
Pew Pew: A Two-Dimensional Space Shooter Game with Roguelike Gameplay Mechanics

GEORGE BATEMAN BA CS
JACOB CANNON BS CS
NICHOLAS DeVOLL BS CS
MADISON SILVA BA CS

FACULTY MENTOR: DR. JORGE REYES SILVEYRA
MAY 27TH 2015

Contents

1	Introduction	1
1.1	Background	1
1.2	Basic Unity Features	2
2	Gameplay	4
3	Requirements	4
3.1	Use Case Scenarios	5
3.2	Sequence Diagram	8
3.3	UML Diagram	9
4	Gameplay Design	10
4.1	Player	10
4.2	Enemies	12
4.3	Weapons	16
4.4	Levels	17
4.5	Objects	17
4.6	User Interface	17
4.7	Art	20
5	Implementation	21
5.1	Current Status	21
5.2	Future Work	21
6	Glossary	24

List of Figures

1	Unity Editor	3
2	Player Death Sequence Diagram	9
3	UML Diagram	10
4	Controls Diagram	11
5	Standard Shooting	13
6	Predictive Shooting	14
7	Purple Enemy	15
8	Green Enemy	15
9	Boss Enemy	16
10	Weapon Stats	16
11	Initial Implementation	18
12	Current Implementation	19
13	Heads-Up Display	19
14	State Diagram Showing In Game Menus	20

1 Introduction

Video games have been a rapidly growing industry in the past 30 years [2]. As such, there are a wide variety of genres and subgenres of games, each taking influence from and building upon successful games of the past. In some cases these games combine past elements in a way that create a new category of game. The goal of this project is to develop a game that will blend space-shooter elements from classic arcade games, such as Asteroids and Galaga, with randomized, challenging elements from more recent games such as The Binding of Isaac. These elements include random pick-ups, permanent player death, and semi-random wave spawning. This combination aims to provide players with new challenges that may not be available in current games. This game was developed using Unity3D and runs in the Unity engine, which features robust tools that allowed us to manipulate nearly any element of the game quickly and with immediate feedback. In the following subsections, we will introduce the basic features of Unity, our inspirations, and a few of the researched technical game engine systems.

1.1 Background

For the duration of this project, we made a video game using Unity. The majority of the research time was used to learn the functionality of Unity. Since Unity is a complete game engine and editor, the game creation process did not require the development of an engine. Other alternative engines, such as the graphics engine Ogre3D, would require the implementation of a game engine in addition to creating the game [11]. Unreal Engine 4 was another engine that we considered as it included many features that would be useful to this project. However, it can be difficult for beginners to learn due to the lack of tutorials. Ultimately, we choose Unity, since it has a wide range of tutorials to help us get started more quickly and will allow the majority of the project resources to go towards gameplay development.

Unity historically supports three languages, Boo, JavaScript and *C#*. Over 80% of Unity projects use *C#* as their scripting language, and most tutorials and sample assets are based around *C#*. [1] The *C#* language was created by Microsoft in 1997, influenced heavily by Java and C++ it is intended to be friendly to the software development process. Its features such as uninitialized variable compatibility and automatic garbage collection contributed to our decision to use it for this project [8]. Even though *C#* is a compiled language, Unity is still able to use *C#* code snippets as scripts because Unity is built in IDE (Monodevelop), compiles the code and packages it with game resources before runtime. This gives the games a robust set of tools, but allows the game to load and run quickly. [10]

Our game strives to follow the principles set out by previous roguelike games. The term roguelike is a broad term used to describe video games that use random elements to create a replayable game that restarts the player on each subsequent playthrough. The original

game that spawned this categorization was the game Rogue. Rogue was programmed at UC Santa Cruz in the 1980s, using ASCII characters to take players on an adventure through a subterranean dungeon for the Amulet of Yendor. Once the game was released to others outside the developer, it spread rapidly throughout the college and the surrounding communities. Rogue inspired many other hits in the video game industry, including Diablo II, Faster Than Light, and The Binding of Isaac. [6]

While developing unique and challenging non-player entities it is crucial to have artificial intelligence (AI). Without AI entities quickly become predictable and non-threatening, even simple enemy AI can make improvements to a game. One of the first steps in creating a basic AI is choosing a method of path finding, so that the enemy can intentionally find the player. An example of a basic but effective path finding algorithm is A*. Amit Patel defines A* as a combination of Dijkstras Algorithm and the Best-First-Search Algorithm. A* searches all vertices by favoring the ones in the direction of the goal, this helps it be more efficient like Best First Search but avoid obstacles more like Dijkstras. [14] A* is primarily used in games with large stationary objects like rivers and walls [14]. Since this game will not have any stationary objects, it is not as necessary to use a path finding algorithms. If the enemy AI calculates a straight path to the player, the path will almost always be clear of obstacles. The second major part of enemy AI is shooting, this adds another level of difficulty to the enemy. These two components give the enemy enough instructions to be able to find and kill the player.

Finally, procedurally generated content can be used in order to build satisfying and challenging areas for the player. Procedural content generation is the process of designing an algorithm that can generate content at runtime. This fits in well with this project. With a relatively small number of inputs, the game can generate numerous different levels where the enemies encountered are found in different combinations. It also embodies the core idea of a roguelike game, changing the game every playthrough to challenge the player differently. There are a multitude of content generation algorithms, one of them is called Perlin noise, which actually refers to two algorithms created by Ken Perlin. These algorithms use a sum of different functions with varying amplitudes and frequencies in order to generate what is called a noise function. It is this function that is used to define different points of intensity. As it pertains to this project, these functions could be used to generate nebulae or star maps, giving a different visual experience to our player every time. [7]

1.2 Basic Unity Features

The Unity development platform contains a number of useful tools that a developer can use during the game creation process. The main view is shown in Figure 1.

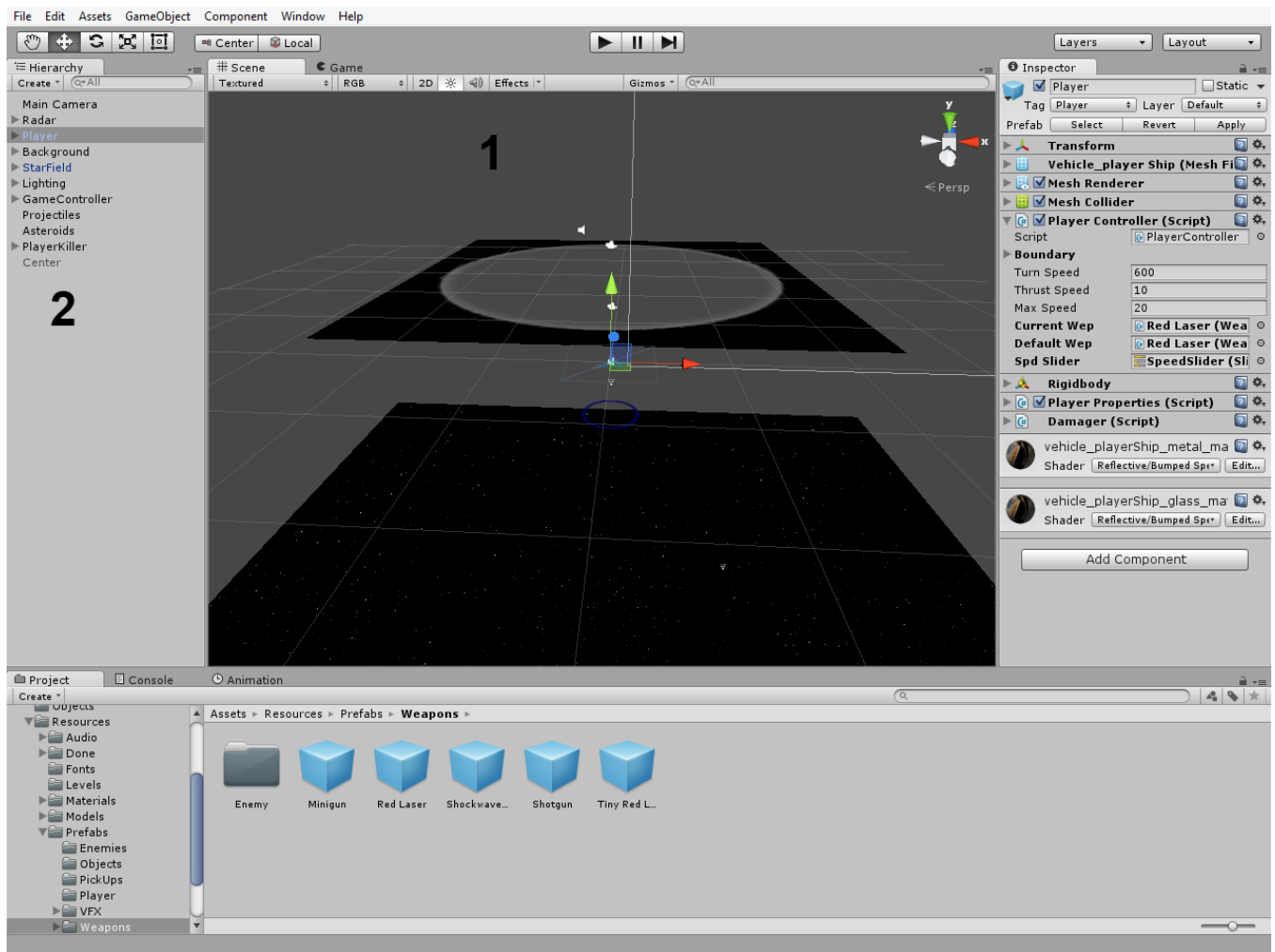


Figure 1: Unity Editor

Some of the components found in this view include:

Scene: The scene view, labelled with the number 1 in Figure 1, shows the current scene and where all of the Unity objects are located inside that scene. Scenes are independent from each other and moving from one scene to another loads an entirely new level with new scripts, objects, etc.

Hierarchy: The object hierarchy, labelled with the number 2 in Figure 1, lists all of the game objects currently in the scene. It also shows how the game objects are laid out inside the current scene. Game objects use a parent/child relationship as defined in the object hierarchy.

Camera: A camera in Unity renders the objects positioned in front of itself to the screen. Multiple cameras can be used to display different views of the scene. Cameras can

be adjusted in the scene view of the editor.

Layer: A layer in Unity defines what in the scene view is rendered to which camera. Each camera contains what is called a culling mask that determines which layers it will render.

Prefab: A prefab is a saved instance of a game object which can then be loaded as either a child of a game object or its own instance. Any game object can be saved as a prefab.

2 Gameplay

Core Concept: Pew Pew can be described as a wave-based, randomly generated, twin-stick shooter. Pew Pew features a small space ship on a two dimensional plane that the player controls with the left analog stick and can shoot with the right analog stick, hence the name "twin stick shooter". The game takes place in a circular arena, in which waves of enemies spawn in random sizes and compositions. The player must defeat all the enemies in a given wave by shooting them with various weapons, received from pickups, to progress onto the next wave. At the start of a wave more enemies will spawn than the previous wave. After defeating a set amount of waves, the boss will spawn in the arena. If the player manages to defeat the boss without dying, they achieve victory and will be rewarded with the victory screen. The player can be defeated by taking too much damage from enemy projectiles, hitting too many asteroids, or leaving the area by passing the line known as the boundary or the edge of the field of play.

First Minute: The player enters the game in their ship, and can immediately start exploring the area in which they spawn. The player is given a few moments to get used to the controls before the first group of enemies spawn. This is the first wave; it will contain very few enemies that will be created some where in the game, out of the view of the player. The player must find and defeat all of these enemies before the next wave with new enemies starts.

Victory conditions: The player will achieve victory by defeating all of the enemies in each wave that is generated without dying. The game culminates with a boss fight, and defeating this boss will trigger victory menu and end the game.

3 Requirements

The following use cases and diagrams describe how the game has been implemented in the Unity game engine. While Unity is set up to handle the View portion and some of the Controller portion of the MVC architecture, we had to supply part of the Controller and the Model portion in order to deliver our game.

3.1 Use Case Scenarios

Scenario Name: **PlayerDeath**

Participating Actor Instances:

Player: PlayerEntity
Damager: DamageEntity
Heath: PlayerHP
Shield: PlayerShield

Flow of Events:

1. **Player** collides with **DamageEntity**.
 2. **Shield** decreases based on **DamageEntity**.
 3. **PlayerHP** decreases based on **DamageEntity** and **Shield**.
 - a. **PlayerHP** is greater than zero, continue as normal.
 - b. **PlayerHP** is less than or equal to zero, **Player** dies.
 - i. **Player** is deleted.
 - ii. Explosion animation spawned in place of **Player**.
 - iii. An end game menu prompts **Player** to either play again or return to main menu.
-

Scenario Name: **LeavingFieldOfPlay**

Participating Actor Instances:

Player: PlayerEntity
Projectile: PlayerKiller

Flow of Events:

1. **Player** leaves the field of play.
 2. Warning alarm sounds, warning flashes on screen.
 - a. **Player** chooses to turn around and return to field of play, everything returns to normal.
 - b. **Player** advances farther out of the field of play.
 - i. **PlayerKiller** kills **Player**.
-

Scenario Name: **SpawnEnemyWave**

Participating Actor Instances:

Enemy: EnemyEntity

CurrentWave: float

GC: GameController

Flow of Events:

1. **GC** waits a set amount of time before attempting to spawn a wave of enemies.
 2. The allotted amount of time has passed.
 - a. **CurrentWave** is not equal to the number of total waves.
 - i. Enemies are spawned in according to how many waves the player has defeated.
 - b. **CurrentWave** is equal to the number of total waves.
 - i. The boss is spawned in and the player must fight it to win.
-

Scenario Name: **MoveAndShoot**

Participating Actor Instances:

Player: PlayerEntity

Bullet: Projectile

Flow of Events:

1. **Player** presses a movement button, which defaults to the WASD keys, while simultaneously pressing a fire button, which default to the $\uparrow\downarrow\leftarrow\rightarrow$ arrow keys.
 2. **Player** moves in the direction of the movement button pressed, while **Bullet** is fired in the direction of the fire button pressed.
 3. **Player** continues moving in the way of the first directional input until an additional input is received, at which time **Player** will begin to thrust forward, while **Player's** direction will quickly change to face towards the second input.
 4. **Bullet** travels in the direction it was initially shot without the possibility of changing direction.
-

Scenario Name: **SpawnAsteroids**

Participating Actor Instances:

Asteroid: AsteroidEntity

GC: GameController

Flow of Events:

1. **GC** performs a check to see where an asteroid will spawn.
 - a. **GC** puts the asteroid in the view of the player.
 - i. The **Asteroid** is not spawned, and **GC** attempts to spawn another.
 - b. **GC** puts the asteroid out of view of the player.
 - i. An **Asteroid** object is spawned with a random movement vector and rotation speed.
 - ii. The **Asteroid** careens through space until it hits something or goes outside of the level's absolute boundary.
 1. The **Asteroid** is then destroyed.
-

Scenario Name: **BulletFizzle**

Participating Actor Instances:

Player: PlayerEntity

Bullet: Projectile

Flow of Events:

1. **Player** shoots a **Bullet**.
 2. **Bullet** does not hit anything within the view of the **Player**.
 3. **Bullet** travels out of the view of the **Player**.
 - a. The **Bullet** collides with an entity
 - i. The **Bullet** causes damage to that entity, and terminates itself.
 - b. The **Bullet** does not collide with anything.
 - i. The **Bullet** is terminated after leaving the projectile boundary which surrounds the **Player**.
-

Scenario Name: **SpawnPickUp**

Participating Actor Instances:

Player: **PlayerEntity**
Enemy: **EnemyEntity**
Pickup: **PickUp**

Flow of Events:

1. **Player** destroys an **Enemy** with its weapon.
 2. Before it dies, the **Enemy** generates a random number to see if it drops anything.
 3. The **Enemy** drop check succeeds and **Enemy** drops a **Pickup** based on the number that it rolled.
 4. **Player** touches the **Pickup**, equipping it and gaining its contents.
-

Scenario Name: **PauseGame**

Participating Actor Instances:

Player: **PlayerEntity**

Flow of Events:

1. **Player** presses the pause button.
 - a. A screen displays over the main screen, informing the **Player** the game is paused.
 - b. The game's timescale is set to 0, freezing any objects currently in the scene.
2. **Player** presses the pause button once more.
 - a. The pause screen is disabled and removed.
 - b. The game's timescale is set back to 1, resuming the action.

3.2 Sequence Diagram

The sequence diagram represented in Figure 2 corresponds to the use case *PlayerDeath*, and outlines the process behind the player being hit with a projectile to a death state, followed by the appropriate response by the *GameController*. Health will only be subtracted if the player has no remaining shield. If the player no longer has any health or shields after taking damage, then the *Player* object will report its death to the *GameController*. Sequence diagrams for other use cases are functionally similar, albeit with a few differing actors and method calls.

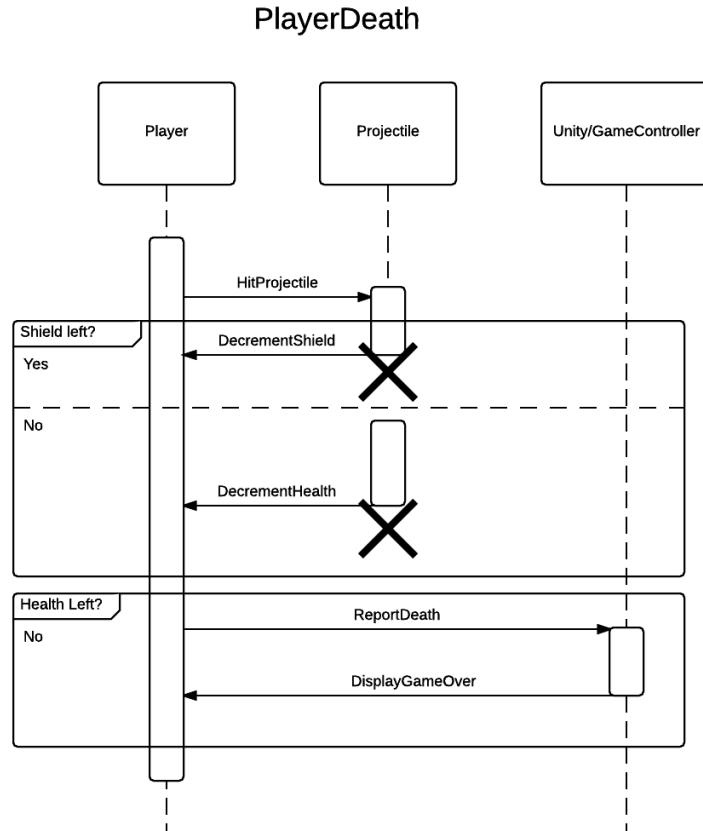


Figure 2: Player Death Sequence Diagram

3.3 UML Diagram

Figure 3 shows the core class design for the game and the relations that classes will have with each other. Classes are a representation of Unity GameObjects or Prefabs that are used in the game. Classes will also be interacting with existing classes of the Unity engine in manners which are not displayed in Figure 3. *GameManager* will be a singleton class that will be responsible for holding the *Player Entity*, this way the game can easily transition the *Player* between scenes without risk of creating a duplicate *Player*. *GameController* will handle everything that happens inside the level itself, and will be able to do this with the `onAwake()` function, provided by Unity, that activates when the script has been loaded into the scene. The *Entity* object will provide a basis of all entities in-game that will be able to move and react to the game state, including the *Player*. The *Ship* class will be a prefab

handled by Unity that will allow us to instantiate an *Entity* object with certain variables from the start. Our *Item* and *Weapon* interfaces will handle items and weapons picked up in the game, respectively. It is important that there is a distinction between the two, as all *Weapons* are *Items* but not all *Items* are *Weapons*.

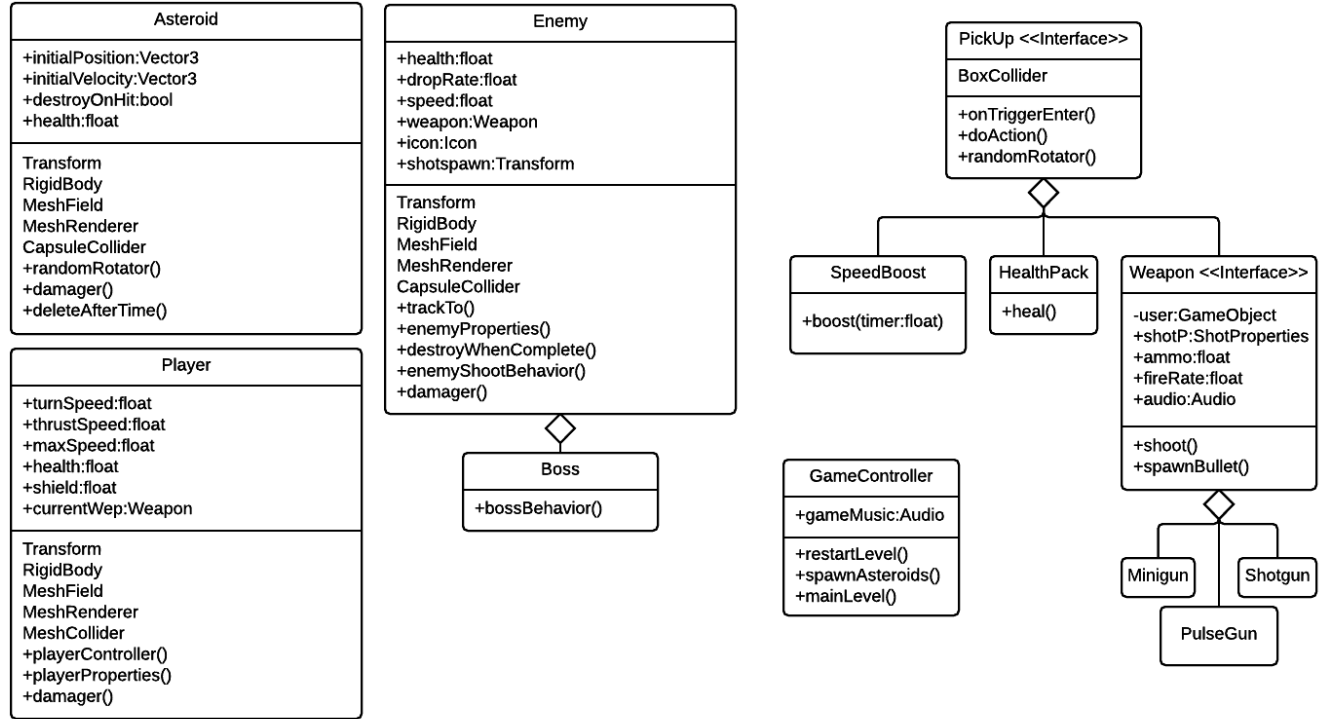


Figure 3: UML Diagram

4 Gameplay Design

Games are dynamic pieces of software that focus on delivering exciting experiences to players, gameplay design is as important as backend design. What follows is the basic design for the aspects of the gameplay that the player will see for our current iteration of the game.

4.1 Player

Player controls: The player controls a small spaceship, and navigates through a space environment. As seen in Figure 4, the player will move with the WASD keys; pressing a key will cause the ship to rotate to and accelerate in that direction. Shooting will be controlled by the arrow keys on the keyboard. Pressing an arrow key will cause shots to be fired in that direction. Since movement and shooting are separate, the player may move in one direction and fire in another. The player can also pause the game with the escape button.

The controls are intended to transfer over to a controller relatively easily, so the player may control moving and shooting with the left and right analog stick respectively.

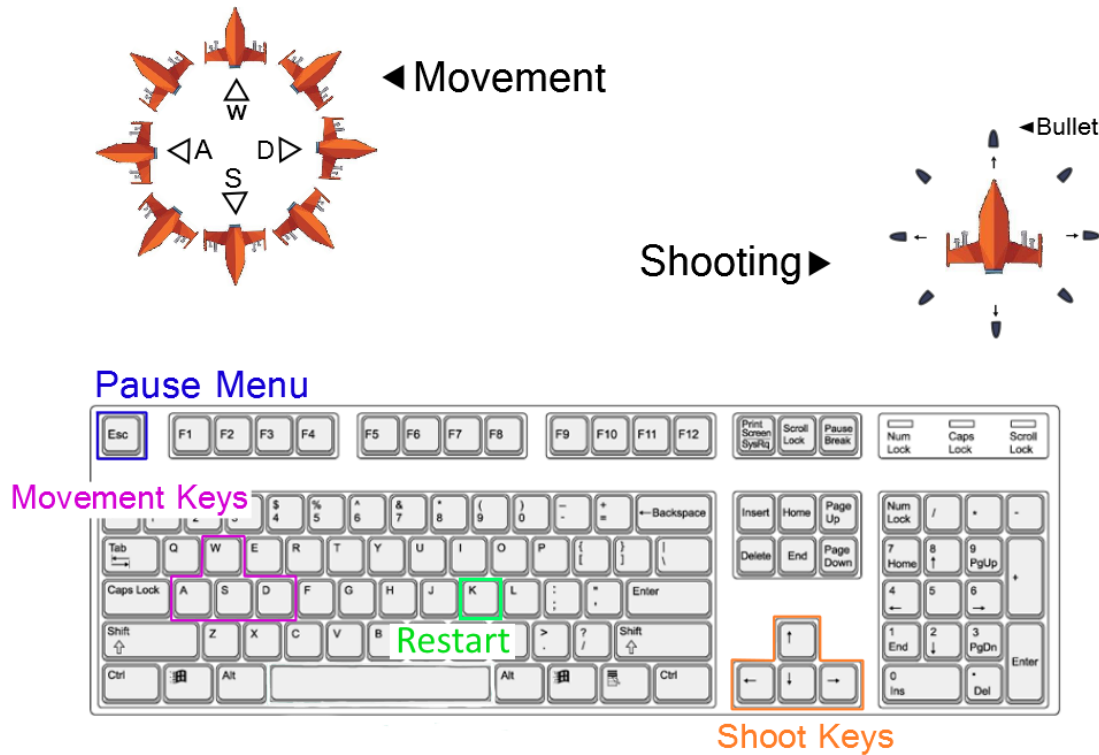


Figure 4: Controls Diagram

Player Movement: As stated above, the player presses either the WASD keys or the left analog stick to begin rotating and accelerating in that direction. Intended direction is generated as a vector, which has X and Y values, allows any degree of rotation to be represented. Since both the directional keys and analog sticks return vectors, both control methods can be used interchangeably. Rotation has a maximum speed, so sudden changes in direction cause the ship to veer in a small circle as it rotates to face the intended direction. Acceleration speed will not be instantaneous and the player will not stop automatically, in order to simulate a sense of weight and the lack of friction in space. The ship will perform these actions quickly enough that controlling the ship feels responsive to the player.

Player Shooting: Player shooting functions similarly to player movement in terms of input. By pressing the arrow keys or on the right analog stick, the player will shoot in the respective direction. The player's ship has a rotating gun attached to it, which allow the

player to aim and shoot in any direction regardless of current movement. However, there is no rotation time for the gun on the player's ship, so the gun will shoot instantaneously in the direction that is being pressed. By default the player's ship shoots a rapid fire laser, which will allow the user to get comfortable with shooting without having to worry about missing or wasting ammo. Various weapon pickups can be used to modify how the player's weapon shoots as detailed in Section 5.3. The control method will remain the same for all weapons so the player does not have to worry about learning different controls for each weapon.

Player ship: The player's ship will have its own health and shield values. The shield will be a rechargeable defense that will be able to absorb a number of shots before the player's ship takes permanent health damage. Recharging of the shield will occur over time after a few seconds of avoiding damage. The ship's health is the amount of damage the ship can sustain before it explodes and the player loses. However, it will not regenerate automatically like the shield, but items will be able to restore it.

4.2 Enemies

The main objective of the enemy ships is to find and kill the player ship. They will have two main types of AI that will decrease their predictability. Other than just posing a threat to the player, an important function of the enemies AI is encouraging the player to always be moving. This is accomplished by having the enemies spaced out around the map at all times. Enemy shooting patterns are different based on the enemy type to encourage variety, so methods of shooting have been broken down into two types, Standard Shooting and Predictive Shooting, detailed below.

Standard Shooting: Standard shooting is when the enemys shooting vector is set to be the same as the enemys movement vector. This makes it so that the enemies shoot straight in front of them. The two main enemies, both of which are always rotating towards the player, use this method, meaning they will usually be shooting in the direction of the player. The shots tend to be inaccurate because the player is always moving, however if the enemies had higher accuracy the game could become to difficult. Standard shooting is depicted in Figure 5.

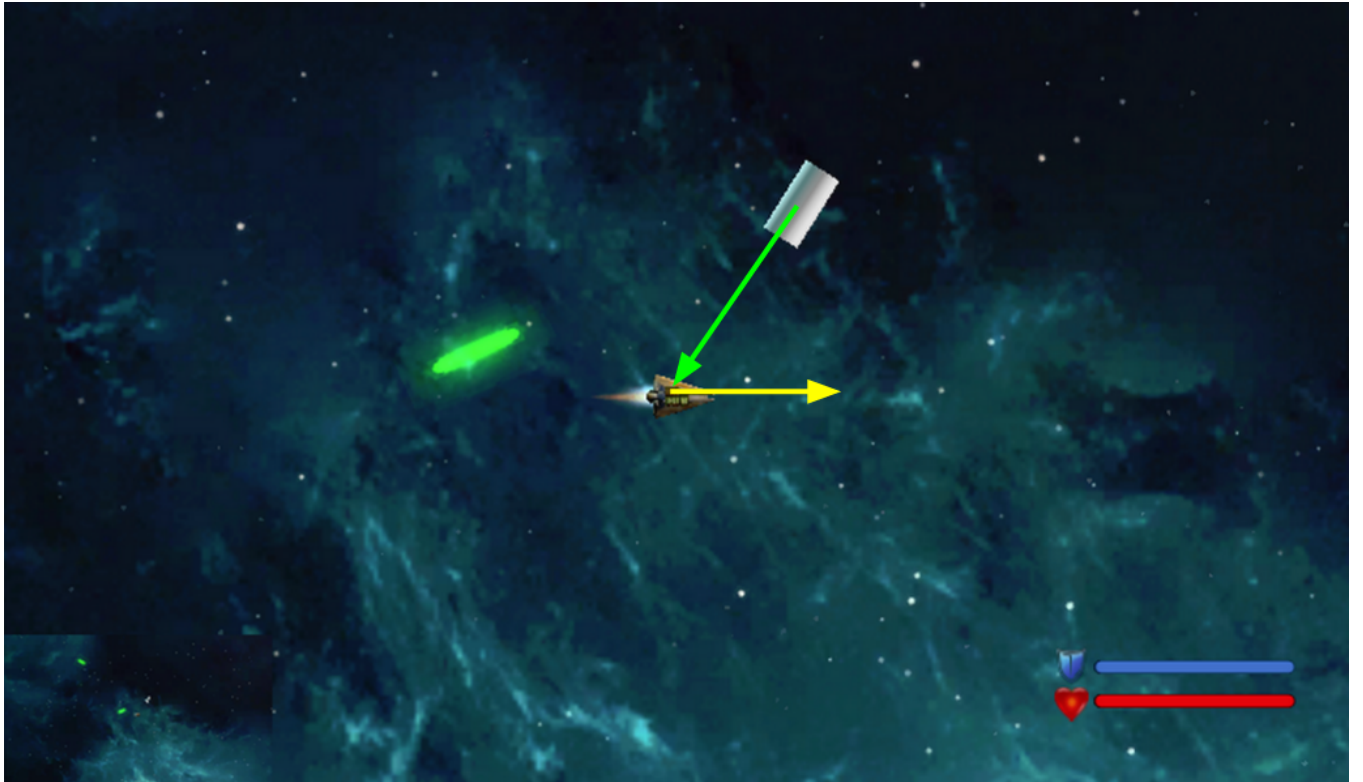


Figure 5: Standard Shooting

Predictive Shooting: Predictive shooting is a method of shooting used by the Boss and PlayerKiller. This method of aiming and shooting allows the two enemies to hit the player if the player continues to move in a straight line by aiming at where the player will be. Predictive shooting uses the current movement vector of the player, the projectile speed and its starting location to calculate the vector to shoot ahead of the player. The predictive shooting projectiles are much more difficult to dodge than Standard Shooting and requires the player to plan their movement ahead of time. Predictive shooting is depicted in Figure 6.

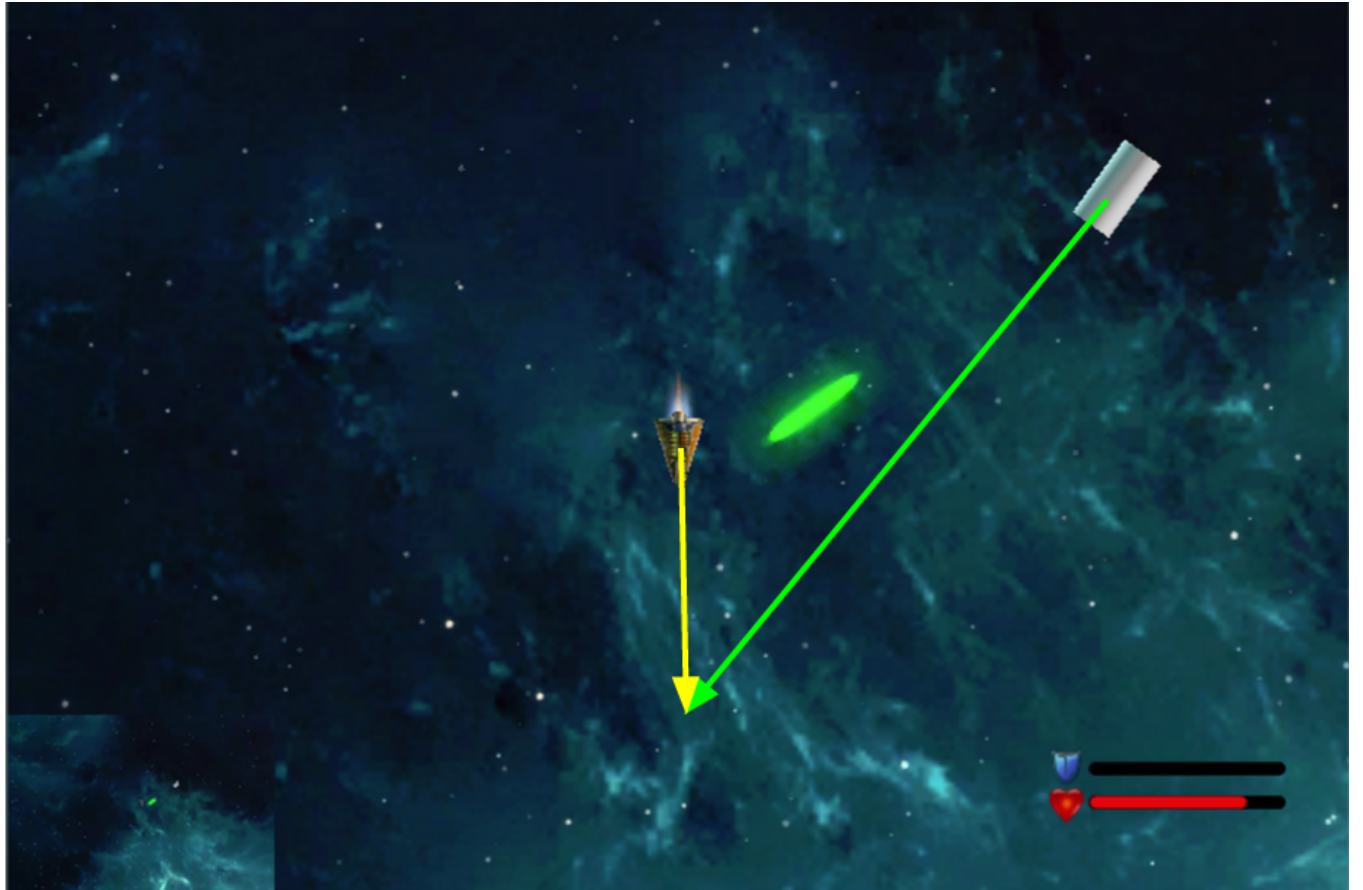


Figure 6: Predictive Shooting

Enemy One (Purple Enemy): The first type of enemy tracks directly toward the current location of the player, by constantly rotating toward the player and then applying forward thrust in the direction it is currently facing. It aims to maintain a small distance from the player to avoid colliding with them. This enemy type has a high probability of shooting at a fixed time interval, using standard shooting.



Figure 7: Purple Enemy

Enemy Two (Green Enemy): The second type of enemy will follow the player more closely than the first type while also trying to get in front of the player as an obstacle to the players movement. To attempt to get in front of the player, the enemy uses the predictive shooting method to calculate a vector to intercept the player and then uses it as its movement vector. This enemy is still able to use standard shooting because it is always rotating toward the player even if it is moving in front of the player. Its projectiles do medium damage, firing less frequently than Enemy One.



Figure 8: Green Enemy

Patrolling: Once an enemy is far enough away from the player, they will begin to patrol instead of track the player, “far enough” is a constant variable called max distance. Max distance is set so that the enemy will be just out side the players field of view when they stop tracking the player. In order to patrol, the enemies will store their initial location as “home”. When the enemy is farther than max distance to the player they will switch from tracking the player to tracking “home”. This will cause the enemy to return to their initial location in the level and circle around it until the player comes back within the max distance. This is the main method for keeping the enemies spaced out around the map, creating a consistent level of difficulty though out the level.

Boss: The boss will give the player a final challenge before ultimately reaching the end of the game and winning. As seen in Figure 9 the boss is significantly larger than the player, this makes avoiding the boss much more challenging than avoiding other enemies. The boss

will shoot large amounts of projectiles to make dodging difficult for the player. Additionally, the boss will have a predictive shooting projectile that will shoot periodically to increase the difficulty further. Finally, bosses will have a larger amount of health than the average enemy. Defeating a boss will end the game and display a win screen.



Figure 9: Boss Enemy

PlayerKiller: PlayerKiller is a stationary, invincible and invisible enemy that is used to enforce the boundary of the level. Upon exiting the arena, PlayerKiller will activate and will begin to shoot at the player until the player is destroyed or re-enters the level. PlayerKiller's projectiles travel very quickly and will destroy the player in one hit, so that the player cannot spend much time safely outside the level.

4.3 Weapons

Weapons: Weapons are implemented through the Unity prefab system. By default, the player is given a low-damage, high rate-of-fire red laser weapon that they can change by finding a weapon pickup which drops off enemies. Once the player makes contact with the pickup, their current weapon is destroyed and the new weapon is loaded for the player. All non-default weapons have a limited amount of ammunition, and once their ammo is depleted, the weapon is destroyed and replaced with the default weapon. A table showing off the various variables of the weapons can be found below in Figure 10.




Weapon	Damage	Icon	Bullets / Shot	Shots / Second	Deviation (°)	Penetrate?	Ammo
Default	10	N/A	1	10	0	No	N/A
Shotgun	10		5	10	40 (fixed)	No	50
Pulse Gun	250		70	0.5	100 (fixed)	Yes	10
Minigun	5		1	50	30 (random)	No	400

Figure 10: Weapon Stats

4.4 Levels

Level design: The level consists of a circular arena or playing field. There is a border around the playing field, represented by a thick blue circle. If the player crosses this border they will be shot, this can be seen in the **LeavingFieldOfPlay** use case in Section 3.1. Enemies spawn in waves inside the playing field but outside the player's view. Asteroids spawn in and around the arena. As the player destroys waves of enemies, new waves consisting of more enemies will spawn. After completing a set number of waves, the boss will spawn in the arena.

Star Field and Background: The background will be a two dimensional tile that can be replicated across the screen to create a smooth, endless field of stars. The star field layered on top of the background image is particle effect consisting of small drifting particles that will move across the screen. The multilayering will add a three-dimensionality to the background and keep the background from being static.

4.5 Objects

Pick-ups: There will be pick-ups available to the player throughout the game, dropping from enemies. Most pick-ups will change some attribute of the player ship, such as giving the player an alternative weapon, a temporary speed increase, or restoring the player's health. Additionally, there is another pickup that will spawn a follower ship that will follow the player ship around that map and shoot at nearby enemies. Some pick ups will be displayed in part of the user interface, such as the speed boost meter.

Items: Items are intended to either provide a permanent boost to the player attributes (such as speed or maximum health), or add an effect that the player can activate by pressing a button, such as a time bomb that would slow down enemies and asteroids when activated, making them easier to dodge. There are currently no items implemented in the game, but could be included in the future.

Asteroids: Asteroids will be obstacles that will fly across the screen, dealing damage to anything they collide with. This forces players to be more aware of their surroundings or risk dying and gives them another danger they must avoid. Asteroids have no set allegiance and can collide with each other and enemy ships. Asteroids will spawn in random locations outside the player view with a random initial velocity and direction. Asteroids will travel until they either collide with a ship, get shot by the player, or travel outside of the level boundaries.

4.6 User Interface

Radar: The radar shows the location of all enemies, pick-ups, and asteroids in relation to the location of the player. It will be displayed in the bottom left corner. In addition, it

also represents the direction of all enemies not shown in the immediate area with a symbol on the edge of the radar, assisting the player with locating all enemies in a wave. The radar was implemented in two separate iterations. In the initial implementation, a camera is used to provide an overhead view of the in game entities. However, it was difficult to tell what was shown using this method, therefore it did not show the player very useful information. Image 11 depicts what this implementation looked like.

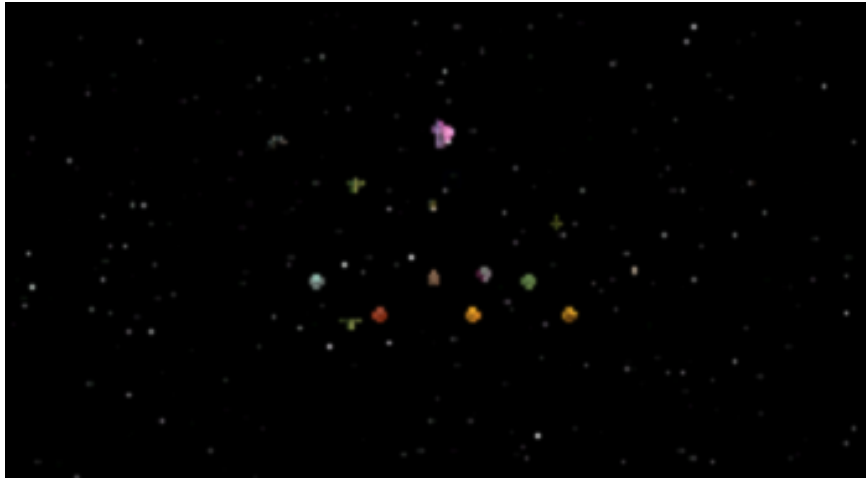


Figure 11: Initial Implementation

The current implementation of radar uses the same basic concept as the first implementation. However, the major difference is that the latter implementation makes use of Unity's layer function. Layers in Unity determine what in game entities are rendered to which camera. By defining new layers, a camera was created that only renders icons that were attached to each enemy, pick-up, and asteroid prefab. In our implementation, two Hashtables keep track of all enemy positions in game as well as a list of icons that alternate between being rendered to the camera and being put in the invisible layer. The invisible layer is not rendered to any camera in the game, and effectively hides any models assigned to it. This is what accomplishes the goal of having enemy positions cling to the edge of the radar image. Image 12 depicts the second implementation.

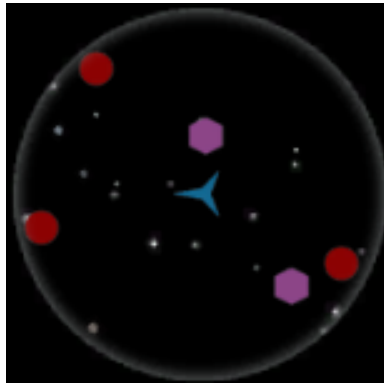


Figure 12: Current Implementation

HUD: The player has access to a Heads-Up Display in the bottom right corner, shown below in Figure 13, which displays vital information: The player's health, shield, currently equipped weapon, ammo remaining, the current wave, and the number of enemies left in the wave. The HUD will also display if the player is currently under the effects of a speed boost and how much time is remaining on their speed boost.

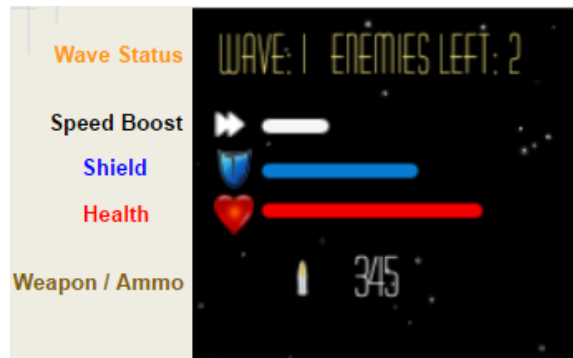


Figure 13: Heads-Up Display

Menus: Menus in the game consist of a selection of several buttons that the user can select with the cursor to navigate through different game states. The primary menu is the main menu, which allows the user to start the game, navigate to the controls menu, or exit the application. The controls menu gives a brief game description and shows the controls, with an option to return to the main menu. Additionally, there is a pause screen that appears when the player pauses in game that shows a few different controls that can be used to resume or restart the game. The state diagram in Figure 14 depicts these menus as they appear in the game as well as the connections between them. Each menu/state corresponds to either a distinct Unity scene or a GUI child menu.

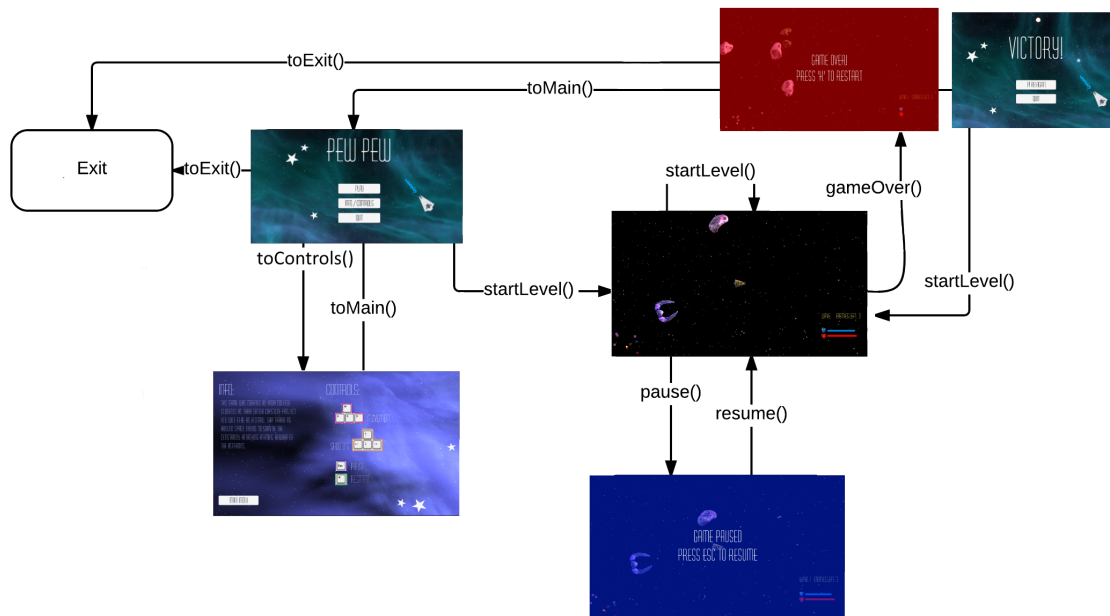


Figure 14: State Diagram Showing In Game Menus

4.7 Art

Story: The Player is an explorer trying to reach the other side of the Universe to meet aliens that sent a treasure map to earth. The Player keeps in close contact with General Weathersbee who guides them to the last known location of the aliens.

Art/models: The assets and models used in this project will mostly be free from the Unity store. Our group created some of the simple icons, used in the radar and the HUD. Some of the assets are combined and blended together to create a unique model specifically for this game. For example, we implemented a gun from one asset pack on top of a ship from another pack to create a new ship type. There is a black with stars background for the main level, with a moving star field overlay. The other scenes have varying space themed backgrounds, with the same star field. The font Huxley is used uniformly through out the game to tie the scenes together.

Music/ sound: Sound effects and music will be retrieved from the Unity store and other sites that allow people to download and use their music for non-commercial purposes. The music is upbeat and fast paced to set the tone of the game. The sound effects will mostly be explosions and gun sounds. For example the laser gun will have a "pew pew pew" sound.

5 Implementation

This section describes the current status, as well as future work that could be incorporated into the project.

5.1 Current Status

In the current state of the project many of the functional goals have been accomplished. By the academic festival there were five different pick-ups, a loss condition, a boss with a corresponding win condition, and a single level complete will correct wave spawning. We reached all of the major goals and created a playable game. Additional features added during development include a predictive shooting method, a follower, and independently rotating guns attached to the player and boss ships. Due to the lack of time, the game's level structure was modified from a multi-level adventure to wave-based gameplay in an arena.

5.2 Future Work

Since our project is in a complete state, anything added would be to expand or enhance the game. That being said, if we had more time to work on this game there are a few things that we could work on:

- **Update Assets:** Finding new assets or learning to create our own assets would be helpful for creating a more consistent art style. As well as creating a more unique game that does not have the same assets as many other low budget space games.
- **Update AI:** Adding more sophisticated and challenging AI would be a way to add difficulty to the game. It would also smooth out the movement of the enemies making them less jumpy and less predictable.
- **Adding Menus:** Menus that add a ship selection, achievements or options like difficulty, would increase peoples desire to replay of the game. For example if the way to unlock more ships was by completing achievements, people would have to play the game multiple times if they wanted to play with a new ship.
- **Adding Levels:** More levels would vary the game play more and make it possible to incorporate our story in to the game.
- **Radar:** The next step in implementing radar would be to render it to the Canvas with all the other elements of the HUD. This can be accomplished by either using scripts to render it to the Canvas, or by defining a target texture where the camera would render.

References

- [1] Aleksandr. Documentation Unity Scripting Languages and You. September 3, 2014. Retrieved May 22, 2015 from Unity: <http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/>
- [2] Ringo, D. History of Gaming: A Look at How It All Began. Public Broadcasting Service, n.d. Retrieved October 20, 2014, from PBS: <http://www.pbs.org/kcts/videogamerevolution/history>.
- [3] Tutorials. Unity Technologies, n.d. Retrieved October 15, 2014, from Unity: <http://unity3d.com/learn/tutorials/modules>.
- [4] Creighton, R. Unity 4.x Game Development by Example Beginner's Guide, 3rd ed. Packt Publishing, Birmingham, UK, 2013.
- [5] Edgar, T. Latex-slides. Department of Mathematics Pacific Lutheran University, Tacoma, WA, September 2011. PDF.
- [6] A Brief History of Roguelikes, Kill Screen Daily. Retrieved November 14, 2014 from Kill Screen Daily, INC: <http://killscreeendaily.com/articles/brief-history-roguelike/>.
- [7] Procedural Content Generation Wiki. Retrieved December 4, 2014 from: <http://pcg.wikidot.com/>.
- [8] Cadet, H. A Brief History of C Sharp. N.p., n.d. Retrieved November 13, 2014 from Hernando Cadet Technology: <http://www.hernandocadett.com/content/brief-history-c-sharp/>.
- [9] Elias, H. Perlin Noise. Retrieved December 6, 2014 from: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.
- [10] Moles, S. How Does Unity Use C# as a Scripting Language? N.p., n.d. Retrived: November 13, 2014 from Game Development Stack Exchange: <http://gamedev.stackexchange.com/questions/51350/how-does-unity-use-c-as-a-scripting-language>.
- [11] About Object-Oriented Graphics Rendering Engine. Retrived November 2014 from Torus Knot Software: <http://www.ogre3d.org/about/>
- [12] Stevens, P. and Tenzer J. GUIDE: Games with UML for Interactive Design Exploration. *Laboratory for Foundations of Computer Science School of Informatics*. Retrived November 16, 2014 from University of Edinburgh: <http://homepages.inf.ed.ac.uk/perdita/guide.pdf>.

- [13] UML for games. November 2003. Retrived November 16, 2014 from GameDev.Net LLC: <http://www.gamedev.net/topic/192120-uml-for-games/>
- [14] Patel, A. Introduction to A*. Red Blob Games, 1997. Retrived November 18, 2014 from Stanford University: <http://theory.stanford.edu/~amitp/GameProgramming>.
- [15] Effectively Organize Your Game's Development With a Game Design Document. Tutsplus. Gamux. Envato, November 11, 2011. Retrived November 18, 2014 from Tutsplus: <http://code.tutsplus.com/articles/effectively-organize-your-games-development-with-a-game-design-document-active-10140>.

6 Glossary

AI - Artificial Intelligence, in our case directing the actions of the NPCs.

Boss - A unique hostile entity that appears at the end of a level.

Entity - An object that can interact with other objects in the game.

Field of Play - The area in which a player will interact with obstacles, enemies, and other events.

HP - Health points, a measure of the players health level often represented as either a fraction or bar.

HUD - Heads Up Display, the in game area of the screen that displays player statistics like health or ammo left.

Level - A collection of similarly challenging fields of play culminating in a boss.

NPC - Non Player Characters, e.g., any enemies or passive characters that act in the game that are not controlled by the player.

Prefab - A function supported by Unity that allows for predefined object specification and quick entity instantiation.

Roguelike Game - A game that is distinct because of two important characteristics: Has highly randomized gameplay with a large amount of replayability. Permanent death, meaning if the main protagonist dies, the game ends and the player must restart from the beginning.

Sprite - The image that represents entities in the game.

Unity - The primary game engine we will be using.

Unity3D - The software kit used to develop games in Unity.