support any number of players, but in practice the game dynamics would decay when the number of players exceeded the expected capacity of the selected map. The game met and exceeded the multiplayer requirement outlined in the specification.

FR2 Fleets A core concept of the specified game was for players to be able to create and customise a fleet of ships, including an option of equipping weapons of the player's choice. Each player is now given the opportunity to customise their fleet constructing ships they wish to include in deployment and (optionally) naming ships. The requirement for including a fleet of ships was fulfilled, but some of the finer details of the requirement such as shield recharging remain incomplete.

FR3 Ship Customisation The specification required ships to have customisable weapon slots, currently each of the fleet's ships can be constructed using one of three different hull types. Each type offers a different ship speed, health, and shield capacity. The larger the ship, the higher its health/shield capacity, and the slower its speed, so the requirement was partially met, but the players are at present unable to select weapon slots.

FR4 Resource System The framework for the resource system was completed. However, the game mechanics for resources remains unfinished. The implementation of the remaining features would be fairly straightforward in the event that this project is continued, but from the player's perspective the requirement is unmet.

FR5 Planetary Capture Players can capture a neutral planet if their fleet secures the area for a sufficient amount of time. Once the planet has been held securely by one player for a sufficient amount of time, the planet then falls under that player's control, which would provide them with a steady stream of resources and contribute towards a victory condition. Planets can also be captured from other players in the same way a neutral plant can be captured, except it takes the capturing player longer to secure the planet, giving an advantage to being the first player to capture a planet. This requirement was completed as required by the specification.

FR6 Fog of War Although it was a relatively simple concept, fog of war was not completed in time since it was low priority and time was scarce. The project failed to meet this requirement entirely.

FR7 AI An advanced artificial intelligence framework was implemented. It took orders from the player and converted them into a lower level plan to execute. Some replanning is possible, but it is not a formal piece of the framework and it could be improved. Intelligent path finding algorithms were created to get ships to take the optimal route to their planned destinations. This AI system mostly met requirements, but is a bit less powerful than originally intended. However, its flexibility should allow it to be extended in useful ways in the future.

FR8 Campaign A campaign of battles between two players was always an 'if there is enough time' feature. Unfortunately, due to time constraints a campaign mode has not been implemented yet.

FR9 Operating System Requirements The game compiles and runs on both Mac OS X and Linux. Windows systems were not tested during development, but measures were taken to ensure the game can be supported on windows, for example there are function calls available in the sockets module to facilitate networking on Windows.

5.9.2 *Non-Functional Requirements*

NFR1 Fun The project group find the game fun, as did a small group of pre-alpha testers, but since there was no time for a formal play testing there is little data to support how *fun* the game experience is. The game experience almost certainly would be improved should missing features been implemented in time; the project group even had concepts to improve gameplay beyond the scope of the specification.

NFR2 Short game sessions Game session timings are not enforced by the game, and there is insufficient data available to gauge average play time, however, it seems that under normal circumstances a game should be no longer than 35 minutes if players are seriously aiming to win. The specification required the game be no longer than 35 minutes, but also didn't specify any action that needed to be taken in the time limit was reached, so the requirement can be deemed partially complete.

NFR3 Approachable Only the host's IP address is required to play a networked game, which is typical of most standalone games. A player is also able to host and join games using an elegant GUI making it very easy to get started.

NFR4 Reliable The game is required to be reliable enough to have a gaming session without crashing. The software was thoroughly

tested to ensure it wouldn't easily crash and results show the game, other than a few minor bugs, it meets the reliability requirement.

NFR5 Secure Since the game runs as with a master server controlling the entire simulation of the game the amount of cheating that is possible for a player is significantly reduced. Incoming messages are verified to be from whichever player they claim to be sent from by checking the originating IP and port. Players are also blocked from sending commands to ships that they do not actually control. However, packet forgery is still possible since there is no authentication or integrity checking at the network layer. This was briefly investigated by using a message authentication code algorithm such as SipHash,[132] but it was decided that it would be better to focus efforts on completing the game logic and return to network layer packet verification at a later date.

[132] J. Aumasson and D.J. Bernstein. Siphash: a fast short-input prf. In *Progress in Cryptology-INDOCRYPT 2012*, pages 489–508. Springer, 2012

Table 5.4: Requirement fulfillment

Requirement	*Priority*	*Requirement Status*
Multiplayer	Vital	Fulfilled
Fleets	Vital	Fulfilled
Reliable	Vital	Fulfilled
Ship Customisation	High	Partly Fulfilled
Planetary Capture	High	Fulfilled
AI	High	Fulfilled
Operating System	High	Fulfilled
Fun	High	Fulfilled
Approachable	High	Fulfilled
Resource System	Medium	Not Fulfilled
Short game sessions	Medium	Fulfilled
Secure	Medium	Fulfilled
Fog of War	Low	Not Fulfilled
Campaign	Optional	Not Fulfilled

As can be seen from 5.4, only one high priority requirement remains incomplete, the only other requirements which were not met were lower priority or entirely optional anyway. The project failed to meet all the requirements laid out in the specification, but it certainly succeeded in meeting enough to meet the projects overall aim – to produce a playable game.

5.10 Evaluating the Success of the Project [133]

[133] The literature review in this section is partly based on Vic Smith's previous assignment for the module Project Management, IB382

Project success depends heavily upon perspective, one stakeholder might consider the project beyond successful whereas another stakeholder may consider the project a total failure. Evaluating success must assess the project from a variety of perspectives and consider whether the project fulfilled each of their expectations.

> An architect may consider success in terms of aesthetic appearance, an engineer in terms of technical competence, an accountant in terms of dollars spent under budget, a human resources manager in terms of employee satisfaction, and chief executive officers rate their success in the stock market.[134]

[134] M. Freeman and Beale. *Measuring project success*. 1992, pages 8-17

The stakeholder's perspectives and success criteria had been formalised in the specification and can be contrasted with what the final deliverable to judge whether the project has been successful for each stakeholder.

Project Team The team's desire was to produce a game using Haskell, from their perspective they have made significant progress produced a high quality deliverable.

Supervisor The project supervisor required the team to stay faithful to the specification and meet external deadlines. She has been kept abreast of the project's progress, no deadlines have been missed and her feedback has been incorporated into development throughout the project. All success criteria for the supervisor has been met.

Client The client is satisfied that the progress report and the final product were of sufficient quality and were delivered on time. The client's positive feedback indicates the project was a success from his perspective.

Projects Organiser The project organiser required that all external deadlines were met, and since they were the project can only be considered successful from his point of view.

Play testers Sadly the play testing had to be pruned from the schedule due to insufficient time towards the end of the project, so the platesetters were unable to provide feedback. Unfortunately this likely means that the play testers consider the project (at least partly) a failure until such time they can experience the gameplay.

Other future users The success according to the future users can only be assessed further down the line, so this will remain unknown for the present assessment.

The Haskell and FP Communities This project has broken new ground in the Haskell community, since it's a pioneering effort to create a game in a language not typically used for game development.

> The project has given rise to a new real-time-strategy game design pattern in Haskell, which can be used as a model for writing similar games, or a basis for discovering improved design patterns. Regardless of whether this project will be considered "poorly architected" in the future, it runs as well as expected and is unique in the Haskell games community, so it is certainly successful in this context.

Project management is a relatively new field of operational research, and there are a number of techniques for quantifying project success, each with it's own bias towards certain types of project. This section utilises the concepts in "Mapping the dimensions of project success" [135], where Shenhar suggests that the traditional measures of project success — namely time cost and quality — are perhaps not the best, and points out the disagreement in the literature on this area. Building on evidence from previous studies, the authors suggest a new multidimensional framework for project success; and notably back this up and expand upon it by utilising empirical evidence.

This evidence takes the form of 127 structured questionnaires, returned from project managers from a variety of industries.[136] This data was then analysed using numerical measures from previous work[137] and the dimensionality of results determined using factor analysis.[138]

Despite an initial hypothesis of three dimensions to project success, the analysis suggests that there are in fact four. These dimensions were then correlated (using Pearson's Correlation) with the overall assessment of project success, with favourable results; and the relative strength of each measured, both for projects in progress and completed projects. The strength was seen to be consistent, apart from the fourth dimension which was significantly better correlated in completed projects. This is the 'future potential' dimension.

This empirical experiment causes this research to stand out in a way other papers on the subject of project success do not, in that the claims are in fact backed up.

Another notable feature of this paper is that in addition to the scientific numerical approach, a significant percentage of the text is given to practical consequences of the results and implications for the practitioner. Given that it is now nearly fifteen years old it is surprising that these ideas do not feature more prominently in project management canon.

The Iron Triangle is a primitive tool for measuring the balance between time, cost and quality. It has no quantifiable properties and is mainly an illustrative tool, but it does somewhat accurately express the relation between the three constraints.

Selecting two of the properties as desirable will be costly to the third, for example demanding a high quality product in little time will escalate the cost. The project had no real notion of monetary

[135] O. Shenhar, A.J. Levy. Mapping the dimensions of project success. *Project Management Journal*, 28(2), 1992

[136] It is claimed in the paper that despite this selection having not been made at random, with no consideration to stratification, and that 'the end products were aimed at a military market', that there is not an inherent bias. Especially in light of the low sample size this should be considered to be a somewhat dubious claim.

[137] Specifically Cooper and Kleinschmidt, 1987; Pinto and Stevin, 1988; Stuckenbruck, 1986; and Dvir and Shenhar 1992. (See full bibliography).

[138] Factor analysis being a statistical technique to determine the underlying dimensionality of a data set without previous knowledge of it. See Brayant and Yarnold (detail in bibliography) for a full treatment.

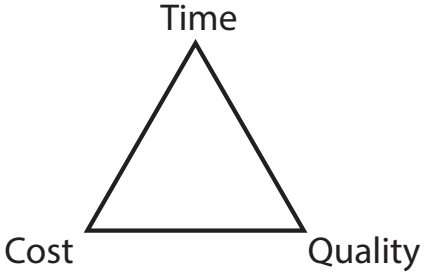


Figure 5.4: The Iron Triangle.

cost, so the balance was largely between quality, project time and personal time, where the personal cost a team members marks from other assignments, time for hobbies, sleep, etc. For almost all project components there would be an imbalance between the three properties, and most often, the *personal time* property would need to be increased to ensure good quality products while keeping to schedule.

Success Dimension 1: Project Efficiency The first dimension is interesting in the way it applies to software projects, since efficiency is extremely difficult to guage. Unlike a task requiring manual labour, efficiency can be assessed fairly easily since most tasks are repeatable and hence can be modelled fairly easily. It's difficult, on the other hand, to compare two software projects, since not only are they very rarely alike, but efficiency is not easily quantifiable. Some managers try to quantify development progress by measuring lines of code per day, as Hertzfeld implied in his article "-2000 lines of code", the number of lines a developer has written per day may have no bearing on the work accomplished.[139] Even it there was a convenient function of project progress to mark efficiency, that efficiency would likely be subject to changes. Suppose a large software project launches, stays on schedule and outputs an excellent number of lines per day, it might be the case that the developers are taking shortcuts and not being prudent, which could cause a massive slump in the efficiency late down the line when it's discovered that the initial few milestones are bug ridden and need to be revised. By contrast a project team missing the initial deadlines might be investing a large amount of thought into their work which could accelerate their efficiency later down the line. The important thing to note is that efficiency hinges on perspective, and it's very difficult to find a function to assess the efficiency of a software project. Given that it is in fact necessary for the model, it would be wisest to measure efficiency by comparing each estimated milestone completion date with it's actual completion date, since the milestones were all met on time, but some features were lacking, it's fair to claim the project progress was reasonably efficient *efficiency* available.

[139] A. Hertzfeld. -2000 lines of code. 1982. URL `http://folklore.org/StoryView.py?story=Negative_2000_Lines_Of_Code.txt`

Success Dimension 2: Impact on client At the time of writing, the project has yet to be deployed for the client, however, all communications with the client have given positive feedback, so provided the deliverable is sufficient then this dimension can be marked successful.

Success Dimension 3: Business and Direct Success Business and Direct success is a measure of whether the project provided "the sales, income, and profits as expected", "increase[d] business results" or whether the project "helped to increase market share" [140]. Shenhar does elaborate that other projects may not fall under the same assessment criteria, but will have some mechanism of measuring

[140] O. Shenhar, A.J. Levy. Mapping the dimensions of project success. *Project Management Journal*, 28(2), 1992

direct success. Since this project is academically motivated rather than monetarily driven, it would be fair to assess the success outcomes the requirements discussed in section 5.9. The project met most of these requirements, and the requirements it failed to meet were considered the least important. The project was largely successful in creating a fun game in Haskell, which met almost all the functional requirements and fulfilled all the non-functional requirements, granting the project at least a partial success under Dimension 3.

Success Dimension 4: Preparing for the Future The fourth dimension investigates how prudent the project is, whether it has invested in future opportunities, whether it explored further ideas, innovations or skills which may be required in the future. Assessment under this dimension can only result in an overwhelming success. The project explored two concepts which are already hugely popular and only growing in popularity, namely indie games and the Haskell programming language. Haskell's growing popularity is undeniable, it's a concise yet extremely powerful language which has recently being adopted by an increasing number of companies including ABN AMRO, Anygma, Amgen, Bluespec and Eaton. It has been used to construct many large scale projects such as ASIC, FPGA (design software), Haskore, GHC, Darcs and HAppS, projects which are of comparable size and complexity of many corporately produced games. [141]

Game development is also a very rapidly growing market, with the advent of mobile devices the target audience for video games barely excludes any social group. The project has produced a wealth of experience for developing games in Haskell...

[141] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008

Each dimension is a measure of success for a time frame, the first being concerned with the immediate results of the project, the fourth being concerned with the distant future and the interactions the project may be a part of in the future. This project was by no means perfect, but each dimension showed strong signs of success according to Shenhar's measurement criteria. The stakeholder's requirements were thoroughly examined to ensure no perspective had been ignored to disguise any of the projects shortcomings or failure.

5.11 Conclusions

Many successful projects experience severe problems, scheduling errors, delays and resource shortfalls, but that in no way brands the project as a failure, what truly measures the project's success is whether the the project management compensates for failure and it able to mitigate the effects. This project was carefully structured so that even under the worst possible circumstances the project could be refactored to mitigate the scope creep so the project remains true to the specification.

The agile development cycles in conjunction with the scheduling required that each week there was a stable release featuring a new component. This model transpired to be over-ambitious, and features were not released on a weekly schedule but were instead released on a bi-weekly schedule, however it's work pointing out that this reflected an error in the development cycle length rather than a delay in development, and that releases were still stable throughout the project. Communication was as effective as communication can be expected to be; sometimes colleagues think they're agreeing when they're disagreeing, sometimes they're arguing over a point they are in fact agreeing on but incorrectly expressing, communication is and will always be an issue in all types of projects. The project's risk management provided the team insight to allow extra time for dealing with problems encountered during development, and mitigate the affect on the overall project.

Some parts of the project cannot even be assessed until long after the project is finished. Often legal or social issues don't become apparent until a long time after the project is completed, most project analysis tools don't become useful until the project is long since completed by which time lessons are lost, stakeholder perspectives change, or the project becomes an relic. What's truly important above all else is to ensure that the correct strategies have been identified, the mistakes have been documented, the teams have learned a lesson and that the experience will be applied to the next project. The team, and the majority of the stakeholders, would definitely consider the project a success.

6 Overall Conclusions

HASKELL IS AN ABSTRACT RESEARCH LANGUAGE USED ONLY IN ACADEMIA, EDUCATION, BANKING, STOCK TRADING, CIRCUIT DESIGN, EMBEDDED SYSTEMS, CRYPTOGRAPHY, OPERATING SYSTEMS RESEARCH, BIOINFORMATICS, PHONE APPS, AND WEB SERVICES.
— Jafet, quoted in *Haskell Weekly News* Issue 265

Much has been learnt over the course of this project, but as always it is hard to crystallise the full advantage of personal experience into a widely accessible form. The importance of continued research into software design principles is clear: software continues to be a driving force for society and computer use is still increasing at a brisk pace,[142] yet actually writing software remains difficult and error prone.[143]

This project aimed to show the paradigm of functional programming in action, and specifically the language Haskell. What was once considered merely a research language is now finding uses in many places in industry, and has an active and numerous online community. Haskell and FP remain outside of the mainstream, however, and education and encouragement will be required before these techniques can be widely taken advantage of. This project has attempted to contribute to this agenda by demonstrating that Haskell is effective in a field thought normally to be the preserve only of C++: that of game programming.

In order to be sure this approach would be viable, various aspects of the method needed to be carefully considered, as detailed in Chapters 1 and 2. The most important point to note here is that the game, *Project Serenity*, was designed purely on the merit of being a good game that was achievable in the time frame, *and that no considerations of what would best suit the language it was to be written in*. This is crucial to the conclusions that may now be drawn.

The details of developing Project Serenity — as detailed in Chapters 3 and and 4 — have held some surprises, but the overall experience really has been like that of any other software project. Some problems were circumnavigated easily, others required some time and the invention of novel solutions. Some parts of the imple-

[142] Y. Borodovsky et al. Marching to the beat of Moore's law. In *Proc. SPIE*, volume 6153, page 615301, 2006

[143] M. Paulk. *Capability maturity model for software*. Wiley Online Library, 1993

mentation, such as the AI framework and data model, very much suited the standard functional approach, whereas others, such as the GUI framework required some thought to code in a functional way. Advantages and disadvantages to Haskell were apparent, and have been discussed. But the main finding to be stressed is that the disadvantages were small: the difference between the classical problems 'suited' to FP, and areas such as graphics and networking — that people assume would be difficult or impossible — are either small or non-existent. The hardest problems to be overcome during the project were instead from unexpected directions and generally not related to the functional paradigm itself, such as dependency problems with Cabal (see Section 4.8.2).

It is important to stress this point, and it can be seen as the most important point that this project has aimed to demonstrate. When talking about the project during the course of the development, the project team were asked frequently questions like 'what parts of the implementation really suited Haskell / really didn't suit Haskell', or 'how has the game you have designed been made to really demonstrate FP'. The point is that the game was designed simply to be a good game, and the development was *all* eminently doable with Haskell. Some areas were harder than others, but this would be the same in any language. If Project Serenity is to have demonstrated anything, it is that the advantages of FP really do exist (testability, maintainability, equational reasoning, etc) while most of the usually cited disadvantages (low performance, hard to understand, cannot handle IO well, even lack of library support) do not. It follows that the real barrier to entry to functional languages like Haskell is not a technical aspect of the language — as often claimed — but merely a lack of familiarity with the paradigm.

6.1 *The Importance of Design Patterns*

One of the main weapons brought to bear against software complexity is that of design patterns, and their importance as software development tools is well established.[144] However, because FP can be so different from OO programming, different design patterns are required; or, alternatively, a different perspective on the *same* underlying principles is required. Therefore, a big focus in the presentation of the results in this report, especially those in Chapter 4, is on design patterns for the functional programmer. This focus is to enable functional techniques to be accessible to a wider audience than the usual academic style of research into these languages normally allows.

[144] See, for example, the well renown software practice guide: R.C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

Many of the techniques used in Haskell programming — such as use of Monads, Monoids, Arrows, etc — has come about through mathematical results and reasoning. Category theory has, somewhat surprisingly, turned out to be a rich source of design patterns to be used in practical programming. While one does not need to know anything about category theory to actually make use of any

of these results, many of the best Haskell techniques discovered in recent years (such as separation of concerns with free monads) are still underused because they are yet to be translated from mathematics into the language of software design patterns. One of the main reasons for this is, of course, that these developments are very new, and that techniques like these are being invented and refined at a very rapid pace. Research normally concentrates on one particular area or design pattern without bringing the full picture into view. It is for these reasons that this project has taken the relatively novel approach of "development as research".

6.2 *Development as Research*

The aspect of this project that makes it unusual, especially for a fourth year group project, is the two layered approach. There is the game itself, which as previously stated was designed independently of all other concerns. This part of the project was a software development project pure and simple, with a customer and a product delivered at the end. But interleaved with this was the primary concern of evaluating Haskell as a development language. The novel idea here is that the first part allows for greater value in the second — value which isn't available in research that focuses only on a single use or pattern.

As this is a new and largely unparalleled idea the method is somewhat unrefined. Keeping track of the research concern while in active development can be difficult, and for this reason great care was taken to record problems or notable events during development. But it is likely that further improvements can be made if future projects of a similar nature are undertaken. It is possible, for example, that undertaking two implementations of a design simultaneously in different languages would yield further insights.

6.3 *Limitations of the Project, and Recommendations for Future Work*

The main limitation of the project has been that, due to time constraints, the complete life cycle of the game could not be completed. The development schedule that was initially planned proved to be quite ambitious, as is often the case in such projects (see Chapter 5 for more details). This has meant that the extensive playtesting that was originally planned was not able to be completed, and some of the aspects of coding in Haskell (i.e. maintenance and reuse) have not been as thoroughly tested as they might have been. Despite this, it can be said that the time available was as well spent as could practically be expected.

Once the academic considerations of sharing work before it is marked are no longer relevant, it is planned to open source at least part, if not all of the Project Serenity code. It is hoped that work on

the game will continue and eventually finished.

While the project does try and infer conclusions about functional programming in general, many of the conclusions can only be said to apply to Haskell itself. Future work could contrast and compare many different languages being used for similar projects, and what design patterns are similar and can apply more universally. For example, it has been suggested that continuation mechanisms in many languages can be used like the Haskell 'do' notation to code monad expressions.[145]

[145] D. Piponi. The mother of all monads, 2008. URL `blog.sigfpe.com/2008/12/mother-of-all-monads.html`

The content of Chapter 4 is an attempt at a 'guide' to design patterns for Haskell programming, but much more work on this subject is possible, and indeed desirable. Free monads, lenses, arrows, functional reactive programming, and so on, are all active development areas; but there is still a paucity of full guides on how to use these techniques in tandem in a large project. Indeed, a whole book on the subject would be possible.

If the advantages of Haskell and other functional languages are to be be as widely available as possible, routes to understanding them and using them from mainstream development are required. Much work could be done on comparing and contrasting functional from imperative design patterns.

6.4 Summary

It has been demonstrated that developing a game fit for the current independent market is entirely feasible using the Haskell language. The game was designed purely on the merits of being a good game, not to be well suited to a functional implementation. None of the problems usually touted against the use of functional languages for such projects were encountered, including in areas requiring heavy use of IO, such as graphics and networking. A number of design patterns were found to be especially advantageous, details of which are to be found in Chapter 4.

There are many advantages to the use of functional languages, yet they are often not used for ill-founded reasons. This project has shown that Haskell is an extremely attractive language, and suggests that it should be taken seriously as an industry tool.

Acknowledgements

The project team would like to thank Sara Kalvala and Matt Leeke for their excellent supervision of, and input into, the project. Thanks also to the maintainers of the *Gloss* library, especially Ben Lippmeier and Thomas DuBuisson who responded to and fixed a space leak bug that was reported by the team during the course of the project; Ertugrul Söylemez for the *Netwire* library; and Professor Steve Matthews for the effort involved in organising and directing all of the fourth year projects. Finally, a big thank you to Warwick Security for being sympathetic to our idiosyncratic schedules in the department's computer labs, and the staff at the Café Library for the vital supply of high quality caffeine.

This report makes extensive use of the Tufte Latex class written by Bil Kleb, Bill Wood, and Kevin Godby (`http://code.google.com/p/tufte-latex/`). The Haskell code in this report was typeset using a modified version of LambdaTEX originally created by Patryk Zadarnowski (`http://www.jantar.org/lambdatex/`).

Full Reference List and Selected Bibliography

Best selling games of 2011: Modern warfare 3 outguns the opposition, 2012. URL guardian.co.uk/technology/gamesblog/2012/jan/11/best-selling-games-of-2011.

Shocking surprises and secrets in video games, 2012. URL gameranx.com/features/id/5634/article/top-20-shocking-surprises-and-secrets-in-video-games/.

Top 15 most popular video game websites, 2013. URL ebizmba.com/articles/video-game-websites.

M. Ankerl. How to generate random colors programmatically, 2009. URL martin.ankerl.com/2009/12/09/how-to-create-random-colors/.

J. Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007.

Entertainment Software Association. Essential facts about the computer and video game industry. URL www.theesa.com/facts/pdfs/ESA_EF_2012.pdf.

R. Atkinson. Project management: cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria. 1992.

J. Aumasson and D.J. Bernstein. Siphash: a fast short-input prf. In *Progress in Cryptology-INDOCRYPT 2012*, pages 489–508. Springer, 2012.

J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10), 1999.

K. Beck. *Extreme Programming Explained*. Addison-Wesley Professional, 2000.

K. Beck. *Test-Driven Development: By Example*. Addison Wesley Professional, 2003.

Y.W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization, 2001. URL developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization.

G. Booch. Object-oriented design. *ACM SIGAda Ada Letters*, 1(3):64–76, 1982.

M. Booth. The ai systems of left 4 dead. Valve, 2009. URL valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf.

Y. Borodovsky et al. Marching to the beat of Moore's law. In *Proc. SPIE*, volume 6153, page 615301, 2006.

F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

F.P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley, 1995.

J.F. Bryan and E.A. Locke. Parkinson's law as a goal-setting phenomenon. *Organizational Behavior and Human Performance*, 2(3):258–275, 1967.

F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.

S. Chacon. *Pro Git*. Apress, New York, 2009.

B. Chavanu. The top 5 youtube video game reviewers. 2011. URL `makeuseof.com/tag/top-youtube-video-game-reviewers/`.

M.H. Cheong. Functional programming and 3d games. *BEng thesis, University of New South Wales, Sydney, Australia*, 2005.

R. Cheplyaka. Reasoning about space usage in haskell, 2012. URL `http://ro-che.info/articles/2012-04-08-space-usage-reasoning.html`.

A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33(2):346–366, 1932.

K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279, 2000.

K. Claessen and J. Hughes. Testing monadic code with quickcheck. In *ACM SIGPLAN Workshop on Haskell*, pages 65–77, 2002.

A. Cockburn and L. Williams. The costs and benefits of pair programming. In *First International Conference on Extreme Programming*, pages 223–247, 2000.

P. Collier. What makes a good game? URL `petecollier.com/?p=243`.

C. Crawford. The art of computer game design. 1984.

D. de Champeaux. Object-oriented analysis and top-down software development. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 360–376. Springer, 1991.

E.W. Dijkstra. On the reliability of programs. circulated privately, 1971. URL `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF`.

E.W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005.

T. Francis. What makes games good. 2011. URL `pentadact.com/2011-05-27-what-makes-games-good/`.

M. Freeman and Beale. *Measuring project success*. 1992.

M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

A. Groen. The death march: the problem of crunch time in game development, 2011. URL `arstechnica.com/gaming/2011/05/the-death-march`.

D. Grune, K. van Reeuwijk, and H.E. Bal. *Modern compiler design*. Springer, 2012.

L. Haddon. Electronic and computer games: The history of an interactive medium. *Screen*, 29(2):52–73, 1988.

T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

T. Hawkins. Controlling hybrid vehicles with haskell. presentation. commercial users of functional programming, CUFP (2008).

A. Hertzfeld. -2000 lines of code. 1982. URL `http://folklore.org/StoryView.py?story=Negative_2000_Lines_Of_Code.txt`.

P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen. Behavior-based acceptance testing of software systems: a formal scenario approach. In *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*, pages 293–298. IEEE, 1994.

P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(04):437–444, 1998.

B. Jacobson. Sound practices of game business and design. Valve Corporation, 2007. URL `valvesoftware.com/publications/2005/AGDC2005_SoundPractices.pdf`.

D. Karlstrom and P. Runeson. Combining agile methods with stage-gate project management. *Software, IEEE*, 22(3): 43–49, 2005.

H.A. Kautz, B. Selman, and M. Coen. Bottom-up design of software agents. *Communications of the ACM*, 37(7): 143–146, 1994.

W. Kramer. What makes a game good? *The Games Journal*, 2000. URL `thegamesjournal.com/articles/WhatMakesaGame.shtml`.

M. Lipovača. *Learn you a Haskell for great good!* No Starch Press, 2011.

Geoffrey Mainland, Roman Leshchinskiy, Simon Peyton Jones, and Simon Marlow. Haskell beats C using generalized stream fusion. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/haskell-beats-C.pdf`.

T. Malone. *What makes computer games fun?*, volume 13. ACM, 1981.

L. Martin. Top 100 best-selling uk games 2012: only black ops, fifa sell 1 million, 2013. URL `digitalspy.co.uk/gaming/news/a450921/top-100-best-selling-uk-games-2012-only-black-ops-fifa-sell-1-million.html`.

R.C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

R.C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall, 2011.

H. Maylor. *Project Management*. Pearson Education Limited, 4th edition, 2010.

S. McConnell. *Code Complete: A practical handbook of software construction*. Microsoft Press, Redmond, 2nd edition, 2004.

A. Mendelow. Stakeholder mapping. In *Proceedings of the 2nd International Conference on Information Systems, Cambridge, MA*, 1991.

K. Menger. The ideas of variable and function. *Proceedings of the National Academy of Sciences of the United States of America*, 39(9):956, 1953.

Y. Minsky. OCaml for the masses. *Communications of the ACM*, 54(11):53–58, 2011.

R.K. Mitchell, B.R. Agle, and D.J. Wood. Toward a theory of stakeholder identification and salience: Defining the principle of who and what really counts. *Academy of management review*, pages 853–886, 1997.

B. Moseley and P. Marks. Out of the tar pit. In *Software Practice Advancement (SPA)*, 2006.

M. Odersky, L. Spoon, and B. Venners. Programming in scala: a comprehensive step-by-step guide. *Artima Inc.*, 2010.

B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

M. Page-Jones. Comparing techniques by means of encapsulation and connascence. *Communications of the ACM*, 35 (9):147–151, 1992.

D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972.

M. Paulk. *Capability maturity model for software*. Wiley Online Library, 1993.

L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

F. Petrillo, M. Pimenta, F. Trindade, and C. Dietrich. What went wrong? a survey of problems in game development. *Computers in Entertainment*, 7(1):13:1–13:22, February 2009.

S. Peyton Jones. Tackling the awkward squad. Technical report, Technical report, Microsoft Research, Cambridge, 2001.

S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

D. Piponi. The mother of all monads, 2008. URL `blog.sigfpe.com/2008/12/mother-of-all-monads.html`.

F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *ACM SIGPLAN Notices*, volume 41, pages 232–244. ACM, 2006.

R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 7th edition, 2010.

S. Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002.

G.K. Rand. Critical chain. *The Journal of the Operational Research Society*, 49(2):181, 1998.

E.D. Reilly. *Milestones in computer science and information technology*. Greenwood Publishing Group, 2003.

D. Roundy. Darcs: distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4. ACM, 2005.

O. Shenhar, A.J. Levy. Mapping the dimensions of project success. *Project Management Journal*, 28(2), 1992.

N. Smallbone. Property-based testing for functional programs. Licentiate thesis, Chalmers University of Technology, 2011.

I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.

W.M. Taliaferro. Modularity the key to system growth potential. *Software: Practice and Experience*, 1(3):245–257, 1971.

A. Temba. The infinite game. 2013. URL `gamedev.net/page/resources/_/creative/game-design/the-infinite-game-r3045`.

E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering (WCRE 2002)*, pages 97–106, 2002.

Vic. A history of counter-strike, 2012. URL `lambdageneration.com/posts/a-history-of-counter-strike/`.

P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

L. Williams. Integrating pair programming into a software development process. In *14th Conference on Software Engineering Education and Training*, pages 27–36, 2001.

R.W. Wolverton. The cost of developing large-scale software. *Computers, IEEE Transactions on*, 100(6):615–636, 1974.

P. Wyatt. The making of warcraft part 3, 2012. URL `codeofhonor.com/blog/the-making-of-warcraft-part-3`.

E.Z. Yang. Anatomy of a thunk leak, 2011. URL `http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/`.