Browser-based Online Multiplayer Roleplaying Game

Final Report for CS39440 Major Project

Author: David Field (dvf9@aber.ac.uk)

Supervisor: Dr. Hannah Dee (hmd1@aber.ac.uk)

April 28, 2014

Version: 0.1.1 (Draft)

This report was submitted as partial fulfilment of a BSc degree in Computer Science (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature .....

Date .....

Consent to share this work

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature .....

Date .....

Acknowledgements

I am grateful to...

I'd like to thank...

Abstract

Include an abstract for your project. This should be no more than 300 words.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background & Objectives

Games are interesting projects to take on; they have a history of being difficult to make and pushing technology to its limits. There are a great many systems and features that can be included in them—AI, physics, graphics, audio, UI, multiplayer and more—and a great many ways to implement each, from simple to very complicated depending on the needs of the project.

Games also have a history of aiming for too many of these features in too little time. In most cases, every one of the features thought up would enhance the final product in some way, from a significant improvement that changes the way the game is played to a minor enhancement that makes things just a little more pleasant for the user. Many of these features may be considered mandatory for the game to be worth making at all. For example, a single-player chess game would probably not be very good if there were no AI to play against.

It is not just players who require features, however; developers of games need tools to implement the game design and a good engine to hang the design off of. A lot of game projects build custom tools that let developers and designers implement things quickly. Many game engines even provide methods for scripting and modding them after their release to players.

It is clear that, given the sheer enormity of the possible things that can be put into any one game, there is not enough time in this project to implement even half of them without a great deal of previous experience and skill. Every one of the major systems mentioned can be extremely complicated, requiring a lot of research, time and effort to make them work.

This chapter will discuss what the project is; why it was worth taking on; how a minimal system was devised that would satisfy enough of the game design requirements to be playable but also be implementable in the time given; and finally the process used to implement the design requirements.

1.1 Background

1.1.1 The Project

Before discussing the decisions made about what was doable and why it was interesting, it's useful to know what the project actually is.

The name of the project—*Browser-based Online Multiplayer Roleplaying Game*—gives a relatively good hint of the nature of the game. "Browser-based" and "multiplayer" are fairly self-evident in meaning: multiple people play together in a game hosted in the browser. "Roleplaying game" is more ambiguous. In this case, it refers to a game in the style of the classic tabletop roleplaying game *Dungeons & Dragons*.

In the context of the project that meant the following things: Firstly, there needed to be two types of players—regular players and a Game Master. A regular player plays the game as a character inhabiting the world they happen to be in. For example, they may be a dwarf in a fantasy kingdom, or a space marine on a futuristic space station.

The Game Master is a player responsible for building the world, telling the story and controlling characters that aren't controlled by the other players (known as Non-Player Characters or NPCs). Traditionally, the Game Master would also be responsible for enforcing the rules of the world. However, in this project the game was to be responsible for that instead, with the Game Master given the option of overriding or changing the rules if he or she wished to do so.

Combat is the most obvious area where the game enforcing the rules comes into effect. Combat is turn-based, with players put onto a grid and given limits on the distance they can move and number of actions they can perform in each turn. When they attempt to do something—such as attack another character or creature or escape from a trap—they have to roll dice, the result of which decides whether they were successful or not, and how well they succeeded or failed. As an example, a player failing to attack a creature with their sword could simply miss, or they could throw the sword away accidentally, depending on the result of the dice roll.

In the above scenario the dice rolls would be simulated by the game, rather than physical dice being used by the players. The game will also decide whether or not an action was valid in the first place. A player hoping to attack a creature with their sword would be unable to do so if the creature was too far away from them.

The design also called for interactions outside of combat. For instance, a player might be faced with a locked door. To get through a player could attempt to use a key they found or, alternatively, they could attempt to bash the door open with an item, such as an axe, or even their bare hands.

There are a few specific implementation details as well. Apart from the game being played via a browser, there needed to be graphics and those graphics needed to be 2D isometric tiles. Further, the server was to be written in Python with the goal of gaining experience in the language.

With this overview of the game—a full account of the original game design can be found in Appendix C—it is now important to answer why the project was worth doing.

1.1.2 Why Make The Game?

The first answer to this is that games are interesting in general. Most obviously, the final product of a game is (hopefully) something fun to play with appeal to a wide range of people. More relevant to the context of a project, however, is that games are interesting from a software perspective.

Games are made up of a lot of different parts, each one potentially being difficult to implement by itself. In this game, the most challenging individual parts are graphics and multiplayer. More important than just the individual parts, however, is making sure they integrate properly. In most cases the game needs to share data between these different pieces—game logic needs to know what an object is doing so it can perform game functions on it; the renderer needs to know what the object is doing so that it can be drawn to the screen correctly; the networking part needs to know what the object is doing so that it can forward on any relevant information to the server.

The game also offers a lot of extendibility. Given more time, more features can be added. Each feature, and fitting it together, offers a lot of potential for learning as well. For example, an extra feature could be AI, which is an interesting area in itself that offers a lot of opportunity to learn something new.

1.1.3 Reading and Research

Most of the research in the gaming world is very much a closed source, commercial affair. Being on the cutting edge of games is expensive work, requiring smart people spending a lot of time squeezing everything they can out of technology, plus a whole bunch of people to make content for the technology to show.

Even with the explosion of 'indie' games made by one person or a very small team that exist these days, most of them are still closed source and commerical. Even free games are rarely free (and certainly not open source)—existing as a vehicle for microtransactions, asking people to buy consumable cosmetic items or in-game advantages, such as speeding something up, for a small fee.

Of course open source games do exist, though they are relatively rare and not very well known. One of the open source games with most relevance to this project is Mozilla's *BrowserQuest* [9, 10, 18], a multiplayer role-playing game created in 2012 to show off the capabilities of modern browsers. It has a few properties that are useful to know for this project, the main one being multiplayer.

However, despite the theoretical usefulness of having access to the source code, it was not as useful in practice as might have been desired or expected. Without documentation that describes the more high level structure of the code it is difficult to read through and understand properly without spending a lot of time on it.

The need to have a higher level overview of structure and concepts leads to books, articles and tutorials.

One of the most popular books available for this is *Game Coding Complete* [8]. It is a lengthy book but covers a lot of things, both high level and code-specific. Most of the implementation details given are inappropriate for this project because they are for 3D graphics, C++ and cover a

lot of things like custom memory management that have no equivalent or purpose in a much more simple 2D, browser-based game written in JavaScript. However, the high level aspects of game engine design the authors talk about are very useful.

The other major book that was found to be useful in this project is a free book currently available online called *Game Programming Patterns* [12]. This book describes many common design patterns used in game programming, some being game specific and others being more well known patterns but applied to game programming, such as the classic Factory pattern. Some of the patterns when put together describe a more general structure, in a way similar to *Game Coding Complete*.

The content of these two books informs a great deal of the design that was settled on for this project, with particular emphasis on the central game loop and the event manager. These two fundamental parts of the design act as the glue for the more specific aspects of the code, like the graphics renderer or input manager.

Some other books were also investigated, mostly pertaining to the more specific implementation details of the project such as HTML5 browser games. Two books were found in this area, one which talked about implementation details in HTML5 [16] and one which did the same thing but with the addition of a focus on isometric graphics [14].

These books were less useful than the previous two mentioned, however; they didn't offer much extra for structure and the majority of the code in the project is fairly platform independent and easily transferable to other languages. Where the platform is relevant, the *Mozilla Developer Network* [7] offers up-to-date APIs that are much easier to search through than a book.

There were also a few online resources used at various development stages. Initial development in the project was started by following a tutorial [13] that allowed a base-line understanding of how HTML5 games are hooked into the browser and the basics of creating tile-maps (which will be discussed in more detail in Chapter 3). The code was all replaced but it was a useful start.

Another online resource which proved to be very useful was a tutorial on how to render isometric tiles in games in a simple way [4]. This tutorial was referenced heavily in early development to get the renderer up and running, but was later adjusted to account for more advanced rendering needs.

Finally, other than the *Mozilla Developer Network,* there were APIs referenced for the various languages and libraries used in the project, which will be introduced in Chapter 3.

1.2 Analysis

With the knowledge that the scope of games can expand to encompass a ridiculous area, and that everything within that scope is likely to be time consuming to implement, it is important to define what features are absolutely mandatory for the game to be worth making, and figure out how long it might take to implement them. This section provides a discussion of how the minimal system was decided upon, with the list of features in the minimal system being available in Appendix D.

1.2.1 Necessary Features

The first of the necessary features is graphics. Graphics are the player's view into the world, letting them see the state of the game and figure out what their input needs to be. Without this view the only way to see into the world would be to debug the code as it was playing, which is not a very efficient way to play a game and the constant pausing would also break the game's timing, which is vital for games to operate correctly, particularly in a multiplayer setting. Of course, it is reasonable to operate a game with a textual interface. However, the game design specified that graphics be implemented and that those graphics be isometric tiles.

Players will need a way to interact with the game. There are two things that affect how this works: the Game Master and the isometric graphics. The Game Master needs to be able to select different characters to control and, in theory, create maps for the other players to play the game in. Using simple keyboard controls probably wouldn't work very well in this case. Isometric graphics also cause issues with keyboard controls. Players attempting to move their character using, for example, the arrow keys on the keyboard might find themselves confused as to which direction their character will move.

The solution to both of these is mouse-based interaction. The Game Master will be able to click to select characters he wants to control and place tiles and items into the world, and movement will be a case of clicking on the desired location and letting the character move there by itself, so that there is no confusion about direction.

The disadvantage to this is that mouse interaction is more complicated to handle than keyboard interaction. With a keyboard you check whether a key is pressed and perform an action based on that. With a mouse you have to work out the location of the cursor and what else is in that location. Further, mouse-based movement requires pathfinding, which ultimately adds another required feature.

The game also requires multiplayer. All interactions in the game happen between players, be they regular players or the Game Master. Without any way for them to interact the game is relatively pointless to play—all you'd get as a player is being able to walk around a map with your character.

Multiplayer implies a few things. The first is that there needs to be a way of clients communicating with some remote version of the game. This could, in theory, be a peer to peer connection. However, there are problems with this. The first is that peer to peer is not such an easy thing to support technologically with a browser, as the web operates on a client-server model.

The second problem with peer to peer is that there is no trustworthy version of the game. All the players need to have a view into the same world and the underlying data needs to be consistent between them or problems would occur. There is nothing to stop any one of the players from modifying their client to perform actions they shouldn't be able to do, and it being run in a browser using JavaScript makes this even more of an issue as all browsers come with easy access to JavaScript debuggers. A single client could be made the authoritative version (with the Game Master being the most logical choice here). However, if that client is tampered with it would affect the game for everyone.

A server solves both of the issues that peer to peer represents. It is, of course, more natural

for a browser to operate in a client-server manner. A server is also far more trustworthy than any individual client, and can be used as an authoritative base for the game so that, even if a player tampers with their client, the other players won't be affected.

The disadvantage to a server is that it requires resources and time to host and keep running. The more people who decide to play the game, the more server power is required. If the servers went down, no one would be able to play at all.

These things represent technical requirements as much as game design requirements. Their existence requires consideration in designing the structure of the game code. However, there are some other features required which could be considered purely game design issues.

The first of these is combat. In this case, combat is what makes the game a game. Without it players really only get to walk around a world together and look at how pretty it may or may not be, depending on the quality of the graphics used.

The full design of combat is fairly comprehensive and involves a lot of possible things. In the game design overview the idea of levels of success or failure was introduced, with things like throwing away a weapon if a character fails really badly on a dice roll. The game checking for valid options was also required, with the example given being whether a player was close enough to hit another character with a weapon. However, in the full game design there is also the option for ranged attacks using projectiles.

These options all make the game a lot more interesting. However, things like being able to throw away a weapon, or even just having levels of failure, require extra implementation time. It was decided that the project only needed to implement a very basic version of combat with other things considered extras to be done given more time. This basic version of combat was binary success or failure based on dice rolls; only 'mêlée' combat, requiring characters to be next to each other; and no items would be implemented in the base game, so no weapons and all combat would essentially be unarmed.

This would allow players to fight in the game, which is an important aspect of the game design, without creating too much work adding extra features that need a lot of development time and testing, such as projectiles or the use of items.

The existence of combat also implies the existence of attributes in characters and possibly other entities that allow them to interact in some way. In this case, the design specification calls explicitly for attributes called Hit Points (HP) and Mana, which represent the number of points of damage an entity can take before it is destroyed and the number of points it has to spend on magical spells respectively. Because magic would not be implemented in the minimal system, only HP and a way to affect it need to be created.

There should also be some way of interacting with the world outside of combat. Ideally these interactions would be almost limitless and allow the Game Master to specify even more based on items they add to the game. However, this is also a huge task. Because of the time consuming nature of adding interactions to the game only a few very basic ones should be included into the minimal system.

Finally, the last feature that is considered to be required in a minimal version of the game is

a chat system. This is important because the game is multiplayer and operates through players interacting with each other. It is unreasonable to rely on players to have a third party tool available for communication, so some way for them to communicate must be provided by the game. In this case, a simple chat system is the most logical choice as it is not too difficult to implement. The basic version of the chat will be global for the game but an enhanced version could allow local chatting between players who are in a specific area or private messaging between the Game Master and a player if the GM wants to do something special.

1.2.2 Unnecessary Extra Features

Given either extra time in the project or unexpected time available at the end of the project it is useful to have a list of features that could be added. Of course, ultimately it would be nice to include everything that the original design specification calls for, but it is also unreasonable to expect that enough time would ever be found to implement all of them. Therefore, the extra features should be given priority based on how worthwhile they are to implement on top of the minimal system.

Improving combat gains the highest priority because it is the most important part of the game outside of letting the players communicate with each other. The addition of items would come first, as players generally expect to be able to equip weaponry and defences like armour. After that, ranged attacks, such as using a bow. Magic would be the final part added to the game because it requires extra attributes to operate correctly and could have a wide range of effects, such as teleporting a user or causing a fire.

The next most important thing is to have a much wider range of interactions outside of combat. Supporting the ability for players to knock down doors or use a key on them, as was given as an example in the game design overview, enhances the game considerably from a player's point of view, opening up a lot of gameplay options. It may also be reasonable to include the use of weapons and magic outside of a combat context to interact with things, such as perhaps using magic to set a door on fire.

There are plenty of other features in the full design that could be implemented, such as voice chat or the ability for players to upload custom graphics. However, the extra features listed here represent a large enough amount of work that it isn't necessary to prioritise everything else.

1.3 Process

With a minimal system decided on the task becomes to decide how the time available is to be divided up between each feature and the systems necessary to support them.

The original process chosen was a custom variation of SCRUM, with many of the roles and tools taken out due to the project having just a one-person team. Primarily, this was chosen due to familiarity. There were some potential advantages to the process other than this, however: a minimal system was known and the functionality of each feature was well described from an end user's point of view, which translates well to the idea of SCRUM user stories.

The issue with this process was that, although the features were well understood in regards to their intended functionality, there was no complete understanding of how to implement them. This meant that a lot of time was needed to prototype and gain understanding of the problem area and what would be required to solve it.

The biggest example here was the structure of the code—the game engine that holds all the discrete features together and allows them to share the data they need to work. Initial research had come up with very little of how to realistically do this and much time was spent in the beginning of the project implementing, gaining understanding and then scrapping and reimplementing this fundamental part of the code. Indeed, this pattern would prove relatively true for all the features, though less extreme.

The issue this caused was that underneath each story in the SCRUM process are specific implementation tasks that are tracked. Because the implementation kept changing as the problem became better understood and the solutions improved, these tasks quickly became out of sync with what was actually occurring. The tasks began to follow the implementation, rather than the implementation following the tasks.

To deal with this wasted time, the process was transformed to do away with specific tasks. Rather, the already existing feature list was simply taken and followed in order of priority, with priority based on the understood difficulty of the feature—an area where previous research was more useful and accurate. Features were assigned to a sprint and considered to either be complete or incomplete. The completeness of a feature was decided based on its description in the minimal system specification.

This simplification dramatically reduced the administrative overhead of the process. There remained a way to keep track of what features were done and were still to do and gave a clear idea of what the current task was overall, without having to spend time writing down what the tasks specifically should be in detail, letting the dynamic and evolving nature of the implementation happen freely. The disadvantage was that there was no way to clearly point out how complete a feature was to someone else. This disadvantage was not a huge issue because the team only consisted of one person, who knew what rough level of completeness the active feature was at anyway. However, were the team to become larger this process would need adjustment to communicate task progress more effectively.

Chapter 2

Design

PUT INTRODUCTION HERE

2.1 Tools

The nature of the project means that there are limitations on the languages that can be used to implement it, which means that the tools and frameworks that can be selected are limited as well. The first of these limitations is the browser, which has only one language capable of being run natively within it, and a few plugin options, with the Adobe Flash plugin being the most popular and providing the ActionScript language. JavaScript was specified for the client for this project and so that was used. The other limitation is the requirement to use Python for the server-side, with the intention of providing experience in the language.

2.1.1 Language Choices

Despite being limited by platform and specific project requirements, it's useful to discuss the various options that could be selected instead, or on top of, the specified languages.

2.1.1.1 Client-Side

When discussing client-side technologies in the web world, the most obvious choice is JavaScript. It is the only language natively supported by all browsers, with a lot of development time having gone into something of a speed race between them all to have the fastest JavaScript platform.

However, while JavaScript is essentially the only reasonable choice for standard web-apps, the somewhat more old-school ActionScript run in the Adobe Flash plug-in remains an option for games. Indeed, looking into how to create games for the web, Flash remains the most common result. For the most part this is simply the legacy of Flash being the only option in the past, with HTML5 and JavaScript catching up, but there are still some advantages to the Flash platform.

The primary advantage of Flash is the programming language used for it: ActionScript. ActionScript is like JavaScript in some ways, but it adds static types and more standard object-orientation. JavaScript is capable of simulating the more standard object-oriented style (as will be seen later in this section), but cannot simulate static types by itself.

The advantage the classical object-oriented style gives is one of easy application of common game design patterns. Game design has been object-oriented almost since object-orientation was invented and all the patterns that have emerged from decades of game design experience utilise it heavily. By deciding to go against object-orientation you are forcing yourself to reinvent a lot of what is already done, for no particular reason.

As mentioned previously, JavaScript can simulate this style, but with some limitations, primarily that it has no concept of public, private, protected or any of the other useful method and property types.

The lack of type information in JavaScript also causes issues during development. Many problems arise from incorrect object types being passed into methods because the design has evolved and changed. It's an issue that the ability to add typing information to things would solve but that is simply impossible to do in JavaScript.

So, considering the advantages that ActionScript offers over JavaScript in games programming, why not go for that instead?

The primary reason in this project is that the goal was to create a HTML5 game, using canvas, which ActionScript cannot interact with. The second reason is that to use ActionScript you require that the Flash plugin be installed. For most users this isn't an issue—Flash remains a big part of the web thanks to YouTube and other similar sites—but mobiles users in particular would suffer. Although mobile platforms were not targeted for this project, using HTML5 and JavaScript does make it a lot easier to adjust to those devices in the future. There is also the case of being completely unfamiliar with ActionScript and the Flash development environment, particularly in regards to how to interact with a server.

Although JavaScript was selected for the project, it was not required that all its disadvantages just be accepted and dealt with. There are ways to create some of the desired functionality in JavaScript, and further there are languages created which compile down to JavaScript.

JavaScript was a language designed in a week to implement simple functionality on an otherwise static web page. Because of this it lacks a lot of features which make creating larger applications a lot easier, like the aforementioned static typing and classical object-orientation, which software engineers are already familiar with. Many attempts have been made to deal with these problems but there is only so much that can be done in the context of the language itself, without changing it fundamentally.

Because one cannot change JavaScript itself, there have instead been attempts to create languages that compile down into JavaScript instead. The most popular of these are CoffeeScript, TypeScript and Dart.

CoffeeScript's purpose is to act in the same way as standard JavaScript but with different syntax, making it more similar to Python by removing things like braces. Its other purpose is to add useful

abstractions and functions to fill out JavaScript's feature set, and this includes a way to specific more traditional object-oriented classes. However, it does not offer static typing.

TypeScript is another language that attempts to add features on top of JavaScript, created by Microsoft. However, it doesn't change the syntax and standard JavaScript can be used in TypeScript just fine. It adds traditional classes as well as the option for specifying type information, though it is optional to do so. The ability to use standard JavaScript code when writing TypeScript gives it the advantage of letting it use native JavaScript libraries without issue.

Finally, Dart is a Google project that attempts to create an entirely new language with a native browser-parser (available as a plugin), or an option to compile down to JavaScript. It too offers classes and optional typing in the same way as TypeScript, though normal JavaScript is not valid Dart code so you cannot as easily use common JavaScript libraries.

TypeScript was heavily considered for this project. It had the advantage of types and a more familiar syntax over CoffeeScript, and the advantage of being able to use JavaScript libraries over Dart. However, in the end it was decided against using it because it would require time to learn about the TypeScript development environment, compiling it and how to serve the compiled files from the server, which was already using an unfamiliar language.

This left normal JavaScript, which still has all the previously mentioned disadvantages. The issue with typing could not be solved as JavaScript simply has no way to support it. The only way around this is to have a large amount of boilerplate code that manually checks the types of things before anything is done to them, which creates a huge amount of extra work. However, the more classical style object-orientation is at least partially possible with JavaScript. To get this functionality, a library by was used (INSERT CITATION), which simulates the standard class structure found in other languages.

2.1.1.2 Server-Side

The server-side language chosen was Python, for the simple reason for wanting to gain experience in it. However, it is worth exploring what advantages Python has and what other choices were considered.

Python is a very popular language. This popularity means that it is never unreasonable to expect there to be a library or framework available that implements functionality you want to use. Its popularity is often attributed to the language itself being very good, with a clear syntax and well designed language features.

Python is also popular as a platform for creating simple games or teaching games programming and has several libraries made for it to make these tasks easier. It is also able to be object-oriented in a traditional way, which, as discussed in the previous section, is useful when creating games.

However, despite these things the web programming world mostly seems to have left it alone and taken up other languages instead. Many modern web development ideas and libraries are lacking in the Python eco-system. For example, the SocketIO framework that implements web-sockets is well supported in node.js but was surprisingly only recently well supported in an easy to use way

in Python. Because this project requires an active server to support the multiplayer aspect, these problems caused some issues, though they were ultimately resolved.

The other languages considered for this project were PHP and node.js. PHP was considered due to familiarity but offered no real advantages over Python anyway. node.js was a much more interesting option because it is JavaScript, which gives it the advantage of being able to share code with the client-side. On the other hand, it shares all the disadvantages of JavaScript on the client-side as well.

2.1.2 Frameworks and Libraries

There are a lot of frameworks and libraries available for creating games—both in JavaScript and Python—which offer everything that was created in this project and simply require you to implement your own game logic and art assets. However, the goal of this project was not to create a well polished game, but rather to implement an engine to run the game from and gain an understanding of game programming. Because of this it was decided against using any of the available frameworks.

That said, there were frameworks and libraries used in this project. On the client-side the class.js library has already been mentioned. This allowed for a more traditional object-oriented class style when writing JavaScript, instead of JavaScript's own prototypal style.

The other library used on the client-side was one that implemented the A* pathfinding algorithm (ADD REFERENCE NERD). The decision to use this library over implementing it directly was because it was already well made and included some useful optimisations.

Across both server and client the library SocketIO was used, which is a more powerful implementation of the websocket technology that came about in (html5). On the client this was used directly, as by default it is a JavaScript library. On the server-side a plugin for the Python framework chosen was used instead.

The framework mentioned above is Flask. Flask is a minimal framework for Python that provides basic features for a web application server, such as routing and rendering of templates to send to the client. Additional functionality is added in through plugins, such as the aforementioned SocketIO plugin to provide websockets.

Beyond these few things, all code was custom written.

2.1.3 Database

Most modern game designs offer the saving of state to the user. This can be achieved in a lot of ways, depending on the needs of the game. For a single-player game, it may be reasonable to save the game's state in the user's client-side storage offered by their browser, for example.

However, in a multiplayer game the server needs to keep track of everything, and there is also potentially more state to keep track of. Because of this a database is required.

The database implementation used in this project is SQLite, which stores the database in a single file and requires little set up to get running. Although it lacks many advanced features of some of

the other databases available it offered enough for this project to run well. Its only disadvantage is that it is not as fast as something like MySQL. However, because of the use of a Python framework to manage the database, swapping out which one is used is a case of changing only a few lines of code. This means that, should SQLite ever become a bottleneck for the game—which is unlikely given the design used—upgrades are easily implementable.

2.2 Overall Architecture

The overall architecture of a game is very important to making sure that features can be added and changed easily. A common example in the game world is that of heavy use of traditional object-oriented inheritance, where there is a very deep hierarchy. Having a few simple types of object in the world makes this manageable, but the moment you start wanting to add a lot of other different types, or special versions of something, you end up with a mess that's very hard to get out of.

The solution to this is what is known as a Component-Entity System. In this model you have generic entities that are made up of lots of individual components that each have separate functionality. In this way you can easily create a lot of different types of objects without having the mess of inheritance that the more traditional model creates. However, while the CES model is most certainly better, it is a lot harder to implement and creates overhead for development time. In this project the decision was made to avoid going for a full CES model, as there are only two types of entity that exist: a basic entity and a character.

The client and the server share very similar architectures in concept. The server is intended to act in the same way as a client but have its state accessible by the clients when they connect or reload, and to save the state of the game in the database when the game is ended.

2.2.1 Game Loop

The central part of any game is what is known as the game loop. This is responsible for managing the entirety of the game, making sure every other part of it runs when it is supposed to.

An important concept to understanding the purpose of the game loop is the **frame.** This is a slice of time in the game world, with the amount of time each frame represents being decided by how many frames you wish to have per second. Common numbers chosen here are 30 and 60, meaning a frame is 33.3 milliseconds and 16.7 milliseconds respectively. While it is usually trivial to have a simple game running at 60 frames per second, it isn't always the right choice. More frames per second means more processor time used, which drains more power or could cause issues on weak devices. For this project it is also unnecessary, as the game world does not require a simulation at that level of fidelity. 30 frames per second was chosen as a good balance between smoothness of gameplay (such as moving around) and demands on power.

The purpose of the game loop is to update the simulation every frame while keeping track of whether the simulation is running at the correct speed. Every time it runs it first needs to check how much time has passed since the last frame was processed. If a frame took too long to process the last time, for some reason, then the next update will run as many times as necessary to catch

back up to where it should be, regardless of whether that means things are moving too quickly or not.

The game loop itself is not directly responsible for updating the simulation, however. Rather, it calls every entity that is currently active and asks it to update itself. Once all entities have updated themselves the frame is complete.

2.2.2 Entities

Entities are the next most important part of the game architecture, alongside the event manager, which will be discussed next. Each entity is an object in the game world. In the Component-Entity System mentioned previously, it would be made up of many individual components, linked to a component manager so that they could talk to each other and share necessary state. The entity itself would have no real idea about what it is or what it does, it simply gets each component to do something in turn, in a way similar to the game loop asking the entity to update in the first place.

However, because implementing this is very complicated, entities in the project game don't work this way. Instead, they are more traditional objects, with methods and properties. This works fine because there are only two types of entity in the game: a basic entity and a character.

When an entity is asked to update itself, it checks the list of events that it has received from the event manager since the last update and runs through them, applying a series of functions in order. Once it is done the game loop asks the next entity along to do the same thing. Basic entities don't do anything on an update, whereas characters will check for input from a player or the server and update their positions or perform an action.

2.2.3 Event Manager

The event manager is an important piece of the game engine, and acts as the communication method for all the other parts.

Unlike almost all the other parts of the game, the event manager is not run per frame or controlled by the game loop. Rather, it receives and propagates events as it gets them. This means that events that are created in the middle of a frame's update are still sent to the things that need them.

This has some advantages and some disadvantages. The main advantage is network latency. Because the network latency is several frames long, the ability to have the network send events back in the same frame that they happen helps to reduce any time wasted.

The main disadvantage is in the case of entities affecting each other. In the case where an entity performing an action on another comes first in the update cycle, then the affected entity gets a chance to update itself based on that information in the same frame. In the case of an affected entity being updated first, however, it will only take into account the new information in the next frame. This isn't a huge issue, and a user should never really notice that something has happened one frame after, but the inconsistency is there and could potentially cause problems in situations where frames are taking a long time to process, or in very fast paced games.

2.2.4 Renderer

The renderer is responsible for displaying frames to the player, acting as the view into the simulation. It is also another part of the game engine that is separated from the game loop. It isn't required for this to be the case in order for the game to function; indeed, with the simplicity of the game in this project the renderer could be tied to the game loop's frames pretty easily and cause no problems.

However, it is considered to be good practice to separate them, and for good reason. The renderer is almost always the most taxing part of the whole game. By adding the renderer to the central game loop's update process you cause each frame in the simulation to take a lot longer to complete. If this addition doesn't cause the frame to run over time then no problems will be noticed. However, it is very easy for the renderer to cause frames to run over time and, if it happens consistently, the game loop could get stuck in an infinite loop attempting to catch up.

By separating the renderer out, you can make sure that the central game loop always updates correctly and keep the simulation running, regardless of how many frames the renderer is managing to actually display to the user. It is, of course, desirable to make sure that the renderer is managing to display all the frames, but it's less of a problem if the renderer can only display 20 of them than it is to have the central game loop stuck trying to catch up.

2.2.5 Input Manager

A game isn't much of a game if the user cannot create any input. To handle the user's input there is an input manager, which is responsible for capturing all of a player's input, working out what purpose that input has and then creating an event for it in the event manager, which propagates it to all interested parties.

2.2.6 Network

The networking class is responsible for managing the connection the game has to the server. At the end of every frame it will inform the server of anything the client's player has done, such as moved his character to a new location.

If the server sends a message, however, the network manager will immediately create an event for it. Although this is mostly due to technical implementation reasons, there are advantages in this. Primarily, it means that entities being affected by network events can be updated sooner as they don't necessarily have to wait till the next frame. This is important for the same reason as the event manager, by reducing latency from a very high latency aspect of the game.

2.2.7 Server Differences

The game simulation running on the server operates with the same fundamental architecture, with the exception of having no input manager as there can never be any direct input to the server. However, because the server has to operate many games at once, there is another level higher than

the client-side has.

The server has a traditional web application router that receives and routes events from the client. These events are things like joining or leaving a game as well as the things a client cares about like movement. Each individual game is run in a separate thread due to the blocking nature of the game loop. The server keeps track of who is connected to what game and sends events that are relevant to a game to that game alone, which means that there is no pointless spamming of irrelevant events to games that don't care about it.

This higher level part of the server is also responsible for creating game threads when a user first connects to a game. However, the game loop itself is responsible for shutting itself down, though the router can ask the game to do so as well.

Chapter 3

Implementation

The implementation should look at any issues you encountered as you tried to implement your design. During the work, you might have found that elements of your design were unnecessary or overly complex; perhaps third party libraries were available that simplified some of the functions that you intended to implement. If things were easier in some areas, then how did you adapt your project to take account of your findings?

It is more likely that things were more complex than you first thought. In particular, were there any problems or difficulties that you found during implementation that you had to address? Did such problems simply delay you or were they more significant?

You can conclude this section by reviewing the end of the implementation stage against the planned requirements.

Chapter 4

Testing

Detailed descriptions of every test case are definitely not what is required here. What is important is to show that you adopted a sensible strategy that was, in principle, capable of testing the system adequately even if you did not have the time to test the system fully.

Have you tested your system on ☒real users☒? For example, if your system is supposed to solve a problem for a business, then it would be appropriate to present your approach to involve the users in the testing process and to record the results that you obtained. Depending on the level of detail, it is likely that you would put any detailed results in an appendix.

The following sections indicate some areas you might include. Other sections may be more appropriate to your project.

4.1 Overall Approach to Testing

4.2 Automated Testing

4.2.1 Unit Tests

4.2.2 User Interface Testing

4.2.3 Stress Testing

4.2.4 Other types of testing

4.3 Integration Testing

4.4 User Testing

Chapter 5

Evaluation

Examiners expect to find in your dissertation a section addressing such questions as:

- Were the requirements correctly identified?
- Were the design decisions correct?
- Could a more suitable set of tools have been chosen?
- How well did the software meet the needs of those who were expecting to use it?
- How well were any other project aims achieved?
- If you were starting again, what would you do differently?

Such material is regarded as an important part of the dissertation; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree.

There will be good things and room for improvement with any project. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved.

Review the discussion on the Evaluation section from the lectures. A recording is available on Blackboard.

Appendices

Appendix A

Third-Party Code and Libraries

If you have made use of any third party code or software libraries, i.e. any code that you have not designed and written yourself, then you must include this appendix.

As has been said in lectures, it is acceptable and likely that you will make use of third-party code and software libraries. The key requirement is that we understand what is your original work and what work is based on that of other people.

Therefore, you need to clearly state what you have used and where the original material can be found. Also, if you have made any changes to the original versions, you must explain what you have changed.

As an example, you might include a definition such as:

Apache POI library The project has been used to read and write Microsoft Excel files (XLS) as part of the interaction with the client s existing system for processing data. Version 3.10-FINAL was used. The library is open source and it is available from the Apache Software Foundation [3]. The library is released using the Apache License [2]. This library was used without modification.

Appendix B

Code samples

2.1 Random Number Generator

The Bayes Durham Shuffle ensures that the psuedo random numbers used in the simulation are further shuffled, ensuring minimal correlation between subsequent random outputs [15].

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0 - EPS)

double ran2(long *idum)
{
  /*-------------------------------------------------*/
  /* Minimum Standard Random Number Generator        */
  /* Taken from Numerical recipies in C              */
  /* Based on Park and Miller with Bays Durham Shuffle */
  /* Coupled Schrage methods for extra periodicity   */
  /* Always call with negative number to initialise  */
  /*-------------------------------------------------*/
```

```
int j;
long k;
static long idum2=123456789;
static long iy=0;
static long iv[NTAB];
double temp;

if (*idum <=0)
{
  if (-(*idum) < 1)
  {
    *idum = 1;
  }else
  {
    *idum = -(*idum);
  }
  idum2=(*idum);
  for (j=NTAB+7;j>=0;j--)
  {
    k = (*idum)/IQ1;
    *idum = IA1 *(*idum-k*IQ1) - IR1*k;
    if (*idum < 0)
    {
      *idum += IM1;
    }
    if (j < NTAB)
    {
      iv[j] = *idum;
    }
  }
  iy = iv[0];
}
k = (*idum)/IQ1;
*idum = IA1*(*idum-k*IQ1) - IR1*k;
if (*idum < 0)
{
  *idum += IM1;
}
k = (idum2)/IQ2;
idum2 = IA2*(idum2-k*IQ2) - IR2*k;
if (idum2 < 0)
{
```

```
    idum2 += IM2;
  }
  j = iy/NDIV;
  iy=iv[j] - idum2;
  iv[j] = *idum;
  if (iy < 1)
  {
    iy += IMM1;
  }
  if ((temp=AM*iy) > RNMX)
  {
    return RNMX;
  }else
  {
    return temp;
  }
}
```

Appendix C

Game Design Specification

An online, multiplayer roleplaying game similar to *Dungeons & Dragons* hosted in the browser. The game will have a Game Master and several players in a group. The players move around the world and interact with it and the people/creatures in it and the Game Master guides the players, sets rules and manages the experience. They will be able to manipulate the game world and do things players can't do. For example, they could teleport a player to another location, spawn new monsters or change the weather.

It will include isometric graphics and aim to be as permissive as possible in what players are allowed to do (for example, interact with the environment using items they have, such as burning down a house with a torch). This means that there will be several systems, including combat, movement, character/creature creation, interacting with other characters (players and non-players, with chat as well as game system rules) and the environment (things like objects having heat and a burning point, in the fire example).

The nature of the design means that the project is open-ended, allowing for a basic game with a few simple systems and a single dungeon and character type to a game with several systems, full environmental interaction, a map and character editor and the ability to upload your own artwork.

It will require a server to manage the clients and let them interact, relay chat and act as an authority in the game world to keep clients in sync. It will also have to keep track of where players are in the world and whether they can see and interact with each other. There will be a graphical front end client that the players use and, in theory, multiple different clients could be made, such as a mobile app.

Appendix D

Minimal System

BUTTS BUTTS BUTTS BUTTS BUTTS BUTTS BUTTS

Annotated Bibliography

[1] "Roll20, online tabletop game engine." [Online]. Available: http://roll20.net/

Roll20, an online game engine for playing tabletop games on your computer.

[2] Apache Software Foundation, "Apache License, Version 2.0," http://www.apache.org/licenses/LICENSE-2.0, 2004.

This is my annotation. I should add in a description here.

[3] ——, "Apache POI - the Java API for Microsoft Documents," http://poi.apache.org, 2014.

This is my annotation. I should add in a description here.

[4] J. Bose, *Creating Isometric Worlds: A Primer for Game Developers*, May 2013. [Online]. Available: http://gamedevelopment.tutsplus.com/tutorials/creating-isometric-worlds-a-primer-for-game-developers--gamedev-6511

Tutorial series used to understand the basics of rendering isometric graphics.

[5] H. M. Dee and D. C. Hogg, "Navigational strategies in behaviour modelling," *Artificial Intelligence*, vol. 173(2), pp. 329–342, 2009.

This is my annotation. I should add in a description here.

[6] S. Duckworth, "A picture of a kitten at Hellifield Peel," http://www.geograph.org.uk/photo/640959, 2007, copyright Sylvia Duckworth and licensed for reuse under a Creative Commons Attribution-Share Alike 2.0 Generic Licence. Accessed August 2011.

This is my annotation. I should add in a description here.

[7] M. Foundation, *Mozilla Developer Network*. [Online]. Available: http://developer.mozilla.org

Online APIs for JavaScript and HTML5.

[8] M. McShaffry and D. Graham, *Game Coding Complete, Fourth Edition*, 4th ed. Cengage Learning PTR, Mar. 2012. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/1133776574 1133776574.

Popular and well respected book on the fundamentals of creating a modern game engine.

[9] Mozilla, L. Workshop, F. Lecollinet, and G. Lecollinet, "BrowserQuest Source Code," GitHub, 2012. [Online]. Available: http://github.com/mozilla/BrowserQuest

BrowserQuest source code.

[10] ——, "Mozilla's BrowserQuest Game," 2012. [Online]. Available: http://browserquest.mozilla.org/

The BrowserQuest game itself.

[11] M. Neal, J. Feyereisl, R. Rascunà, and X. Wang, "Don't touch me, I'm fine: Robot autonomy using an artificial innate immune system," in *Proceedings of the 5th International Conference on Artificial Immune Systems*. Springer, 2006, pp. 349–361.

This paper...

[12] B. Nystrom. Game Programming Patterns. Online. [Online]. Available: http://gameprogrammingpatterns.com/index.html

Online book detailing common game design patterns, as well as examples on how to use more general design patterns in the context of games.

[13] N. G. Obbink, *Javascript tile engine tutorials*, 2012. [Online]. Available: http://nielsgrootobbink.com/wokflok/jte/

Series of tutorials going through the creation of a simple RPG in JavaScript.

[14] M. A. Pagella, *Making Isometric Social Real-Time Games with HTML5, textsccss3, and Javascript*. O'Reilly Media, Sept. 2011. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/1449304753 1449304753.

A book about creating isometric games in the browser using textschtml5 and JavaScript

[15] W. Press *et al.*, *Numerical recipes in C*. Cambridge University Press Cambridge, 1992, pp. 349–361.

This is my annotation. I can add in comments that are in **bold** and *italics and then other content.*

[16] P. Rettig, *Professional HTML5 Mobile Game Development*, 1st ed. Wrox, Aug. 2012. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/B0094P2TU6

Book on creating browser games using HTML5 and JavaScript

[17] Various, "Fail blog," http://www.failblog.org/, Aug. 2011, accessed August 2011.

This is my annotation. I should add in a description here.

[18] L. Workshop. (2012) Little Workshop Information on BrowserQuest. [Online]. Available: http://www.littleworkshop.fr/browserquest.html

Web page by the authors of the BrowserQuest game giving some information on it.