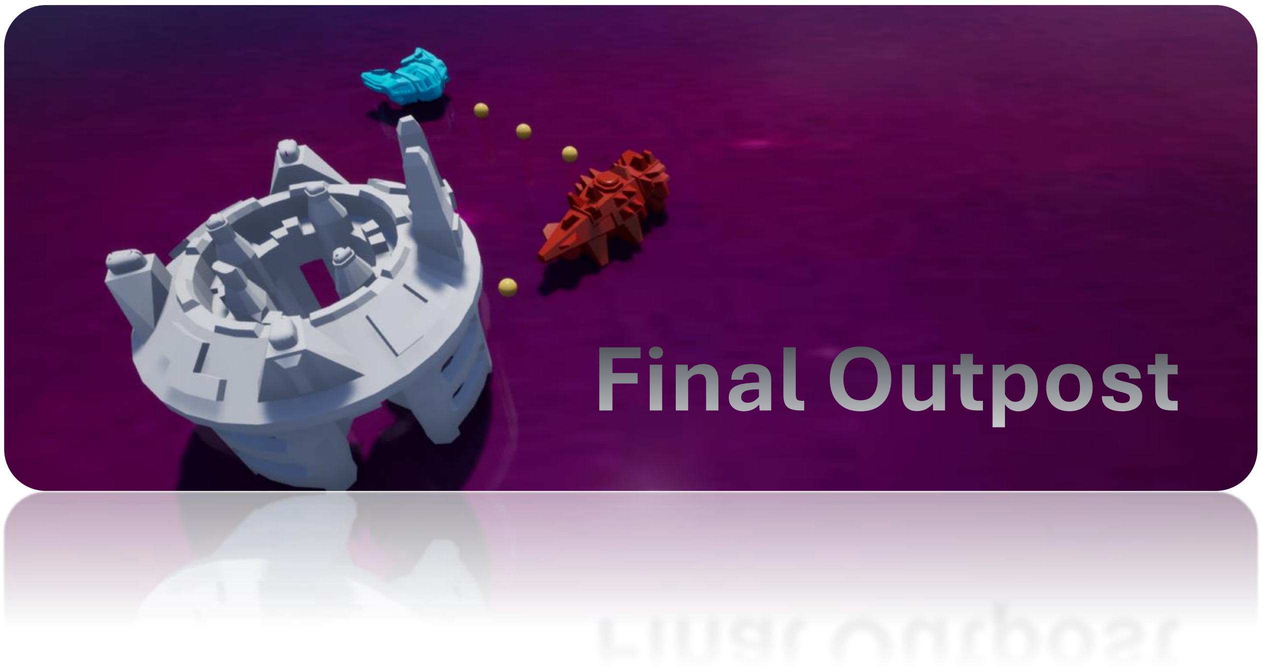
CPSC 599.11 – WINTER 2024

TEAM BED – FINAL REPORT



Brody Wells	Report Editor, Layout & Menu UI.
Elise Chevalier	GOAP Implementer, General AI, Gameplay Loop & Gameplay UI.
David Zevin	Powerups System & Gameplay Loop.

TABLE OF CONTENTS

GAME DETAILS

TITLE

'Last Outpost'

ENGINE

Unreal Engine 4.27.2

GENRE

Arcade Style

Twin-Stick Shooter / Wave Defence

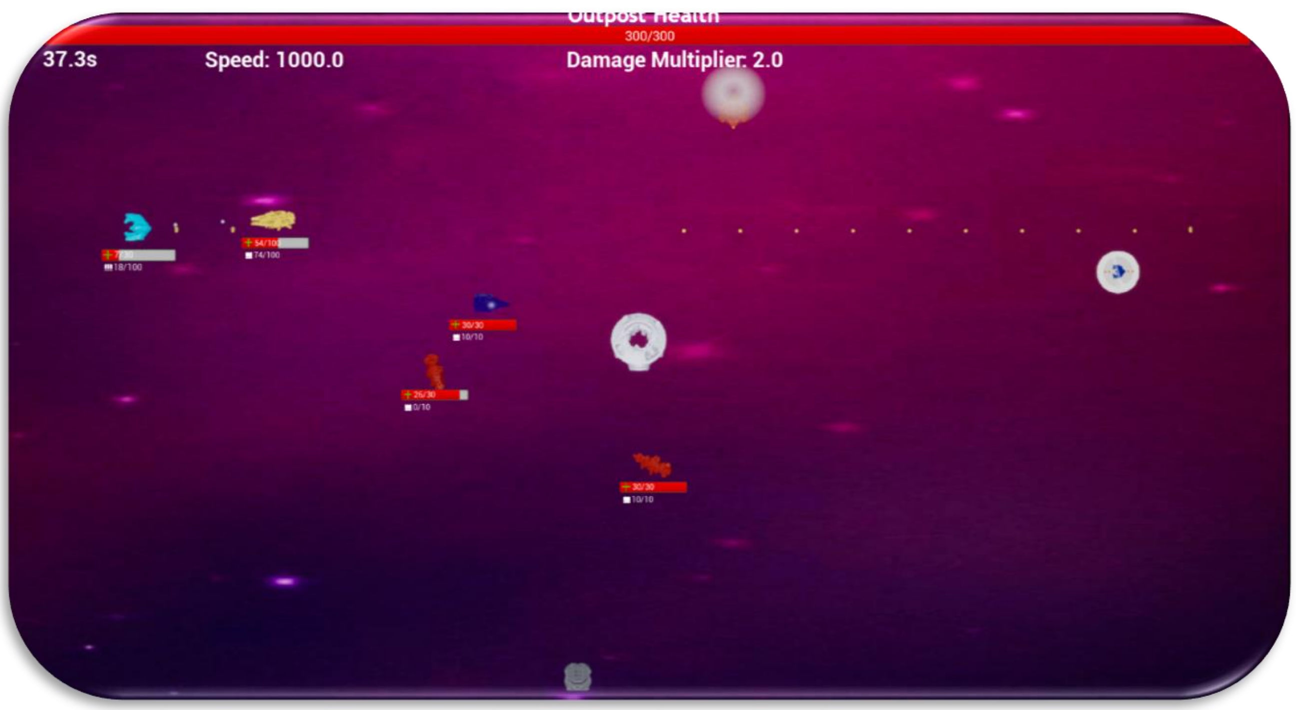
Controller: Keyboard or Gamepad

THEME

Spaceship Combat

CAMERA

Top down 2D (as shown)



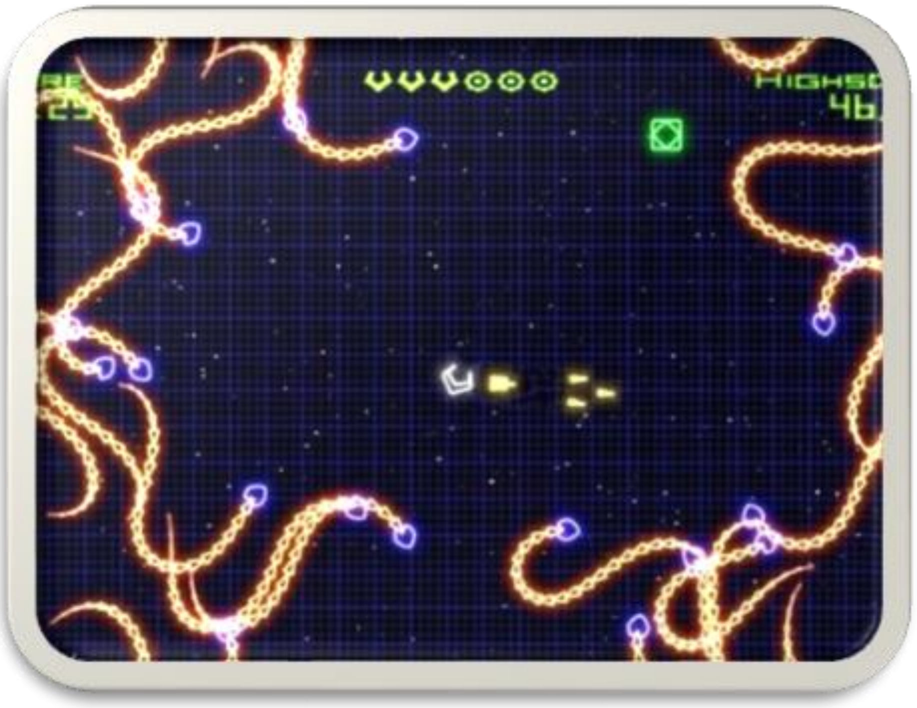
INSPIRATION

Twin-Stick shooter games where the player controls a spaceship. The player is tasked with surviving as long as possible as well as keeping the base alive, while avoiding and shooting obstacles.

GAMES IN THIS GENRE

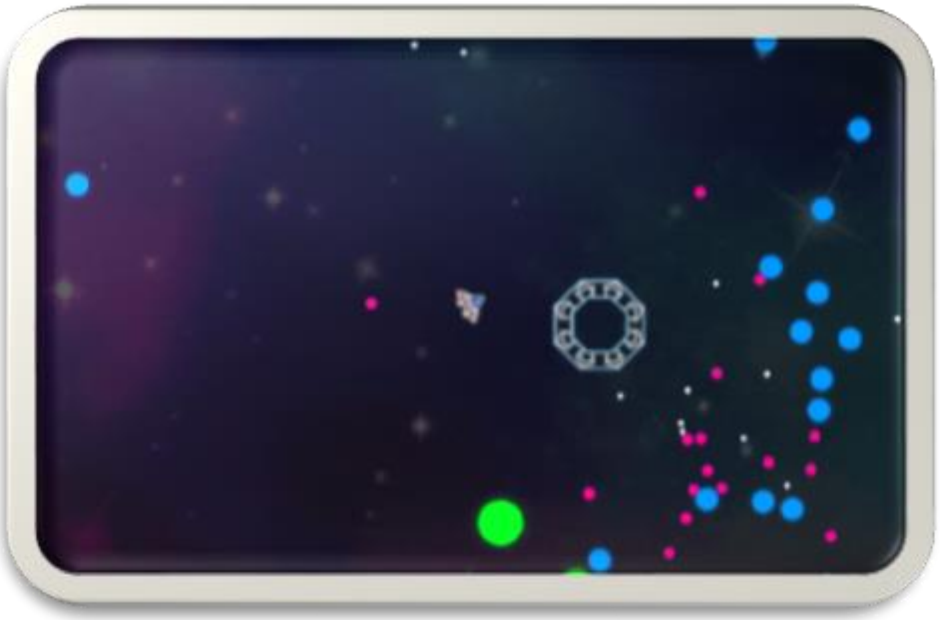
GEOMETRY WARS

Geometry Wars gameplay centers on controlling a small ship that can move and shoot in any direction within the screen space. Players need to dodge and destroy increasing numbers and types of enemies to stay alive and earn points. Each enemy behaves differently, pushing players to adjust their strategies.



BALLATILE

In Ballatile, a spaceship is maneuvered within the circular map with the aim to stay alive as long as possible. They must avoid hitting circular objects that randomly spawn and float around the map, increasing in quantity over time. Targets can break up into multiple smaller objects.



DESCRIPTION

Top down 2D Twin-Stick shooter, set in space, where the player controls a spaceship defending an outpost, the eponymous 'last outpost', from oncoming waves of enemy ships. Hostile ships spawn into space every ~10 seconds, attempting to destroy the base and the player's spaceship.

An arcade style scope, focused less on map detail and more on enemy interactions. An increasing number of enemy ships spawn, with more complex AI as the time progresses. The enemy ships will attempt to destroy the player and the base.

THE STORY

In the year 2547, humanity has ventured beyond the confines of Earth, establishing colonies across the galaxy. Amidst this era of exploration and expansion, a dire threat emerges. A relentless alien force, known as the Void Swarm, has begun its assault on human outposts, leaving destruction in its wake.

You are the pilot of the Starblade, an elite spacecraft engineered for combat and defense. Stationed at the forefront of the galaxy's defense, your mission is critical. As the last line of defense, you must protect the Final Outpost, a pivotal hub for human survival and resistance against the alien onslaught.

With each passing moment, the enemy grows stronger, their tactics more cunning. The fate of humanity rests on your shoulders. Prepare for an intense battle where every second counts. Your mission is clear: Survive. Defend. Prevail.

GOAL

Survive for as long as possible by defending the base from attack and avoid damage to the players spaceship. The score is determined by how long the player survives and a game clock.

CONTROLS

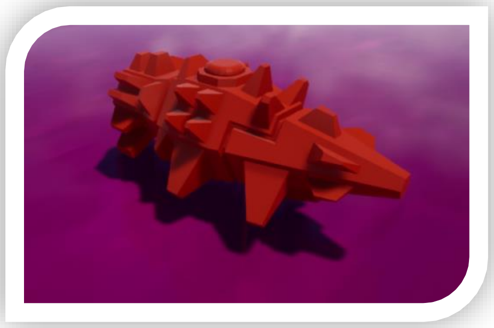
The player can control the spaceship movement with WASD or the left thumb stick on a controller. The player can fire shots from the spaceship with the arrow keys or the right thumb stick. Shots will travel in the direction pressed / held. The player can use bombs with spacebar or right trigger.

FAILURE STATE

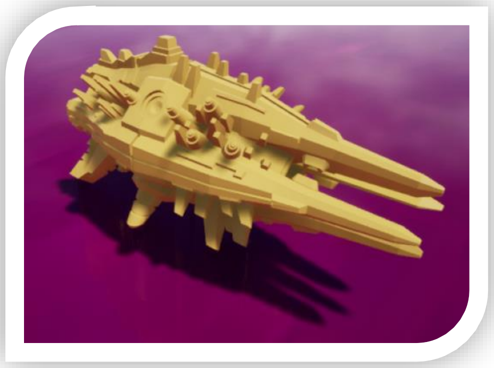
Base health or player health gets reduced to zero. It is undecided yet if the player may have multiple lives however once the base is fully destroyed, or the player has no lives, the game ends.

ENEMY CLASSES

FIGHTER

Fighters main goal is to attack the player ship. Their spawns are more frequent, but they are weak and light on fire power. If they take any hits, they must refill health often.	

HUNTER

Hunters main goal is to unleash a fury of artillery onto the player. They have heavy armor and a large ammunition supply in addition to a rapid-fire rate.	

BASE DESTROYER

Base destroyers' main goal is to attack the outpost, however if they player ship comes to close, they may change their goal to fire upon the player ship instead. Like the fighters, they are low on health and fire power and must refill ammo frequently.	

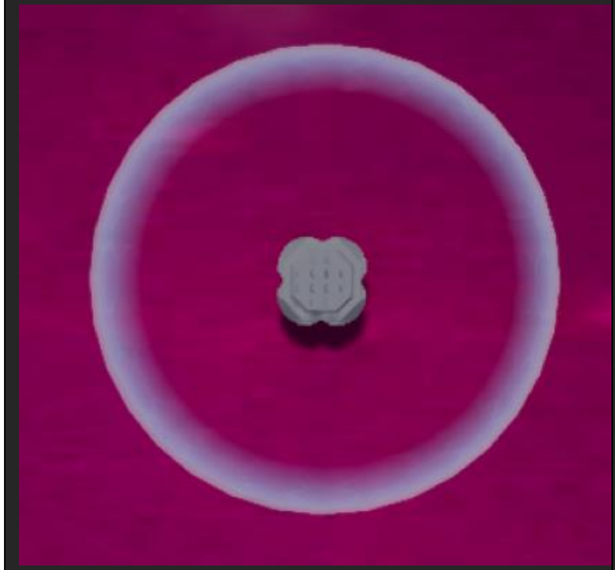
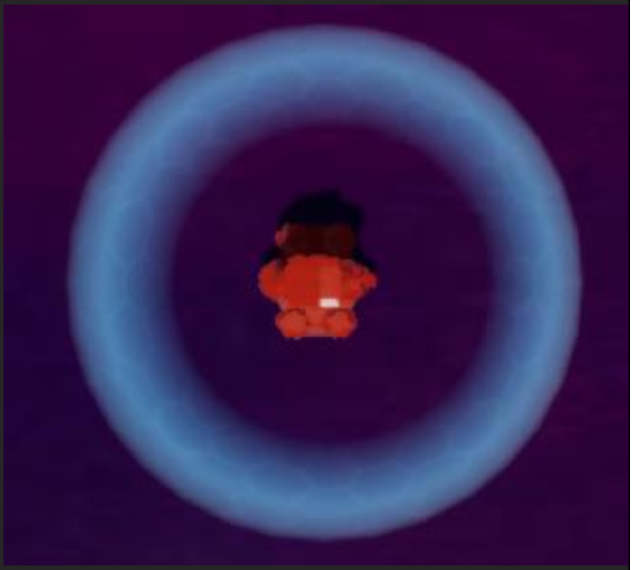
POWERUPS

Powerups spawn at random locations around the map at regularly timed intervals. Player must fly overtop the powerup before the enemy does to collect it. Some powerups are permanent and stacking, whereas others are temporary boosts.

Powerups				
Increasing maximum ammo capacity	Ammo supplies		Increases maximum health	Restores some health
Restores full health	Temporary speed boost	Drop Bombs with spacebar or RT. Recharges every 30s	Increases damage	Increases firing rate
Increases projectile speed	Shots are fired from multiple directions		Increases the number of guns	Increases the shots fired per gun. Bullets fan out from center.

HEALTH AND AMMO

Two refilling stains rotate around the outer edges of the map. When a player or AI is low on health or ammo, they can go to respective refill station to restore full capacity. The station refills at a constant rate and the ship must stay within range to continue to be refilled. Stations offer a force field that protects ships from attack while refilling.

Stations	
	
The ammo refill station appears gray in color	The health refill station appears red in color

AI & GAME DESIGN

BASIC DESIGN

This section outlines the core mechanics required for gameplay. The main objective involves progressively spawning enemy ships with distinct behaviors and visual cues, allowing players to recognize their type and abilities.

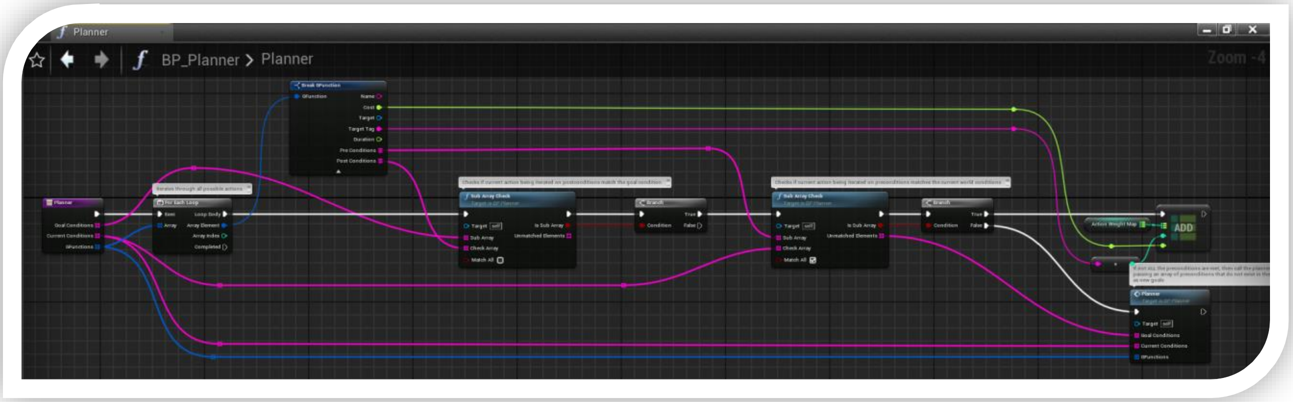
Enemy ships will target both the player and a central base, each having specific objectives. The game ends if either the player or the base's health depletes to zero.

GOAP SYSTEM

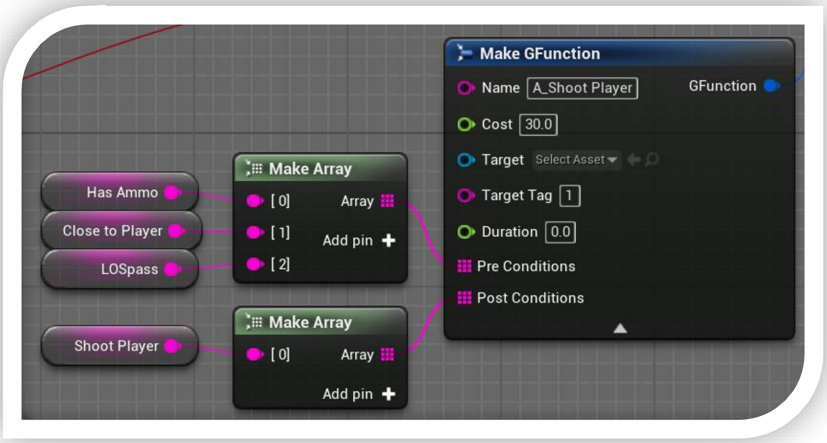
We choose to implement a GOAP system for enemies. Assigning different goals to different enemy types, with weighted priorities. The main goal for the enemy is to destroy the player or the central base, but intermediate actions would need to be executed depending on current conditions. For example, if the enemy's goal is to shoot the player it must have ammo, and if it does not then it needs to refill ammo, but to refill ammo it must be near the refill station, etc.

THE PLANNER

Enemy AI each call a central planning blueprint:



Input parameters to the planner is an array of string strings representing **goal conditions**, another array of string strings representing **current conditions**, and an array of structs **GFunctions**. The structs represent each action the enemy ship can perform, and the preconditions required, the effect it has (post conditions), and the cost associated:



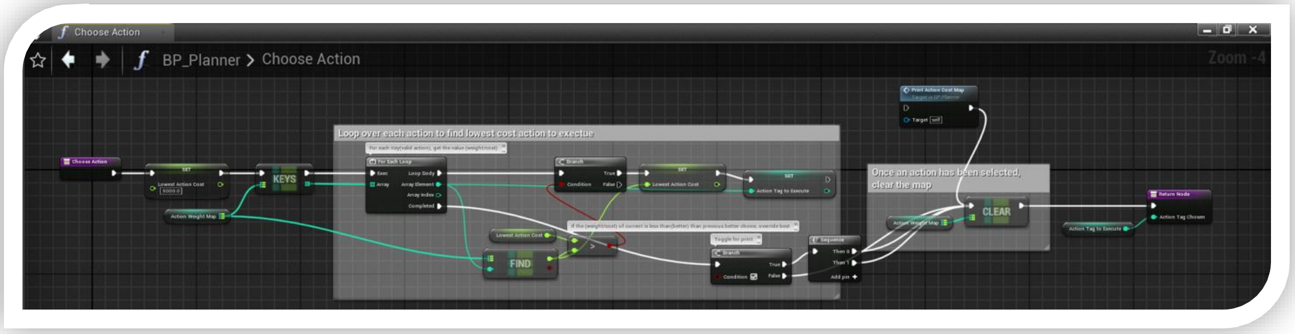
The planner recursively loops over this struct array to find an action path that leads from current conditions to the goal conditions. For successful paths, the total cost is recorded in an **action weight map.**

PLANNER – PSEUDO CODE

```
For each GFunction Struct (which represents an action)
    Check if post conditions match goal conditions
        If YES, then check if preconditions match the current conditions of the enemy
            If YES, then get the cost of action and add to an action weight map
            If NO, then recursively call the planner, but with new goal conditions being the
            unmatched conditions
        If NO, then continue
```

CHOOSE ACTION

Once the GOAP system recursively found all potential action paths it selects the lowest cost action to execute.



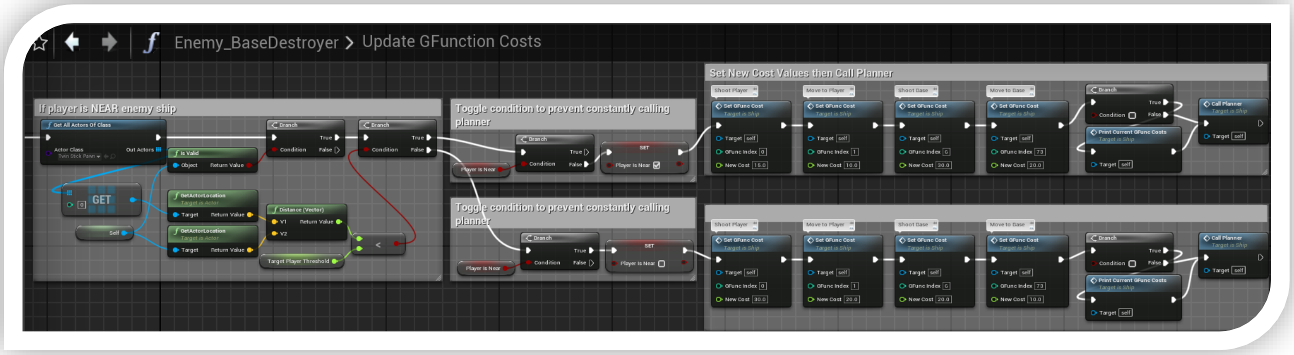
The action to execute is then sent back to the enemy pawn that called the planner. To save on CPU time, the enemy pawn will not call the planner again until it has a change to its current conditions.

CHOOSE ACTION – PSEUDO CODE

```
Reset current lowest cost action value

For each action in the action weight map

        If the action cost less than current lowest cost action

                Set current lowest cost action = this action

        Else continue

Clear the action weight map

Return action to execute
```

UPDATE GFUNCTION (ACTION) COSTS

To demonstrate dynamic updating of action costs, we added a function to the Base Destroyer enemy type called **Update GFunction Costs.** Unlike the other enemy AI types, the Base Destroyer has goal conditions of moving to and shooting player, as well as moving to and shooting the base. Unlike the other classes, the costs of each action is updated based on if the player ship enters a threshold distance around the Base Destroyer.


Enemy_BaseDestroyer > Update GFunction Costs
If player is NEAR enemy ship
Toggle condition to prevent constantly calling planner
Toggle condition to prevent constantly calling planner
Set New Cost Values then Call Planner
GET
Self
Target Player Threshold
Player is Near
Shoot Player
Move to Player
Shoot Base
Move to Base
Set GFunc Cost
Branch
Print Current GFunc Costs
Call Planner

UPDATE ACTION COSTS – PSEUDO CODE

```
Check if distance to player is less than the threshold value
    If YES check if it’s a change of condition
        If YES update actions costs (lower the cost of moving to and/or shooting the player)
        Call the GOAP planner
    If NO check if it’s a change of condition
        If YES update actions costs (Raise the cost of moving to and/or shooting the player)
        Call the GOAP planner
```

ACTIONS OF ENEMY PAWNS

Each enemy creates an array of structs when it first spawns. These structs represent actions the enemy ship can perform, the preconditions required, the effect it has (post conditions), and the cost associated. The system handles the following possible actions:

Action	Cost (e.g.)	Pre-Conditions	Effect (post-conditions)
Shoot Player	30	HasAmmo, CloseToPlayer, HasLineOfSight	Shoot Player
Shoot Base	20	HasAmmo, CloseToBase, HasLineOfSight	Shoot Base
Move to Player	20	FarFromPlayer	CloseToPlayer
Move to Base	10	FarFromBase	CloseToBase
Move to Ammo Station	10	FarFromRefillStation	CloseToRefillStation
Move to Health Station	10	FarFromHealthStation	CloseToHealthStation
Refill Ammo	10	CloseToRefillStation, NeedsAmmo	HasAmmo
Refill Health	10	CloseToHealthStation, NeedsHealth	HasHealth

The costs of actions can be changed as conditions change. For example, the base destroyer enemy will weight shooting the base lower than shooting the player, however if the player comes within a certain range of the enemy, it will dynamically reduce the cost of shooting the player, which will change its action execution from shooting the base to shooting the player.

PATHFINDING AI

The pathfinding used by the AI in game is the built-in A* pathfinding algorithm from Unreal.

WAVE SPAWNING SYSTEM

In the first two seconds of the gameplay beginning a lone basic enemy ship will spawn to allow the player to get used to the controls and encountering a single enemy ship. Every ~10 seconds after the beginning of gameplay a random assortment of ships will spawn.

Ship types are selected from an array, with weighted random distribution. The basic enemy ships have a 60% chance of being picked to spawn in the next wave, the base destroyers have a 30% chance of being picked, and the hunters have a 10% of being picked.

The number of ships to be spawned in the next wave is calculated as:

$$\frac{Number\ of\ waves\ already\ spawned}{2}$$

After 30 seconds have passed, each ship will receive one upgrade every 10 seconds. These upgrades also use a weighted random and have limits to prevent the enemies from becoming too strong. Movement speed has a 20% chance of being upgraded, fire rate 10%, health 25%, ammo count 15%, damage 5%, number of guns 5%, projectile speed 10%, multishot tier 5%. These values are all capped differently depending on the enemy type.

Stronger ships being spawned after the 30 seconds mark, is to provide the player with a balanced level of challenge. It is expected for the player to have grown in power from collecting powerups after these 30 elapsed seconds. The difficulty of the game is slightly randomized due to the randomized nature of the powerup spawning.

PLAYER ABILITIES

Movement using either keyboard controls or twin-stick gamepad for omnidirectional movement.

Shooting in the direction of Right stick if on controller and arrow keys if on keyboard.

Health system. Health bar with one life. Health bars update color based on health.

TEST CASES

Please note: In Case 1, the AI determines if there are any Ammo powerups nearer to it than the station is. It then goes to the nearest one. If the nearest powerup is taken by another AI or the player, it will update its destination again. The same goes for Case 2, but with Health powerups.

CASE 1 – AMMO REFILL

In this scenario the goal state of the enemy fighter ship is to shoot the player. There is a successful sequence of actions in can take to accomplish this goal, which is first to move to the player, then shoot the player. However, this can only continue until the enemy fighter runs out of ammunition.

Current Conditions: HasAmmo, FarFromPlayer, HasHealth

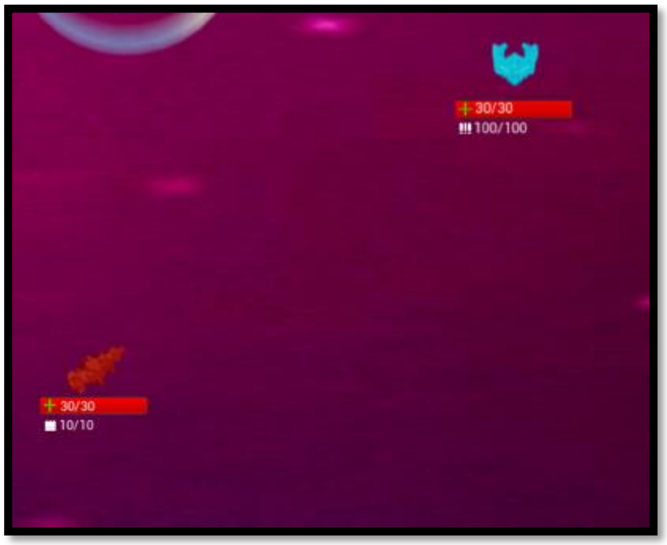
Goal: ShootPlayer

Successful action path: MoveToPlayer, ShootPlayer

Action: MoveToPlayer (Shown Below)

Pre: FarFromPlayer

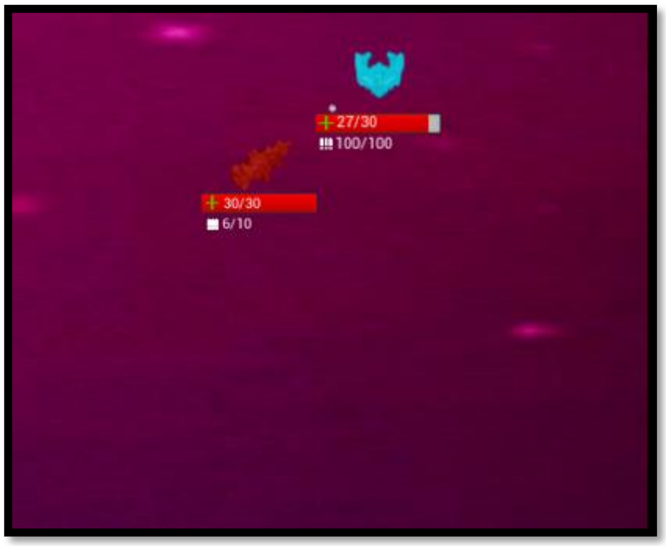
Eff: CloseToPlayer



Action: ShootPlayer (Shown Below)

Pre: HasAmmo, CloseToPlayer, HasLineOfSight

Eff: ShootPlayer



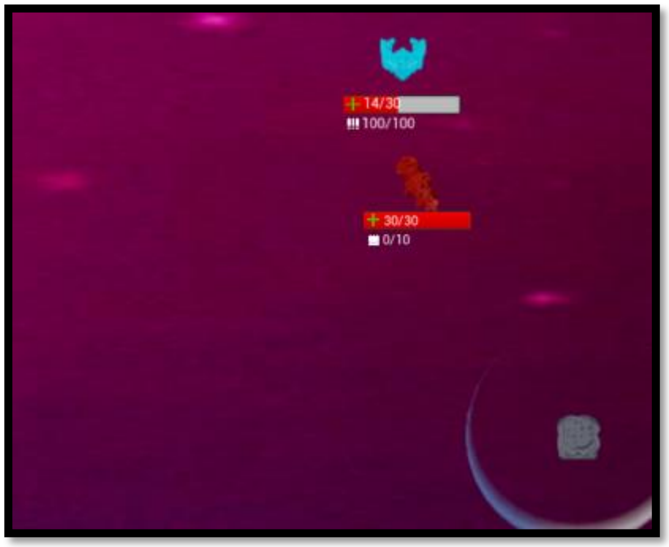
**** Change to current conditions: -HasAmmo*

After all ammo has been used up the current condition of HasAmmo is removed and the planner must be called to find a new solution to the goal. This now includes first going to the ammo refill station to reload.

Action: MoveToRefillStation (Shown Below)

Pre: FarFromRefillStation

Eff: CloseToRefillStation



Once at the station the pawn would perform the action of refilling ammo and can then continue with the end goal of shooting the player.

CASE 2 – HEALTH REFILL

In this scenario the goal state of the enemy fighter ship is to shoot the player, however in the process of doing that it is health drops to a critical level and its main goal then becomes survival by replenishing its health.

Current Conditions: HasAmmo, FarFromPlayer, HasHealth

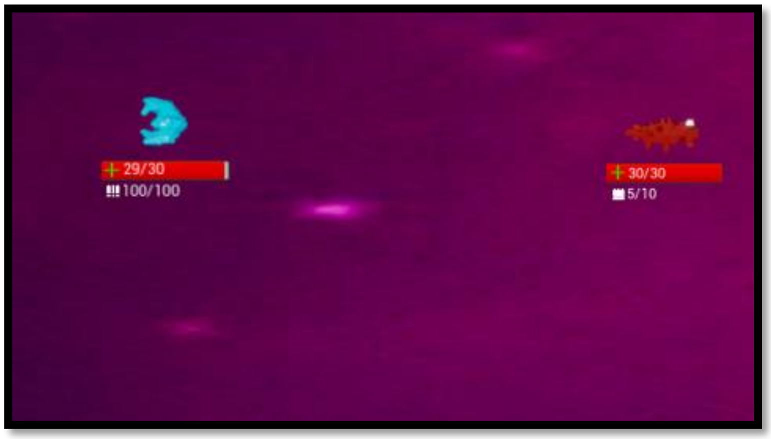
Goal: Shoot Player

Successful action path: MoveToPlayer, ShootPlayer

Action: MoveToPlayer (Shown Below)

Pre: FarFromPlayer

Eff: CloseToPlayer



Action: ShootPlayer (Not Shown)

Pre: HasAmmo, CloseToPlayer, HasLineOfSight

Eff: ShootPlayer

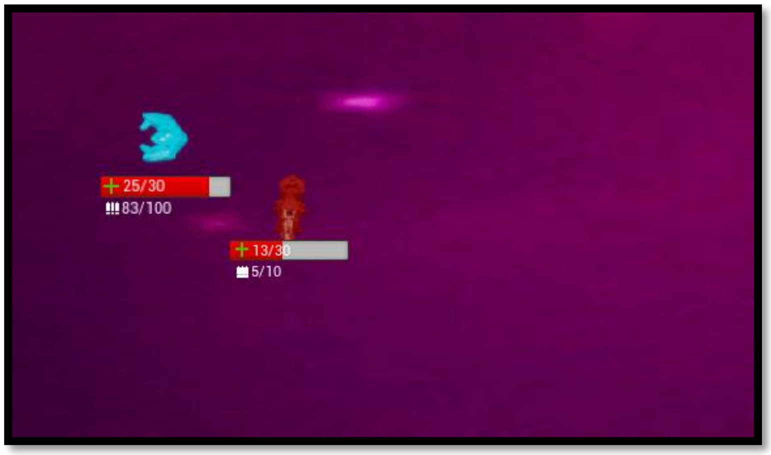
**** Change to current conditions: -HasHealth*

The planner will find refilling health is the ideal goal based on action cost analysis

Action: MoveToHealthStation (Shown Below)

Pre: FarFromHealthStation

Eff: CloseToHealthStation



Action: RefillHealth (Not Below)

Pre: CloseToHealthStation, NeedsHealth

Eff: HasHealth

After refilling the health, the enemy fighter can resume is main goal of shooting the player

CASE 3 – PLAYER CLOSE TO BASE DESTROYER

In this scenario a base destroyer enemy AI has the main goal of shooting the central base, however the player pawn enters its detection range and so it dynamically lowers the cost of its action to shoot the player, making it the new goal.

Current Conditions: HasAmmo, FarFromPlayer, HasHealth

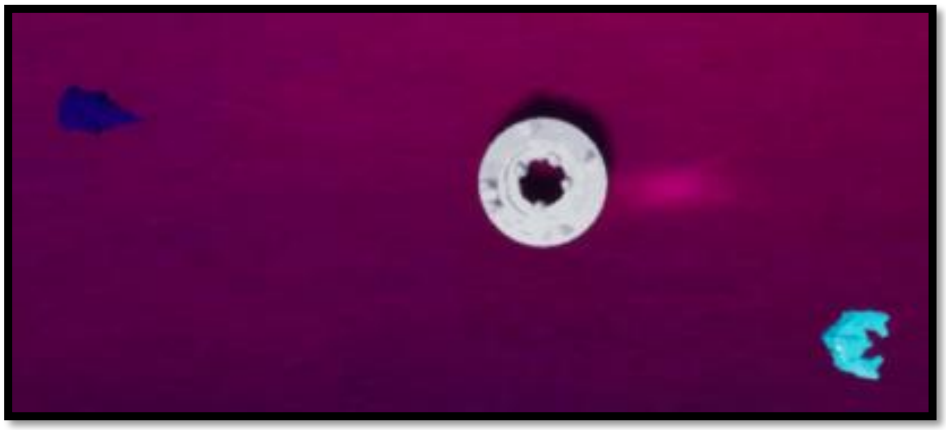
Goal: Shoot Base

Successful action path: MoveToBase, ShootBase

Action: MoveToBase (Shown Below)

Pre: FarFromBase

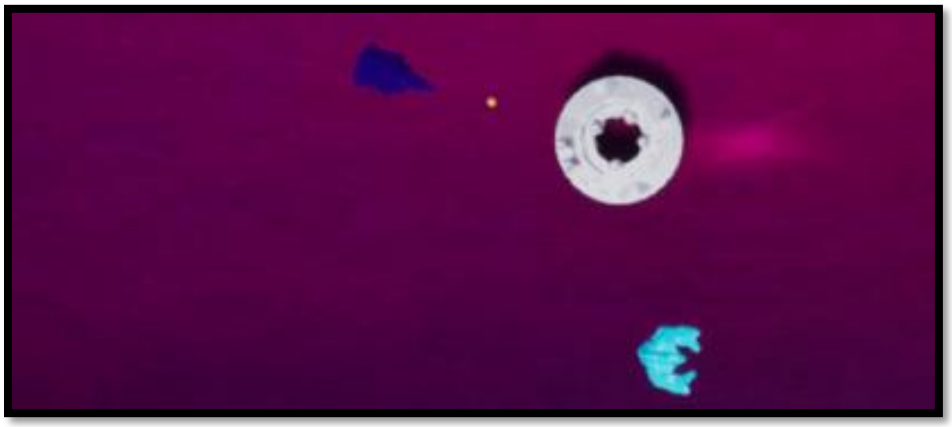
Eff: CloseToBase



Action: ShootBase (Shown Below)

Pre: CloseToBase, HasAmmo

Eff: ShootBase



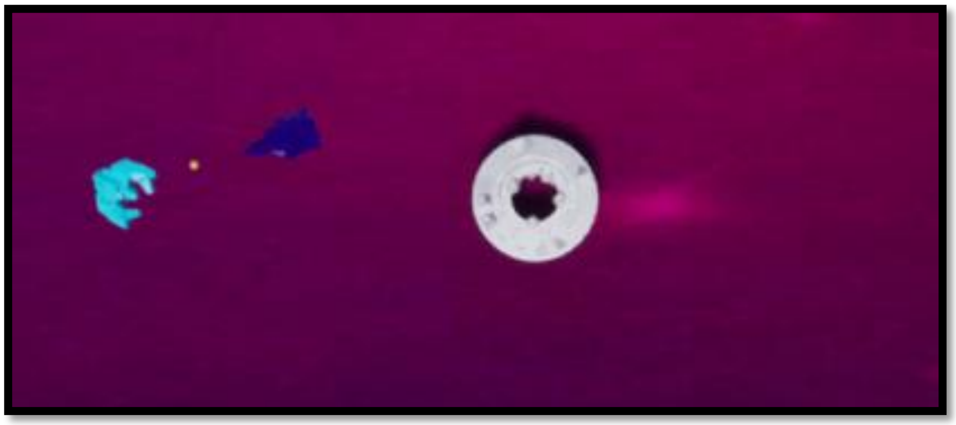
**** Change to current conditions: -FarFromPlayer +CloseToPlayer*

The planner will find shooting the player is the ideal goal based on action cost analysis

Action: ShootPlayer (Shown Below)

Pre: CloseToPlayer, HasAmmo

Eff: ShootPlayer



DISCUSSION

LIMITATIONS

There are limitations to using a GOAP system including increased difficulty in the initial setup and design, processing overhead in planning, difficulty in debugging unexpected AI behaviors, and limited reactive behavior, just to name a few. We will cover how some of these limitations were seen in our development.

DESIGN DIFFICULTY

Designing our GOAP system took most of our development time, with many revised versions and weeks of debugging before we finally had a system that could work for our use case. One of the biggest challenges in the design was figuring out how to ensure the AI had successfully executed each individually action before moving onto the next action. This was accomplished only calling the GOAP planner when there was a change in the current conditions of the AI pawn. So, for example, if an AI had an intermediate action of moving to the player (desired effect being: FarFromPlayer -> CloseToPlayer), then once the move action is called, the planner will not be called again until after the condition CloseToPlayer is in effect.

Some of the more basic AI decisions could have quickly been implemented using a behaviour tree, however a GOAP system would allow us to implement more advanced, dynamic, and longer-term planning strategies. The BaseDestroyer AI has the lowest cost action of attacking the base, however the cost of attacking the player is dynamically adjusted based on how close the player pawn is. If the player pawn comes close enough it will affect the planner strategy of the BaseDestroyer, causing it to attack the player instead. In future development we want to create more advanced AI strategies such as enemy swarming behaviours and group attack tactics.

PROCESSING OVERHEAD

Each time an AI pawn calls the planner, it must do multiple recursive searches to find all possible action paths that will lead from the current conditions to any goal conditions. While doing this it also has to keep track of the total action costs for each successful action path it finds, selecting the lowest cost action in the end. Such recursive algorithms are computationally expensive and processing requirements grows exponentially with the depth and breadth of the search trees. Our AI's need to take at most four actions to achieve any goal. For example, the goal of ShootPlayer when the AI is out of ammo, may require the actions MoveToRefillStation, Refill, MoveToPlayer, then ShootPlayer. Each enemy currently has between 6-8 actions it can take, representing the breadth of search at each recursive call. Adding more actions, conditions, and steps to the AI, will greatly increase the computational cost of the GOAP system.

UNEXPECTED AI BEHAVIORS

Debugging has been an ongoing battle, and we are still experiencing unexpected AI behaviors. It is difficult to determine what is being cause by the GOAP system and what are general unreal programming errors. One example is sometimes when an enemy spawn there is a delay before they start performing action. Another example is an AI might spawn and begin performing the current action of another AI, such as reloading, even when it's current conditions wouldn't dictate it should take that action.

LIMITED REACTIVE BEHAVIOR

To save on computational resources, the planner must be called as infrequently as possible. This makes it more difficult for the AI to quickly adapt to sudden changes in the environment. This translates into sluggish or jarring looking behaviour. A twin stick shooter game like ours benefits by having responsive AI, and a hybrid system would be optimal, having a behaviour tree responsible for quick action responses, while the GOAP planner decides on the longer-term actions.

BREAKING THE AI

The AI can be easily interrupted by lower cost actions, one way the player can break the AI is by forcing the AI to perform a cheaper action to interrupt the action the AI was currently executing.

A simple example present in the gameplay is when the AI runs out of ammunition it will ignore everything else and move to the nearest place to refill ammo. Whether that be a powerup or the refill station does not change anything. The movement is predictable which allows the player to shoot down the AI with relative ease. Once the AI hits a threshold of health where the condition will trigger for it to require health, it will switch its action from needing to refill ammunition to needing to refill health. This will interrupt the AI leading to situations where the AI will not even reach the ammunition refill station and switch its movement towards the health refill station, increasing the time it is defenceless.

GENERAL DISCUSSION

Our understanding of how to build and design a GOAP system, as well as when and why to use it, have grown significantly over the project. Developing this AI model for our twin stick shooter demonstrated key points of how to implement a GOAP system. It may not be the ideal AI method for our use case if the goal is developing a production ready end game but was perfect for discovering and overcoming the challenges and limitations of GOAP.

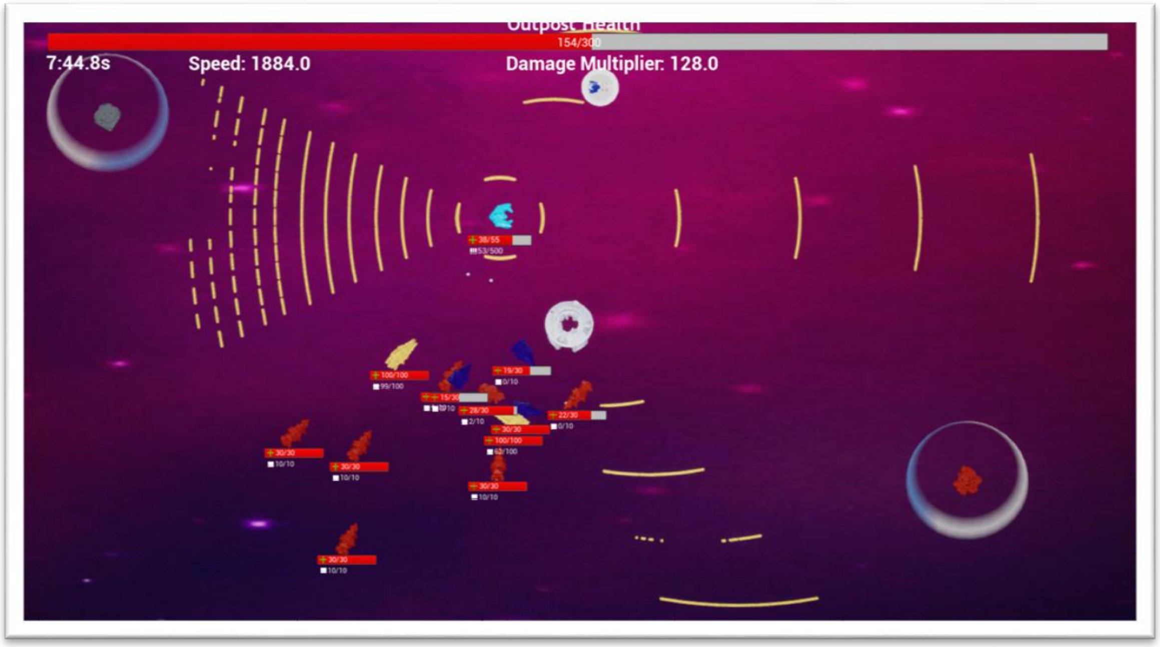
LINK TO PROJECT

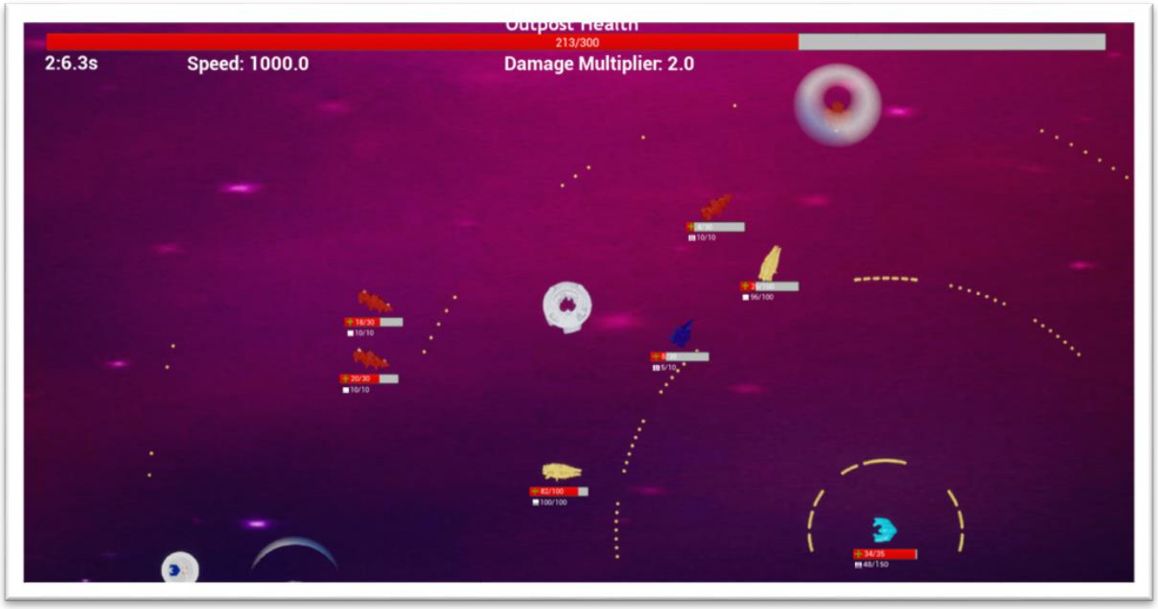
https://github.com/staytheknight/CPSC599.11

REFERENCES TO EXTERNAL ASSETS

3D Ship models: https://www.cgtrader.com/designers/saunick

APPENDIX – GAMEPLAY PHOTOS


Outpost Health
154/300
7:44.8s
Speed: 1884.0
Damage Multiplier: 128.0


Outpost Health
213/300
2:6.3s
Speed: 1000.0
Damage Multiplier: 2.0