Brian Adams

CSC 413-02

Fall 2021

Term Project

1. https://github.com/csc413-SFSU-Souza/csc413-tankgame-brianadams731
2. https://github.com/csc413-SFSU-Souza/csc413-secondgame-brianadams731

2) Introduction:

a) Project Overview:

The term project is a combination of two separate games. These games must be functional, adhering to prespecified guidelines and be designed in a way where they largely adhere to good modern programing practices like not breaking encapsulation, using solid and implementing inheritance or composition to keep code dry. These games are an exercise in code design as games often have complex state and logic that must be organized in a way that is manageable and extendable so that as additional features get implemented, we do not need to make significant rewrites.

The first project is a tank game, where skeleton code was offered, and a rigid set of requirements were given. This game was a more “on rails” experience to write as the videos accompanying this game gave helpful tips regarding how to approach and solve various problems. The second game was largely hand off, we could choose from a series of games, or build a game that we came up with. There were less resources to point us in the correct direction, although drawing upon the experience from the previous game and rewatching the videos to get an idea on how to solve similar problems could be incredibly helpful.

b) Tank Game:

The general idea for the tank game is that two players will each control a separate tank. These tanks need to be able to rotate, move and shoot. My tank game wound up being a cooperative experience where two players teamed up to shoot hordes of zombies that were spawned in waves. The goal of the game is to simply

survive as many rounds as possible. The game ends when both tanks run out of lives. There are three different power ups that respawn every round. The first being a health pickup, this restores lost hp. There are no overheating mechanics, this means if you pick up a health pack when your hp is at 95/100, it will only refill back to 100. The second power up is a gun modification. This turns your bullets into lasers. This will damage anything in the path, providing no breaks and moves much faster than the default bullets. The other upgrade is a machine gun. This is a fast-firing gun where the bullets move much slower than the laser and have a cool down between each shot. The zombies have a finite amount of health each, and the amount spawned increases as the number of rounds increase. Players take damage when a zombie touches their tank. Once players hp reaches 0 they lose a life and respawn.

c) Platformer:

The general idea for my second game is a sidescroller platformer very similar to Mario. You control a player who moves through a 2d side scrolling world. There are various obstacles like pits, spikes and enemies. If the player collides with the spikes or falls down a bottomless pit they die and are given a choice to try again. If the player lands on top of an enemy, the enemy will die, and the player score will be incremented as well as being propelled upwards. If the player collides with an enemy and is currently squashing the enemy from above the player will die and be prompted to try again. There are also coins that a player can collect that will increment their score. There is basic physics like one would expect in a platformer e.g., gravity and foreground collision. The player can only jump when they are on a solid platform which prevents double jumping.

The enemies follow the player and will jump to get over any obstacles that might be between them and the player. The game ends when the player reaches the end of the stage. Once the end of the stage is reached a game over menu will fade in informing the player of their score and giving them an opportunity to replay the stage.

3) Development Environment

a) I used Java 14 to develop the tank game.
b) For the first game I used IntelliJ IDEA, for the second game I used VS Code.
c) For the first game I used Java and LibGDX. For the second game I used TypeScript, Pixi for WebGL rendering, Howler for managing audio playback and Vite for bundling.

4) How to build or import your game in the IDE

a) To import the tank game, you simply need to open an IDEA and open the import project panel, point to build.gradle in src directory. IntelliJ should correctly import project based upon spec in build.gradle

   To build the tank game open the build config, use com.tank.game.desktop.DesktopLauncher as the diver class, and endure the working directory is pointed at .../csc413-tankgame-brianadams731/src/core/assets,

   you will need an absolute path so prepend the 3 dots with your path to the assets. If you are missing any dependencies gradle should automatically download them.

b) To import the second game, you simply need to open the root folder in VS Code, this can be done in the terminal by navigating to the root folder and using the “code .” command. To install the necessary dependencies, you will need to have node, and npm installed. I used npm version 7.24 and node version 14.18 in development. In either case simply navigate to the root directory of the project in your terminal, that is the same directory with the package.json file, then run “npm install”. This will install all the developer dependencies, as well as project dependencies for the project. I also included my python (Python 3.9) script that I wrote to convert excel workbooks into json files, if you would like to check the functionality for whatever reason you simply need to navigate to the xlstosjon directory in your terminal and install the dependencies provided in the requirements.txt file, you in Windows you simply need to run “pip install -r requirements.txt”, and in Linux “pip3 install -r requirements.txt”. I would also recommend creating a virtual environment if you do not want a global installation of the dependencies, this can be accomplished by using “python -m venv venv” in Windows and “python3 -m venv venv” in Linux. This will create a venv directory which contains an activate script that will load you into a venv session. To build the game you simply need to navigate to the root directory, that is the directory with the package.json file and run npm run build.

c) To run the jar simply navigate to it in your terminal and use java -jar TankGame.jar

5) How to run the Game:

To run the game one, you need to run the provided jar, this jar is in the jar folder which is in the root directory of the project. Once there you need to use the “java -jar TankGame.jar” command in your terminal, or double click the jar file if you have jar extensions configured to execute. The controls are-

Player One
Forward : Arrow Up
Backward : Arrow Down
Rotate left : Arrow Left
Rotate Right: Arrow Right
Shoot: Mouse 1
Rotate Turret Left: Mouse Position
Rotate Turret Right: Mouse Position

Player Two
Forward : w
Backward : s
Rotate left : a
Rotate Right: d
Shoot: Space
Rotate Turret Left: q
Rotate Turret Right: e

The rules of the game require the player to avoid zombies while eliminating them for as long as possible. You can use the tile map to funnel and prevent the zombies from chasing you while you focus on whittling down a smaller group. Be careful as the zombies can break breakable blocks. The zombies decide which player to chase based upon which player is closest to them. To eliminate the zombies, you need to shoot them.

To run the second game, you can either navigate to https://platformer413.netlify.app/ and play the game there, or once you have installed the dependencies for the project to your local environment, navigate to the root of the project in the terminal, (the directory that contains the package.json, and use the “npm run dev” command. This will spin up a local development server that hosts the project on port 3000. You can either click the link provided in the terminal or open your browser and navigate to localhost:3000. I used Chrome and Firefox when developing the game. I cannot confirm whether the game will run on Safari as Apple's refusal to offer their browser on Linux or Windows, and inadequate and spotty support features that have been agreed upon and supported in Chrome and Firefox make testing my projects compatibility with Safari and fixing any bugs that may result from this a major imposition.

The rules of the second game require the player to reach the end of the stage located in the furthest right portion of the map, while avoiding any hazards. The hazards include spikes, where if the player touches any part of them, they will die, bottomless pits, where if the player falls them, they will die, and enemies that will chase the player, and if the player does not defeat them by landing on top of them will continue to chase the player. If the enemy touches the player without being squished by the player, the player will die. The player can also collect coins along the way to increment the score counter.

The controls are -

Run right: d
Run left: a
Jump: Space

6) Assumptions Made:

For the tank game I assumed that LibGDX would be relatively opinionated and straightforward to work with. This was largely true although this might be due to me not using many of its features like scenes. I used LibGDX more like a canvas with extra utility functions. I decided to use LibGDX like this after a quick glance at the docs simply because I didn't want design decisions made for me in this specific project, even if they would have without a doubt been easier to implement. I also assumed that assets would be easy to find, while assets were provided with the project, I opted to find my own. This was not the case, and if I were to redo this project, I would absolutely have kept with the assets provided. Another assumption I had was that the design of the game would be relatively intuitive because of its smaller size. This was somewhat true, in that I saw a clear divide in what actors, controllers and sprite managers etc. should do, but I ran into ambiguous cases regarding who should own the BulletBag class and whether the controller or the actor should process the collisions.

I assumed that menus would be easy to implement in LibGDX and this was not the case. Menus were largely a post hawk addition which was a mistake. I imagine if I had spent more time with the documentation, I would have been able to create a scene where I could use the predefined menu components. In either case time was a major factor in my decision to simply draw sprites and switch them out to represent various button states while also taking advantage of the event system to emulate menus. The result is less than desirable with buttons firing on a key down instead of what would be the more anticipated key up event. I tried various solutions to work around the lack of a mouse key up event, but in almost every case they made the menu feel less responsive.

Finally, I anticipated that I would miss the flexibility of higher-level languages like TypeScript while writing this game. While Java was a little more verbose, I found it to be incredibly and surprisingly enjoyable to work in. I did miss first class functions and some of the more functional features of other languages, but the tooling and stricter nature of its typing system of Java largely made up for the loss (although the lack of variable interpolation within strings is puzzling).

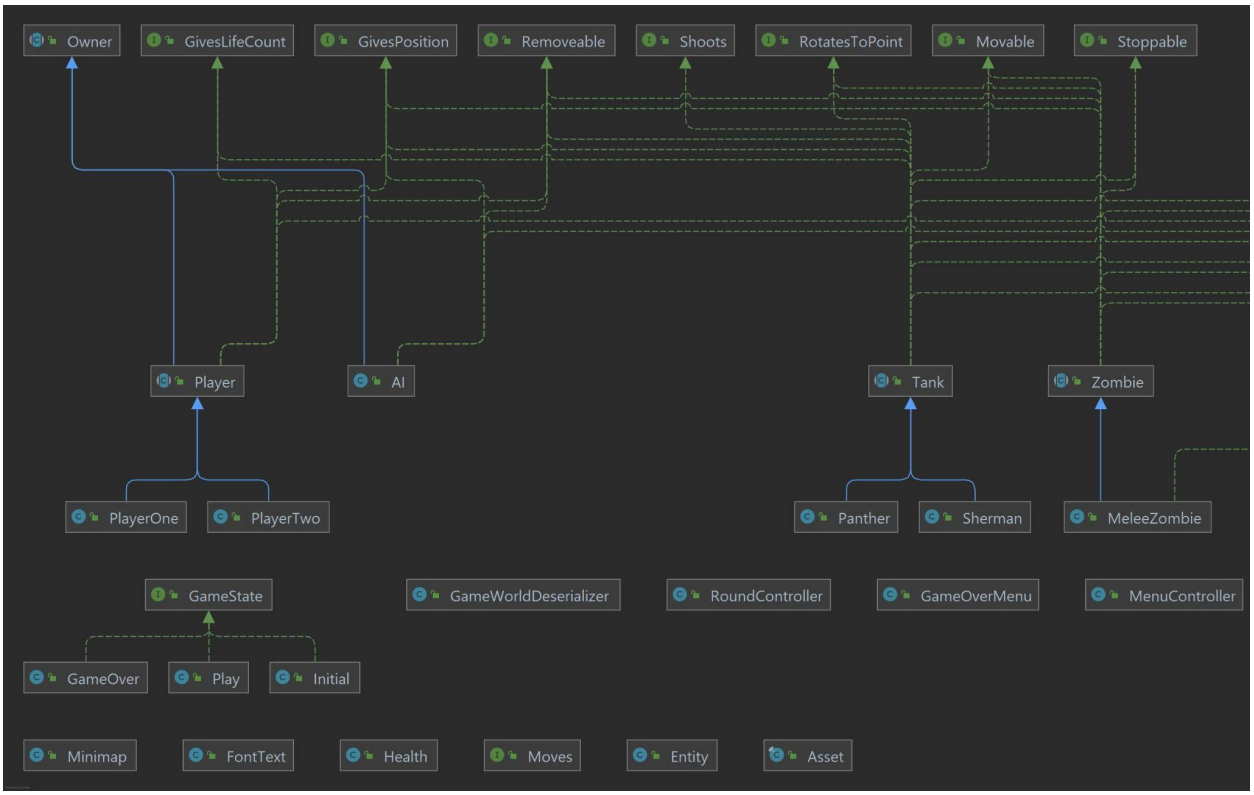
For the platformer game I assumed that working with WebGL through Pixi would be more difficult than using canvas. I was very pleasantly surprised to find that Pixi is incredibly easy to use. The documentation was great, the api is incredibly straightforward and Pixi had full TypeScript support without needing to search Definitely Typed. I was very happy using Pixi and plan to consider using it in future.

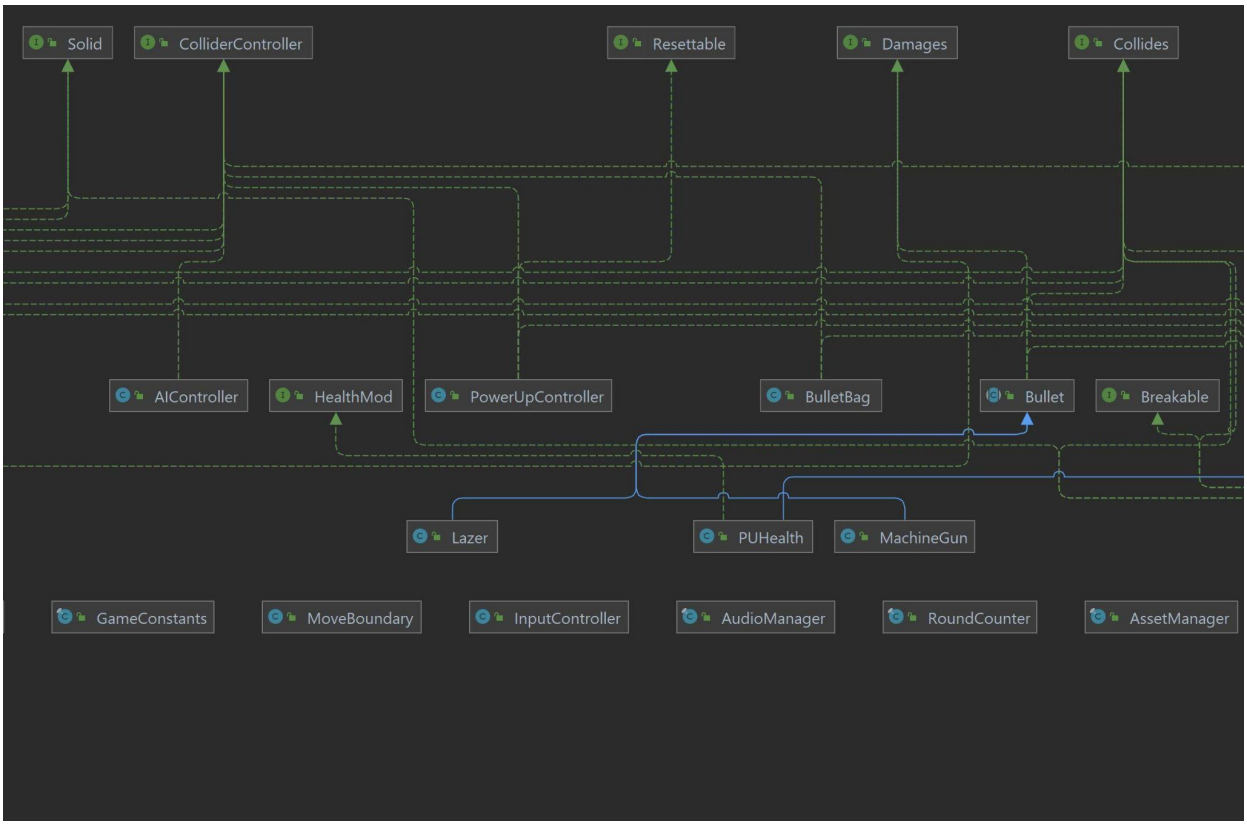
The next assumption I had was that physics would be relatively easy. Early in the development I decided to use MatterJS, which is a complete physics engine. The prototype that was mocked with Matter felt great but as I added blocks, I realized that it was overkill and not necessarily built for the style of 2d block-based platformer I was building, so I scrapped MatterJS and decided to implement physics myself. The gravity and player movement were simple, but the collisions proved to be much more difficult than I had anticipated. I tried several different solutions, all of which were great at detecting when a collision happened but were not great at accurately detecting which sides of the objects collided. I finally settled on an algorithm based upon Minkowski sum, with some minor alterations to prevent characters from catching on the edges of blocks while falling against a wall.

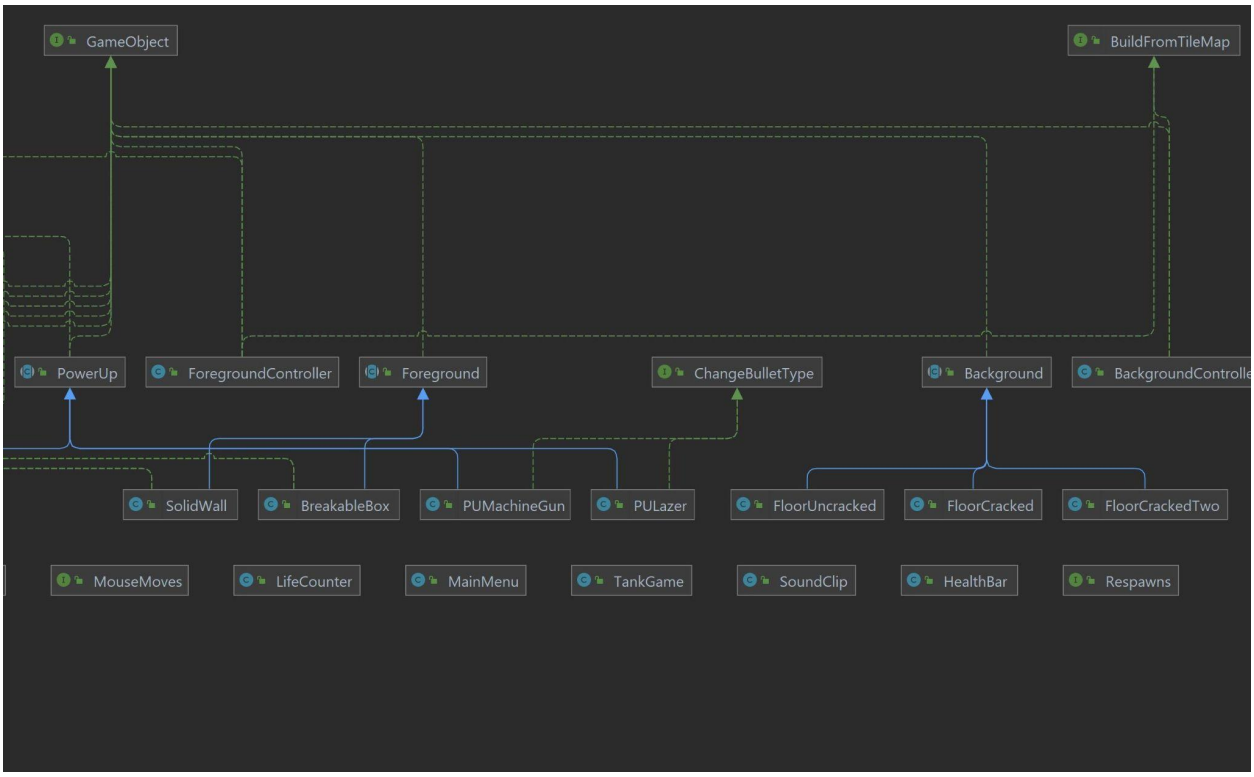
Another assumption I made was that menus would be incredibly easy to make, this was proven to be true. Instead of leveraging the canvas I simply appended and removed elements from the DOM accordingly. Taking advantage of the excellent tools for UI building provided by HTML, CSS and JS. This made creating a smooth, animated menu a trivial task.

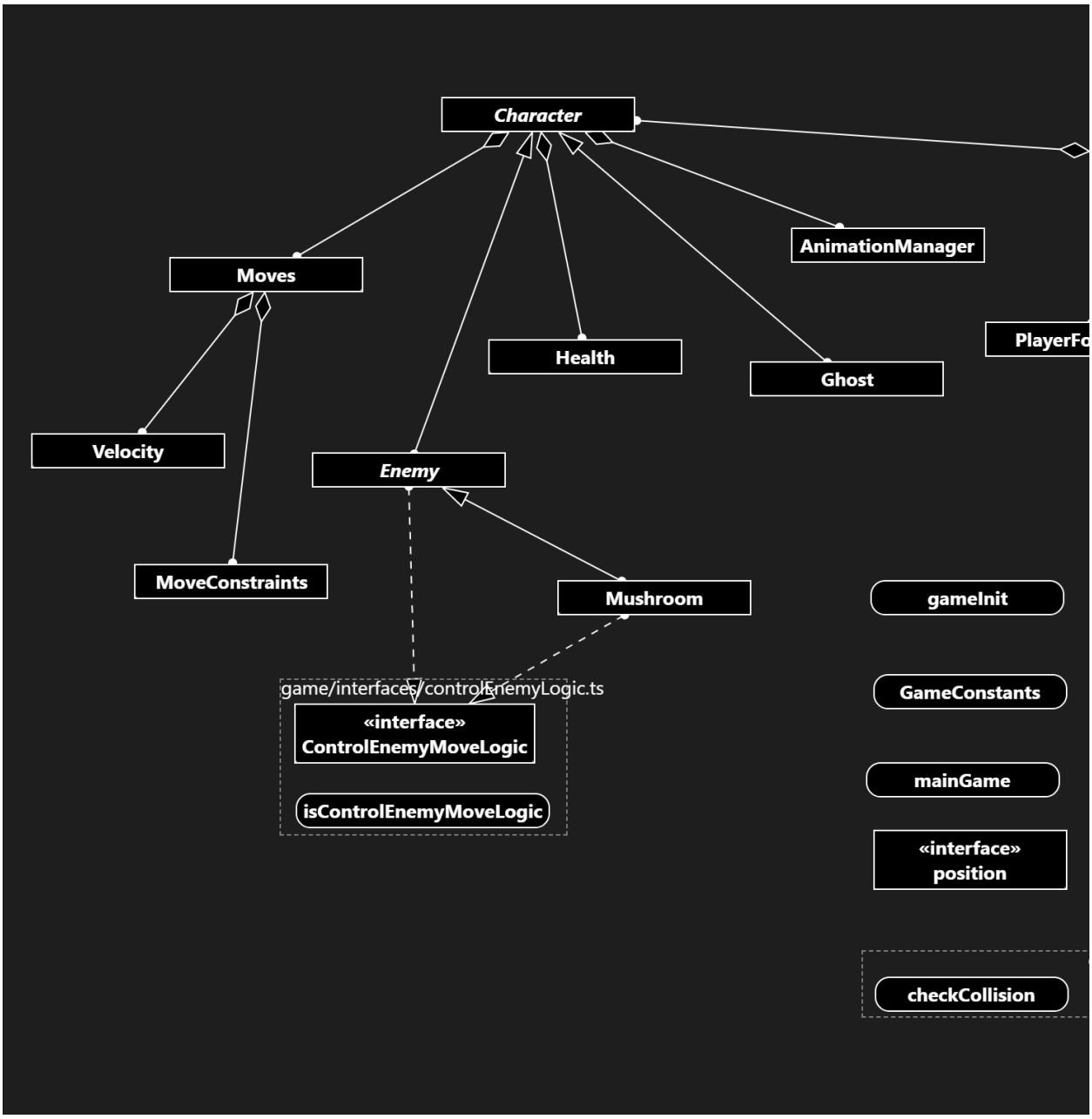
Finally, I assumed that Vite would be a largely hassle-free bundler, this was for the most part true, but I did run into some major issues trying to integrate a library for unit tests. I typically use Parcel to bundle and Jest to test, this is a great combination as Jest has first class support with Parcel. This is not the case with Vite as it requires the developer to use Babel to transpile your code for Jest, and ESBuild to transpile your code for everything else. This means that I would need to include additional dependencies on top of Jest and more config files. At this point I was inclined to go back to Parcel but decided to stick with Vite just for the sake of experiencing tools outside of my normal workflow. In the end I did enjoy the speed of Vite's dev server and HMR, but this came at the expense of unit testing. I will most likely continue to use Parcel in the future.

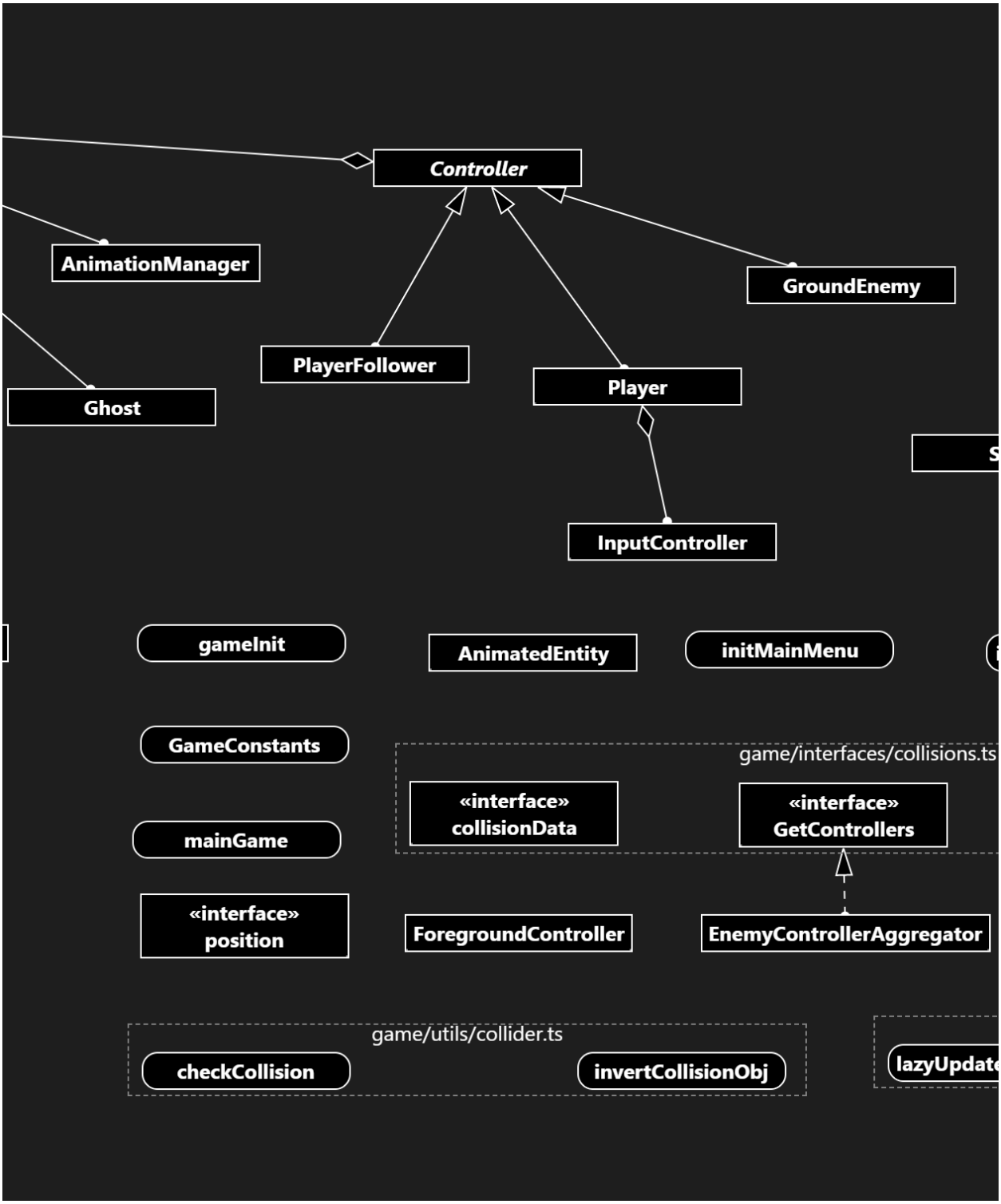
7) Tank Game UML Diagram:

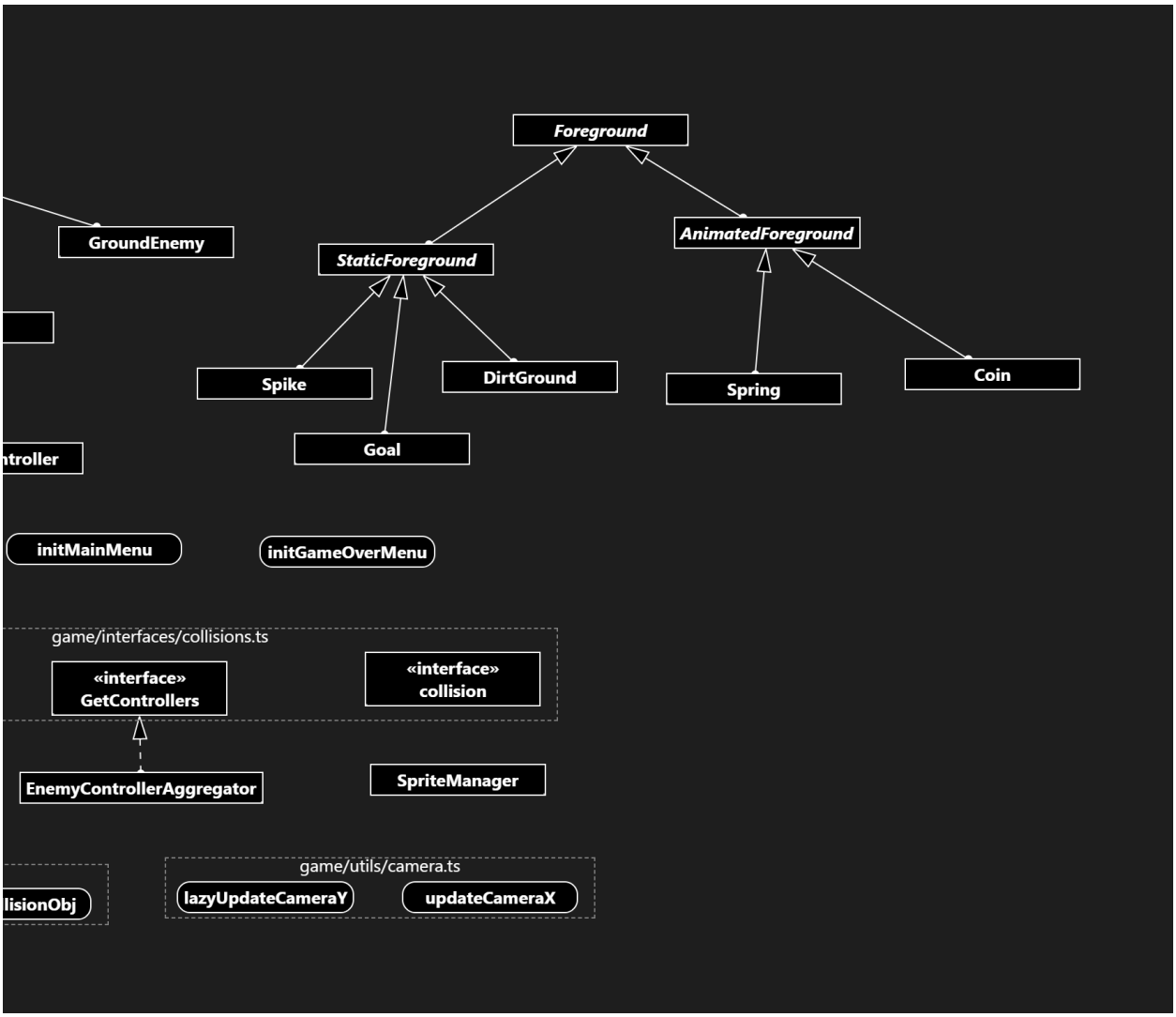

Owner
GivesLifeCount
GivesPosition
Removeable
Shoots
RotatesToPoint
Movable
Stoppable
Player
AI
Tank
Zombie
PlayerOne
PlayerTwo
Panther
Sherman
MeleeZombie
GameState
GameWorldDeserializer
RoundController
GameOverMenu
MenuController
GameOver
Play
Initial
Minimap
FontText
Health
Moves
Entity
Asset


Solid
ColliderController
Resettable
Damages
Collides
AIController
HealthMod
PowerUpController
BulletBag
Bullet
Breakable
Lazer
PUHealth
MachineGun
GameConstants
MoveBoundary
InputController
AudioManager
RoundCounter
AssetManager


GameObject
BuildFromTileMap
PowerUp
ForegroundController
Foreground
ChangeBulletType
Background
BackgroundControlle
SolidWall
BreakableBox
PUMachineGun
PULazer
FloorUncracked
FloorCracked
FloorCrackedTwo
MouseMoves
LifeCounter
MainMenu
TankGame
SoundClip
HealthBar
Respawns


Character
AnimationManager
Moves
PlayerFo
Health
Ghost
Velocity
Enemy
MoveConstraints
Mushroom
gameInit
game/interfaces/controlEnemyLogic.ts
GameConstants
«interface»
ControlEnemyMoveLogic
mainGame
isControlEnemyMoveLogic
«interface»
position
checkCollision


Controller
AnimationManager
GroundEnemy
PlayerFollower
Player
Ghost
InputController
gameInit
AnimatedEntity
initMainMenu
GameConstants
game/interfaces/collisions.ts
«interface»
collisionData
«interface»
GetControllers
mainGame
«interface»
position
ForegroundController
EnemyControllerAggregator
game/utils/collider.ts
checkCollision
invertCollisionObj



9) Shared Classes:

Since both games were written in different languages, I was not able to share any code between the two. I did however use a few classes from the tank game to model classes in my platformer game. Specifically, regarding how collisions were processed, movement was determined, and health was represented. In the tank game collisions were calculated outside of the controller entities, then passed in for the characters that the controllers controlled to process. This was handled very similarly in my platformer game. I also abstracted away health and respawn status away in very

similar Health classes in both games. Finally, I abstracted away movement very similarly between both games as well.

10) Class Description for the Tank Game: (working left to right on UML Diagram)

Owner:

The Owner class was the base abstract class which the player and ai classes would be derived from. This simple class exposed the basic functionality of an owner, that is to update, draw and dispose of their owned items.

Player:

The player class is an abstract class in which Player One and Player Two are derived from. This class manages the details needed by the main loop to manipulate the player. This includes managingCollisions, exposing colliders and controlling the updating of the tank from the outside.

PlayerOne and PlayerTwo:

These classes are really the implementation of the player class. They decide which sprite they will be controlling and the input method in which the sprite will be controlled by.

AI:

Much like the player class that manages the ownership over the AI sprite. This class interacted with other classes in the main loop, delegating updates, drawing and collisions.

GameOver, Play and Initial:

These classes all implement the GameState interface and are used to cycle the various states of the game, being before game start, playing the game and game over. The play class being the largest provides the implementation for the main game in the game loop. This is the class that updates, draws and calculates the actual collisions that are owned by the player class.

Minimap:

The minimap class provides the implementation for the minimap. This class transposes player, powerup, wall and enemy positions into a map that is fixed to the upper left corner of the viewport.

FontText:

This class is a wrapper for the existing BitmapFont class in LibGDX. This class provides extra functionality that isn't immediately exposed by using a BitmapFont without using the accompanying font generator like text sizing, color and the centering of the font.

Health:

This class is consumed by all in game actors like players and enemies. This class manages the incrementing and decrementing of health, while providing some convenient apis for throttling damage and hp resets for respawning.

Entity:

This is a wrapper class for the existing sprite class. This class enables me to easily swap out sprites when I want to represent characters taking damage, computer characters center position, consume sprite sheets and expose the necessary parts of the sprite class.

Asset:

The asset class is a wrapper for textures, this exposes a convenient api to get offset in sprite sheets and measurements for the texture supplied.

GameWorldDeserializer:

This class parses a CSV file that has been exported by excel into a matrix that will be consumed by my foreground and background controller to populate the game world.

RoundController:

This class manages the incrementing and tracking of rounds.

Tank:

This is the abstract class in which the majority of the functionality of the tank asset is implemented. This includes how the tank should move, how the tank reacts to various collisions, drawing, updating and managing the tanks health. Much of the functionality is abstracted away in other objects contained within the tank class like the Health object, and BulletBag object. The tank class also orchestrates both the top and bottom (track and turret) tank parts, allowing them to exist together but move independently.

Panther and Sherman:

These are derived classes from the tank class. These simply provide some default value for speed, rotation speed, and sprite being used. This allows multiple tanks with different characteristics without having to rewrite the Tank class for each.

Zombie:

Much like the tank class the zombie class holds the functionality for Zombie assets should behave. This includes how zombies move, how they should be drawn, how they interact with collisions.

MeleeZombie:

The melee zombie is a derived class from zombie. This derived class provides the default stats, and sprite.

GameOverMenu, MainMenu:

These classes provide the implementation for the menus. This includes the functionality for buttons and assets that will be drawn to the screen.

MenuController:

This class controls the drawing and updating of the menus.

GameConstants:

This class provides default values that are referenced by various classes.

AIController, PowerUpController:

These classes aggregate their various objects like power ups and entities into a single array. This provides a nice way to loop over and update all the power ups, and enemies in one place. These controllers handle the interaction from the main game loop and the various items that these controllers' control.

MoveBoundary:

This class normalizes the movement values based upon whether the object that implements this class is touching other objects. This class prevents game entities from clipping into walls and other objects.

InputController:

The input controller listens for keydown events, then provides an api that offers whether specific keys are currently being pressed to the class that implements the input controller. This is used to capture the inputs that move the players around the map.

AudioManager, AssetManager:

These classes load and manage all the textures and sprites that are used in the game. Both of these classes are singletons which prevent the same assets from being loaded twice.

RoundCounter:

This class provides the overlay for the round counter which is displayed in between rounds.

BulletBag:

The BulletBag provides the means to manage bullets that individual characters own. This class provides means to switch bullets, cull bullets that are offscreen and curry collision objects to the bullets themselves.

Bullet:

This abstract class describes the behavior for bullets. Determining what should happen when bullets collide, how they move and how they should be drawn and updated.

Lazer, MachineGun:

These classes are derived from bullets. They provide default stats regarding how quickly the bullet should travel as well as the damage that each bullet should do. The sprite that represents the bullet is also passed in from here.

Power Up:

Much like bullets, this abstract class provides a unified way that all power ups should behave. This includes what happens when various actors touch them, drawing and updating.

PUHealth, PUMachineGun, PULazer:

These classes provide the implementation for each power up. This includes sprites and flags that will be consumed by classes the players that collide with these power ups.

ForegroundController, Background Controller:

These classes contain all the foreground/background objects. These classes handle looping over the array of objects, updating, drawing and removing objects as necessary.

Foreground:

This abstract class provides the basic implementation for Foreground objects. This includes managing the sprite, update, and drawing.

SolidWall, BreakableBox:

Both of these classes are derived from foreground and are the actual implementation of foreground objects. This includes supplying a default sprite received by the asset manager, and for the breakable box tracking the hp and managing whether it should be flagged for removal.

Life Counter:

This is a class that manages the life counter shown on the UI. This single class manages both player one and player twos life counters. This is accomplished by passing in a boolean that if true will draw white skulls on the left side for player one, and if it is false draw red skulls on the right side for player two.

TankGame:

This is the game container. This class initializes the game and owns all the asset/asset controllers. The actual rendering, and disposing is done by this class. This class also contains the GameState field, which cycles between various classes that implement the GameState interface. These classes contain the logic for the game before it starts, during the game playing, and when the game has ended.

SoundClip:

Much like how an entity wrapped the sprite, a sound clip wraps a sound object. This class exposes various methods like volume and a play method.

Background:

Similar to foreground objects these are square sprites that make up the map. Unlike foreground objects these only serve aesthetic purposes and have no collision or state beyond their location.

FloorUncracked, FloorCracked, FloorCrackedTwo:

These are various classes that represent the different tiles that can make up the background.

11) Class description for the Platformer: (From UML Diagram, Top to bottom, left to right)

Character:

The character class is an abstract class the acts as the base for all actors in the game. This class contains the move class, health class, animation manager class. The character class also contains a common state that is found in all actors, like collision properties, x and y position, and also handles removing sprites from the stage, providing default methods for collision interactions like collision with solid, collision with spring etc.

Moves:

This class contains the movement properties of the actor, and more specifically mange’s the velocity. This class exposes methods to the character like moveRight, moveLeft and jump.

Health:

This class manages the incrementing and decrementing of character health. This class also manages the damage timeout which acts like a throttle, only allowing a character to take damage every ½ second. This prevents a character that is touching a damage from immediately burning through their hp.

AnimationManager:

This is a class that controls the animation logic. This logic considers what direction the player is moving then setting the animation and sprite direction accordingly.

AudioManager:

This class contains all the audio clips. When an audio clip needs to be played this class exposes a method to play a specific clip and properly prevents specific audio from being played twice like the soundtrack.

Velocity:

This velocity class contains the velocity state, as well as methods to normalize the velocity which prevents it from exceeding the maximum velocity, and uses data provided by the move constraints class to prevent velocity from being added if the actor is being blocked from moving in a specific direction.

MoveConstriants:

This class holds the state that determines whether or not a character can move in a specific direction. This class also normalizes x movement, with data provided by velocity to update the characters' location.

Enemy:

This class extends the character class, and provides data like shouldJump, and shouldMoveLeft that an AIControler can consume to carry out logic. This class also exposes common enemy logic like followPlayer.

Mushroom:

This mushroom class implements the Enemy class, and simply provides a sprite, various sprite animation frames. This is a class that creates a mushroom enemy.

Ghost:

The ghost class is the class that represents the character that the player will use. This class contains a special resolve collision method that pertains specifically to collisions that apply to only the player, like touching coins, and the goal. This class also contains player specific state like score and reached goal.

Controller:

This controller class owns a character. This abstract class provides bindings for its derived classes to manipulate the character like moveRight, moveLeft. This class

also provides basic update and draw functionality that can be overridden in derived classes.

GroundEnemy:

This class is an implementation of a controller class. This class provides binding for an enemy that remains on the ground, and simply chases the player around.

Player:

The player class is an implementation of the controller class. This class is used to manipulate its character by using an inputController. This relays requests from the input controller, to the character. This class also exposes state that is relevant to the main game like the score and hasReachedGoal.

InputController:

The input controller listens for various keydown events and alters its state accordingly. This class is used to collect keyboard input from the player, which is then translated into state that is consumed by the Player class.

Foreground:

This abstract class contains general state and methods for foreground elements, like whether it should be removed, collisionProperties and methods to remove the foreground object from the stage, as well as abstract methods to help remove foreground objects from the stage, to draw foreground objects and to update them.

StaticForeground:

This abstract class is a derived class of foreground and represents foreground objects that are not animated. Since these foreground objects are simpler, do not need to be rerendered, and do not have an animation manager they are much simpler by only providing methods that draw, remove and provide collision properties of a foreground object.

Spike:

The spike is an implementation of a StaticForeground. This class provides a reference to a sprite, and collision properties that when a player collides with, kills the player.

DirtGround, Other Ground:

These are basic building blocks of the game world. These blocks simply provide a reference to a sprite to display and solid collision properties which opposes a player's movement.

Goal:

Like the previous two classes mentioned this class simply provides a sprite, and collision properties. The collision properties for this class enable the player to collide with a point that flags the player as completed the stage.

Animated Foreground:

This abstract class is derived from the foreground class and represents animated foreground objects. These objects contain an animated sprite that is exposed to its derived class. Similar to Staticforeground draw, collision and removal from stage methods are exposed here.

Coin:

This class provides a reference to a sprite, and logic to play the coin rotation. This class also contains collision properties that cause the score to be incremented when the player collides with this object.

Spring:

This class provides a reference to a sprite animation and contains logic for the spring to bounce when it comes into contact with a player. This class has collision properties such that when a player collides with it, it will send the player up in the same way as when a player lands on an enemy.

gameInit:

This function provides default values for the pixi app and appends a resize event listener to the window.

AnimatedEntity:

This class wraps Pixi’s animated sprite class. This class exposes specific methods I need like width, height and x and y coordinates.

initMainMenu, initGameOverMenu:

These functions create and append the menu nodes to the DOM. Both of these functions accept a call back function which is the main game. Both of these menus are animated and contain logic to prevent several games being started or restarted at the same time.

GameConstants:

This object contains game default values, I envisioned a much larger use for this object, but it wound up only containing constants for scale and gravity.

MainGame:

This function contains the “guts” of the game. That is the reference to the Pixi app, initializes the canvas, contains a reference to the sprite manager, audio manager, and all actors and controllers. This function also controls all the drawing and updating, as well as providing the main game loop.

ForegroundController:

This class aggregates all the foreground objects and parses a JSON object into various foreground objects. This class exposes methods to draw, update and remove all the foreground objects.

EnemyControllerAggregator:

Like the foreground controller this class aggregates all enemy controllers. This provides an easy way to draw, update, and remove all the characters. This class also parses a JSON object which creates enemies to be placed in the stage.

SpriteManager:

This class is in charge of loading and holding a reference to all the sprites in the game. This class exposes methods for other classes to hold references to the sprites as well.

checkCollision:

This function accepts collision objects, then compares them to decide if they are colliding, and if they are colliding which sides collided. This function uses an implementation of the Minkowski sum that I found online. I modified the algorithm to better suit the game's mechanics by prioritizing vertical collisions before horizontal collisions.

invertCollisionObj:

This function accepts a calculated collided object (the output of checkCollision) and inverts the collided object data so that the opposing collider can receive data about its collision, without having to rerun the collision in reverse. An example would be if a player and enemy collided and the player object was passed as the first argument in checkCollision, then checkCollision would return data about that collision from the players perspective. InvertCollision would accept the data about the collision from the

player's perspective and invert it so it would appear as if the collision had been calculated from the enemy's perspective. I.e., is a player landed on top of an enemy the player's collision object would have the bottomCollided field being true, but the enemy would have a topCollided field being true.

lazyUpdateCameraY:

This is a function that handles the camera tracking the player in the y direction. Having the camera fixed on the player feels jarring when jumping and landing so this class slows down the camera, creating a dead zone then zones radiating out where the camera will pick up speed to follow the character.

updateCameraX:

This function simply fixes the camera to the player in the x direction

12) Self-reflection on Development process during the term project:

I really developed a better understanding of the solid principles, and common design patterns like the Singleton and State pattern through the term project. Using overarching concepts like the importance of writing decoupled code, and programming to abstractions helped me write an application that was easily extendable and less brittle. In the tank game a judicial use of interfaces really helped highlight the points of contact between two classes, and further emphasized what kind of behavior was expected from specific classes. I found interfaces to be less useful in my TypeScript application because TypeScript oddly enough intentionally does not enforce the

signature of methods in interfaces as strictly as it would the actual implementations provided in a class. I felt like I was compromising some of my type safety using interfaces which really disincentivized me from using them like I had in the Tank Game.

Design patterns like the State pattern really helped organize my game loop in the Tank Game by reducing my dependency on conditionals peppered throughout my application to control the flow of state. Instead, I was able to define various classes representing various points of my applications life cycle, then use an interface which allowed other portions of my application to treat all of these classes as the same entity. I could then freely switch between these classes as the state changed providing a neat and organized way to manage the side effects of changes in state. This was useful in the start, play and end cycle for my game.

I used the Singleton pattern for my asset managers. This was a convenient way to guarantee that I would only have a single instance of my asset manager class at any time. This is important as loading and maintaining assets in memory is not an inexpensive operation. This provided a guard against other parts of my application generating their own asset managers which would lead to wasted memory, or potentially unexpected behavior.

Programming to an abstraction really helped me keep my code concise and extendable. A great example was my ability to create different bullets that behaved differently but be consumed as if they were the same entity. I accomplished this by using a bullet interface to expose common behavior. This allowed me to treat every bullet variation as the same damaging entity that could be stored and consumed by various parts of my application without the need to make major modifications. This

allowed me to add different types of bullets without having to rethink how various classes will consume the new type of bullets.

I feel like the term project really helped grow technical understanding of code design, as well as the actual implementation of patterns that are standard in the industry. I am looking forward to expanding my knowledge of design patterns, as well as gaining a more intuitive sense of when it is and is not appropriate to use them. I found that both of the projects were at times challenging but never insurmountable, and that both projects did a great job of stressing the importance of the concepts learned earlier this semester.

13) Project Conclusion:

I really enjoyed writing both the games for the term project. I felt like the requirements for the first game were clearly stated which made planning and designing much easier. I also felt like you provided enough skeleton code to lead us in the right direction. I did not end up using all of the skeleton code provided, because I used LibGDX, but I did reference bits and pieces of it like the tank turning code. The videos that went along with the tank game were also really helpful. Overall, the Tank Game was a great way to ease into the second game.

The second game was my favorite of the two to write. I really appreciated the flexibility offered with both the tech stack, and the game itself. Being able to work with libraries that I would not typically use in my day-to-day development like MatterJS and Pixi was very enjoyable. Sitting down and completely designing a game by myself, then working through that design was a great experience. The design processes really

helped me develop a better understanding of what one should think about before starting a larger project, as well as accounting for the flexibility you will inevitably need as new problems arise. This has been my favorite project assigned at SFSU so far.