CSC 413 Final Project Documentation

Spring 2020

Jennifer Finaldi

920290420

CSC413.03

github.com/csc413-03-spring2020/csc413-tankgame-jfinaldi32

github.com/csc413-03-spring2020/csc413-secondgame-jfinaldi32

Table of Contents

1 Introduction

1.1 Project Overview

This project involves two 2D videogames complete with a graphical user interface, each with its own unique theme, gameplay, and style. The idea was to create a code design and class hierarchy structure for the Tank Game, then try to use as much of that code as possible for the second game, requiring as much code abstraction as possible.

1.2 Introduction of the Tank Game (general idea)

This tank game was created as a player versus player battleground game in a 2 dimensional game environment with a top-down birdseye view. Each player controls a tank that can shoot an unlimited number of bullets, or heavy bullets if the player tank picks up the ammo upgrade. The object of the game is to either kill the other tank three times, or catch the stimulus package at the very center of the map. Once either of those have been accomplished, the game will end and display to the screen who won and who lost.

When the game begins, each half of the screen will display the view for a player tank. Each tank spawns at opposite ends of the map. As they use their key input to control their tank, they will move around the map, encountering obstacles such as walls, breakable crates, and turrets that are constantly firing bullets in different directions. The tanks must dodge enemy bullets, destroying turrets and crates to reveal new paths. The players can prioritize upgrades to their tanks, or getting to the center of the map. There are little hearts on the ground which regenerate lost hitpoints for the tank that picks it up. Ammo upgrade pickups will automatically assign a tank with 15 heavy bullets that do double damage and get fired automatically when the user shoots, until the ammo runs out.

A minimap for each user interface is displayed on the screen, showing a miniaturized version of the whole world map. Users can use this to find enemy tanks to travel to, or to find out where the stimulus is. Sound effects help signal tank progress in the game for both players, telling them who got a powerup and who died. When a tank dies, it will blow up and then automatically respawn at their original spawn location, with one life less than they had before. Once a player gets to zero lives, the other tank wins. Unless that player gets to the stimulus first. The stimulus is an automatic game win, so it will be difficult to get to.

1.3 Introduction of the Second Game (general idea)

Tower Defense is a 2D game designed and animated to look like a 3D third-person view real-time strategy game, heavily influenced by an original mod for Warcraft3 from the late 1990's that paved the way for the Defense Of the Ancients and League of Legends spinoff Multiplayer Online Battle Arena games that are popular today. It involves two bases at either end of the map, with one lane that goes from the Radiant base (humans), to the Dire base (orcs), with towers for each side lining the lane until both reach No Man's Land at the very middle of the lane path.

The player of this game controls a knight hero character named Lothar. Lothar spawns at the start of the game in front of the Radiant base, along with four npc creeps who will automatically run along the lane path, attacking the closest enemy or tower to them. The player controls Lothar to assist his creep wave in destroying all of the enemy towers and eventually the Dire Ancient base. Should all four creeps die along the way, not to worry, four creeps spawn in front of the Dire base every twenty seconds.

The Dire base spawns their own little orc creeps every twenty seconds as well, however since they do not by default have a hero on their side, the orc creeps are much stronger and will require the help of Lothar in order to defeat a wave. Each enemy creep or tower that falls within Lothar's vicinity will give him experience. Once he gets enough experience, he will gain a level, increasing his damage, speed, and hit points. Once Lothar reaches max level 6, he will be very strong and difficult for the orc creeps to defeat.

The game ends once the Radiant base or the Dire base falls.

2 Development Environment

2.1 Version of Java Used

Both games were developed using Java SDK 11.0.5. Language level for the Tank Game is 8, and the Tower Defense game uses Java 11.

2.2 IDE Used

IntelliJ IDEA 2019.3 for MacOS Catalina v10.15.3

2.3 Special Libraries/Resources

All libraries needed will be included in the source code, along with resources.

3 How to Build/Import your Project

3.1 Importing the Games

Build/Run both games using IntelliJ compiler:

1. Download and unzip the zip file from the master branch of each game
2. Open IntelliJ IDEA and select import project from existing sources.
3. From the directory that you downloaded the zip file to, select the root directory:

   Root for Tank Game: csc413-tankgame-jfinaldi32-master

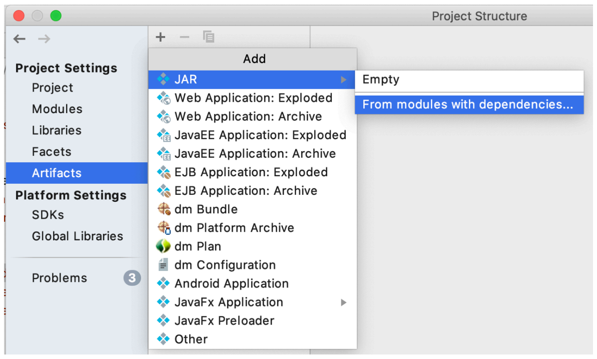
   Root for second game: TowerDefense
4. Use the default settings for all of the remaining prompts
5. Once in the project, go to File->Project Structure Settings. If you are in the Tank Game project, change the project language level to 8. If you are in the Tower Defense game, leave the language level at 11.
6. In the Project Structure settings, select the Modules tab, then select the resources folder and hit 'mark as Resource' then click 'apply' and close the Project Structure Settings.
7. In the Project Directory column in the left side of the IDE window, expand the root directory,

followed by the src folder. If you are in the Tank Game, you should see the Game class file. If you are in Tower Defense, you will need to expand the GameModifiers package directory to see the Game class file.

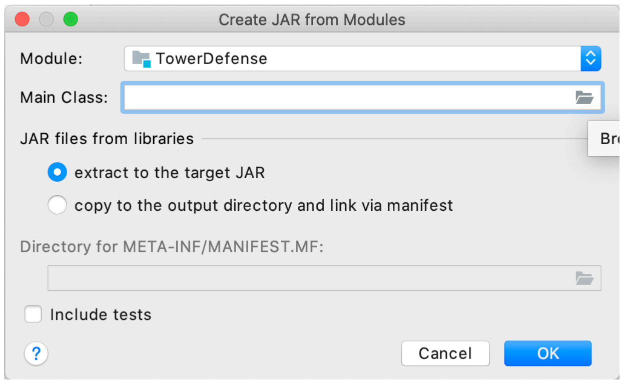
8. Right click the Game class and select Run 'Game.main()'

3.2 Building the JAR

1. Go to File->Project Structure Settings within IntelliJ. Select the 'Artifacts' settings in the left side list.
2. Click the '+', then select JAR in the drop down menu, then select From modules with dependencies:



3. One you see this window, click the folder in the right of the Main Class: text area, then select the Game class as your main class. Click Ok for both of these windows.


Create JAR from Modules

Module: TowerDefense

Main Class:

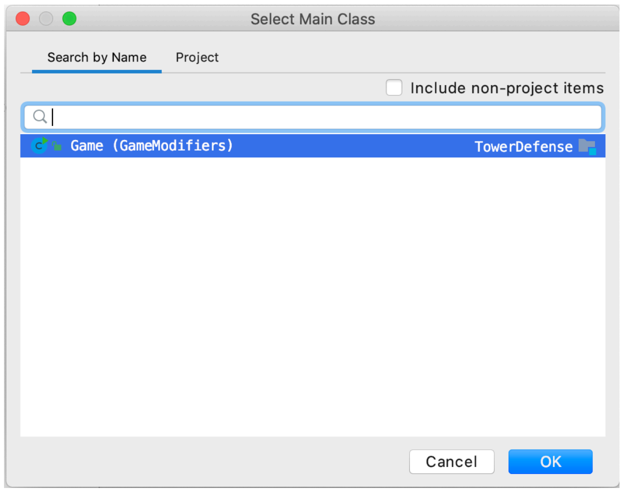
JAR files from libraries

[✓ extract to the target JAR]

copy to the output directory and link via manifest

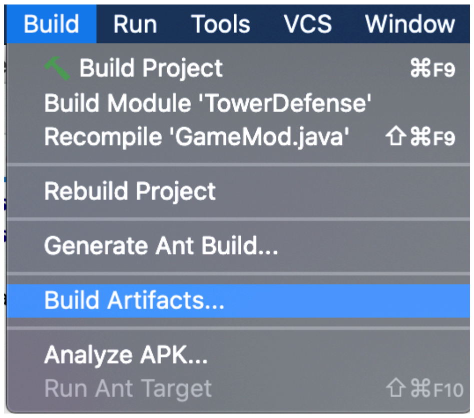
Directory for META-INF/MANIFEST.MF:

Include tests

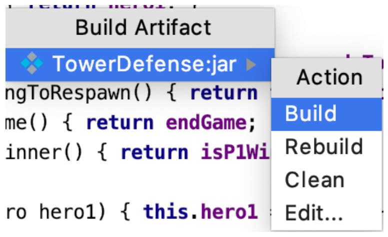


4. Click 'Apply' in the Project Structure Settings window, then close.

5. In the top menu, select Build->Build Artifacts...



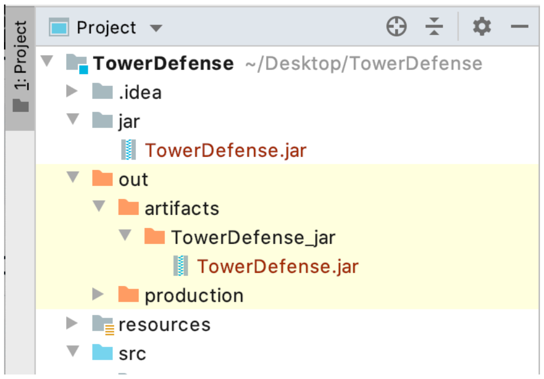
6. Once you see this little area pop up in the code area, select Build, just like you see here:



```
ngToRespawn() { return
me() { return endGame;
inner() { return isP1Wi

ro hero1) { this.hero1
```

7. The jar will build and generate an 'out' directory in the project directory as seen here. Expand 'out'->'artifacts'->[root directory name]_jar to reveal the jar file, named the same as the root directory:

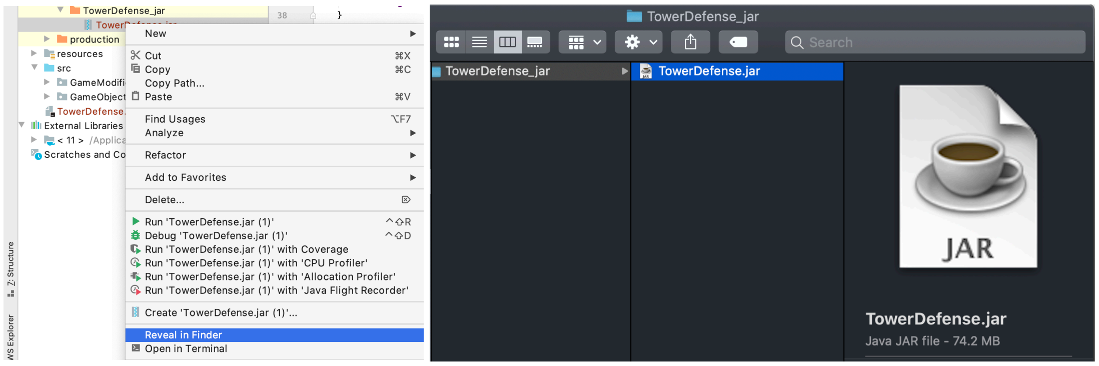


3.3 Running the Built JAR

Run both games using the jar file:

1. In the IntelliJ interface, expand the root directory in the Project directory column on the left, and then expand the jar directory to reveal the jar file that was included with the project.
2. Right click the jar file named TankWars.jar or TowerDefense.jar, select from the drop-down menu: Run 'TankWars.jar' [or Run 'TowerDefense.jar']

3. Alternatively you can right click the jar and select open in Finder (for those using a mac) and then run from there:



4 How to Run the Games/Controls

Tank Game-

1. When the game begins, full screen the game window to be able to view the entire screen and user interface.
2. The object of the game is to either kill the enemy tank three times, or to venture to the center of the map and capture the stimulus package to automatically win the game. Shoot the

turrets before they get a chance to shoot you! Break crates to create new paths to take, but if you die, you will respawn right back where you started from!

3. Player 1 (left screen) Controls:

   -W: move forward

   -S: move backward

   -A: rotate left

   -D: rotate right

   -[space]: shoot

4. Player 2 (right screen) Controls:

   -[up arrow]: move forward

   -[down arrow]: move backward

   -[left arrow]: turn left

   -[right arrow]: turn right

   -[backslash \]: shoot

5. Ammo powerups are found throughout the map. They will grand you 15 heavy bullets that do double damage. Once they run out, your tank will automatically shoot regular bullets.
6. Health powerups will give you hitpoints you lost, once picked up.
7. Good luck!

Tower Defense-

1. Welcome to the Tower Defense game! The object of this game is to destroy the enemy's base before they destroy yours!
2. Control your hero through point-click mouse movement in the game window area in the user interface. Just point and click on wherever on the ground you want your hero to go to and he will automatically run there and stop. Continue clicking on different spots to get him to go wherever you want him to go.
3. Every twenty seconds a group of four creeps (your allies) will spawn in front of your base, heading towards the enemy base. Follow them with your hero and assist them as they destroy the enemy creeps, towers, and hopefully the Dire Ancient! Be careful though, as your creeps are spawning, the enemy creeps are also spawning on the other side of the map, heading straight towards you and your base!
4. To attack a creep, tower, or base, hover your mouse cursor over the object and you will see a red circle appear on the ground around it. This means that you are ready to click to

attack it! Once you click, your hero will automatically run over to it and begin attacking. He will continue attacking either until the enemy dies, or you click somewhere else to move him. If an enemy runs to your hero and attacks him, he will auto-retaliate until you move him away from it.

5. The more enemies that die within your experience radius (including enemy towers), the more experience points your hero will gain. Your hero will use these points to level up automatically, increasing his hitpoints, speed, and damage per second.

6. You can monitor your hero's stats in the left side of the user interface. Your experience points will be displayed with a large blue bar that shows your xp progress till the next level. Once you reach max level at level 6, the experience bar will be a solid violet color.

7. To enable GOD MODE, click on Luthor's portrait in the user interface on the left side, beneath the minimap. Toggling this on will make your hero indestructable and give him a very fast run speed. To toggle GOD MODE off, simply click the portrait again.

8. The Dire (enemy) creeps will be very string, since they have to make up for not having a Hero for their side. Overcoming this at low levels will require some clever strategy, such as kiting them towards your towers or away from your Radiant creeps.

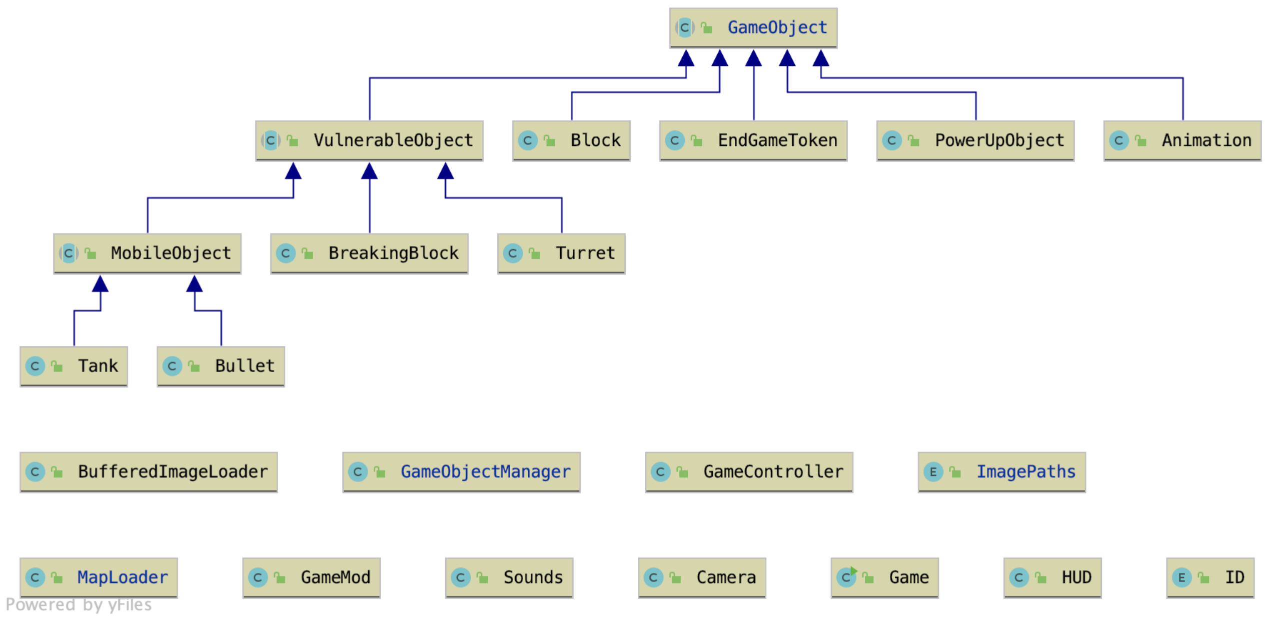
9. Good luck!

5 Assumptions Made Implementing Games

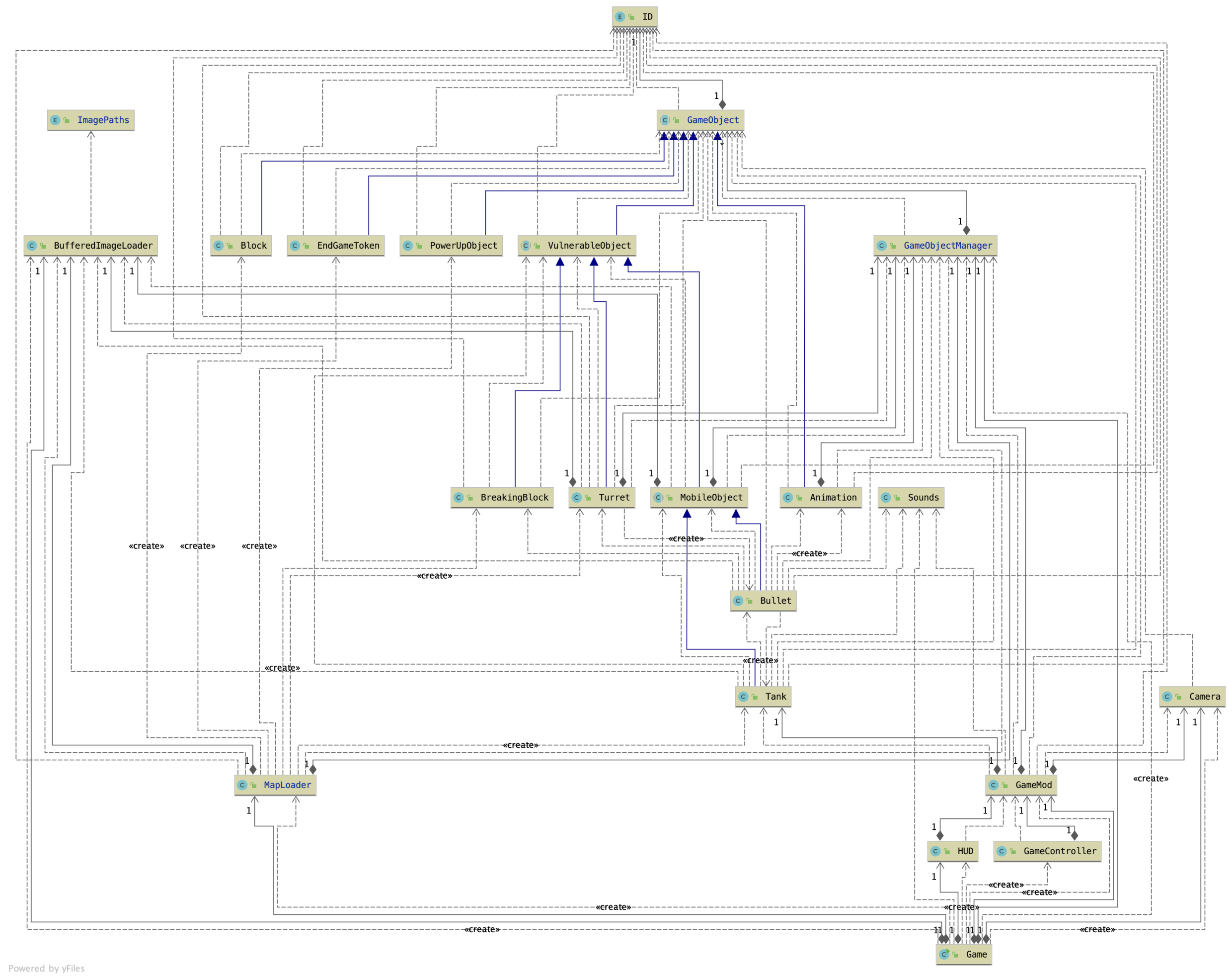
With both games it is assumed that the user is playing the game exactly according to the rules and not intentionally trying to break or overload the game. The user would have some familiarity with both 2D game mechanics, as well as real time strategy battleground games. The user would also already know that the game would be controlled with the keyboard or by the mouse, since it is assumed that the user had read the end-user documentation that describes how to play the game and what controls to use. It is also assumed that the person running the game has either IntelliJ IDEA or other IDE that can compile or run jar files, or the user has a computer that already has Java SDK native installed. I assumed that when running this program, the user will understand that once the game is finished and the winner/loser screens are output, that they can close the program and reopen it to play the game again.

6 Tank Game Class Diagrams

Class Hierarchy only:

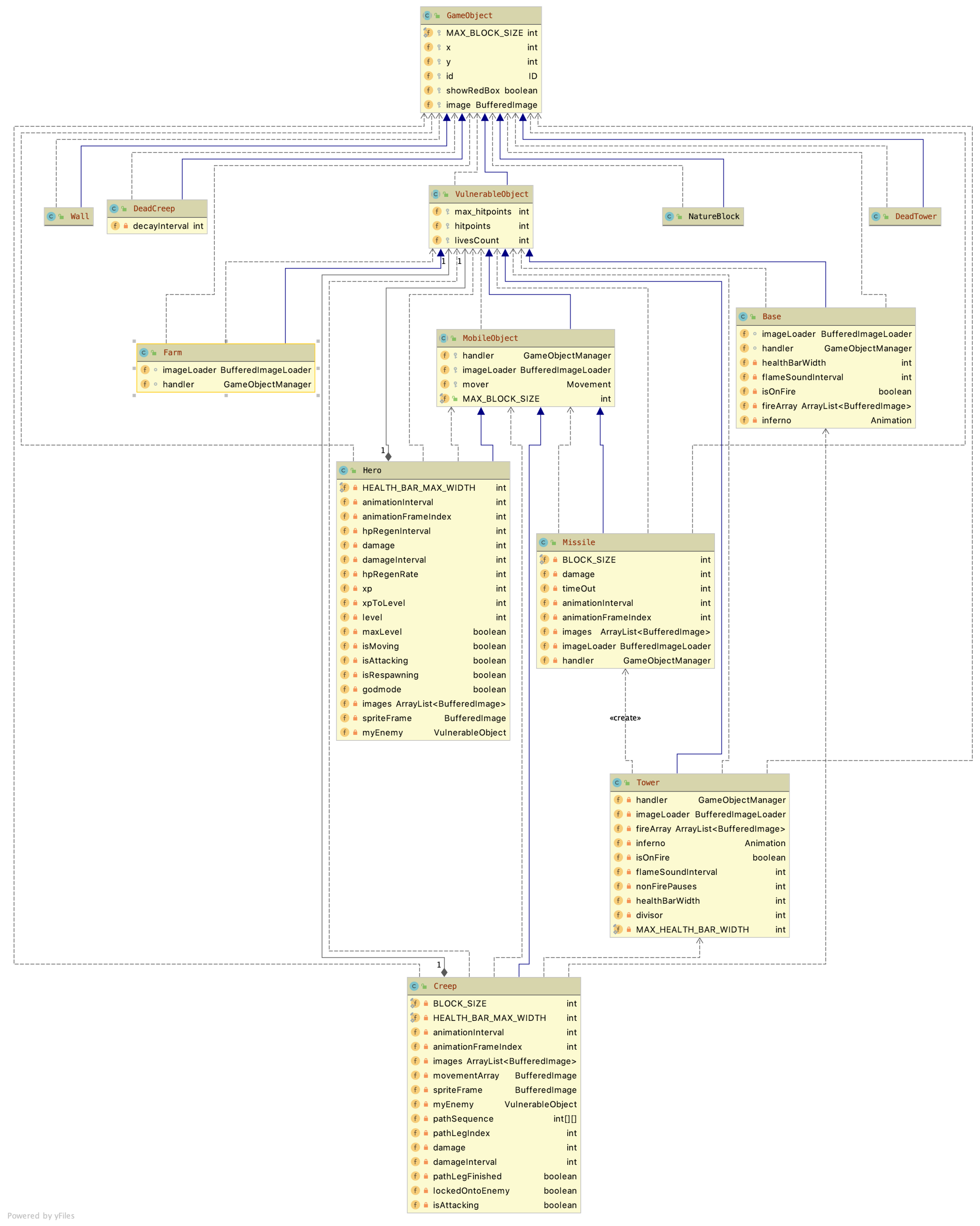


With dependencies:

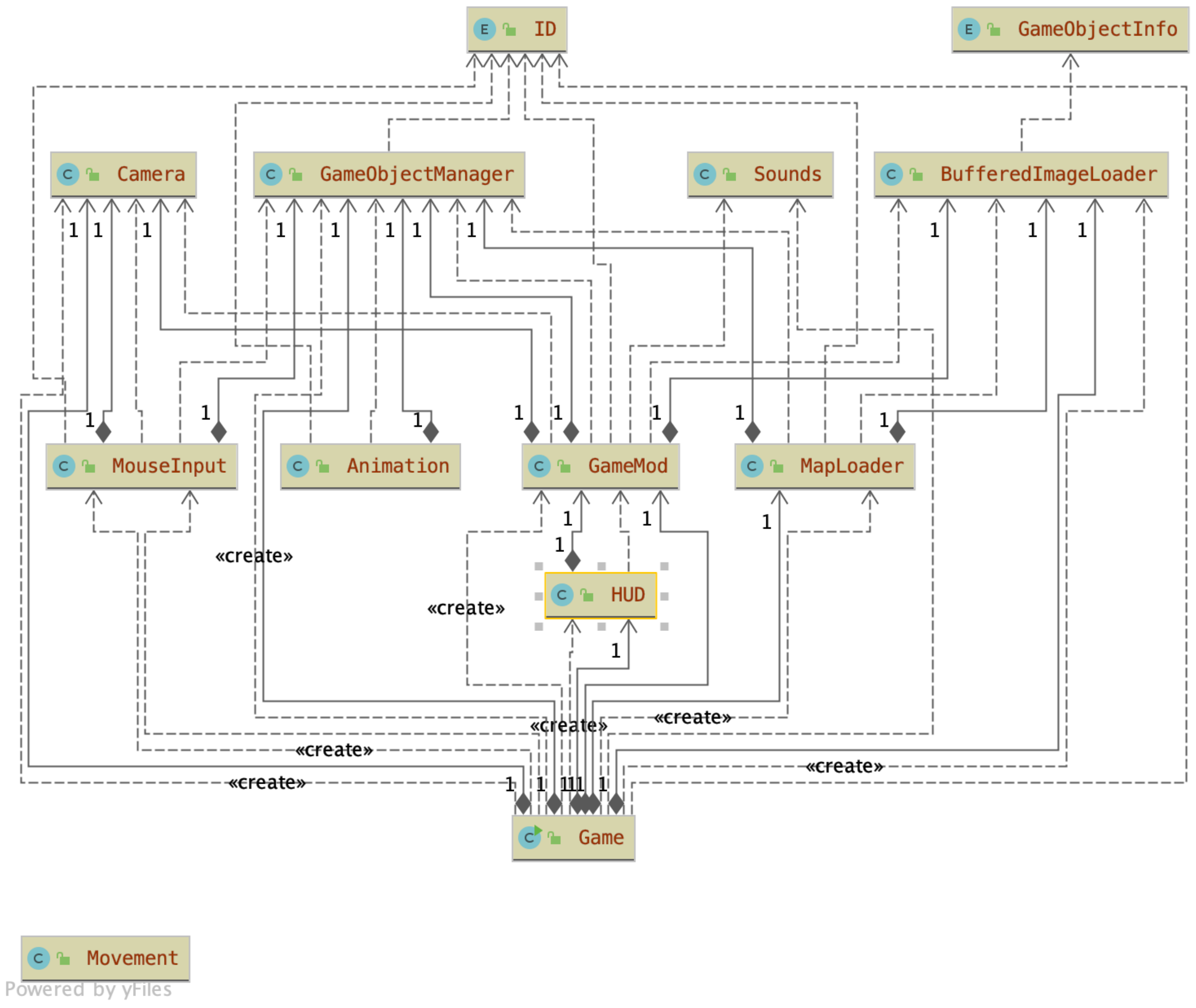

ID
ImagePaths
GameObject
BufferedImageLoader
Block
EndGameToken
PowerUpObject
VulnerableObject
GameObjectManager
BreakingBlock
Turret
MobileObject
Animation
Sounds
Bullet
Tank
Camera
MapLoader
GameMod
HUD
GameController
Game
«create»
1
Powered by yFiles

7 Second Game Class Diagrams

Package GameObject:



Package GameModifiers:



8 Description of Shared Classes

Animation-

The animation class handles the rendering of particular animations such as explosions. It receives an ArrayList of BufferedImage objects in its constructor, along with an interval at which each frame gets to be displayed on screen before being switched to the next frame of the animation. Typically, I use 30fps for most animations in these games. The animation object also takes the x, y coordinates of where it gets rendered on screen. Depending on what kind of animation it is, it will either terminate itself once the animation is complete, such as an explosion. Or for animations like fire, which don't expire, it will play the animation on a loop until the object that is on fire gets destroyed.

The general idea for having a class for Animations comes from codingmadesimple.com, though parts of the actual code were found in the comments here: youtu.be/BjQ9mMCZTHM, and were heavily modified.

Block (renamed to 'NatureBlock' in Tower Defense)-

Block objects are game objects that inherit directly from GameObject abstract base class. It represents an obstacle that cannot be broken or walked through. They are typically all 32x32 pixels in size, and can represent anything from a wall, to a tree, to a goldmine. All the Block class really does is create a block at a certain xy coordinate, render it, and return the bounds of that object for collision purposes.

BufferedImageLoader-

BufferedImageLoader handles all of the game images or graphics in the entire game. At the start of the game, an imageLoader object is created that creates a Hashmap of Strings/BufferedImage pairs that are pulled from the resources folder, using information stored in the ImagePaths/GameObjectInfo enumeration files. Once an image library is created, the program needs only to refer to the imageLoader object in order to retrieve an image resource. This imageLoader object is passed to most of the classes in the game, for convenient and easy retrieval, especially in situations when a tower falls and it needs to be converted into a dead tower object of rubble, for example.

The BufferedImageLoader class has a few functions aside from the constructor that reads the enum object pulling the info from it to load into the hashmap. It also contains a function to convert a sprite sheet into an ArrayList of single sprite images to be used to animations. This function takes a BufferedImage and a block size in order to calculate what size chunks to cut up the image into. This is particularly useful because it allows animated GameObjects to store sprite sheets as their main images, rather than a single sprite, giving each GameObject ownership over their sprite images.

Camera-

The Camera class is responsible for calibrating a camera viewport area that follows a player's character around the map. It keeps track of the player's character location and during each frame update, it will calculate where the anchor (upper left corner) x,y coordinates are, as well as the bottom right corner x,y coordinates. The camera's job is to follow the player sprite around the map, keeping the player in the middle of the screen, unless the character gets too close to the edges of the map. The camera will never extend further than the bounds of the world map. In order to do this, the Camera has to keep track of how big the world is, and then calculate the maximum and minimum x,y coordinates for both the anchor and the destination coordinates of the camera viewport. After performing all of the calculations it will feed the x,y coordinates to the Game class where it is used to display portions of the world map to the screen.

Game-

Game is the main class of the game, where all of the base functions happen. Game inherits from JPanel, which is a class that builds a graphics panel that gets put inside a JFrame object. This allows the use of functions such as paintComponent( ), and repaint( ), which trigger the sequence of painting graphics objects onto the panel during each frame rendering of the game.

The main function exists in the Game class, which first starts the game music, then starts the game loop, which runs as long as isRunning is true. It renders 60 frames per second, and was chosen to be this number based on a typical convention of older games and mobile apps typically running at this rate. It could have been set to a higher amount but to ensure that this game would run reliably on a wider variety of devices, it seemed the safer option.

The initializer function inside Game creates a number of objects that are the main staple of the game creation. This includes a GameObjectManager object that stores all of the game objects, a Camera object for one or two players, an imageLoader to load all the game images, a GameMod class that keeps track of the Game rule progress, a map loader object to load the game world, a HUD object to render the user interface, and a JFrame that creates the program window. The init function also adds any input control listeners such as MouseListener or Keyboard Listeners.

The Game class contains the main tick function that updates all of the game objects. It triggers the handler's tick function which updates all game objects, and it also updates the camera. Every time tick runs it checks to see if the game has reached the end of the game, so it can trigger endgame events accordingly.

The paintComponent function takes a graphics object that is responsible for drawing graphics to the screen. It will trigger other functions to perform tasks such as painting tiles on the floor of the world, followed by triggering the handler to render all of the game objects. It listens to if the gamestate has entered an endgame state. If so, it outputs the screen slightly differently than it does when the game is in an active state of gameplay.

Function getCameraSnaps observes whether or not we are in the endgame phase or not, and gets camera locations accordingly. Typically when the gameplay is active, it simply grabs the camera coordinates an relays them back to the Game class to be used for grabbing a specific subimages of the world map.

A function to draw the play screen handles the drawing of a specific player's point of view. For games that implement a split screen, the drawPlayScreen function will draw two separate halves of the game window, one for each player. For the TowerDefense game, drawPlayScreen will only draw one character's point of view, handling respawn countdown graphics. For both games, the drawPlayScreen function draws the graphics that tell the users who won and who lost the game.

For TowerDefense, an endGameEvent function is added to Game that keeps track of who won the game. This is what is triggered by drawPlayScreen when the game ends. It will output a particular graphic to the screen if the user wins, and a different one when the user loses the game. endGameEvent also triggers the running state of the game to false, so that the game loop halts.

The run function handles the game loop so that the main function does not have to. It contains a while loop that checks the run state of the game and if running is set to true, it will call the tick function and the repaint function, then sleep for 1000/60 milliseconds before repeating the loop. The run function also is what triggers the game music to begin playing, as well as calls the function to end game music.

<u>GameMod-</u>

GameMod is responsible for working together with the GameObjectManager object to alter certain GameObjects such as spawning player characters, updating their camera coordinates and the HUD for each player. The game mod acts like a conduit to avoid the Game class having direct access to

GameObjects. Gamemod also monitors the progress of the game, checking to see if the end of the game has been reached so it can flag for endgame events to occur and gameplay to end.

In the GameMod tick class, it calls a function to find the heroes in the GameObjectManager object so that it can always keep track of what the heroes are doing by creating shallow copies of them to keep in their data members for easy access and code that is easier to read. Doing this allows the GameMod to respawn heroes, and to pass hero stat information to the hud to be displayed to the user(s). In the TowerDefense game tick is altered to also spawn the creep waves at an interval of one wave every 20 seconds, spawned at the same coordinates every time. In Tank Wars, the tick function also monitors if either tank has obtained the stimulus object, as the game needs to be flagged for endgame if this occurs.

In TowerDefense, a function was added to count frames until a player is allowed to respawn. This slows down the game a bit, to discourage players from dying, creating a sense of urgency and more exciting gameplay experience. This also means that a function to respawn also needed to exist.

TowerDefense also contains a function named resetAttacker. This function will release an attacker from attacking an enemy that has already died. It cycles through the game objects until it finds either a Creep object or a Hero, then checks to see if that character was attacking the dead object passed into the function. If so, it sets their enemy being attacked to null, when it previously contained a shallow copy of their enemy.

TowerDefense required a function to check on the status of all creeps and towers to check if they have died. It cycles through the game objects, using their id attributes to identify them. It then checks their lives count and if it's less than one, the function adds a dead version of that object to the deadthings list, calls the resetAttacker function to release their attackers, then removes the object from the game objects list. It also triggers the Sound class to play the appropriate death sound for that object.

GameMod also has a function to set the endgame state flags, to be relayed back to the Game class.

<u>GameObject-</u>

GameObject is an abstract base class that starts the hierarchy structure for all game objects. It has data members for:

- x,y coordinates: where that object is located on the map

- id: the id of what type of object it is, according to the ID enum

- image: the bufferedimage sprite or sprite sheet for a particular object

and has abstract functions for:

- render: to be overrided to render an object

- getBounds: to get a rectangle of the dimensions of the game object for collision purposes

- tick: what each object needs to do to update itself

In TowerDefense, GameObject also contains a boolean data member called showRedBox, which flags true if the mouse motion listener detects that the mouse is hovering over that object. Since all game objects can be hovered over, it makes sense to have that boolean in the base class, to be accessed by any object that needs it.

<u>GameObjectManager-</u>

GameObjectManager stores all of the game objects in an ArrayList of GameObject objects. Whenever a new instance of a GameObject or child of GameObject gets created, it is stored in the GameObjectManager, to be retrieved whenever needed.

GameObjectManager contains a tick function, which goes through every object in the list, calling their respective tick functions. A function to delete an object off the list exists, as well as a function to add an object to the list.

<u>HUD-</u>

HUD is the heads up display for the user interface. It outputs a graphical respresentation of current game statistics to the user, such as the minimap, character name and portrait, lives count, hitpoints, experience points, damage, bullet ammunition, etc.

The HUD is one of the few classes that doesn't need a tick function because it doesn't have anything to update that isn't related to graphics. It is merely a relay of other object's characteristics. As such, it does contain a render function that handles the drawing of anything that isn't related to the gameplay screen. It uses the GameObjectManager to get camera information and information about our Tanks or Heroes to display to the screen.

<u>ID-</u>

ID is an enumeration of semi-generic GameObject type IDs. The benefit to assigning an enumerated object type id to every game object is that it cuts down on the processing related to using calls such as 'instanceof' when testing whether or not an object is of a certain class type. In addition to this, it fills a need that instanceof lacks, since games have two different sides and id can be used to test whether a tank is player1 or player2 or whether a creep is a Radiant creep or a Dire creep, since instanceof just tells us that an object is a Creep, but not which side the creep belongs to. Id is used a lot in testing collisions with different game objects, since a Hero will behave differently if it collides with an enemy Tower object than it would with a friendly Tower object.

As ID is merely a structure where data without fields are stored, it doesn't require any getter or setter functions and exists only to store a list of ID types.

<u>ImagePaths (renamed to 'GameObjectInfo' for Tower Defense)-</u>

ImagePaths, like ID is also an enumeration only that pertains to the storing of image paths that are used to retrieve image resources to be used by the BufferedImageLoader class. Because ImagePaths contain two fields, a string for a name of an image, and a string path to the image in resources, ImagePaths does require getter and setter functions to retrieve this data from the enum structure.

<u>MapLoader-</u>

MapLoader handles the loading of the entire game map environment. It is called at the beginning of the game, reading a bitmap image of the level design that gets parsed pixel by pixel, laying game objects in the same pattern as the bitmap image, according to what the rgb values of the particular pixel are. For example, if a pixel is pure green (0-red, 255-green, 0-blue) the map loader will create a specific game object, such as a wall, and store it in the GameObjectManager.

This bitmap parsing is exactly what happens in the loadLevel() function. It makes a note of the width and height of the bitmap image and multiplies that by the pre-determined block size (32pixels) to get the actual dimensions of the world map. This requires a bitmap level design to be created beforehand as a resource, which I did via msPaint. A black image was created in msPaint of the same number pixels as I wanted blocks for the map, then zoomed in, coloring each pixel a particular color according to which object I wanted to appear there. The rgb values of each color were recorded to be later entered into the if-statements in the loadLevel function. This allowed a streamlined approach to level generation, by creating a scalable level creation system. In order to add more maps to the game, it just requires some extra bitmap level designs to be added into the resource folders and the level loading would behave the exact same way, provided the rgb values always stand for the same types of objects.

In TowerDefense, a drawFloor function had to be added to be used instead of drawFloor in the Game class, because additional environment objects such as rocks and dirt patterning tiles added a bit more complexity to the base floor of the map, unlike Tank Wars, which just had the same grey tile repeated throughout.

<u>Sounds-</u>

Sounds class handles the loading of wav files from the resources folder, and playing of these files. It contains a function for playing game music on a loop for the entire duration of the game. Another version of this function was created as an abstract function for playing any game sound effect. Separate functions dedicated to each game sound effect call the generic function, passing in a string path to the function, where the resource can be loaded from. This allows convenience of other functions to call a sound effect by its actual name, while eliminating the need for repeating code, as we don't need to redefine a playSound function for each sound effect. The nice thing about the Sounds class is that because it is static it does not need to be instantiated, and its functions can be called from any class. The downside to this class is that these sounds are all handled asynchronously with their own thread, so if too many game sounds are created at the same time, stack overflow can occur. This was precisely the reason why creeps only make sound effects while attacking towers, instead of towers, Heroes, and each other.

<u>VulnerableObject-</u>

VulnerableObject is an abstract class that inherits directly from GameObject, adding another level of abstraction between GameObjects and derived classes that need to handle things such as hitpoints, lives count, and other functionality pertaining to an object that can take damage and die.

There are a few data members that VulnerableObjects have, such as hitpoints, max hitpoints, and lives count. Vulnerable objects also have a function defined to take an integer of points and add it to hitpoints. Passing in a negative integer results in damage taken, a positive integer results in health being regenerated. This function also tests if hitpoints dip below zero, in which it restores hitpoints to max hitpoints and then deducts one from lives count.

In Tower Defense, VulnerableObject objects contain an abstract function called vision( ) that returns an Ellipse2D.Double object representing an area around the object that can be used for experience points addition or for an object to detect enemies around them.

MobileObject-

MobileObject is another abstract class that inherits directly from VulnerableObject, representing objects that are vulnerable but also able to move around, separating Heroes, Creeps, and Tanks from Turrets and Towers which cannot move. Because these objects are mobile, they need to store information related to movement, such as vector x and y variables and angle the object is oriented.

In TowerDefense, these vector and angle variables are stored in an instance of Movement object, since character movement calculation is a lot more complex and requires more functions and game logic.

9 Description of Unique Tank Game Classes

BreakingBlock-

BreakingBlock is a child class that inherits from VulnerableObject. It represents an obstacle in the game that can be broken and removed to reveal a new path to travel. It takes a few hits to break a breaking block, only two hits if a tank has heavy ammo. This class is almost identical to Block, only it has a lives count of 1, and 40 hitpoints, as well as a different sprite image. The tick function doesn't require any implementation since the block never needs to be updated, it just sits there taking damage until it is destroyed. The render function and get bounds function are overridden with its own unique implementation which, again, is very similar to Block.

Bullet-

Bullet represents a projectile entity that collides and is destroyed. Bullet objects do not bounce so as soon as they collide with something they have completed their purpose and are no longer needed. The id of a bullet can have one of the following values:

- TurretBullet: bullet came from a turret and gets treated as such
- BulletP1: a normal bullet from player 1
- BulletP2: a normal bullet from player 2
- BulletP1Heavy: a heavy bullet from player 1 that does extra damage
- BulletP2Heavy: a heavy bullet from player 2 that does extra damage

Unlike other mobile objects, Bullet doesn't need to calculate vectors in the tick function, since it only needs to calculate trajectory once and then increment, so this can all be done inside the constructor. The tick function only needs to increment vx and vy.

Bullet needs to have a collision function, since there are many different game objects that a bullet can collide with, each requiring different logic. If any type of bullet collides with a block, it siply gets removed from the list of GameObjects. If a turret bullet collides with another turret, it shouldn't damage it. If a player1 bullet collides with player2 it should do damage, however a player1 bullet should not damage a player1 object. There are several ways to avoid a players spawning bullet from damaging itself, and that's the method that was chosen for this game. Bullets should always do damage against breaking blocks, and if a bullet does damage it should explode.

This is the reason a triggerExplosion function exists. It creates an explosion animation and depending on which Tank the bullet came from, it will either be a green explosion or a normal explosion, so triggerExplosion handles that logic. It also adds the new explosion animation object into the GameObjectManager, and triggers the Sound class to play an explosion sound, again which differs depending on which tank caused the explosion.

Bullet also overrides the render and getBounds functions from the abstract classes.

EndGameToken-

EndGameToken represents a game object that can be picked up to instantly win the game. The token is pretty difficult to get to, as it sits in the middle of the map, surrounded by turrets and blocks. The class is responsible for creating a basic GameObject and in retrospect it probably didn't need to exist as it's own class but rather an instance of Block, with a different id and sprite image, since it does almost exactly the same thing as a Block as far as game mechanics goes. Its meaning can just be interpreted based on the id. For all intents and purposes consider this class's functionality and definition nearly exactly like Block.

GameController-

GameController handles all of the keyboard input for the users. It implements a KeyListener interface to override the keyPressed and keyReleased functions. Whenever a user presses a key, the key code gets passed into the function and the variable can be used to decipher what logic to apply to that key. Because this class uses key input to control a tank, the gameMod object must be passed into the constructor for GameController.

In keyPressed, the function determines if any of the predetermined key bindings were activated, then if so, it either triggers the gameMod to apply changes to Tank1 or Tank2. These are setter functions for up, down, left, right, shoot, that feed a value of 'true' into them, as defined in the Tank class, where they accept a boolean value in the parameter.

In keyReleased, it behaves almost exactly the same way, only it listens if a key that was previously pressed was released. If the key is released, it has the gameMod object trigger the setter functions in Tank with a value of 'false', which should cease movement.

PowerUpObject-

PowerUpObject, much like EndGameToken, is defined almost identical to Block. It represents an object in the map that can be picked up to apply some type of modifier to a Tank object. It really could have been eliminated in favor of instantiating these objects using the Block class with a powerup object id, and corresponding sprite. Therefore, please see Block for class definition details, as they are the same.

<u>Tank-</u>

Tank is the class that represents a user playable Tank object. Tank extends MobileObject and thus inherits the vector and angle related members and functions. A Tank object can have an id of either Player1 or Player2. It handles the logic of tank gameplay functionality such as its movement, it overrides the render, tick, and getBounds functions, and it can shoot and handle collisions.

For tick, it checks if the boolean flags for up, down, left, right, shoot are true, then invokes their corresponding functions to toggle movement or shoot. Then it calls the collision function to check for the tank colliding with another GameObject.

In shoot, the function recalls the angle of the tank, setting it to the angle the bullet will fly. Then, depending on which id the Tank has, as well as if the tank has heavy ammo, it will spawn a bullet that coincides with those IDs.

In collision, the tanks bounds are compared to the bounds of every object in the GameObjectManager class. If a tank collides with another tank or immobile object such as a wall or a turret or breaking block, the function calls a collision helper function that corrects the position of the tank to be outside of the bounds of the object being collided into. If the tank collides with an object that can be picked up, such as the EndGameToken or PowerUp objects, it will apply the appropriate powerup object and then delete the object from the list of game objects.

There are also functions to move forwards, backwards, rotate left and rotate right, which do all the appropriate calculations to adjust the xy coordinates of the tank.

<u>Turret-</u>

Turret represents a stationary artillery object that shoots bullets in two different directions at a steady rate without stopping. There are two subtypes of turret, a left facing turret and a right facing turret. This needed to be differentiated because the angle of the turrets are calculated differently depending on if it faces right or left. The turret never shoots straight ahead, it shoots diagonal to its right and then pauses for 80 ticks/frames before rotating diagonal to its left and then repeating. Bullets spawned by turrets can harm both player tanks, creating an environmental obstacle in addition to pvp, making the game extra challenging. Players can shoot the turrets to destroy them, which can also add a coop option to the game, should players choose to team up against the turrets.

Turret inherits from VulnerableObject and as such has hitpoints and a lives count of one. In the tick function, it's only responsibility is to count the frames in between shots and once it gets to 80, it shoots a bullet.

The shoot function determines the angle at which the turret is shooting and then uses that along with the orientation (left or right) to feed that into a shoot helper function. The shoot helper function determines where the bullet should spawn, since it will be different according to which way the turret is angled, then it will spawn a bullet to be added to the GameObjectManager object.

The render function uses AffineTransform class to rotate the image of the turret depending on which way it is facing, then draws it to the Graphics object.

The getBounds function returns the dimensions of the turret object for collision purposes.

10 Description of Unique Second Game Classes

Base-

This class represents a base on either team that when attacked can take damage and be destroyed, ending the game. It inherits from VulnerableObject which handles all of its hitpoint adjustments and has a livescount of 1. It is responsible for handling all of the characteristics pertaining to having a base that can catch on fire if its hitpoints drop below half, and it keeps track with a boolean flag that is true if the building catches fire. If it's on fire, the tick function will count to 60 and then play a fire sound effect. If the object catches fire, it will create a fire Animation object called inferno that renders a fire over the object until it is destroyed, since a Base cannot recuperate hitpoints.

The render function outputs the health bar of the base which is green for the Radiant base, and red for the Dire base. It then renders the image of the base, which happens to be the largest object in the game. Finally, if the building is on fire, the render function then triggers the render function of the inferno animation object.

Creep-

The creep class handles all of the non-player character creeps that spawn in front of each base. Creep extends MobileObject, so it already inherits a Movement object that handles its movement for it. In addition to the movement class handling basic movements, the Creep class also establishes three sets of coordinates that represent a movement path. The first set of coordinates represents the first leg of a path that goes to a particular point on the map (essentially, they just follow the road). When the creeps reach that point, the class will change the destination to the next set of coordinates, and once those are reached, will set the last set of coordinates to right in front of the enemy base. A creep's journey along their path can be interrupted at any time if it detects an enemy within their aggravation radius. Should a creep detect more than one enemy, they will approach the enemy who is closest to them and lock onto that target.

A creep will attack only one enemy at a time. If they kill their opponent, the GameMod class will release it from attacking that object so it is free to detect another enemy to run and attack or, if there are no enemies within their vicinity, they will go back to traveling their predetermined path sequence.

A wave of creeps consists of four creeps who spawn in front of their base. Creeps do not move very fast, and to give the illusion of independent movement and personality, each iteration of the tick function will recalculate their speed randomly, so that they speed up and slow down ever so slightly. This keeps them from looking robotic and promotes a more fluid and dynamic look to the wave.

In order to balance the game and keep it challenging, the Dire creeps are a lot stronger than the radiant creeps, since they do not have a Hero to help them. As such, their damage and hitpoints a multiplied by a factor of four at the time they are constructed.

In addition to randomly calculating speed, the tick function grabs information from the Movement object to determine how often to change the animation sprite to give the illusion the creep is running or attacking. The tick function also calls the tick function in Movement to recalculate the trajectory of the creep. It also checks if a creep has reached the end of their path leg, so it can switch to the next. If a creep is attacking, the tick function calls a continueAttack function, otherwise it will attempt to detect enemies and detect collision.

The Creep class requires a vision function that returns an Ellipse2D.Double object that acts as a large radius around the creep, to test if any other game objects get near the character. The detectAggro function uses this vision function to test if GameObjects get too close to the creep. If they do, the creep will execute behavior in accordance. If a creep detects an enemy, it will lock onto the enemy and then attempt to run to it. If that enemy leaves the creep's vision radius, the creep will "lose interest" and try to find another enemy, or continue on their path.

In the collision class, the creep checks to see if it collides with the bounds of another game object in the list. If it collides with a friendly creep, it does nothing, as keeping the AI simple means that creeps are able to move through each other. If a creep collides with any enemy though, it will create a shallow copy of that enemy object to store in their enemy field. It will deal damage every 20 frames (1/3 of a second) until that enemy dies or the creep dies. Creeps will only make sound effects for attacking if they are hitting an enemy tower or enemy base.

The pullAnimationSheet function receives a BufferedImage of either a walk sprite sheet or an attack sprite sheet containing 8 columns and 5 rows of creep images. It then grabs the angle of the creep from the Movement object to find out which way it is facing. Depending on the angle, determines which movement array to pull from the sprite sheet. It then returns an ArrayList of five BufferedImages to be displayed in their movement animation.

DeadCreep-

The DeadCreep class builds an empty creep object that represents a corpse on the battlefield. It replaces a formerly alive creep with a dead, non-functioning one that can only be rendered for 10 seconds, before disappearing and being destroyed. The reason this had to be its own class is because it seemed better to separate alive objects with dead ones, keeping them in their own ArrayList inside of the GameObjectManager class. That would allow them to all be rendered before the GameObjects list, so that any living Creeps and Heroes would always be walking on top of them, instead of beneath them. This is a workaround when attempting to give a 2D game a 3D look.

As such, the DeadCreep class just renders a dead creep sprite on the floor and then disappears after a period of time.

DeadTower-

Like the DeadCreep class, the DeadTower class builds a destroyed tower image where the tower used to be before it burned up and collapsed in a firey inferno. Unlike the dead creep, however, the dead tower does not disappear and stays on the map for the rest of the game in order for players to easily keep track of their progress, and to add a little more realistic feel to the gameplay experience. Dead towers do not have collision, and only get rendered, and instances of this class are also stored in the dead objects ArrayList in the GameObjectManager, to be rendered before everything else so that they appear to be rubble on the ground to be walked over.

Farm-

Farm class represents a building near a base that originally is designed to take damage, catch fire, and then be destroyed. It does none of that functionality in this version of the game but can be upgraded later on to do all those things, plus provide experience points to the hero that helps destroy it. It can be a good way to add other options for a hero to gain levels without having to attack a creep or a tower. If a currency system were introduced into the game, a farm could give gold periodically to the Hero to which it belongs, or it can give gold upon being destroyed.

Right now, the farm exists only for environmental aesthetic purposes only and as such only overloads the render and getBounds functions, since a farm shouldn't be walked over by the hero.

Hero-

The Hero class blueprints the creation of a player-controlled object, handling behaviors such as collision detection, triggering the movement object to update, rendering the animation of character movement, and rendering the health bar above the character's head.

The tick function first tests if godmode is enabled to which it fills up max hitpoints if true. The tick function also triggers a small amount of hitpoint regeneration every 180 ticks (2 seconds). It checks to see if the hero has exceeded the threshold for experience points, to which it triggers a level increase. It then updates the xy coordinates by feeding the current coords into the movement object's tick function, which (unlike the other gameobject tick functions) actually returns an array of integer xy coordinates. The tick function also triggers animation frame changes, as well as the continueAttack and collision functions.

The collision function within Hero behaves almost exactly like the Creep class, only the logic of what to do in a collision is limited to whether it collides with a piece of nature, a building, or enemy tower and creep.

The Hero class also contains a pullAnimationSheet class that creates an array of buffered images to display as an animation, depending on what angle the hero is facing.

A levelUp function handles the increase of the player's level, maxing the hitpoints, and checking to see if max level has been reached. It increases the damage dealt by the hero, along with the max hitpoints, move speed, and hitpoint regeneration rate.

The render function draws the current animation frame of the hero on the graphics object, and it also draws the health bar of the hero above its head.

Hero is unique in that it is the only game object that deals with experience points, so in addition to getBounds, hero also needs a function named xpRange that returns an Ellipse area around the hero for the program to check whether an enemy has died inside this area so the gameMod can increase the xp of the hero accordingly.

Missile-

The Missile class builds a projectile object that acts like a fireball flying from a tower in the game. It inherits from MobileObject, gaining a movement handling object. Like other moving objects in the game, missiles are also animated as they travel, getting smaller as they travel further from the tower, to simulate a fireball falling towards the ground. A missile can have one of two ids: DireFireBall or RadiantFireBall, to help distinguish which characters a fireball should harm, since friendly-fire is not possible in this game.

The tick function for missile updates the xy coordinates during its path of travel, and then updates the animation frame, if applicable. If the missile misses its target an explosion is triggered and the missile is removed from the list of game objects.

The collision function checks to see if the bounds of the missile intersects with the bounds of a player or a creep. Tower projectiles do not harm other towers. If a missile makes contact with an appropriate enemy the collision function will play a fire hit sound effect, trigger adjustment of hitpoints in the enemy object, and then remove the missile object from the list of game objects.

Because the missile gets animated as it travels to its target, there needs to be a function to pull the correct animation array depending on which angle the fireball is traveling. This function behaves the same as the ones in Creep and Hero.

A triggerExplosion function exists to create an explosion animation array and feed it into a new animation object which then gets added to the list of all game objects.

In addition, since Missile is a GameObject, it also overrides the render, getBounds, and vision functions.

Tower-

The Tower class is very similar to Tank Wars' Turret class, but it needed enough extra functionality that it ended up being unique to Tower Defense rather than both games. Tower inherits from VulnerableObject so it inherits hitpoint management methods, as well as a lives count manager. The instances of Tower can hold one of two ids, either RadiantTower or DireTower, since a tower will need to automatically shoot enemies in their vicinity, but will never shoot a friendly ally.

The tick function calls the detectEnemies function, then animates a fire animation over the tower if it's hitpoints are low enough.

The detectEnemies function uses the Towers vision function to cycle through the list of all game objects, testing if those objects are in the tower's vision radius and if that game object is an enemy. If so, it fires a missile at that enemy, using the enemy's xy coordinates as the missile's destination. The tower will only shoot a missile every two seconds, repeatedly, at any enemy that hangs around inside the tower's vision radius.

The render function is overridden to display the tower, a health bar above the tower, trigger the fire animation object's render function, and if the player is hovering their cursor over the tower, the red circle will also be rendered around the tower on the ground.

<u>Wall-</u>

The Wall class represents an indestructable wall object that acts like an obstacle the Hero cannot pass through. It's a castle wall that helps protect the enemy base from sneak attacks from the Hero. It inherits directly from GameObject and only overrides the render and getBounds functions. The Wall class behaves exactly like the Nature class, and as such, could have been eliminated in favor of a nature block blueprint just with different id.

<u>MouseInput-</u>

Much like the GameController class for Tank Wars, the MouseInput class handles the receiving of all user input throughout the game, only it pertains to mouse input instead of keyboard. Because the user interface should not recognize clicks outside of the gameplay window (except for toggling god mode), the camera object needs to be passed into the MouseInput class to use its xy coordinates to offset the xy coordinates that are recorded by the mouse listener in the JFrame object in Game. Also, because the MouseInput class needs to recognize if a tower or base was hovered over, a GameObjectManager object needs to be passed in so it can use the list of game objects to recognize a click within their bounds.

The class inherits from MouseInputAdapter, allowing for the overriding of mousePressed, and mouseMoved functions. in mousePressed, the xy coordinates of the click location are passed in and the function first checks if the xy coordinates happen to be on the user interface where the hero portrait lies. If so, it activates god mode. If not, the camera coordinates are used to offset the click xy coordinates, before cycling through the list of game objects to move the hero.

The mouseMoved function listens for movement of the mouse cursor over the whole game window. If the cursor is recorded to be over a tower or a base, the function determines which object it is, then triggers that object to display the red circle around it when it is rendered.

<u>Movement-</u>

The Movement class handles all of the math operations pertaining to vector calculations for movement for all mobile objects. It takes in x,y starting coordinates from its moveable character, calculates its new coordinates, then gives them back to the object. Since Movement is not a game object it is not part of the hierarchy, as it only acts as a game object modifier.

The tick function is different than all of the other tick functions inside the game. Since it is not a game object, the tick function does not have to behave in the same way and thus takes in parameters for xy coordinates from its object, then returns the new xy coordinates after performing some calculations.

The setAngle function uses trigonometric operations to determine the angle that the object is facing. It calculates the slope and uses the start and end coordinates to calculate the inverse tangent in degrees. It uses a helper function called getDestinationQuadrant that determines which quadrant in the xy axis (centered on the moving game object) that the destination coordinates lie in, helping determine negative movement vectors when needed.

The moveForwards function uses the angle and the start/end coordinates to calculate the distance needed to travel, using the Pythagorean Theorem. It moves the xy coordinates based on these calculations and then tests if the distance is close enough to the destination coordinates to trigger the end of movement calculations.

11 Reflection on Development Process

This was my first time creating a game with an OOP structure so it was a little overwhelming at first, especially when trying to design a hierarchy system. I knew that no matter what I came up with for the design was going to be almost nothing like what I ended up with by the end. Thankfully the starter code provided for the Tank game gave me a little insight into the very basic idea of how a game loop runs, and the idea that each frame of a game is like a slide in a slide show that gets shown at a fraction of a second, giving the illusion of animation.

Despite this, I still had to consult some outside sources to get a better idea of how a game program should be implemented, watching various tutorial videos on the subject of 2D game animation and behaviors. This gave me a better idea on how to proceed with the program. It gave me the idea for my collision and render function structure and location, as well as the GameObjectManager class system, which ended up being very helpful for understanding the management of all game objects. 9

Learning how to use APIs that handle the building of a Graphical User Interface was a big challenge, as I had never created a game with a GUI before, and had no idea where to even begin. JFrame and JPanel and BufferedImage and Graphics2D were strangers at the start, but familiar friends by the end of development on TowerDefense. By the end of the tank game, I felt comfortable enough using image generation methods that I decided to step up the graphics for Tower Defense by making the sprites animate as they moved along the screen.

There are a number of things I wish I had done differently as far as class structuring goes. First of all, there was a fair bit of hard coding in my Tank Game classes that should have been abstracted, and made a little bit more work for the second game. I should have made the Animation class a bit more developed, perhaps even making it similar to the handler, where it would store all of the animations that should be rendered independently from game objects. Animation class should have also stored all of the animation handling in the second game, instead of having three different classes all have the same identical pullAnimationSheet method. ImagePaths/GameObjectInfo probably could have been stored inside of BufferedImageLoader, since that is the only class that actually uses that enum object. I also can't help but feel like my Game class knows way too much about the other classes and should probably have added another layer of encapsulation around it, like an interface perhaps.

For TankWars, I am pretty happy with most of the structure of the game, but I could have made classes like HUD a bit more modular as far as methods go, as well as abstracting more of the variables to scale based on overall world width, etc. As it was, it was very difficult to port HUD over to the second game because all of the location variables needed to be recalibrated. I also could have added a Collision class that stored all of the game's possible collision events. It could have been a static class like Sounds, triggering a particular event whenever a Tank collided with a wall or a bullet collided with a turret, for example. This would cut down on a lot of repetitive code found in collision methods throughout the program, and allow an easier time modifying code pertaining collision behavior. The game objects that can be picked up throughout the map really should have been just instances of a Block object and did not need to have their own classes, since they behaved exactly the same as a block in the game, only they could disappear. Last of all, GameMod should have handled the powerups in the game, rather than the collision function in each of the game object classes. This was knowledge that the gamemod was built for, and thus was not utilized to its full potential.

For TowerDefense, there are a number of problems with this game. The AI for creep waves is mostly pretty good. They attack when they find an enemy, they stop attacking when they defeated an opponent, and they resume moving towards their goal when not attacking. In other words, the creeps are always staying busy doing something. The pathing is flawed, however, as the creeps will sometimes skip a path leg and start running through the trees straight to the base before getting to the corner of the road first. I haven't figured out why some of the creeps do this while others don't, but more testing needs to be done to figure out the nuances of this AI system. Another thing I am not happy with regarding creeps is that they don't have any collision handling other than with enemies. This means the creeps can run through walls, trees, other friendly creeps, friendly towers and friendly buildings. The reason this was left out is to keep the creep AI as simple as I could, while still having them perform all of the necessary behaviors. I wasn't able to implement rerouting with the creeps if they collided with one another or an obstacle, as it would have involved a lot of complex graph algorithms such as A* or Dykstras shortest path, and seeing as complicated AI behavior like this would have been overkill and taken up much needed time required to build more important code, it was decided to be left out and so creeps remain collision-less with friendly units and buildings. As mentioned before, the Farm and Base classes behaved exactly the same so I should have just combined them into one class named Building, differentiating them with their id attribute. At the time I created these classes I wanted code that was easier to read in the MapLoader class, so I left it in, but looking back on it, it was not the best choice.

Knowing what I know now, I would have designed most of the game object hierarchy the same, but I would have changed some of the methods they contain to make them more abstract. The game modifier classes would have been structured differently, with additional classes being added to handle things like animation or collision events. Right now I am not sure that either of these programs would be easily scalable or easily modified by someone else or a team, so I could always be better about that, and this project could always be improved in the architecture.

Extra features would have been nice to add into the game, such as a spash screen on start-up, a second screen that displayed the player controls to the user, and then the option for the user to control the state of the game such as start, end, and pause.

Despite the program's shortcomings, I learned a massive amount about game design, gui animation, and 2D animation with a 3D look. I also learned how to implement keyboard and mouse listeners and how to handle user input in a GUI environment. And game programming aside, I also learned quite a bit about graphic design in the game development world, as I had to use paint to create my own custom level designs with rgb bitmaps, and I also had to create sprite animation sheets by manually cutting out images and pasting them into a transparent sheet using an image editing software. One of the most valuable skills I took away from this project was the method in which an image can be parsed manually, through nested for-loops for x and y coordinates of pixels, analyzing the rgb values of each pixel. I can see that knowledge coming in handy if I were ever to write software that handles image processing.

12 Project Conclusion

Both programs, though there is always room for improvement, behave as intended and are complete working videogames. They contain a game phase and an endgame phase, which communicate to the player the status of the game and what is going on. The GUI is implemented in a way that should be user friendly, and the controls behave in a manner that is consistent with gameplay. The game starts when it is supposed to, the sounds play when and as they should, and the game ends when it is supposed to end, telling the user who won and who lost. There are some small bugs with the second game's AI, but despite that the game is still balanced and challenging.