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)

May 6, 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* [4–6], 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* [7]. 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* [9]. 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 [12] and one which did the same thing but with the addition of a focus on isometric graphics [11].

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* [8] 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 [10] 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 [2]. 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

Game design has evolved a lot over the past few decades, taking advantage of more powerful devices and advances in programming generally. Game programmers were quick to take up the object-oriented banner, and were quick to become aware of its issues as well, eventually finding replacement patterns that solved the problems they came across.

There are a lot of frameworks and pre-made game engines around that allow a user to quickly get up and running in making their own games by handling the complicated parts such as rendering and input handling, asking the game designer to simply add some logic and art assets. However, these frameworks can have issues when you attempt to make a game that does not fit their predefined notions, and a lot of time could be spent hacking around their limitations to create an unusual game type.

Although it is extra work, implementing the game engine yourself gives you a lot of freedom in deciding what is necessary and what is not. Unusual aspects of the game design—such as a mix between real-time and turn-based interaction—can be accounted for and designed into the game engine much more easily, without having to first delve into unfamiliar code. This project also had the intention of learning by implementing these more technical aspects of the game engine, rather than to make a polished game on top of something that already existed.

This chapter will discuss the languages and frameworks that were used to implement the game and its engine, followed by the design of the engine itself, offering a higher level overview of the architecture as well as a more detailed look into the individual components that make up the game engine and how they fit together.

2.1 Languages and Tools

Even ignoring the direct specification of languages and technologies that the project required, the need for it to be run in a browser already limited the choice of languages and tools available to develop the game engine in.

2.1.1 Languages

2.1.1.1 Client-Side

Previously, the most popular and obvious choice for browser-based development of serious applications was ActionScript, the language that runs in Adobe's Flash plugin. This language has a lot of advantages over JavaScript—which is the language specified and the other main option—primarily in its traditional object-oriented style as well as the use of static types. Both of these are useful for creating large applications in general, but the classical object-oriented style is especially useful in game programming, as most standard patterns used in game programming were invented for languages using it; moving away from it begins to create problems of having to reinvent a lot of what has already been figured out.

ActionScript has also remained popular for creating games, despite its disappearance from other types of web application. This means that there are plenty of resources available for it, from books to libraries and frameworks.

However, despite all these advantages, ActionScript and Flash are unsuitable for this project for a few reasons. The first is ActionScript's requirement to be run in a Flash container. The project specified the use of HTML5 canvas, which is a native browser construct that isn't usable by Flash, because Flash is a plugin. Its plugin status is also a potential problem for accessibility. Although it remains the most ubiquitous browser plugin for desktop use—a status most likely still conferred by its continued use for video sites like YouTube—it is an issue for mobile users who cannot install it at all, and there is plenty of evidence that it is a dying platform on the web more generally [REFERENCE], even if games are slower to catch up than other forms of web application. There is also the case of it being an unfamiliar environment, particularly in regards to how to interact with the network, a problem exacerbated by the server being written in the equally unfamiliar Python.

JavaScript has the advantage of being both familiar and natively supported by browsers, letting it interact with the HTML5 canvas element. Indeed, JavaScript is the only language natively supported by browsers. However, as previously mentioned, JavaScript has some disadvantages for creating games and other larger applications. While neither the lack of a more traditional object-oriented structure nor the lack of static typing make it unworkable for creating games, it does cause unnecessary issues. The lack of typing is of course the traditional (and commonly argued about) trade-off between easy prototyping and easier refactoring and debugging. The prototypal inheritance style verses the classical class structure is more of an issue, however, as all game design patterns are designed to be implemented in the latter.

The most extreme way to solve these issues is to look into languages that compile down into JavaScript while providing the features that make them more attractive to program in. Three major options exist in this area: CoffeeScript, Microsoft's TypeScript and Google's Dart.

CoffeeScript is an attempt to redo JavaScript's syntax to be more succinct, while providing some quality of life enhancements in the process. Although it does offer a way of specifying classes in the classical way, it doesn't offer typing, optional or otherwise. It is probably the most popular of the options, having been available for the longest, but has the disadvantage of needing to learn its syntax, whether that syntax is better than JavaScript's or not.

TypeScript is a superset of JavaScript, and regular JavaScript code is perfectly valid TypeScript code. This immediately gives TypeScript an advantage because it is already familiar. On top of JavaScript it adds classes and optional types—both the major features missing that are useful for game development.

Finally, Dart is a language that is intended as a complete replacement for JavaScript, even offering a native VM to run it within some experimental versions of the Chromium browser, though it can compile down to JavaScript as well. It offers many of the advantages of TypeScript, but with CoffeeScript's disadvantage of unfamiliarity.

All three languages also have the disadvantage of adding an extra compilation step, and potentially problems in making sure that the compiled code is actually sent to the browser, especially used inside the unfamiliar Python language and framework used on the server-side.

TypeScript was the most tempting option but was ultimately rejected because, although it is familiar in many ways, the extra steps and tooling surrounding it were considered potential issues and there was a lack of desire to add too many unfamiliar parts to the development process when the server-side would be entirely alien.

Although these options were rejected, at least one of JavaScript's problems was able to be worked around. JavaScript's prototypal inheritance style can be coerced into looking like a more classical class structure, and to that end a library known as class.js [REFERENCE] was selected that implemented this. Although the library cannot deal with JavaScript's other issues, such as a lack of easy scoping inside a class or the lack of types, it created an environment that was much easier to work with when implementing the common game design patterns.

2.1.1.2 Server-Side

The project specification demanded that Python be used as the server-side language, due to a desire to to gain experience in it. The main reason for this is that it is a very popular language and is used in a lot of areas, from major science to simple scripting. It is also popular in games, acting either as a scripting language embedded into C++ code, or as the entire game engine itself. Indeed, several frameworks and game engines exist in the Python world that are designed to make games programming easier, offering support for interacting with common graphics libraries like OpenGL, and UI libraries like Qt.

Python is also somewhat popular in the world of web applications, offering frameworks and libraries to handle many types of networking situations, including HTTP and websockets as used in this project. However, it is certainly not the most popular language, and in some regards lags behind others in this area, particular as web development is a very fast moving world.

If Python had not been required, Node.js would likely be the choice of language for the server-side in this project. Although it is not as popular as Python in terms of the number of sites actually using it [REFERENCE], it has been steadily gaining traction in the web development world and a lot of modern tools and frameworks in this area are created for it and other trending languages over Python. The SocketIO library used in this project to implement websockets was made originally for Node.js, for instance.

Node.js also has the major advantage of being JavaScript, which means that it is not only familiar, but almost entirely the same as the client-side. Not having to switch languages when developing across the client and server is a definite advantage and, even though it also has JavaScript's weaknesses, both CoffeeScript and TypeScript can be used instead [REFERENCE] if the language replacement route is being taken.

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 some 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 [REFERENCE]. The decision to use this library over implementing it directly was because it was already well made and included some useful optimisations. This saves a lot of time compared to custom implementing it just for the project.

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 Python 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 and a plugin to provide easy database access.

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 relevant

to this project is that it is not as fast as something like MySQL [REFERENCE]. 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—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. The advantage of all this abstraction is counterbalanced by vastly increased complexity in implementation. Because every part of the functionality of an entity is compartmentalised, you end up needing extra managers to handle that.

Because the game in this project was going to be very simple, with the minimal system having only two types of entity, it was decided that the Component-Entity style was not worth the extra development effort; there is only a base entity and a character type entity which inherits from it, though there is a small concession to the idea of Component-Entity on the client-side, which will be discussed in the next section.

Before getting to the details of each part, an overview is necessary to give context to their purposes. There are only a few individual components to the whole system, though they all have need to interact.

The most important component in the engine is the game loop. This is responsible for the updating of the game simulation every frame by asking each entity it knows about to update itself in turn. When asked to update themselves, the entities check their event queues for any new events and process them as required, then proceed to perform any necessary updates to their state, such as movement, as well as creating their own event to inform the server if relevant. These events are passed around the system by an event manager, which sits outside of the game loop. Any part of the system that wants to know about something registers itself to receive event updates related to that thing and is then given them the moment the event comes in. Finally, there is an input manager and a network manager, which handle input (and output in the case of the network manager) from players and the server.

This architecture applies fully to the client but the server makes some changes. These changes come about primarily due to implementation details and the differences between JavaScript and Python. However, the biggest difference by purposeful design is that the server needs to handle multiple games running simultaneously but apart, rather than having to deal with only one. This

means that the server has a higher level manager responsible for creating games and routing messages to and from the server-side games and the clients.

2.3 Detailed Design

2.3.1 `Engine` and `Scene`

The `Engine` class is the central class in the game. It manages the creation of all other system classes, including the event manager, the network and input managers and the renderer, and contains the central game loop. Once the game is loaded it tells the server that the client is ready so that it can be registered to receive network events. It is also responsible for creating and giving access to the HTML5 canvas used to render the game.

The `Scene` class is what describes the actual content of the game simulation at that moment. Each scene represents the map the players are currently on, and contains the terrain as well as all the entities. When the game simulation is updated, it is the entities in the `Scene` class that are affected.

To load the scene from the server, the `Scene` class contains several methods that first communicate with the server and then handle the result. The server keeps track of which scene is the currently active one in a game, so `Scene` gives the server its game id and is given back the scene data in response. This data is processed and turned into actual terrain and entities. No other part of the game is able to create new entities.

The `Scene` class also remains the only place where the new event manager was not implemented. Due to the issues with the previous event management system, discussed in section 2.3.4, network connectivity had to be implemented directly into `Scene`, rather than abstracted to the correct `Network` class. With the event manager, this shouldn't be necessary, but the retrofitting work was not completed in this case.

2.3.2 Game Loop

The central part of any game is what is known as the game loop. It is run from the main `Engine` class and 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, for some reason, a frame took too long to process the last time 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 in real-time 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 network manager is asked to inform the server of everything that happened it might need to know about and the frame is complete.

2.3.3 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`.

The basic `Entity` class is a simple object in the game world like a wall. It has no properties beyond it's size, position and the sprite that represents it if one exists. A `Character` is a lot more complicated, and has methods for handling user input and updating it's position, as well as dealing with any other interactions that might occur, such as taking damage from another entity.

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 `Character`s will check for input from a player or the server and update their positions or perform an action.

The existence of two types of player with different capabilities created some issues. Initially there was actually a third entity type that inherited from `Character`, called `Player`. The `Player` class had extra functionality to find paths for itself, and interpreted player input but not server input. However, because a Game Master needs to be able to change which entity they are controlling, the `Player` class was merged with `Character` and a method for swapping between them was created.

As there were two ways to interpret any input events, and they needed to be switched out, two functions were made into components: input handling and path finding. The input handler for an active player listens for events from the input manager, whereas the input handler for a remote player listens for network events. The pathfinding component on an active player actually creates paths, whereas the remote entities simply get given them from the network.

2.3.4 Event Manager

The event manager is an important piece of the game engine, and is the communication method for all the other system components, acting as a sort of glue between them.

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. 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. Another small time-saver in this regard is that an entity affecting another entity may be able to have that effect be known about in the same frame, as long as the acting entity is processed before the entity being acted upon. 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.

Despite the seeming obviousness of its use, the event manager originally didn't exist, and input was handled by having a single event queue stored in the `Engine` class. This event queue was given to entities from the update call in the game loop and they ran over each event looking for something relevant to them. It was also very difficult for entities to put anything into the event queue themselves, such as network input. Because of this, network events were created directly. Worst of all, because there was no way to know whether events were handled, the entire event queue was cleared on each frame, which meant that no events could be carried over.

Obviously this approach had several disadvantages. Firstly it was inefficient. Although generally there was only going to be one or two events in any given frame, there are potentially a lot of entities that have no interest in events that are stored in the event queue, so having them loop over those events is useless. Secondly, in order to update the network or add anything back into the event queue, entities had to be given the network and engine objects and pass them around. The emptying of the entire queue also meant that in situations where one entity wanted to affect another but was updated after the entity it was affecting, the affected entity would never know about it.

Replacing this system with a proper global event manager allowed for entities to no longer care about events that don't matter to them, as well as much more easily carry over events from frame to frame, even leaving events unhandled for a long time if necessary. Unfortunately the refactoring to use the event manager wasn't entirely completed, as seen with the previously mentioned `Scene` class. However, the vast majority of the game uses it.

2.3.5 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.

It is, however, 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. Unfortunately, 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.

The renderer does not render the world in one go. Rather, it renders the terrain first, and then renders entities on top of it. The reason for this is one of efficiency. The terrain never moves and never crosses tile boundaries. It also comes pre-ordered in a grid. These attributes make it easy to render by simply drawing each tile to the screen one by one in order. Entities, however, can have different heights and can move around and cross tile boundaries, which requires that they get sorted properly, creating rendering overhead. By rendering the terrain first, there is less that needs sorting afterwards, thus speeding up the renderer.

2.3.6 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.

The input manager is only able to work out what user input means, not what that input applies to. In other words, it knows that a user clicking means the user wants to move somewhere, checks that it is a valid input and then creates an event to say that there has been a movement input and the tile to move to. This event is passed to all entities that subscribe to movement events and it is up to them to work out whether it is relevant or not.

2.3.7 Network

The `Network` 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.

2.3.8 Server Differences

The server's purpose is to act as a central version of the game that links all the clients together, and is responsible for being something of an authority in the game simulation to make sure that clients see the correct things. Because of this it makes sense that the server and client have the same structure.

However, there are, in fact, differences between them. The reason for this is entirely due to implementation—which will be discussed in more detail in the next chapter—but regardless of the reasons it's important to understand these differences.

The most important and major difference is that the server runs many games simultaneously, rather than just a single game at a time. To handle this, a higher level is needed to manage the games and route messages to and from players.

The router that handles this is a traditional web application router, made using the Flask framework. It takes incoming HTTP connections and works out what they are supposed to do based on the URL. Once a game is up and running and users are connected, they communicate with it through the SocketIO library providing websockets, which the router also manages.

Each game itself is run in a separate thread, providing an architecture as similar to the one in the client as possible. Communication between the router and any individual game is handled through thread-safe queues, as it is impossible to call a method in the thread directly due to the game loop blocking and taking control.

The game loop itself is probably the most major difference in the internal game architecture, entirely by way of implementation details. The client's JavaScript version of the game loop is essentially non-blocking as it has to always yield control back to the browser. However, Python has no native way of achieving this style—known as an event loop—and the libraries available to implement it are complicated. Because of this, the Python implementation of the game loop is a more traditional `while` loop, running until a flag indicating the game is to be shut down is set to false.

An implementation of the JavaScript style was tried without using complicated libraries and in some ways it worked. However, when making the game simulation run at the full 30 frames per second, it ended up blocking the thread anyway, making it no better than the more traditional loop but with a lot of extra logic.

Although this does not affect the fundamental way of updating entities, which is the bulk of the work the game simulation performs on the server, other aspects of the architecture are changed because of it.

There is no proper event manager in the style that was added to the client, as there is no easy way for it to be run outside of the game loop without creating another separate thread for it—an undesirable option for a server that will already be having to manage many of them. Because of this, the server event management remains in the original style that the client had before the event manager was implemented: the game loop contains a data structure that stores all the events and asks the entities to take a look at them in turn and see whether they care.

In a concession to sanity, however, the server uses a queue to store input events (accessible by the router so that they can be passed in) and doesn't delete unhandled events at the end of every frame. Output is placed into another queue that the router can check in order to inform clients of anything interesting.

Other parts of the architecture are simply removed: there is no renderer because the server's view is entirely remote—no one is ever going to be trying to play the game on the server directly. The server also doesn't have an input manager for the same reason. The network manager is pushed up into the router, rather than being a special part of the game engine.

This design is, of course, not perfect. The biggest issue is one of scalability. This model works fine for one or two games running at once, but because the router has to manage connections for all active games it can very quickly get overloaded, especially when players decide to spam actions that create network events such as movement commands.

Ideally each game would be responsible for handling its own connections and the router would simply exist to get players connected to the right game. This creates independence of the router and game servers and is a much more scalable solution. The reason it doesn't do this, however, is just because the implementation knowledge necessary for this was not available and time did not allow for further investigation into it. A less than ideal working solution was considered to be the better choice than potentially having no solution at all because the implementation was too hard given skill, knowledge and time available.

Chapter 3

Implementation

Despite what was thought to be a reasonable list of features for a minimal implementation, the actual act of implementing them proved to be a lot tougher than anticipated, with a lot of problems arising that were completely unforeseen. The server in particular proved to be a serious problem, though the client was not without it's share of issues as well.

This chapter will go into detail about some of the features implemented in the game. In particular it talks about the graphics renderer, the input and movement handling systems and the server, including a lot of the problems that were encountered and what they meant for the project.

3.1 Renderer

The graphics renderer was the first piece of the game engine worked on. Having the graphics to provide a view into the game world was important in order to easily see whether things were behaving as expected and proved extremely useful in detecting problems with other parts of the game later on.

This section will go over firstly what isometric projection is and then the ways that graphics are actually drawn to the screen. Finally, the difficulties that the renderer caused during implementation will be discussed.

3.1.1 Isometric Tiles

3.1.1.1 What Is Isometric?

When talking about isometric projection in the videogame world it is important to know that it is not truly isometric. True isometric is an "axonometric projection in which the three coordinate axes appear equally foreshortened and the angles between any two of them are 120 degrees" **TODO: reference**. What the gaming world calls 'isometric' is in fact *dimetric* projection **TODO: reference**—a projection where two axes are the same but the third is not.

The reasons for doing this are essentially one of aesthetics when drawing objects at a very low resolution with visible pixels. At these very coarse resolutions true isometric projection creates an aesthetically displeasing line. However, by changing to a dimetric projection, where the line grows twice as fast horizontally as it does vertically, the line becomes much more consistent and visually appealing. **TODO: figure**

3.1.1.2 Cartesian To Isometric

The term "2D isometric tile" used in this report is a little ambiguous. The game world itself is represented by simple 2D Cartesian coordinates that map to a grid of tiles. Isometric space, however, is 3D. The term used here refers to the type of graphic used to represent the tiles in isometric space, rather than the isometric space itself. These graphics are simple 2D images, drawn in such a way as to make them look 3D. **TODO: figure showing cartesian and isometric**

Of course, this means that a conversion has to be made between the two coordinate systems in order for the renderer to be able to display things in their proper place. As an example, figure **TODO: figure** shows what the game world looks like if it is rendered without this coordinate conversion—clearly very wrong.

In order for the renderer to show things correctly in the isometric space there needs to be a way of converting between the game world's Cartesian coordinates and the renderer's isometric coordinates. Luckily, this is incredibly simple:

```
1  function cartesianToIsometric(cartesian_x, cartesian_y)
2  {
3      var isometric = {};
4
5      isometric.x = cartesian_x - cartesian_y;
6      isometric.y = (cartesian_x + cartesian_y) / 2;
7
8      return isometric;
9  }
```

Listing 3.1: JavaScript implementation of a function to turn Cartesian coordinates into game isometric coordinates. Original algorithm from [3].

Now when the renderer draws the game to the HTML5 canvas, it can first call the `cartesianToIsometric` function, letting it place the tile graphics in the correct place as seen in figure **TODO: figure**.

3.1.2 Rendering

After the renderer has converted between Cartesian and isometric spaces it needs to be able to draw things to the canvas. The actual drawing of a game tile to the canvas is straightforward, simply requiring a call to the canvas' `drawImage()` function and supplying it the image you wish to draw

and the canvas coordinates for where you wish it to be drawn.

The difficult part comes before that. Firstly, you need to be able to work out what the canvas coordinates are. The conversion between isometric coordinates and the canvas coordinates works as follows:

```
1 canvas_x = isoCoords.x * (TILE_WIDTH / 2);
2 canvas_y = isoCoords.y * (TILE_HEIGHT / 2) - (image_height - (TILE_HEIGHT / 2));
```

Listing 3.2: Conversion between isometric coordinates and canvas coordinates.

A few things should be explained about this. Firstly, `TILE_WIDTH` and `TILE_HEIGHT` refer to the set size of a single tile and are used to work out the tile's positioning in the world. The `image_height` value refers to how tall in pixels an image is. Theoretically an image can have any size and isn't limited to how large a tile is. However, in practical terms, an image can only be arbitrarily tall; width needs to be kept within the tile limits or there will be problems with depth sorting, for reasons seen in section 3.1.2.3.

TODO: a figure showing the difference between a tile and the image representing it

Also worth noting is that there are separate rendering functions used for the terrain and the entities. The terrain has no height value, but even if it did it is rendered in such a way as to make it irrelevant, so the drawing of terrain tiles excludes the `image_height` part of the calculation.

Once you have the coordinates you then need to get the tile graphic to use as the image being rendered. Getting the image itself is a very simple task, but getting the order in which to draw them right is more difficult.

The HTML5 canvas is a very simple element and it has no ability to specify the relative depth of things being drawn to it; whatever is drawn first will be occluded by what comes afterwards if they overlap. This overlapping is often not a problem when drawing simple 2D tiles with a top-down or side-on perspective. However, because the isometric perspective is 3D the issue of depth becomes an important one.

TODO: Figure of what it looks like when depth sorting is broken

There are two methods for solving this used in the renderer: a simple painter's algorithm and a depth sorting algorithm. The first is used for the terrain and the second for entities.

3.1.2.1 Terrain Rendering

The terrain is very simple to render as far as isometric projection goes. Terrain images are the same size as the tiles, which means that as long as they are drawn in order the images will never overlap.

The method for achieving this ordering is known as a painter's algorithm[1] and is very simple.

[1]The painter's algorithm is so called because of a common painting technique whereby distant parts are painted before closer parts that might cover them.

Firstly, all the terrain is stored in a pre-sorted, two dimensional array. Columns are labelled `y` and rows are labelled `x`. Images are drawn by looping through each row in each column and drawing them one by one, rendering the images from back to front, row by row.

TODO: Diagram showing how the painter's algorithm works

This is a very efficient and cheap way to render the images to the screen, and works perfectly for the terrain. Even if the terrain were to be made no longer flat, as long as it never broke out of the tile grid there would be no need to change how it is rendered.

3.1.2.2 Entity Rendering

Unfortunately, entities are less easy to deal with. At least some entities can move around—crossing tile boundaries when they do so—which means they cannot be stored in the same grid that makes up the tiles themselves. Because entities can exist outside of the tile boundaries, working out their depth cannot be done just by iterating through them like the terrain. Another way has to be devised in order to render them in the correct order.

To figure out the order that entities need to be rendered it is first necessary to work out what their relative depth in the scene is. To get an entity's depth, the following calculation is used:

```
1 function calculateDepth(entity) {
2     return Math.round((entity.x * TILE_WIDTH / 2) + (entity.y * TILE_HEIGHT / 2) +
          entity.z);
3 }
```

Listing 3.3: Calculation to work out an entity's relative depth in the scene.

In this calculation the most important part is the addition of an entity's position data, `x`, `y` and `z`. The `z` attribute is not currently used in the game, but if it were it would represent relative height from the ground plane and is thus important to the depth calculation. The use of the `TILE_HEIGHT` and `TILE_WIDTH` attributes is due to the small values of the position data. Because tiles can potentially have small values like `(1, 1)`—though they could also in theory be very large—it is often common for depth values to be worked out as being the same. Multiple entities sharing a depth value will often cause them to be rendered in the wrong order. The other issue is that positions can be non-integer, such as `(1.250, 1)`. The sorting method used can't handle this, so it's necessary to round to a whole number in order to make sure it doesn't break.

The sorting method itself is a modified pigeonhole sort, as shown in Listing 3.4.

```
function depthSort(entities) {
    var buckets = [];

    for (entity in entities) {
        var depth = this.calculateDepth(entities[entity]);

        if (!buckets[depth])
            buckets[depth] = [];
        buckets[depth].push(entities[entity]);
    }

    var result = [];

    for (bucket in buckets) {
        for (entity in buckets[bucket]) {
            result.push(buckets[bucket][entity]);
        }
    }

    return result;
}
```

Listing 3.4: Depth sorting entities in a scene using a modified pigeonhole sort.

The pigeonhole sort works well for this task. The first reason for this is that the number of entities and number of keys are almost always the same, and even when not the smaller number of keys does not affect the performance of the sort. The number of entities being placed into the sort is also usually very small, which means that memory is not a concern, though with a very large number of entities the memory footprint could be a problem. The other reason the pigeonhole sort works well is that it keeps intact the link between a depth value and the entity it represents without any extra work. This means that the sorted array containing the entities can be used directly in the renderer, saving development and processor time.

The sorted array that comes out of the pigeonhole sort is flat, so the renderer only needs to loop through it once in order to render all the entities to the canvas at the correct depth.

3.1.2.3 Depth Sorting Limitations

The depth sorting used in the project works well and performs as expected. However, it has a major limitation in that it will not work for entities that can exist across multiple tiles. Currently, all entities are the size of a single tile—though they may have arbitrary height within it. This makes calculating the depth a simple case of looking at position. If an entity were to be larger than a single tile, however, the position data alone would not be sufficient to determine depth.

There are two ways to have entities that can exist across multiple tiles. The first is to have the entities be actually made up of multiple tiles. There are two frustrations with this. The first is that artwork needs to be split up into multiple tiles, which means that content in the game is harder

to create. The second is that entities need to keep track of what tiles they are taking up, creating further work for the sorting algorithm to do in order to figure out what is where and gaining little over simply having lots of separate entities pushed together to appear whole.

The better option here is to keep a model of an entity's position in 3D space. Although the entities themselves are rendered with 2D images, because of the isometric projection they still exist in 3D space. By keeping data that models this three dimensionality it is possible to work out the depth of an entity without associating it with more than one tile. Instead, it is given a central tile and the data about its size fills out the rest of it.

This approach affects more than just the rendering. By having an actual size in 3D space entities can be interacted with in a more fine-grained way. For example, the current system of collision detection simply checks to see whether a tile position contains an entity or not. This is relatively simple to implement but it is not very versatile—position data is very coarse. By turning the game world into a 3D simulation it is suddenly possible for position data to be more fine-grained. Not only can the renderer work out the depth of something that is not bound by whole tiles, collision detection is no longer bound by whole tiles either.

Of course, doing this is very complicated and would require a complete rewrite of almost all of the game engine's underlying systems. Given the difficulties already encountered when implementing what was there it was decided that the huge effort involved was not worth it.

3.1.3 Renderer Implementation Difficulties

Being the first major feature to be worked on meant that the renderer was subject to a lot of change and iteration as the project developed. Things that appeared to be working fine at first turned out to be inadequate when other features were added. The renderer was probably the most rewritten and refactored part of the entire code base.

The terrain has been the most stable part, with the rendering method being unchanged since it was first implemented. However, the entity rendering was changed a great deal. The initial implementation had the entities rendered in a way similar to the terrain, by placing them into an array at the correct locations and then iterating through it. While this technically worked, it was ugly and was soon found to be completely inadequate once entity movement was introduced to the game.

The method for rendering entities including the depth sorting also did not work perfectly the first time either. At first the pigeonhole sort was not working properly, as entities were overwriting each other when they had the same depth value. This was solved by introducing the idea of buckets that could store multiple entities at once when they had the same depth value. The next step to solving this was to make sure that the depth values weren't so small. Because the depth value needed to be rounded in order to not break the sorting function it was often the case that entities were sharing the same depth value. Although this rarely affected entities that were sitting still in a tile, entities moving around next to others were regularly having their depth be calculated incorrectly. This was solved by adding the tile values to the calculation, which created much more variation in depth and mean the rounding was less significant. It is now far rarer for entities to share a depth value, and when they do they do not overwrite each other.

Beyond these specific problems, the renderer's place as first feature implemented also suffered from the lack of experience, in JavaScript, use of the HTML5 canvas and games programming in general. A lot of time was spent researching and iterating on things that now would be much easier to implement and be done in a lot shorter time.

3.2 Input and Movement

Movement is one of the most important user facing features of the project. It also makes use of several parts of the game engine to work properly, relying on the game loop functioning correctly to advance the entity at the correct pace, the renderer to display this movement across tiles correctly and both the input manager and network manager to receive (and send) commands.

This section will discuss how input is handled and how that relates to movement, as well as how movement works. In both cases, problems that arose during implementation will be covered, as well as how those problems were solved.

3.2.1 Input

The first and most important of these major parts is the input handler. Originally, the input handler was contained entirely in the `Engine` class. However, with the advent of the event manager it was refactored out into its own class.

Input in the game is mouse-based, rather than keyboard-based. Being mouse-based, input is a little more difficult to handle than if it were possible to simply capture a button press. It is necessary to work out what location exactly the player has clicked, then go through a series of functions to turn the click location into game-world coordinates. Once the coordinates are known, they need to be checked for validity and then passed on, along with the event type, to any entities that might care to know about it.

The first step in this is capturing the user's input. Browsers are helpful in that they keep track of where the user has clicked, which means that the browser event can be captured and used for the game's purposes. To listen for user clicks, a click event listener is attached to the game's canvas. Unfortunately, however, despite being attached to the canvas, the click event coordinates are global and in the context of the window, rather than being local to the canvas. What this means is that click coordinate `(0,0)` is not, in fact, the corner of the canvas, but instead the corner of the browser window that contains the canvas.

To solve this, it is necessary to work out the click's position relative to the canvas. This is accomplished using the code in Listing 3.5.

```
canvasPosition.x = ((event.clientX - this.engine.camera.x) - this.engine.canvas.
    offsetLeft) - (TILE_WIDTH / 2);
canvasPosition.y = ((event.clientY - this.engine.camera.y) - this.engine.canvas.
    offsetTop);
```

Listing 3.5: Transforming a user's click to be relative to the canvas, rather than the window.

There are a few things to note in this code. First of all, the `event` used here is the one passed in from the click event listener, and is an object supplied by the browser. The next thing to note is the `camera` object. This is a game object that acts as the user's viewport into the world. Because the world can be bigger than the canvas itself, it is necessary to have a method of moving the view around the world, which is where the camera comes in. The camera object has an offset, which the renderer takes into account when positioning entities onto the canvas. Of course, this offset needs to be taken into account when detecting where, exactly, the user has clicked in the world, and so it is used here to do that. The coordinates are then set to be relative to the canvas instead of the window by using the `offsetLeft` and `offsetTop` values on the canvas itself, which refer to the canvas' position in the browser window.

After this, a set of functions are used to first transform the canvas' isometric coordinates into 2D Cartesian coordinates, and then from there to turn those into actual tiles, shown in Listing 3.6.

```
var cartCoords = (function(x, y){
    var coords = {};
    coords.x = (2 * y + x) / 2;
    coords.y = (2 * y - x) / 2;
    return coords;
})(canvasPosition.x, canvasPosition.y);

var tileCoords = (function(x, y){
    var coords = {};
    coords.x = Math.floor(x / (TILE_WIDTH / 2));
    coords.y = Math.floor(y / (TILE_WIDTH  / 2));
    return coords;
})(cartCoords.x, cartCoords.y);
```

Listing 3.6: Turning the canvas isometric coordinates into 2D Cartesian coordinates and then into tile positions. Original algorithm to transform isometric to Cartesian space from [3].

With the tile coordinates acquired, they are sent into a method to work out what to do with them. It first checks for the validity of the tile coordinate by checking whether it is inside the bounds of the map. Assuming the coordinates are in bounds, a check is then done to see whether the player has clicked on an entity or whether they have clicked on a terrain tile. Clicking on an entity could mean either an invalid click location or a command to change to controlling that entity, depending on whether the user is the Game Master or not. Clicking on an empty tile—that is, one not occupied by an entity—is a movement command. Holding shift while clicking on an entity is a swap-to-entity command, available only to the Game Master and only for entities that are not already controlled

by another player.

3.2.2 Movement

There are two methods of giving an entity a movement command. The first is from the direct input as detailed in the previous section. The other is input from the network. These work similarly but not entirely the same.

When handling input, all moveable entities are subscribed to events for both direct input and network input. Whether they do something with the direct or network input depends on the component they are using to handle it. For a player character, a component to handle the direct input event is used; for a remote character a component to handle the network event is used instead.

3.2.2.1 Pathfinding

With the movement event processed, a series of other methods are called to actually move the entity. The first step is the pathing step, which makes use of the other changeable component in the `Character` class. For direct input, the entity needs to figure out the path it is going to take to move from its current position to the newly selected one. The first step here is to check whether or not the entity is currently following a path. If it is following a path then its location may not be a whole tile, which would cause the pathfinding algorithm to break. In those cases where the entity is already following a path, its next tile destination is used as the entity location instead of its actual present location.

The pathfinding algorithm used is a library implementation of A*, detailed in appendix 1.1. This implementation provides what is known as the Manhattan method of pathfinding, which does not allow diagonal movement. Although this is not the most efficient form of pathfinding, it was decided as being the best option for a variety of reasons. The first of these reasons is one of gameplay; with the strict, tile-based nature of the game, diagonal movement feels somewhat odd. The second reason is one of artwork; in theory, having eight directions of movement creates a need for extra animations and graphics for the players—although in reality no animation system was put in, nor were proper character graphics made.

TODO: Manhattan pathfinding diagram

The final reason was to do with collision detection and rendering. Because of the way collision detection works, entities crossing each other diagonally actually end up intersecting with each other, which is an ugly graphical artefact. As was discussed in section 3.1.2.3 the work needed to make this a non-issue is considerable and there is little reason to allow for diagonal movement anyway.

TODO: OPTIONAL: diagram showing entities intersecting diagonally

While an entity handling direct input will find a path itself, those entities receiving input from the network instead get given a pre-determined path to follow. This is because it is possible for an entity to find a path on one client different from the path found on another. Although the game in its current form would not be overly affected by this, implementation of full interaction between

entities could prove those differences disastrous. For example, a client may think it has the ability to interact with an entity, even though it actually does not.

TODO: diagram showing entities taking different paths on different clients, just cuz

If a frame was responsible for handling direct user input and generated a path and movement, an event is created to inform the server that the entity is going to move and to supply the path it is going to take to get to its destination.

As an aside, currently, network input is simply taken from the client that created it and echoed by the server. Of course this is not ideal and in theory could produce terrible results due to malicious players. However, difficulties with the server implementation that are discussed in section 3.3 will explain why it was done this way. That said, despite the less than ideal situation this represents, the client would still create paths for itself even if the server were to ultimately override them. This is because waiting for the server to respond with a valid path would be far too slow. Responsiveness of the game is very important, and a user having to wait several hundred milliseconds for their action to be translated to movement on the screen is completely unacceptable. The client therefore needs to be able to simulate the movement itself, with the server responding with a canonical path that the client can swap to using when it gets it.

3.2.2.2 Following The Path

The path returned from the A* algorithm is a series of tiles. To follow this path an entity first takes the next tile in the list and sets that as its destination. Whenever it reaches a destination, the path is checked again and a new destination selected. Once the path is completed there are no more destinations to get and the function does nothing. This following of the path is done every frame, regardless of whether there is input or not, and the path does not have to be generated anew every time a destination is reached.

To actually move towards a destination, the entity uses the `updatePosition` function—the code for which can be found in appendix 2.1—which performs a series of checks to determine the entity's current location compared to its destination. When the direction of movement is established the entity's speed value is added to its position. This speed value is low, being just 0.125, taking eight frames to move an entire tile—though adjusting the speed value would allow an entity to move faster or slower.

3.2.3 Game Master

The Game Master has the special ability to select which character they want to control. This is only available on characters that are not already controlled by another player, and these types of character are represented by blue blocks in the game, rather than the red blocks representing players. The introduction of this ability caused a lot of changes in the way entities are put together and handled and is the reason the `Player` class was merged back into the `Character` class, as discussed in Section 2.3.3.

3.2.3.1 Swapping Characters

As mentioned at the end of Section 3.2.1, the Game Master can swap between characters by holding the shift key and clicking on an uncontrolled character entity. To handle this feature the `Character` class has two important methods—`processInput` and `updatePathing`—that can be swapped out depending on whether the entity is under player control or not.

The functionality to swap the methods out is implemented by making use of the fact that JavaScript functions can be stored in variables. Global variables are set in the files `components.js` which contain the functions used. Swapping between them is handled using the input components that are assigned to `processInput`. For example, the `CharacterInputComponent`, shown in Listing 3.7, handles the Game Master taking control of the entity.

```
1  var CharacterInputComponent = function(entity) {
2      while (entity.eventQueue.length != 0) {
3          var input = entity.eventQueue.pop();
4
5          if (input.type === "server_move" && input.data.entity_id === entity.id) {
6              return {"input": input.data.path};
7          } else if ((input.type === 'change_entity' || input.type === '
               server_change_entity') && input.data.x === entity.x && input.data.y ===
               entity.y && !entity.user) {
8              // GM changing to this entity
9              entity.user = USER.id;
10             entity.updatePathing = PlayerPathingComponent;
11             entity.processInput  = PlayerInputComponent;
12         }
13         return false;
14     }
15 };
```

Listing 3.7: Input component for handling server input applied to non-player-controlled characters.

As can be seen, this function handles the case where a non-player-controlled entity is given a movement command, but it will also handle the case where the Game Master is taking control of it. It does this by assigning the user attribute on the entity to the user ID of the Game Master, and then changing the `updatePathing` and `processInput` methods to use components designed to handle the direct input from a player. `PlayerInputComponent` handles the case where a Game Master is swapping to another entity instead by unsetting the user ID and changing the methods back to the character components.

3.2.4 Issues With Input and Movement

Input and movement as features were actually relatively smooth to implement, in comparison to the graphics and server code. However, they were not without their share of problems.

Input handling mostly caused problems due to taking a few iterations to work out the correct

calculations to convert between the various different coordinate spaces. However, this is was mostly an issue of delaying further implementation, rather than a serious problem. The only noticeable bug to come from input handling itself was a subtle one, where click coordinates appeared to shifted on the x axis by half a tile. This was solved by subtracting a half tile width from the click coordinate, as seen in previous Listing 3.5.

Movement posed a bigger challenge. The first problem was with the A* pathfinding library. Or rather, the problem was with a mismatch between the meaning of x and y in the game engine, and x and y in the library. In the game engine, y represents a column of tiles, with x representing the rows. However, in the A* library, this was the other way around.

Initially the problem was not noticed, as clicking a tile would still have the player entity move to it. However, subtle problems existed that became more obvious as further testing was done. The most obvious problem was that the pathfinder seemed to think entities were in places they were not. This meant that the player entity was finding paths that had it moving through other entities on the map, and avoiding empty spaces. The other problem was that, on the rectangular test map, the bottom part was unusable. Both of these problems were caused by the pathfinder's representation of the game world being rotated 90° compared to the game engine's representation of the game world.

The initial attempt to solve this problem was to change the game engine, rather than the library. This worked in some ways, with the terrain rendering correctly, at least. However, the rest of the game world became even more broken, with input failing to work at all. Ultimately, the solution was to go into the library and flip references to x and y. This didn't take very long, but it did fix all problems that the mismatch had caused.

The second problem movement had was with the code used to actually move an entity, seen in Appendix 2.1. Initially this code produced very odd results, with entities moving in multiple directions at once directly towards its end goal and occasionally getting stuck in a jittery state just before stopping in a tile.

The first problem was actually caused by a misunderstanding of the JavaScript array API. For some reason it was assumed the `pop()` function got the first result in the list, rather than the last. This was solved by changing the call to `pop()` with a call to `shift()` instead. The second issue was caused by a case where the addition of the speed value was causing the entity to slightly overshoot its target. On the next frame, the entity would then move backwards by the same amount, then move forwards again on the frame after that. This would repeat endlessly, leaving the entity stuck jittering in place. That was solved by checking to see whether the addition of the speed value to an entity's position would cause it to overshoot its target. If that happened, the entity instead sets its position to its destination, making sure it cannot get stuck.

3.3 The Server

The server was, in theory, to be no more difficult to implement than the client. Indeed, some of the server-side code was going to just be a Python version of code already written for the client. After

that, features like combat would have their logic implemented entirely server-side so there was no more duplication of functionality across them both, with the client being something more like a dumb view into the server.

Unfortunately, reality proved this theory to be badly false. The server took up an enormous amount of time, implementing things that had nothing to do with the game logic. Instead, handling the networking aspect as well as the management of the games themselves proved to be an incredible amount harder than expected. This was combined with the realisation that Python and JavaScript are very different languages, and even the code that was supposed to be replicated across the two proved more difficult to implement than anticipated.

This section will detail how the server works, handling connections with the client, the management of the multiple games that need to run at once and how the game logic is implemented server-side. The difficulties faced in doing these things will also be covered in detail.

3.3.1 Connecting Clients

The primary purpose of the server in the game is to provide a way of connecting clients together and thus allow people to play the game with each other. To achieve this there needs to be a way of managing those connections and passing messages between the clients.

Being a web-based game, the server is responsible for other things too. In particular it needs to be able to get players to the game in the first place, and to that extent is responsible for handling a lot of the UI and infrastructure that lets players log into the game, find games to join, and run games they have already joined.

To handle these aspects a framework was used, called Flask. Flask is "microframework", which provides built-in functionality to handle the routing of HTTP connections and the rendering of HTML templates. Further functionality is provided through extensions to the framework, which in this project provide database access and websockets, as well as a way of handling forms containing user input.

3.3.1.1 HTTP Routing

The majority of the routing work on the server-side is the handling of HTTP connections. This includes logging in and out, finding games and joining games. These routes also contain a lot of logic for managing games, with more detail on that in section 3.3.2.

The first responsibility HTTP routing has is logging players in and out. This is handled in a standard way for web applications, having a database table containing usernames and hashed passwords. When a user logs in they provide an email address and password combination in a form, which is POSTed to the server for validation. An extension to the Flask framework is used to manage the hashing of passwords and the validation of user accounts. If the validation passes, a session cookie is set that keeps the user logged in.

The route for a login can be seen in Listing 3.8.

```
1  @app.route('/login', methods=['GET', 'POST'])
2  def login():
3      from forms import LoginForm
4      form = LoginForm()
5
6      if request.method == 'GET':
7          return render_template('login.html', form=form)
8
9      elif request.method == 'POST':
10         if form.validate() == False:
11             return render_template('login.html', form=form)
12         else:
13             session['email']    = form.email.data
14             session['username'] = User.query.filter_by(email = session['email']).first()
                   .username
15             session['user_id']  = User.query.filter_by(email = session['email']).first()
                   .id
16             session['active_games'] = {}
17
18             return redirect(url_for('gamelist'))
```

Listing 3.8: Route for handling logins on the server-side.

Routing in Flask works by having a decorator, seen on line 1, that defines the URL that applies to the function that follows it. The decorator defines the URL relative to the root, so in this case the full route for `/login` would be `example.com/login`. The route can also define which HTTP 'verbs' it wants to handle. By default it handles only GET requests, but further ones can be defined as seen here.

The function itself can do anything. In this case, the function first imports a form class, which is used to define the data the incoming request will have, like an email and a password. Next, the route checks to see whether it is being called from a GET or a POST request. If GET, it just returns the form for the user to fill out; if POST, it validates a form coming in and does something based on the result of that validation. A failed validation will display the login form again and a successful validation will create a new user session and redirect the user to their list of games.

All routes to some extent follow this pattern, depending on their purpose, with some having the very important responsibilities of creating games or letting a user join an already existing game.

3.3.1.2 Socket Routing

The other type of routing the server does is that of websockets. To achieve this an extension for Flask is used, which implements the SocketIO library. Currently there are three types of event handled by sockets: movement, connection and disconnection.

```
@socketio.on('player_move', namespace = '/game')
def player_move(data):
    if data['game_id']:
        running_games[data['game_id']]['input_queue'].put({'type': 'player_move', 'input
            ': data})

    emit('server_move', data, room = data['game_id'])
```

Listing 3.9: Socket for passing movement events between clients and server.

Listing 3.9 shows the socket responsible for passing movement around between clients and informing the server. As can be seen, there is a decorator responsible for defining what the route is, followed by a function to handle the route—the same as the HTTP routing. The difference is, of course, that this handles websockets using SocketIO, rather than handling HTTP connections.

The route is defined as being a socket name—in this case `player_move`—and a namespace. The namespace means that there can be multiple sockets of the same name being passed around that do different things. As an example, if there were a socket called `message` that existed in two namespaces, `/game` and `/chat`, they would not interfere with each other.

This route's function has two purposes. The first is that it informs the server about the movement so that it can simulate it. How that is achieved is detailed in Section 3.3.3. The other purpose is to then inform the clients connected that a movement has occurred, using the `emit` function. This emits a socket message that contains the data about the movement, in this case which entity moved and the path it is taking to do so.

To make sure that clients don't receive messages that are from different games, the concept of a room is used. These rooms are provided by the SocketIO library and contain a list of socket connections with the key being the game ID it relates to. Because the game ID is always sent with a socket message, the rooms can be used to make sure that only clients connected to that particular game will receive messages from it.

Connection and disconnection don't emit anything back to clients. Instead they simply inform the server-side game simulation that a user has connected or disconnected and leave it up to the game to decide what to do with that information. Section 3.3.2 details how this works.

3.3.2 Managing Games

As important as routing is to the functioning of the game, a huge amount of logic server-side has the responsibility of managing the games themselves. Managing the games refers to creating entirely new games, starting the game loop on the server and making sure players are placed into the right ones. This section will also deal with how games are shut down, which the game process itself is responsible for handling.

3.3.2.1 Multiple Games And Threads

One of the major reasons that the server-side proved more difficult to implement than expected was the necessity of running the game loop in a separate thread. The reason for this is that the game loop is blocking, which means that once it starts nothing else can run outside of it. If it were not run in a separate thread the server simply wouldn't function—as a web server or for having multiple games running.

To handle the threading functionality the game loop is contained within a class called `GameLoop`, which inherits from the class `threading.Thread`. The constructor for `GameLoop` first calls the superclass' constructor to activate the threading capabilities, and then continues with its own constructor.

The constructor in `GameLoop` does a few things. Firstly it creates instance variables linking it to the queues used to communicate with the router and then goes on to create the entities in the game that it needs to keep track of. For threading the entities don't matter very much as they are kept in an `entities` variable in the class itself. However, the queues are important; without the queues the game loop would simply run on its own, isolated, and never do anything because it couldn't receive any input.

The queues are instances of the Python `Queue` class and are created by the router when it first starts up a new game loop process. They are then passed into the `GameLoop` constructor and also kept around in the router so it can use them too. There are two queues, one for input from the router and one for sending information back out, known as `input_queue` and `reply_queue`. The input queue is read by the game loop every frame to check for messages it needs to handle. The reply queue is only used in specific circumstances when the router expects to get a response. Because the router cannot have its own loop and still function correctly there is no reasonable way to regularly check for replies from a game. This means that the game loop can only respond to input and cannot generate events by itself because those events would never get seen. Luckily, this isn't such a big issue for the game as there is no AI aspect to generate events without player input.

Finally, the game loop sets a variable called `alive` to true. This variable is used by the game loop itself to keep running. When the variable is set to false the loop will stop running and the thread will ultimately end. Starting the thread, however, is done by the router calling a method called `start()`, inherited from the `Thread` superclass, which calls the `run()` method defined in `GameLoop`. If `run()` is called directly a new thread won't be started, which will cause the game loop to block the server.

Because there are multiple games running at once in separate threads it is necessary to have a a system in place that can manage those threads, starting them up and directing connecting players to the correct one. This functionality is given to the router, as all of these functions are responses to connections and requests made by users.

3.3.2.2 Creating Games

Before games can be started and joined, they first have to exist. From a game design point of view this is the starting point of the Game Master, but from a technical point of view it is the creation of a basic terrain and the addition of some entities to the world.

Game creation is started from the route `/game/create`. A user would get here by clicking a link on their game list page. They then proceed to enter a name for the new game they are creating and hit submit. Once the form is submitted with the new name the route proceeds to create a predefined, default world with a small map and three entities—two walls and a userless character that the Game Master can take control of. The map and these entities are committed to the database. The user that created the game is set as the Game Master for it, gaining whatever privileges that entails.

3.3.2.3 Starting And Joining Games

Starting up and joining a game come from the same route, seen in Listing 3.10.

```
@app.route('/game/<path:id>')
def game(id):
    if 'email' not in session:
        return redirect(url_for('login'))

    current_game = Game.query.filter_by(id = id).first()

    session['active_games'][id] = True

    import game, Queue
    if id not in running_games:
        scene = Game.query.filter_by(id = id).first().current_scene
        entities = Entity.query.filter_by(scene = scene).all()
        input_queue = Queue.Queue()
        reply_queue = Queue.Queue()

        running_games[id] = {'game': game.GameLoop(entities, scene, input_queue,
            reply_queue), 'input_queue': input_queue, 'reply_queue': reply_queue}
        running_games[id]['game'].start()

    return render_template('game.html', current_game = current_game)
```

Listing 3.10: The route responsible for starting a game thread.

When a player decides to join a game, they go to this route, the URL being in the form `/game/<id>`, with `id` representing the game ID they wish to join. The route first checks to see if the user is logged in and, if so, it goes to the database to get the game from that ID. The final step before actually handling the game starting itself is to add the game to the user's session. This is necessary to handle disconnect events as will be seen in Section 3.3.2.4.

The next part deals with starting a game. First, the `running_games` variable is checked to see if the game is already active. This is a global variable held by the router and is different from the session variable linked to each individual user.

If the game already exists the route just returns the page that runs the game in the client. If a user is the first to try and join the game, however, then the game needs to be started up. To do this the router first gets the current scene and then the list of entities linked to that game from the database, followed by the two queues it needs to communicate with the game loop once it's running. It then creates a new entry in the `running_games` list, and in doing so creates a new instance of the `GameLoop` class, passing in the scene, entities and queues. It also attaches the queues to the `running_games` list so that it can get them later. Finally, the game is started and it then renders the page containing the game for the user.

To actually get game information to the client another route is needed. This route gets what is known as the scene, which contains the map and the entities that are in the game. The full code for this route can be found in Appendix 2.2.

The route first checks to see if the game is running. If so, it asks the game to return its current state. Doing this means that clients that are connecting after the game has started are getting the state of the game as it currently is, rather than what was previously persisted to the database. However, it still needs to grab information from the database about entities because not everything is kept in the server-side state. For example, which sprite represents an entity is not stored server-side because it is meaningless information there. Terrain is also acquired from the database as the database version is always the current one. It then goes through every entity and makes sure it has the current information instead of the old persisted information and then packages it all up as JSON to send back to the client.

The final step of joining a game is the client sending a socket message called `game_loaded`. Upon receiving this event the server-side game is informed that a new user has connected and the user is added to the socket room for that game.

3.3.2.4 Disconnecting

In theory disconnecting is a simple part of the implementation, especially as SocketIO provides disconnect events itself. However, in reality the disconnection of users proved to be one of the most challenging parts of the server implementation, primarily because of the game being hosted in a browser and the need for empty games to be shut down automatically so as not to leave unused games lying around.

Individually neither problem is particularly hard to solve. Together, though, they are a serious issue, and one that was not at all foreseen when considering the creation of a server hosting a web game. The issue stems from the fact that when running a game from a browser users can refresh the page. This causes a disconnect event but in reality the user only replaced their old connection with a new one. Worse, the disconnect event is delayed, sometimes for up to 15 seconds, due to SocketIO only noticing a disconnect has occurred when a client fails to reply to a heartbeat check. When keeping track of how many users are connected to a game to know if it is necessary to shut

it down, this fact can cause a player to end up in a situation where the game server isn't actually running even though they are supposedly playing the game.

The first step to solving these issues is to get the disconnect event itself. This is handled through a built-in socket message called `disconnect`, provided by SocketIO. It is at this point that the storing of the active game in the user's session becomes necessary, as the disconnect event cannot carry extra information, such as what game the disconnect is relevant to. When this event is received, the router passes it along to the game, which is keeping track of all the users currently connected to it.

Because users can refresh the page and thus create false disconnects, when adding a player to a game the game keeps a count, which is incremented whenever a player add event occurs. When a user disconnects, their count is decremented instead. This has the effect of keeping track of the difference in connection events and disconnect events, so that when the disconnect events come through they are not taken to mean the user has actually disconnected.

If the user connection count does happen to reach 0 and the user in question is the only one connected then the server will proceed to shut itself down. This is done by unsetting the `alive` flag, which will stop the game loop from running the next frame. Once this is done the game proceeds to ask every entity in it to perform its own shutdown function, which has the entities make sure they are on a whole tile rather than transitioning between. Once this is done the entity is persisted to the database so that its state can be saved for the next time the game is loaded.

When all this is done the thread has nothing left to do and so it stops itself. The next time a user attempts to connect to the game it will go through the process of starting up again.

3.3.3 The Server-side Game

Although figuring out the management of the games took up most of the time dedicated to the server-side—and ultimately the project in general—the game still runs and there is logic dedicated to actually getting the game to function properly.

Mention has already been made of the `GameLoop` class and its role in handling the management of the games, both directly and in terms of forcing certain decisions to be made. However, no mention has been made of exactly how it works, particularly with respect to its differences with the client-side loop.

On the client-side it is necessary to break the loop to allow the browser to perform various actions, such as updating the page. This means that if the loop were implemented as a normal `while` loop the game state could be changed but it could never be shown, as it would be blocking the browser from rendering anything. To solve this the loop is instead implemented by using `setInterval`, which calls the loop function every 33.3 milliseconds.

The server-side doesn't operate in the same way, as Python does not, by default, run inside an event loop like JavaScript does. There are ways to implement one using libraries such as Twisted **`TODO: reference`**, but it is not easy. Instead, the server-side uses a traditional `while` loop and thus takes complete control of the thread. This primary loop operates as long as the `alive` variable—described in the previous section—is set.

Every time this main loop goes round it performs some timing, keeping track of the time elapsed since the last time it ran. This elapsed time is added to a variable called `lag`. When lag reaches 33.3 milliseconds or more then another `while` loop is started that runs as long as `lag` remains above this time, reducing the lag time by 33.3 milliseconds every time it runs through. This is pretty much the same as how the client operates and handles the concept of frames running at a specific interval.

During a frame the server checks the input queue, filled by events from the router. The events that it can handle directly are for adding or disconnecting players, returning the entities it has running for the sake of a newly connected user, as well as an event to manually stop the thread. The functioning of these is discussed in the previous section. If none of these events is relevant the loop passes the input onto the entities in the game, calling each entity's own update function. This is the same concept as used in the client-side to handle frame updates.

The entities on the server-side function in a similar way to the client-side, though unfortunately their functionality is slightly reduced as they cannot find paths for themselves, which means that they cannot act as the canonical version of the game state they are supposed to be. Although adding a pathfinder to the server-side would not be too difficult a problem—simply being a case of adding a library similar to the client-side—it is not quite as simple to make sure the client receives these new paths and handles them smoothly, and time ran out dealing with the rest of the server. Despite the lack of ability to create their own paths, however, entities on the server are able to follow paths given to them from clients and handle their movement between tiles in exactly the same manner as the client. This allows the server to at least offer up the game state to newly connecting or refreshing clients.

Chapter 4

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

1.1 A* Pathfinding Library

Blah blah I modified this because stuff.

Appendix B

Code samples

2.1 `updatePosition` Function

This function is used by entities in the game to move across the world. A destination is set from nodes created by the pathfinding algorithm, as referenced in appendix 1.1.

2.2 Route To Acquire Scene

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

Minimal System specification goes here.

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] 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.

[3] ——, *Starling Game Development Essentials*. Packt Publishing, Dec. 2013. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/178398354X 178398354X.

Book about isometric game development in an ActionScript framework. The algorithms to convert from Cartesian to isometric coordinates and back are taken from this book.

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

BrowserQuest source code.

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

The BrowserQuest game itself.

[6] Little 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.

[7] 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.

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

Online APIs for JavaScript and HTML5.

[9] 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.

[10] 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.

[11] M. A. Pagella, *Making Isometric Social Real-Time Games with HTML5, CSS3, 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 and JavaScript

[12] 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