IGB383 - AI for Games

Swarm Intelligence and Predator/ Prey Simulation

Assessment 2 - Written component

Scott Barley
N11558059

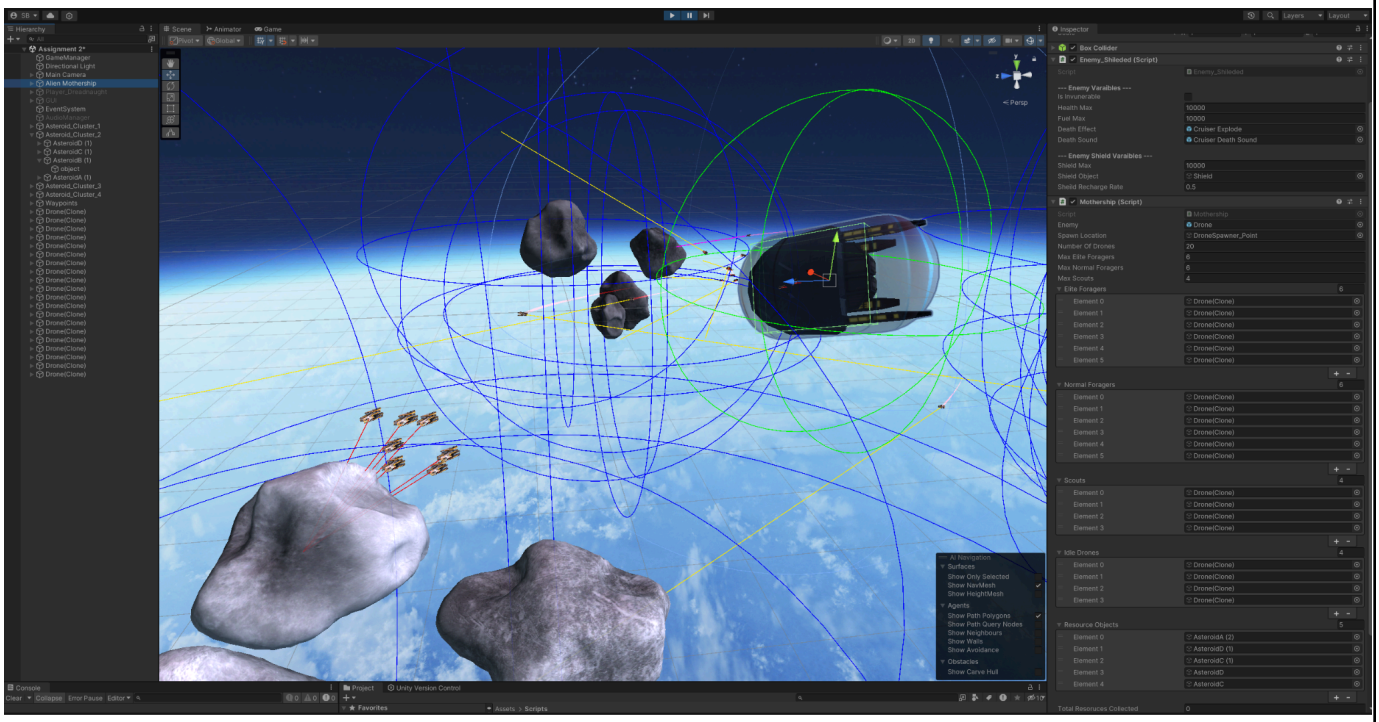
Table of Content

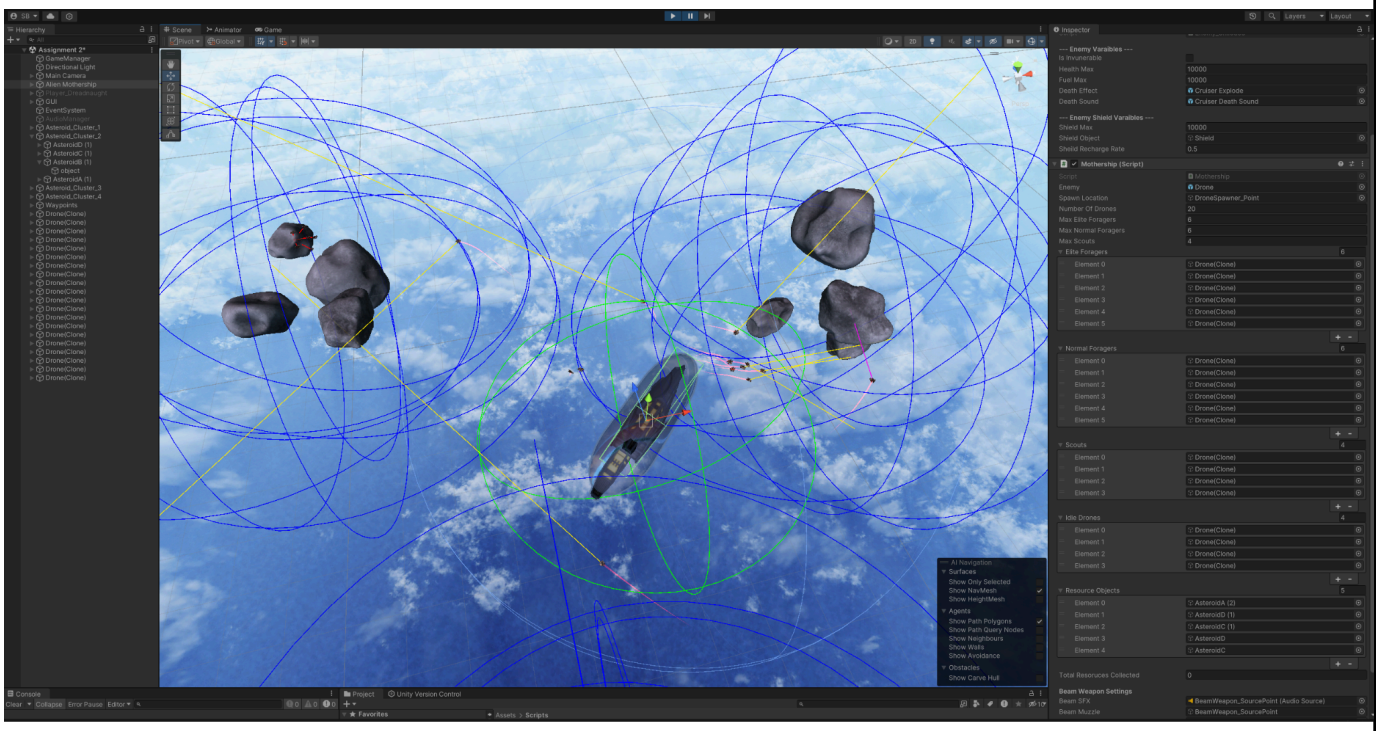
B) Statement of Completeness

All aspects of the assignment were completed in relation to the assignments CRA requirements and some additional extra credit extensions were additionally included as outlined below, this includes;
Implementation of

- An implementation of Boids using physics for swarming and behaviours
    - Implements Dynamic Dissociation mechanics using a Decay & Top-Up Model
- An implementation of Bee's Algorithm in which drones are assigned to 3 Primary job (Elite Forager, Normal Forager & Scout)
    - Utilises Utility Functions (Drone Fitness & Asteroid Desirability)
    - Utilises Dynamic Asteroid Assignment based on the number of 'known' asteroids
    - Utilises Neighborhood Shrinking (Refined localised search driven by elite foragers)
- An Implementation of Fight & Flee Mechanic, 4 States (Intercepting, Engaged, Fleeing, & Repair)
    - Utilises a Utility Function 'Risk'; based on current health and number of local allies
    - Utilises Dynamic Interception, finding a interception vector the reduces in distance inversely to drone distance to target
    - Utilises Optimal Escape Vector based on Drone & Threat Position and Momentum Vectors

Figure B.1: Visualisation of Drone Bee's Algorithm Mechanics in Action





Visual Key

- Blue rings - Local Search Neighborhood
- Green Ring - Mothership, Idle Drone Healing & Refuel Zone

- Red lines - Drones Collecting resources
- Yellow Lines - Drones Conducting Search
- Magenta Lines - Drones moving to resource collection

Boids Implementation

Primary Implementation:

Implementation Location: 'Drone.cs, Line 405, 'BoidBehaviour_SinglePass()'
While we were provided with an implementation of voiding in the workshops, the provided implementation had several issues including not dealing with the local group or cohesion point well, which manifested in the form of issues such as drift towards the world origin which increased inversely in relation to the number of boids. As such I decided to implement my own version of boids which worked on the basis of the local group, to better relate to Reynolds (1987) definition of cohesion as related to the local group.

In relation to Boid 'Alignment', I opted to leave the provided implementation as although not accurate, it proved sufficient and was primarily overridden by the MoveTowardsTarget function the majority of the time, so generally unimportant in terms of accuracy.

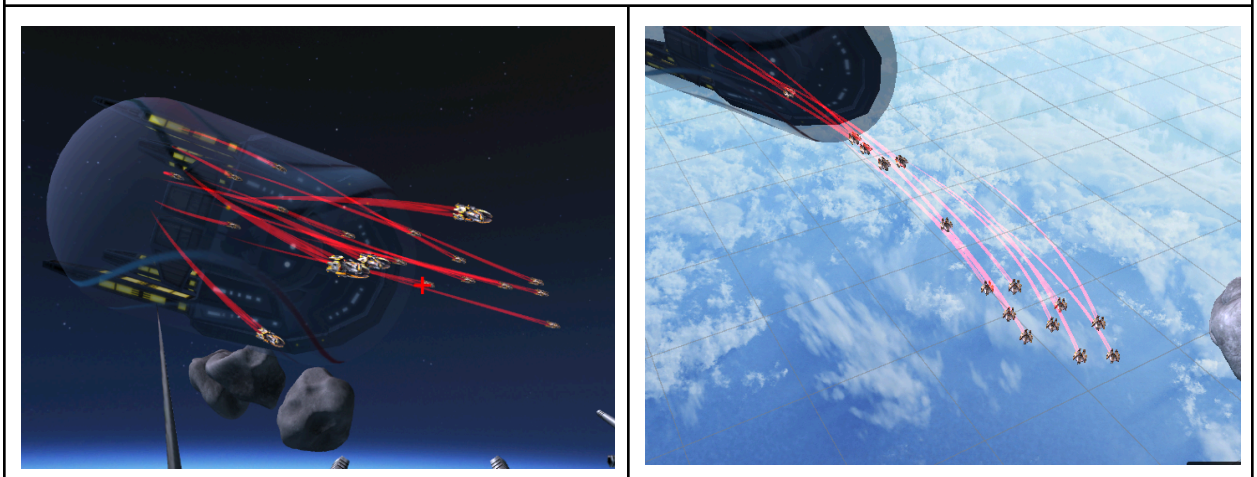
Efficiency; although by switching over from the original method which effectively iterated via the Update loop call to one used implementation that loops through all boids each frame, the efficiency is substantially decreased, for my use case this had little effect on overall performance so I decided to stick with it in its current state. If I wanted to make it more efficient I would have likely batched the boids and used a tick timer system, or used DOTS/ETS system in the future instead of switching back to the Update function as the iterator.

Figure B.2: Issues with original boiding behaviour

```
//Reset cohesion position
cohesionPos.Set(0f, 0f, 0f);
```

```
// Calculate the current cohesionPos
cohesionPos = cohesionPos + pos * (1f / (float)gameManager.inSceneEnemy_array.Length);
```

Figure B.3: Basic Boiding Behavior



Extension Implementation (DissociationCoefficient):

Implementation Location: ‘Drone.cs, Line 324, BoidDissociationCoefficientAOE_RadialLERP()’

- ***NOTE: Uses Blumberg’s Decay & Top-Up Model***

In addition to the boiding behaviour, I utilised the boid variables to drive dynamic reactions to ‘threat’ when the boids are attacked, or destroyed by the players. When a drone is attacked or destroyed, the ‘BoidDissociationCoefficientAOE_RadialLERP()’ function is called in the TakeDamge Override function, which gets all local boids and increases the dissociation coefficient value, literally in relation to the distance to the source point. This significantly increases the Separation Distance and reduces the Coheason force so as to effect the boids spreading out from each other.

Figure B.4: Implementation Calls of Boid Dissociation

```
#region Inheritance Overrides
4 references
public override void takeDamage(float dmg)
{
    // If taking damage warn surrounding boids to dissociate from each other
    BoidDissociationCoefficientAOE_RadialLERP(boidDissociationCoefficient_DistressCallRadius, dmg * boidDissociationCoefficient_DamageToDissocoitionMultiplier);
    base.takeDamage(dmg);
}

3 references
protected override void HandleOnDeath()
{
    // If Die warn surrounding boids to dissociate from each other (big Death cry)
    BoidDissociationCoefficientAOE_RadialLERP(boidDissociationCoefficient_DistressCallRadius * 2, 200);

    //Disable Trail Renderer befor the object is destoryed to stop error issue
    GetComponent<TrailRenderer>().enabled = false;
    base.HandleOnDeath();
}
#endregion
```

C) Bee's Algorithm Implementation

Beehive Simulation Description

We were asked to implement Bee's algorithm which is a nature-inspired optimization algorithm based on how honey bees optimise nectar gathering in nature. Its part of the family of swarm intelligence algorithms, along with boids, which utilise decentralised control and decision making to drive emergent behaviour. Our implementations are set in real-time inside a game engine in the conceptual form of mining drones harvesting resources from asteroids in space. It consists of agents which conduct global random searches to locate new asteroids 'scout drones', agents which conduct local random searches 'elite foragers' and resource collection agents the 'foragers'. By utilising multiple agents with heuristic driven allocations the algorithm balances exploration and exploitation, in theory ensuring a global optimum solution. The heuristics which determine 'resource desirability', drone 'fitness' are calculated via utility functions which assign numerical value to the criteria driving the decision making process, by which allowing a suitable outcome behaviour to be decided on.

While the are a few different version of Bee based algorithms including; Honey bee (Tovey et al. 2004) BeeAdHoc (Wedde et al. 2005), BeeHive (Wedde et al. 2004), these algorithms are more focused on graph navigation optimization then our use case, and while interested and providing many additional ideas, for the sake of the complexity of attempting to utilise the concept in real time 3d space, the implementation opted for is a relatively simpler interpretation, which still attempts to capture the elements of both local & global search optimisation for the optimisation of resource collection.

CRA Requirements:

A Beehive simulation: Implement a hive of bee Boids and Bee's algorithm for harvesting resources. Bees harvest resource prior to engaging with the Player.

Implementation:

Implementation location: split between: Mothership.CS, Drones.CS, Drones_FSMBehaviours.CS, Asteroid.CS & GameManager.CS,

The algorithm is primarily driven within the mothership script where a number of heuristic values are called and calculated to drive the assignment of drones to locate & gather resources from the surrounding area. This is achieved in a number of key steps;

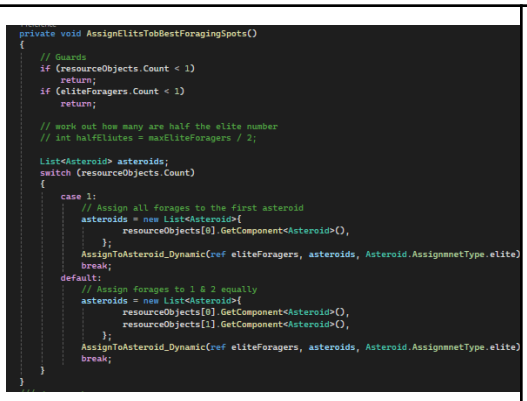
Mothership.CS Related Component of the Bees Algorithm Implementation:

- **Step 1: Update/Calculate Heuristic values**
  - PrioritiseDronesByFitness()
    - All available idle drones are sorted & ordered by their 'fitness'; a weighted linear fitness heuristic that returns a normalised value based on the current health and fuel of each drone. Located; line 146, Drone.Cs
    - ```
      public float fn_Fitness_Heuristic(float healthWeightPct = 0.6f, float fuelWeightPct = 0.4f)
      {
          currentFitness = healthWeightPct * fn_GetCurrentHealth_Pct() + fuelWeightPct * fn_GetCurrentFuel_Pct();
          return currentFitness;
      }
      ```
  - PrioritiseResourcesForCollection_Asteroids()
    - All 'known' asteroids are assessed, sorted and ordered based on their Desirability 'CalculateAstaroidDesirability()'. This assesses all known asteroids to assign normalised scores for distance and resource amounts, which are then fed into a weighted linear fitness heuristic calculation which assigns a value based on a 80% weighting to distance and a 20% weighting to current resource amount Located; line 238, Mothership.Cs
    - ```
      238     float Utility_ResourceDesirability(float distance, float dis_weighting, float resources, float res_weighting)
      239     {
      240         return distance * dis_weighting + resources * res_weighting;
      241     }
      ```
- **Step 2: Assign Drones to Roles**
  - Assignment of Elite Foragers 'AssignDronesToBecome_EliteForagers()', 'AssignDronesToBecome_NormalForagers()' & 'AssignDronesToBecome_Scouts()'
    - Using the target number of each of these forage types and the current number of drones assigned to each associated pool (stored as lists), these pools are topped with the idle drones based on fitness order, with the fittest going to elites and least fittest becoming scouts.Located; Mothership.CS
    - ```
      public List<GameObject> eliteForagers = new List<GameObject>();     // Elite Foragers Drones
      public List<GameObject> normalForagers = new List<GameObject>();    // Foragers Drones
      public List<GameObject> scouts = new List<GameObject>();            // Scouting Drones
      public List<GameObject> idleDrones = new List<GameObject>();        // Idle Drones
      ```
- **Step 3: Assignment of Resource Spots**
  - With the resources prioritisation complete and the drone task assignment complete, the assignment of resources takes place; 'AssignElitsTobBestForagingSpots()' & 'AssignNormalForagersToForagingSpots()'
  - These functions work by iterating through the Elite & Normal Forager drones to find drones that have not yet been assigned a resource(asteroid) to target, then cyclically assigning them to alternating asteroids, down the priority list. This works in conjunction with a 'total assignment number' stored on each Asteroid.CS to ensure the drones are evenly distributed between the current priority targets. This stops 'over assignment' of drones. For example; Elite drones the single most desirable asteroid, by limiting the number to half of the total number that can be assigned to that asteroid, instead ensuring an equal distribution between the 2
      ```

most desirable asteroids. The relevant values are calculated dynamically as demonstrated in the scripts below.

-

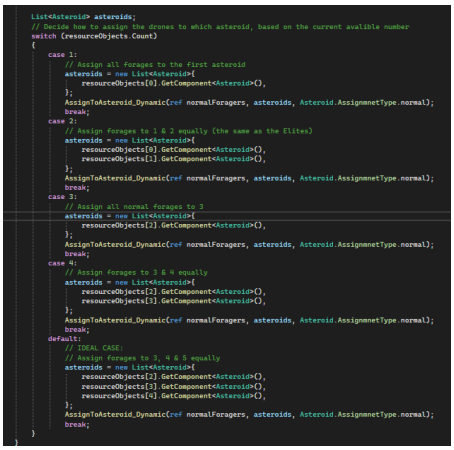
Figure C.1: Dynamic Assignment of drones to resources (asteroids)



```
private void AssignElitsTobBestForagingSpots()
{
    // Guards
    if (resourceObjects.Count < 1)
        return;
    if (eliteForagers.Count < 1)
        return;

    // work out how many are half the elite number
    // int halfElites = maxEliteForagers / 2;

    List<Asteroid> asteroids;
    switch (resourceObjects.Count)
    {
        case 1:
            // Assign all forages to the first asteroid
            asteroids = new List<Asteroid>{
                    resourceObjects[0].GetComponent<Asteroid>(),
                };
            AssignToAsteroid_Dynamic(ref eliteForagers, asteroids, Asteroid.AssignmnetType.elite);
            break;
        default:
            // Assign forages to 1 & 2 equally
            asteroids = new List<Asteroid>{
                    resourceObjects[0].GetComponent<Asteroid>(),
                    resourceObjects[1].GetComponent<Asteroid>(),
                };
            AssignToAsteroid_Dynamic(ref eliteForagers, asteroids, Asteroid.AssignmnetType.elite);
            break;
    }
}
```



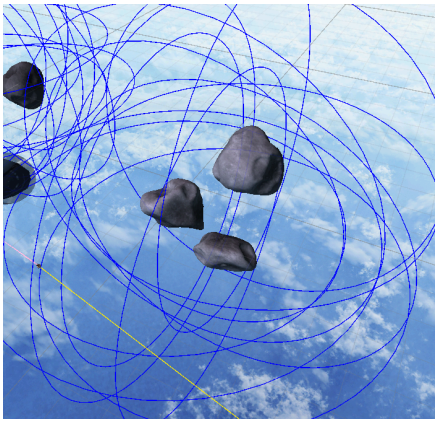
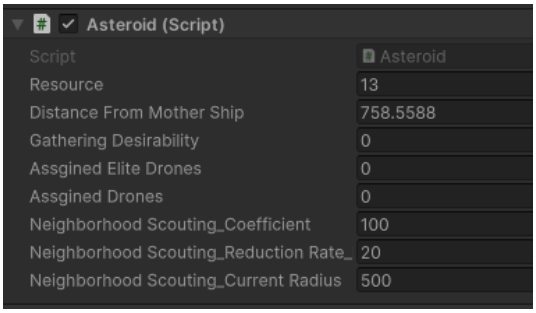
```
List<Asteroid> asteroids;
// Decide how to assign the drones to which asteroid, based on the current avalible number
switch (resourceObjects.Count)
{
    case 1:
        // Assign all forages to the first asteroid
        asteroids = new List<Asteroid>{
            resourceObjects[0].GetComponent<Asteroid>(),
        };
        AssignToAsteroid_Dynamic(ref normalForagers, asteroids, Asteroid.AssignmnetType.normal);
        break;
    case 2:
        // Assign forages to 1 & 2 equally (the same as the Elites)
        asteroids = new List<Asteroid>{
            resourceObjects[0].GetComponent<Asteroid>(),
            resourceObjects[1].GetComponent<Asteroid>(),
        };
        AssignToAsteroid_Dynamic(ref normalForagers, asteroids, Asteroid.AssignmnetType.normal);
        break;
    case 3:
        // Assign all normal forages to 3
        asteroids = new List<Asteroid>{
            resourceObjects[2].GetComponent<Asteroid>(),
        };
        AssignToAsteroid_Dynamic(ref normalForagers, asteroids, Asteroid.AssignmnetType.normal);
        break;
    case 4:
        // Assign forages to 3 & 4 equally
        asteroids = new List<Asteroid>{
            resourceObjects[2].GetComponent<Asteroid>(),
            resourceObjects[3].GetComponent<Asteroid>(),
        };
        AssignToAsteroid_Dynamic(ref normalForagers, asteroids, Asteroid.AssignmnetType.normal);
        break;
    default:
        // IDEAL CASE:
        // Assign forages to 3, 4 & 5 equally
        asteroids = new List<Asteroid>{
            resourceObjects[2].GetComponent<Asteroid>(),
            resourceObjects[3].GetComponent<Asteroid>(),
            resourceObjects[4].GetComponent<Asteroid>(),
        };
        AssignToAsteroid_Dynamic(ref normalForagers, asteroids, Asteroid.AssignmnetType.normal);
        break;
}
```

- **Step 4: Refuel And Repair**
  - In this step ‘RefuelAndRepairDrones()’; drones are given the conceptual opportunity to ‘rest’ i.e. repair and refuel, at the expense of the collected resources. This results in the drones cycling through the different jobs as they burn fuel and decrease in fitness and for drones to be replenished to max fitness in longer periods between job assignments.
- **Step 5: Loop End**
  - The loop is effectively completed by the drone FSM behaviour ‘HandleReturnToMotherShip()’ (Located in Drone_FSMBehaviours.CS) which is called on their return to the mother ship on completion of their task. This deassigns the drone from their allocated job, removes them from the related job pool (Lists) they are in and returns them to the ‘Idle’ Pool, to wait for a fresh assignment.

The above cycle; allows for an assessment and dynamic response to a continuously changing environment, so that resources can be located, assessed and assigned to drones for collection to achieve the optimal outcome. The other half of the Bee’s algorithm is delivered by the Drones and their related FSM behaviours associated with their assigned jobs/roles.

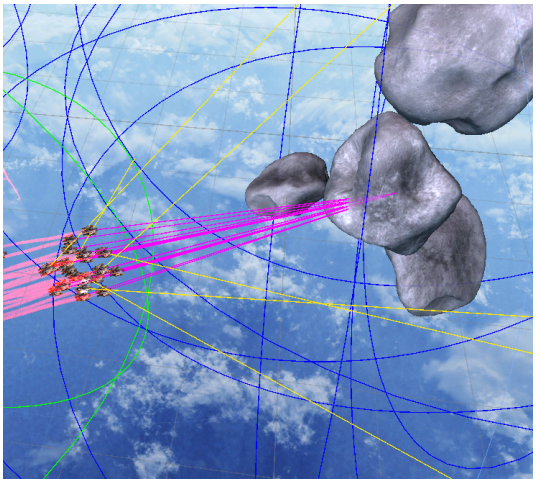
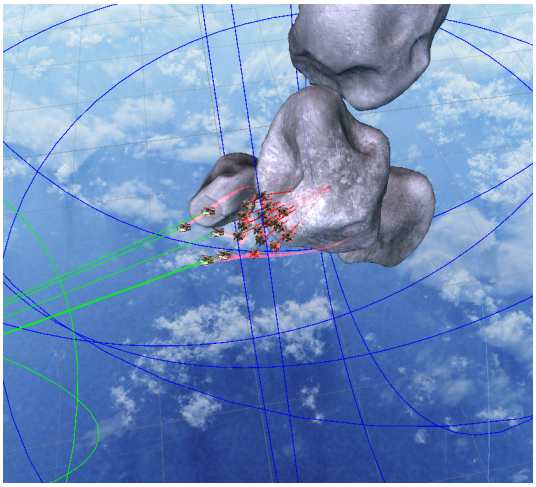
Drones.CS & Drone_FSMBehaviours.CS; related components of the Bees Algorithm Implementation:

- **Job / Behaviour 1: Elite Foraging**
  - GeneralBehaviour_EliteForaging(); found in Drone_FSMBehaviours.CS
  - This states behaviour function by conducting an ‘elite scout’ behaviour, which was conducted in relation to neighbourhood variables held by the Asteroid.CS that they had been assigned. With each time the location around the target asteroid was scouted without the location of a newly identified resource the ‘neighbourhood scouting radius/area’ is reduced, thereby increasing the efficiency of future visits while balancing the discovery of new / better resources.
  -

Figure C.2: Localised Search / Local Neighbourhood Shrinking	
Visualisation of Local Neighbourhood	Neighbourhood Variables on Asteroid
	

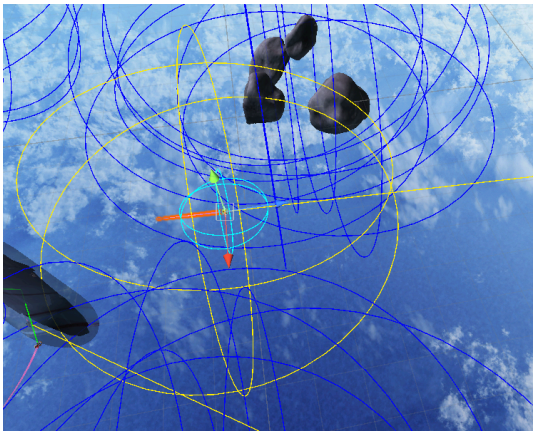
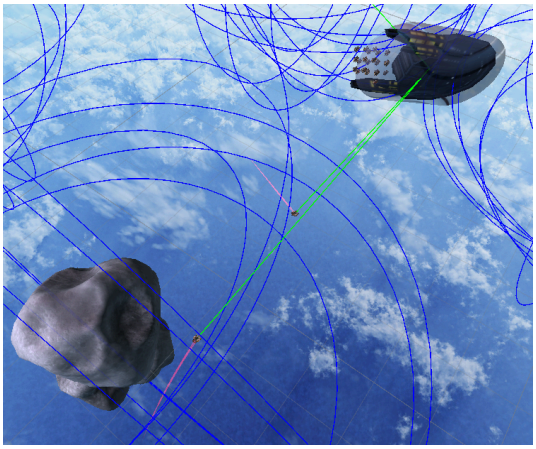
- Once a scouting attempt had been made; if a new resource had been identified the drone would return to the mothership to report it, else if not, the drone would forage from its target asteroid.

- **Job / Behaviour 2: Normal Foraging**
  - GeneralBehaviour_Foraging(); found in Drone_FSMBehaviours.CS
  - The Foraging behaviour, directs the drone towards their assigned asteroid then once in range they begin to extract resources, up to their carrying capacity prior to returning to the mothership to deposit them.
  -

Figure C.3: Foraging	
Showing the first asteroid located. Yellow lines are scouting lines from the Elites while the purple lines are the ‘moving to resource’ vectors of the basic forages	Showing the discovery of the new local resource by the elite drone; green return to mothership vectors & red resource collection vectors of the normal forages
	

-

- **Job / Behaviour 2: Scouting**
  - GeneralBehavior_Scouting(); found in Drone_FSMBehaviours.CS
  - Scouting is conducted by scouting assigned drones, and is achieved through movement to randomised position vectors around the mothership, with periodic checks for resources within the scout detection radius. If a resource is found the scout returns back to the mothership to 'report' the discovery.
  - Additionally the 'DetectNewResources()' function was updated from the originally provided version to find the best 'unknown' asteroid.

Figure C.4: Scouting Visualisations	
Yellow Wire Sphere - Scouting Detection range Yellow Line - Scouting Path	Green Path - Return Vector to Mother Ship once a new resource is located
	

  -

- **Supporting Functionality**
  - The is additional a range of support function for assigning and deassign the drone from tracking variables as they move through the different states.
  - HandleReturnToMotherShip();
    - On returning to the mothership deassign drones from role related variables and returns them to the drone Idle Pool

Finally, to support a prolonged simulation, the asteroids are set to respawn periodically although I couldn't think of a good reason why this would occur narratively. This is managed in the AsteroidClusterManager.CS script attached to each asteroid cluster.

D) Predator/Prey simulation:

CRA Requirement

A Predator/Prey simulation: Implement predator/prey behaviour using models for fear, hunger and hunting as presented in the Lectures. All bee's should exhibit these behaviours only when they engage with the Player.

Implementation

The predator/prey behaviours are driven in the Combat Behavior States, called through the FSM GeneraBehavior_Combat() Function. It works in two primary states; Fighting, or Fleeing, this in turn drives 4 resultant behaviours; Interception, Engagement, Fleeing & RepairAndRegroup. As an aside rather than referring to it as 'fear' I've referred to it as 'risk' as it feels more appropriate for robotic drones.

Figure D.1: Primary Functions driving Predator/Prey Behaviours

```
/// <summary>
/// DOES: Runs 'General Behaviors' of FSM
/// </summary>
1 reference
private void Update_RunGeneralBehaviour()
{
    switch (droneBehaviour)
    {
        case DroneBehaviours.EliteForaging:
            GeneralBehaviour_EliteForaging();
            break;
        case DroneBehaviours.Foraging:
            GeneralBehaviour_Foraging();
            break;
        case DroneBehaviours.Scouting:
            GeneralBehavior_Scouting();
            break;
        case DroneBehaviours.Combat:
            GeneralBehavior_Combat();
            break;
        case DroneBehaviours.ReturnToMotherShip:
            GeneralBehaviour_ReturnToMotherShip();
            break;
    }
}
/// <summary>
/// DOES: Runs Combat Sub FSM
/// </summary>
1 reference
private void GeneralBehavior_Combat()
{
    DetermineFightOrFleeCombatBehavior();
    RunCombatBehavior();
}
1 reference
private void DetermineFightOrFleeCombatBehavior()
{
    // DOES: Calculates a risk huristic value based on current health & number of close allies
    current_RiskHuristic = RemainingHealt_pct * 0.7f + getNomalisedAlliesThreatRating(minimumNumberOfCloseAlliesToFeelSafe) * 0.3f;

    // Exclude Resting Drones, to allow for Regrouping
    if (current_RiskHuristic > 0.5 && droneCombatBehaviour != DroneCombatBehaviours.RepairAndRegroup)
    {
        // NOTE: FIGHT
        if(!(droneCombatBehaviour == DroneCombatBehaviours.Intercepting || droneCombatBehaviour == DroneCombatBehaviours.Engaged))
            droneCombatBehaviour = DroneCombatBehaviours.Intercepting;
    }
    else
    {
        //NOTE: FLEE
        if (!(droneCombatBehaviour == DroneCombatBehaviours.Fleeing || droneCombatBehaviour == DroneCombatBehaviours.RepairAndRegroup))
            droneCombatBehaviour = DroneCombatBehaviours.Fleeing;
    }
    Sub Functions
}

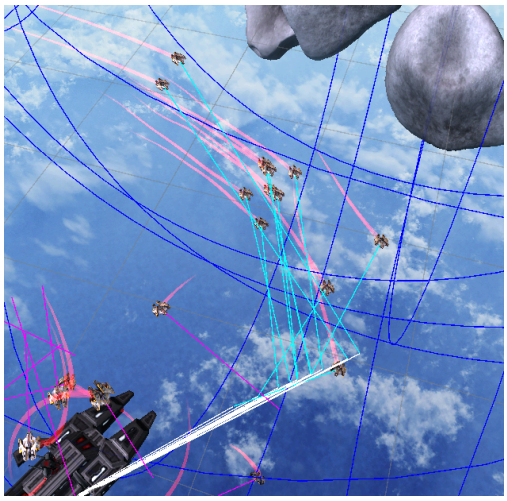
1 reference
private void RunCombatBehavior()
{
    // DOES: Run Appropriate Combat Behavior
    switch (droneCombatBehaviour)
    {
        case DroneCombatBehaviours.Intercepting:
            CombatBehavior_Intercepting();
            break;
        case DroneCombatBehaviours.Engaged:
            CombatBehavior_Engaged();
            break;
        case DroneCombatBehaviours.Fleeing:
            CombatBehavior_Fleeing();
            break;
        case DroneCombatBehaviours.RepairAndRegroup:
            CombatBehavior_RepairAndRegroup();
            break;
    }
}
#endregion
```

Implementation Stages:

- **Stage 1: DetermineFightOrFleeCombatBehavior();**
  - Located in the Drone.CS Script
  - It works by calculating a risk heuristic value by combining different factors like remaining health and the number of allies.
  - The calculated value is then used to determine if it should be in a ‘Fight’ related state, or a ‘Flee’ related state, if this is different to the current state it is in, it is placed into the entry state of either Fight (Intercept) or Flee (Fleeing).

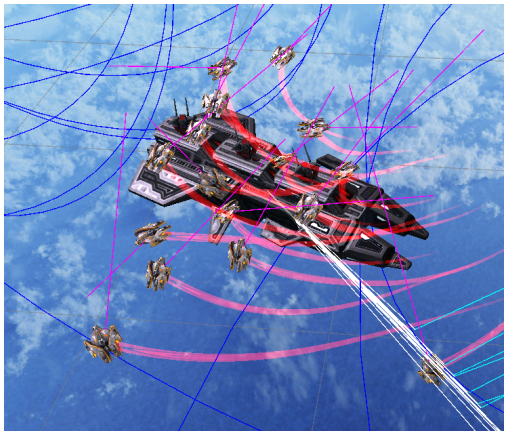
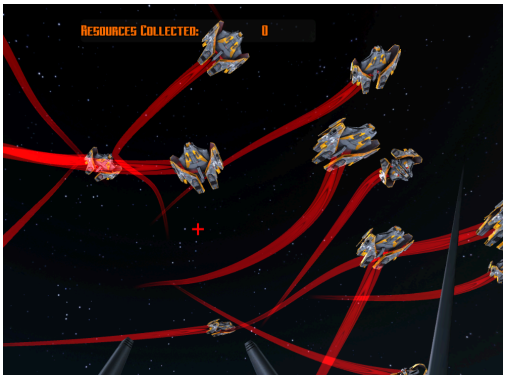
- **Stage 2: Run the Appropriate Combat Behavior**
  - Called by: ‘RunCombatBehavior()’ Located in the Drone.CS Script

- **Stage 3: The Combat Behaviours**
  - **CombatBehavior_Intercepting()**
    - The intercepting behaviour uses a dynamic interception point that decreases in magnitude the closer the player gets to the interception target, i.e the player ship, thus resulting in a pretty curved interception which is visually appealing. I’ve additionally added a Y value offset, so the interception path stays in the players line of sight from their camera position to support this.

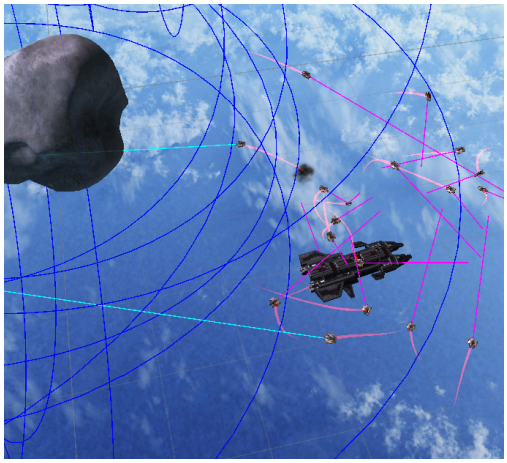
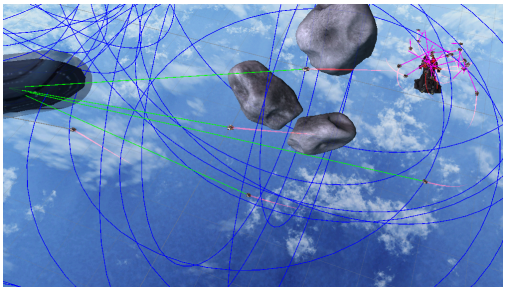
Figure D.2: Visuals of Interception Behaviour	
Blue Lines - Interception Vector White Line - Interception Vector (from player ship)	**Visual Effect:** the group sawm in and curve into the path of the player before entering their Harassment Mode
	

    -

  - **CombatBehavior_Engaged()**
    - **Implementation:** Using a calculated vector based on the threat target movement, similar to the interception vector, but at a fixed distance in front of the PlayerShip so to stay above the moving player ship, the drones individually and periodically choose random positions in an area above and around the top of the player ship to swarm between.
    - **Intended Effect:** by choosing positions to move to above the players ship it is intended to induce the effect the player is being harassed and swarmed.

Figure D.3: Visuals of Engagement / Harassment Behaviour	
Purple lines - Harassment Vectors	Visual Outcome:
	

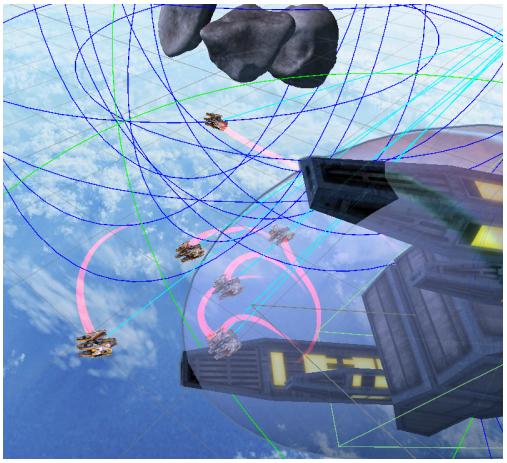
- **CombatBehavior_Fleeing()**
  - The combat fleeting system is two stages, first an optimal escape vector is calculated based on the drone and threattarget position and velocity vectors. The Drone then moves towards this. Once sufficiently far from the player ship the drone is then redirected to the mothership to heal and regroup.

Figure D.4: Visuals of Fleeing Behaviour	
Stage 1: Escape Vector Light Blue - Escape Vector	**Stage 2: Return to Mothership** Light Green - Return To Mothership Vector
	

- **CombatBehavior_RepairAndRegroup()**
  - The repair and regroup behaviour is also two stage, first to move towards a ‘regroup’ position so that all repairing drone are within each others local ally detection range, then

second stage, once a sufficient number of local drones are of sufficient fitness they all re-engage and change to ‘Intercept’ mode at the same time, creating a new micro swarm.

- **IMPORTANT NOTE:** for this behaviour to work the mothership must have resources which are used for repairing and healing the drones.

Figure D.4: Visuals of Repair And Regroup Behavior	
Grey Line - Repair and Regroup Position Vector	Showing the drones collectively switching back to Interception mode
	

E) Conclusion

Overall Summary:
In general I'm pretty happy with what I've had a chance to create in this assignment and unit in general. Although I've previously experimented with a lot of the elements that went into this assignment, it was a great learning opportunity to attack it from a slightly different angle in which I discover new challenges and ways of overcoming them. I mostly enjoy getting to create the cyclical combat experience of shooting the drones then having them fly away, repair, regroup then come back again, I found it a satisfying and visually pleasing cycle.

Building On Previous Experience:
Having previously experimented with building the ant colony simulation in Unity DOTS using the entity-component workflow and creating reactive boids for my IGB100 assignment, where much of the focus in both projects was on efficiency & ensuring hundreds / thousands of boids behaved smoothly using batching, grouping, and tick-time solutions. Initially, this current implementation made me uncomfortable due to its inefficiency, especially since much of the logic ran through the Update loop. However, in this use case, where scalability wasn't a priority and I found myself focusing less on optimisation and more on exploring different options and system components. This shift allowed me to enjoy the process more, as I wasn’t consumed by the need to solve efficiency problems.

New things I learned:
On a similar note the concept of using the Update as an iterator to loop through a group of Units doing calculations was something I’ve not encountered previously

Planning, Design & Implementation of FSM:

One of the core challenges and issues I encountered was related to my initial design choices, in particular implementing an overly simplified Drone FSM system.As I initially designed my implementation around only delivering the basic requirements of the assessment, I utilised a very basic state switching system where the switch statements existed in the states themselves. This choice backfired later in the development process as I opted to create more states, leading to a lot of wasted time reworking and battling the system to be capable of dealing with each new state I added. My take away learning point from this experience is that if I'm implementing a FSM system I should design the system with better extensibility with a more formulated FSM methodology, as previously explored in Assignment 1, from the outset of the project in future.

References

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. Computer Graphics, 21(4), 25–34. https://doi.org/10.1145/37402.37406
https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/modeling-natural-systems/boids.html

Tovey CA. (2004). The honey bee algorithm, a biologically inspired approach to internet server optimization. Engineering Enterprise, Spring, pp.13-15

Wedde HF., Farooq M. and Zhang Y. (2004) BeeHive: An Efficient Fault-Tolerant Routing Algorithm Inspired by Honey Bee Behavior. ANTS, LNCS 3172, pp.83–94.

Wedde HF., Farooq M., Pannenbaecker T., Vogel B., Mueller C., Meth J., and Jeruschkat R. (2005). BeeAdHoc: An Energy Efficient Routing Algorithm for Mobile AdHoc Networks Inspired by Bee Behaviour. GECCO’05, June 25–29, USA.