Mirrored Fantasy

Andrew Bell

California State University Monterey Bay

CST 499 - Capstone

Jesse Cecil

10/6/2020

Table of Contents

Part 1

Introduction

Mirrored Fantasy is a competitive moba style game where teams are separated in their own lane where each team must complete three dungeons. A player vs player dungeon where a team member must go through narrow corridors that is exposed to fire from the other team. A puzzle dungeon which the team must solve with light exposure to the other team. Finally a player vs environment dungeon with waves of enemies and a boss fight to complete. Each player will have their choice of magic based abilities ranging from a standard fireball to seeds that grow into plant turrets. Each of these skills will have a direct counter from another magic type. Each type of magic has two different skills, a projectile based skill and a terrain based skill.

The Problem

The problem this project is solving is that the MOBA (multiplayer online battle arena) genre has grown stale with most games having the same formula of three lanes and a jungle where you just push towers till you destroy the other team's base.

The Solution

The solution to this problem that this project is going to implement is remove the standard map and put teams in isolated lanes where there is one focus on team interaction in the player vs player dungeon, and bring ability interaction and counters from popular RPGs such as

Pokemon’s type system (Pokemon Wiki) or Divinity Original Sin’s environmental effects (Divinity Wiki).

Goals and Objectives

Goals	**Objectives**
Balanced gameplay	Ability type system with the base of a rock paper scissors balance (similar to Pokemon's type system) where each type has one that is good and bad against
Interesting map with mirrored lanes	A map with 3 dungeons with different objectives in each of the 2 mirrored lanes
Goal for the game is to be the team to finish all 3 dungeons first and put their objects on the victory monument.	A player vs player dungeon where the person running is very exposed to attacks from the other team
	A player vs environment dungeon where the players have to combat, most likely will be similar to a bullet hell or a raid boss style boss monster.
	A puzzle dungeon where the player must solve a basic puzzle to get the objective.
An easily extensible 2d Javascript game engine	A Vec2 class for holding all of the vector and point information
	A Map class that holds the map information, and the entities that are on the map
	A Collision engine that calculates when something collides with another object and calls the entities hit.
	A entity that holds all of the information about an entity (name, health, speed, position,

	look direction, hitbox, image source)
	A lot of other classes that build off of these
Server that feels quick and responsive.	NodeJS server running express to serve the client code and run the game server. This also encourages code reuse between the client and the server

Community and Stakeholders

The stakeholders for this project would be the game developers. The amount of time and resources that will be used to create this game would determine how successful and how many features that would be included in this game. Further interest in this project can result in adding additional features as well as more content such as more maps and puzzles. There may also be more different game modes that use this game engine.

The community that would be affected by this project would be the players of this game and possibly community developers that plan to expand upon this project. Having a smooth and interesting game would help with the game's popularity and willingness of others to play it. Since this game touches upon the multiple aspects of competitive multiplayer while dealing with several obstacles and puzzles, this should appeal to multiple audiences. The popularity of the game will create a major factor of potential expansions of the game or even ideas for future games. Some of the possibilities includes player made maps, new game modes with new abilities and puzzles.

This game's overall structure and mechanics is based off of Minecraft's Race for Wool (SkiTrip), where there are two mirrored lanes that will have teams race to be the first to complete three objectives and bring back the objective completion reward to the victory monument.

This project changes the formula by changing from a first person perspective to a top down perspective, where I will be taking the control scheme from Battlerite which "is an action-packed Team Arena Brawler focused on competitive PvP combat" (Battlerite) which has a standard MOBA (Massive Online Battle Arena) perspective of top down, except it has movement on "WASD" instead of mouse movement. I am going to have left click be a basic melee attack, right click be a bow charge style ranged attack, "Q" be a ranged projectile as a damaging ability based on one of the three types, and "E" a terrain placing ability again based on one of the types.

The project is going to have a rock paper scissors balanced in its abilities, with fire, water, and grass, which unintentionally lines up with the 3 main types of Pokemon, and like Pokemon I would like to expand types for abilities in the long term (Pokemon Wiki). However, these types will only have two options; a projectile that generally does damage and a secondary effect on hit and a terrain placement.

Evidence the Project is Needed

The need for this project is building a crossover between a lot of the major genres in gaming while also shaking up the standards in each of their genres. I am building on top of a MOBA base in the idea of a battle arena but I am doing away with the standard lanes and champion based character selection, and building in RPG style abilities with the choices of which abilities you want to have at any point in the game and switching between them on the fly as needed. The game is very objective based rather than the MOBA tower defense gameplay.

Feasibility Discussion

The feasibility of the project comes from the constant growth in eSports and gaming. Every year gaming becomes more and more of a profitable form of entertainment, recently becoming the most profitable form of entertainment. (D'Argenio, 2018). The closest to this project that currently exists is a Minecraft game mode called Race for Wool and was the basis of the game, I am adding abilities and having it be a 2d game rather than the 3d from Minecraft.

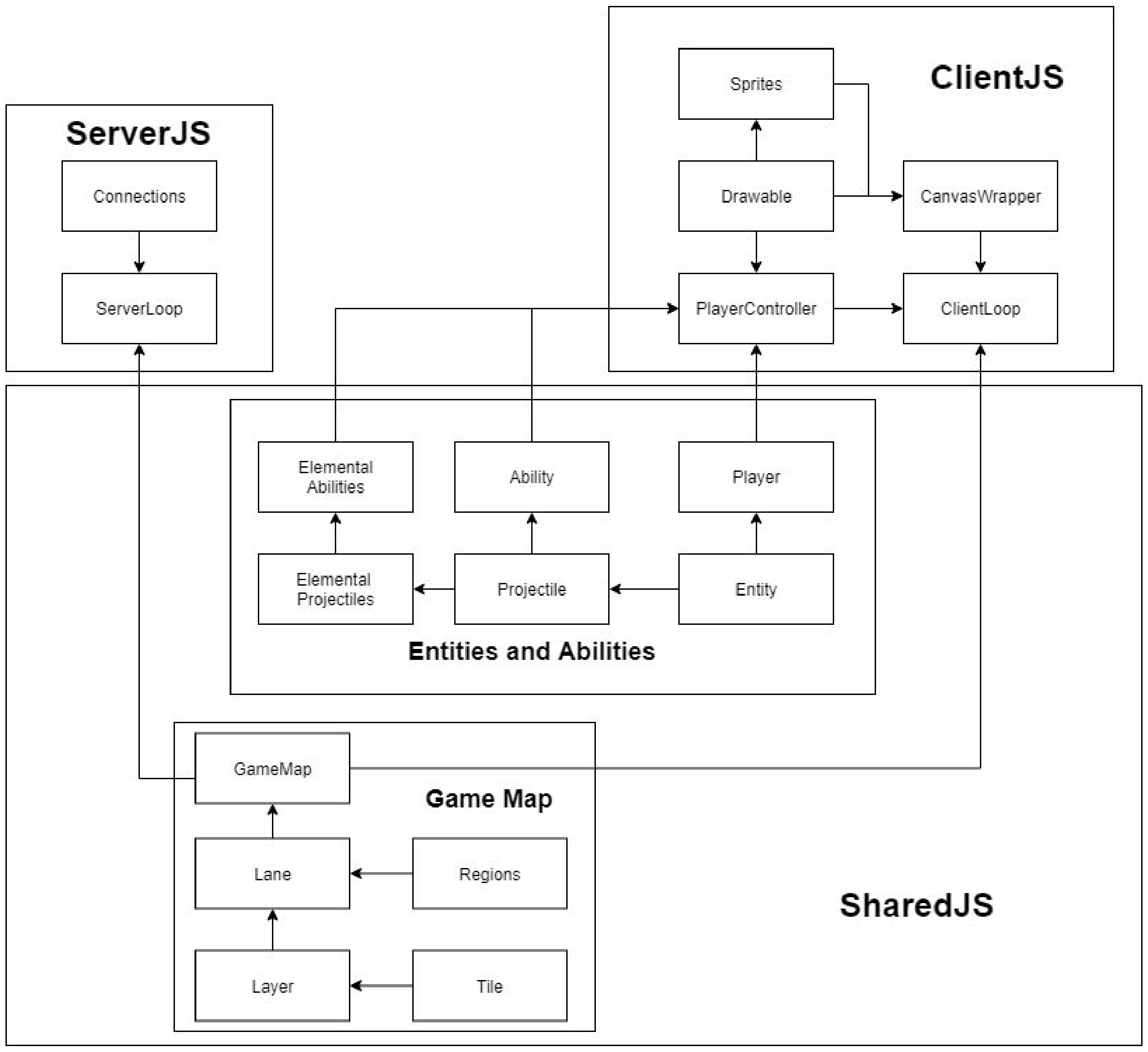
Other than that there are similarities with other games that I am also borrowing concepts from. One of which being Battlerite's control scheme being WASD for movement and Q E R for abilities, which is becoming a more common control as opposed to League of Legends click for movement. I also would like to have abilities with actual interaction so that hitting a plant with a fireball will cause a bond fire and actually lingering on the map for awhile, this is inspired from the concept of Divinity Original Sin's environmental effects. Having these ability interactions will build more skill based depth to the game giving more replayability and allow players to show off how much they have learned about the game by swapping abilities at the right time.

The design approach of going 2d with the game is coming from speed limitations of wanting to make an easy to pick up browser game that will run on almost any device. Two dimensions also lowers development time as there are less optimizations that need to be made with one less dimension and a lot easier to develop for. This also brings less requirements for art and 2d game assets are a lot easier to find for free on the internet through sites such as opengameart and itch.io further lowering development costs. This project is mostly going to fill the space as a proof of concept that can be expanded and built upon as this project will remain open source with the map editor available and easy map selection at the start of each round to encourage the community to get involved when available.

The three dungeon design is to be as accessible and inviting to different types of players as there will be something for someone who wants to focus on player versus player combat and lockdowns of objectives, as well as player versus environment sections to build something for players that enjoy focusing on learning raid bosses and getting faster and more skilled against AI opponents.

Part 2

Functional Decomposition



A Basic Diagram of Class interactions

Major functions of the game start with the server and client loops, these are similar but have slightly different functions in them such as the client loop handles drawing to the screen while the server has functions focused on relaying information to the clients. This relaying of information runs through the Connections class, this is used as a wrapper around the socket.io library and holds all of the clients connected to the server, and references to the game map, collision engine and server loop. When a client connects to the server it gets added to a lane in the game map and to the collision engine's players Map. Then the new client's player gets relayed to the other clients which do similar things when they get the message as well as add it as a drawable to the canvas wrapper.

The player controller handles all of the player input. This leads it to handling using abilities and moving the player. The use of abilities gets handled with the ability class and subclasses for each elemental ability. The ability class checks if the ability is off cooldown and then uses the player and projectile that is about to spawn's hit box to determine the offset of the ability and then the player's look direction to set the direction that the projectile is about to go in. The player controller then sends that spawned projectile to the server which relays the new projectile to the other clients.

The next major section is the Game Map class and all of the other classes that make up the game map. The game map holds 2 lanes one for the right and one for the left in a Map so that if someone wanted to make a map with more than 2 lanes it could easily be done. Each lane holds an array of the layers that make up each of the lanes and has regions, one for the lane itself and then an Map for all of the other regions. The Region classes are to mark the different regions on the map for each dungeon. But have been subclassed for added functionality, such as the objective regions give the player the objective when they enter the region. Other regions such as the PVE Dungeon handle starting the PVE Dragon boss so that it only is looking for players when there is a player in the PVE Dungeon.

Lastly the Collision engine which is the main heart beat of the game. It has two Maps, one for players and another for other dynamic objects such as projectiles, breakable tiles and the Dragon. It also has two QuadTrees (basically a 2d binary search tree), one for the players and dynamic objects that gets reset every frame, and another for all of the static objects such as the map tiles. Last there is an Array of the regions that are on the game map. In the update function for this it resets the dynamic collision tree. Then checks if there are any players that are overlapping the regions, this generates the begin and end overlap events. It then checks for any collisions on the players and dynamics by querying both of the QuadTrees. If anything was hit it would then call the hit functions of those entities which the hit function handles any changes that need to be made from that collision. The update function then returns a delete list which is an array of any entities that need to be deleted to be handled by either the server or client loop.

Design Criterion

This project's goal is to be able to have the game run effectively on either a local host or a basic server while being simple enough for even older computers to be able to play the game. This game uses javascript in order to layout the map and perform the basic game mechanics while using NodeJS and ExpressJS as the server side portion where it will allow players to play on the same game.

Game readability was another main goal of the project. To achieve this I went with 16x16 pixel tiles for the ground scaling them up to 32x32 so that detail could be put into the visually important features that the player needs to be looking at such as using 32x32 pixel art for the projectile sprites drawing more attention to them rather than the ground. I also was trying to stick to contrasting colors for the assets, but working with only free assets I went with the best looking ones that I could find while still being mostly in line with contrasting art.

The next important aspect of the project was efficiency of algorithms, using JavaScript Maps and Sets for O(1) lookup everywhere that any lookup might be needed. The QuadTree data structure is the key feature that makes the collision even remotely possible as it takes the collision engine collision lookup from O(n^2) if I was checking every collidable object with each other option, down to O(nlogn).

A design requirement that came up mid development was the Intellisense integration from the TypeScript compiler and type definitions using JSDoc, as even though this is a JavaScript project having the type checks in the linter. This has saved a ton of time solving bugs that were originally popping up in the first weeks of development.

This project is also focused on being intuitive for people who are new to games and veteran gamers. This game clearly displays the objectives and the conditions for victory. Each room with the victory condition is straightforward to clear and should not pose too much of a challenge. Players can then try to optimize their method to clear the rooms with the victory conditions in order to finish at a faster time. What makes this game unique is that it introduces player interaction with other players in the form of PvP so that players can affect the other team as they try to complete their room. Players can either focus on clearing as fast as possible while trying to avoid opposing team's interactions, try to impede on the progress of the other team, or both.

Final Deliverables

NodeJS and ExpressJS server that runs the server side game loop as well as distributes the client side code. A full game engine, with consistent server client communication and 2d collision engine and map editor.

Approach/Methodology

The process of making this game, I first took a look at games with similar features that I can use as ideas to implement into this game so that it would be intuitive and easy for potential players to get into quickly. Then I took a deeper look at the different abilities that the players can use. These abilities should be able to counteract one another while balancing their effects and usages so that ideally there would not be one ability that is too strong.

Next step would be building the puzzles and obstacles that would be used in this game. There are 3 different obstacles that each team must clear in order to win the game. The first one would have heavy influence with the other team where they can interact with and interfere. The second with less team interaction still has some way of preventing the other team from completing. And finally the last one would be mostly free from opposing team's interference.

I then found images that would go with the player models, the abilities, and also the map resources in order to create the map.

Afterwards, I worked with the interactions with the different parts that the player model has with the map and abilities. This includes collision detection for the player models with the map walls and interactables that the player can work with to clear the objectives. The abilities that the players have should also affect the players. Some of the possible effects that should be considered are pushing, slowing, and freezing controls.

Finally, all these parts are put together in order to create the game. Heavy testing would be required in order to test for bugs and check for improvements to the game.

Ethical Considerations

One of the major concerns that involves video games is addiction. Addiction in video games mostly comes from either luck based mechanics or long term commitment in order to progress. Some of the luck based mechanics that are commonly used in larger games includes loot boxes and random loot drops. The other method of long term commitment would be like having a character saved onto an online server where progress is saved with long term goals of character growth and progression or having a ranking system where players compete to get the best score or ranking.

To mitigate the concerns about addiction, this game will focus on being a competitive game where it is easy for the players to pick up and play a quick game anytime without being concerned about dealing with progression, leaderboards, or loot boxes.

In terms of general accessibility this game is going to be an easy to run browser game so anyone with access to a computer should be able to play it. The goal is to make it light weight and well optimized while still being able to handle complex game mechanics and environmental interactions. This is part of the reason why the game is going to be 2d rather than 3d, as it greatly reduces the CPU load and gives greater flexibility when coming up with ability interactions that can be made.

As this is a game there will be lots of play testing needed so I will be keeping the privacy of our testers to a maximum and keep all feedback anonymous. Being unsure as to whether player testing falls under human testing I will look into the policy on human subjects in research to be safe.

Legal Considerations

The major legal considerations for this project will be getting art and sprites that will display the player and abilities, as well as map layout. Our solution would be to use sites such as itch.io and opengameart.org with their selection of free assets. Finding assets for the game would be simple and quick to find in order to fit the theme of the game while giving credit to the artist of those art assets. Any other assets that would be needed for this project would be custom made using tools such as Express, Node, and ES6 Javascript. The other minor detail that should be addressed is checking if the name of the project is similar to other games.

After going through the environmental survey, this project should be unique enough so that this game does not infringe on copyright.

Part 3

Usability Testing

When designing testing I was going with my original timeline which was to have the game functioning by 9/12/2020 (week 3 of the class), however that didn't go as planned as a few parts of the project ended up taking a lot longer than expected and I had only just gotten full game events working 10/3/2020 almost a month after originally planned. So when testing was run 10/6/2020 these game states were not very stable so I ended up focusing testing on two parts.

The first being the dungeons giving objectives and the Victory Monument taking away objectives triggering the end game state. For this I got together some people that didn't play too many games but were familiar with games. I hosted the game on Heroku, which ended up being really slow with the free tier, this pushed me to host the game on my Raspberry Pi after running these two test events. One of the first things that came up during this testing run was that assets were not getting fully loaded and the dragon ended up being rather shy and not showing up most of the time. This was something that had not happened during local testing as the load time of the site was fast enough to finish before the code had to load the dragon sprites.

The second part of testing was combat testing and the overall feel of movement in the game. For this testing I got a few people that had played games a lot, one of which being a game development teacher and game modder. Luckily no bugs were found during the combat testing as this was one of the first pieces of the project that was built; it was also one of the most tested and used parts. However there was good feedback with how the game feels to play. The biggest one was that the map needed more cover to hide behind during combat. This was something that I hadn’t really put much thought into but when brought up it made a ton of sense to give people a bit of a way to hide. Another bit of feedback was adding a respawn timer which is definitely something that I will be adding into the game to make deaths more punishing.

Outside of player testing I also wrote unit tests on a few of the major classes and had plans to write unit tests and get close to full branch coverage in the major classes that are used by almost all of the other classes, the most important ones and ones that I took the time to write early in development was the Vec, Shapes, Point, Player, and Entity classes. This was really important as when writing and running these tests I found a few really important bugs, the main one was my divide and set function of my Vec class. I forgot to do the setting when I made the function. This would have been really bad and hard to catch outside of the isolated unit test.

Final Implementation

Starting with the NodeJS server and ExpessJS as this is the starting point for both the client and the server. This is a really basic framework for what can later be built upon. There are only 4 routes right now. First the root path ("/") which just loads a landing page which is meant to be a basic introduction to the game as well as saying what the controls are however this page has not been filled yet as I spend the whole development cycle working on the game. The next is the actual game path ("/game") which renders the game’s client code and is the starting point for the client of the game (Appendix C: follow graph from “index.js” in ClientJS). After that is the map editor path ("/mapEditor") which is where you go if you want to edit the map for the game or create new maps. Lastly is an api path ("/api/getMap/:mapName") for loading the map json data from the server, this takes a map name as a path variable so that it can be used to load any map. The last thing in the main node server file is set up for the server game loop (Appendix C: follow graph from "serverLoop.js in ServerJS).

The game map is probably the best place to start when describing the base of the game. The GameMap class holds a Map from string laneName to the lane, right now this is just for the two lanes but could be expanded if you wanted to make a map with more than 2 lanes. Also it holds the pixel size of the tiles which is used for going from the layers tile index to the draw location when drawing the tiles and adjusting the region x and y locations based on tile indexes. The next thing that the GameMap handles is the void region, which actually ended up being a region where it's beginOverlap function resets the players position back to its previous location. Rather than what I had originally thought of doing which was to have it filled with void tiles, but I felt like having those tiles checking for collisions would have been a hit on the performance of the game. At the start of the game the map gets loaded on the server side by just reading the map json file, and on the client side it uses the JavaScript fetch API to query the server's API route which sends the map's JSON information back to the client. Then using the makeFromJson function which extracts the tileSize, voidWidth and verticalLanes which is if the lanes are in a vertical or horizontal orientation. As well as the leftLane which then calls the Lane.makeFromJson function with the leftLane's JSON information (Code in Appendix D). Then it builds the GameMap based on the lane, voidWidth, tileSize, and orientation of the lanes. During the game the draw function gets called passing in the tileSprites and decorationSprites, this call will get expanded if anything else needs to be passed in. The draw calls might also in the future get refactored into a client side draw utility as images don't play nicely with NodeJS.

The Lane class is a glue class between the GameMap and the tiles or regions that are in the game. It holds an Array of Layers, and a Map of regions. When built with its makeFromJson function it creates a Vec2 for the dimensions of the lane, as the regions and layers which both get built from their JSON with again a makeFromJSON function call. After the lane is built from this function the GameMap can then mirror it calling the lane's mirror function with if the lanes are vertical and where the new lane's top left corner is. The mirror function handles mirroring all of the region and calling each of the Layer's mirror functions (Code in Appendix E). The next important functions are the generate functions. The first is generateStatic which goes through all of the layers and has them generate statics based on the tiles in that layer and pushes the layers statics into an array to be returned, this array then gets pushed into the CollisionEngine's statics QuadTree. The next is the generateRegions which returns an array of all of the regions in the lane which gets pushed to the CollisionEngine's regions array for easy calling during its update function. For the client there is a draw function that simply calls the draw function for all of the layers and regions. For the map editor there are two major functions in this class. Update which takes a region start and end, the layer that you want to place the tile on and the tile that you want to be placed there. As well as addRegion, which again takes a region start and end, as well as the region that is being placed and the name of the region. Last there are a few utility functions that are important. First being the spawnDragon function this gets called in the Lanes constructor when there is both a PVE Dungeon as well as a PVE Objective. Finally attachVMDecoration which attaches the updatable decoration to the Victory Monument so that the sprites can easily be changed when an objective is placed on the Victory Monument.

The last container class in the map section is the Layer, the main thing this contains is a 2d array of Tiles. When being built using it's makeFromJson it just unpacks all of the tiles that it contains, rebuilding its 2d array and flagging the Layer as empty if it doesn't contain any drawable tiles. The mirror function simply creates a new layer with the tiles mirrored either vertically or horizontally. GenerateStatic is an interesting function because it goes through all of the tiles in the layer and if the tile is either not walkable or not passable it creates a hitbox for the tile based on the tiles position and the topLeft of the lane that the layer is in pushing it to an array that gets used in the CollisionEngine's statics QuadTree. For the map editor there is an update function which replaces the existing tiles in a region to be what the new tiles need to be. On the client side a draw function is called to draw the layer. This function goes through all of the tiles in the 2d array if it has a TileSprite it calls a getArround function with the location of the tile as well as which tiles this tile can connect to, this builds an around bitmap which is used when drawing the TileSprite to choose the correct sprite for drawing so that all of the edges look like edges and line up with the surrounding tiles (Code in Appendix F). If it has a DecorationSprite it simply draws that decoration as none of the decorations need to attach to anything else.

From here there are the non container classes, the first of which being the Tile. The tiles have a few attributes, the first of which being the tile's location which is simply an x and y location. Then the tile's name which is used for looking up the correct sprite to draw. The next is the traversal options, walkable and passable. The next thing is the around value, which is used for drawing the right section of the TileSprite. Tiles also have a category which is always set to tile and a type which for now all of the tiles are basic, this could however be used to affect the typed projectiles in the future.

The Region classes are very important classes as it is what the player interacts with to get the different objectives and place them on the victory monument as well as the region for respawning. Figuring out how the event beginOverlap would work was an important part of getting these regions to work. I decided to go with two resetting Hash Maps, one for the current frame and one for the last frame, and when adding a player for overlapping it has two functions that it will call. If there is a player overlapping the region when there was nothing in the previous frame, this is used to start up the PVE boss AI only when the player is in the boss room. The other is a more general purpose begin overlap function that gets called on the first frame that the player enters the region. This is used for adding the objective to the player when they first enter the objective region, as well as removing any objectives from the player when they approach the Victory Monument. The VictoryMonument region is special because it needed to hold the decoration which is what gets visually updated on the map when a player places an objective, it keeps track of the objectives by holding a Set so that it can quickly look up which it has as well as ignoring duplicates of which color of objective has been placed (Code in Appendix G).

The next big component of this project is the entity classes. The base entity has most of the information, such as the entity ID, this caused a few issues because originally I was building these IDs client side and sending them to the server, but quickly realized that the other clients were getting the same ID causing projectiles and such to be deleted when they weren’t supposed to because the IDs weren't unique sense I was just using a count for the ID. This got changed to keeping the ID count on the server side and whenever a client makes a projectile it waits for the server to give it an ID. The next attribute the entities have is their location as well as their old location. This is for keeping track of where the entity is and placing the hit box in the correct location when being generated. The next is the entity's look direction, this is used when launching projectiles to go in a certain direction as well as used for drawing the image in the right orientation. The speed of an entity is used when it is being moved, generally in the direction of its look when applied to projectiles (Constructor Code in Appendix H). The main functions for the entities are makeObject which turns the entity into a json object for sending between server and client. MakeShape which turns the entity into a circle for running collision checks, as well as makePoint which is also used similarly for collisions. The move function moves the entity in the look direction at the speed of the entity. Lastly for the base class is the updateInfo this pulls information from the json object and applies it to the entity.

Projectiles are a subclass of Entity, they hold the ID of their owner, and how much damage they do. The biggest factor for this class is the hit function that checks if the thing it hit was hit has the same id as the owner to make sure that the projectile doesn't hit its owner. Then it checks the type of the projectile is the same as what it hit, the hit object is not a tile it returns false. If the hit object is a tile it sets a remove boolean to true. Then checks if the object hit was damagable, player, or dragon. If it was, it calls their hurt function. There are also subclasses for the fireball, waterball, and plant seed, all of which have modified hit functions that if the type is something that it is super effective against it takes the damage that the other projectile would have done (Code in Appendix I).

From the map and entities there are the game loops, both server and client are very similar where the only real differences is that the server runs the monster's AI and handles communication between it and the client, and the client handles the player controller and drawing to the screen. The loops start by loading and initializing the GameMap, then creating the collision engine based on the size of the GameMap. The server loop then creates a connections wrapper to handle all of the incoming connections from the client. Then they set up the static collidable objects, regions, and any dynamic map elements to the collision engine. Prior to running the client loop the client code sends a connection message to the server and then waits for it to respond with player information, then build a player controller from that information.

The loops, if running, every tick which is set in the Time class (Code in Appendix J) which handles updating the time current time and dt (time since last frame). Each frame the game loop updates the game state, be that on the client loop running the player controller to let it move and use any abilities based on what keys are pressed. From there they call the collision engine update function which runs all of the projectile's move functions then checks for any collisions. The collision engine returns a delete array which is anything that returns true for being deleted from it's hit function. If it is a player and their current health is less than zero the loop calls the player's kill function, it also sends player information. If the item in the delete array is a projectile it calls remove on it which removes it from the collision engine and on the client side it also removes from the canvas wrapper's drawables. If the item is a victory monument then it sends the victory monument info to the server or client. Lastly if it is a dragon and the dragon's health is less than zero it adds the remove function as the dragon's delete call back so that when it is done with its death animation it can delete itself. From there on the client side the loop clears the canvas and then draws the game map layser, followed by the canvas's drawables, then the player controller, this is where I would add an over layer to the game map if I wanted stuff to hide the player and projectiles.

The last piece that makes this actually a game is based in the connections class and the server loop class, the actual game events. In the connections class. When a player marks themselves as ready the connections class adds to a ready count then checks if the server is running already. If it is, it tells the client to start quietly, which means that the client will not kill the player to reset their location. Then it checks if the ready count is greater than or equal to the number of connected players, which if it is it then broadcasts start game, which the client loops kill their players to make sure everyone is at the spawn as a safety measure even though none of the players should have been able to move until this point. Then the serverloop starts running.

Discussion

These are the problems that happened in the development process of the project. The first problem in designing was how to handle collisions in an efficient way, this was solved by using a few QuadTree data structures, one which resets every frame of the game loops for dynamic elements and the other is for the static objects from the map.

The next problem that came up in development was keeping the server and client in agreement with what projectiles or where the players were. This was solved mostly by constantly sending updates whenever something changes such as deleting a projectile, moving a player, something dies, or something gets placed on the victory monument.

A design problem that came up was the need for drawing different tiles on top of each other, such as putting a path on top of a grass tile, or the decoration on the victory monument tiles. This was solved by putting layers into the lane of the game map so that I could place tiles on top of each other like how layers work in photo editing applications.

When designing the game I wanted the player's intentions to be visible, so I wanted the player to be drawn looking at where they are aiming with the map. Having a drawImageLookAt function was a great place for starting so that other players can see approximately where you are targeting without exactly showing the cross hairs.

When it came to the PVE dungeons boss the dragon I really wanted to tie what the dragon was doing to specific frames on animation, this was the first time I had designed such a system so what I decided to do with it was write down key frames for each animation for example "attack2_160" was the fireball key frame, where attack2 is the phase name and name of the animation that it was on and 160 was the frame of the animation that it was (Code in Appendix K).

An issue that snuck up on me in the development of this project was the loading of images, this was a place where Javascripts asynchronous nature hurt the project as it would not wait for the image to be loaded fully before trying to draw it. To fix this I ended up putting all of the images in the html document with hidden tags so that they would not be displayed, then only started the client on the document’s ready state of complete.

The last big issue in development of this project was designing the regions overlap events, while I am not sure that this is the most optimal solution it was the solution that I implemented. Which is two resetting Maps, one for the current frame and one for the last frame and then at the end of the frame the game loop it would check the symmetric difference between the two. Players that were in the current frame but not the last frame get the begin overlap called with them, the players that were in the last frame but not this frame were called with end overlap.

Conclusion

Mirrored Fantasy fills an area in the MOBA genre that has grown stale with most games having the same formula of three lanes and a jungle where you just push towers till you destroy the other team's base. Mirrored Fantasy solves this problem by removing the standard map and putting teams in isolated lanes where there is one dungeon with focus on team interaction in the player vs player dungeon, and brings ability interactions and counters from popular RPGs such as Pokemon’s type system (Pokemon Wiki) or Divinity Original Sin’s environmental effects (Divinity Wiki). Much of the player controller and movement designs were taken from the game BattleRite with the movement and ability keybindings.

I learned a lot from this project as it was chosen as a culmination of all of my computer science learning. I wanted to integrate as much from each part that I could making this a rather big project. The first thing that I learned was how to quickly integrate code examples found and alter and integrate it where needed. I learned a lot about how to handle game states and syncing states and information between server and client.

For the future of this project I plan on finishing the things that I didn't get to this cycle and expanding ideas that came up while making the game. First finishing the puzzle dungeon by giving it unique mechanics that make it interesting to play multiple times and rewarding to those that learn its mechanics. Then creating the terrain abilities so that you can build your own defences giving more skill expression in the PVP dungeon. The last big thing that I want to add is more abilities and interactions with each other and the map, such as powering up the waterball when it goes over a water tile.

References

Battlerite. “‘Battlerite Is the Best Team Fight You've Ever Had.’” *Battlerite*, Stunlock Studios, 2019, arena.battlerite.com/.

D'Argenio, A. (2018, July 10). Statistically, video games are now the most popular and profitable form of entertainment. Retrieved September 23, 2020, from https://gamecrate.com/statistically-video-games-are-now-most-popular-and-profitable-form-entertainment/20087

Divinity Wiki. (2020, July 11). Environmental Effects (Divinity: Original Sin). Retrieved August 27, 2020, from https://divinity.fandom.com/wiki/Environmental_Effects_(Divinity:_Original_Sin)

Pokemon Wiki. “Types.” *Pokémon Wiki*, 2020, pokemon.fandom.com/wiki/Types.

SkiTrip. *New? Here's How to Play*. 2017, wool.run/f/general/27.

Appendix

Appendix A: Usability and Testing Plan

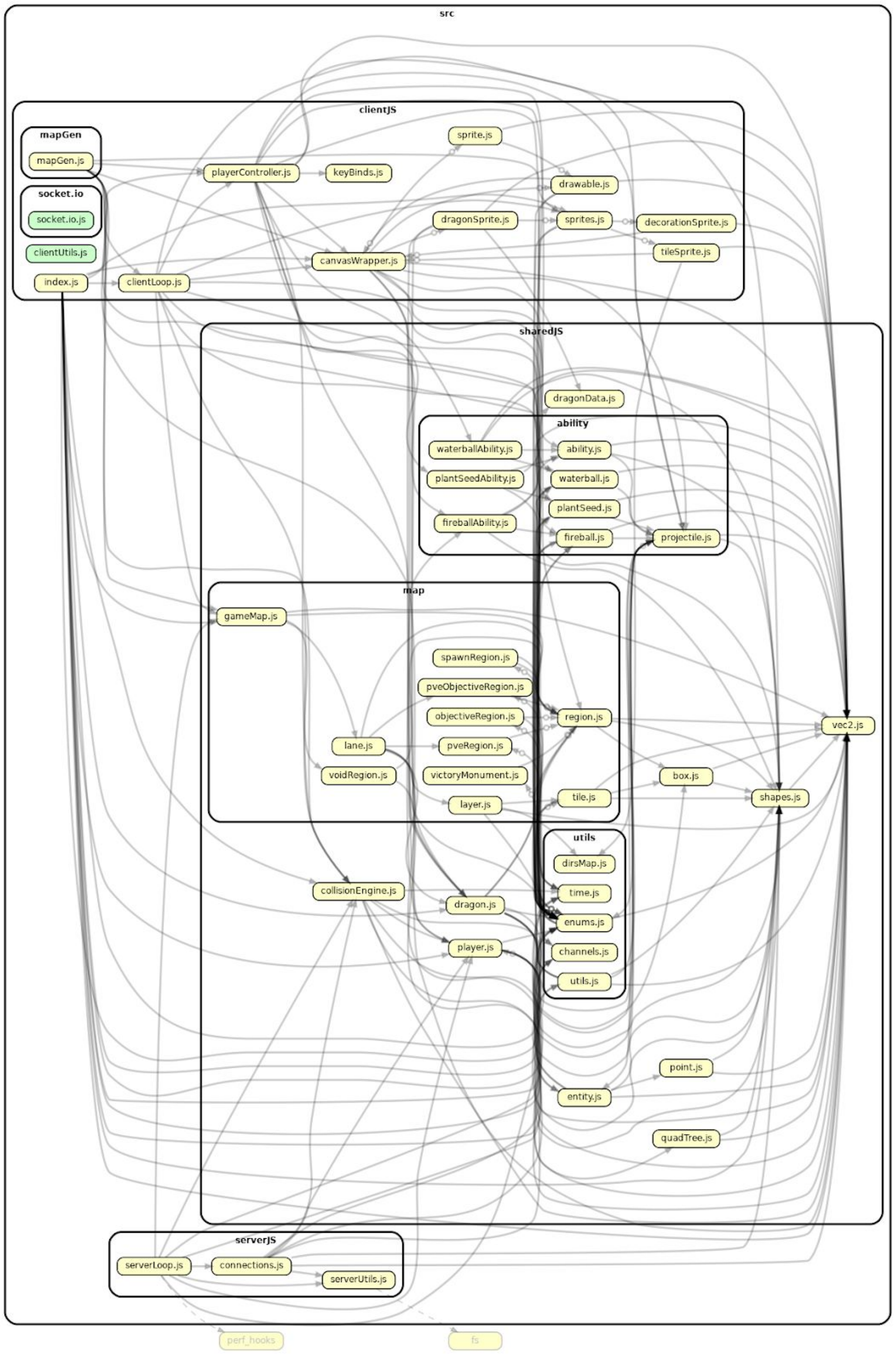
Unit Testing will be run using Mocha and Chai run in node js to test classes, and integration testing will be using cypress to automate testing of multiple browsers and compatibility. Getting close to 100% branch coverage of the base classes and their functions to ensure that the base components are working properly.

Getting at least 4 people together to play the game, have some of them stream or record their screens so that we can rewatch later and analyze what players movements and priorities look like. A satisfaction survey asking which parts of the game the players feel like could improve, any bugs they found, and which parts of the game they liked. While having the players play I asked them to do their best to complete all game objectives to see if any of the objectives don't place on the monument or fail to get picked up. As well as checking the end game state triggering from the server. After testing I also asked how the players felt about the speed of the game, ranging from movement speed, projectile speed, and how long the matches take.

Appendix B: Team roles and responsibilities

Andrew Bell: Lead game designer, lead programmer, and manager of testing.

Appendix C: Dependency Graph of All Code Files



Appendix D: Lane.makeFromJSON - /src/sharedJS/map/lane.js

```
static makeFromJSON(json, tileSize) {
   const {
        name, dimentions, layers, regions
    } = json;
    const dims = new Vec2(dimentions.x, dimentions.y);
    const tilesize = new Vec2(tileSize.x, tileSize.y);
    const topleft = new Vec2(); //assume top left is at 0,0 for left
lane
    const newLayers = layers.map((layer) => Layer.makeFromJSON(layer,
dims));
    const newRegions = new Map(regions.map((region) => [/** @type
{string} */ (region[0]), Region.makeFromJSON(region[1])]));

    return new Lane(name, dims, layers.length, tilesize, topleft,
newLayers, newRegions);
}
```

Appendix E: Lane.mirror - /src/sharedJS/map/lane.js

```
/**
 * @param {boolean} vertical
 * @param {Vec2} laneTopLeft
 */
mirror(vertical, laneTopLeft) {
    //clone the layers and mirror them
    const mirroredLayers = this.layers.map((layer) =>
layer.mirror(vertical));
    //find the new lane center
    let laneCenter =
this.dimentions.multiplyScalar(0.5).add(laneTopLeft).multiplyVecS(thi
s.tileSize);
    //mirror the regions
    const mirroredRegions = new Map();
    for(const region of this.regions.values()) {
        const centerToRegion = region.center.sub(this.region.center);
        if(vertical) centerToRegion.x *= -1;
        else centerToRegion.y *= -1;

        mirroredRegions.set(region.name, new
REGIONS[region.name](laneCenter.add(centerToRegion),
region.dimentions.clone(), region.name, region.color));
    }

    //create new mirrored lane
    return new Lane("", this.dimentions.clone(), this.layers.length,
this.tileSize, laneTopLeft, mirroredLayers, mirroredRegions);
}
```

Appendix F: Layer.draw - /src/sharedJS/map/lane.js

```
/**
 * @param {import("../../clientJS/canvasWrapper.js").default} canvas
 * @param {Vec2} topLeft
 * @param {import("../../clientJS/sprites.js").tileSprites}
tileSprites
 * @param {import("../../clientJS/sprites.js").decorationSprites}
decorationSprites
 */
draw(canvas, topLeft, tileSprites, decorationSprites) {
    if (this.empty) return;
    const [width, height] = this.dimentions.getXY();

    for (let j = 0; j < height; j++) {
        for (let i = 0; i < width; i++) {
            //grab tile name
            const tileName = this.tiles[j][i].name;
            //check if it is a tile or decoration sprite
            if (tileSprites.has(tileName)) {
                //get the sprite
                const sprite = tileSprites.get(tileName);
                //get the around value
                //TODO move to loading map so it doesn't get run
every frame
                const around = this.getAround(i, j, sprite.connects);
                const tile = TILES[tileName].clone().init(new Vec2(i,
j), around);
                //sprite.draw(canvas, tile.location, tile.around);
                tileSprites.get(tile.name).draw(canvas,
tile.location.add(topLeft), tile.around);
            } else if (decorationSprites.has(tileName)) {
                decorationSprites.get(tileName).draw(canvas, new
Vec2(i, j).addS(topLeft));
            }
        }
    }
}
```

Appendix G: VictoryMonument objectives - /src/sharedJS/map/victoryMonument.js

```
/**
 * @param {Array<Tile>} newDecorations
 */
setDecoration(newDecorations) {
    for(const decoration of newDecorations) {
        this.decorations.set(decoration.name, decoration)
    }
}
/**
 * @param {string} objective
 */
activateDecoration(objective) {
    console.info(objective, " added to VM");
    this.objectives.add(objective);
    const decorationStr = `${objective}Pillar`;
    if(this.decorations.has(decorationStr)) {
        this.decorations.get(decorationStr).name = decorationStr +
"Active";
    }
}
/**
 * @param {Player} player
 */
beginOverlap(player) {
    if(player.objectives.size) {
        for(const objective of player.objectives) {
            this.activateDecoration(objective);
        }
        if(this.objectives.size === 3) {
            console.info("A winner is you");
        }
        player.objectives.clear();
    }
}
```

Appendix H: Entity.constructor - /src/sharedJS/entity.js

```
/**
 * Makes a new Entity
 * @constructor
 * @param {Vec2} location The start location of Entity
 * @param {string} imgSrc Display image
 * @param {Circle|Rectangle} hitbox Hittable region
 * @param {number} [speed=0] How many pixels the entity moves per
second
 * @param {number} [scale=1] The scale factor to get to 64 pixels per
tile
 * @param {Vec2} [lookDirection=Vec2(1,0)] Which direction the entity
is looking at
 */
constructor(location, imgSrc, hitbox, speed = 0, scale = 1,
lookDirection = new Vec2(1,0)) {
    console.assert(location instanceof Vec2, "Loaction not a Vec2",
location);
```

```
    if(!(location instanceof Vec2)) {
        throw TypeError("Entity: Location not Vec2");
    }
    this.id = (idGen++).toString();
    this.location = location.clone();
    this.oldLocation = location.clone();
    this.imgSrc = imgSrc;
    this.hitbox = hitbox.clone();
    this.lookDirection = lookDirection;
    this.speed = speed;
    this.scale = scale;
    //mostly for debugging now
    this.overlapping = false;
    this.damage = 0;
    this.maxHealth = 0;
    this.currHealth = 0;

    this.type = TYPES.basic;
    this.category = CATEGORY.none;
    this.lastHit = null;
}
```

Appendix I: Projectile.hit - /src/sharedJS/ability/projectile.js

```
/**
 * Basic projectile just hits players
 * @param {Player|Projectile|Entity|Tile} other
 */
hit(other) {
    if (/** @type {Player} */(other).id === this.ownerID) {
        return false;
    }
    let remove = false;
    if (other.type === this.type) {
        //Same type do nothing
        if (other.category !== CATEGORY.tile)
            return false;
    } else {
        if (other.category !== CATEGORY.tile)
            remove = true;
    }

    //if hitting a player deal damage
    if (other.category === CATEGORY.damageable || other.category ===
CATEGORY.player || other.category === CATEGORY.dragon) {
        /** @type {Player} */(other).hurt(this.damage, this.id);
        remove = true;
    } else if (other.category === CATEGORY.tile) {
        //Projectiles go over passable tiles
        if(!/** @type {Tile} */(other).passable) remove = true;
    }

    return remove;
}
```

Appendix J: Time class - /src/sharedJS/utils/time.js

```
export default class Time {
    /**
     * @param {import("perf_hooks").Performance} performance
     * @param {number} [now] Current time
     * @param {number} [last] When last frame ran
     * @param {number} [dt] Change in time since last
     * @param {number} [tickRate] Ticks per second game loop runes
     */
    constructor(performance, now = performance.now(), last = 0, dt =
0, tickRate = 1/30) {
        this.now = now;
        this.last = last;
        this.dt = dt;
        this.tickRate = tickRate;
        this.performance = performance
    }
    /**
```

```
     * Updates now and dt with performance.now()
     */
    update() {
        this.now = this.performance.now();
        //calculate dt
        this.dt = this.dt + Math.min(1, (this.now - this.last) /
1000);
        //run frames while they need to run fixed timestep gameloop
        return this;
    }
}
```

Appendix J: keyFrames Map - /src/sharedJS/dragonData.js

```
export const keyFrames = new Map(
    [
        ["attack1_55", "melee"],
        ["attack2_120", "fireball"],//launch fireball
        ["death_160", "kill"],//remove lock on objective
        ["death_300", "delete"],//remove hitbox from collision
    ]
);
```

/src/sharedJS/dragonData.js

```
if (keyFrames.has(`${this.phase}_${this.frame}`)) {
    const action = keyFrames.get(`${this.phase}_${this.frame}`);
    if(action === "melee") {
        //TODO melee attack
    } else if(action === "fireball") {
        //Launches fireball in serverLoop
    } else if(action === "kill") {
        this.kill();
    } else if(action === "delete") {
        this.deleteCall(this);
    }
}
```

/src/sharedJS/dragon.js:129