Sacramento State University

Tower Defense - Final Report

Release: 1.0

May 14, 2020

Logan Hollmer & Quinn Roemer

CSC 133

Dr. Daryl Posnett

Premise:

Space Force Commander is the idea that Logan and Quinn came up with and implemented as a tower defense game. This document will discuss many aspects of the game from gameplay to specific class implementation. However, due to the long nature of this report we felt it appropriate to give a high level explanation of some of the features of our game. All of these can be read about in full detail in the following paragraphs. Some of the highlights of our game include: having 3 unique towers that can be upgraded, a projectile that will track it's target, 3 types of enemies which are all animated, a start screen, 3 different levels with unique paths and backgrounds, curved paths that the aliens follow, both a pause and a speed up button, sound effects for the towers and enemy deaths (including a mute button), and a place to read the rules. To implement these features we used many of the different techniques that we learned over the course of the semester, these include: using the factory pattern, builder pattern, observer pattern, model-view-controller, strategies, collision detection, creating an object pool, and making use of OPP principles. Please note, our class names will be italicized in this report.

Game Play:

The gameplay is similar to many tower defense games but with a few twists. The goal of the game is to survive 5 waves of alien cyborgs who are trying to reach the player's base. If 20 aliens reach the player's base before the 5 waves are over, then the player loses the game. The player has 3 options for towers to defend their base. They can place a plasma turret for 50 dollars, these have a high rate of fire, but low damage and range. They can place a laser tower for 100 dollars, these are a well rounded tower, dealing medium damage with an average rate of fire. Finally there is the rocket tower for 200 dollars. These deal massive damage and the rockets will track their targets, however this tower fires extremely slowly. Towers can also be upgraded once they are bought. Everytime a tower is upgraded the cost for the next upgrade increases. The cost of the towers and the upgrades force the player to manage their money. The player starts with 100 dollars and can earn more by killing the cyborgs. The stronger the cyborg the more money the player will earn from killing it. The cyborgs come in 3 types. Drones are the weakest and most common of the cyborgs. However, they are fast and can even out-run the rockets. Soldiers are the standard infantry unit. They possess an average health pool and move at a reasonable speed. Then there are the behemoths. These are highly armored cyborgs with huge amounts of health. However, they pay for this with their slow movement. The cyborgs are smart though, every wave they increase their resistance to one of the 3 damage types (plasma, laser, or rockets) based off of which one deals the most damage. So, if a player only buys towers of one type they will quickly be overwhelmed by the hyper resistant cyborgs.

Gameclasses:

There are six main game classes in our app. These being *GameActivity*, *GameEngine*, *Gameworld*, *GameView*, *GameObject*, and *GameMap*. We followed the Model-View-Controller pattern when designing the internal workings of our game. *GameActivity* is the class that starts up the game during initial program execution. Its code was modelled after prior projects in the Pakt book. *GameEngine* (the controller) holds the game loop. Inside this class we call updates (when an update is required) and draw using *GameView* (the view). This class also manages our thread while passing touch events off to our input handler. *GameWorld* (the model) is used to hold all our game variables. This includes lists for our alien enemies, projectiles, and towers. This class also holds many Boolean values that outline the current state of the game. For example, if the game is paused, a Boolean flag is set so that other classes know what is happening to the game state. *GameView* is our handler for drawing things on the screen. Instead of having to call many separate draw methods each time we update the screen, we can instead call the single draw method inside of *GameView*. *GameView* holds private Canvas variables to simplify the process of drawing onto the screen. *GameObject* is an abstract class that we use as a base for many of the objects in our game (enemies, turrets, and projectiles). Lastly, *GameMap* is used to define the background of each level. When a level is chosen, by the user on the menu, the associated level background and pathing/spawning is loaded into *GameMap*. This class is also responsible for converting our path into an array of points used by the enemies to move, and returning hitboxes for things like the base, and the path. This class further simplifies the process of drawing the games path and backgrounds.

User Interface/Event Handling:

User input is a huge part of any game, and this game is no exception. With all the different options available, we needed several different classes to help prompt and handle user input. To give the user the options they required we created 3 classes. There is a *HUD* class which displays the buttons and information while a level is running. This also has an object of type *InfoContainer*, which holds information about a current game object. The player can select different game objects to see information about them. For example, if a tower is selected, then that tower is passed to *InfoContainer* and it displays and formats the appropriate information. It also gives a button for the tower to be upgraded if the player wants to. The aliens can also be selected to see their resistances, or when a tower is selected to purchase, it displays the tower's stats before the player places it. *InfoContainer* is one area where we used overloading, to set the information there is one method name, but depending on whether a tower or alien is passed, a different method gets executed. We also have *StartScreen* which handles all the information of the “main” or “start” page. This includes displaying the rules and level select. To handle all of this input, we designed a *UIController* to decide what to do on an input event. The *UIController* also helps check for bad inputs and displays error messages. For example if a player tries to place a tower when they do not have enough money, it will display a message saying that there is not enough money. The *UIController* uses an observer pattern to listen for the user input.

Sound:

We approached sound in a similar way to the last project (Snake game). We used a strategy pattern to select from 3 different strategies. Our sound could either be *PreLollipop*, *PostLollipop*, or *SilentStrategy*. The lollipop strategies were for dealing with the different Android API's. The silent strategy was used to mute our game if the user wanted to turn the sound off. We decided to implement 4 sound effects, one for each of the three towers and one for the alien explosion. See the paragraph at the end for a discussion of our assets.

Towers/projectiles:

The towers in our game used a factory pattern to be created. When the game needs a new tower it can simply call the factory pattern *TowerFactory* and pass the enum of the tower it wants. This is very helpful as the tower and projectiles are a complex part of this game. Each tower has its own class and so does every projectile. This is due to the unique way each tower and projectile interact with the game and the aliens, however all of the towers inherit from an abstract *Tower* which in turn inherits from *GameObject*. The general approach for the towers are similar, once they are placed they cannot be moved. Note that they cannot be placed on a path, or on top of another tower, or on the base. They all have a certain speed at which they fire, if they are able to fire at this time, they then rotate and fire at an alien in range. When the laser tower fires, it immediately damages the target and then draws the visual graphic. The *LaserProjectile* is simply a green line that lasts a specific amount of time and has no collision detection. We made this decision because we wanted the laser tower to always hit its target and it deals damage immediately because the laser beam moves at the speed of light and is precise. When the plasma tower fires, it creates a *PlasmaProjectile* object which is added to the list of projectiles. The plasma projectile moves toward the location of the target it was aimed at. If it hits an alien it deals damage and is removed, however, it is possible for the plasma projectile to miss it's target and fly off the screen. When the rocket tower fires, it creates a *RocketProjectile* which is added to the list of projectiles. The rocket projectile is passed the *Enemy* object that the tower fired at. Thus the rocket is able to "follow" the alien. As the alien moves, the rocket adjusts its trajectory and the direction that it is facing. The rocket will continue moving until it hits an alien, if the alien hits the base or dies then the rocket continues in a straight line path until it exits the screen. The rocket can hit another alien if it gets in its way. Finally, all the towers can be upgraded. Every upgrade will increase the range, damage, and rate of fire. However, each upgrade will also make the next upgrade more expensive. So it soon becomes much more efficient to buy more towers then just upgrade a few.

Aliens/animation:

The Aliens in our game are designed with a combination of the builder and factory patterns. A factory called *AlienFactory* in our source code is responsible for building the corresponding enemy type based on the enemy type enumerator sent over. Once decided, the *AlienFactory* then builds the correct enemy type using the inline builder class called *EnemyBuilder* inside of *Enemy*. This class builds an *Enemy* game object and allows for the customization of many alien factors. For example, you can customize the bitmap size, start location, resistance values, money, info, movement strategy, etc. Once built, the *EnemyBuilder* returns the customized *Enemy* to *AlienFactory*, which it then returns to the caller class. Generally new enemies are only ever built by *LevelSpawning*. Enemies in our game pull their bitmaps from a bitmap container class hilariously called *BitMapContainer*. This is an area in which we used an object pool. Each enemy has several "frames" that are cycled through each time the enemy moves. Every time the enemy is drawn, it calls a method inside of *BitMapContainer* to retrieve the index for the next frame. This is repeated in a loop to animate the enemies as they move. Because our enemies all fly, we animated the engine flares on the drones and behemoths and added a bit of up and down motion to the soldier who lacks a visible engine. Once an enemy's health pool reaches zero, the enemy type changes to type four, or as I call it, type explosion. While in this type, the enemy will iterate through the "explosion" animation bitmap list. Once finished, the *Enemy* reports back to *GameEngine* (via a Boolean flag) that it is dead and can be removed. Please note, enemies are also removed when they hit the players base. In this case, they are removed after decrementing the player's life count. Lastly, our enemies rely on a movement strategy to move. These strategies are assigned to the enemy based on the selected type in *AlienFactory*. When called to move, an *Enemy* calls the corresponding move method in its movement strategy. For now, the enemies speed is the only thing that differs in each strategy.

Spawning, Maps, & Paths:

As previously stated, when the user selects their desired level on the game menu screen, *Gamemap* is informed by *GameWorld* what background and path to load. *GameMap* selects the background by asking *BitMapContainer* for the associated bitmap for the level that was chosen. The base stays the same for each level and is stored within *GameMap*. The path is chosen by sending over the current level value to *LevelPath*. This class creates and returns a path object based on the selected level. When passed back into *GameMap*, the path object is converted into a list of points (used by the enemies to move) using a private method.

Spawning is handled by the *LevelSpawning* class. When a level is selected, weights are set that define the likelihood of certain enemy types spawning. For example, in the first, and hopefully easiest level, the chance for spawning the hardest enemy type is merely 5%, while in the last and intentionally hardest level, the chance is 20%. Enemy spawning is based on a tick system. Each time an enemy spawns a waiting time is set before the next enemy can spawn. Varying amounts of enemies spawn for each wave, with five waves per level. Once a wave is defeated a waiting time of 250 ticks is set before the next wave spawns. This gives the player some time to hopefully recover if the previous wave did not go as planned.

Enemy resistances are set at the beginning of each wave. *DmgDealt* is a class that we use to collect the damage that is dealt throughout the wave. Once the wave is dead, *LevelSpawning* asks *DmgDealt* for the max damage type (that is the tower that dealt the most damage to the enemies), it then records the resistances for the next wave in our *Resistances* class. The resistance for a damage type doubles for each time it is seen as the max damage type. Damage to the aliens is divided by this resistance before subtracting health.

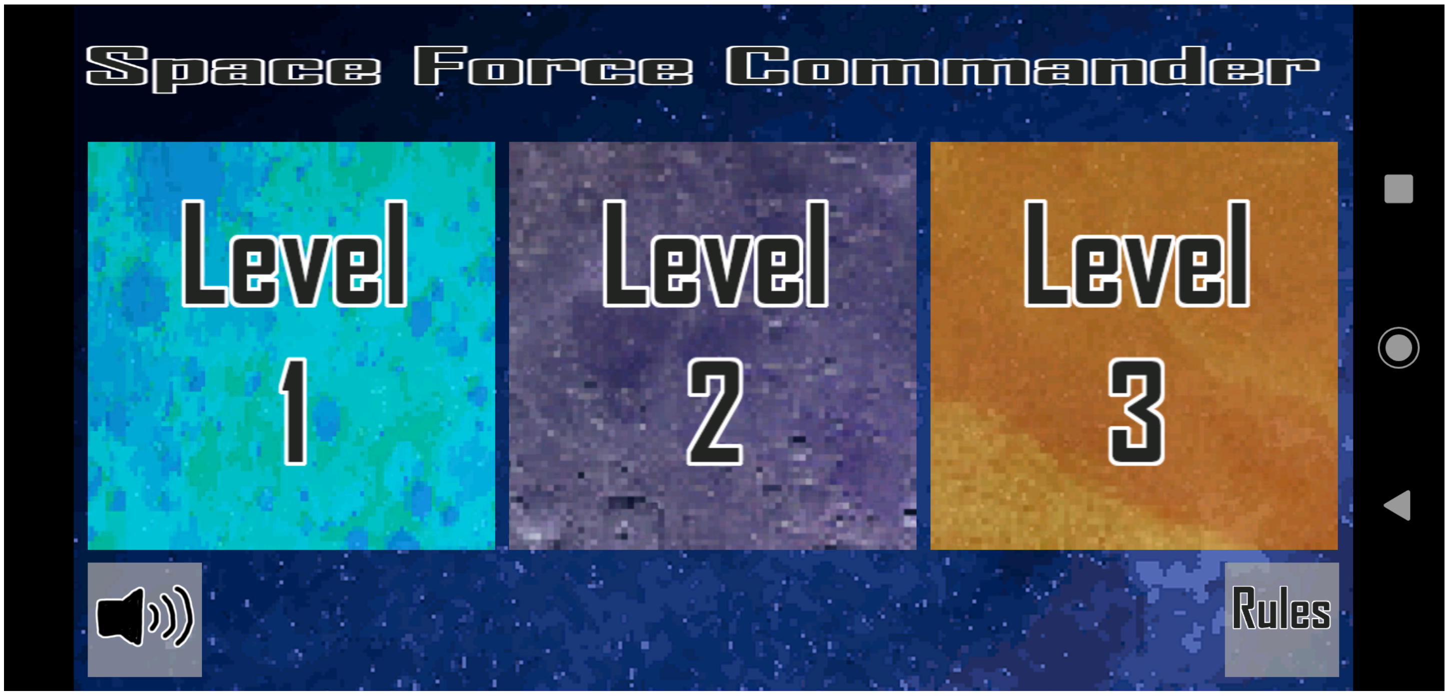
Assets:

The assets for our game were designed by both of us. Quinn is responsible for most of the pixel art for the enemies, towers, and backgrounds while Logan worked on many of the games projectiles, and menu features. The tower sound effects in our game came from online sources licenced under CC0. All of the sound effects were further edited after being acquired. The sound for the plasma tower is “Plasma Rifle Gun Shots - Various” by Synthy95 (see Synthy95). The sound for the laser tower is “laser” by Kafokafo (see Kafokafo). The sound for the rocket tower and for the explosions were made by Quinn in the freeware program BFXR.

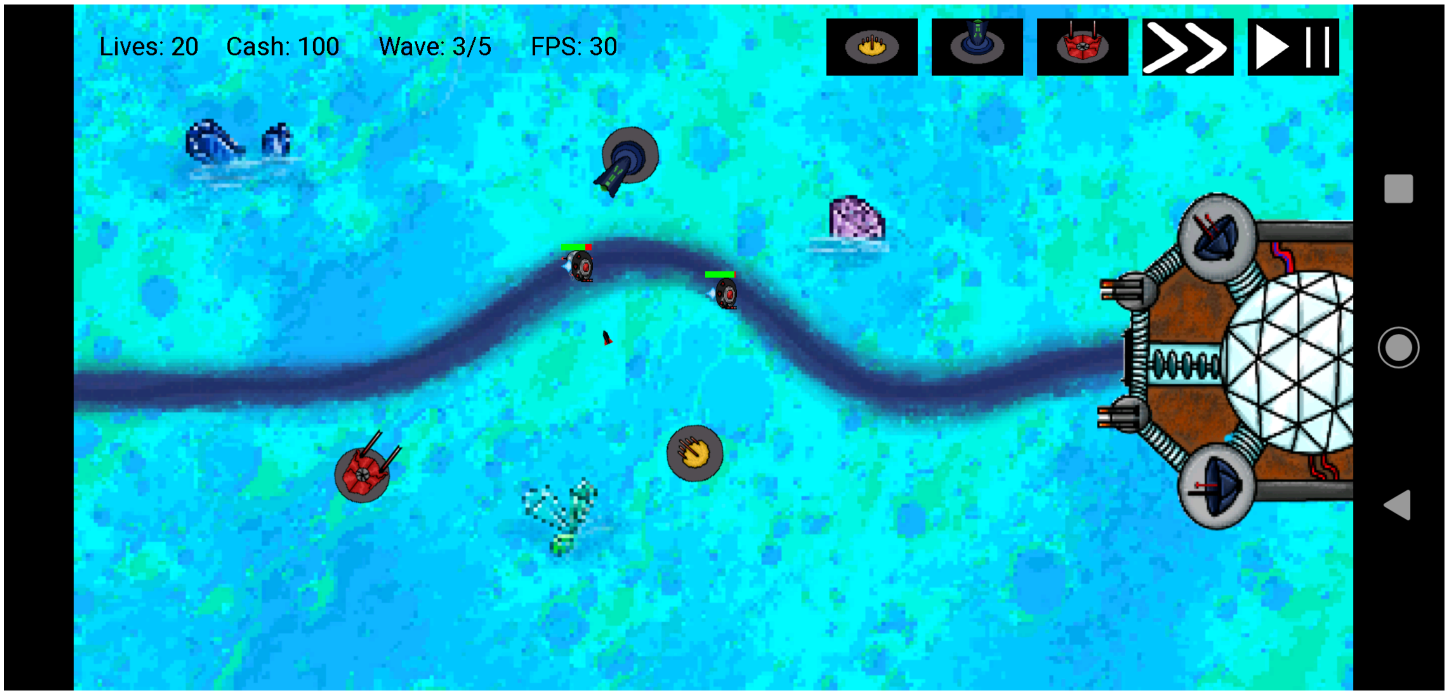
Conclusion:

In conclusion, we are quite happy with our game. While there are some features that we never got around to implementing (more towers, better UI system, and more complex enemies that fight back) we feel like we learned a lot. For both of us, this is the first Android game we have ever developed. It was a great learning experience. If we could go back and change one thing, we would implement more multithreading to improve our games performance. Overall, we hope you enjoy our game, Space Force Commander! (we’re working on the title).

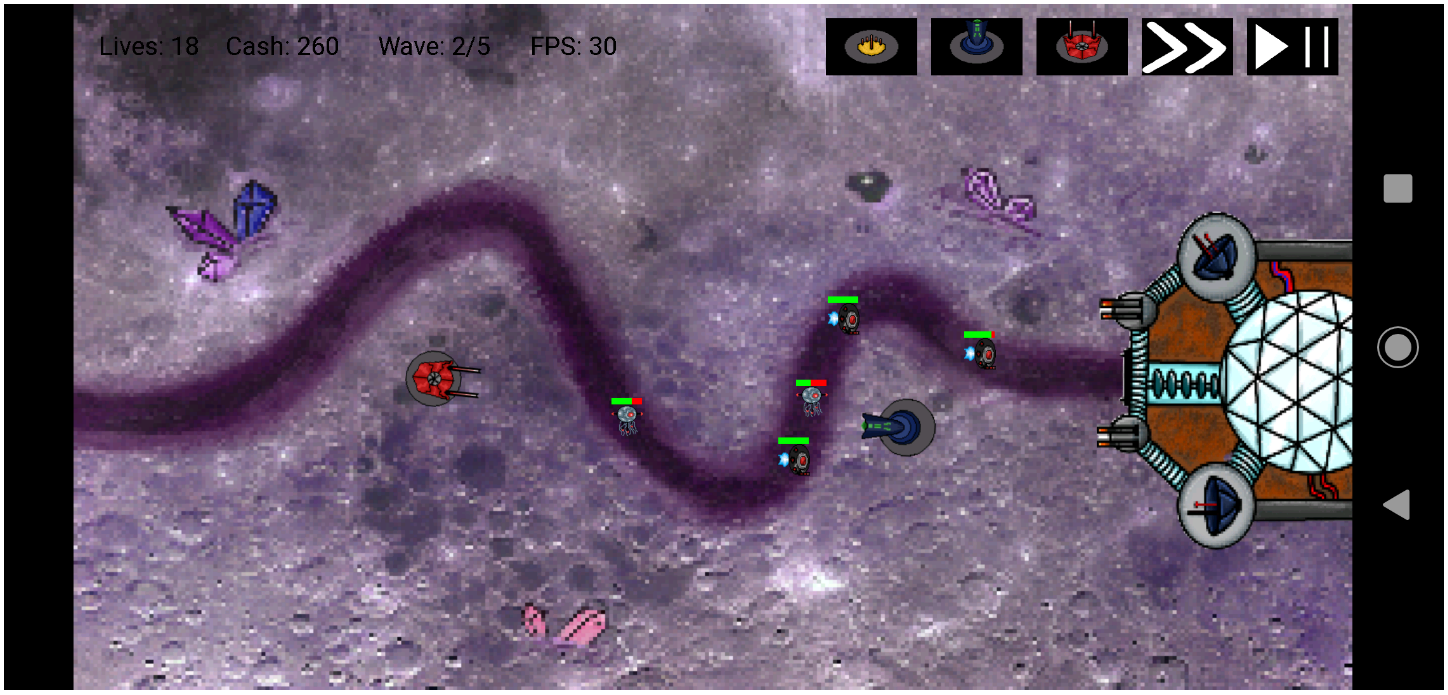
Screenshots:



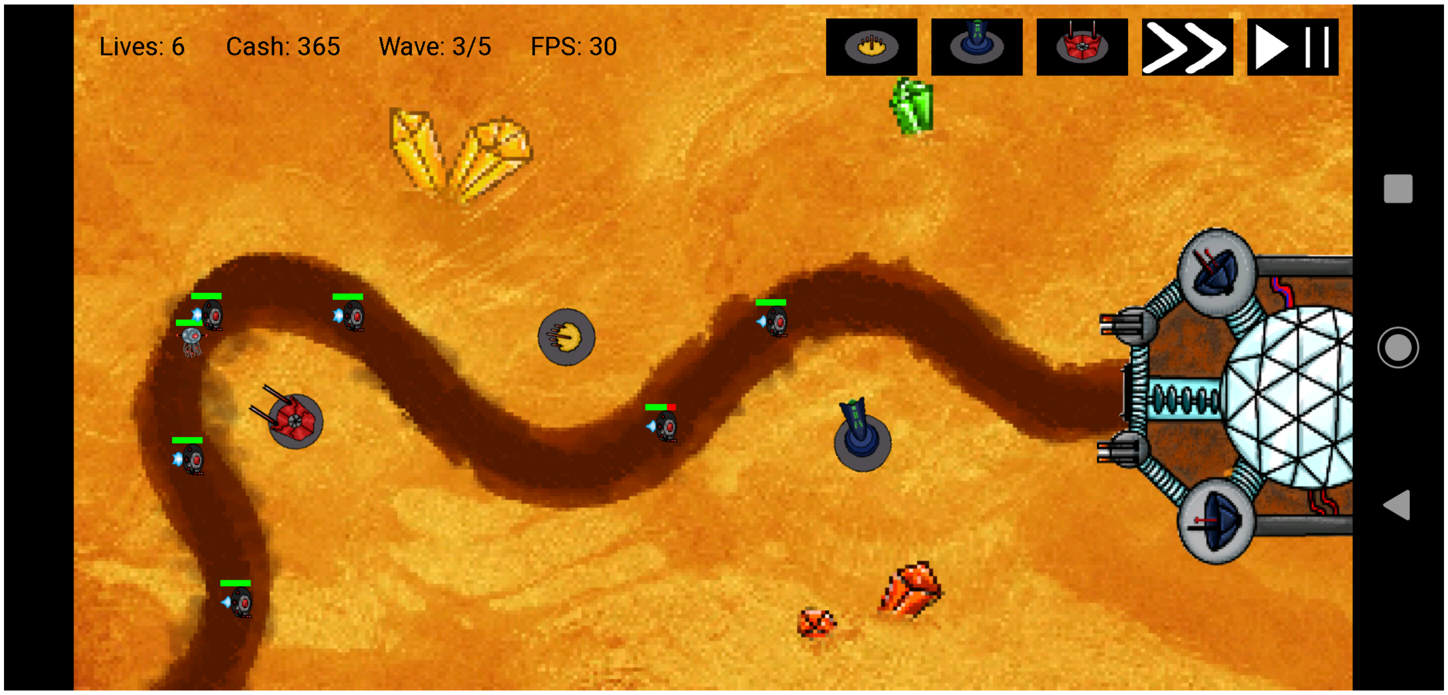
(Main Menu)



(Level 1)



(Level 2)



(Level 3)

Citations

Kafokafo, “laser”, licensed under CC0, uploaded 9-15-2011
https://freesound.org/people/kafokafo/sounds/128229/

Synthy95, “Plasma Rifle Gun Shots - Various”, licensed under CC0, uploaded 12-16-2015
https://freesound.org/people/synthy95/sounds/331240/