Two-Dimensional Space Shooter Game with Roguelike Gameplay Mechanics

George Bateman BS CS
Jacob Cannon BS CS
Nicholas DeVoll BS CS
Madison Silva BA CS

Contents

List of Figures

1 Introduction

Video games have been a rapidly growing industry in the past 30 years [1]. 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 will include random usable items and equipment, permanent player death, and randomly generated level design. This combination will aim to provide players with new challenges that may not be available in current games. This game will be developed using Unity3D and will run in the Unity engine, which features robust tools that will allow us to manipulate nearly any element of the game quickly and with immediate feedback.

1.1 Research

For the development of this project, we will develop a video game using Unity. The majority of the research time will be used to learn the functionality of Unity. Since Unity is a complete game engine and editor, the game creation process will not require the development of an engine. Other alternative engines, such as the graphics engine Ogre3D, would still require the implementation of a game engine in addition to creating the game [10]. Unreal Engine 4 was another considered engine, and included many features that would be useful to this project. However, many individuals have said that it is difficult for beginners to learn due to the lack of tutorials. Ultimately, we settled on 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 primarily uses $C\#$ as its scripting language. $C\#$ was created by Microsoft in 1997. It uses the .NET framework and is the most popular .NET language used today. It was influenced heavily by Java and C++, and it is intended to be friendly to the software development process so it has features such as uninitialized variable compatibility and automatic garbage collection [7]. 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. [9]

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. [5]

While developing non-player entities it is crucial to have an AI. Without an AI the entity becomes a stationary object and if they are an enemy that is not very exciting. One of the first step in creating a basic AI is choosing a pathfinding algorithm, so that the entity will move properly. An example of a basic but effective pathfinding algorithm is A*. Amit Patel defines A* as a combination of Dijkstras Algorithm and the Best-First-Search Algorithm [13]. Dijkstras Algorithm searches all cells around the starting point moving out and searching the next farthest out cells in each direction until it finds the goal. In a game where time is valuable searching in the way would be inefficient. Best- First-Search(BFS) is faster because it searches only in the direction of the goal. This algorithm is not efficient for games because if there is an obstacle then the path that BFS takes can become inefficient. A* searches all vertices by favoring the ones in the direction of the goal, this helps it be more efficient like BFS but avoid obstacles more like Dijkstras. [13]

Finally, procedurally generated content will 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 the common ones being 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. [6]

2 Gameplay

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 wave spawns. In this first wave, only a few enemies will be created at the other side of the field. The player must find and defeat these enemies to clear the field and move on to the next wave.

Gameflow:

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 will be 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 will have 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. **PlayerShield** 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: **GameComplete**
Participating Actor Instances: **Player: PlayerEntity**
Boss: HostileEntity
Bullet: Projectile

Flow of Events:

1. **Boss** is destroyed by a **Bullet** shot by the **Player**.

2. **Boss** will enter an animation where it explodes.

3. **Player** enters a five second grace period where they must survive.

3

a. **Player** does not survive the grace period and dies, restarting the game.

b. **Player** survives the grace period.

    i. **Player** is treated to a victory screen.

    ii. **Player** is given the option to play again or quit.

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**
                                        Wave: float
                                        GC: GameController

Flow of Events:

1. **GC** waits a set amount of time before attempting to spawn a wave.

2. The allotted amount of time has passed.

    a. **Wave** is not equal to the number of total waves.

        i. Enemies are spawned in according to how many waves the player has defeated.

    b. **Wave** is equal to the number of total waves.

        i. The boss is spawned in and the player must fight them 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 ↑↓←→ 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 move in the way of the second directional input.

4. **Bullet** travels in the direction it was initially shot without the possibility of changing direction.

Scenario Name: **SpawnAsteroids**

Participating Actor Instances: **Enemy: EnemyEntity**
                                               Asteroid: AsteroidEntity
                                               GC: GameController

Flow of Events:

1. **GC** performs a check to see if an asteroid will spawn.

2. The check fails.

    a. Nothing occurs, and **GC** waits before attempting to spawn another.

3. The check passes.

    a. An **Asteroid** object is spawned with a random position, movement vector, and rotation speed.

    b. The **Asteroid** careens through space until it hits something or goes too far away.

        i. 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 a short distance.

Scenario Name: **SpawnPickUp**
Participating Actor Instances: **Player: PlayerEntity**
        Enemy: EnemyEntity
        Shotgun: PickUp

Flow of Events:

1. **Player** destroys an **Enemy** with its weapon.

2. **Enemy** rolls a check to see if it drops anything.

3. The check succeeds, and **Enemy** drops a **Shotgun** pickup.

4. **Player** touches the **Shotgun**.

5. The **Shotgun** is equipped, giving **Player** 50 ammo and the **Shotgun** weapon.

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 Use Case Model

Figure 1 shows which external actors participate in each of the Use Case Scenarios included.
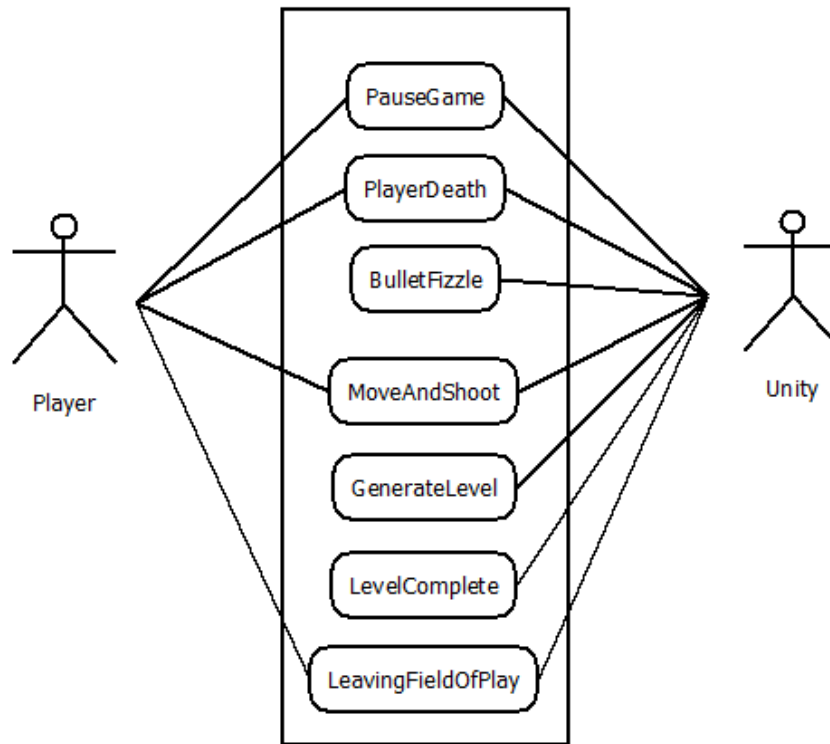

Figure 1: Use Case Model

4 Design

4.1 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.
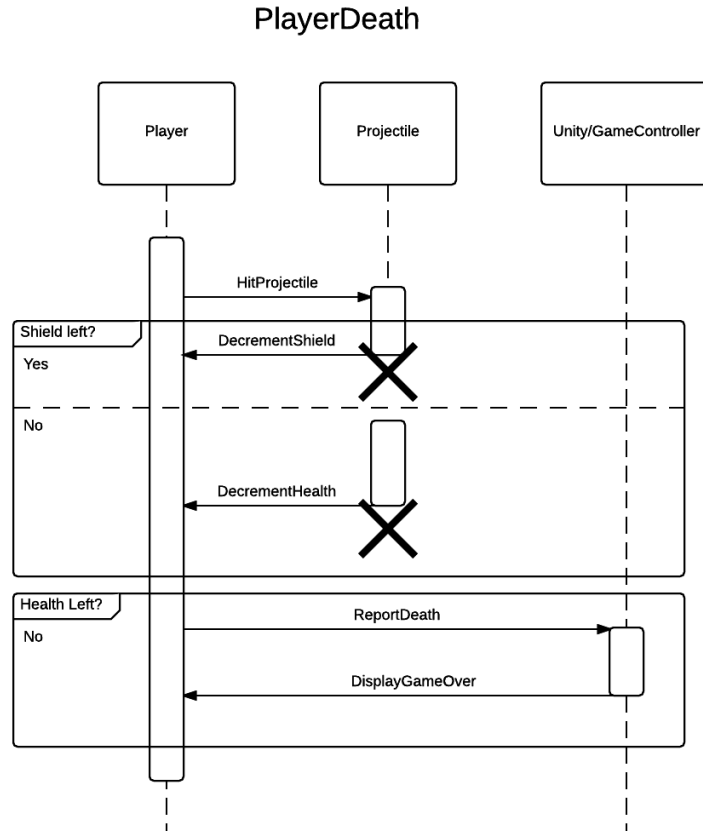
PlayerDeath

Figure 2: Player Death Sequence Diagram

4.2   UML Diagram

In Figure 3 shows the core class design for the game and the relations that classes will have with each other. Classes will be interacting with existing classes of the Unity engine. *GameManager* will be a singleton class that will be responsible for holding the *Player Entity*, this way the game can easily transport 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

8

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*.
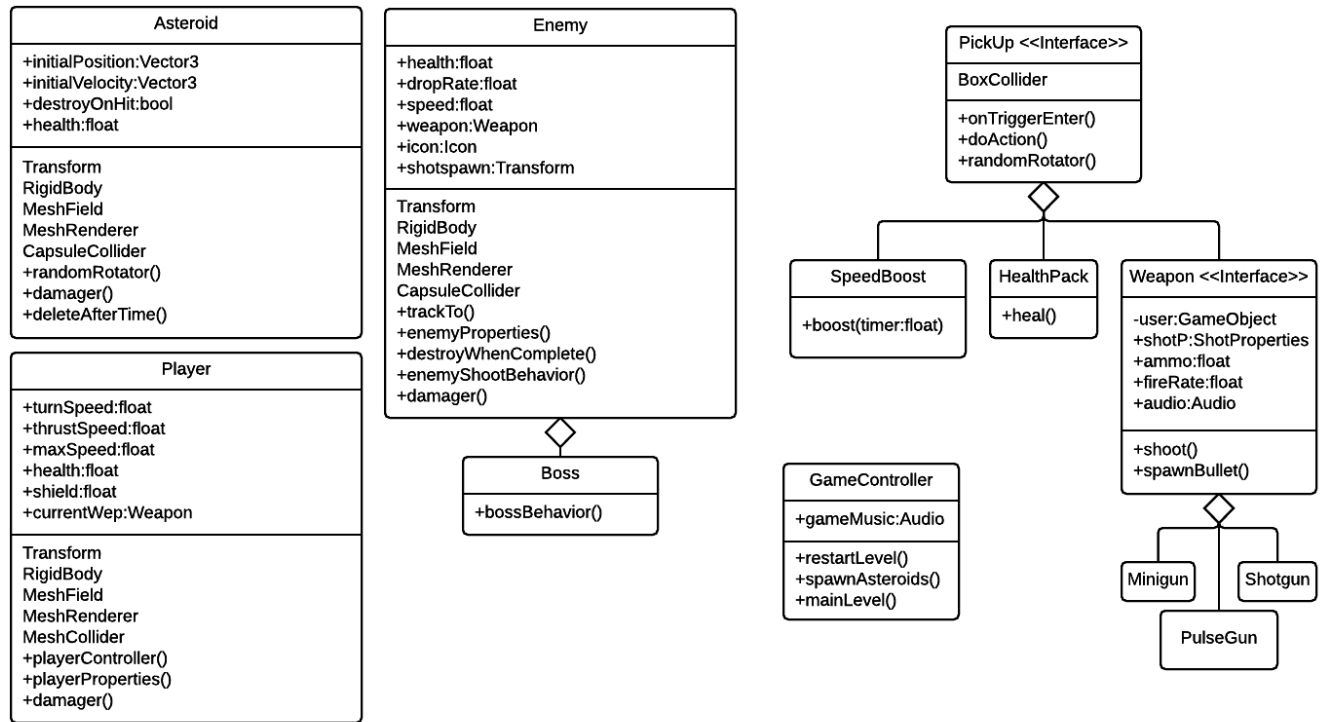

Figure 3: UML Diagram

4.3   State Diagram

Figure 4 describes the transitions between various scenes and menus in the game. Each state corresponds to either a Unity scene or a GUI child menu.
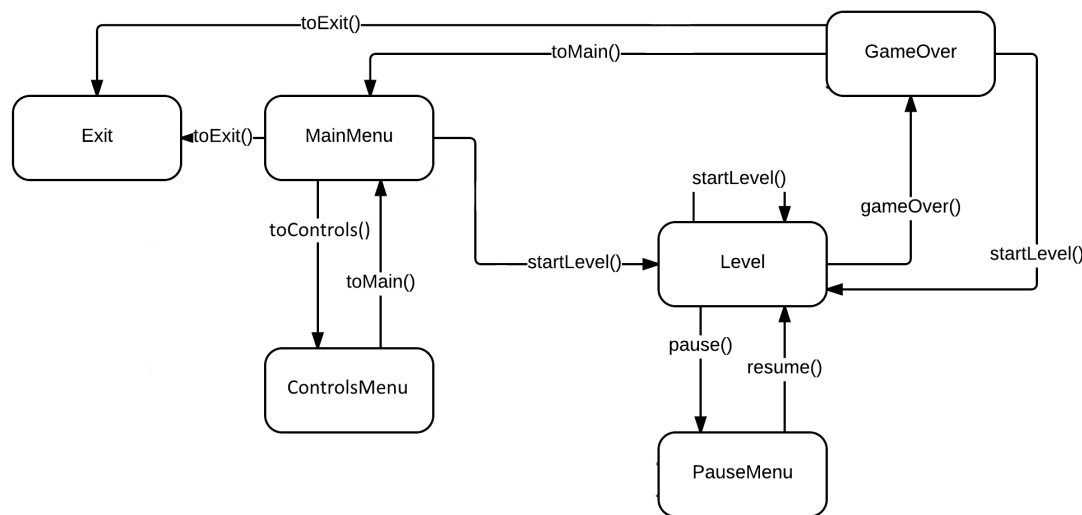
Figure 4: State Diagram

Entity Relationship model: ???

Testing plans: The group members will play the game every step of the way to make sure that the individual additions to the project are functioning as expected. Once there is a version of the game that allows a player to navigate an environment and fight basic enemies, the group will reach out others to play it and give feedback in the form of a simple survey.

5 Gameplay Design

Games are dynamic pieces of software that focus on delivering exciting or interesting 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.

5.1 Player

Player controls: The player controls a small spaceship, and navigates through a space environment. As seen in Figure 5, 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. The controls diagram is located at Section 5.7 of this document.

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. Rotation has a maximum speed, so quickly changing direction causes 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. Be 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 (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 players 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 players ship takes permanent health damage. Recharging of the shield will occur over time after a few seconds of avoiding damage. The ships health is the amount of damage the ship can sustain before it explodes and the player loses. It will not regenerate automatically like the shield, but items will be able to restore it.

5.2   Enemies

Enemies: The main goal of enemy ships is to kill the player ship. They will have two main types of AI that will decrease the predictability of enemies and prevent the player from staying still for too long.

Standard Shooting: Standard shooting is when the enemy just shoots straight in the direction it is currently facing. This is usually in the direction of the player.

Predictive Shooting: Predictive shooting is when the enemy uses a special method to predict where the player will be when the shot reaches it. This creates an effect where the enemy is shooting in front of the player so that the shot hits the player most of the time.

Patrolling: In order to patrol the enemies will set their spawn point to "home". Every time they are to far away from the player to track them, they will return "home" and circle

around it until the player comes back in to range.

Enemy Type One: The first type of enemy tracks directly toward the current location of the player while it moves. It will try to stay a small ways away from the player to avoid colliding with them. This enemy type has a high possibility of shooting at a fixed time interval, using standard shooting.

Enemy Type Two: 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. This type uses standard shooting with medium damage shot, firing less frequently than the first type.

Bosses: Bosses will give the player a final challenge before ultimately reaching the end of the game and winning. Bosses will shoot large amounts of projectiles to making dodging difficult for the player. Additionally, the boss will have a predictive shooting projectile 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.

5.3 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.

1. **Shotgun:** The shotgun has the same damage and rate of fire as the default weapon. However, every shot of the shotgun fires five bullets in a short, fixed spread, rather than one bullet straight ahead. Has 50 ammo.

2. **Pulse Gun:** The pulse gun is a high damage, low rate of fire gun that shoots one large pulse of 70 green bullets in the direction the player presses. Bullets fired by this gun do not destroy themselves when colliding with an entity, allowing for a "piercing" effect. Has 10 ammo.

3. **Minigun:** The minigun has a much faster rate of fire than the default gun. However, its damage is half of the default gun's per shot and the exact direction the bullets will be fired in has a variance, not guaranteeing accuracy. Has 400 ammo.

5.4 Levels/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 UI (Section 5.5), 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.

Level design: The level consists of a circular arena. There is a circular border around the playing field that the player will be shot at if he crosses. Enemies spawn in waves inside the arena but outside the player's view. Asteroids spawn in and around the arena. As the player destroys waves of enemies, new larger waves will spawn. After completing a set number of waves, the boss will spawn in the arena.

Star Field and Background:

Asteroids: Asteroids will be obstacles that will fly across the screen, dealing damage to anything they collide with. This will force players to be more aware of their surroundings. These obstacles have no set objective and can collide with each other as well as enemy's ships. (Add how they are spawned)

5.5   User Interface

Radar: The radar will be displayed in the bottom left corner. It shows the player the location of all enemies, pick-ups, and asteroids in the immediate surrounding area of the player. In addition, it also represent the direction of all enemies not shown in the immediate area with a symbol on the edge of the radar.

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

Menus:

5.6   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, there is a gun from one asset pack combined with 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.
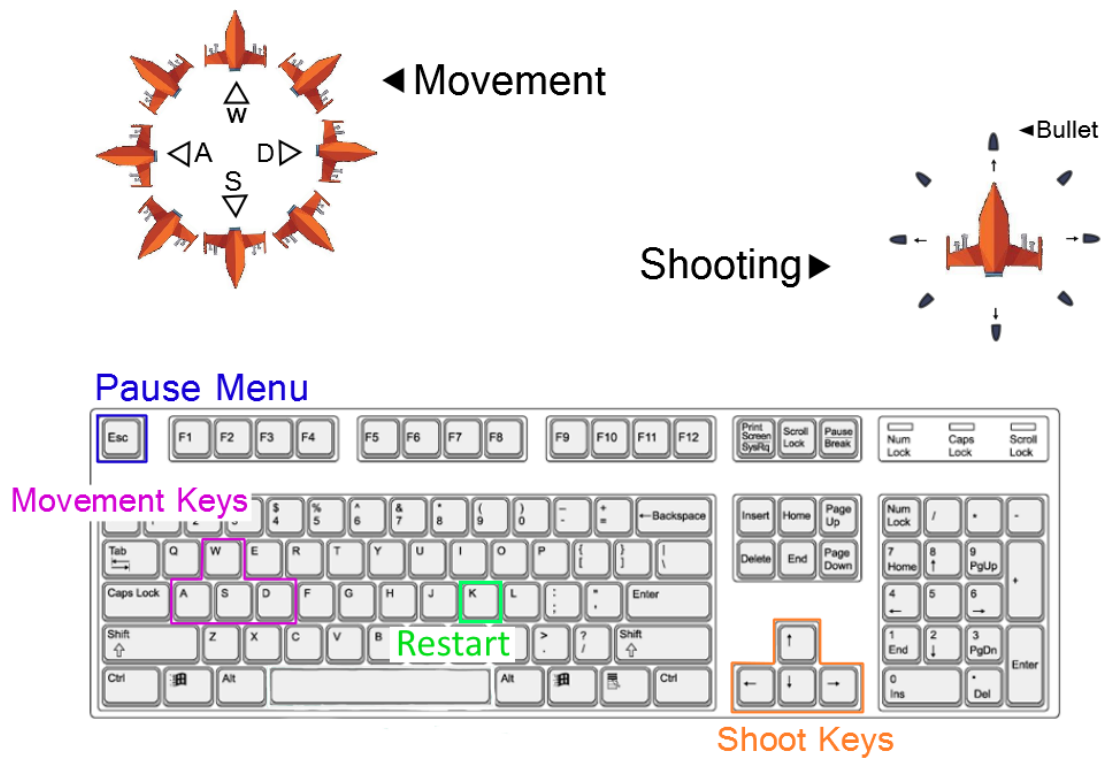
5.7 Controls Diagram

Figure 5: Controls Diagram

6 Implementation

This section describes the completed work, current status, and future plans that are to be included to the project.

6.1 Work Completed

The Unity website offers in-depth tutorials on how to create basic games using the Unity engine. We have completed two specific tutorials in preparation for this project: The Beginner rated project, entitled Space Shooter, and the Intermediate rated project, Project: Stealth.

The Space Shooter tutorial introduced us to the basics of Unity, and explained how to use the engine to create a top-down, two-dimensional space shooter game using given assets, including a three-dimensional ship model, a provided background, and models for asteroids and enemy ships. It also provided sample scripts to use for the completed game, written in $C\#$. The completed project spawned varying waves of asteroids that the player would have to either shoot or dodge, with a score counter incrementing every time the player shot an asteroid and the game ending when the player took damage from colliding with an asteroid and dying.

The Project: Stealth tutorial went into much more depth than the Space Shooter tutorial, and guided us through making a fully functional three dimensional stealth-based game. Using provided assets, the tutorial showed us how to use animations to change collisions, enemy AI that reacted to various events that the player had some limited control over, and advanced $C\#$ scripting work that went into more detail than the basics in the Space Shooter tutorial. The finished project was a fully functional game in which the player would attempt to reach the end of a level without alerting various AI guards.

We intend to use many of the elements presented in both of these tutorials as building blocks in our own game. From the Space Shooter tutorial we intend to use the fundamental concepts presented in the tutorial, namely scripting, collisions, and two-dimensional top-down shooter action, in our product. From the Project: Stealth tutorial we intend to use some of the overall concepts, such as enemy artificial intelligence and events occurring in response to our game state in our product.

6.2 Current Status

Development is starting to move out of preliminary stages as we get the group back together for spring semester. We have finished mutiple Unity tutorials and have a basic foundation of the game. We are currently working on creating a new time table to get develpment on track. The design document was completed and turned in on December 18th, 2014, and an updated version of the document was turned in February 19th, 2015.

6.3 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.

References

[1] Ringo, D. History of Gaming: A Look at How It All Began. Public Brodcasting Service, n.d. Retrieved October 20, 2014, from PBS: http://www.pbs.org/kcts/videogamerevolution/history.

[2] Tutorials. Unity Technologies, n.d. Retrieved October 15, 2014, from Unity: http://unity3d.com/learn/tutorials/modules.

[3] Creighton, R. Unity 4.x Game Development by Example Beginner's Guide, 3rd ed. Packt Publishing, Birmingham, UK, 2013.

[4] Edgar, T. Latex-slides. Department of Mathematics Pacific Lutheran University, Tacoma, WA, September 2011. PDF.

[5] A Brief History of Roguelikes, Kill Screen Daily. Retrieved November 14, 2014 from Kill Screen Daily, INC: http://killscreendaily.com/articles/brief-history-roguelike/.

[6] Procedural Content Generation Wiki. Retrieved December 4, 2014 from: http://pcg.wikidot.com/.

[7] 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/.

[8] Elias, H. Perlin Noise. Retrieved December 6, 2014 from: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[9] 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.

[10] About Object-Oriented Graphics Rendering Engine. Retrived November 2014 from Torus Knot Software: http://www.ogre3d.org/about/

[11] Stevens, P. and Tenzer J. GUIDE: Games with UML for Interactive Design Exploration. *Laboratory for Foundations of Computer Science School of Informatics.* Retrieved November 16, 2014 from University of Edinburgh: http://homepages.inf.ed.ac.uk/perdita/guide.pdf.

[12] UML for games. November 2003. Retrived November 16, 2014 from GameDev.Net LLC: http://www.gamedev.net/topic/192120-uml-for-games/

[13] Patel, A. Introduction to A*. Red Blob Games, 1997. Retrived November 18, 2014 from Stanford University: http://theory.stanford.edu/ amitp/GameProgramming.

[14] 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.

7 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.

Parallax - The effect whereby the position or direction of an object appears to differ when viewed from different positions, e.g., elements far from the camera appear to move more slowly than foreground elements.

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

Ramming - When one entity attacks another through direct collision.

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.

Usable Items - Items the player can use by pressing the use item key.