MageCraft

Ricards Graudins

B.Sc.(Hons) in Software Development

April 16, 2018

Final Year Project

Advised by: Martin Hynes

Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

About this project

Abstract Over the past three decades video games have drastically improved in all aspects and with hardware getting upgraded approximately every two years according to Moore's law, video games will only continue to reach new limits as time passes. In recent years augmented reality has taken a giant step forward and with the release of new virtual reality tech the gaming industry will continue to grow which is great news for anyone who partakes in gaming as a hobby. In this study we attempt to create our own variation of a multi-player video game called MageCraft that was released in 2011. One of the goals of the project is to rebuild and try to improve the game using newer technology which in turn will allow us to learn how to use the new technology and of its limitations. MageCraft was selected over other games because it's not that much different from the current games that have been dominating the market for a while and that provides a great learning opportunity, the knowledge gained from having completed this project will without a doubt prove useful in the development of future games.

Authors

Ricards Graudins is a student studying Software Development at the Galway-Mayo Institute of Technology for the past 4 years (2014 - 2018) from Latvia, Riga. Enrolled into a computer science course out of interest and passion for developing video games, eager to leave a positive mark in the computer game development industry.

Chapter 1

Introduction

In this chapter we aim to answer the following questions:

- What is MageCraft and what is it all about?
- Who came up with the idea for MageCraft and developed it?
- When, where, why, and how was MageCraft created?
- Why did we choose this for the project?

MageCraft is a multi-player arcade game built on the StarCraft 2 [1] engine that was developed by Blizzard Entertainment [2]. The engine was specifically designed for StarCraft 2, it took approximately 5 years to perfect (2005 - 2010) and 16130 builds according to the following source [3] which also demonstrates how the engine improved throughout all those builds. On July 27th 2010 when StarCraft 2 was released, everyone who purchased the game had access to the engine and some people started creating various mods that either improved certain aspects of the game or created mini games such as MageCraft. Although it wasn't until May 12th 2012 [4] when Blizzard officially released arcade mode which essentially promotes mods and allows players to play them without going to 3rd party websites, downloading mods and then modifying their own version of StarCraft 2 in order to play them. With the release of arcade mode, all of a sudden all these mods were gaining popularity and arguably some of those mods were actually more entertaining to play than the actual base game itself and near the top of the list was MageCraft with an already established player base since it came out a year earlier with frequent updates and content which in turn attracted players. To be precise, MageCraft itself was released near the end of March / early

April 2011 [5], developed by a small group of developers called Team Syntax whom have since moved on after the success of MageCraft forming a small indie company that mainly focuses on creating games using the Unity engine and Unreal Engine 4 [6]. Their original goal with MageCraft was to create a battle arena that was based off a mod from WarCraft 3 called "Warlock" and with the release of the StarCraft 2 engine it provided them with all the resources they needed to create a fun, interactive, skilled-based game [7].

1.1 Gameplay

In terms of game play - each player controls a mage with the objective of being the last remaining survivor in a free for all mode where the battle arena is surrounded by lava, the lava creeps in over time and eventually swallows the majority of the arena except for the remaining 4 planes. One of the primary goals of the player is to stay alive and this becomes more and more difficult since the arena shrinks limiting the area where the player can move while avoiding getting hit by the enemy spells and avoiding lava which deals a tremendous amount of damage if caught in it. There are a variety of different spells to choose from and certain spells synergize/combine for a more devastating spell, however each player is limited to 6 offensive spells and 2 defensive spells - once picked, the spells are locked for the remainder of the match therefore each player should consider carefully which spells to take in order to counter the opponent/s and achieve victory. Some of the spells push the players backwards upon collision and in fact the most effective way to win the game is to push other players into lava with spells, once the other player is on the lava you can further cast other spells to push the other player even further or block off the path to get back on the arena. The game is round based meaning in order to fully achieve victory the player has to be the last surviving mage multiple times such that at the end the player with the most amount of round wins, wins the game. Between each round the players have the option of purchasing spells or upgrading their selected spells to do more damage, also the arena gets reset to it's original number of planes i.e. if in the previous round the majority of the arena got covered with lava - it gets reset. Naturally the aim of this project was to recreate MageCraft however we have adapted some of its game concepts such that it still reassembles MageCraft but it differs in some ways e.g. some of the spells are modified, different player controls etc. All of these changes are expanded upon and explained in chapter 5.

1.2 Why MageCraft?

This project was chosen for a number of reasons, primarily the passion for video games and the fact that this course lacks game design modules made the project interesting and provided a great learning experience as to how multi-player works and in general just how video games are made among other things. The idea to develop MageCraft in particular arose from the fact that the original game was abandoned around 2015 and has since been highly outdated to the point where several game concepts are broken and made the game unplayable. Upon further investigation we arrived at the conclusion that MageCraft is not much different from the various games that have been dominating the market for years e.g. League of Legends [8], DOTA 2 [9], Heroes of Newerth [10], SMITE [11] etc. This realization made this project an ideal opportunity to learn how to create and how all of these incredible games function in the first place, using the knowledge from creating this game could provide us the opportunity of possibly working or maintaining one of the top games that are already released or may be in the future. Another main reason for choosing MageCraft is the fact that StarCraft 2 has been losing its fan base steadily over the past few years and arcade games built on that engine are eventually abandoned and forgotten but recreating some of the more popular arcade games can ensure that fans of those games have a bit more time to enjoy playing them in a new light and reminiscing of the good old days.

Chapter 2

Project Context

The objective of this section is to outline the objectives of the project as well as to briefly describe the GitHub repository [12] contents. Section 3 Methodology, covers how we went about the project i.e. planning, meetings, testing, frequency, research etc. Section 4 Technology Review unveils all the various technologies used in this project as well as why they were chosen. Section 5 System Design provides a detailed explanation of the overall system architecture and is followed by System Evaluation which evaluates the project against the objectives set out in this section. Last but not least, Section 7 Conclusion summarizes the project and highlights the outcomes of the project.

The objectives of the project are as follows:

- Player Movement: Create a character for the player that can be moved either using mouse input or keyboard input.
- Player Spells: The player should be able to cast a variety of spells that are different from each other and since they are a very major part of the game they should be interesting and fun to use.
- User Interface: Create a UI that displays all the various spells the player can cast as well as their cooldowns i.e. how long before the spell can be cast again. The UI should also display player health points, status effects and gold earned.
- Spell System: Implement a system that enables players to switch and select which spell to use. The system should also be able to track spell cooldowns and if the spell is on cooldown prevent the player from casting that spell.

- Collision Detection: Implement a system that detects collision between the players, AI and spells.
- Multi-player mode: Create a mode where up to 8 people can connect over the Internet and participate in a free for all brawl. Ultimately this objective was not met due to scope underestimation and eventually was replaced with "Tutorial mode". Although this objective was not completed, the ideas behind it were explored and are further outlined in section 7.
- Tutorial mode: Create a mode where a single player can learn how to play the game at their own pace versus a narrow AI.
- AI: Create an AI that moves towards the player and upon collision the player takes damage.
- Cloud: Host the project up on a suitable cloud service such that the game runs smoothly.
- Account System: Create an account system that allows users to register, login and preform a variety of other account related services. This system should be developed properly such that all the user data is stored in a secure manner.
- Chat System: Implement a chat that allows players to communicate and have certain input commands entered via the chat change game settings.
- Animations: Add animations for the player, spells and the AI possibly using Blender to create models.
- Menus: Add menus(pause menu, game over menu) that allow the player to pause the game, resume, restart, mute music and return to main menu. The main menu should allow the player to select the game mode and log into their account.
- Background: Set up the game environment background i.e. animated lava, the player should be able to move on this layer however it deals a huge amount of damage as long as the player remains on it.
- Player Grid: Create a grid which sits on top of the lava layer and as the game progresses shrinks in size until only a few planes remain. The code responsible for creating the grid should be adaptive i.e. can easily switch to a different size depending on the number of players in a game.

The GitHub repository [12] contains the entire code for this project as well as the following content in the readme file:

1. Short Intro: Describes what the project is all about and is basically a summarized version of the introduction section provided in this paper.

2. Cloud URL: A link to the page where MageCraft can be played which is hosted using Heroku.

3. Demo URL: A link to a short youtube video that demonstrates the game play.

4. Technologies Used: A summarized version of the technologies used for this project, however it lacks the reasons for why the those technologies were chosen, refer to section 4 of this paper.

5. Bugs Section: A short section explaining why the game does not run correctly on FireFox version 58.

6. How To Run Locally: This section outlines the steps necessary to run the project locally, it lists all the prerequisites that must be installed and provides links / instructions on how to set up all of them within a short period of time as well as the command necessary to run the project.

7. Commit Summary: This section briefly outlines every major commit, they are numerically labeled which allows anyone to easily find when the commit occurred and all of the changes that happened as a result of the commit via the commit history feature provided by GitHub.

8. References: This section provides direct links to the technologies used in this project which contain additional information regarding said technologies and download links.

9. Code: The code itself is laid out using the generic flask layout [13] and all of the folders and files are labeled and contain commented code at the top which explains their purpose except for procfile, config.cfg and requirements.txt which are necessary files used on the Cloud.

Chapter 3

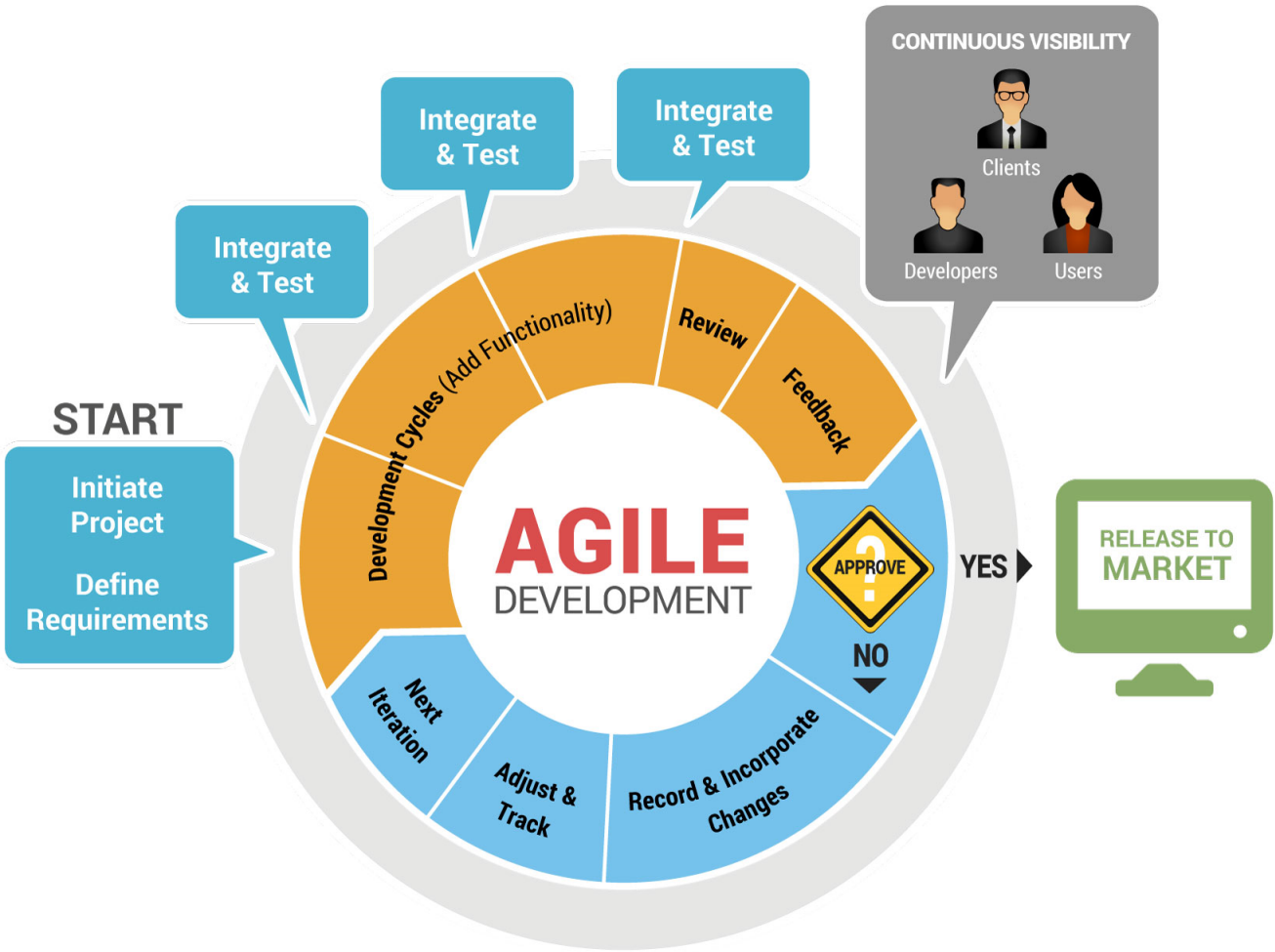
Methodology

The methodology used for this project was agile. After getting the project idea approved alongside the various technologies to be used, we decided to work on the project for a minimum of 10 hours a week which can be described as a sprint to get a certain feature working. Every week we would meet with our supervisor and discuss the work completed during that week and outline how we went about implementing that particular feature/goal which allowed us to get constructive feedback and these meetings proved very useful throughout the development of MageCraft. After having reviewed the work completed during the week we would discuss the course of action for the following week and having discussed that with the supervisor we would get a critical assessment that told us if the set goal can be completed within the week or if the goal was being underestimated and needed more than a week to implement this point in question gave us an accurate second opinion and allowed us to plan the goal more precisely. Furthermore, before every meeting the feature that was being worked on was committed to GitHub with a decent level of documentation and a summary of the feature being added was also available to anyone interested.

Before coding started, approximately 2 weeks was spent on researching and selecting the appropriate technologies to be used, a lot of consideration was placed into selecting the technologies and deciding if they were in fact the correct ones to use out of several alternatives. Thereafter a list of all the project goals was created and followed throughout development with the list being split into smaller achievable tasks every week. One of the main objectives to be completed was set out during the October presentation - have the entire backend of the project completed before winter holidays and this objective was met however at a cost which led to a halt in the project development until mid January due to playing catch up with the other course

modules. After the development interlude, the agile approach was back on track however the sprint intensity amplified in order to reach the initial goals that were laid out.

The Agile Methodology Development Life Cycle [14]



The above diagram displays a ”release to market” label, in our case it was committing the feature to GitHub if it passed the iteration cycle. In terms of customer feedback, we ourselves determined if and when the feature was of befitting quality, secondary opinions were provided by the supervisor and any colleagues that were interested in the project progress.

3.1 Validation and Testing

In terms of validation and testing; each segment of code was thoroughly tested throughout development to see if the feature being implemented worked as intended without any unexpected implications on already existing features and if the particular feature was suspected of having a negative effect on an already existing feature, then the already existing features would also be tested for any abnormalities. In order to help spot any issues/errors that may slip by the debugger we implemented a debug layer provided by BabylonJS [15] which when enabled, allows the inspection of meshes relating to BabylonJS. Prior BabylonJS version 3.0 the debug layer was a very simple HTML interface helping the developer to retrieve some basic data about the project however with the new version the debug layer was replaced by a complete inspector tool that can be used to browse an immense amount of data which in turn helps find issues e.g. a mesh is assigned to a particular material which has it's properties changed later on in the code and the mesh may be affected in an unexpected way which is where the debug layer comes in and helps identify which properties are set accordingly or improperly.

3.2 Error Handling

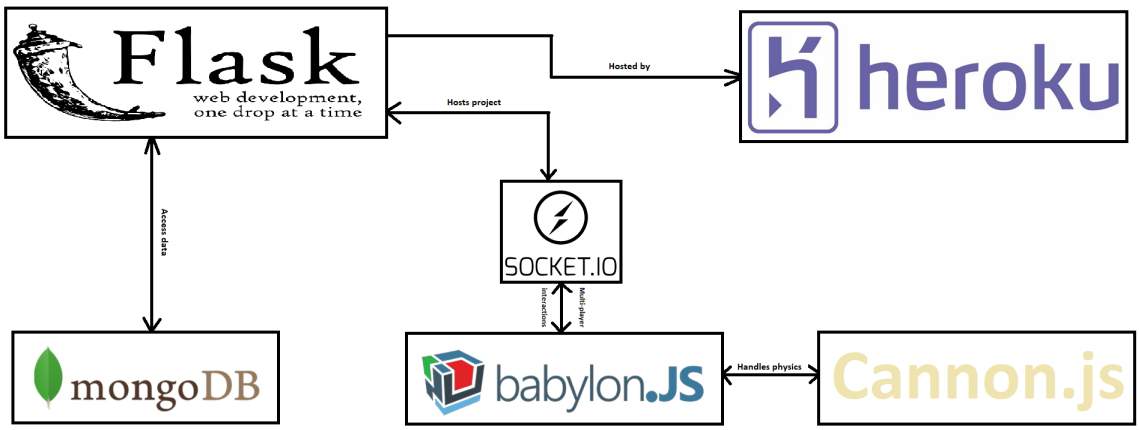
Most of the project issues arose when using BabylonJS since it was the majority of the project and then there was SocketIO which severely lacks any proper tutorial relating to developing multi-player games, there is a vast amount of SocketIO chat tutorials however they were far too basic to be of much use in developing multi-player game concepts which is the reason why the multi-player part of this project was eventually proclaimed out of scope. In the event an issue arose while using BabylonJS and debugging using the debug layer or reading the official docs did not provide a solution / workaround, the majority of the issues were solved after researching the issue, and in most cases the solution to the problem led to the BabylonJS forum [16] where users faced similar issues.

There were also a few problems that arose when developing the backend however using the documentation provided by Flask for each extension as well as the official documentation for those extensions provided more than enough information as to how to properly implement features and avoid issues.

Chapter 4

Technology Review

In this section we aim to cover all the various technologies used in this project as well as why they were chosen over the alternatives. The following diagram depicts the main technologies used:



4.1 Flask

Flask is a micro web framework written in Python and based on the Werkzeug toolkit and Jinja2 template engine [17]. Flask is called a micro framework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented

in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools. Extensions are updated far more regularly than the core Flask program. Flask was the first technology I chose for this project and it wasn't a difficult choice since having used this technology in several past projects I concluded that it is one of the best web frameworks available. It only takes about 5 lines of code to get a basic application running which allows the developer to quickly set up flask and move onto testing other technologies. There are quite a number of extensions to choose from and in our case using several of them allows us to create a secure and fluid backend which handles hosting several HTML pages as well providing a secure account system.

The alternative to using Flask was Django, Tornado or even both at the same time since they are complementary but not in the same process, essentially the reason these 2 were a candidate for the web framework was because while in the research stage, numerous references to the duo were mentioned when it came to developing real time multi-player games. Tornado is a web server and a web framework but it is quite a minimal framework when compared to Django - the Tornado modules are loosely coupled so it is possible to just use the web server component however there are a lot of areas where Tornado doesn't have as much functionality e.g. lack of model abstraction. Django on the other hand is not asynchronous, meaning that the project will block when using Django components. The asynchronous part is a big deal for this project for multi-player since we want the server to preform a variety of tasks for each user at the same time so in order to achieve this a Django project must first be created and then have Tornado functionality added to it. The following source [18] provides a simple way of creating a basic project using both technologies.

Eventually we decided to use Flask over the duo because as mentioned earlier, it is very simple to get a basic application up and running within minutes whereas having no prior experience with the duo would have resulted in a lot time being used in order to get the basics running. With a hasty fundamental application set up, it allowed us to start working on the more important features using extensions. To further expand on Flask, the word "micro" does not mean that the whole web application has to fit into a single Python file although it can, nor does it mean that Flask is lacking in functionality. The "micro" simply aims to keep the core simple but extensible via extensions. Flask does not make major decisions that force the developer to build an application in a particular way i.e. you aren't forced into using a

specific database or how the server responds to client requests etc. With the freedom of being able to mix and match extensions without being specifically forced down a particular path, provided us the effusive ability to choose the following extensions:

4.1.1 Flask-SocketIO

This extension gives Flask applications access to low latency bi-directional communications between the clients and the server which is a must for real time multi-player. The client-side application can use any of the SocketIO official client libraries in JavaScript, C++, Java, Swift or any other compatible client to establish a permanent connection to the server. The reasoning as to why this extension was chosen is further explained in the "SocketIO" section of this chapter.

4.1.2 Flask-Mail

This extension essentially provides a simple interface to set up simple mail transfer protocol (SMTP) with the Flask application and to send messages from the views and scripts. SMTP is an Internet standard for email transmission that is used across the board nowadays that was first introduced in 1982 and last updated in 2008 with extended SMTP additions by RFC 5321. SMTP connections are secured by transport layer security (TLS) which is a cryptographic protocol that provides communications security over a computer network, commonly used in applications that are used for web browsing, email, instant messaging and voice over IP. Although popular web-mail systems such as Google mail and Yahoo mail have their own non-standardized protocols for accessing mail box accounts on their own mail servers, all of them use SMTP when sending or receiving mail from outside their own systems. Flask-Mail was the chosen extension precisely because it is a simple interface to set up SMTP and there is no reason whatsoever to create our own protocols when SMTP is widely used globally. [19]

4.1.3 Flask-PyMongo

This extension bridges Flask and PyMongo, so that the developer can use Flask's normal mechanisms to configure and connect to MongoDB. PyMongo is a Python distribution containing tools for working with MongoDB and it is in fact, the recommended way to work with MongoDB when using Python. The alternative to using this extension would be to use the PyMongo library directly but that doesn't provide any benefit and just carries out the same

purpose. Similarly to SMTP, PyMongo uses TLS and prior to that used SSL which is now prohibited by the Internet Engineering Task Force (IETF), SSL is basically the predecessor to TLS.

4.1.4 Flask-WTF

This extension provides integration between Flask and WTForms, including cross-site request forgery (CSRF), file upload and reCAPTCHA. CSRF is basically an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. These attacks specifically target state-changing requests, not theft of data since the attacker has no way to see the response to the forged request, essentially a successful attack can force the user to preform a state changing requests like transferring funds, changing email address, deleting their account and so on. Although this project doesn't store a lot of user sensitive information other than the registered email address, we thought it would be interesting to see how to integrate WTForms and keep user data secure. To build up on that security, we added reCAPTCHA which is a CAPTCHA-like system designed to determine whether the user connecting is actually a human or a bot trying to log into the website. There are bots out there that are capable of solving the reCAPTCHA but a lot of unsophisticated bots are stopped in their tracks when faced with a reCAPTCHA feature, even if a bot manages to solve the reCAPTCHA it still has to enter an appropriate username and password and if we wanted to it is possible to implement another level of security that tracks how many times a person/bot has failed to enter their details from a specific IP address and after a number of tries the system can timeout/prevent the IP address from accessing the login permanently or for a set duration.

4.2 SocketIO

As briefly outlined earlier in the Flask section, SocketIO is a library for real-time web applications, in this case we're discussing the JavaScript library used on the client side that communicates with the Flask-SocketIO extension. Naturally we're using the JavaScript library to communicate with the server since the majority of the code is already in JavaScript and there is no reason to over complicate things. SocketIO enables real-time, bi-directional communication with between client-side library that runs in the browser and a server-side library, both components have nearly identical API. SocketIO is event-driven, meaning the flow of the program is determined by events such

as user actions e.g. mouse clicks, key presses etc. In our case - multi-player wise, whenever a player moves their character, casts a spell or types a message into the chat the result of doing so should be updated for everyone who is connected via the sockets to that particular game i.e. player 1 is at the following coordinates (0,0), player 1 moves their character to (1,1), that character's position should be updated to (1,1) on everyone's screen very quickly to ensure a real-time experience. This sort of real-time experience cannot be achieved using the traditional method where the client sends a packet to the server and waits for a response over and over again, this would simply be too slow which is the reason why we're using sockets in the first place.

There are a number of alternatives for SocketIO and we went with SocketIO for the following reasons: It's simple to use once the user understands how it works - it basically abstracts the web socket connection similarly to how jQuery Ajax abstracts xml HTTP requests. It provides fallback in situations where web sockets cannot be created, in those situations SocketIO falls back to the HTTP connection gracefully making it consistent. The main advantage of SocketIO is that it simplifies the usage of WebSockets and provides more features the developer can make use of and it is generally recommended to use SocketIO over directly using WebSockets unless the developer is very familiar with those environments and knows their limitations. During the research stage we were concerned that perhaps SocketIO wasn't fast enough for multi-player since it uses Transmission Control Protocol (TCP) instead of User Datagram Protocol (UDP). UDP permits individual packets to be dropped and to be received in a different order than that in which they were sent which in turn allows for better performance, as such UDP is widely used in video conferencing and real-time computer games. Whereas TCP packets ensure that the data being sent is delivered without failure and in a particular order, meaning that if a packet gets lost all of the packets sent afterward must wait for that particular packet to arrive, if it doesn't - the packet will be re-sent with every packet that had been sent after it. The retransmission causes the overall throughput of the connection to drop. Some might say that the Internet is now pretty fast and reliable however that is debatable - the bandwidth has dramatically increased but latency is still quite high. Essentially the problem comes down to deciding whether you want the game to be reliable(TCP) or very fast(UDP). TCP is fine for a strategy game where each player takes a turn however UDP is preferred for real-time games, we would have liked to use UDP for MageCraft however the technology just simply isn't quite there yet for browser games since web browsers are built on top of TCP, as such we have defaulted to using SocketIO which uses the WebSocket transport and WebSocket is a TCP connection not UDP.

4.3 Heroku

Heroku is a cloud platform as a service (PaaS) that supports quite a few programming languages used as a web application deployment model. The service started in June 2007 when it only supported Ruby, since then it has been called a polyglot platform since it lets the developer build, run and scale applications in a similar manner across a number of languages. Applications that are run on Heroku typically have a unique domain name following this type of naming convention - applicationname.herokuapp.com. Each of the application containers or dynos are spread across a dyno grid which consists of several servers and Heroku's Git server handles application repository pushes from users. The service works practically the same way as GitHub, simply download the Heroku toolbelt or the new Heroku CLI, create a git file for Heroku and then push changes to your registered Heroku repository, a repository is created automatically if you don't have one available. The great feature of Heroku is that it lets the developer manage the application entirely through the command console and you can do things like scale the application or view logs and so on.

During the research stage we aimed to select a cloud service that would be fast and keep the latency at a minimum, thus early on we tested a number of cloud services - Amazon EC2, PythonAnywhere and Heroku. Amazon EC2 was ruled out fairly quickly since past experiences deemed it to be rather slow to load resources and in this project theres quite a bit of that which comes with BabylonJS. PythonAnywhere was tested next and we had high hopes for it since the service is designed specifically for python applications however due to its complexity we moved onto testing Heroku and we got the idea to use Heroku after researching last year's projects and this particular cloud service came up quite often but wasn't covered in any of the modules. It didn't take a long time for us to set up Heroku and we deemed it good enough for this project. In the event MageCraft needed to scale up, it would be simple to scale the application by simply changing the dyno type / size on Heroku which also provides a variety of statics to determine which dyno is the correct one for the application.

4.4 MongoDB

MongoDB stores data in flexible, JSON-like documents, meaning that fields can vary from document to document and the data structure can be changed over time. The document model maps to the objects in your application code,

making data easy to work with. Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze the data. MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use. Once we had selected Flask we intended to select a database for the account system that would work fluidly with Flask and we concluded that the data structure should be easily changeable since we didn't know what kind of data we would like to store later on other than username, password and email. Having covered Flask and a number of databases that work flawlessly with the technology in a past module called Data Representation and Querying - choosing the database was more of a choice based on personal preference and one simple requirement i.e. being able to change the data structure later on and as such SQLite and sqlalchemy were ruled out since adding new data to an already existing table can get quite chaotic and tedious if there is a large number of already registered users. The other 2 alternatives were CouchDB and PouchDB. PouchDB is basically an open-source JavaScript database inspired by CouchDB that enables applications to store data locally while offline, then synchronize it with CouchDB and compatible servers when the application came back online. PouchDB was ruled out precisely because it's JavaScript based - the majority of the code was already going to be JavaScript and we wanted to keep the backend services in the same programming language i.e. Python. CouchDB on the other hand is not much different from MongoDB since it keeps its data stored in JSON with the major difference being that MongoDB focuses on consistency and partition tolerance whereas CouchDB focuses on availability and partition tolerance, availability in this case meaning the same synchronization service offered by PouchDB that allows users to work offline and then sync up when they go back online. We decided to go with MongoDB since we didn't need the availability feature offered by CouchDB and in that case the consistency offered by MongoDB seemed like the better choice. The database was setup on mLab which is a fully managed cloud database that focuses on hosting MongoDB databases and it was a rather simple decision since mLab is quite popular when it comes to hosting MongoDB databases and the queries that are preformed are extremely fast and secure.

4.5 Bcrypt

Bcrypt is a password hashing algorithm designed by Niels Provos and David Mazières, based on the Blowfish cipher, and presented in USENIX in 1999. There are numerous other hashing algorithms such as MD5 or SHA1 but in

our case we decided to go with Bcrypt for a very particular reason - since we are not cyber security experts we need a strong algorithm to protect user data and Bcrypt provides just that because it is **slow**. In essence a hashing algorithm takes a string and changes that string into a random set of letters and numbers, the same word always outputs the same hash and the point of hashing algorithms is that you can very easily go from a string to a hash but it is very difficult to go from a hash back to the original string. In order to explain why Bcrypt is better than MD5 / SHA1 in this project we're going to discuss an example; lets say the user Bob creates an account and our database stores his username, password and email, if the password is simply stored as plain text and the database gets compromised then the hacker can log into the game using Bob's information, in an even worse scenario Bob might be using the same password for his email which allows the hacker access to his email and then possibly Facebook, Twitter or something else, this is precisely why data is hashed. In an alternative scenario lets say the data is hashed using MD5 / SHA1 which are both fast algorithms without the use of salt which is basically a string of characters that gets hashed alongside the password - the database gets compromised and the hacker once again steals Bob's data but since the data is hashed its going to take the hacker a while to decrypt the information by preforming a dictionary attack which is a very long list of passwords and their hashes, if the stored hash matches a hash in the dictionary, the data is once again stolen. On the other hand if MD5 and SHA1 were hashed using a salt then Bob's data would be safer but not out of the hacker's reach i.e. if the salt also gets compromised then the hacker can simply adapt his already existing dictionary list to use the salt meaning the data can once again be stolen. At this point Bcrypt is a better hashing algorithm than MD5 or SHA1 because even if the database gets compromised alongside the salt, it would take the hacker a very long time to update his dictionary because Bcrypt is slow unlike MD5 or SHA1.

4.6 BabylonJS

BabylonJS is a complete JavaScript framework for building 3D games with HTML 5 and WebGL. A lot of browser games can be built simply using JavaScript however there are limitations to this approach and in order to reduce the difficultly of drawing more complex shapes and animations we will be using this library. During the research stage we concluded that we want to develop MageCraft in 3D and after a lot of research and comparing various engines we narrowed the list down to two frameworks - Phaser and BabylonJS. Eventually BabylonJS came out on top for three reasons:

1. There is a lot more documentation provided for BabylonJS when compared to Phaser and that was really important since this is a reasonably big project and the more information we have on how to implement certain features the better.

2. After examining various existing applications that use either or technology, we came across 2 particular games called Shell Shockers and BrowserQuest, both built using the BabylonJS framework. Both of these games represented the sort of look we had intended for MageCraft and that discovery nudged us towards using BabylonJS.

3. Ultimately we concluded that Phaser is more suited to 2D games and having created 2D games in the past we decided to try and challenge ourselves by creating a 3D game.

There is an incredibly long list of the various things that one can do using BabylonJS with basic demos available on their official website [20] and there are nearly as much extensions for the framework that mainly focus on dynamics terrains, modifying materials to represent real life materials such as water or fur, procedural textures, post process libraries, various generators, cloning systems, GUI and a few other ones that aim to make the developer life's a bit easier e.g. debug layer. Instead of going into depth about BabylonJS concepts which is somewhat explained throughout chapter 6, first we're going to take a look at a really basic example of BabylonJS and this is basically what each BabylonJS project should have in order to function. First off in order to create anything a scene must be declared and passed to the engine, a scene is in essence the stage on which all the meshes are placed to be seen. Next up a camera is set up which displays the object in a particular way, there are a number of cameras to choose from however the 3 main ones are: universal camera, arc rotate camera and the follow camera, there are other cameras available for devices and virtual reality. The developer can also determine if the user can interact with the camera. Prior to object visibility at least one light source must be set up and just like the camera there are a number of lights to choose from which determine how colors and shadows appear: point light, directional light, spot light and hemispheric light. At this point we have the 3 fundamental things any BabylonJS project should have in order to display meshes, the following code snippet demonstrates how to set up a basic scene with a sphere mesh being displayed.

```
var createScene = function () {
//Create the scene space
var scene = new BABYLON.Scene(engine);

//Add a camera to the scene
var camera = new BABYLON.ArcRotateCamera
("Camera",Math.PI / 2, Math.PI / 2, 2, BABYLON.Vector3.Zero(), scene);

//Attach control to the camera - allows the user to move the camera
camera.attachControl(canvas, true);

//Add lights to the scene
var light1 = new BABYLON.HemisphericLight
("light1", new BABYLON.Vector3(1, 1, 0), scene);

var light2 = new BABYLON.PointLight
("light2", new BABYLON.Vector3(0, 1, -1), scene);

//At this point we can start creating and manipulating meshes
//e.g. sphere
var sphere = BABYLON.MeshBuilder.CreateSphere("sphere", {}, scene);

//Return the scene
return scene;
}
```

4.7 CannonJS

CannonJS is an open source JavaScript 3D physics engine. Unlike other physics engine libraries ported from C++ to JavaScript, CannonJS was written in JavaScript from the start and can take advantage of its features. When compared to other engines it does lack in features however it is easier to get into and more comprehensible than the alternatives which makes it the perfect physics engine for this project since we are not focusing on complicated concepts but rather implementing some really basic ones. Initially we did not plan to add a physics engine and intended to simply create our own physics throughout the project however towards the end of the project we encoun-

tered an issue where certain meshes would ascend rather than descend which was somewhat problematic and the way we decided to solve this issue was to implement gravity using a physics engine. The library supports the following shapes: sphere, plane, box, cylinder, convex polyhedron, particle and height-field. This collection of shapes matches the collections used by rendering engines such as Three.js and BabylonJS which allows these libraries to work hand in hand with each other. When researching which physics engine to use we came across Oimo.js, Energy.js and Cannon.js. Currently Energy.js is unavailable to the public and Oimo.js tends to handle a few more complicated aspects but since we were only looking for a simple way to implement gravity CannonJS seemed like the perfect fit.

Chapter 5

System Design

When it came to system design one of our primary objectives was to keep the entire code organized by sorting every single file into their respective folders. This approach also applied to the code i.e. there are quite a number of scripts that are all split up instead of having the whole code in one file, we have a few loosely coupled game concepts split into their own files which allows the game to be taken apart piece by piece without essentially breaking it completely. All of the scripts are highly function based and all of these functions are declared within their respective files however certain variables and functions are actually called from within the game.js file so if one were to remove tutorial.js, the code relating to tutorial.js that is initialized and called from within game.js would also need to be removed or commented out, if you would like to disable a particular feature it is much easier to simply leave whichever file alone and just comment out the functions relating to that file from game.js.

5.1 Account System

New players can create an account and are encouraged to do so in order to upload their gold value earned at the end of a match. The idea behind the gold currency is for a future feature that allows players to spend their gold to purchase various cosmetics that alters the appearance of their character. Currently the gold earned value simply represents how long the player survived in tutorial mode. The registration process utilizes Flask-WTF forms with CSRF tokens alongside validators that essentially forces the user to enter input for the 3 required fields - username, password and email. The validators also ensure the username and password have at least 4 characters, up to a maximum of 12. The minimum requirement makes sure that the in-

formation is secure enough and the maximum ensures that a user can't come along and store an extremely long password which would eat up database resources. The validator on the email field ensures that the user actually enters an email, although the email validator is quite basic and simply checks if the 'at' symbol is included. The username and email information is stored in plain text in a MongoDB database set up on mLab, before the password is stored is it hashed using the Bcrypt algorithm in order to protect the data. The login process is rather simple, if the user inputs an existing username and the correct password the user is logged in and a secure session is created in the client's browser. The session stores the username of the logged in user and can be accessed like so:

```
if 'user' in session:
        user = session['user']
        print user
```

In order to keep a user's information safe the session stores a minimum amount of data and it is also in fact very difficult to access from an external machine due to the way Flask sessions are set up i.e. the data is encrypted using base64 encoding based on the secret key that is set up in the runme.py file which should be kept hidden. Once the user is logged in, the user has access to the typical services provided by an account system - change password, reset password, delete account and logout. In order to change the password the user must first enter the current password followed by the new password which must also be between 4-12 characters, the user must also re-enter the new password in the following field in order to change it. After changing the password an email is sent to the registered email address to inform the user the password has been changed. The reset password allows the user to reset their password to a randomly generated string that consists of uppercase letters and digits. In order to reset the password the user must simply enter the username of their account and the server then changes their password to the randomly generated string which is sent to the registered email address and can be used to log the user in. The delete account service simply checks if the user is in session and then deletes the record from the database and pops the session. The logout service simply logs out the user by removing the session.

5.2 Background & BabylonJS setup

The background lava effect was one of the first things we worked on when we got started with BabylonJS. Before we could work on the background we had

to setup a typical BabylonJS project which consists of a scene, camera and a light. The entire game is displayed using just one scene although it is possible to set up multiple. As mentioned earlier the game is set up in such a way that there are a lot of scripts that preform various tasks and the main control for those scripts is located within the game.js file - in order to have that sort of environment the scene had to be made global and the camera as well as the light had to be initiated within the game.js file. Other than that there was a bit more setup much later on in the project when we started using CannonJS for physics, the gravity settings were set up in background.js and the stencil tool was enabled in the Babylon engine which allows mesh highlights. Alas after all the setup we finally moved onto implementing the lava background. In order to set up the lava effect we used a Babylon extension for the lava material, by adding a lava texture to that material alongside modifying some of its properties we created the background. We also added a transparent ground layer since the lava texture moves making it unstable and causes the animations to overlap with other meshes, therefore this transparent layer was placed above the lava and it covers the entire game area - this is the actual layer the player can move on but the player takes damage from standing on the lava, later on we add a player grid which sits on top of this transparent layer and the players can move on this grid without taking damage.

We mentioned earlier that CannonJS was purely added to create gravity since some objects would float upwards instead of falling down, this mainly affected spells but it also fixed a bug where the player would lose control of their character since it would begin to float at a random angle after moving between ground layers i.e. the player grid sits on top of the transparent ground layer which sits on top of the lava, when the player moved off the grid to the ground layer the player would lose control of their character since the grid sits slightly higher than the ground layer and with gravity added it forces the character to stay on the ground at all times which fixes the bug.

5.3 Player Grid

We've established that the grid sits on top of the transparent ground layer furthermore we will explain how it works. While trying to create the grid we expected BabylonJS to have a simple built in function for creating a grid however that was not quite the case. It is possible to split a plane into subsections but those sections cannot be accessed individually and the same applies to the grid material that splits a mesh into a grid. We needed to have a grid where each section can be modified such that we can have various

textures as well as being able to shrink the grid over time by reducing the visibility of a particular section and have the grid be flexible i.e. change it's size easily without breaking the game. Thus we decided to create our own type of grid that is basically a series of planes all sorted into arrays that come together to form a grid. The planes are sorted into a series of smaller arrays over one big one because it makes it easier to modify particular sections of the grid. The way the grid shrinks involves taking the parameter planes and turning their material to the color red indicating that they're about to become inactive after which the grid boundaries decrease and the lava occupies the area where the grid parameter previously existed. This process is repeated over and over again until there are only a few grid planes left, meaning that the longer the game goes on the more difficult it becomes to survive since theres less and less area to move on. There are a series of timers that control when the grid shrinks and the entirety of the grid code is set up in such a way that the grid properties (size, timers, textures etc.) can be changed without breaking the game.

5.4 Player Movement

Before we added proper character models, sprites, animations or any effects we used plain shapes that represented objects like the player which allowed us to jump right in and start working on other features such as player movement. Initially we implemented movement for one player using jQuery i.e. we bound the WASD keys to key press events so whenever the player pressed a key the character moved on the screen. While experimenting with jQuery movement we tried out using SocketIO to make that particular character move for everyone connected via sockets and the way that worked involved sending the coordinates of where the player moved to the server which then broadcast those coordinates to everyone connected and upon receiving those new coordinates the character's location was updated for all players. Having finished with the SocketIO movement we set that code aside and decided to improve on the player's movement because jQuery wasn't handling the movement gracefully, in other words there was a bit of a jitter whenever the player moved thus we got rid of jQuery completely replacing it with DOM key down event handlers which made the movement smoother. The idea behind using those events over jQuery was essentially the same, whenever the player presses one of the WASD keys the character moves in a particular direction and stops moving once the key is released.

5.5 Player Spells & Spell System

Just like player movement, we used regular plain shapes for the spells before any animations and the like were added. Currently there are 6 offensive spells and 2 defensive spells. Each of these spells are bound to a key (1-6 offensive, G and H for defensives), unlike player movement which is bound using DOM, these keys are bound using an action manager provided by BabylonJS which allows the registration of certain events e.g. key presses. Whenever the player presses one of those spell keys the action manager identifies which key was pressed and selects the appropriate spell, we refer to this system as the spell system. So in essence whenever the player presses a spell keybind the spell system makes sure that all the other spells are not selected and only that one particular spell the player selected is available to cast, in order to cast the spell the player must then aim using the mouse followed by either right clicking or left clicking to fire that spell towards the mouse direction. Once the spell is cast, a mesh representing the spell spawns at the player's location and begins to move a certain distance towards the clicked location. Some of the spells have different ranges, speed as well as having different animations depending on the location the spell is moving towards. Another part of the spell system is the spell movement animations. Each spell has a registerBeforeRender loop that handles translating the mesh towards the clicked location every frame, these loops execute similarly to the engine render loop but just like the name suggests they execute before the engine render loop. RegisterBeforeRender loops are more commonly used in BabylonJS projects that utilize more than one scene, in our case we could have stuck the spell movement code inside the engine render loop but in order to keep the code in their proper respective files we used the RegisterBeforeRender loops which kept the code tidy instead of splitting it by placing the movement logic directly into the engine render loop inside the game.js file. The following table summarizes what each spell does, how much damage they do and the cooldown duration before they can be recast.

Spell Summary		
Spell	Value	Cooldown(seconds)
Fireball: a classic mage spell that simply does damage upon collision.	5 Damage	5
Frostbolt: inflicts damage and freezes the enemy in place for 6 seconds.	5 Damage	30
Splitter: splits into 8 projectiles that all inflict damage.	3 Damage each	25
Recharger: inflicts damage and it's cooldown is reset on hit.	5 Damage	15
Molten Boulder: inflicts damage to all enemies caught in it's path.	10 Damage	20
Warlock's Sigil: spawns a zone that temporarily slows down enemies.	0 Damage	40
Deflection Shield: surrounds the caster in a bubble that blocks all damage.	Lasts 6 seconds.	40
Cauterize: heals the caster for a certain amount.	10 or 20 Health.	40

5.6 Animations

Instead of having plain meshes with still images, we tried to make the game as interactive as possible by adding various sprites, sprite animations, particle systems, highlights, various materials, Babylon extensions for materials, various textures, images etc. This was a rather big and time consuming section of the project, it was also an important one since a lot of people judge games based on the visuals. It is not uncommon to create models using Blender and then import those models into Babylon however due to the complexity and how long it takes to create models using Blender, we decided to stick with open source sprites. The player, AI and some of the spells use sprite managers which essentially acts like an object that can be manipulated to create sprite animations. The sprite manager is quite efficient however it does have one primary flaw - the sprite manager uses cell size that represents a square meaning that the width and height of the sprite should be equal otherwise parts of the sprite will be excluded. This posed quite a number of problems since a lot of the sprite sheets out there have different widths and heights, other sprite sheets can have an unequal distance between frames which prevents us from creating proper animations since objects in those frames would appear/disappear or overlap - this is because the sprite manager takes the

cell block then in the next frame registers and next cell block which is perfect as long as the distance between those frames are the same. Another problem was finding the correct cell size of a sprite sheet in the first place since they aren't specified therefore its a guessing game to find the correct sprite size which should be pixel perfect in order to avoid any abnormalities during animations. We determined that a lot of the open source sprite sheets may look commendable however most of them don't fit into sprite manager without some form of editing. To make the editing process easier, we used Photoshop and Paint.net to mostly correct the distance between cells such that they are equal and in situations where the width and height were different we removed certain cell blocks that overlapped with the ones we wanted to use for animations. Instead of using sprites for every single spell, some spells used plane shapes (mostly sphere) that had a twist on their materials/-textures. We used a couple of Babylon material extensions - fire, water and lava. Lava worked flawlessly as demonstrated by the background however fire and water have their complications. The fire library highly depends on using lighting and shadow maps, in our case we had lighting already set up, even so we proceeded to add additional lighting and enabled shadow maps with the end result being an incredibly dark game environment which did not seem appealing so we reverted the lighting to the way it was and used the fire procedural texture library instead. The water material has very impressive visuals however after testing it out and doing additional research, we discovered that it wasn't updated for the newer Babylon version and switching to an older version would cause unwanted bugs so as a result we continued to use the fire procedural texture instead while tweaking its properties to achieve a different visual effect.

Highlights were used for the recharger and splitter spells, the highlights basically provide an additional glow to the spells. Recharger has the glow on at all times whereas splitter only lights up after it travels a particular distance and when it stops and begins to spin it begins to glow while spawning projectiles, this effect is something quite little but the devil is in the details and these small technicalities can eventually add up and be very appealing. A couple of particle systems were added for the fireball spell, molten boulder spell and warlock's mark/sigil. These particle systems are commonly used to simulate hard-to-reproduce phenomena like fire, smoke, water or abstract visual effects like magic glitter and faery dust. Theres a built in system in Babylon for creating particle systems, the way they work involves taking a particle texture and emitting a series of cloned particles from an emitter which in Babylon is limited to box, sphere and cone. This particle system is designed to be very efficient, the number of particles that spawn from

the emitter shouldn't cause any frame problems even on low-end devices. When using the particle system the developer has a lot of control over how it functions i.e. you can easily change the number of particles that spawn, where they spawn from, their spread, lifetime, size, colors, direction, gravity, rotation, speed etc. For the fireball spell we designed 2 particle systems, one for smoke and the other for fire, they both originate from the fireball mesh which causes the fireball to leave a trail of smoky fire as it travels. The molten boulder spell uses a similar system to the fireball originating from the molten boulder itself however the trail that is left behind is more dangerous looking since it deals more damage than the fireball. The warlock's sigil uses 2 particles system that are alike, these particle systems emit from transparent boxes on each side of the sigil. The direction of the particles is towards the sigil which makes it seem like the sigil is absorbing the particles which is a proper representation of the spell as it slows down movement of all enemies caught within it's vicinity.

5.7 Collision Detection

Collision detection using BabylonJS is very simple and can be achieved with just a few lines of code. The fact that collision detection is simplified is a great change of pace, having worked on 2D collisions in the past we know that there is a lot of math calculations in determining the point of intersection which of course depends on which shapes are being tested for collision and depending on those shapes different formulas have to be used. In this project each mesh that needs to be checked for collision detection has a hitbox that overlaps and follows that mesh anywhere it goes. In essence the hitbox is just a transparent sphere. Currently collision detection is preformed between the player and the AI as well as collision detection between all the spells and the AI. When an enemy collides with the player, the player takes damage. When a spell collides with an enemy, the enemy takes damage. This collision detection is preformed on the hitboxes and not the actual player, enemies or spells. The following code snippet demonstrates how easy it is to check collision detection between 2 spheres which could represent anything e.g. enemy and player.

```
if (sphere1.intersectsMesh(sphere2, false)) {
console.log("Collision detected!")
} else {
console.log("There is no collision.")
}
```

5.8 User Interface

The goal of the UI is to display all the various spells the player can cast as well as their keybinds and cooldowns i.e. how long before the spell can be cast again. The UI also displays player health points, status effects and gold earned. Unlike the menus the UI is created within the scene using Babylon and the GUI extension provided by Babylon. The GUI extension offers a variety of features that helps build UIs. Firstly we created a simple plane that faces towards the player, followed by adding smaller planes with various textures that represent the spells, each spell plane has a border which indicates if a spell can be cast - if the border is green the player can cast the spell, if it is red then the spell is on cooldown. Next we added a couple of sprites for the player health and gold. Lastly we added status text and text for gold value. The status text informs the player if something important happened such as getting hit by an enemy whereas the gold text simply displays the amount of gold earned in the current game. In order to write 2D text in the scene, we first have to create a mesh then create an advanced dynamic texture which takes in the created mesh as a parameter - the advanced dynamic texture basically allows you to write on the passed in mesh using the HTML5 canvas. An important aspect of a lot of video game UIs is that they're glued to a particular location on the screen and don't move from that location. Since our UI is created within the scene using a number of planes, the UI needs to move with the player otherwise when the player moves the UI gets left behind in the position it was created. So in essence the UI moves in the direction the player is moving but it always remains in the same location on the screen i.e. the bottom center.

5.9 Menus

In this section we're going to briefly discuss the pause and game over menus. Both of these menus are designed in the same manner using HTML and CSS as well as a bit of JavaScript. In essence both of the menus are simple HTML tables that are hidden until the player either pauses the game or dies(in game). The pause menu allows the player to resume, restart, mute music or return to the main menu whereas the game over menu allows the player to restart, upload gold earned online, return to main menu and it displays the gold earned. All of these menu options are buttons that have an on click event which executes a bit of JavaScript to achieve the deserved result. Behind the scenes, whenever either of these menus are displayed the game should be paused along with making it impossible for the player to

interact with the game i.e. if the player dies we don't want him to continue earning gold or able to fire spells at enemies. This can easily be achieved by breaking the game render loop (engine.stopRenderLoop()), the good thing about this function is that we can later resume the game by starting up the engine loop again exactly where we left off because the game data is not automatically reset due to stopping the engine.

5.10 Chat System

Currently the chat system allows everyone who is connected to the game on the cloud to communicate. It is possible to create separate chats where only the players who are connected to a particular game can communicate but since multi-player was scarped for the time being, it is better to have the "global" chat that allows everyone to communicate. The chat system isn't really complicated and uses SocketIO to broadcast messages. When a player launches the game a socket connection is established, to send a message the player simply enters input into the input field beside the chatbox and clicks on send. Afterwards the server receives the message and broadcasts it. All the clients that receive the message, take that message and append it to the chatbox list. Other than that, there are a couple of commands the player can enter into the chat which change the AI movement patterns in tutorial mode. The purpose of these commands is to increase the difficultly of the tutorial if the player is finding it far too easy and wants more of a challenge. There are four commands in total and the player can decide how much more difficult to make the game, the player may want to instantly go from easy mode to hard mode by using all 4 commands or 1 at a time to ease into the game.

5.11 Tutorial Mode & Enemy AI

Tutorial mode was initially intended to be a bonus feature if we finished everything else however it was completed instead of multi-player because it seemed achievable within the time frame that was remaining. The idea behind this mode is having an AI to play against. The AI isn't specifically designed to be difficult to beat or be actively trying to murder the player, it is simply there for the player to learn how to play the game and to demonstrate various game concepts that can also be applied in multi-player mode such as collision detection, animations, player movement etc. The enemies are created using a function that creates a mesh, the proper material is assigned

to the mesh, a hitbox is generated, a sprite manager is created and the spawning location is assigned. The function is also responsible for movement patterns, a lot of collision detection and sprite animations. In essence this one particular function is quite compact and it has all the code necessary to be initialized within the game.js file in order to spawn a lot of enemies and then call each enemy method within the game render loop. The enemies are spawned at the left, right, top and bottom edges of the game world at random. The standard movement pattern basically causes the enemies to move towards the player from the direction opposite of which the player is moving towards i.e. if the player moves left, enemies from the right move towards the player, this way no matter which way the player moves theres always going to be enemies coming towards the player. These movement patterns can be changed using chat commands as mentioned in the chat system section. The commands force the enemies on a particular side to move not towards the player but across the grid e.g. the command !Right forces enemies on the right to move left across the grid. If all 4 commands are triggered then the player has enemies coming from every side which get more and more difficult to avoid since the grid shrinks over time. On a side note, these enemies have endless respawn, so if the player kills an enemy it is spawned after a short duration at its original starting location and once again begins to move towards the player. In order to prevent enemies from going off the map we set up boundaries on the edges of the game world so if an enemy goes past that boundary it is immediately reset to its original starting location. One last point to mention, even though the enemies spawn at random they collectively move towards the player in that particular formation they spawned in, this is only until the player starts killing them off after which the formation is broken and the enemies start moving towards the player in different formations which adds a level of unpredictably to the enemy movement.

Chapter 6

System Evaluation

In terms of evaluation, the main objective that was kept in mind through the entire project was to keep the game running at 60 frames per second (FPS). This limit was set by BabylonJS since the engine can only run at a maximum of 60 FPS. Ultimately this objective was met... until it was tested on a lower end device. When it comes to using BabylonJS, hardware plays a big role especially in this project so if the game is played below 60 FPS it feels very clunky, it all comes down to how fast the device can handle the engine render loop. Though the upside is if your machine can run a really basic BabylonJS project at a high frame count then it shouldn't have much difficultly running this project within a reasonable frame range because the majority of the systems that BabylonJS has are very efficient. The FPS counter that was implemented was also a useful means of testing if the code is working efficiently. To clarify, if the game is running at a steady 60 FPS until a new feature is added after which the FPS drops drastically, well then thats a clear indication that the newly added feature is inefficient and needs to be revised.

Other than keeping an eye on the FPS count to ensure the system is robust, whenever an issue arose while using BabylonJS and debugging using the debug layer or reading the official docs did not provide a solution / workaround, the majority of the issues were solved after researching the issue, and in most cases the solution to the problem led to the BabylonJS forum where users faced similar issues. The colossal benefit of using the forum is that it is very active with tens of thousands posts and not only do experts versed in the art of Babylon respond but so do the actual developers which point the posters into the right direction as well as showing the proper way of going about a certain issue. The layout of the forum is comparable to Stack Overflow where the posters are encouraged to post sample code with the issue, and

using the Babylon playground it is possible to quickly paste the issue into the playground which displays the visual outcome to the right of the code. Typically when an answer is given back to the problem/question the person also provides a Babylon playground sample with the answer coded up which makes it an excellent place to learn how certain things are done and certain issues are solved.

6.1 Scene Optimizer

BabylonJS provides a tool that is designed specifically to help projects reach a specific frame rate by gracefully degrading the rendering quality at runtime. In the event a low end device loads up a BabylonJS project this tool can help make the scene viewable / usable up to a point. Unfortunately we did not have the time to implement this tool for this project. In essence the optimizer tool has a few settings that can be tweaked to make the project more usable on a low end device e.g. we can specify the target frame rate which is 60 by default or we can set up a priority list in which we can decide in which order certain features are loaded. Additionally there is a static helper that simplifies the optimization process. There is also a more advanced usage to the optimizer that can control which Babylon settings are enabled or disabled e.g. we can disable post processing, shadows, particles etc. Overall this seems like an effective tool if the project uses a lot of resources and the engine render loop preforms a lot of heavy operations however it still won't quite make the project runnable on very low end devices if the hardware is already under a huge amount of work from simply opening a browser.

6.2 Firefox Version 58

When we initially got started with Babylon we were testing code on Firefox. Most of the time the code worked as intended however as time progressed and we started testing various materials, we started encountering more and more abnormalities and we straight away assumed that there was something wrong on our end with the code. After a while of testing various approaches to get certain parts of code working we started doing a lot more research as to why these abnormalities were occurring and it took a while before we arrived at a forum post that said Firefox version 58 has broken quite a number of existing libraries causing them to malfunction and it was then we learned the hard way that BabylonJS was one of those affected libraries. Surely, once we switched to Google Chrome the abnormalities were gone and the code

functioned properly. However being stubborn we still attempted to create the same game experience for Firefox users. One of the main bugs that occurred was the way Firefox handled materials i.e. the first change (e.g. color/alpha) applied to one material is also applied to any and every other material in the game. The code that demonstrates this bug is available in the bugs folder in the GitHub repository. Another problem with Firefox in general not just version 58 is that it is significantly slower to load resources and has lower frame rate when compared to other browsers and this isn't just the case for BabylonJS but also applies to other WebGL libraries such as Three.js. To support this we researched and preformed a benchmark comparison [21] on the popular browsers and to summarize the following table representing overall test results:

Chapter 7

Conclusion

To summarize: The end goal of this overall project was to create our own variation of MageCraft. All of the objectives set up in chapter 2 were completed bar multi-player mode.

This feature had to be abandoned about half way through the project as a result of the project scope being underestimated, it was replaced by tutorial mode. We started off on the right foot by trying to figure out how we're going to implement multi-player which led to testing player-movement using SocketIO as described earlier in chapter 5 which worked out dandy, then we transitioned to actually creating the game and had to leave the multi-player part behind since it makes sense to create the game for single player, make sure that everything works then afterwards add multi-player capabilities. The mistake that we made was writing a lot of code that is specifically designed for one player and as a result we had a lot of features that would have to be broken apart and rebuilt with multi-player in mind, all of that stacked up and we came to the conclusion that it would be impossible to get it all working for multi-player before the deadline. What we should have done is - work on a feature for single player, ensure everything works as it should then take that feature and design it for multi-player instead of letting it stack up, ideally this approach would have resulted in finishing the project with tutorial mode complete for single player and have a complete multi-player mode. Although, there is also a possibility that neither the tutorial or multi-player would be finished on time due to the amount of work it takes to implement those features, in which case an additional team member could be very useful. Another issue would be determining the correct team size for a project such as this and it is quite difficult to measure the scope of certain projects though it probably becomes easier as time goes on and more experience is accumulated from working on big scale projects.

All the various technologies that were chosen for this project were well picked that all integrated together without major issues, this was due to the immense amount of research carried out before coding started. Although there were a few minor hiccups such as the Firefox issue which could have been avoided if we weren't as quick to jump to conclusions and did even more research beforehand though no matter how much research is done there will always be some sort of bump in the road. If we had the option of going back in time and switching out a particular technology for a different one - we would leave it just the way it is because this project provided a great learning experience on how to use quite a few technologies that can be beneficial in the future for other projects.

Following up on the system evaluation, we would have liked to incorporate the scene optimizer in order to see just how well it works if we had a bit more time. Aside from using the optimizer tool, it could have been possible to increase performance of the overall game by reducing the number of materials used. This doesn't specifically mean reduce the quality of the game by removing a bunch of materials - it means materials that preform similar enough roles or duplicates should be re-used more often on the meshes that need them. We could perhaps take all the existing materials and put them into a separate file which would make it easier to identify which materials have similar properties, get rid of certain materials that are similar and simply apply the remaining materials to meshes. Ultimately this would cut down on the amount of objects that are created in memory and make the game run faster if the engine is not already capped at 60 FPS.

Ultimately we reckon that this project helped us take a giant step forward and the main contributing factor to that was learning how to use new technologies, to be more specific BabylonJS. This library helped us transition from creating games in 2D to 3D and since this was our first 3D game we can say the overall project was a success. To clarify, we are not discrediting all the other technologies that were used, simply stating that BabylonJS taught us the most and not only will that technology be useful in the future but so will the knowledge gained from using it.

Bibliography

[1] M. H. David Kim, Dustin Browder *et al.*, "Starcraft 2." Publisher: Blizzard Entertainment, Official Website: `https://tinyurl.com/ybupwaym`, Jul 2013.

[2] M. Morhaime, "Blizzard Entertainment." Official Website: `https://tinyurl.com/y74g8ely`, Feb 1991.

[3] Blizzard Entertainment, "StarCraft 2 Engine Development History." Source: `https://tinyurl.com/pohp8m5`, Oct 2011.

[4] Blizzard Entertainment, "Arcade." Source: `https://tinyurl.com/clae3v3`, May 2012.

[5] Aphotic, "MageCraft Release." Source: `https://tinyurl.com/ybw2cy36`, Mar 2011.

[6] "Team Syntax." Source: `https://tinyurl.com/y869lf44`.

[7] "Forum Post." Source: `https://tinyurl.com/yc7nxj6x`, Mar 2014.

[8] T. G. Steven Snow *et al.*, "League of Legends." Publisher: Riot Games, Official Website: `https://tinyurl.com/y9u9qrld`, Oct 2009.

[9] E. J. Jason Hayes, Tim Larkin *et al.*, "DOTA 2." Publisher: Valve Corporation, Official Website: `https://tinyurl.com/yd4r7mtp`, Jul 2013.

[10] "Heroes of Newerth." Official Website: `https://tinyurl.com/y8cv8jc9`, May 2010.

[11] "Smite." Publisher: Hi-Rez Studios, Official Website: `https://tinyurl.com/gnxdcov`, Mar 2014.

[12] R. Graudins, "MageCraft GitHub Repository." Source: `https://tinyurl.com/y997fy5l`, Oct 2017.

[13] R. Picard, "Flask - Organizing your project." Source: `https://tinyurl.com/yde6a25h`, 2014.

[14] "Agile Methodology Diagram." Source: `https://tinyurl.com/y8h653ws`, Nov 2014.

[15] "BabylonJS Debug Layer." Source: `https://tinyurl.com/yayq5dyb`.

[16] "BabylonJS Forum." Source: `https://tinyurl.com/no2sb6x`.

[17] A. Ronacher, "Flask." Publisher: Pocoo, Source: `https://tinyurl.com/y5vd3tt`, Apr 2010.

[18] M. Yaghi, "Django Tornado Setup." Source: `https://tinyurl.com/y7fqbp2d`, Sept 2012.

[19] John C. Klensin, "Simple mail transfer protocol," *RFC 5321*, Oct. 2008.

[20] V. V. David Catuhe *et al.*, "BabylonJS." Source: `https://tinyurl.com/yckc7ga2`, Aug 2013.

[21] HTML5 Test, "Browser Benchmark." Source: `https://tinyurl.com/y8tbzaqt`, Apr 2018.