CS407 October 2012

Haskell as a Game Development Platform

Specification

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

Supervisor: Sara Kalvala

Client: Matthew Leeke

The Department of Computer Science
University of Warwick

Contents

1
Introduction

IF YOU AREN'T SURE WHICH WAY TO DO SOMETHING, THEN DO IT BOTH WAYS AND SEE WHICH WORKS BETTER.
— 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 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, helped in part by its compatibility with the Java Virtual Machine (JVM) and object oriented design.

One of the 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.[6] However, it is not a given that any program would 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. And with modern machines getting ever faster, the domain of problems that require high levels of efficiency is getting smaller.

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

Performance problems alone cannot account for the fringe position of functional programming. The efficacy of the functional approach has been touted for many years,[7] yet it is still rare for mainstream projects to make any use of functional languages.

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

Instead of researching and discussing the theoretical advantages of the functional paradigm, this project will attempt to demonstrate the value of functional programming by utilising it in a problem domain that should pose a significant challenge that is not normally considered a 'good' domain for FP. The chosen application for the project is a game.

1.1 Why a Game?

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.[8] A game is also a tangible, sizeable piece of software, yet achievable for a four person group over two terms.

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

As well as demonstrating FP over a wide range of areas, a game also represents a serious business venture.[9] 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 big companies, down to indie companies and fan groups.

[9] Entertainment Software Association. URL `www.theesa.com/facts/pdfs/ESA_EF_2012.pdf`

This project will certainly not be the first game ever to be developed in a functional language, nor is it likely to be the last. The project shall therefore not only deliver the finished game, but fully document the process — explaining what went well and how the functional approach benefited the construction, as well as what proved challenging.

1.2 Picking the Language

When considering the choice of functional programming language, Haskell emerged the obvious choice when considering the following factors: —

Concision Haskell code is concise, yet readable. This is a very real advantage as it allows both for fast writing of code, as well as fast refactoring and maintenance.

Purity Haskell has all the advantages of being a pure functional language. However side-effects and mutability are needed for real programs (especially games) and Haskell has excellent tools for solving these problems, via the IO Monad, State monads, etc. The 'do' syntactic sugar makes Haskell one of the best languages for this.

Speed Haskell has very good compiler support, and the Glasgow

Haskell Compiler (GHC) is capable of producing highly efficient code.

Type System Haskell has a very advanced type system, which makes bugs and errors in refactoring easy to detect quickly. Automatic type inference allows for these advantages without the type system slowing down the programmer.

Testing The purity of Haskell enables automated testing techniques not possible even in other functional languages. There are also well supported testing libraries available for both pure and impure code.

Community and Library Support The Haskell community is very active and there are extensive libraries available for it. Due to Haskell compiling to C, most C system libraries have Haskell bindings. There are OpenGL and OpenAL, for example.

Familiarity All members of the group have some experience with Haskell, and consider developing with it to be very enjoyable.

1.3 Existing Systems

To avoid confusion we shall consider separately research into functional programming for games, and what game we will actually make given current and historic trends in gaming.

1.3.1 Existing research into Functional Programming of Games

Probably the most well known game written in a functional language (Haskell, as it happens) is *Raincat*,[10] written in Haskell and developed by Carnegie Mellon students in 2008. There is also a game company, *ipwn studios*, who exclusively use Haskell for their products.[11] Despite this, there is little work on the academic research side that supports or opposes functional languages for games.

There was a similar project in 2005 by Mun Hon Cheong, an undergraduate at the University of New South Wales;[12] and though Cheong did manage to create a complete 3D game and gave detailed descriptions of some of the code techniques, the project did not provide a detailed insight into what exactly was effective, or challenging, about the use of a functional language.

The fact that an exhaustive search of the literature in this area revealed only a single undergraduate dissertation highlights the paucity of available research in this area, and coupled with the growing interest in functional languages for game development shows the case for this project.

[10] Source available online from `raincat.bysusanlin.com`.

[11] See their website: `ipwnstudios.com`.

[12] M.H. Cheong. Functional programming and 3d games. *BEng thesis, University of New South Wales, Sydney, Australia*, 2005

1.3.2 Existing Games

Needless to say, the complete history of gaming, even if only restricted to computer games, would be too lengthy to examine here.

The gaming industry has grown hugely since the early commercial computer game systems, in parallel with the huge developments in the capabilities of computers themselves. And while some games today are magnificent technical achievements with breathtakingly detailed graphics, sounds, and physics engines, there are many popular titles from indie game companies using only 2D graphics and simple engines enjoying success. It is not just the cutting edge of technology that can make a game fun.[13]

[13] This is a complex issue. For a more complete treatment see T. Malone. *What makes computer games fun?*, volume 13. ACM, 1981.

In order to serve the purposes of the project, it will be necessary to create a game designed to compete in the current market. This doesn't mean it has to be as complete, complex, or technically sophisticated as a triple-A title, but it does have to be a game that, if fleshed out fully, would be considered fun and suitable for a small company or similar organisation to sell. Without this any conclusions drawn about the efficacy of the approach will not be sufficiently valid to developers.

For this reason it is worth briefly reviewing a few games — some modern, some less so — in order to identify what would be an appropriate brief for a game to help achieve the project's aims.

Figure 1.1: A screen from *Ultimate Doom* (1995).

An early game that was hugely successful, as well as controversial, is *Doom*, a first person shooter developed by John Carmack and John Romero of id Software and released in 1993. Doom was marketed using a shareware model — the first third of the game being distributed for free, and the rest available for purchase. Doom represented a revolution in what was possible in a computer game, and is widely accepted as the game that popularised the first-person genre. The slickness of the graphics engine, the thought provoking levels and puzzles, the controversial satanic imagery, all contributed to Doom's success. But arguably the greatest innovation with the most effect on future gameplay was its multiplayer modes. Over modem or local serial connections, players could join forces to complete the main game, but could also battle each other in violent showdowns, coined *deathmatches* by Romero. The name stuck.

Even though Doom is now very old, and the graphics look hugely out of date, it is still relevant to consider just how successful the multiplayer model in Doom was and still is. Battles are short, skilful, and highly addictive. Carmack and Romero predicted that Doom would become the number one cause of non-work in offices across the world, and they were right. Especially given the limited time constraints of the project, and the amount of writing time single player plot lines can involve, a compelling multiplayer mechanic would be a sound basis for the new game.

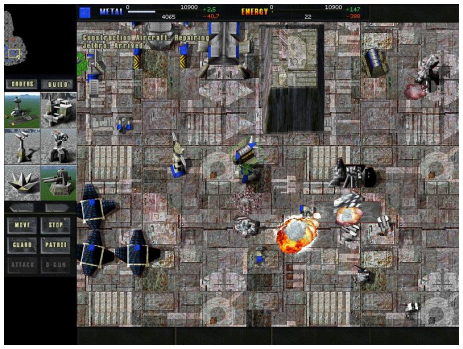

Figure 1.2: Total Annihilation screen showing several unit types and some wrecks.

A game hugely influential in the realm of real-time strategy is *Total Annihilation* (TA), released by by Cavedog Entertainment in 1997. The reception to TA was extremely positive, and the game is still actively played to this day. TA was notable for an advanced resource system that required careful balancing, and

Faster Than Light, a turn based strategy game with real time

strategy battles, is set in space, with the player taking command of a ship with the objective of getting to the other side of the galaxy.

The ship has an upgrade system that allows the player to upgrade various systems/parts of the ship giving advantages in the next battle, giving the player an incentive to battle, since winning a battle grants currency. The ship has energy that needs to be allocated to the various systems such as shields, weapons, and life support. This gives the player a greater control over their ship, whilst allowing them to tweak the capabilities of the ship during battle, ie temporarily dropping life support to boost their weapons. During battle, the player's responsibilities can vary completely from having to do nothing to having to pause the game every few seconds to calculate the next optimal move. Having a varied level of responsibility adds to the dynamic gameplay, however this game varied too much, ranging from complete boredom to a single battle taking much longer than it should. A multiplayer mode was missing from the game, causing the gameplay to become predictable, and not replay-able.

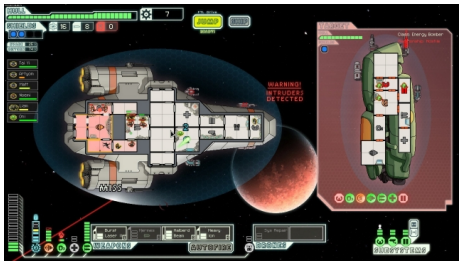

Figure 1.3: Faster Than Light.

Sins a of Solar Empire was a futuristic real time strategy based in space, with the game world modelling a graph of planets, where the player could only move ships between planets that were connected. The objective was to wipe out the opposing faction(s) by destroying their ships and capturing their planets. The gameplay didn't have many twists to the outcome of a battle, resulting in the dominant player continuing to gradually take ground, resulting in very long and boring gameplay. Due to the layout of the world, certain planets would become bottleneck where the majority of battles occurred. A race would ensure to capture these planets that would become the bottlenecks, adding to the player's overall strategy. The downside of this was that it was apparent who would likely win based on who had captured these bottleneck planets. The layout of each game was procedurally generated, making each game unique and greatly improving the replay-ability factor for the game. A resource system existed that had 3 resources: Credits, Metal, and Crystal. These resources were only used for the building of fortifications and ships, and would not effect the outcome of the battle directly.

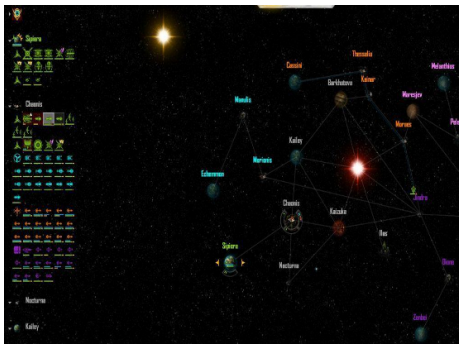

Figure 1.4: Sins of a Solar Empire.

Mech Commander 2 (MC2) is a real-time tactics video game developed by FASA Interactive. The game features simple 3 dimensional graphics and special effects for it's time, but proved to be very popular due to it's unique game style. MC2 allowed the player to deploy a number of mech units (constrained by a maximum weight capacity) to the battlefield. Once on the battlefield the player was vastly outnumbered by enemy mechs and had to rely on stealth and superior tactics to complete a number of objectives. Higher priority objectives were more heavily defended, so the play had to invest more time devising strategies, but received a higher payoff for the mission when they completed the higher priority objectives. What made MC2 particularly interesting was the deploy-

Figure 1.5: Mech Commander 2 customise mech screen. Components can be dragged onto the mech from the weapons cache (not shown), heat capacity and armour is shown below the mech.

ment phase, in which the player had the opportunity to customise mechs with weapons, armour, and system upgrades (e.g. sensors and jump-jets). Mech customisation allowed the player to have full control over their strategies by outfitting mechs to meet their requirements on the battlefield (e.g. Designing a lightweight mech with jump-jets to bait heavier guarding mechs into an ambush).

Gratuitous Space Battles (GSB) could be described as a tower defence game, where the player fully customises a fleet of ships prior to a skirmish but then has no influence over the battle once it begins. Unlike a tower defence game, the objective is simply to eliminate the enemy fleet. GSB provided a very elaborate ship customisation screen, where each component added would affect a large number of the ship's properties (mass, heat capacity, crew required etc.). Although low level unit customisation has proved very popular, GSB took a great risk in scaling the customisation up so much and removing the ability to influence the battle. Overcomplicating the ship building interface led to information overload for new players, making it difficult to understand how many small changes compound and affect the overall design. The lack of control over ships meant boring, drawn out battle scenes. Perhaps one lesson to take from GSB is to keep and customisation interfaces simple, and give players free reign of control as much as possible.

Figure 1.6: GSB module selection screen.

1.4 *Methodology*

After a considerable number of different ideas were examined and roughly prototyped, a game concept was settled upon. The chosen genre is real time strategy between spaceships, with the emphasis on fast battles that last no longer than half an hour. Differing levels of control are envisaged, where the single flagship might require a high level of detailed control, other ships are given more general orders that an AI then interprets and acts upon.

It is felt that this is a game concept that could compete in the current market and has the potential to be very enjoyable. The main part of the project will be development of this game, culminating in user testing, and a full and detailed write up. However it is essential to the aims of the project that the nature of using a functional language is properly captured in this final report. To this end it is planned that during development each programmer will keep a 'diary' of code wins/challenges. Any observations about the method need to be captured immediately as they happen, otherwise they won't be remembered with enough clarity at the end to form truly useful insights.

It is hoped that there will be enough material gathered during the development that it can be combined in the final report into a kind of guide to game programming in Haskell. This could form the basis for a much larger piece of work to guide functional development of games or similar applications. Also there will be identified a wish list for future libraries and features available to

Haskell programmers.

A significant amount of research has gone into these decisions, which is partly presented later in this document. The final report will give a much more detailed examination of these issues.

1.5 Legal, Ethical, and Social Issues

One potential legal issue faced by this project is the use of third party software. It must be ensured that any third party libraries included in the code are licensed appropriately. This means only using software with a permissive license (e.g. Apache, BSD, or MIT licenses) and no unlicensed proprietary software.

Game publishers such as Electronic Arts, and Lion Head, consider their games as intelliectual property and copyright their games. It is infeasible to check that all previous games published do not bear great similarities which could result in a court case.

Games can be highly addictive, resulting in their players investing many hours into the game. If the game is pay-to-play, this can also result in large sums of money invested into the game. The game being developed uses a fixed length campaign of 5 battles, resulting in convenient game play periods for the user to quit the game.

The game is based in a fictional world, with a non-realistic graphical representation of this world. Current issues with games, such as racism, and violence are not an issue with this game.

The game will only support the English language. This prevents users who can't read English from using the game.

1.6 Working Title

Every game needs a working title. *Serenity*, has been chosen as the working title for the game to be produced as part of the project, and the project as a whole has been codenamed *Project Serenity*. As the game becomes more fully developed a more appropriate title may be substituted.

2
Project Requirements

RIEN N'EST PLUS DIFFICILE, ET DONC PLUS PRÉCIEUX, QUE D'ÊTRE CAPABLE DE DÉCIDER
— Napoléon Bonaparte

Here presented are the aims, endpoints, controls, methods, schedule, and every relevant detail of the project; laid out unequivocally for reference before, during, and after the required work.

2.1 Objectives and Deliverables

The first objective of the project team is to produce a playable prototype of the game specified. The prototype can be extremely basic, provided the playing experience gives a promising outlook for the final release. The prototype release is both an internal motivating factor and a requirement for the progress report at the end of term 1.

The progress poster is the first external deadline, and is a customer requirement for ensuring the project is on schedule. The progress poster will contain technical detail of the project progress to date, as well as screen captures of the latest running game prototype.

The final game should be completed to the requirements laid out in the specification and to amendments agreed on by the customer. 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).

The presentation and final report are the last deliverables of the project. They give the team the opportunity to deconstruct the project's progress over the academic year, and give an analysis of the finished product. The report documents the design approach taken, relevant research which influenced the project, quality standards review, testing, user manuals, and any critical decisions made during the project, and the presentation will give the project team a chance to introduce the game, and offers the customer an opportunity to question the team.

Deliverable	*Deadline*
Prototype Game	Dec 6th 2012
Progress Poster	Dec 6th 2012
Final game	April 25th 2013
Report	April 25th 2013
Presentation	May 7th 2013

Table 2.1: Summary of Deadlines.

2.2 *Requirements*

2.2.1 *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.

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 play over the internet, as latency and packet loss issues will be less intrusive.

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.

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, the latter also split into the areas on the ship it can be installed on. System slots allow for extra internal systems to be added to the ship, for example extra shields or long range scanners. Weapons slots allow the player to choose which types of weapons their ships will use; however, a fleet budget will prevent a user from using the best weapons on every ship. See figure 2.1.

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 any 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 used by the more powerful weapons available.

Planets generate a constant supply of resources, so players can gain extra resources via planetary capture. The resources generated by the planets owned by a player feed into that player's global stockpile. Individual ships then draw resources from this stockpile.

Planetary Capture Planet ownership is the only method of generating resources. Planets can only belong to one player at a time at most. To capture a planet, a player must have their ships in control 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.

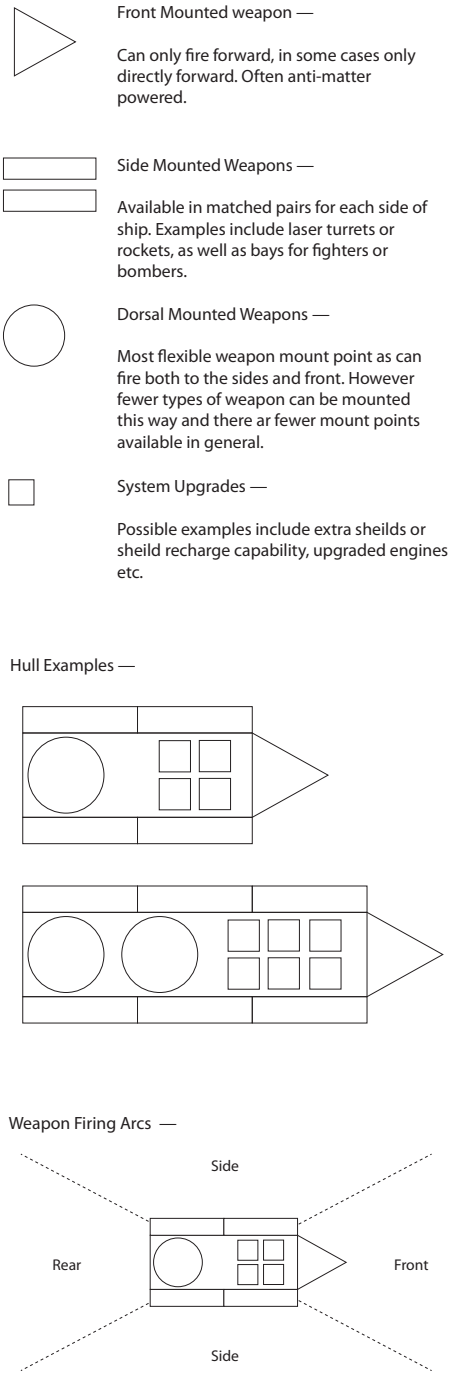

Figure 2.1: Diagrams showing basic conception of ship customisation and weapon configurations.

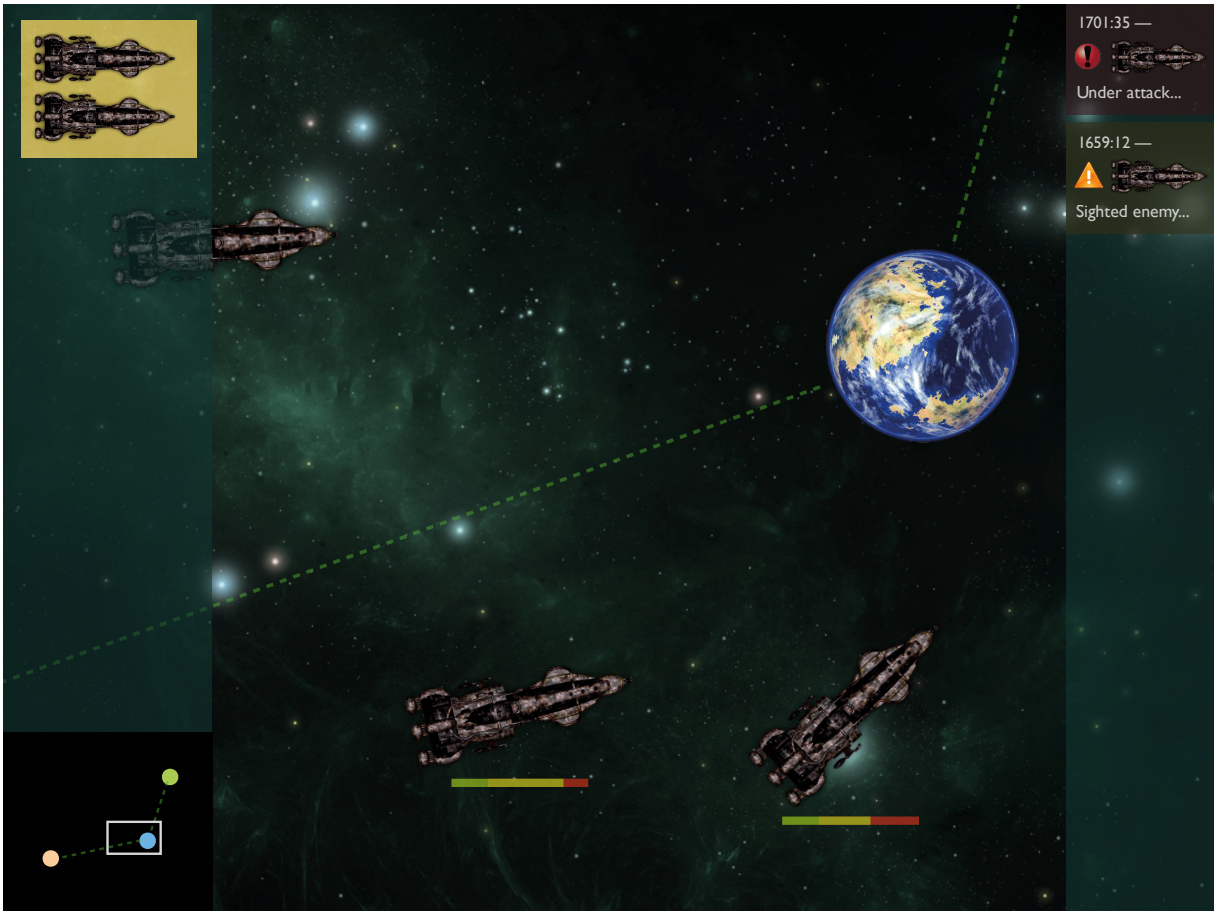

Figure 2.2: Initial mockup of a screen during gameplay, showing a planet, navigation lines, selected ships, minimap, and reports from other ships.

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.

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.

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.

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.

2.2.2 *Non-Functional Requirements*

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.

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 possible optional features that could extend this to provide inbuilt support for longer playing sessions.

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.

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

2.3 Design Approach

Two methods were considered for our design approach: Top Down, and Bottom Up. These are generic strategies for prioritising what to design first, both have their trade-offs, but can be hybridised to suit the project at hand.

Bottom Up typically starts with existing software modules that are integrated together to achieve a grander system.[1] Since the existing softwares provide the needed functionality, the majority of the implementation stage is piecing these software products/modules together to form a cohesive product. This approach falls short, when the software has to be written from scratch, since its focus is more on integration of existing products.

[1] H.A. Kautz, B. Selman, and M. Coen. Bottom-up design of software agents. *Communications of the ACM*, 37(7): 143–146, 1994

Top Down is used to design a system from scratch. Its focus is on simplicity, only breaking down one aspect of the design at a time, into its smaller sub components.[2,3] The design stage will continue to expand the subsystems of this design, until the subsystems begin to overlap with the implementation level, at this point, all further subsystem nodes are expanded at the implementation level.

[2] G. Booch. Object-oriented design. *ACM SIGAda Ada Letters*, 1(3):64–76, 1982

[3] 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

A hybrid approach will be used between the two design philosophies, at the design level, the entire product will be designed and implemented by the team, however at the implementation level, third party software products will be decided upon before the implementation stage, and will need factoring in to the design.

When designing the game, key aspects of the game are designed first to meet the requirements of the game. For instance the AI used within the game, the role of the player, how multiplayer will work. When design conflicts arise later down the timeline, the initial high priority decisions will take precedence, allowing the conflict to be resolved whilst not compromising the requirements of the game.

2.4 Quality Controls

Quality controls are a set of methods that allow the product to be tested against the specification, identifying cases in which the product will not meet the specification. Quality control can be automated for requirements that test the behaviour of the product, such as functional requirements. Non-functional requirements, which specify the the qualities that are required for the project to achieve a desired behaviour, are generally not quantifiable and therefore cannot be automated.

2.4.1 Unit Testing

Unit tests are used to test small, individual units of source code and ensure that the code meets its intended design. Unit testing is a very useful practice because it helps catch errors in code and allows a developer to refactor code by ensuring that it continues to work as before.

Unit testing is very well supported in Haskell. There is a library called *QuickCheck* that can take advantage of referential transparency in order to create random test instances. There is also more standard unit test support with *HUnit*. Together these allow easy to write, highly effective testing strategies.

2.4.2 *Component Testing*

Component testing is similar to unit testing, but focuses on larger pieces of the source code; it is used to test the integration of a number of units. Component is useful to ensure that the small units of code, which, through unit testing, are known to work individually, work together correctly.

2.4.3 *Continuous Integration*

Unit tests are only really useful if they are run regularly. Doing so allows developers to catch bugs as soon as they are introduced. This is useful because, as van Emden and Moonen note, the cost of fixing a bug is much lower if it is discovered earlier in the development cycle.[4] Therefore, a continuous integration server will be used to perform a full build and test of the software every time a new change is pushed to the source control repository. If the test suite fails then project members will be notified via email. This means that it is easy to see which change caused a failure and the team can quickly fix the problem.

[4] 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

2.4.4 *Playtesting*

Playtesting is the highest level of testing, where unit testing was testing the smallest common element of the game, Component testing was testing the interactions between a subset of the components, Playtesting is testing the built game, with all the compoents. The tests are still aimed at finding bugs, however at this level the only way to test the game is through the VDU(Visual Display Unit) and speakers, which cannot be automated. It is much easier to identify bugs at this level, since all low level bugs are propagated up to this level, however it is very hard to identify the location of these bugs, which is why the other testing methods are used. Performance is an aspect included under playtesting, since it can effect the outcome of a siutation within the a real time game. Playtesters will try different hardware that is considered within the game's minimum system requirements, to elimiate hardware specific bugs.

2.5 *Success Measurement*

The various aims and goals of the project need to be measured for success individually to ensure that the project is successful overall. The following measures will be used.

2.5.1 *Stage Gate Model*

The project is organised into separate phases with defined control points, referred to as *stages* and *gates*.[5] The gates ensure that each phase must be completed to an acceptable level of quality before the project can continue.

The precise measures used at each gate are described in section 2.4, but are mostly combinations of acceptance tests and feedback from playtesters.

[5] This is the stage gate model, see D. Karlstrom and P. Runeson. Combining agile methods with stage-gate project management. *Software, IEEE*, 22 (3):43–49, 2005.

2.5.2 *Acceptance Testing*

Acceptance testing is a tried and proved method for making sure project deliverables are of the required quality.[6] An acceptance test is carried out on behalf of or by the client, and aims to determine if the product has met a certain requirement. A full suite of tests should aim to cover everything that is relevant to quality of a deliverable. The use of acceptance tests in this project is to be foremostly at the gates (see above), ensuring that sufficient quality is reached before the next phase of the project can be entered. This both assures quality and avoids over spending in the case the gate cannot be traversed.

The precise nature of the acceptance tests are to be worked out with the client, and each weekly iteration can be run against them to monitor progress.

[6] See, for example, 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.

2.5.3 *Playtesting*

The objective of this is to improve the gameplay by poeple playing the game, reviewing it an

The objective of this is to improve the gameplay so it both meets the specification and is fun to play. Due to the intangability of "fun gameplay", a feedback loop is used which is repeated until the changes on each iteration become few or very granulated:

1. playtester plays the game
2. playtester reviews the game, commenting on aspects they like, don't like and any minor changes to the gamplay believed to improve it
3. all the changes are reviewed, compiling a change list of changes that are agreed to be added
4. the changes are added to the game
5. repeat

2.6 *Foreseeable Challenges*

There are a number of challenges that are anticipated during this project. By identifying these in advance its possible to allocate extra resources to them to ensure that the project is a success.

2.6.1 *Time constraints*

Game development projects are famous for scheduling issues that threaten to delay the release of a product. Developers often find themselves facing “crunch time", a period of extreme work overload, in an effort to deliver a game on time.[7] A survey of problems encountered in game development performed by Petrillo et al. found that two of the most common issues are missing deadlines and crunch time that results from this.[8] Although delays are a challenge common to all projects, the survey found that the need for multiple disciplines working together (programming, graphic design and music composition for example) to create a quality game causes deadline problems to occur even more frequently. These common problems have their roots in the time constraints imposed on a particular project.

[7] A. Groen. The death march: the problem of crunch time in game development, 2011. URL `arstechnica.com/gaming/2011/05/the-death-march`

[8] 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

This project has approximately twenty five weeks in which to develop a fully functioning game that meets the requirements specified previously. This is a relatively short amount of time in which to deliver a complex game. By adhering to the project management and software development techniques laid out elsewhere it is hoped that the project can be kept on schedule and the final deliverable be released on time and to specification.

In his essays on software development, Frederick Brooks argues that the complex communication structures in a team is a major cause of delays to software projects.[9] Fortunately, this project is run by a small team of four and so should find that communication overhead is less of a problem. The problem of bringing any new team members up to speed can also be ignored since this is a static team. However, the short time frame available for completing the project is still a major challenge to be overcome, but this will be mitigated by the techniques described in chapter 3.

[9] F. P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley, 1995

2.6.2 *Writing a successful AI*

Artificial intelligence can often be a make-or-break factor in determining the success of a game.[10] 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.

[10] 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.

2.6.3 *Efficiency problems*

Not only does the AI need to run efficiently, so does the game as a whole. Unfortunately the choice of a functional programming language could lead to performance issues. 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.[11] This difficulty makes it harder to develop efficient programs.

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.[12] 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.

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

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

2.6.4 *Simplicity of graphics libraries*

Some investigation into the Haskell graphics libraries available has already been undertaken. The Gloss package has been identified as a suitable candidate because it exposes a clean functional API and hides away the details of OpenGL. Unfortunately it is a relatively simple library and does not provide some required features such as windowing and clipping. This means that the behaviour required to be developed on top of that provided by the framework may be significant.

2.6.5 *Measuring success*

For commercial game publishers the main measure of success for a game is if it is profitable or not. However, this project is not a commercial enterprise and so the measures for success are not as easily determined. The ideal outcome for the project is an enjoyable game that meets the requirements described in section 2.2. Along the way it would be good to discover aspects of game development that are improved, or made harder, by the use of a functional programming language. Unfortunately, measuring these things will not be simple. However, the use of playtesters will help determine if the game is fun to play and it is hoped that development diaries will be able to track drawbacks and benefits of functional programming.

2.7 *Work Breakdown and Schedule*

The work breakdown structure (WBS, shown in figure 2.3) is important to understand the complexity of the project and how components are structured. This information is particularly useful for

agile development, so that releases can be scheduled based on a selection of the WBS, such that each component requires roughly the amount of time available in one development cycle.

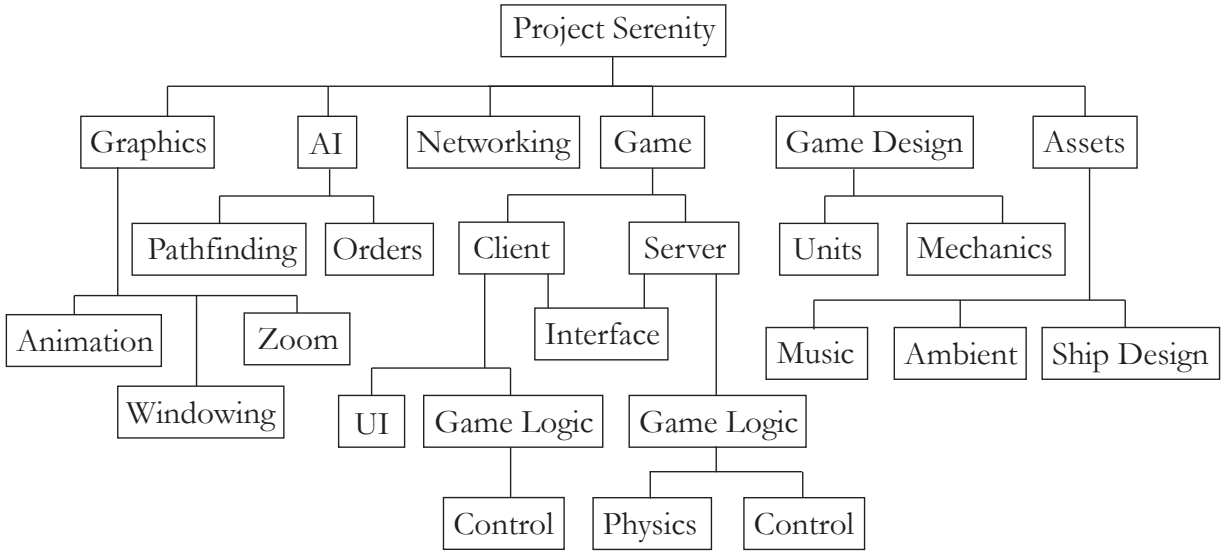

Figure 2.3: Work Breakdown Structure of the game components.

From the WBS we can further break the project down into components which are suited for weekly development cycles.

Term 2 releases will include a UI, an AI system and further improvements to the components as required. Assets such as music and additional ship designs are non-essential and will likely follow in a later release.

2.8 Schedule

The project schedule has been mapped out in a stage-gate diagram (see figure 2.6) which breaks the project down into components, showing how tasks are allocated between team members. The Alpha phase has been carefully planned as it's the most relevant to the near future of the project and the team's work hours can only be definitely allocated in the near future. The Beta and Delivery phases have been planned in less (bust still sufficient) detail, allocating enough total time to address their components, however, the specific breakdown cannot be decided until the end of term 1, when the project progress is compared to the original schedule, the reaming time is reallocated as needed. Of course a stage-gate diagram has limitations in that overlapping phases cannot be represented easily, so we use a simple gantt chart (see figure 2.5) to monitor phase transitions and ensure the project team is not mislead by a constrained representation of the schedule.

The dependency tree shown in figure 2.4 reveals which components are directly dependant on others, and show how delays will

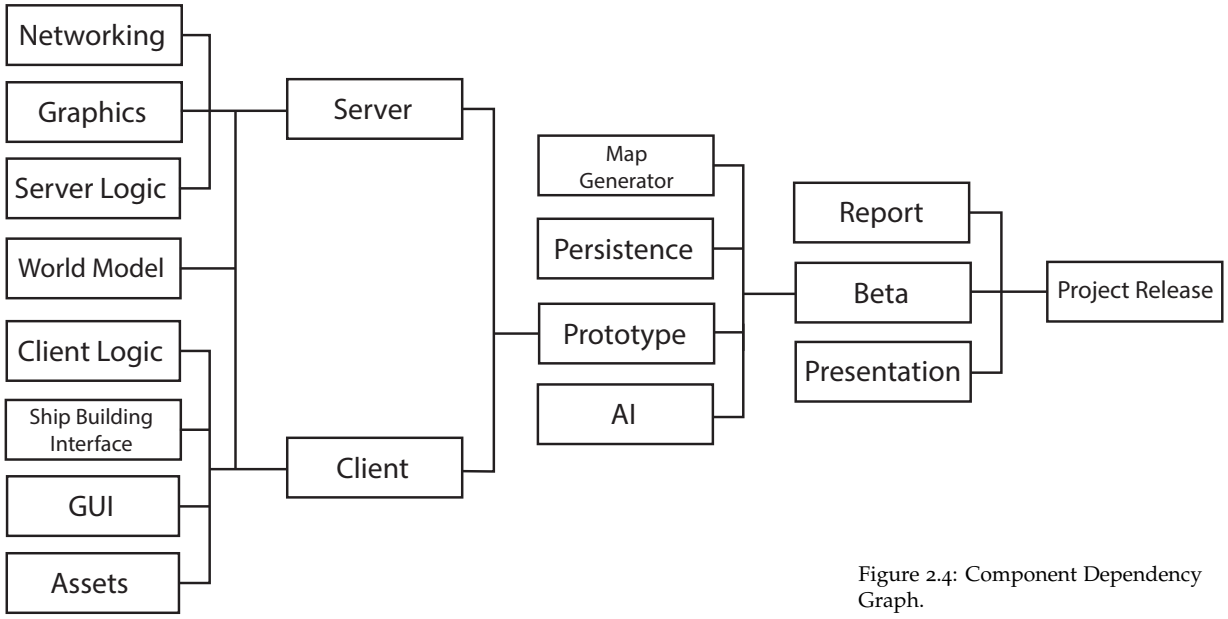

Figure 2.4: Component Dependency Graph.

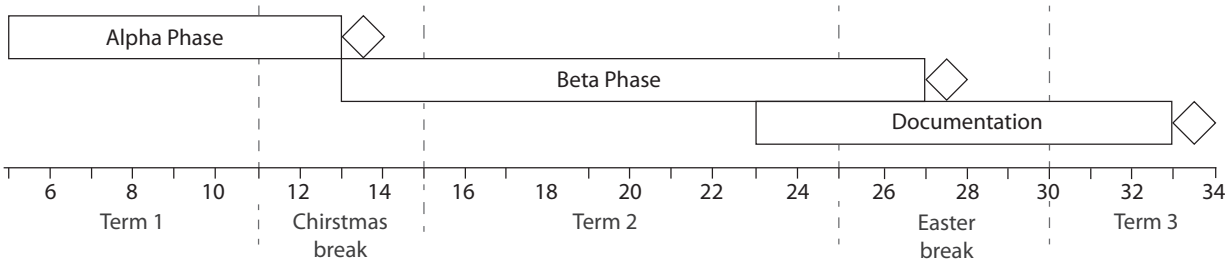

Figure 2.5: Phase level Gantt chart.

propagate through the project. The project dependency diagram shows that potential loss is minimised after a prototype is complete, that is to say there are fewer critical dependancies from the prototype level onwards. This information emphasises the priority of the server and client components should the project schedule require revising at a later stage.

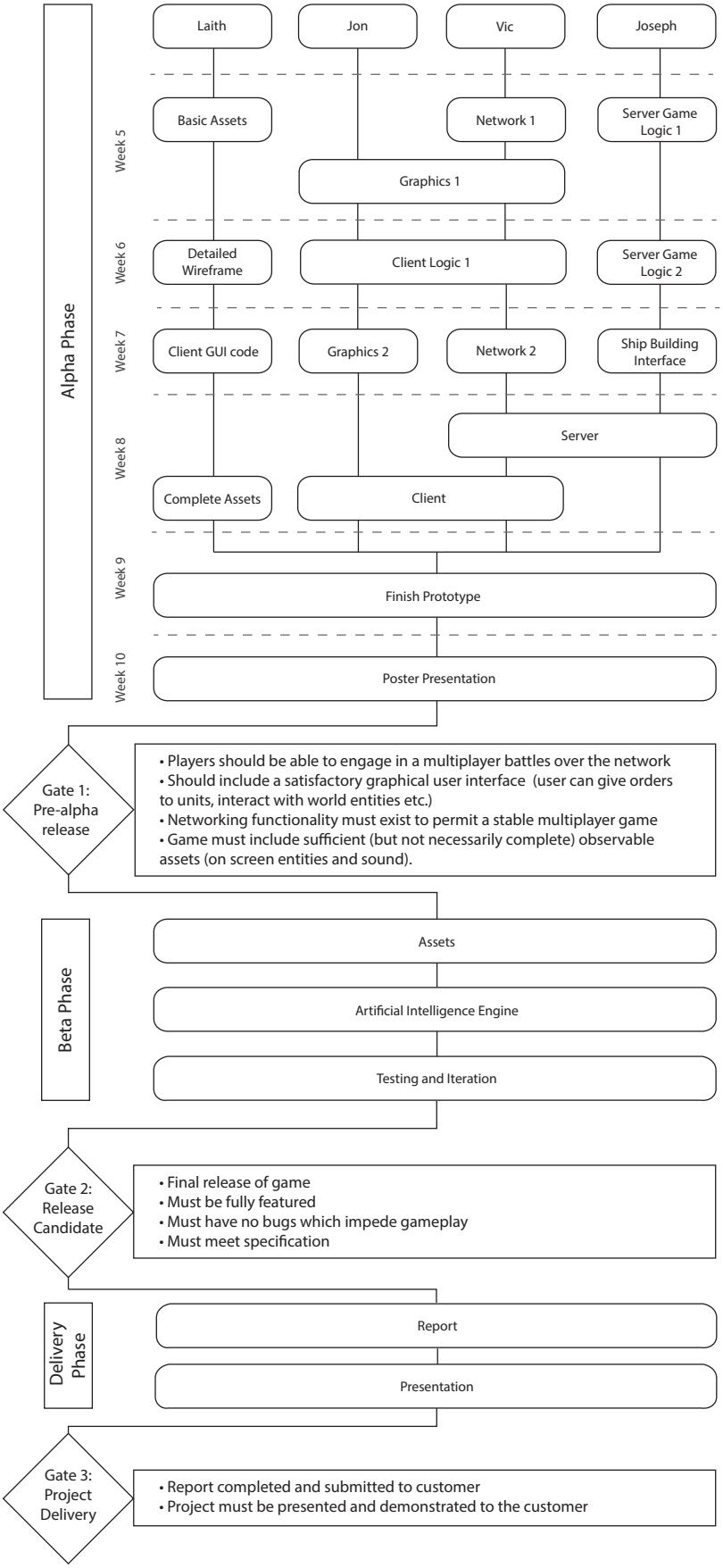

Figure 2.6: Stage-gate model of work breakdown structure showing the division of labour across the various project phases. The alpha phase is more thoroughly planned so we are able to see a weekly breakdown of tasks in the near future. The gates are project milestones, each of which details the requirements for proceeding through that gate into the next phase.

3
Project Management

"ALL THINGS ARE CREATED TWICE; FIRST MENTALLY; THEN PHYSICALLY. THE KEY TO CREATIVITY IS TO BEGIN WITH THE END IN MIND, WITH A VISION AND A BLUE PRINT OF THE DESIRED RESULT."
— Stephen Covey

Project management is often incorrectly considered to be the coordination of steps a project must move through to be completed on time, the real challenge of project management is constantly incorporating the requirements of the customer into an ever changing system to ensure that the progression of the project is always in the customers interest. A major concern of project management is to minimise wasted time and resources without compromising the quality of the project, wasting time on this project would result in a poorer quality product.

3.1 Roles and Responsibilities

Here are described the roles and responsibilities required for the project. The holders of these responsibilities at the start of the project are shown below, but it is envisaged that there are some roles that could rotate (for example Software Librarian). Some roles are to be held by a single person at a time, others may have several.

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.1.5 Common Roles

Some roles and responsibilities will be shared by the whole project team.

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

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

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

3.2 Stakeholders and Communication Plan

The various parties identified as stakeholders are shown in Table 3.1 (overleaf). The relationship between the stakeholder and the project is shown, along with a rough estimate of their power and interest.[1] This grid will form a reference for making sure that all interested parties are communicated with appropriately throughout the duration of the project.
Communication within the project team is examined in detail elsewhere in this document, so the remainder of this section is concerned with the other stakeholders.

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

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 3.1: Stakeholders for the project.

3.2.1 Supervisor Meetings

Regular communication with the project supervisor is likely to be a critical factor in success of the project. For this reason a weekly meeting with at least one member of the group if not more will be high priority.

3.2.2 Client Meetings

The client is clearly vital to the success of the project, and continual feedback on each release will allow for early identification of any problems. At least one meeting per release (ie each week) will be required, as well as further meetings and correspondence as needed.

3.2.3 *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, extended levels of communication are unlikely to be necessary. Brief updates pertaining to deliverables is all that should be required. But if the project organiser initiates communication then they should be made a high priority.

Communication with the second accessor is, for the most part, not appropriate, excepting when within the remit of the deliverables, i.e. the report and presentation themselves.

3.2.4 *Playtesters and End Users*

End users are clearly important to the goals of the project, but they will have little interest or influence in the early stages. Online updates, an email to report bugs to, and a mailing list for any events that are organised will be sufficient communication.

3.2.5 *The Haskell and Functional Programming Communities*

The overall end goal of the project is not just a game, 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 while it is running. Communication back to the community should therefore be largely via the final report, as well as the methods for end users above.

3.3 *Risk Management*

A proactive approach to risk management has been taken. This is to maximise the probability of avoiding risks instead of having to move into 'fire-fighting mode' if something goes wrong.[2]

As part of this proactive risk management strategy, a number of potential risks have been identified. These risks are shown in table 3.2 along with their estimated probabilities of occurring and impact if they do occur.

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

Risk	*Description*	*Probability*	*Impact*
Length underestimate	The time required to develop the software is 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
Size underestimate	The size of the deliverable has been underestimated	Medium	Medium
Requirements change	Large number of changes to requirements during development	Low	Medium
Ambiguous requirements	Requirements are not fully understood or misinterpreted leading to loss of development time as the specification is recreated	Low	Medium

Table 3.2: 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.[3] 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 3.3.

[3] 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 *Tools and Techniques*
Size underestimate	Detailed work breakdown structure
Requirements change	Thorough change management system, see *Change Management*
Ambiguous requirements	Thorough planning phase

Table 3.3: Risk mitigation and management.

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

3.4 *Grievance Policy*

It's imperative to document a grievance guideline to ensure that any grievance procedures are fair and impartial. Because the developer team is small, an internal dispute would have a high cost to the project, so grievance issues need to be dealt with promptly.

i) Attempt to resolve the grievance issue informally by the team manager.

ii) If the issue cannot be resolved informally, and affects the entire project group, then address the issue at the next meeting and attempt to find a resolution in a group environment.

iii) If the issue cannot be solved by a formal group meeting then a managerial confrontation is required to reduce the impact on the project.

iv) If the issue still cannot be resolved then a complaint should be made to the module supervisor and university procedure should be followed from then on.

3.5 Agile Methodology and Weekly Releases

The project team has adopted the agile development methodology, the project is broken down into smaller increments which require minimal planning and are unlikely to require long term consideration. Development iterations allow the team to regularly deliver working software, reducing the likelihood of delays going unnoticed and becoming obstructions at a later stage. The development cycle the team has selected requires a component release every Tuesday evening, making each iteration span exactly one week. Each release should include the latest completed iteration of the project, the first milestone will be the game prototype which is scheduled for release on 27th November 2012, further milestones include: beta, and finally the release candidate by 19th March 2013. The team has readily adopted pair programming, as well as test-driven development and continuous integration to help maintain code stability through multiple iterations, techniques which are strong advocates of the agile development methodology.

There are aspects to development cycles beyond the milestones and time scheduling, programmer welfare is a large focus of agile development strategies. Ensuring programmers are not overworked or lose interest in the project is an emergent factor in maintaining quality in software projects, and although the team and project managers cannot realistically limit the hours a team member spends programming over a week (mainly due to other programming assignments in parallel to the project), Wednesday afternoons and weekends have been avoided in the weekly timetable to guarantee personal time every week.

3.5.1 Standup Meetings

Standup meetings are held every week to discuss individual progress on the project, any issues an individual has been encountered, and what they will attempt to accomplish in the following week.

3.6 Change Management

As the project develops new ideas and approaches may become apparent, care must be taken to avoid blindly integrating changes into the original specification to protect the project from scope creep. The change management procedure involves assessing the viability and benefits of the change request, deciding whether the change would stop the project meeting the requirements, and if so, whether the customer and managers can reach an agreement which incorporates this change into the project or whether the change request should be rejected. The official protocol has been summarised (overleaf) in figure 3.1. Change viability is decided on the difference in cost and time required, whereas the requirements assessment is heavily dependant on the customer, and whether they think such a change would prevent the project meeting their requirements.

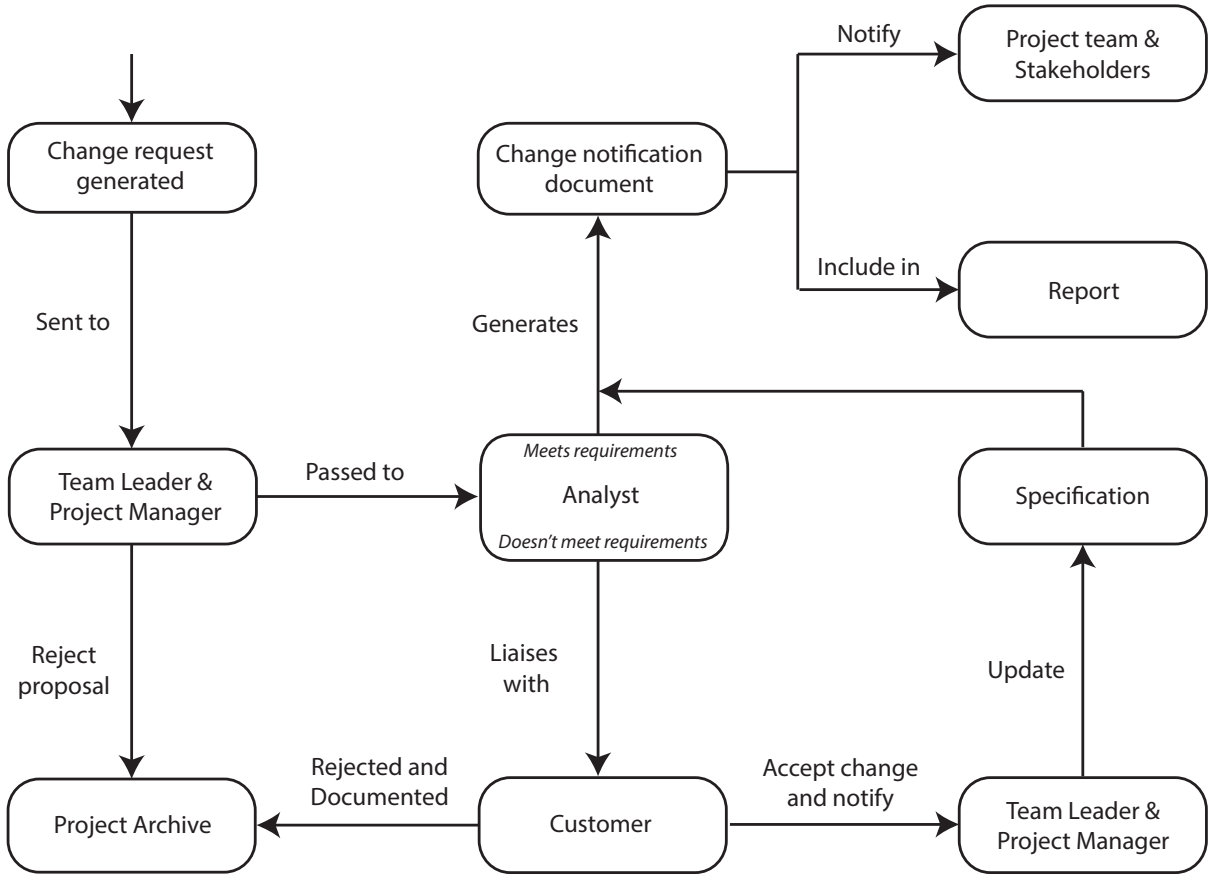

Figure 3.1: Change management workflow

3.7 *Tools and Techniques*

These are the tools to be used in the day to day running of the project.

3.7.1 *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.[4]

The Git source control system[5] has been 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".[6] 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.

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

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

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

A centralised master repository will be hosted on Github, a popular code host.[7] This is to make it easier for team members to share their changes. 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.

[7] https://github.com/

3.7.2 *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.

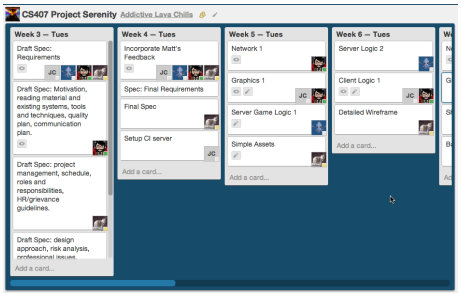

Figure 3.2: Trello

3.7.3 *Bug Tracking: Fogbugz*

Trello does not entirely alleviate the need for a full bug tracker. Fogbugz is a very fully featured service, and has the highly useful feature of automatically converting emails into bugs and auto-assigning them as required.

3.7.4 *Wiki*

Wiki pages are exceptionally useful for maintaining an informal archive of the project as it develops. The project team uses Github's wiki feature for documenting weekly progress and any issues encountered during agile cycles.

3.7.5 *Continuous Integration: Jenkins*

Section 2.4 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[8] will be used for this project. Jenkins was chosen because it is a widely used open source project.[9] Jenkins also works well with projects hosted on Github because of its Github plugin[10] and Github's Jenkins specific post-receive hook.

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

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

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

3.7.6 *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. The use of Git and Github for source control make it simple to ensure that all source code is located on multiple computers. Also, the free service provided by Dropbox[11] will be used to share and backup any other files.

[11] https://www.dropbox.com/

3.7.7 Cabal

Cabal is the Haskell package manager and build system. It helps manages dependencies and running of the test suite.

3.7.8 Whiteboard

The project team still finds it extremely useful to brainstorm and refine ideas. Therefore, meetings and programming sessions are routinely scheduled in the vicinity of whiteboards.

Full Reference List and Selected Bibliography

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. URL `www.theesa.com/facts/pdfs/ESA_EF_2012.pdf`.

G. Booch. Object-oriented design. *ACM SIGAda Ada Letters*, 1(3):64–76, 1982.

F. P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley, 1995.

S. Chacon. *Pro Git*. Apress, New York, 2009.

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.

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.

A. Groen. The death march: the problem of crunch time in game development, 2011. URL `arstechnica.com/gaming/2011/05/the-death-march`.

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.

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

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.

T. Malone. *What makes computer games fun?*, volume 13. ACM, 1981.

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.

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.

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

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.

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.

E.D. Reilly. *Milestones in computer science and information technology*. Greenwood Publishing Group, 2003.

I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.

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.

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