Carleton University

Project ID-24

SYSC 4907

Final Report

Battle University

2D Massive Multiplayer Online Role-Playing Game

Supervisor: Professor Roy Franks

Group Members:

Ryan Boucher: 101049347

Nnamdi Okwechime: 100993125

Manel Oudjida: 100945382

Kyle Smith: 101045797

Declaration

We hereby declare that this document and all referenced materials is our own work and to the best of our knowledge contains no previously published materials or facts written by another person. All material used for research, inspiration, or reused has been acknowledged and properly cited.

Ryan Boucher,

Nnamdi Okwechime,

Manel Oudjida,

Kyle Smith

Acknowledgements

We would like to greatly thank our supervisor, Dr. R. Gregrory Franks for his supervision and support in bringing our project into reality. Moreover, his input and guidance allowed for a smooth development and implementation of the game. We would also like to thank the various games we pulled inspiration from, namely; Maplestory, World of Warcraft, Terraria and Celeste

Project Abstract

The purpose of this project is to develop a multiplayer video game that tells a story. It is a 2D, multiplayer, role-playing game using the Microsoft XNA framework in C#. The game was created with an object-oriented design style, and best practices were followed for code and documentation creation. Additionally, the project used an agile development process to provide a flexible development schedule, as well as some aspects of TDD (test-driven-development) to allow for a faster start to the coding process. Moreover, the game uses semantic versioning and GIT for version control. The game follows the Massive Multiplayer Online Role Playing Game stereotype (MMORPG), pulling inspiration from games like World of Warcraft, Maplestory and similar 2D games like Terraria. Following the MMORPG stereotype, the game is expected to have a large world where individuals can interact, team up and battle monsters together.The game tells the story of a student attending a university for battling and draws parallels to the real life university experience. The story is primarily told through talking to Non-Player-Characters (NPC's) and completing quests. These quests will ask the players to collect items, kill monsters, or clear areas in return for rewards in the form of currency, equipment and story development. The game will also contain some platforming aspects to add further challenge and depth to the game. The main objective of the game is to develop your character's attributes and skills through leveling up. The two main subsystems discussed in this document are the server and the client. Each is discussed in detail in this document, outlining their functionality, design, testing, and implementation.

Table of Contents

List of Figures

List of Tables

Glossary

NPC - Non-Player-Character

2D - 2-Dimensional

MMORPG - Massive-Multiplayer-Online-Role-Playing-Game

TDD - Test Driven Development/

OOP/OOD - Object-Oriented-Programming / Development

AI - Artificial intelligence

TCP - Transmission Control Protocol

UDP - User Datagram Protocol

GUI - Graphical User Interface

1.0 Introduction

1.1 Problem Background

The video game market is extremely lucrative, grossing 36.87 billion dollars in the US in 2019 [2]. The objective is to break into this lucrative market and produce a game that could be used to launch a company in this industry. This project seeks to provide a light-hearted, fun, experience in a video game setting while also addressing meaningful trials and topics of a real university experience. Similar games exist that provide a deep role-playing experience, but very few exist that depict real-life mental health struggles and challenges associated with school. The nature of an MMORPG (Massive-Multiplayer-Online-Role-Playing Game) allows for a unique story-telling experience that is both engaging and exciting. The game is intended to be used for recreation while still giving more serious lessons through the story. The story for Battle University enables a wide range of functionality by moving away from realism and into the fantasy domain. The functionality gained will allow for a more exciting user experience.

1.2 Problem Motivation

As software engineering students, we are required to work on a capstone project. All students were given a list of possible projects they could join while some supervisors allowed for self proposed projects. During the selection phase a group member, Kyle Smith, came up with an idea to develop a 2D video game. After proposing the project to supervisor Roy Franks, more research was done until the project settled on using the C# XNA library to create a 2D MMORPG. We had a couple reasons for that. Firstly, we had to decide what kind of game to

make. Some of our original concepts of a survival game, or co-operative base builder were deemed to be too difficult to design. One of the most popular games, World of Warcraft, generated nearly $10 billion in revenue in the first 10 years after its release[3]. This lead to our decision to make an MMORPG driven by a storyline, just like World of Warcraft. Next, we had to decide what kind of story to tell, as this would be the driving force behind the game. We created a couple concepts before finally deciding on Battle University. This would allow us to follow archetypical MMORPG progression with the flexibility for deeper concepts pertaining to the struggles of school. Finally, we had to decide on a development environment. A few different options came up, but after research during the design process, we decided on C#'s XNA library. This library choice was inspired by the popular video game Terraria, which uses the same engine. Overall, there are thousands of games that have attained relative success off of even simpler concepts and we feel this game could possibly become a successful indie game once it gains enough attention.

1.3 Problem Statement

The problem being addressed in this project is creating a game that can break into the lucrative market that is the video game industry. To do this, the game needs to have its own atmosphere, feel, and unique story that keeps players engaged and coming back. Additionally, the game needs to gain popularity to attract more players.

1.4 Proposed Solution

The concept of Battle University, an MMORPG with platforming concepts is our solution. By following the lead of other popular games Battle University can be designed to be attractive in

similar ways as World of Warcraft and Terraria. Providing a unique story about a university of battling that draws parallels to real life hardships. The combat and movement are designed to feel satisfying to the user. Additionally, the game promotes team play and most content is geared towards being completed in groups. This will encourage users to get their friends to play, increasing popularity. Moreover, the game is intended to be free to play so as not to restrict the people who will play it just to try it. Possible revenue generating material like exclusive in game cosmetic items can be added when the game gains popularity.

As a real-time video game, network latency needs to be minimized as much as possible and server tasks optimized. To address this the project is planned to use a distributed architecture where heavy computation is done on the client to speed up the server.

1.5 Report Overview

This report addresses engineering concerns such as health and safety, contributions of team members and project management in section 2. The requirements of our proposed solution is outlined in section 3 where functional and nonfunctional requirements are discussed. Important design decisions and models driven by the requirements are in section 4. The implementation of this design is explained in section 5 and testing in section 6. Lastly section 7 concludes the report with a summary of the entire project.

2.0 Engineering Project

2.1 Health and Safety

General Health and Safety Principles [1]

- Food and beverages are not permitted in the lab. Consume food and beverages only in properly designated areas. (Ontario Regulation 851 Section 131)
- Use appropriate personal protective equipment at all times. (OHSA Section 28(1))
- Use laboratory equipment for its designed purpose.
- Confine long hair and loose clothing. (Ontario Regulation 851 Section 83)
- Use a proper pipeting device. Absolutely no pipeting by mouth.
- Avoid exposure to gases, vapours, aerosols and particulates by using a properly functioning laboratory fumehood.
- Wash hands upon completion of laboratory procedures and remove all protective equipment including gloves and lab coats.
- Ensure that the laboratory supervisor is informed of any unsafe condition. (OHSA Section 28 (1)(d))
- Know the location and correct use of all available safety equipment.
- Determine potential hazards and appropriate safety precautions before beginning new operations and confirm that existing safety equipment is sufficient for this new procedure. (See Appendix 3, Laboratory Risk Assessment)
- Avoid disturbing or distracting other workers while they are performing laboratory tasks.
- Ensure visitors to the laboratory are equipped with appropriate safety equipment.

- Be certain all hazardous agents are stored correctly and labelled correctly according to Workplace Hazardous Materials Information Systems (WHMIS) requirements. (Ontario Regulation 860)
- Consult the material safety data sheet prior to using an unfamiliar chemical and follow the proper procedures when handling or manipulating all hazardous agents.
- Follow proper waste disposal procedures. (See Appendix 4, Disposal of Hazardous Waste)

2.2 Engineering Professionalism

Through the research, design, development and implementation of the project, we applied thorough engineering ethics and professional principles. If any work from a third party source was used, it has been cited and has not been in any way copied or stolen from a third party source.

2.3 Project Management

At the beginning of the project, Kyle Smith volunteered to act as project lead and drive the project. Together as a group, we created an overview of the work that needed to be done and broke it down into iterative Milestones. Furthermore, these milestones were broken down into individual tickets and distributed to project members in an agile fashion. Milestones were assigned hard deadlines that created a development life cycle making it easier to complete the project on time. After a couple milestones, we set the scope of our project defining what our fully implemented, partially implemented and planned features would be at the conclusion of this project.

2.4 Contributions

2.4.1 Project Contributions

Kyle Smith

- Entities
- Movement and Collision
- Server
- Network Protocol

Ryan Boucher

- Testing
- Network Protocol
- Database

Manel Oudjida

- Art
- Menus
- Security

Nnamdi Okwechime

- Story
- Quests
- Items

2.4.2 Report Contributions

Kyle Smith

- Contributions to Sections 1, 3, 4, 5, 7

Ryan Boucher

- Contributions to Sections 1, 6, 7

Manel Oudjida

- Contributions to Sections 2, 3, 4, 7

Nnamdi Okwechime

- Contributions to Sections 3, 4, 5

3.0 Functionality

3.1 Functional Requirements

The functional requirements are driven by how the game is to be played. The character will start as a classless vagrant. Upon reaching level 15 they will be given the choice to become either a Mercenary, Ranger, or Mage. These classes further split the skills and attributes they will use. Each class will bring different mechanics to the game but share common functions of their base class. Mercenaries will be sword wielders, choosing either to be a sneaky rogue-like, reckless barbarian type, or armored knight. The Rangers will be bow or gun wielders, either a short-bow with rapid fire, a long bow with slower but more powerful attacks, or a pirate gunslinger. The Mages will be the spell wielding class, choosing between fire and earth based attacks, water and air based attacks, or choosing to be a light and darkness mage providing healing as well as minimal damage to a party. The world is split into small segments or zones where there are different NPC's or enemies. The server's job is to validate all input and interactions of the player and relay player information to other clients. Additionally, the server shall handle all NPC actions and syncing, thus acting like a client for every NPC.

3.1.1 Entities

Entities are the actors within the game world. This encompasses anything that needs to be physically displayed in the world that holds some state or value. All entities have an image, size, and position.

Figure 3-1: Hooded Man (NPC) Entity

3.1.1.1 Players Entity

The player will have functionality other than just appearing in the world, but in this section we are only concerned with the physical entity. The player will have collision detection on tiles (section 3.2.1.1) for platforming and on enemies (section 3.1.1.2) for receiving damage. The player entity will be able to move around the world and will have animations for each of its movement states. The player entity will move with user inputs, and will update the server with location. Player entities will make use of weapon entities (section 3.1.1.4) to attack other enemies.

3.1.1.2 Enemy Entity

Enemy Entities are NPC's that contain all the same requirements as the player entity with the exception of input based movement. The server will compute the movement of non player entities like this and friendly entities (section 3.1.1.3). Enemy Entities may or may not make use of weapon entities (section 3.1.1.4) to attack without collision.

3.1.1.3 Friendly Entity

Friendly Entities are NPC's that may or may not move. All friendly entities have the unique option to engage in dialog (section 3.1.5.1). The server will compute any movement of this entity. Friendly enemies will not engage in combat and will not need collision detection for damage.

3.1.1.4 Weapon Entity

The weapon entity is a unique entity that will be utilized by players and enemies to attack. The weapon entity will have a hitbox represented with a rectangle. The weapon entity will have two main modes of operation. It will either be held in place, for a period of time or move along a path. Additionally, weapon entities will have options for targeting (single target mode, or multitarget mode). Spells will be represented with Weapon Entities. Weapon entities can apply status effects to a target it hits (section 3.1.3.3). Weapon entities will be able to both deal or heal damage. Weapon Entities will be able to cycle through sprites, rotate, change color or size, in order to create an animation.

3.1.2 Platforming

Platforming is a game mechanic where the player must jump from platform to platform to move around. This encompases a few other requirements that come together to make platforming work.

3.1.2.1 Tiles

Tiles will be used to create platforms out of individual pieces. Each tile will need to be observed for collision detection (section 3.2.1.3).

3.1.2.2 Tile Maps

Tiles will be combined into a grid-like structure defined by tile maps. Tile maps will be loaded from local files. Tile maps will depict the location of each tile within a specific mapping. Tile maps. In game zones will consist of two layers of tile maps, a collision map, and a background map. Background tiles will be tinted grey (to appear as though they are farther back) and have no collision. The collision map will be displayed in a scaled down form to provide a minimap to players.

3.1.2.3 Collision Detection

Collision Detection determines if two rectangles are touching. Collision detection is not exclusive to the tiles and entities, but can also be for entities on entities (more commonly called hit detection for colliding entities). If an entity is touching a tile it will force 0 momentum in the opposite direction of the face the entity is touching. If a player entity is touching an enemy entity the player will take damage.

3.1.3 Combat System

The combat system is a complex, attribute based system. The main idea of the combat system is that entities have an attribute for health, and upon reaching 0 health they will die. For enemy entities this means dropping loot and giving the player experience or quest progress. For players this means respawning at a safe zone. The systems main interactions comes from spells, weapons, player entities and enemy entities. There are a number of other attributes that contribute to the affect spells and weapons will have, or when they can be used

3.1.3.1 Entity Stats

Each entity will have these attributes called stats. These attributes will be affected by equipment, player level, or innately given (like with enemies). When a player levels up they will be awarded with stat points that they can use to increase their stats.

3.1.3.1.1 Health

Health will be the number of hit points an entity has. This attribute will have a max value, which will be the amount the entity can have. This means the entity will also have a current value, which represents the amount of health the entity currently has. An entity that reaches 0 health will die. For players this will be respawning in the nearest safe zone. For enemies this will be dropping loot and giving experience to the player.

3.1.3.1.2 Mana

Mana will be the number of magic points an entity has. Mana will be consumed to cast spells. Like health, mana will have a max and current value. Mana will be restored over time, or with unique spells that restore mana.

3.1.3.1.3 Strength

Strength will be a damage modifier for physical attacks. Spells or attacks that use the players weapon will use the strength modifier to determine the damage dealt by the attack.

3.1.3.1.4 Intelligence

Intelligence will be a damage modifier for magical attacks. Spells or attacks that use magic will use the intelligence modifier to determine the damage dealt by the attack. Some spells may use both Intelligence and Strength to determine damage.

3.1.3.1.5 Armor

This is a numerical value representing the protection from your equipment. When an entity takes damage the damage will be multiplied by the formula $100/(100 + Armor)$ where Armor is the value of the entity receiving damage.

3.1.3.1.6 Move Speed

This is how fast your character moves. Move speed is a direct translation into horizontal velocity. Move speed can be increased by some equipment and spells or through leveling. There will be a maximum move speed that no entity will be allowed to exceed (This is to avoid a velocity high enough to go over entire tiles without collision)

3.1.3.1.7 Experience

This attribute is the amount of experience the player has. Each level will have an amount of experience required to level up that the player must obtain. Experience is rewarded from quests and killing enemies. Experience required per level is logarithmic, allowing fast leveling at the beginning but slowing near the end.

3.1.3.2 Spells and Weapons

The game will have a large amount of spells and weapons that will be used to attack or provide status effects (section 3.1.3.3). See Weapon Entity (section 3.1.1.4) for how these function. Spells and attacks will have cooldowns, which is a duration before the given spell or attack can be done again. These cooldowns provide a balance so stronger spells can be limited to being used less frequently, while basic spells can be used more frequently. Most spells are learned through leveling up, which gives the user skill points. These skill points can be spent to learn a skill in the selected skill tree that can then be used.

3.1.3.3 Status Effects

A status effect (also commonly referred to as buffs and debuffs), is a negative or positive effect that can be applied by an attack or spell. Status Effects are largely variant and can mainly be described as a temporary boost or a decrease in some Entity Stat (section 3.1.3.1). All status effects have a timer for when they will run out, and during their duration will apply their effect to the entity. For example, a character might use the spell Quickstep, which gives the player +5 movement speed for 30 seconds.

3.1.4 Item System

There will be a multitude of items within the game. Each player will have an inventory where they can hold items on their person, as well as a bank where they can store items. The players inventory will also have equipment slots where they can equip certain items. Additionally, some items may be able to stack into 1 inventory slot. For example, you might have 10 potions, rather than take up 10 inventory spots, it takes 1 and can stack up to 100 potions in 1 slot. Items can be divided into 4 main types; Quest Items, Equipment, Consumables, and Crafting Material.

3.1.4.1 Quest Items

Quest Items are items that are required for a quest. These items are typically obtained when accepting a quest, killing an enemy, or found in the world. Quest items are handed in upon a quests completion and removed from a players inventory.

3.1.4.2 Equipment

The player can wear equipment that provide stats. There are 7 equipment slots in a players inventory; Head, Chest, Arms, Legs, Feet, Main Hand, Off-Hand. Each piece of equipment has a required level for equipping, and a slot it goes in. Additionally, items will be restricted to certain classes. For example, only a ranger can equip a longbow. The main hand and off-hand slots are for weapons, while the rest are for armor.

3.1.4.3 Consumables

Consumables are 1 time use items. These items will have some action associated with their use. For example, a lesser healing potion will restore a small amount of health when used.

3.1.4.4 Crafting Materials

Crafting Materials are items that are used in crafting. These items will be consumed in a recipe to create a new item. For example, a red herb and a yellow herb may be combined to create a lesser healing potion. Crafting materials are either collected from the map or dropped by an enemy.

3.1.5 NPC’s and Quests

NPC stands for non-player character. NPC's are essentially the actors of the story in the game. The story is driven by quests, which are kept track of in a quest log. The quest log will display all quests, their requirements for completion, and where to go. NPCs will distribute quests to the player through Dialogs (section 3.1.5.1).

3.1.5.1 Dialog

NPC's will use dialog interactions with the player to give info, begin and end quests, or begin a transaction (section 3.1.6.1). The player will be able to initiate a Dialog with any friendly NPC. The Dialog will have at least 1 option (minimum 1 so that you can always have an end chat option). Dialogs will clearly depict which dialog options begin a quest or turn in a quest.

3.1.5.2 Quests

Quests drive the storyline and the player to progress. Each quest will have a single paragraph summarizing the quest. Each quest will have 1 or more objective, (talk to someone, kill something, collect an item, go somewhere...etc.) and a defined start and end location. These values will all be displayed in the quest log. Additionally, to make quest givers and quest reward givers easily findable they will get a '!' or '?' above their head respectively.

3.1.6 Multiplayer and Characters

Multiplayer is implemented using a distributed system where heavy computation is done by the server. The protocol for

3.1.6.1 Transactions

Transactions are a transfer of items between two entities. This will either be between two players, or a player and a NPC. This is a critical section of code and must be handled in a safe manor.

3.1.6.2 Relay Server

The server behind the game needs to receive messages and relay the message to other connected clients the message is intended for. Additionally, the messages are consumed by an action validator as described in section 3.1.6.3.

3.1.6.3 Action Validator

The action validator will inspect all messages sent by the clients to make sure they follow protocol(see section 5 for more on the protocol). Additionally, it will spawn threads to update the database(section 3.1.6.5) or the NPC handlers (section 3.1.6.4) with any changes if necessary.

3.1.6.4 NPC Handler

The NPC Handler shall control the NPC's in one zone. As such, there will be 1 NPC Handler for every zone. These could potentially be on different nodes of the server if required. It will update all clients connected to that zone of an NPC's next action, or the response of a dialog. When a client connects to that zone it shall also be responsible for informing the client of all NPC's in the zone.

3.1.6.5 Database

The database is used to store all player information including stats, position, level, quest progress, and items. Additionally, the database shall store the file names of the local files containing sprites, map info, and music files. These files shall be stored as binaries in the client under the game directory.

3.1.7 User Interfaces and Menus

The user interface is the portion of the game that the user interacts with. There are various menus that allow the user to access options, skills, items and see their current state. The various UI elements that appear on the screen at all times are part of the HUD. The HUD is meant to allow quick viewing without requiring the user to move from their usual viewpoints.

3.1.7.1 Main Menu

The main menu is the start up screen of the game. Here the user is walked through log-in, character selection, and has the chance to edit their options. After logging in the main menu will let you choose or create a character. Creating a character is done with the Character Creation interface (section 3.1.7.2).

3.1.7.2 Character Creation

The character creation interface will let the user select their style options and name. Name will be validated on the server to ensure no duplicates.

3.1.7.3 Action Bars

Action bars are a main element of the HUD. These are two lines of 12 boxes that span the bottom of the screen. These boxes will allow for any action or spell to be assigned to the box. They will also display the cooldown on any particular action. Furthermore, each box can be assigned a unique key that can be pressed to execute the action.

3.1.7.4 Stat Bars

There will be visual representations of the three main entity stats;Health, Mana, and Experience. Each will be represented by a bar that fills and empties as the Stat is gained or lost.

3.1.7.5 Inventory

This will be a menu that shows all the players equipment and items. This will be represented in a grid format similar to the action bar. These items should be easily move-able through right-clicking or dragging and dropping. A duplicate menu will be used through an NPC to access your bank.

3.1.7.6 Skills Menu

This menu will display the players selected skill tree. This will display the learned, able to learn, and locked skills. A user will be able to select a skill to learn. This will let them use it on their action bar.

3.1.7.7 Stats Menu

This will be the full stat breakdown of the player. This menu will also allow for spending stat points awarded from leveling up. Stat points let the player increase their base stats however they like.

3.1.7.8 Dialogs

Dialogs are simple interactive menus that let the user talk to NPCs. They will be able to show multiple conversation options. They will provide a silent transaction with server entities. See section 3.1.5.1 for more on dialogs.

3.1.7.9 Options Menu

The options menu allows the user to change settings about their game. This will include, window size, volume settings, and graphic settings.

3.2 Non-Functional Requirements

3.2.1 Scalability

A typical software engineering project never reaches 100% completion, always leaving room for improvement and additional functionality. Additionally, there are lots of features that cannot be implemented in the required time span. These features would be implemented in future versions of the product. As our player base grows, the load on our server would grow and cause a degradation in the quality of the game. For this reason, scalability is a crucial point we kept in mind while developing this application.

3.2.2 Performance

Two key features, Multiplayer, and the Combat System, require a fast, real-time updating aspect. This means that any loss of data, or lag can severely reduce the quality of the game. As such, the performance of the game is also a crucial point. If there is any network latency or lag, the user should not experience it to make the gameplay as smooth as possible. The solution to our performance requirement is tied in with the solution to our scalability requirement.

3.2.3 Security

Since the game will be an online multiplayer experience, users will be required to register with email to an account before playing. Any space online that allows multiple users to interact demands privacy and security, and our game follows suit. User authentication, and anti-cheating countermeasures are all discussed in more detail in the Implementation section.

3.2.4 Reliability

Reliability is another important non-functional requirement. A game that is not reliable cannot be played, and a game that cannot be played has failed as a game. When a player inputs a command to do something, the expected action should occur. Our system's storage should be reliable enough to prevent players from losing their progression data. These are very important to ensure the success of any game in the industry.

3.2.5 Portability

Portability is all about making our game multiplatform. The XNA engine allows games developed with it to be ported to microsoft's XBOX with ease. Allowing our players to play the game on a variety of platforms while ensuring that the game remains the same is a concern that we address. By making our game portable, we also allow future developers to enhance our game on any of the compatible platforms. As a result of this flexibility, the lifespan of the game in the market can be prolonged.

4.0 Design

4.1 Design Decisions

The application is intended to provide an exciting, engaged, and unique storytelling experience. There were a number of challenges and limitations in developing such a large scale game. After gathering requirements, some critical design decisions needed to be made.

4.1.1 Distributed design

To properly validate all user input for errors takes a decent amount of computation. To ensure that this computation can be done quickly, stress needs to be eliminated from the server. The distributed design does this by relying on the client to do the heavy computation. Additionally, each client contains identical copies of all the game content such as sprites, NPC dialog, and map info which distributes the load from the server to send these files to the client[4].

4.1.1.1 Drawbacks of a Central Server

A central server requires a very powerful host computer that can store a large amount of data as well as handle a large number of updates. It would have to manage every action of every entity in every zone and would quickly become a single point of failure or a bottleneck [4].

4.1.1.2 Drawbacks of chosen design

To implement the pattern and really alleviate the load, NPC movement must be calculated on the client and the new position reported back to the server. Since the direction, or action of the NPC is given to each client they should all calculate the movement the same. However, this

adds a bit of complexity to the structure in trade of performance. Additionally, without multiple nodes this design can still be a single point of failure.

4.1.2 TCP versus UDP

UDP is commonly known to be less reliable than TCP. UDP is like throwing the message into the internet and hoping it reaches its target. While TCP is like shaking hands with someone and then passing them a piece of paper. The protocol has built in functionality for ensuring packets are not lost, corrupted, or duplicated. In a real-time system like this, reliability is especially important. While most common for video games to use UDP, a similar game and the biggest inspiration for Battle University, World of Warcraft uses TCP[5]. These combined ultimately led to our decision of TCP.

4.1.3 Console Based Server in C#

While designing the server it was noted that it is not necessary to have it in C# like the rest of the game and could instead be in another language like java since it does not need to use the XNA library, only the TCP protocol. However, since the client and server need to have mirrored code for reading and writing different types of messages, it made more sense to leave it in C# in favor of reusing code and keeping the server in the same IDE increasing maintainability.

4.1.4 Handling Connection Errors

For any real-time system using the internet, it is likely to encounter connection issues.These can cause problems for the user and the server if not handled gracefully. As such, the server should handle all message sending and receiving asynchronously to not get blocked by a timing out

connection. Furthermore, if a client times out their connection is closed and a disconnect message is relayed to all relevant clients.

4.1.5 2D vs 3D

The decision between 2D and 3D was made very early. 3D video games are very popular in the industry, however it adds the requirement to 3D model textures using special applications. Additionally, 3D video games are most commonly developed in either Unity Engine or Unreal Engine. Both of which have a very steep learning curve for the developer. In contrast, 2D textures can be easily made in any image editor and there is a wide array of possible engines or libraries that can be used to develop a 2D game. Therefore, simply due to the skillset possessed by the group, a 2D style was chosen in favor of labour costs.

4.1.6 XNA Framework / MonoGame Engine

The choice of engine or framework with a 2D game is tough as there are a wide array of options. When deciding which engine to use we looked at 4 main concepts:

- Developer learning curve
- The feel of games made in that engine (Mainly movement)
- The restrictions imposed by the engine
- The tick rate of the engine

Once going over a few we narrowed it down to 3 choices.

4.1.6.1 Unity 2D

Pros

Unity is the most popular game engine used right now meaning it will have a lot of documentation and tutorials available. Moreover, they allow you to use their engine in a commercial game for free until you earn $100k with the project. Unity 2D games feel very smooth as they have built in functionality for physics. There are very limited restrictions on the engine and even the tick rate is variable.

Cons

The unity IDE uses a lot of visual scripting which implies a learning curve of familiarizing oneself with the IDE to even begin. Moreover, code in Unity is all in C# which only one developer on the team is familiar with.

4.1.6.1 RPG Maker

Pros

RPG maker is for doing just that, making an RPG. RPG maker is a great choice because of this. Games made in RPG maker are very easy to make and require minimal knowledge to do.

Cons

RPG Maker only allows for limited scripts using javascript if the functionality is not already in the visual editor. A visual editor means learning the IDE before you can even begin. Multiplayer is not supported by the engine, and therefore would have to be in javascript entirely. Games made

in RPG maker are turn based, restricting our ability to make the movement and combat feel satisfying. Lastly, the tick rate is low, meaning a real-time system would be slowed down.

4.1.6.1 XNA Framework / Monogame engine

Pros

XNA uses a very simple update-loop that ticks 60 times per second. This means that development is very straight forward. Games made in this engine like one of our inspirations Terraria have a great feel to them in both combat and movement that is easily replicable. Tutorials exist (though limited) of how to implement multiplayer and many of the key features of Battle University.

Cons

XNA is less popular (Top 20 in terms of popularity, whereas the other two options are top 10), meaning less documentation and tutorials than a more popular engine. The code is in C# which only one developer on the team is familiar with.

Overall, XNA was the choice as the learning curve is only moderate due to it's simple structure and C# being extremely similar to Java. Additionally, the engine has a high tick rate of 60 ticks per second which is also why the movement and combat in games made in this engine feel so good. Lastly, and most importantly, the engine is very non-restrictive and some of the most complex functionality can be implemented with creativity.

4.1.7 MySQL database

MySQL is free, open source, and very light weight. Additionally, all team members are familiar with MySQL. Originally, a NoSQL database was considered to store sprites and content of the game that would not need to be stored local to each client. However, doing so would add weight to the server and go against the distributed architecture. As such, the additional functionality of NoSQL was deemed unnecessary and MySQL was chosen. An added benefit is that MySQL does not require a commercial license since we are not redistributing MySQL with the Client[6].

4.2 Network Protocol

To properly implement a networking game a standard protocol for network traffic must be designed. Since the communication is TCP, messages can be variable widths without the need for an end of message indicator. With this in mind, a simple design was created where the first byte of the message will always indicate the type of message. Dependent on type, the message will contain the necessary values for that message. All messages should be as small as possible and only the necessary information be passed over the network. Moreover, sent messages will begin anonymous, only identifiable by the IP they are sent from. The server who will always receive the messages first will append the ID of the user who sent the message to the end of every message before distributing it to other clients. This ID is strictly for messaging a client during the session and assigned when the TCP client connects. This ID does not persist upon disconnect, and can be reassigned as needed. For specifics on types of messages please see section 5, implementation.

4.3 Use Cases Description and UML Sequence Diagrams

A large scale application like this has thousands of use cases and as such we decided to model the critical sections and other important functionalities.

4.3.1 Use Case Descriptions

4.3.1.1 User connects to a server with another user on the same map

In this case, there is a client on the server (Client2) and Client1 joins the same map as them. The server entity here does not show interactions of its internal subsystems.

1. The user enters login details to the client
2. Client1 sends info to the login server
3. Login server verifies the info and returns success
4. The user selects character
5. Client1 attempts to establish connection with game server
6. Server responds connected
7. Client1 sends MapJoined message
8. Server relays message to Client2
9. Client2 responds MapJoined
10. Server relays message to Client1
11. Client1 sends enemyLoad message
12. For each NPC on the map:
    a. Server responds with a message for each NPC and monster on the map.
    b. Client loads entity

4.3.1.2 User moves with another user on the same map

In this case, there are two clients on the server, and one client moves.

The server entity here does not show interactions of its internal subsystems.

1. User inputs movement to Client1
2. Client1 sends Player Moved message to server
    a. Client1 displays movement to user
3. Server relays message to Client2
4. Client2 displays Client1 moving

4.3.1.3 User kills an enemy with another user on the same map

In this case, there are two clients on the server, and one client attacks and kills an enemy.

The server entity here does not show interactions of its internal subsystems.

1. User attacks using Client1
2. Client1 sends weaponCreated message
    a. Client1 swings their weapon
3. Server relays message to Client2
4. Client2 displays weapon swinging
5. Client1 sends enemyDied message
6. Server relays message to Client2
7. Client2 displays enemy death

4.3.1.4 User gets hit by an enemy with another user on the same map

In this case, there are two clients on the server, and one client gets hit.

This use case is intended to depict how the server's internal subsystems interact.

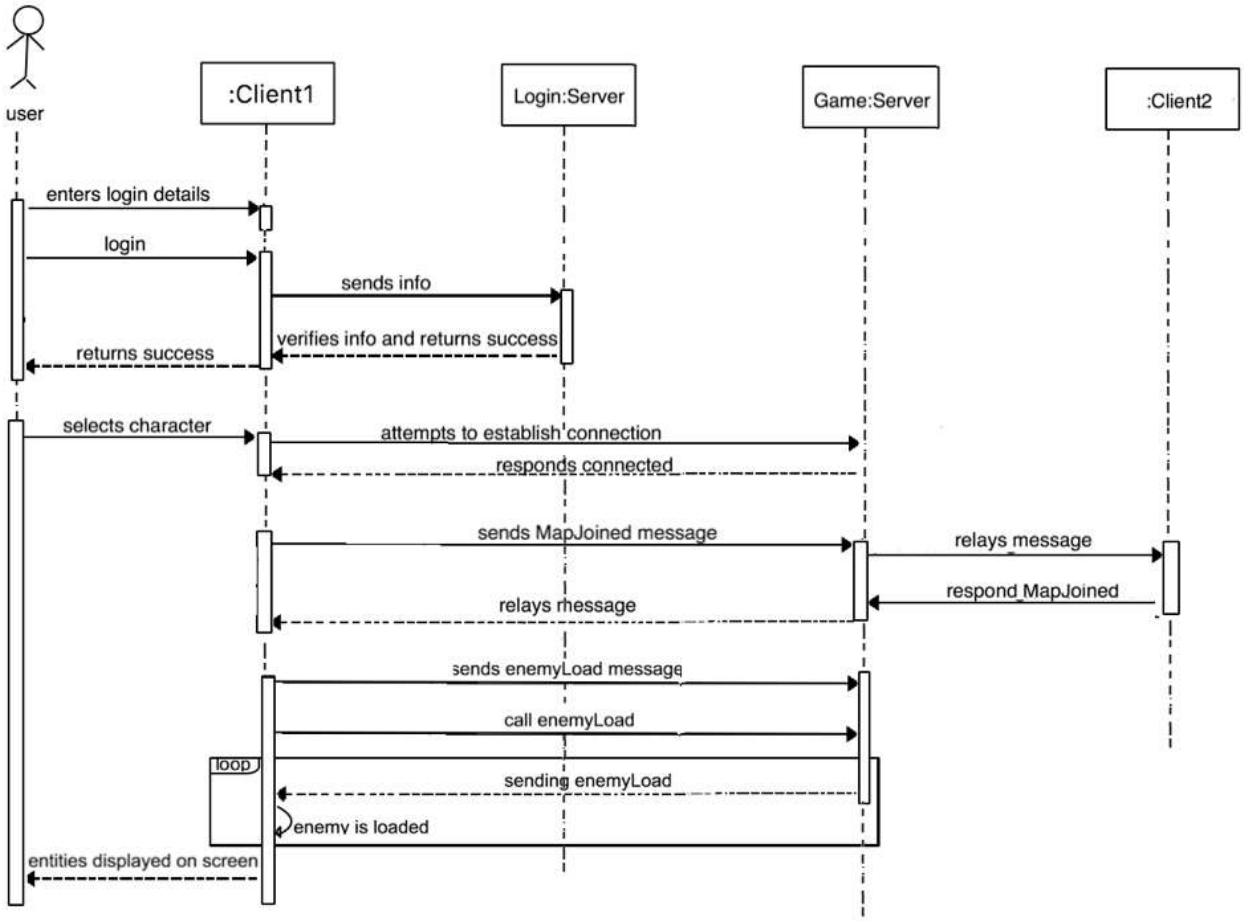
1. Client1 sends playerHit message
2. Listener creates action and sends to validator
3. Validator inspects message and sends to Dispatcher
4. Dispatcher relays message to the client
    a. Dispatcher sends message to the database link
5. Database link updates database
6. Client2 displays player getting hit

4.3.2 UML Sequence Diagrams

To demonstrate how the client and server interact, some of the interactions have been broken down into sequence diagrams. Additionally, a sequence diagram has been made to demonstrate the sequence of the internal server subsystems.

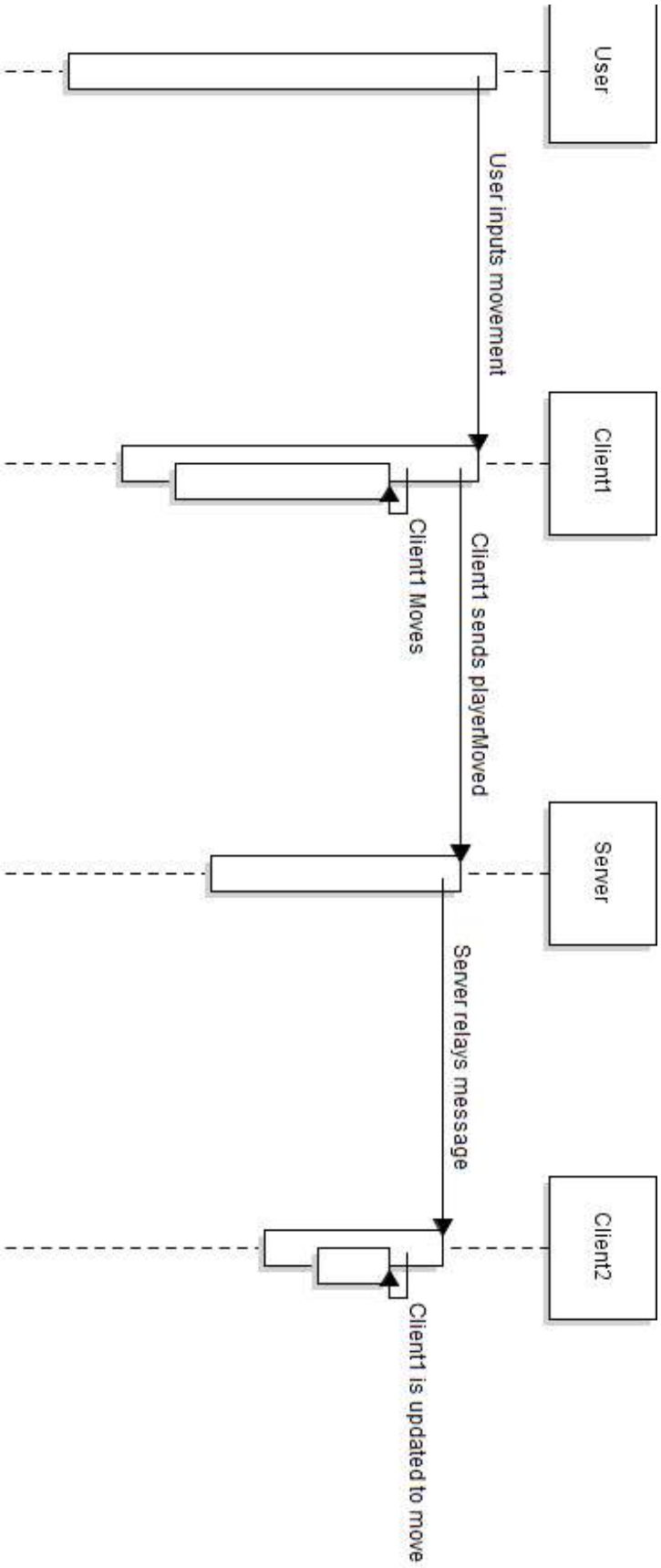
4.3.2.1 User connects to a server with another user on the same map

Figure 4-1: Client connects



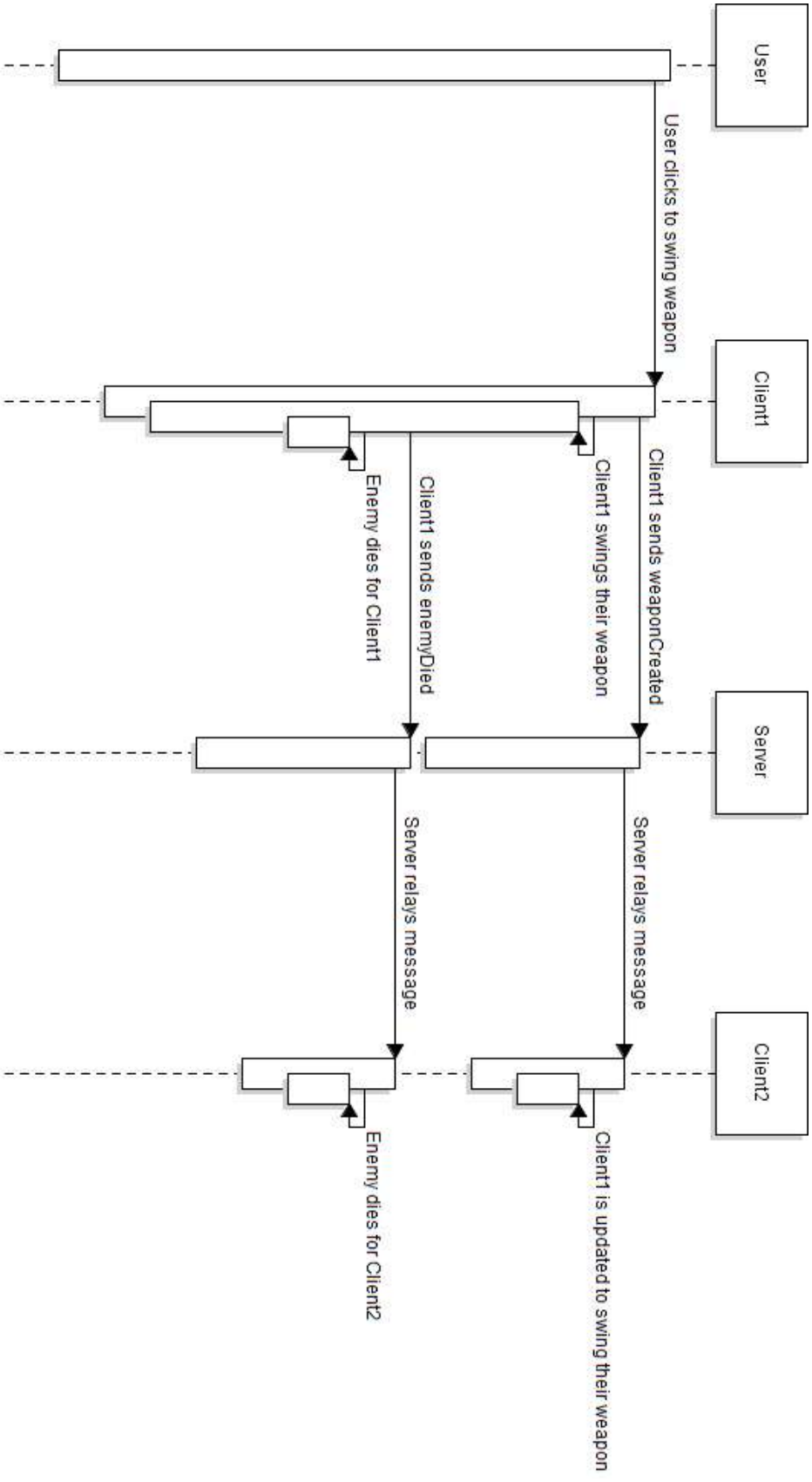
4.3.2.2 User moves with another user on the same map

Figure 4-2: User Moves



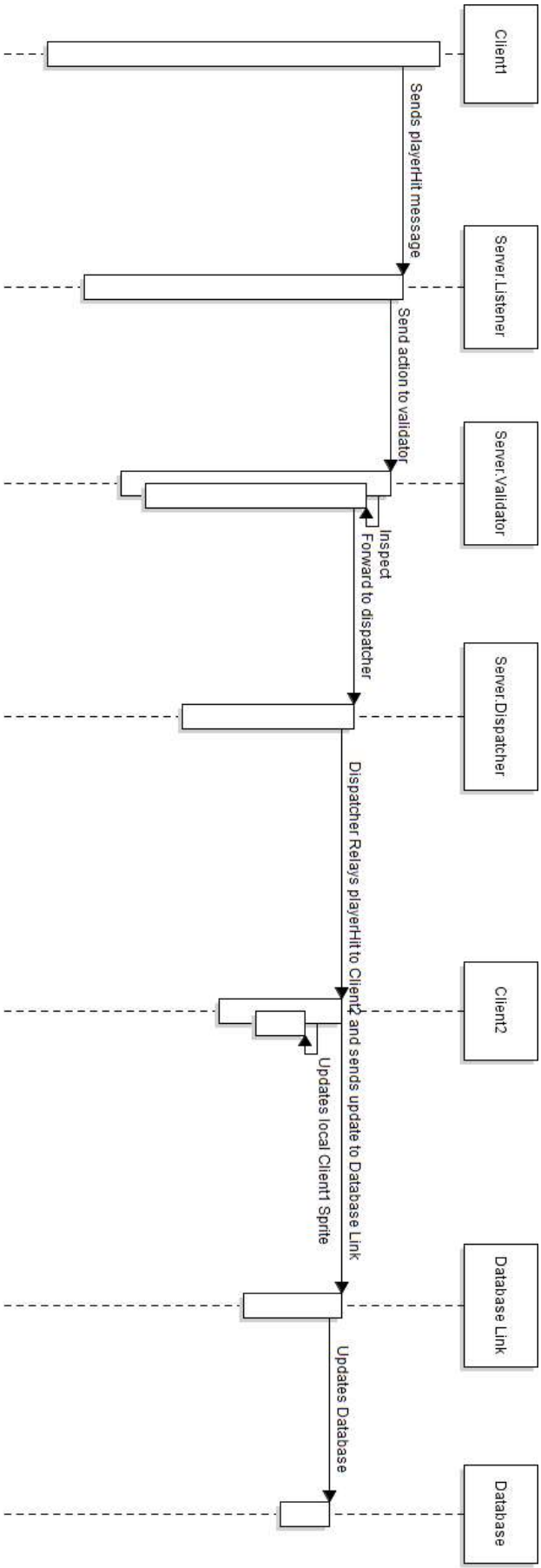
4.3.2.3 User kills an enemy with another user on the same map

Figure 4-3: User kills an enemy



4.3.2.4 User gets hit by an enemy with another user on the same map

Figure 4-4: User gets hit by an enemy



4.4 UML Design Diagrams

4.4.1 Component Diagrams

Our system is made up of two subsystems: The server and the Client. The server acts as a relay between clients, a validator on actions, the source of NPC info and monster actions, and as the persistent storage for player information. The client calculates all visuals, physics, collision, and movement. It is also the source for almost every message aside from the time reliant enemy action changed messages produced by the NPC handler on the server.

4.4.2 Class Diagrams

4.4.2.1 Application Class Diagram

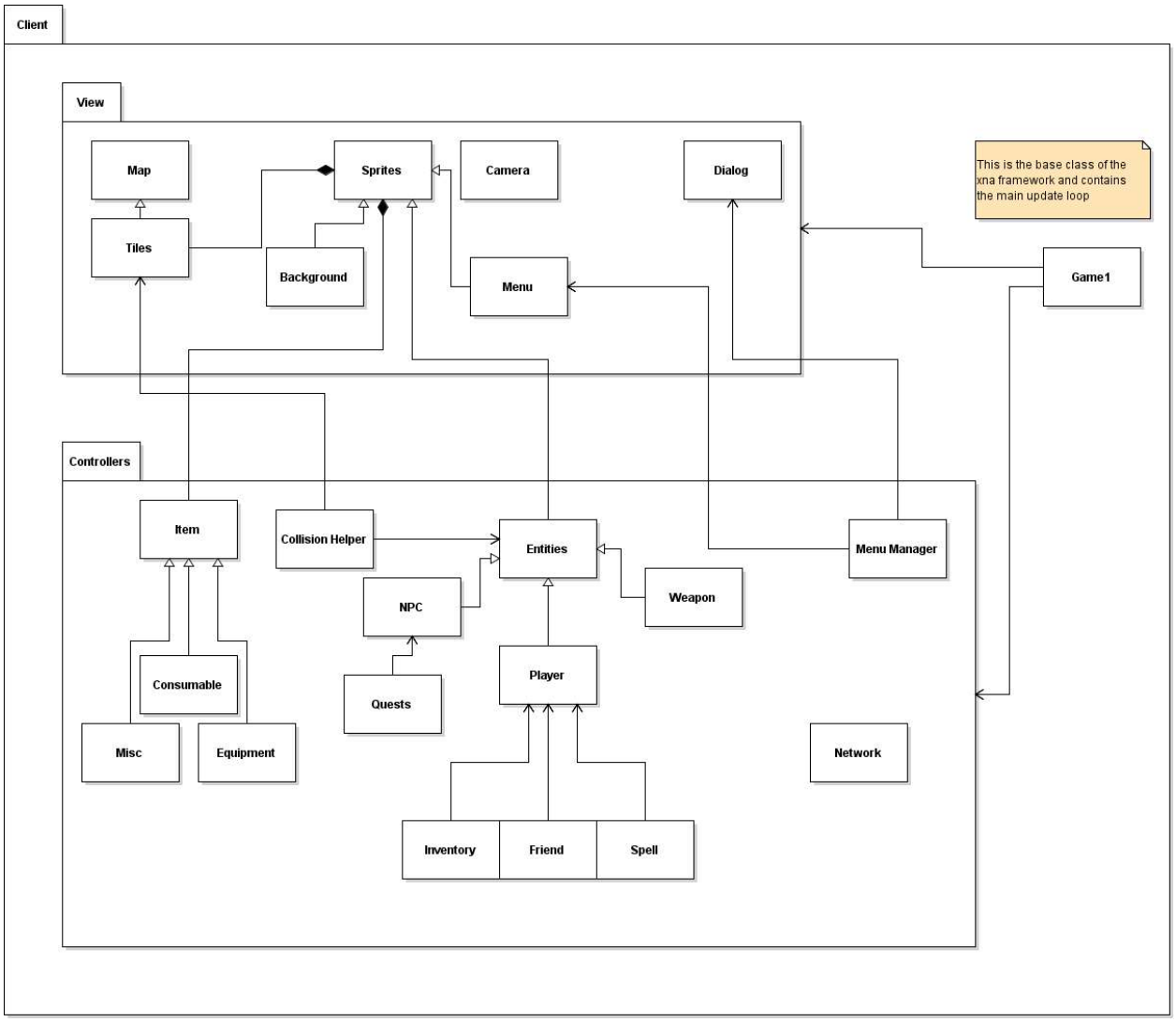


Figure 4-5: Application Class Diagram

4.4.2.2 Server Class Diagram

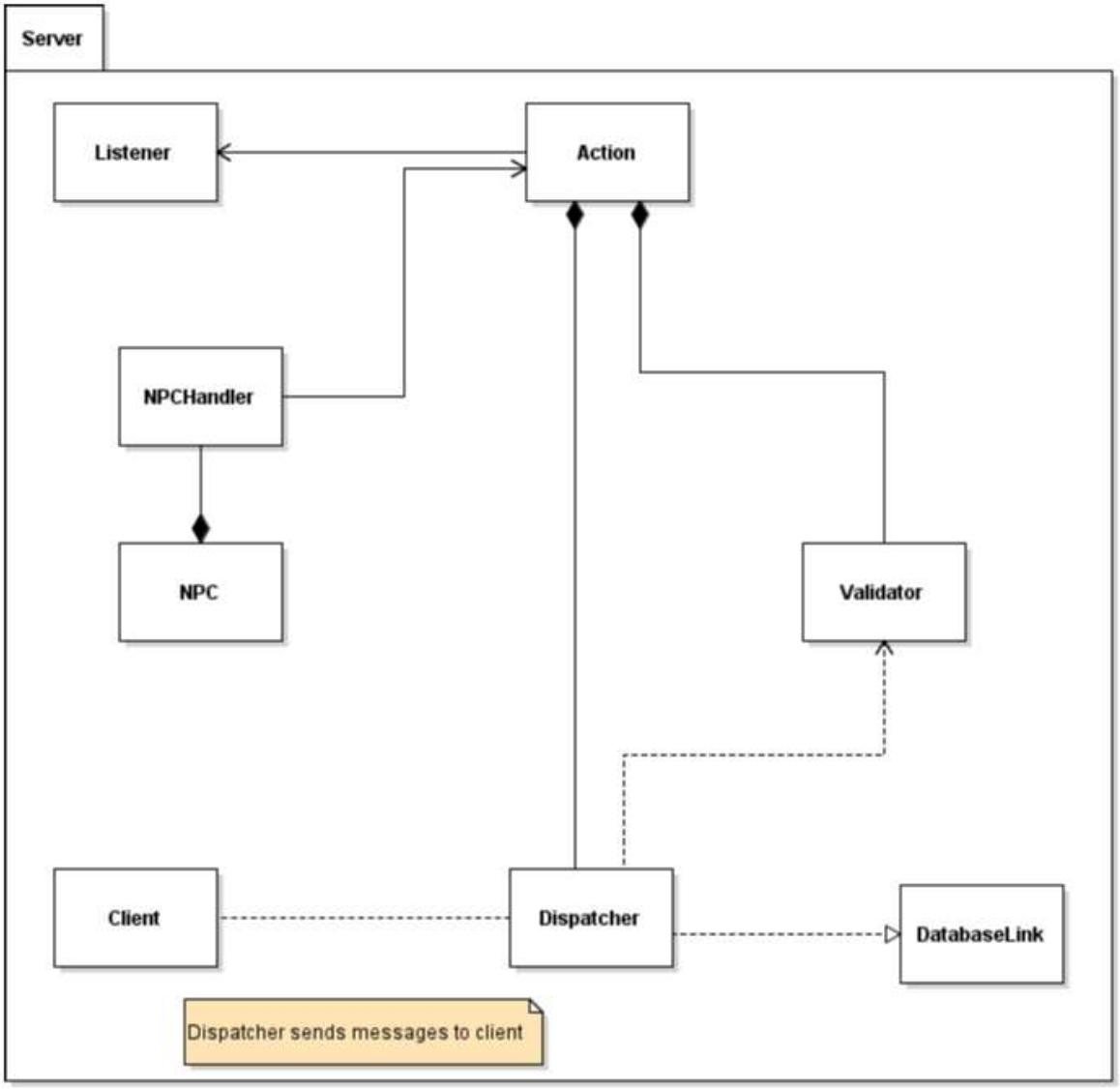


Figure 4-6: Server Class Diagram

5.0 Implementation

The majority of the design was successfully implemented with little to no changes. The technical details of the implementation are outlined here.

5.1.1 Entities

Entities were implemented similar to the design. The base class of the game, Game1, contains two lists of friendly entities, one for players, and one for NPC's, a list of enemy entities, and a list of weapon entities that are all populated by the server.. The base class of entities implements useful methods used by its children such as animateLeft and animateRight. This animation is done by creating a small rectangle out of a large picture containing each frame of the animation. The rectangle is moved along the picture according to the current frame. Which is increased or decreased using the animate methods. Below is a sample of the type of image that is cycled through:

Figure 5-1: An animated sprite image

5.1.1.1 Players Entity

The player entity acts as a local cache of the player's details that are loaded upon log in and character selection. It implements methods for movement and combat using the ASD keys, the

space bar and right and left click. This is done by checking the state of these keys on the users keyboard in the update method. This also contains code for collision detection with enemies.

5.1.1.2 Enemy Entity

The enemy entity is similar to the Player entity but only uses the stats health and strength. It contains a method for movement dependent on the current action of the enemy. This action is updated from the server, which sends new actions to the enemy every 3-6 seconds. Exact length is randomly selected to avoid NPC's moving in one big swarm. This class contains a method for collision detection against weapons. Enemy entities can either be passive, meaning they will run when attacked, or aggressive, meaning they will chase you when attacked.

5.1.1.3 Friendly Entity

Friendly entities have a movement method only used on the friendly players list that computes movement the same as the player only with no checks on keyboard state to add velocity. Instead, the velocity is updated when the client receives a player moved message, as well as the position synced. NPC's do not use any movement methods. They instead have a method to detect if a player is touching them and also hits the ENTER key to begin dialog with them. The implementation of dialogs is discussed in section 5.1.5.1.

5.1.1.4 Weapon Entity

The weapon entities are used for weapons and spells. It contains methods to rotate, move, and change color. Whenever a player attacks or casts a spell, a weapon entity is created and uses the attack method to determine its movement. The attack method is overloaded for each child type of weapon so each type of weapon can be uniquely coded. Additionally, when a player uses a weapon it is temporarily set on cooldown before it can be used again.

5.1.2 Platforming

Platforming was implemented as planned in the design. The base class of the game, Game1, contains two map objects. One for the collision tiles, and one for the background tiles. Background tiles have no collision and thus must be separated from the collision tiles.

5.1.2.1 Tiles

Tiles were implemented using a single character to discern the type of tile. This allows for 256 different tiles. A tile is a 50px by 50px square and all tile data is stored locally.

5.1.2.2 Tile Maps

Tile maps are implemented using text files stored locally in the game directory. The name of these files are stored on the server, and provided to the player when the map is switched or game is loaded. These files are literal 2D maps of the tiles, using the single character that represents the type of tile at each point. Each map has a second file with same name but the prefix back that is used to load the background tiles of the map. A sample text file representing a map is depicted on the next page in Figure 5-2.

Figure 5-2: World1 Tile Map File

5.1.2.5 Collision Detection

A collision helper was created to do special collision detection for tiles. Unlike normal collision detection, this checks if an entity is touching the top, bottom, right or left of the tile, and provides velocity in the opposite direction to prevent the player from going through.

5.1.3 Combat System

The combat system is only partially implemented, containing only 1 type of weapon and 1 type of spell. However, the entity stats were all implemented as discussed in the design. Since these attributes are described in the design section, there is no need to repeat it here. In the implementation of the combat system, leveling up allows the player to increase a selected

attribute. Any entity that is hit is made briefly invulnerable to additional attacks, knocked back slightly, and turns red to indicate invulnerability and damage. There is also just 3 types of entities implemented; Slimes, Zombies, and a Giant Slime.

5.1.3.2 Spells and Weapons

As mentioned, the partial implementation of weapons and spells contains only 1 type of each. A sword weapon and a heal spell. This is intended to be the only skills you have in the first year and you unlock access to more as you progress through additional years. Weapons have cooldowns to delay their use in quick succession.

5.1.4 Item System

The item system is partially implemented. The player is provided with one item in first year, a sword. Further weapons are unlocked as you progress through the years. Since only first year is implemented, the item system is only partially implemented.

5.1.5 NPC’s and Quests

NPC’s and quests are fully implemented as defined in the design section. Their implementation is talked about in the Entities section.

5.1.5.1 Dialog

NPC’s begin dialog when the user is touching them and presses the enter key. This opens a dialog menu containing different text depending on quest eligibility and status. The text for this dialog is loaded from client side files, the name of the file being provided by the server.

5.1.5.2 Quests

Quests are implemented using a Quest class. The quest class acts as a local cache of quest details so the server need not be concerned each time progress is made. It contains target entity, quest status, quest progress, and quest requirements. When a new quest is started it is loaded into a list of quests in the Game1 class where this info is kept. Each time the user logs in this quest list is repopulated.

5.1.6 Multiplayer and Characters

Multiplayer was implemented as described in the design with the exclusion of transactions. This means players and NPC's can not trade items. The server is implemented using mostly the native TCPClient class but it is wrapped in the listener and dispatcher as depicted in the server class diagram in section 4.4.2.2.

5.1.6.1 Protocol

To implement different actions, different messages are needed. The first byte of every message over the network depicts the type of message. Each type of message can contain different associated values. The relationship between message type and its contents is depicted on the in Table 5-1.

Note: The automatically appended player ID is excluded from message contents

Message Type	Indicator Byte	Contents	Relay Message?
Disconnected	0	empty	Yes
Connected	1	empty	Yes
MapJoined	2	int X, int Y, string map, string name	Yes
SavePlayer	3	String name, int health, int maxhealth, int mana, int maxmana, int strength, int mspeed, int aspeed, int intelligence, int experience, int level, int x, int y, string map	No
PlayerMoved	4	float velx, float vely, int posx, int posy	Yes
WeaponCreated	5	int attackType	Yes
EnemyActionChanged	6	int id, float x, float y, int action	Server Generated
EnemyHit	7	int id, float x, float y, int action, int hp	Yes
EnemyDied	8	int id, float x, float y, int action, int hp	Yes
PlayerHit	9	float velx, float vely, int posx, int posy	Yes
EnemyLoad (from server)	10	String map, int id, string texture, int health, int strength, float x, float y, int width, int height, int frames, int spawntimer, bool isPassive, int action	Server Generated
EnemyLoad (to server)	10	String map	No
EnemySync (from server)	11	Int id	
EnemySync (to server)	11	String map, int id, float x, float y	No

NPCLoad	12	String map, int id, string texture, float x, float y, int width, int height,	Server Generated
NPCDialog	13	String map, int id	No
QuestProgress	14	Int questID, int status, int progress	No

Table 5-1: Server-Client Protocol

5.1.6.2 Database Schema

The database is implemented as described in the design. It stores all player information including stats, position, level, quest progress, and items. Additionally, the database stores the file names of the local files containing sprites, and map info. The simple schema is depicted in figure 5-3 on the following page.

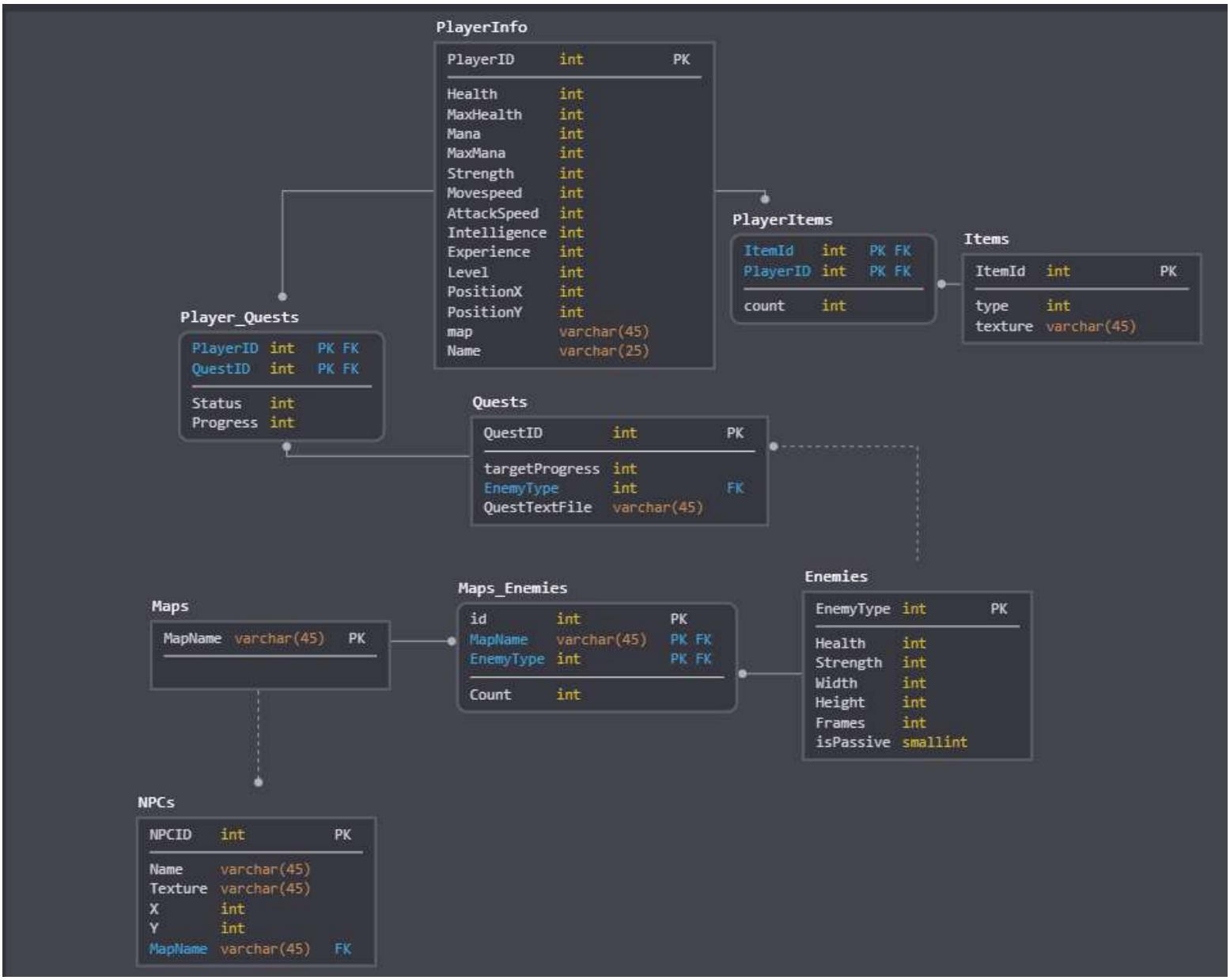


Figure 5-3: Database Schema

5.1.7 User Interfaces and Menus

The user interface is partially implemented. Functionality for the skills menu, action bar, inventory and skills menu was moved to planned functionality and therefore the user interface for it was not implemented.

5.1.7.1 Main Menu

The main menu is the start up screen of the game. Here the user is walked through log-in, character selection, and has the chance to edit their options. After logging in the main menu will let you choose or create a character. Creating a character is done with the Character Creation interface.

5.1.7.2 Character Creation

The character creation interface will let the user select their style options and name. Name will be validated on the server to ensure no duplicates.

5.1.7.3 Stat Bars

The visual display for health and mana were implemented into the stats menu. Their implementation can be seen in figure 5-5 below.

Figure 5-5: Stats menu and stat bars

5.1.7.7 Stats Menu

The stat menu is accessible by holding the shift key, which extends the stat summary to the full menu. This is also done automatically when the player levels up to allow them to select a stat to upgrade. The stat menu in both summary and extended form is depicted above in figure 5-5

5.1.7.8 Dialogs

Dialogs are simple interactive menus that let the user talk to NPC's and begin or turn in quests. It is a simple box with text and buttons that allow the user to select dialog options. Figure 5-6 below depicts the implementation of the dialog menu.

Figure 5-6: The first dialog menu of the game

5.8 Non functional requirements implementation

5.8.1 Scalability and Performance

We are handling this scalability issue at the design level by making use of Asynchronous programming and multithreading at the server side and multiple server nodes. Multithreading allows our server to serve clients in the queue faster. In a single-threaded server, client transactions and communications are queued up. The delay between the client sending out a request and the client getting a response would result in visible lag on the players' perspective. Having a multithreaded server eliminates this queue and clients can communicate with a server of their own thereby limiting lag. There will be cases where the base server will be overloaded with requests from clients and where multithreading will not be enough. Taking this into consideration, we have also introduced load balancing by implementing functionality for multiple server nodes. Requests from clients will undergo load balancing and be directed to different server nodes to prevent overloading. This will further reduce stress on our servers.

Asynchronous server design aids with time taken for a server to load up items and entities on the map (game world). This common issue occurs in large video games. When a player enters a new map and the map is populated with items and entities that are supposed to be there. When in a scenario where a lot of items and entities need to be loaded in, the server suffers and the player could end up lagging in an empty map for some time while the server struggles to populate the dense location. When scaled, this issue could be more common. By having an asynchronous server design, we can alleviate this stress on the server. We have parallelized processes that can be parallelized as opposed to having them occur sequentially.

These are the methods we use to satisfy the scalability non-functional requirement.

5.8.2 Security

The security goal is met in two different ways in the implementation of the game. Firstly, when logging in, all network traffic is encrypted to ensure the users confidential information is protected. This is done using C#'s built in cryptography library. Secondly, all user input is validated before it is consumed. The validator in the server architecture ensures that the message follows protocol but that it is also within reason. For example, when a character is saved on a users disconnect, if the stats are too high for the players level, or the map or position is not the same as what the server thinks, the message is deleted and logged in the server output. This ensures that messages being sent are not being modified externally. Thus protecting the integrity of the messages.

5.8.3 Reliability

To serve this goal, we make use of the following 3 methods to ensure the reliability of our system.

- Condition Monitoring
- Passive Redundancy (Warm Spare)
- Degradation + Update

Condition Monitoring: This is the fault detection technique that we use to check on the status of critical components. For instance, We have our server, monitoring the state of our database. In the case of corruption or failure, through condition monitoring our server will detect the fault and prompt fault recovery techniques to restore our database.

Passive Redundancy (Warm Spare): This is the fault handling technique that we have employed. Passive redundancy periodically updates our backup systems. We want this done periodically rather than synchronously as the nature of our project is not so critical to warrant the extra expenses a hot spare would incur.

Degradation + Update: This is the fault recovery technique that we use in the event of an error in any of our components. When a fault is detected in a component, that component will be degraded to the state of the warm spare through an update to refresh the main component. Through these 3 techniques, we can ensure the reliability of our system.

6.0 Testing

6.1 Unit Testing

The main components of our project that were tested, are as follows:

- Character Progression (Levelling, stats, experience)
- Character inventory (items)
- Character damage (given and taken)
- Character Item Usage
- Zones (loading, NPCs, enemies, etc.)
- Quest assignment

6.1.1 - Examples of Unit Testing

Test Case 1	**Character Levelling**
Expected Result	When the player character gains experience putting them over the threshold for their current level, the current level of the player increments by one. If experience gained in a single transaction is large enough, multiple level-ups may occur.
Actual Result	Functions as expected
Result	PASS

Table 6-1: Test case 1

Test Case 2	**Character Damage (Received)**
Expected Result	When the character collision box makes contact with an enemy collision box (projectile, or body), the player character loses HP equal to the damage of the enemy collision box.
Actual Result	Functions as expected
Result	PASS

Table 6-2: Test case 2

Test Case 3	**Quest Assignment Completion**
Expected Result	When the player successfully fulfills the requirements to complete a given quest, the quest is marked as completed.
Actual Result	Functions as expected
Result	PASS

Table 6-3: Test case 3

Test Case 4	**Character Item**
Expected Result	When the player picks up an item, the item disappears from the zone, and the item is added to the inventory of the player character.
Actual Result	Items added to inventory, but items still persisted in overworld.
Result	FAIL

Table 6-4: Test case 4

Test Case 5	**Instantiating a Zone**
Expected Result	When a zone is loaded, the zone contains the correct number of enemies and NPCs.
Actual Result	Functions as expected.
Result	PASS

Table 6-51: Test case 5

6.2 Performance/Stress Testing

Performance testing for this project is primarily focused on two separate issues: Rendering the game (local), and rendering other player-characters (network based).

For local performance testing, the goals of the testing are to overload the number of computer-controlled entities, to reduce the performance of the engine running the game. This was achieved by generating the starting-zone of the game, and placing many collision-capable objects into the zone. In our tests, it was determined that at 30 entities (combined count of NPCs, players, and enemies), our game begins to slow to a noticeable degree. What is noteworthy about this, is that this rule does not seem to apply to objects that do not have collision detection; i.e. swords can be spawned by the 100s without a noticeable decrease to performance. This has led to the theory that a refactoring of our collision code (engine-based) could lead to performance gains.

For network based testing, the goals of performance testing are to overload the number of player-controlled entities, to slow down the receiving of data over the network, getting the game into bad states. Our game was able to support up to 30 player-characters before we started to get noticeable lag, but determining if this is due to our hard-limit on entities with collision ability, or due to our networking code slowing down, is difficult.

6.3 Integration Testing

Piecing different parts of the project together are covered in Integration Tests. Largely, these tests aim to test correct behaviour between a given client, and the database connection.

An example of integration testing done, is as follows:

Creating and Persisting a Character

1. Create a character
2. Play the character, level them up, give them stats that give them individuality
3. Log-out of the game, closing the program.
4. Restart the program, log back in as the user
5. Check the stats and inventory of the player - they should have been persisted in the DB on logout, and pulled from the DB on login.
6. Ensure data Integrity

This style of testing allows for different parts of our codebase to be modularized, and tested together - the core idea of integration testing.

Another example:

Character Interactions over a Network

1. Log into the game as your player-character.
2. Have someone else log into the game, and position them near your character.
3. Have the second player perform a basic action (moving, jumping, attacking, etc.)
4. Ensure that the first player is able to see the second player being updated in relative real-time.

This test checks our network integration into our project, ensuring that the base game, as well as the online component, are able to function together, and function correctly.

6.4 Test Stubs

Our test stub used the most was a player-simulation, that we used to simulate a “simple” online player. The test stub would generate a base-level default character that had the ability to move towards the left. This was easy to generate, and was used in both our stress testing, and integration testing.

For stress testing, we spawned many of these “simple” player characters, and had them all walk towards the left. Like mentioned above, we capped out at about 30 collision enabled entities on the starting map.

For integration testing, we used this test stub to generate a single character, and ensured that as this character walked across the screen, we ensured that on our separate instance of the running game, that our screen was updating correctly.

7.0 Conclusions

When the project was first conceptualized, we were aiming to break into the video game industry by following the lead of the people at the top in the industry. We knew right away that a game that could break into the industry would require a lot of work. After requirements elicitation and planning, we began to understand just how difficult this problem is. So we came up with reasonable deadlines to have the bulk of our initial goals complete and a clear path forward for the future of the game.

Following our deadlines and plan we achieved a good amount of fully completed functionality. Some of the functionality of key systems is still only partially implemented and did not make it into the demo. These partially completed functionality are intended to be implemented in the current phase. After which, the project would move to its next phase where our planned functionality would be implemented. The details of all the fully complete, partially complete, and planned features are described below. Additionally, see appendix A attached at the back of the report for future developers who join the team in need of a crash course on the project.

7.1 Fully Complete Functionality

7.1.1 Entities

As discussed in the implementation the entities are the main actors within the game, including the player. Their movement and interactions with enemies and players is all complete. The architecture has also been set up so that any new types of weapons can simply inherit from the weapon class and override the attack method. They have been set up so that the entities can have variable size, animation frames, health, textures, and damage values which are all pulled from the database. Additionally, all entities are synced by the server and work flawlessly in multiplayer.

7.1.2 Platforming

A key feature of the game, and one of the most popular with the users who helped test this game, platforming, is fully implemented. Map files can be generated easily with any text editor, additional tiles can be added without changes to the code as long as the textures are added to the content of the client. Gravity, collision detection, and movement are all fully complete.

7.1.3 NPC's and Quests

This functionality is what makes the game an MMORPG and not just a mindless killing game. The NPC's and quests are fully implemented and only require additions to the database to add more. There is already a first year of quests in the game that was used for demoing the functionality. A dialog box class on the client side can be used to do more than just accept quests as well.

7.1.4 Multiplayer and Characters

This game features no single-player and is strictly multiplayer so this is a key feature. An asynchronous and scalable server that can be split into multiple nodes is fully complete with a database. All current multiplayer interactions are documented in the protocol in section 5.1.6.1 as well as the database schema in the section after, 5.1.6.2. The server contains an NPC Handler class intended to be replicated for as many maps are in your game. The NPC handler class can be used to define any new entity behaviour (for example a boss with special attacks). Moreover, the Validator class can be used to do any checks on messages flowing through the system. Lastly, the server relays all messages the validator deems necessary to relay to any connected clients. The server also contains its own Client class that holds information on the server so database access is not necessary to filter who you send messages to.

7.2 Partially Complete Functionality

Some features were partially implemented due to the shortage in time and scale of our original concept. As such it has been pushed to future milestones.

7.2.1 Combat System

The combat system is partially implemented. The fundamentals of the system like the attributes of entities, taking damage, dealing damage and even a spell and weapon are complete. There are a few key features missing. The first of which being the class based skill system. Players are to choose a class or role that gives them access to gear and spells that are exclusive to that class. This was partially implemented in a test branch but as the end of the project approached, the mark was missed on implementing it into the demo. Furthermore, status effects have not been implemented either, as these two features go hand in hand since spells are the primary source of status effects. Similarly, weapons are only partially implemented as only 1 type exists. Each class should have its own type of weapon that they primarily use.

7.2.2 Item System

Items are a very important part of this game and were partially implemented in this phase of the project. Tables in the database were set up for it, an inheritance structure of classes was designed for it, but the item menu or inventory was only partially complete on a test branch. Implementing the rest of the item system without a GUI for it was deemed unnecessary and pushed to future milestones. So the partial implementation in the demo has just a single item, the default sword.

7.2.3 User Interfaces and Menus

Creating menus in XNA was found to be a lot more challenging than was first assumed which is one of the major causes for this only being partially complete. Completed is a main menu, options menu, stat menu, dialog menu. Partially completed is the quest menu, which is just

simple text with only quest requirements and name on the right hand side of the screen, and the character creation screen, which is only a name picking screen instead of a style changer. The item menu and skills menu both did not achieve any implementation in the demo.

7.3 Planned Functionality

Some features were expected to be implemented, but due to the lack of time, these features were pushed to future milestones. Additionally, some features missed in requirements elicitation were later identified and as such added here to the planned functionality.

7.3.1 Item System + Inventory Menu + Character Creation

The item system is integral to the game as items and currency are what add value to your account and makes it worth it to keep playing. While you can add additional items to the database now, there is no way to see what items you have or way to use them. As such, an inventory menu needs to be created where items can be viewed, used, and equipped. This goes hand in hand with the character creation menu as making your character look like they are wearing the armor you equip will most likely use the same or similar code. Afterwards, quests can be tweaked to reward players with equipment as well, instead of just experience.

7.3.2 Combat System + Spell Menu + Classes + Action Bars

The combat system is planned to be completed and expanded to increase the satisfaction a user gets from playing the game. Spells can add a huge variety of effects to the game like teleporting, dashing, going invisible, shooting fire balls etc. which can all feel super satisfying to do. Moreover, a menu to choose and level up those spells would need to be implemented. The

spells you can choose, and ultimately how it will feel to play the game, will be dependent on the classes as described at the beginning of section 3.1. The classes described, mercenary, ranger, and sorcerer, all bring their own options for spells. The difference in classes will let players pick a fight style that suits them, whether it be a healer who dies fast but supports their allies or a sneaky rogue who can do large bursts of damage. Class selection, and development, should all be implemented through the story line, so each class gets a truly unique experience from one another. This will draw players to replay the game as another class if they get bored of their own class. Moreover, in order to use the spells they learn, they will have to assign it to a hotkey. This is done in World of Warcraft through action bars that you can place your spells on. Each slot on the action bar can be bound to any key the user likes and when that key is pressed it will cast that spell, or consume the item in that slot. Additionally, status effects applied to entities would be implemented in this combat system. These status effects can be boosts to your player like speed, or damage over time effects like poisoning an enemy.

7.3.3 World Map + Quest Menu

This requirement was missed during the initial requirements elicitation and was later found when one of our play testers asked about a map. This is a core feature of every MMORPG, otherwise discovering a massive online world can be difficult and players would get lost. Additionally, the map can show the user where they can complete the next objective in their quests.

7.3.4 NPC shops + banks + crafting + consumables

This ties into the item system, and could only be implemented once the item system is in place. NPC’s should be able to buy and sell items with the player. NPC’s would typically sell common crafting supplies that the player can use to craft things or consumable items like food. This could be as simple as an NPC who sells empty vials. A player would need to purchase this, as well as

collect mushrooms dropped by some monster and combine them to create a potion. The crafting system would primarily be used for crafting consumable items. Moreover, with all the items in the game, the player would require a place to store their belongings to make room in their inventory to get more. An NPC bank would allow for safekeeping of items within the safety of the university campus or otherwise.

7.3.5 Chat System + Friends + Party System + Dungeons

The classes discussed previously are designed specifically to promote teamwork. One class's flaws may be accommodated by another's strengths. As such a proper system for interacting with players in the game and making allies should be in place. This would be a simple menu that lets you add users to your list of friends that can be private messaged at any time. Additionally a local area chat that lets you talk to anyone nearby can help you find friends to team up with in the game. To properly team up, experience and loot gained from killing monsters should be distributed amongst the players you are teaming with. Therefore, a group or party system that allows players to share loot, experience and quest progress should be put into place. Currently, anybody who has the same quest as you will advance your quest progress as they advance theirs. This is because no proper grouping system was implemented but we still wanted our users to combine their efforts. Most importantly, creating a large party of 5 people would allow the players to take down much more difficult enemies. A dungeon system containing powerful enemies and lots of loot can be made only accessible to full parties of 5 people. In the dungeon a player would typically kill through a bunch of monsters and 1 big boss at the end with the assistance of their allies.

7.3.6 Player Trading + Player Dueling

A much later planned feature that would require both the item system and spell system to be implemented to be relevant is player interaction. Players should be able to trade each other their items and currency to further encourage playing with friends. Additionally, our play testers were asking if they could fight each other and thought it would be fun. This can be implemented with a dueling type system, where one player can challenge another and they fight until someone loses and is left at 1hp, or a battle arena where it is a free for all with no death penalties, or both.

7.3.7 Animation upgrades + music + art upgrades

Another planned feature was to upgrade the animations of the characters. Right now, the only animations are simple walking animations, but it was planned to expand this so casting spells, and swinging your weapon would get better animations. Lastly, no atmosphere is complete without art to set the mood and music to set the tone. The goal with the art and music is that every unique sector of the world will have unique music. Moreover, some battle music can add a lot of intensity to the game.

7.4 Final Reflection

While the project did not fully reach completion, it has enough of the groundwork laid such that an up and coming business or group of developers like ourselves could continue development and deliver a finished product. Our early playtesters really enjoyed the physics and feel of the game and gave us feedback on where to improve. For a game that is still sprouting it’s roots, the feedback was highly positive. If anything, this feedback proves that Battle University could one day break into the lucrative gaming industry.

References

[1] Laboratory Health and Safety Manual,

http://www.sce.carleton.ca/courses/health-and-safety.pdf

[2] Video game sales by country | Statista.

https://www.statista.com/statistics/308454/gaming-revenue-countries/

[3] World of Warcraft Leads Industry With Nearly $10 Billion In Revenue - GameRevolution,

https://www.gamerevolution.com/features/13510-world-of-warcraft-leads-industry-with-nearly-10-billion-in-revenue

[4] Distributed Vs. Centralized Architectures,

http://www.timeoutofmind.com/pdf/Central_vs._Distributed_Systems.pdf

[5] Traffic Analysis and Modeling for World of Warcraft,

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.1756&rep=rep1&type=pdf

[6] MySQL Support and Licensing,

https://www.oreilly.com/library/view/mysql-reference-manual/0596002653/ch01s04.html

Appendix A: Crash course for developers

Setting up the project:

-install visual studio from https://visualstudio.microsoft.com/

-install xna into visual studio using this guide :http://flatredball.com/visual-studio-2019-xna-setup/

-open your project solution file (.sln) in visual studio

How to use XNA and add features to our client side:

When you run an XNA application, it is similar to running an infinite while loop. Before the loop begins an Initialize function is called. Here is where any initial values should be set. In this project, the read and write streams are initialized as well as the camera and empty map objects. Next it calls load content, here is where you will want to initialize any objects that load content from the game directory using Content.Load<T>(fileName). This can be done later, but if you already have menu textures or button textures that can be loaded now you should. Now the game will enter its main loop. You can imagine this loop as

```
while(true)
{
        update(gameTime);
         draw(gameTime);
}
```

This loop runs up to 60 times per second, so if you are checking user inputs some form of debouncing should be implemented. In this project, the update and draw methods operate on a switch statement dependent on an enum we made called CurrentGameState. The update method is used to check input, or do some other computation or editing on the values of the objects in the project. To get mouse information you can use Mouse.GetState() which returns a

MouseState object which has useful attributes like its position and clicked state. Similarly, keyboard state can be obtained using Keyboard.GetState(), which we commonly use to check if a key is down using Keyboard.GetState().IsKeyDown(Keys.WhicheverKeyYouWantToCheck). While the draw method is for drawing your objects onto the canvas. The drawing of objects requires a SpriteBatch object, which is essentially a collection of settings for how you will draw the sprites. If you do not have any crazy drawing requirements, a single spritebatch with the default settings will usually suffice. Additionally to draw, you need an object called a rectangle and a Texture2D. A rectangle has a Position vector and a width and height. A Texture2D is the image file you load from the game Content. In our project, the Sprite class we made contains these two things as attributes. Our sprite class also contains its own update and draw methods, that are called from the games update and draw method. Our entity class inherits from the sprite class and adds functionality for movement, animation and simple collision detection. Simple collision detection just checks the rectangle of two sprites, and uses the .Intersects() method to see if the rectangles are overlapping. To implement movement that doesn't feel jittery we use an additional vector for velocity. Velocity is simply added to the Position every game tick in the Entity update method. This should be enough to get a good understanding of the client and how you can add functionality. Its worth noting that all menus must be drawn the same way as entities. However, you can also draw individual strings if you want to display text. So a menu like the dialog menu is implemented by first drawing the background box of the menu, then strings or buttons or other sprites on top of it.

How the server and multiplayer works:

Multiplayer is implemented by receiving the information about entities asynchronously through a TCP data received event that then runs the method through a ProcessMessage method. The

process message method is essentially the implementation of the Server-Client protocol defined in table 5-1. The server operates similarly where messages are received asynchronously through events and the message is processed through the validator. Unless the message falls into a case where the message is not to be relayed, the client who created the message has their ID appended to it and the message is relayed to all relevant clients. Thus when a client sends the server a message that it moved, the message is passed on to all relevant clients. To send and receive data through the TCPClient it is simplest to use a memory stream. The BinaryReader and BinaryWriter methods allow you to decide types (int, string, single, boolean, byte, etc..) to write directly onto the memory stream or read directly off of. This makes it easier to read out the same type you wrote onto a stream instead of trying to piece together the byte[] the tcp client sends. This is best described with an example:

This is how the MapJoined command works

Sender:

```
writeStream.Position = 0;
writer.Write((byte)Protocol.MapJoined);//Indicator bit
writer.Write(Convert.ToInt32(Player.Position.X));
writer.Write(Convert.ToInt32(Player.Position.Y));
writer.Write(Player.mapName);
writer.Write(Player.name);
SendData(GetDataFromMemoryStream(writeStream));
//The method GetDataFromMemoryStream converts the write stream to a byte[] for sending
//over TCP
```

//see next page for reader

Reader:

```
            readStream.Position = 0;
            readStream.Write(data,0,data.Length);
            readStream.Position = 0;
           p = (Protocol)reader.ReadByte();//Here the first byte is read to determine type
        If (p == Protocol.MapJoined)
        {
            int x = reader.ReadInt32();
           int y = reader.ReadInt32();
           string map = reader.ReadString();
           string name = reader.ReadString();
           byte id = reader.ReadByte();
              //add the new user to the local list of players
        }
```

As you can see the values can then be read out of the reader in the same order they were added. Please note, our project does most of the steps for you already, meaning all you have to really do is add your indicator byte to the Protocol enum, and then cover the case where the protocol is of that type in the Clients ProcessMessage method, and the servers Validator class.

If you're going to be doing work on the database, we use MySQL Workbench which lets you write queries or use the GUI as you please. It may be overwhelming at first, but once you dig into the code it will all make sense

Appendix B: Weekly Meeting Notes

First Meeting

September 9, 2019 9:51 AM

-it is indeed our intellectual property
-everyone must work equally
-Franks is a resource, and we can always ask him for help.
-10 pages in 4 weeks

-final report, if late, you fail. 100 pages double spaced non boiler plate stuff. roughly 25 pages each.
- -requirements(fall)
- -design(fall)
- -implementation(beginning of winter)
- -testing
- -reflection
- -bugs
- -whats great
- -what sucks

-Poster fair where something should be working. No more code afterwards.
- -oral presentation in january, a pitch for the program. roughly an hour (10-15 minutes).

This week: 1 page project description: a breif explanation of what we are looking to acheive
This month: 10 pages on requirements analysis. Division of work based on time. Requirements, functional and nonfunctional.
- -4-5 milestones: that are realistic and functional, with smaller individual parts divided by person

Second meeting

September 19, 2019 4:18 PM

-Justify why we chose C# and XNA.
-Start with a simple storyline
-requirements cap will also be big part of design.

in 3 weeks:

- get a basic storyline
- flush out the requirements
- administrative users
- use cases with NPC's
- think about back end (database)
- graphics
- get all 4 of us playing at poster fair
- pert chart for when things will be done
- remember to take notes at our meetings
- need to break down the perc chart ASAP

Milestone 1 should be storyline

- rough
- implementation design

Milestone 2 platforming and enemy basics
-basic movement
-basic platforms with different textures
-

Third Meeting

September 26, 2019 4:03 PM

-Look over the graduate attributes, particularly 2-4 for the report, and 6-7 for the project in general
-relevant publications: references for XNA, maybe OOP references. Scholarly articles are the best
-brainstorming the milestones and whos gunna do what
-need to setup some agile support
-database, graphics, actors, see how the parts fit together to map the individual responsibilities.
-need to make that timetable and put it together
-get a clear description of the game going
-always give reasons for decisions

By next week, finish the timetable/functional blocks/milestone document
Roughly 5 milestones, last milestone being poster fair

December, half the report written

4th meeeting

October 3, 2019 4:07 PM

-Skip side quests, story board the main quest then decide to add them after if time permits.
-Complete the requirements analysis and see what requirement come from the story.
-you can break the game up into beginning, middle and end.
-5 minute demo by april is the main goal, everything else just adds depth.
-NPC(Non-Player Character) : make sure all acronyms are recorded.
-get the story line written down for next week. Just basic, driving requirements.
-think about test cases and an automated user to test the game.
-showing how testing is done.
-oral presentation and poster fair are like sales pitches..
-poster on what kind of process we followed.

For Next Week:

- main plot
- storyline written down
- how its going to interface with the game
- identify major NPC's, objects, enemies, sketch the map.
- 3 drawings

5th meeting

October 10, 2019 4:04 PM

RENAME MILESTONE GOOGLE DOC AND COMPILE ALL DOCS INTO ONE DOCUMENT
-Ryan, You've been assigned testing by greg franks lol
-Connect multiplayer for 1 person on 1 PC at first then expand
-Architecture we want to follow, do some modeling of classes
-Use cases for multiplayer ETC.
-Instructions on how to play
-Perhaps add some Admin users with cheats
-Critical sections should have minimal computation
    -Majority of overlap will be between two players and players and the world.
        -Trading between two players
        -Fighting the same monster
        -Fighting any monster
-Definitely research the possible economic value of the game
    -Similar things on the market/ how it compares / price points
    -Oral presentation selling point: Like dragons den (1 hour)
Next Week Deliverables:
    -Doc with everything in it (atleast 10 pages by next week)
    -JPGs of sketches in the document
    -Breakdown of whos going to do what
Further Deliverables:
    -Big blobs and class diagrams
    -Expand requirements
    -prioritization

Final Report:
    -Chapter 1: Introduction
        -Basically the proposal/Abstract
    -Chapter 2: Boilerplate
        -health and safety... etc.
    -Chapter 3: Requirements Analysis
        -What were working on right now
        -Include requirements elecitation
    -Chapter 4: Design
        -Architecture
        -Class models
        -use cases
    -Chapter 5: Implementation
        -Milestones
    -Chapter 6: Testing
        -unit tests
        -Integration testing
            -how it was done
        -Acceptance testing
        -What the test cases accomplished.
    -Chapter 7: Conclusions
        -What did we do
        -What sucks/What rocks (reflection) ***This is important
        -Possible future development

6th Meeting

October 17, 2019 4:10 PM

We need to get the requirements written and finalized.

-distributed architecture has sync downsides
-centralized has latency downsides
-spread out as much as possible on the map (make it big)

-any design will have downsides, make sure the issue is known, when the issue is addressed is another story

Define subsytems.
References and citations
-MMO-RPG background info for chapter 1.
-3 or 4 IEEE references or scholarly references or books
-Research how Maplestory and/or world of warcraft AND terraria do multiplayer
-look for technical documents for MMO's
-Need XNA documentation
-100% we're talking about the mechanics were stealing from games cite the game itself if no documentation exists
Lets get some architecture going
Get Manel to scan in the images.
REMEMBER YOUR LOOKING STUFF UP FOR A REASON, AND ITS NOT FOR THE REFERENCES QA

Two weeks:
- -Architecture (Kyle + Ryan)
- -Requirements (Ryan + Kyle)
- -Sketches (Manel)
- -Finishing touches on story (Nnamdi)

7th meeting

November 1, 2019 2:07 PM

Scalability we got wrong. Its how many people can play at once. How many enemies, etc.
Expandability is what we described. replace scalability with expandability and add scalability as defined above.

Response time- whats the latency. make this a sub category under performance
Add to non-functional requirements the actual measurements and how we display it in game.
The minimum acceptable response time is roughly 150. Research for response times to quote that number. get measurements from the game.

Reliability-Game should be up most of the time. Talk about why its important. Fault in one part should not propagate into a system failure.

Story board - get one going

use cases move to requirements. Use cases for anything you think you need to test.
-Login use case
-Transaction use case
-Two players kill the prof use case
-Death use case
-future designer adds level possibly.

Application diagram, server client diagram.

Possibly a battle process that makes shared resources scale down to just those who need it. Try and make interactions as minimal as possible. Stick the story in functional requirements.

scalability is our big non functional requirement, and should drive the architecture

careful with shared variable

cache large unchanging resources

Next week:
-Architecture
-get ideal ping references
-keep coding

8th meeting

November 7, 2019 4:01 PM

-Distribute database to improve scalability and remove chokepoints
-set up architecture such that multiple server nodes can be run (distributed)
-who should i talk to? get returned the address
-scaling is a major non functional requirement
-we can have it on 1 node as long as we show its supposed to be able to be on multiple.

-December 6th is the deadline to post availability for the oral presentation
-progress report is not due

Next week:
-Mad coding now
-Try and show how the work will be divide
-try and finish milestone 2

9th meeting

November 28, 2019 3:56 PM

1-page progress report
-How we are right now
-How we are moving forward
-What planning will take place in the coming term
-summary/reflection

Oral presentation:
-All four must be there
-Each person allocated 15 mins ~ 10 minutes talking each then questions
-Motivation for doing it (why, outcome) 1.
-Put in report/pull from report
-Talk about the project itself (scalability, flexibility, reliabiity, security)
-Talk about the project design (talk about architecture, why we did it, why we choose what we've chosen)
-Slides (no code, no paragraphs, no reading, add pictures/diagrams, 5-10 slides each, if were going to do a demo, film it, story board slides, map slides)
-3rd week of january

-submit 1 page progress report
-1-page progress report
-How we are right now
-How we are moving forward
-What planning will take place in the coming term
-summary/reflection
-unit tests/ integration testing / acceptance testing

10th meeting

January 13, 2020 2:28 PM

Oral presentation:
-All four must be there
-Each person allocated 15 mins ~ 10 minutes talking each then questions
-Motivation for doing it (why, outcome) 1.
-Put in report/pull from report
-Talk about the project itself (scalability, flexibility, reliability, security)
-Talk about the project design (talk about architecture, why we did it, why we choose what we've chosen)
-Slides (no code, no paragraphs, no reading, add pictures/diagrams, 5-10 slides each, if were going to do a demo, film it, story board slides, map slides)
-3rd week of january

-Motivation why
-Inspiration for game and story
-Design Decisions

-Talk about game itself
-Talk about where we are
-Non functionals and how we will address them

roughly 20 slides without images
images are very good tho
5-7 bullets max
no code
Perhaps a short two minute demo

Put together slide deck and share with franks

11th meeting

January 20, 2020 2:27 PM

Oral presentation:
-10 slides each
-Presenter notes
-Might as well say were using tickets
-get numbers for performance

in depth research for scalability for response time
a few references 6-8
    -justifying architecture
    -conquer through partitioning

-Add functional requirements

Why are we doing this
Why did we settle for these options

1 slide about the game market

Talk about the tickets

Last slide is milestones, outstanding, and questions

usb key with pptx is best option

why why how

non-functional questions

Meeting 13

February 24, 2020 2:31 PM

- ASK for network switch and 5 blue cables
- following their format we should be good
- demo-level is good (reset to beginning of quest)
- Poster
  - Motivation
  - instructions
  - Scalability / performance
  - Story
  - catchy images
  - a little testing

next week short demo

Appendix C: Rough Work (Not to be graded)

STORY CONCEPT

CONCEPT 1

The story of the game will be a mash of different stories from different popular games. As the game will consist of different sub-maps/stages/floors, each stage would be influenced by a different game. A case could be a stage modelled after the game Super Mario. In the stage we would include Easter eggs from the original Super Mario like at the end of the stage, an NPC would tell the player that their ‘princess’ (or whatever the aim of the stage was) is at a different castle. This line is a line that Super Mario is known for and thus could serve as an Easter egg for

the stage. For this concept to work a set number of stages would need to be defined to put a limit on the number of different stories that would be generated from different games. This merge of different games will be due to a ‘collision’ between dimensions that would see our

players travelling between worlds in order to set things right.

CONCEPT 2

The story of the game would be a parody of a student's life at university. Players would play as students starting off in their first year as vagrants and would battle through weeks, semesters and years till they reach the top as a Graduate. This concept could be explored with great depth and different twists. A case could be the student finishes the first year as a vagrant but at the second year phase, the player is ready to pick a class and move forward with it. Enemies could be 'gods' of tests, exams, courses, student pressures like anxiety, social awkwardness, depression etc. With a final boss at each year being an all powerful professor.

Project Milestones

MAJOR REQUIREMENTS BUCKET:

-options menu

-mechanics (movement, attacking)

-Inventory/Item System

-Bank System

-Quest System

-NPC's with dialog

-Equipment System

-Crafting system

-Gathering / Collecting crafting materials

-Spell/Class System

-World Map

-Individual maps / zone content

-Monsters in the world

-Storyline (main quest)

-Additional content`

-Multiplayer **

    -client server setup

    -messaging between client and server

    -database

Milestone 0: The storyline (13 days)

-Storyline (main quest) / side quests (design)

-World Map (design)

-Individual Map/zone content (design ideas/background images)

-gather additional requirements from this phase

Milestone 1: The Fundamentals (34 days)

-Options menu

-Basic Mechanics (Movement, Attacking, Stats)

-Basic platforms with collision detection and textures

-Basic Enemy Slime

-Background music and image

-Basic spell

-Basic action bar

Milestone 2: Multiplayer / Items (55 days)

-account creation

-character creation

-Items System / Money System /Crafting System

-Bank Storage setup

-Server Database setup

-Server inventory tables

-Messaging between client and server

-Client Inventory implementation

-Client item sprites

-Mobs loaded from server

-Mobs drop items

-Idle NPC

Milestone 3: Flushing Out Gameplay(21 days)

-Implementation of world map and individual maps

-NPC with dialog

-NPC with shop system (giving/receiving items/money)

-Bank NPC for storage

-Additional monsters on individual maps

-Equipment system

-Spells/Class System

-Gathering / Collecting locations

Milestone 4: Quest Overhaul (21 days)

-Most NPC’s should be on the map

-Main storyline implemented into NPC’s

-Quest log interface for viewing quests

-Party system, sharing exp on mobs, and progress for quests

Milestone 5: Music Overhaul + Optimization (21 days)

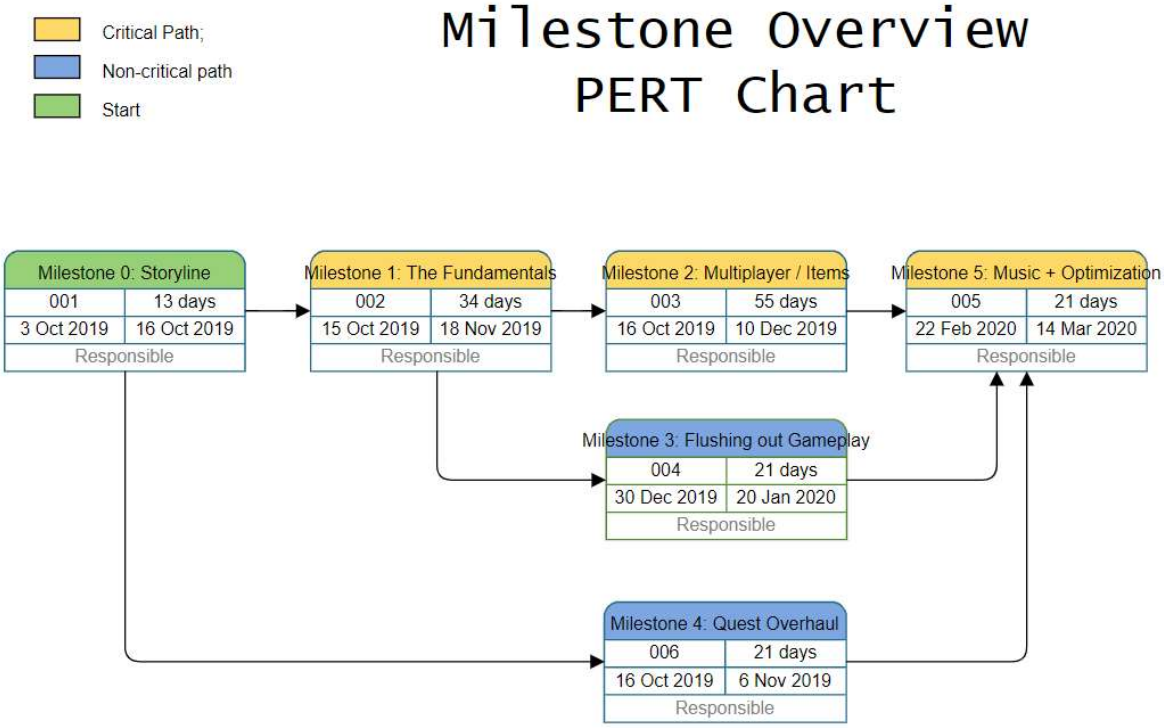
-Zone music (Zone based, in or out of combat)

-Mob idle sounds,

-Improve the balance of classes, mobs, and items

-create test suites / fail safes to make future development easier and running simple

-tweak performance to make multiplayer as seamless as possible



STORY

TITLE: Battle University

BEGINNING

A vagrant's application to a university (Carleton university?) is accepted and he arrives at the "registrar's office'. He's given the rundown of the school. Carleton University is a 4-year university program to turn the strongest and most promising vagrants into warlords ready for the real world.

1st Quest – The registrar's office attendant tasks the student to select a weapon and base armor.

- The vagrant is then shown how to move and attack with a weapon. A practice dummy is provided for the vagrant to get accustomed to the system.
- The vagrant has completed the quest after returning back to the registrar's office attendant with a weapon and armor and is rewarded with the weapon and armor he equipped.

2nd Quest – Vagrant uses his student ID (playerID#?) to move on to the next stage where he meets a Professor. The first course will be on fighting villains and basic skill moves (weapon based). Task will be to defeat 20 villains(slimes?). Quest is completed when vagrant returns after killing 20 villains. Vagrant is rewarded with a better weapon and armor?

2ND YEAR

3rd Quest - (if crafting system is to be implemented) – Player arrives at his next class (next map location?) and fights through some minor villains to get into the class where he finds a mad scientist that is a professor. The professor gives the vagrant some health potions (if mana bar or something like that is available then mana potions are given as well). The professor tasks the player with gathering 10 slime

drops (depends on enemy type). 1st half of task is completed when player returns to the professor with the ingredients. The professor explains the crafting system to the player and tasks the player with crafting health potions. Task is completed when student crafts the health potions. Player is rewarded with more potions and ingredients and a special potion that cannot be used until quest 4.

4th Quest – Player level has reached level 10 and is now able to select a class to develop as. Mad scientist professor tells the player to consume the special potion he was given in quest 3. After consumption the player is able to choose a class to develop as. After class is chosen student is tasked with going to speak with a professor in charge of the department of the class he has chosen to develop as. Player is rewarded with low tier class skills and access to the class system?

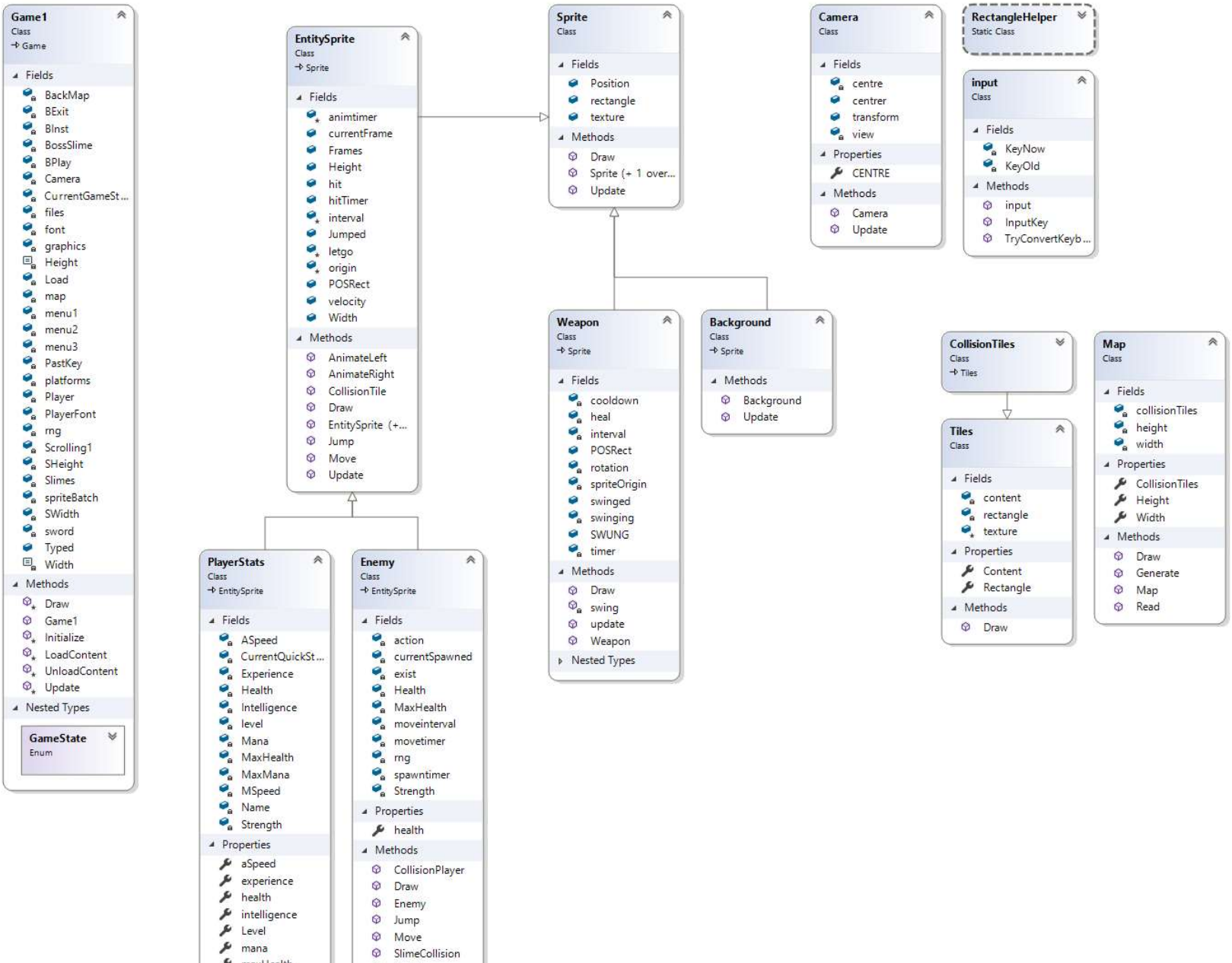
3RD YEAR

5th Quest – Player meets the university counselor and the counselor explains the (mini?) bosses of the 3rd year to the player. These (mini?) bosses are Depression, Anxiety and Stress. The Players is tasked with overcoming these (mini?) bosses by defeating them in order to unlock special skills to be able to defeat professors. Task is completed after Player has defeated these (mini?) bosses. Player is rewarded with access to a special skill line?

6th Quest – Player moves on to the location of boss professor A. Player's task is to pass the professors exam and defeat him in battle. Task is completed when professor A has been defeated. Player is given access to high tier locations on the map.

4th YEAR

7th Quest (FINAL MAIN STORY QUEST) – Player has passed boss Professor exam and is now ready to take on the final project quest of battle university to graduate as a battle-ready player. Boss professor gives player a list of professors to defeat in battle. Quest is completed when all 5 professors on the list have been defeated. The list of professors to be defeated are professor x, professor y, professor z, counselor from 5th quest and finally the undergraduate chair and his fearsome snowboard. Upon completion, player is awarded with a graduation armor set and a



weapon relative to player’s class. Then the player is granted access into new areas in the map for post-completion gameplay.

Progress report:

Currently the project is on track and on the critical path. Rather than referring to the original proposal, we will use our milestones as a marker of progress. For reference, I have added a PERT chart of our milestones. We set out to have the story-line complete by October 16th which was easily met. We then began the construction of our client, with the fundamental functionality for milestone 1. This was completed before the expected date, and went through a series of acceptance tests to make sure we have been hitting our design goals of scalability and reliability. Currently, we are in the process of implementing milestone 2, the multiplayer and item update. This is taking longer than expected due to the complicated fashion that is multiplayer game design. We've continually been researching the best approaches and looking at similar games implementations. We determined the best way moving forward was to break apart functionalities into groups, and assign tickets (or tasks) to each member for one group of functionality. We have only broken apart milestone 2 into tickets until we have the multiplayer working, and will further break down the milestones into tickets once multiplayer is working on the fundamentals of the game. In summary, we completed our fundamentals which include the player, basic enemies, a weapon, a spell, and all the factors that make the world (tiles, maps, backgrounds, and collision detection). On our current route we should reach completion of all milestones, but if necessary we will move the music and the extent of our gameplay flush out (size of world/ number of entities, etc..) into future development. Overall, we are doing well, participating evenly, and being proactive with research and testing while still hitting our deadlines.

Please see the next page for the PERT chart depicted our expected deadlines