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

Faculty Mentor: Dr. Jorge Reyes Silveyra
May 15th 2015

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 with their chosen ship, and can immediately start exploring the area in which they spawn. They will see the controls of the game described on the field, introducing new players to the game. After discovering how to control their ship, players will be able to leave the starting area in a single direction, leading them to the first area of the game. In this first area, enemies will be created at the other side of the field. The player must defeat these enemies to clear the field and move on. As the player moves about the field, the enemies start reacting, moving towards the player to defeat him. When battle resolves, any dropped items and currency are picked up by the player, and they are given a choice where to go next.

Gameflow:

Victory conditions: The player will achieve victory by completing each area that is generated without dying. The game is planned to culminate with a boss fight, and defeating this boss will trigger victory 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**.
    a. **PlayerShield** decreases based on DamageEntity.
    b. **PlayerHP** decreases based on DamageEntity.
        i. **PlayerHP** is greater than zero, continue as normal.
        ii. **PlayerHP** is less than or equal to zero, **Player** dies.
            1. **Player** is deleted.
            2. Explosion animation spawned in place of **Player**.
            3. An end game menu prompts **Player** to either play again or return to main menu.

Scenario Name: **LevelComplete**
Participating Actor Instances: **Player: PlayerEntity**
Boss: HostileEntity
Rocket: Projectile
Item: InteractiveElement

Flow of Events:

1. **Boss** is destroyed by a **Rocket** shot by the **Player**.
2. **Boss** will enter an animation where it explodes and drops an **Item**.
3. **Player** has a grace period where they can pick up the **Item** dropped by the boss, and then theyre transported to the shop and inventory management menus.

4. After they are done with the end of level menus, **Player** is taken to the beginning field of play for the next level.

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 returned to normal.
   b. **Player** advances farther out of the field of play.
      i. **PlayerKiller** kills **Player**.
      ii. PlayerDeath().

Scenario Name: **GenerateLevel**
Participating Actor Instances: **LG: LevelGenerator**
GM: GameManager

Flow of Events:

1. **GM** notifies **LG** that a level needs to be generated.
2. **LG** creates a level layout.
   a. If **LG** determines that the level layout is navigable and correctly generated
      i. The level is generated
   b. If **LG** determines the the level layout is not navigable and is incorrectly generated
      i. The **LG** creates another layout and checks again.

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. **Bullet** travels in the direction it was initially shot without the possibility of changing direction.
    a. The **Bullet** travels until it collides with another entity.
        i. The **Bullet** causes damage to that entity and terminates itself.
    b. The **Bullet** travels until it fizzles out of the view of **Player**.
        i. The **Bullet** terminates itself.

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: **PauseGame**
Participating Actor Instances: **Player: PlayerEntity**

Flow of Events:

1. **Player** presses the pause button.
2. A menu pops up with three options; resume game, options, and exit game.

a. If resume game is chosen, the menu exits and gameplay is resumed.

b. If options is chosen, the game state is maintained and the **Player** is taken to the options menu.

c. If exit game is chosen, a sub menu pops up giving three options; exit to menu, exit game, return to menu.

   i. If exit to menu is chosen, **Player** is taken to the main screen view.

   ii. If exit game is chosen, the game is closed.

   iii. If return to menu is chosen, the **Player** is navigated back to the prior menu.

3.2 Use Case Model

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

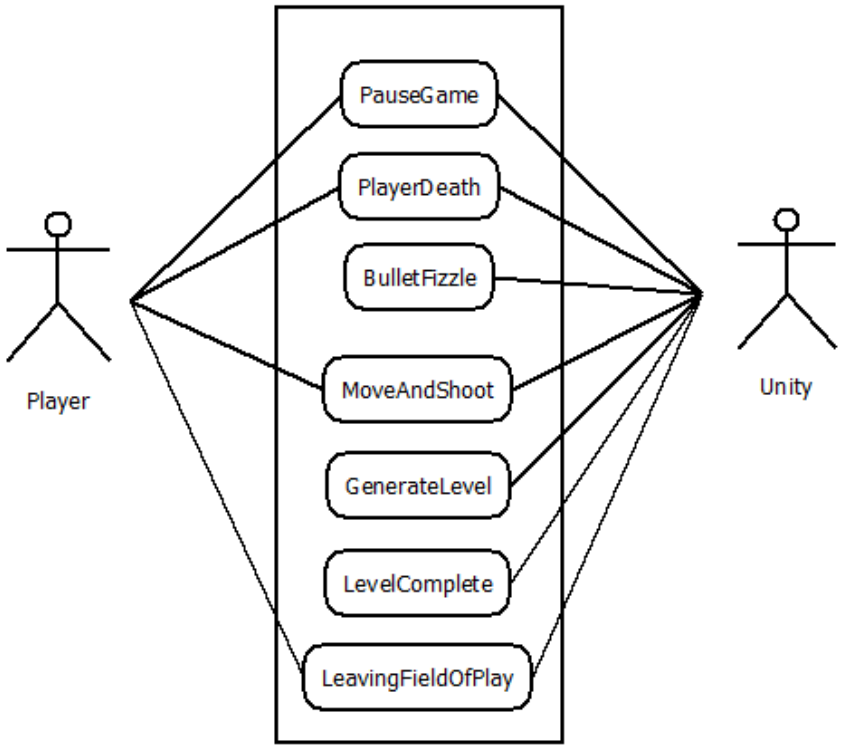

Figure 1: Use Case Model

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.

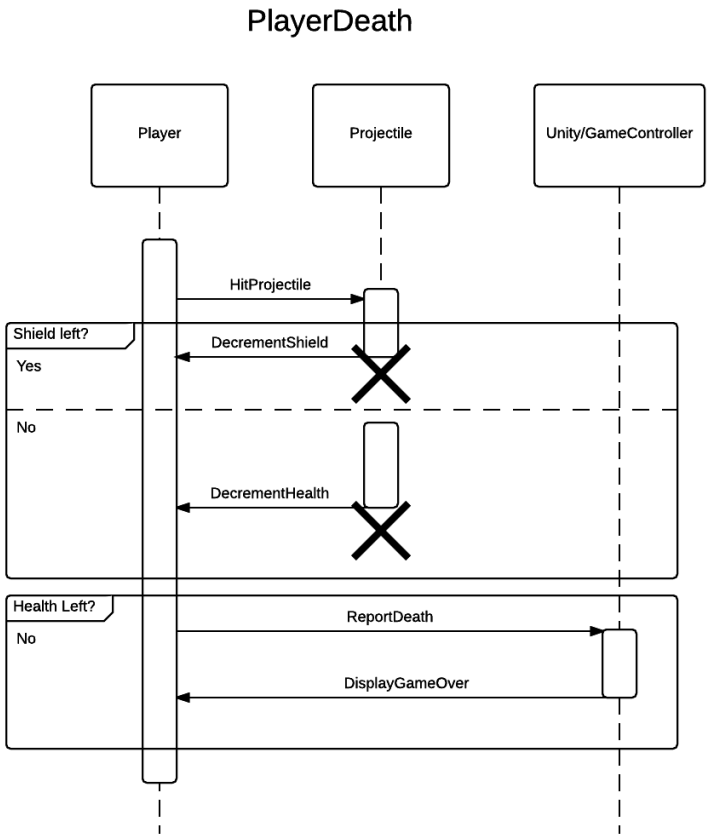

Figure 2: Player Death Sequence Diagram

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 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*.

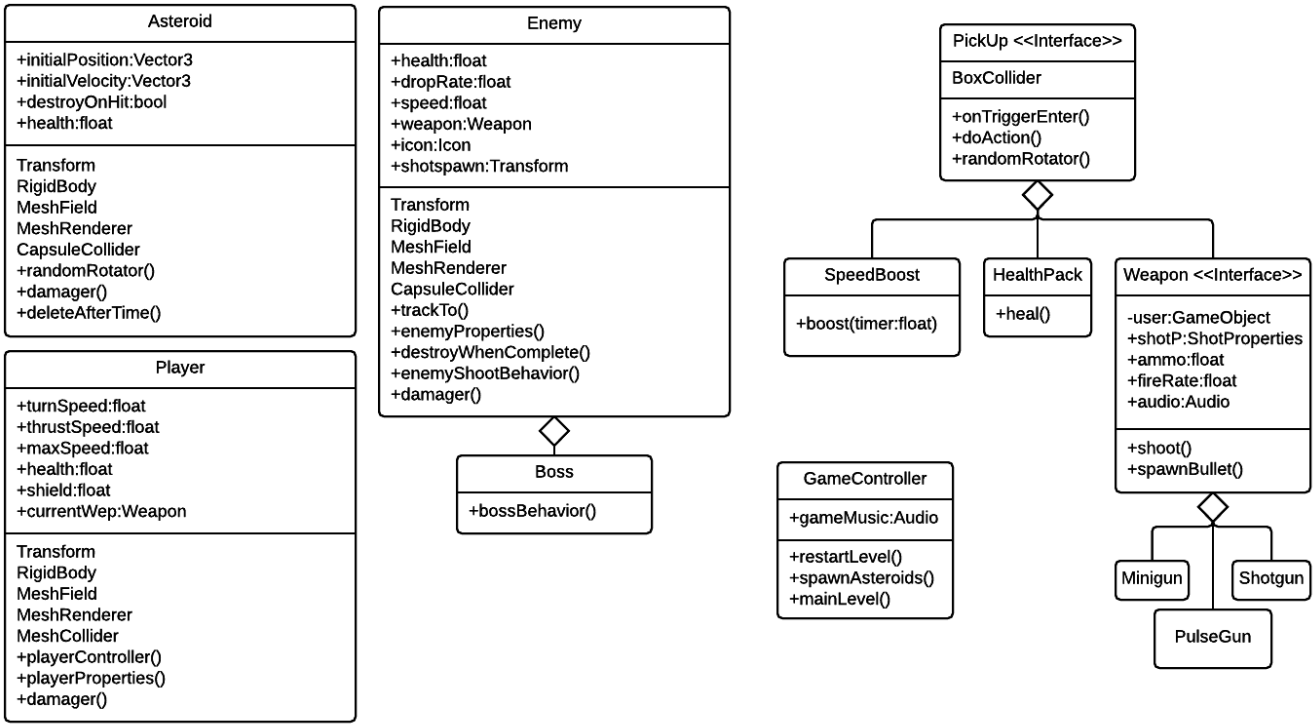

Figure 3: UML Diagram

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.

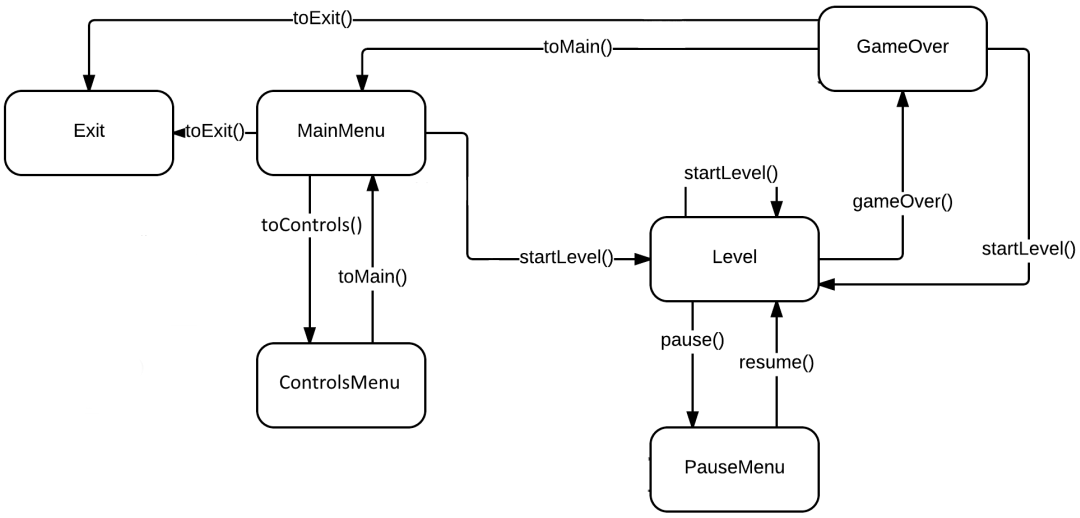

Figure 4: State Diagram

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. There are two main strategies the enemies will utilize to accomplish this. Certain enemies will shoot at the player, forcing the player to dodge if they want to survive, while other enemies will attempt to ram the player to deal damage. As such, there are enemy ships that will fly around in basic patterns while shooting at the player, and there are enemy ships that will charge straight at the player to prevent them from staying still for too long. Additionally, there will be enemy obstacles such as asteroids that will fly on screen to force players to be aware of their surroundings. These obstacles have no set objective and can hit enemy ships as well.

The main goal of the enemy ships is to kill the player ship. There are two main types of enemy that employ different tactics to accomplish this goal. The first type of enemy tracks the player while it moves, having the possibility of shooting at a fixed time interval. The second type of enemy will follow the player closely while trying to get in front, having a stronger shot then that of the first enemy, but also firing less frequently. Additionally, there

will be enemy obstacles such as asteroids that will fly on screen to force players to be aware of their surroundings. These obstacles have no set objective and can hit enemy ships as well.

Standard Shooting:

Predictive Shooting:

Patrolling:

Enemy Type One:

Enemy Type Two:

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 will need to serve two functions: give the player a variety of gameplay options and motivate the player to explore a different gameplay style. For instance, a laser may allow the player to shoot a long range beam that pierces enemies but does not deal much damage, while a shotgun may reward the player with a more damaging weapon, but require the player to engage with enemy ships from a much closer range. The player will be able to switch out weapons such as these to better fit their playstyle.

5.4 Levels/Objects

Pick-ups: There will be pick-ups available to the player throughout the game, dropping from enemies and asteroids. Pick-ups will change some attribute of the player ship, including speeding the movement of the ship, restoring health, and changing the currently equipped menu.

Items: The majority of the items in the game will provide a general bonus to the player without requiring additional action from the player. These could include upgrades like increased shield or hull capacity, increased damage, or faster ship acceleration. Additionally, there will be various items that the player can activate by pressing the Activate Item key that will provide the player with some benefit when used. For instance, a time bomb may temporarily slow down enemy ships and projectiles to allow the player to dodge them more easily.

Level design: Enemy placement will play a critical role in the design of levels. The placement of enemies needs to be dense enough so that there is a challenge but sparce enough so that every level is possible and does not feel the same. Since the game takes place in space, there will not be traditional level elements such as walls and floors. However, there may need to be other obstacles that guide the players movements such as debris or very large ships.

Star Field and Background:

Asteroids: (purpose and spawning)

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:

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 all be free from the Unity store. We will use our own creativity to make all of the assets blend together in a cohesive way. We are using a black with star background for the main level, and then varying space themed backgrounds for the other scenes. The font Huxely is through out the game to tie the art all together.

Music/ sound: Sound effects and music will be retrived from the Unity store and other creative commons sites. It will attempt to replicate a space environment similar to other space themed games. For example the laser gun will have a "pew pew pew" sound.

5.7 Controls Diagram

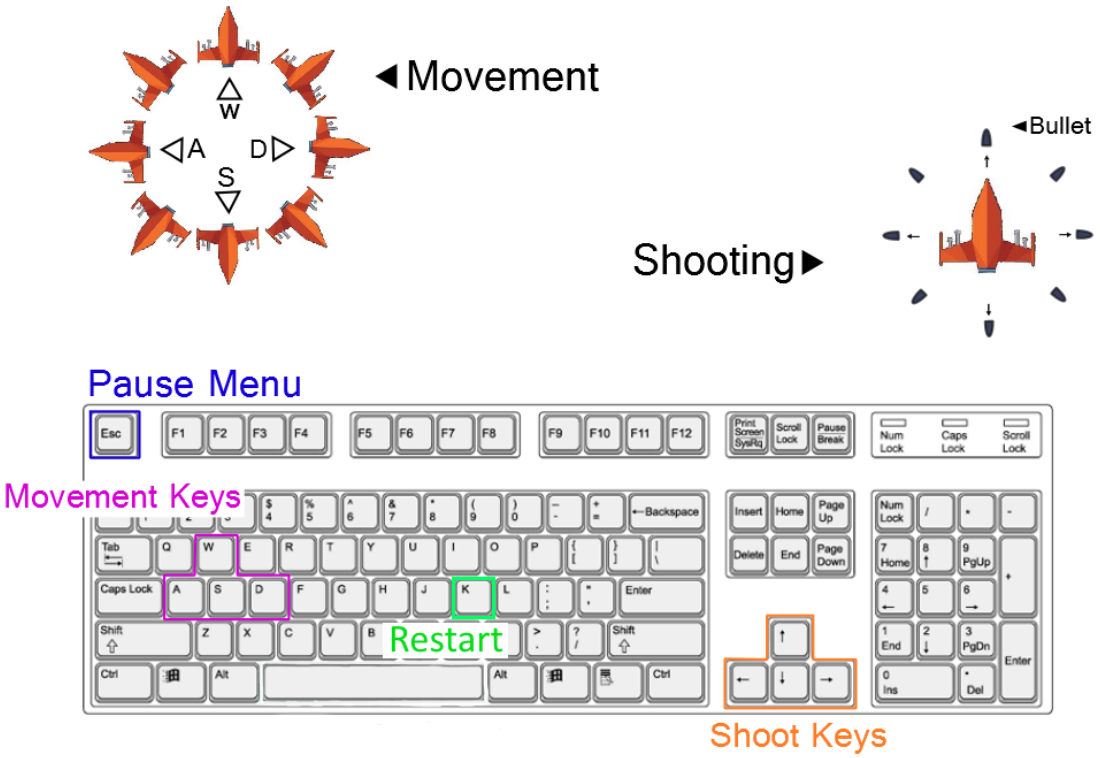

Figure 5: Controls Diagram

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

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*. Retrived 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.
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.