A Simple Society - Game Theory

Jamie Hyland
Alex Thornberry
Supervisor: Allistair Sutherland

Abstract

People in groups behave in different ways, some are cooperative and trusting others are aggressive and deceptive. The task is to try to create a virtual society in which individuals behave in different ways. Each has a different strategy, which may evolve as time goes on. The individuals may learn and improve their behaviour to obtain greater rewards. What strategy works out best in the long-run?
The aim of this project is to simulate a simple society using genetic algorithms and key concepts found in Game Theory. The system simulates two types of agents, passive and aggressive, and attempts to find the optimal scenario where the two can co-exist without over population or extinction occurring.

Motivation

Our motivation for the project was to learn how people in groups behave in different ways. Some are cooperative and trusting, others are aggressive and deceptive. We wished to research these different behaviours, with an emphasis on evolution. With not a lot of knowledge of the topic we were very interested to see which strategies would work out best in the long-run.

Both of us have an interest in gaming, specifically, computer gaming. Developing a simulation in unity allowed us to take many aspects of building a game and add them to the simulation. From our experience in our third year module, Human Computer Interaction, we increased our interest in the design and implementation side of games.

We were interested in this project for the inherent split in the type of work. It was clear from the beginning that we would be coding up this simulation in Unity, making use of different coding methods and 3d models we would create in blender. However we also knew that the analysis towards the end of the project would be almost if not as important.

Research

Genetic Algorithm

A genetic algorithm is a heuristic search algorithm that aims to find a solution to a problem where the solution is unknown or the search space is so large that a brute force approach to solving the problem is infeasible. In essence, they are an optimisation technique, used to solve non linear or non differential optimisation problems. Genetic Algorithms have been used in a variety of fields such as economics and engineering. Genetic algorithms are based on the theory of natural selection and evolutionary biology. To model a society evolving over time the system will need to incorporate 3 key features:

Heredity - also called inheritance, is the passing on of traits from parents to their offspring. There must be a process that allows new agents to receive some properties of the parent agents that made them. For the purpose of this project, it will exclusively occur through sexual reproduction. i.e There will be values passed from n generation to n+1 generation.

Variation - Agents need to have a variety of attributes to ensure there is some sort of change in the population over time. If all agents have the same attributes the ecosystem will become stagnant. This variation can be introduced by the landscape itself through the introductions of new types of agents. It can be altered via a mutation that takes place when reproduction occurs between two agents.

Selection - There needs to be a process for which some agents in the population have the opportunity to mate and pass down some of their genetic traits to the next generation. A part of this process is defining what it means to be a “fit” agent, we generally want a system that allows our agents to be the fittest they can be, in real world examples this might be how fast an animal runs or how strong it is. In our case we’re definings an agent’s fitness as its ability to survive as long as it can into old age without dying due to thirst or hunger or being eaten by another agent.

The main loop in a genetic algorithm goes as follows:

- **Initialize the population** - Generate N number of randomly generated agents with varying values
- **Selection** - Evaluate the fitness of each member of the population, and assign each member a fitness score, this will act as a probabilistic chance that the member gets added to the mating pool
- **Reproduce -** this is repeated until all members of the mating pool have mated:
    - Pair two members up for reproduction
    - Crossover - This is where aspects of each parent are passed down to the new member
    - Mutation - a percentage chance that when passing down a genetic trait to a new member that the value is mutated in some way

- Replace old population with new population

The outline above is for a more formal approach to performing a genetic algorithm, and as such this approach had to be tweaked for implementation in the system. With regards to the selection process of the genetic algorithm the system will employ a hands off approach when selecting members for the mating pool. The system regards an agent's survivability as its level of fitness, as agents who cannot survive will die out, removing their genes from the gene pool.

Game Theory

This project was introduced to us as simulating simple societies. We would simulate these societies so that we could analyse the behaviour of the agents within these societies. Specifically, we were interested in how evolution within the society would affect the behaviour of the agents. Upon our first reading of this, game theory instantly came to mind. It was decided that game theory would have a heavy influence in the decisions we made for the simulation.

Upon researching game theory we learned a few general points on game theory and what it actually entails.
Game Theory is the branch of mathematics concerned with the analysis of strategies for dealing with competitive situations where the outcome of a participant's choice of action depends critically on the actions of other participants.
Game theory studies situations always involving more than one participant and in which not everything depends on one person.
Game theory must involve a payoff for players arriving at a particular outcome. In our simulation, obtaining food is rewarded with less hunger and mating with a partner pays out in the continuation of the player's bloodline through parenthood.

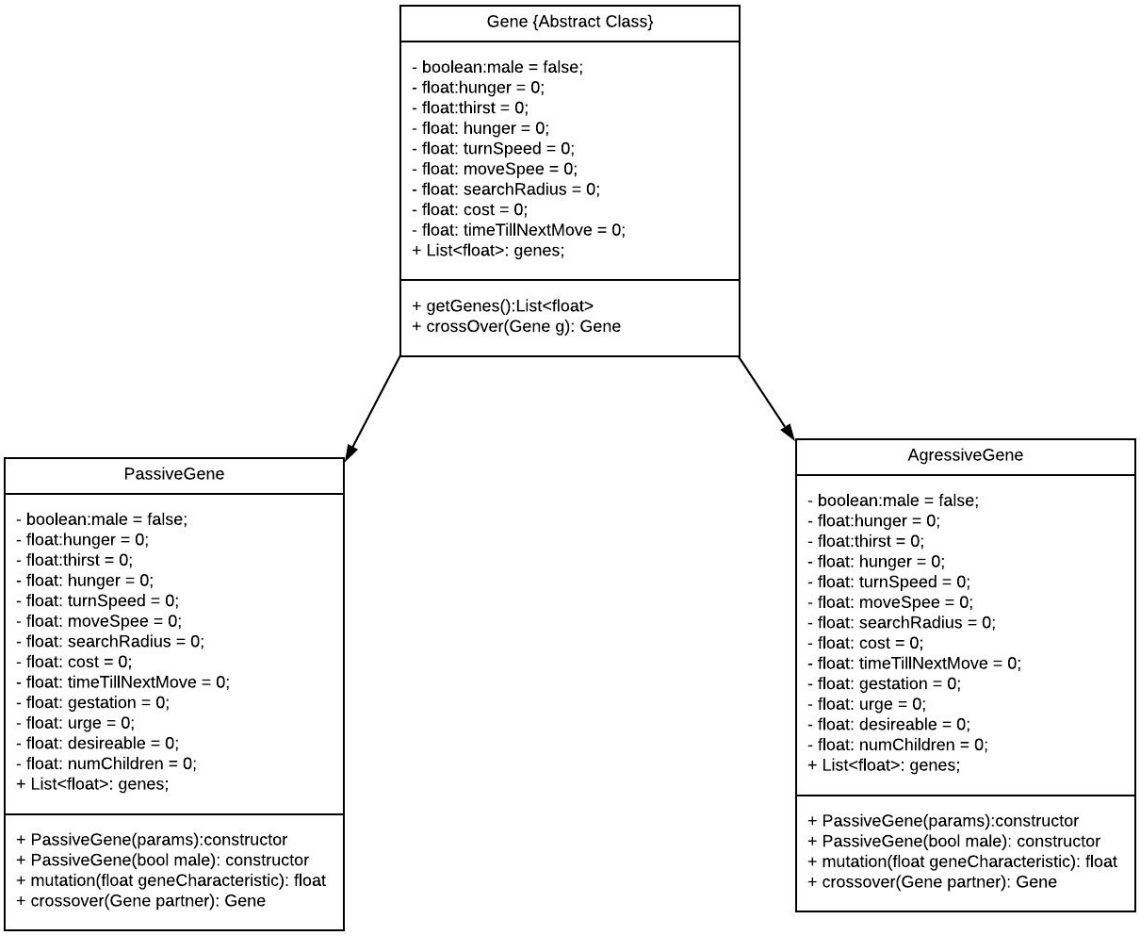
When it came to designing the agents who will inhabit the system, further research into game theory was used for inspiration. We read the book Ecological Applications of Genetic Algorithms. With that research came new addictions to how the agents behaved.

Agent Cost

Agents would now have a specified cost, which would be calculated based off of how favorable/less favorable their starts were. This cost value would be deducted from the agents hunger and thirst value. This added a level of balance to the population as agents who had a higher movement speed, were more likely to tire themselves out and would need to focus on search for water and food more than slower agents.
This was super important to implement as it prevents the agents from becoming too optimised. If we did not add this the passive agents could become uncatchable by the aggressive agents or inversely, the aggressive agents could wipe out the passive agents easily, starving themselves in the process.

Design

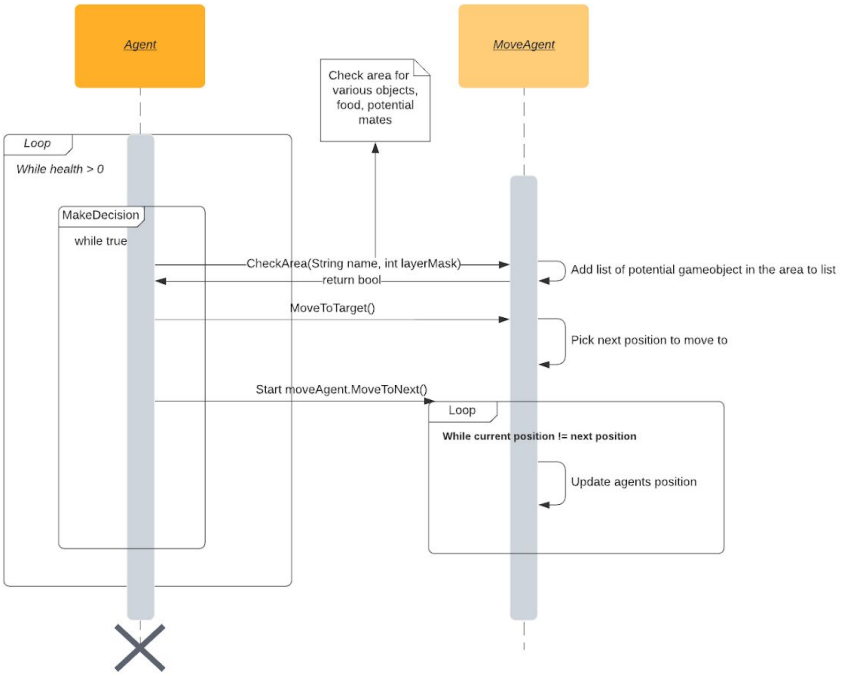
Design Hierarchy for the agent genes



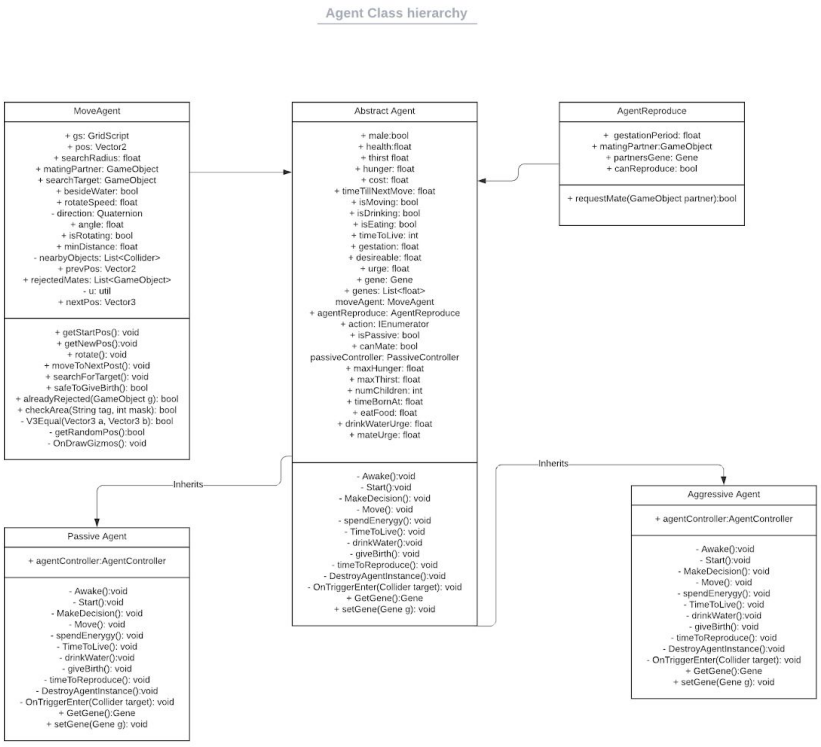
Sequence diagram for agent searching for potential targets

Agent sequence diagram

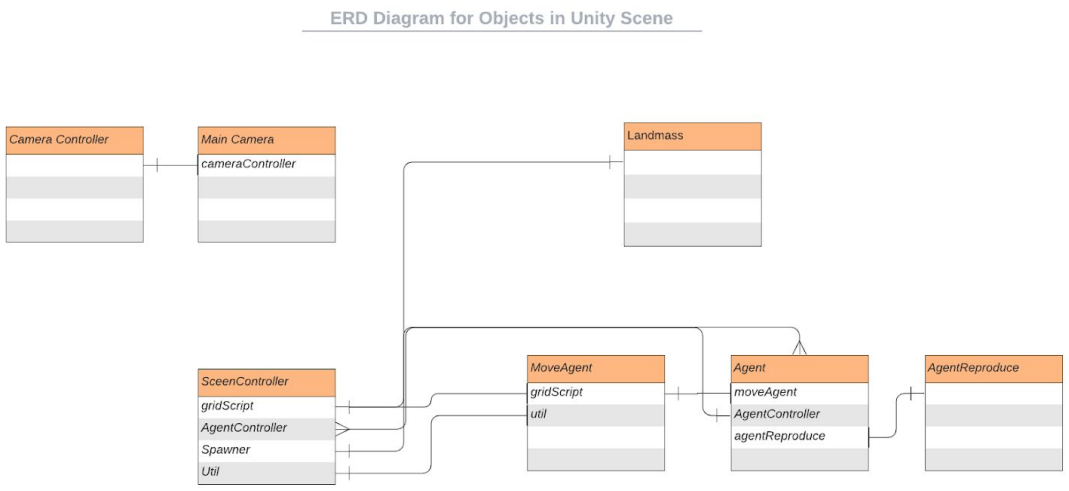
Jamie Hyland | May 15, 2020



Class hierarchy for agent scripts:



ERD Diagram for Objects in Scene



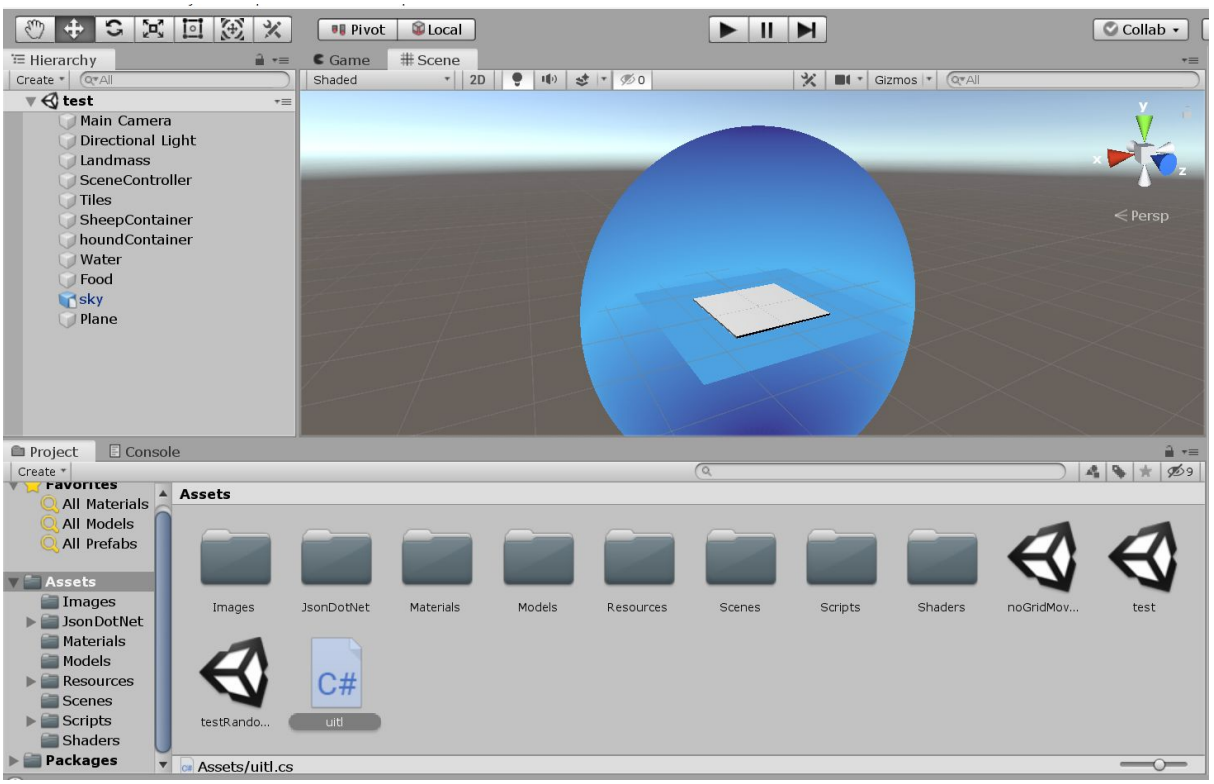
Implementation

The Unity Scene

A ‘Scene’ in the unity game engine is a file which contains a specific layout of objects, menus, scripts and so on. The main scene in our system is comprised of these objects:

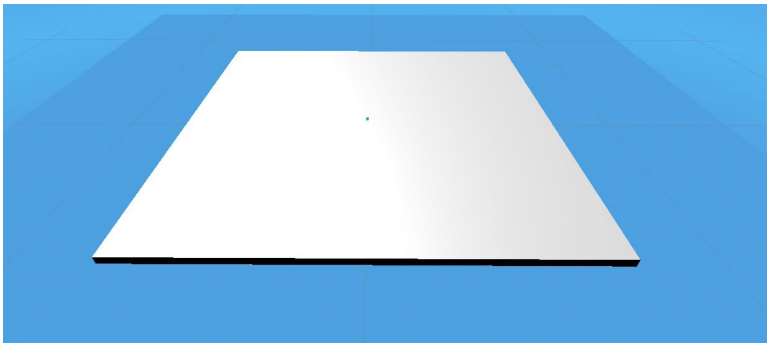
- Landmass - This is the landmass of the simulation. This acts as a surface area for the agents to walk on, it is home to both the food and water sources.
- Main Camera - The main camera from which we view everything in the system.
- Scene Controller - This is an empty GameObject (it contains no 3d model or physical space in the 3d environment) and contains a multitude of scripts that control various aspects of the scene such as:
    - Grid Script - this script takes a landmass and subdivides it into cells for agents to move on. This was used to help the agents maneuver around the landmass without collision or leaving the landmass.

    - PassiveController - This script keeps track of statistics for the passive agents in the scene, such as the number of deaths, and the current number of agents. It is also responsible for outputting a json file of these stats when the system shuts down.
    - Spawner - this script is responsible for spawning N number of objects into the scene, multiple instances of this script is applied to the scene controller to handle spawning of water, food and agents.
- Sheep Container - This is an empty container object that holds all instantiated passive agents, this is just to clean up the scene hierarchy.
- Water Container - An empty container object to hold water instances.
- Food Container - An empty container object to hold food instances.



The Landmass

The landmass was originally built using the 3D modelling program Blender. The idea for the landmass was to create a simple island which contained water sources and trees which would act as food sources. The majority of the area of the landmass is taken up simply by ‘grass’ where the agents can move around. The island was designed in such a way that water sources would be spread out amongst the landmass, this would force agents to actively search for different spots depending on where food was located.



Agent Genes

The agent genes are directly responsible for how fast an agent can move, how likely they are to find a mate, and how far they can travel among other things. There are two types of agents in the system: passive and aggressive agents. Aggressive agents have a higher range of values for their given traits, these higher range of values come at a larger cost.

The gene class is a basic abstract class from which the both passive and aggressive genes will inherit from and contains the method signatures both genes have to implement.

```
20 references
public abstract class Gene
{
    //class methods
    6 references
    public abstract List<float> getGenes();
    3 references
    public abstract float mutation(float geneCharacteristic);
    2 references
    public abstract Gene crossover(Gene one);
}
```

getGenes() simply returns a list of the various traits that each gene has.
Mutation applies a mutation to a specific gene trait, this can be either positive or negative.

Crossover is where the genes of the two parents are mixed together to form a new gene, during this phase is where the chance of a mutation happening occurs.

```
2 references
public override Gene crossover(Gene partner){
    //take half of each gene and mix them into a child gene
    List<float> parent1,child = new List<float>();
    parent1 = partner.getGenes();
    for(int i = 0; i < this.genes.Count;i++){
        float g = (u.randomGause()>0.5f)?parent1[i]:genes[i];
        child.Add((u.randomGause()>0.8f)?mutation(g):g);
    }
    return new PassiveGene(child);
}
```

GridScript - creating the grid

The grid script is responsible for taking a landmass and breaking it down into a set of cells that will be used for the agents to traverse the map. The script will contain two 2D arrays. One which contains 3D vectors, which the agents will access using a 2D vector.

This happens in the createGrid() method, which is a loop that assigns each index into the array a 3D vector. There is also a raycast that will scan the surface of the landmass. If it doesnt detect a surface it assumes it is a spot for a water tile, and does not place a valid position for an agent to move to at that point. This allows the system to handle multiple different types of landmasses that vary in the number of water sources.

```
//Subdivide the landmass into a grid of 3D positions for agents to move to
//Check using a raycast if there is ground underneath, if not specify that area as a place for water
    3 references
    void createGrid(){
        grid = new Vector3[h,w];
        for(int j = 0; j < grid.GetLength(0); j++){
            for(int i = 0; i < grid.GetLength(1); i++){
                float x = (i-landmass.transform.localScale.x/2) + tile.transform.localScale.x/2;
                float z = (j-landmass.transform.localScale.z/2) + tile.transform.localScale.y/2;
                int occupiedVal = occupied[j,i];
                grid[j,i] = new Vector3(x,2.75f,z);
                  r = new Ray(new Vector3(x,2.75f,z),Vector3.down);
                 RaycastHit hit;
                 if(Physics.Raycast(r.origin, Vector3.down, out hit,2f)){
                    var currentTile = Instantiate(water,new Vector3(x, 2.75f,z), tile.transform.rotation);
                    currentTile.transform.parent = GameObject.FindGameObjectWithTag("Tiles").transform;
                    occupied[j,i] = -1;
                }
                if(showGrid){
                    if(occupied[j,i] == 1){
                        var currentTile = Instantiate(visitedTile,new Vector3(x, 2.75f,z), tile.transform.rotation);
                        currentTile.transform.parent = GameObject.FindGameObjectWithTag("Tiles").transform;
                    }else if(occupied[j,i] != -1){
                            var currentTile = Instantiate(tile,new Vector3(x, 2.75f,z), tile.transform.rotation);
                            currentTile.transform.parent = GameObject.FindGameObjectWithTag("Tiles").transform;
                    }

                }
            }
        }
    }
```

Agent behavior - Agent Script

The agent script is the main brain of the agent itself, and contains the logic for deciding which actions to take, and are associated with keeping track of the various agent statistics.

When an agent is initialized the Awake method is called. This method is called once during an object's lifecycle and is used to initialize the various attributes of the agent. This method is a unity specific method which is common to all scripts which inherit from unities MonoBehaviour class. The agent uses this method to generate its specific genes (this only happens during the initial population as any other agents born are from two other agents) and to reference any external scripts such as the gridScript, and the passive/aggressive controller scripts which handle keeping track of the agents statistics.

The makeDecision() Coroutine is essentially the event loop for the agents. This function is called repeatedly until the agent dies. Each agent has 3 specific needs, a need for water, a need for food and a need to reproduce with a mate. Everytime an agent makes an action they expend energy, once this energy is expended 3 values corresponding to the 3 needs of the agent are calculated.

```
eatFood = 1-((hunger)/(maxHunger+thirst));
drinkWaterUrge = 1-((thirst)/(maxThirst+hunger));
mateUrge = 1-(urge-diff)/((urge-diff)+hunger+thirst);
```

This calculates a set of 3 weights which defines which action needs more attention. If the highest of the 3 weights is eatFood, the agent will search for food and so on for mating and drinking water. Once a specific action has been determined an agent will check the current area around them for the specified object (food, water, potential mate), this area is governed by their searchRadius genetic trait. If there are any potential objects around them the various searchFor() methods are called, these find the closest object in the area, and find the closest spot possible to that object.

The Move() Coroutine is a function which is responsible for moving the agent to its desired spot, this is done by calling moveAgent.moveToNextPos().

Drink water is another Coroutine which forces the agent to wait a specified time in one spot while it drinks water.

```
// wait for 5 seconds
// drink water then set action to null
2 references
public override IEnumerator drinkWater(){
    yield return new WaitForSeconds(5f);
    thirst = maxThirst;
    isDrinking = false;
    action = null;
    moveAgent.besideWater = false;
    moveAgent.searchTarget = null;
}
```

The giveBirth() coroutine is responsible for instantiating new agents into the scene. To do this the specific agent prefab is loaded from the assets folder and instantiated at the current position of the agent giving birth. The crossover event between the parents' genes is applied, the new gene created and is then applied to the child agent. The timeToReproduce() coroutine is then set which disables the agents ability to mate for a specific time.

```
//wait until gestation period is over then give birth,
//perform the crossover event between the two parents genes
//set the new gene to the new child
2 references
public override IEnumerator giveBirth(){
    yield return new WaitForSeconds(gestation);
    for(int i = 0; i < numChildren; i++){
        var baby = (GameObject)Resources.Load("Prefabs/Sheep");
        Gene partnerGene = agentReproduce.partnersGene;
        Gene babyGene =  this.gene.crossover(partnerGene);
        var child = Instantiate(baby,transform.position, Quaternion.identity);
        child.transform.parent = GameObject.FindGameObjectWithTag("sheepContainer").transform;
        child.GetComponent<Agent>().setGene(babyGene);
        child.GetComponent<MoveAgent>().pos = moveAgent.pos;
        child.transform.position = transform.position;
        yield return new WaitForSeconds(this.gestation/numChildren);
    }
    agentReproduce.matingPartner = null;
    action = null;
    canMate = false;
    StartCoroutine("timeToReproduce");
}
```

Agent movement - moveAgent Script

The purpose of this script is to hold a handful of methods for searching for, and moving to specific targets in the scene. These methods have been made public as they need to be able to be called from AggressiveAgent and PassiveAgent respectfully.

getStartPos() is responsible for getting references to the gridScript and the util Script from the sceneController. These scripts provide methods for accessing the grid that the agents move on and a utility script with various methods. A starting position is then set for the agent using getRandomPos()

```
//get the starting position on the map
2 references
public void getStartPos(){
    gs = GameObject.FindGameObjectWithTag("SceneController").GetComponent<gridScript>();
    u = GameObject.FindGameObjectWithTag("SceneController").GetComponent<uitl>();
    while(!getRandomPos());
    prevPos = pos;
}
```

getNewPos is responsible for getting a random position in the landmass. As the agents essentially move on a grid. Two random points are selected from an area defined by the search radius of the agent. If those points exceed the bounds of the landmass those values are clamped back to make sure agents don't move off the landmass. As all objects move on the same height, only the x and z parameters of the 3D position vector matter when selecting a position to move.

```
//pick a random x,y point in the search radius of the agent
//if the x,y goes out of the bounds of the landmass clamp it
5 references
public bool getNewPos(){
    prevPos = pos;
    int x = Mathf.FloorToInt(Random.Range(-1-searchRadius,searchRadius+1));
    int y = Mathf.FloorToInt(Random.Range(-1-searchRadius,searchRadius+1));

    pos.x+=x;
    pos.y+=y;

    if(pos.x > gs.maxX)pos.x = gs.maxX;
    if(pos.x < 0.0f)pos.x = 0;
    if(pos.y > gs.maxY)pos.y = gs.maxY;
    if(pos.y < 0.0f)pos.y = 0;

    return true;
}
```

moveToNextPos() is responsible for moving an agent from its current position to the next position. This is done by using a while loop, while the current position of the agent is greater than the minimum distance to the next position specified. The position is updated at the rate of the moveSpeed of the agent * Time.deltaTime. Time.deltaTime is the time taken since the last frame was called, this has the benefit of making the agents moveSpeed frame independant.

```
//Move to the next position
0 references
IEnumerator moveToNextPos(){
    nextPos = gs.getGridPosition(pos);
    yield return StartCoroutine("rotate");
        while(Vector3.Distance(transform.position,gs.getGridPosition(pos))>minDistance){
            this.transform.position = Vector3.MoveTowards(this.transform.position,gs.getGridPosition(pos), moveSpeed*Time.deltaTime);
            yield return null;
        }

}
```

The checkArea method is used to find certain objects in the scene, in a certain area. The method takes in a string for the tag of the specified object, and an integer value for a mask. This mask is used to only find a specific object that belongs to a specific layer. This provides a useful optimisation as the number of objects in the screen increases this prevents needless objects being included in the search for objects that aren't relevant to the search.

```
//return a list of objects around the searchRadius of the agent
6 references
public bool checkArea(string tag, int mask){
    nearbyObjects = new List<Collider>();
    int layerMask = 1 << mask;
    Collider[]  objs = Physics.OverlapSphere(this.transform.position,searchRadius,layerMask);
    if(objs.Length > 0){
        for(int i = 0; i < objs.Length; i++){
            if(objs[i].gameObject.tag == tag && !GameObject.ReferenceEquals(this.gameObject,objs[i].gameObject)) nearbyObjects.Add(objs[i]);
        }
    }
    return nearbyObjects.Count != 0;
}
```

moveToTarget() is a method which takes a gameObject in as a parameter and exits if the position is within the minimum distance to the specified target. If not the position vector for the agent is then set to the position vector of the gameObject passed to the method

```
public float moveToTarget(GameObject target){
 if(target.tag == "Player"){
     getNewPos();
     return 0;
 }
 if(target.tag == "Water" && Vector3.Distance(transform.position,target.transform.position)<=1){
     besideWater = true;
     searchTarget = null;
     return 1;
 }

 if((this.tag == "Sheep" && searchTarget.tag == "Sheep") && Vector3.Distance(this.transform.position, target.transform.position) <= 1 && !alreadyRejected(target)){
     Debug.Log("it worked");
     if(!target.GetComponent<AgentReproduce>().requestMate(this.gameObject)){
         rejectedMates.Add(target);
         target = null;
     }
     getNewPos();
     target = null;
     return 0;
 }
 if(target.tag == "Food" || target.tag =="Sheep" || target.tag == "Water"){
     Vector2 targetPos = new Vector2();
     if(target.tag == "Sheep")targetPos = target.GetComponent<MoveAgent>().pos;
     if(target.tag == "Water")targetPos = target.GetComponent<testWater>().pos;
     if(target.tag == "Food")targetPos = target.GetComponent<foodTest>().pos;

     pos = targetPos;
     minDistance = (target.tag == "Food" || (target.tag == "Sheep" && this.gameObject.tag == "Hound"))?0.2f:1.0f;}
 return Vector3.Distance(this.transform.position,target.transform.position);
}
```

Agent Reproduction - agentReproduce Script

requestMate() is a method which takes in a gameobject as a parameter, and is responsible for deciding whether a female agent is likely to reproduce with a male agent. A random value

is selected from 0-1 and if that number is lower than the male agent's desire score then the female will mate with the male, if not they are rejected. If they are accepted a copy of that agent's gene is initialized to a variable for use later during the crossover event which happens during birth.

Problems Solved

Optimising code to increase agent count

Coroutines

One of the biggest problems that was encountered when developing the system was handling hundreds of agents on screen. The first implementation of the system could run only about a dozen agents averaging at around 30 Frames per second. Checking unity's profiler we found that a large portion of the CPU memory allocated to unity was being allocated to the update() function of the Agents. The update() function is used within any class that extends from Unity's MonoBehavior class and is called once per frame. Normally this would be used to read user input for a game, and would only have a few objects in the scene implementing this method. Containing all our Agent logic within this method was having a large overhead on the system's performance. To remedy this, we used Coroutines. A Coroutine is a Unity Construct that acts like a method, however, can give control back to the program at certain points in the method as designated by the developer. Converting our Agent logic that was contained inside the update() method

Results

Data Gathered on Agents

Prerequisites to Testing

To test how the agents evolve over time, a script was made to output various statistics to a json file which can be examined in R. The values we're testing are:

- Health
- Hunger
- Thirst
- Search Radius
- TurnSpeed
- MoveSpeed
- Gestation
- Desirability

- Number of children
- Cost

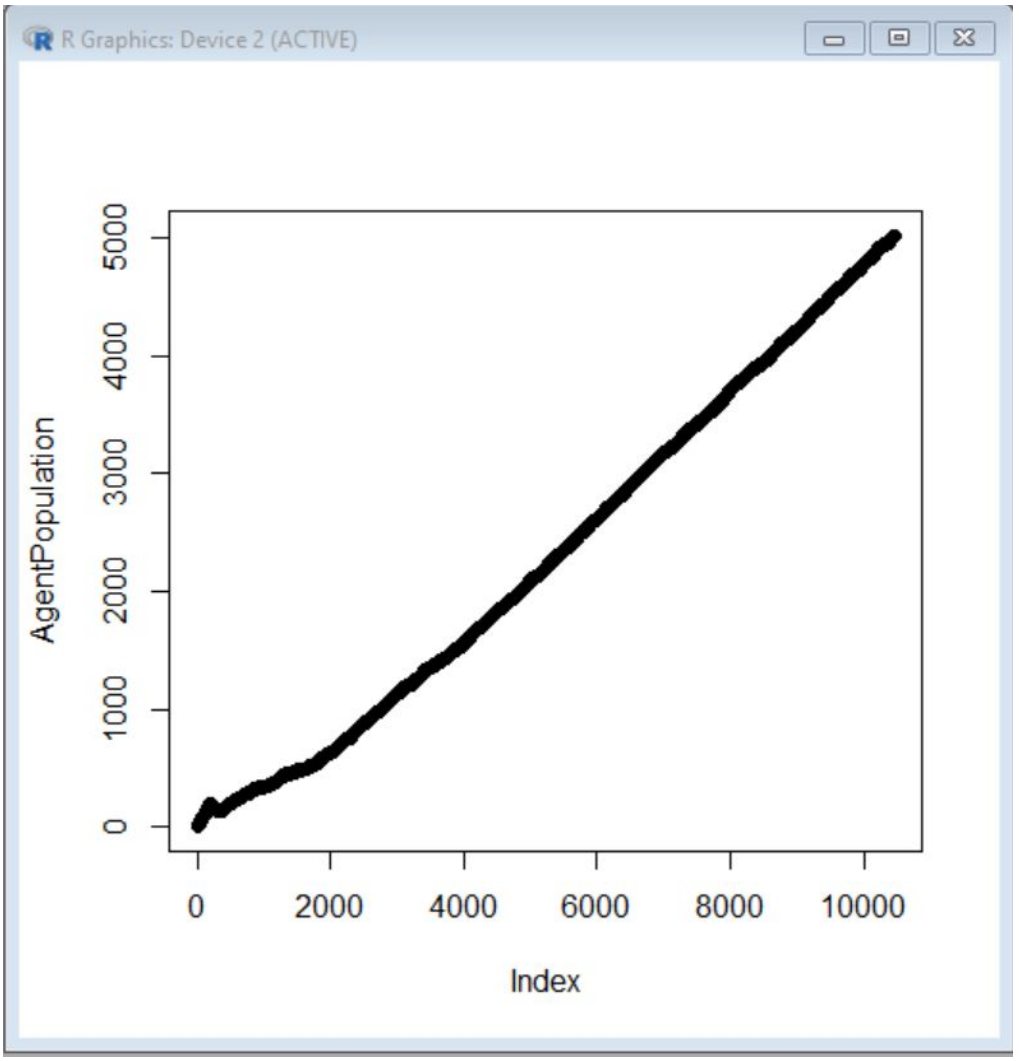
Looking at the evolution of these values over time will allow us to see which traits were important in the survivability of the species and why. Tests were conducted with just the passive agents grazing on the field by themselves, and then both the passive and agents together to test how the predator prey relationship affects the traits of the agents.

The scene is set up with a number of Food and Water Tiles. Both the aggressive and passive agents will drink from the water tiles while the passive agents will eat from the food Tiles. Once a passive agent has eaten from a food tile, that food tile is deactivated for a specified time.Every object in the scene starts on a random tile in the scene. Each test will outline how many water tiles were spawned, how many food tiles were spawned how many passive agents were spawned and how many aggressive agents were spawned. We will also discuss things we noticed from studying the system as it runs as long as studying metrics gathered in our JSON files using R. As a lot of the tests are somewhat similar in the outcome, we will outline general observations we found during the many tests we ran, while highlighting specific cases of running the system.

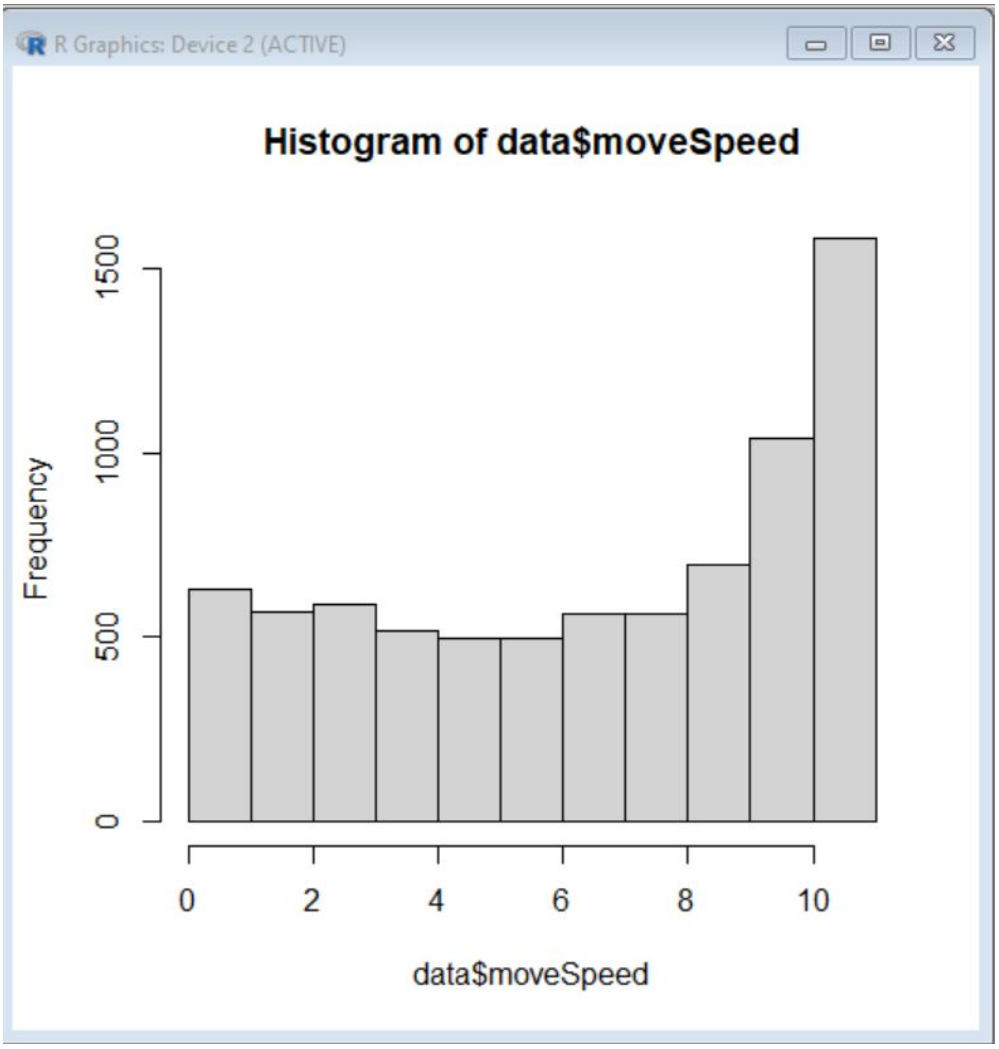
Testing the scene with only passive agents

Our initial tests with just the passive agents showed us some inconsistencies with how we designed the supply of food and water provided. We found that giving Agents too much food and water simply made them complacent, and would not stray further than the nearest food and water source. This became a problem when the population sizes grew smaller, and agents were not able to get near enough to mate with each other. To remedy this we decreased the number of water tiles in the scene, forcing agents to have to discover more sources of water. This in turn would lead to more reproduction, and in turn a greater genetic variety over time.

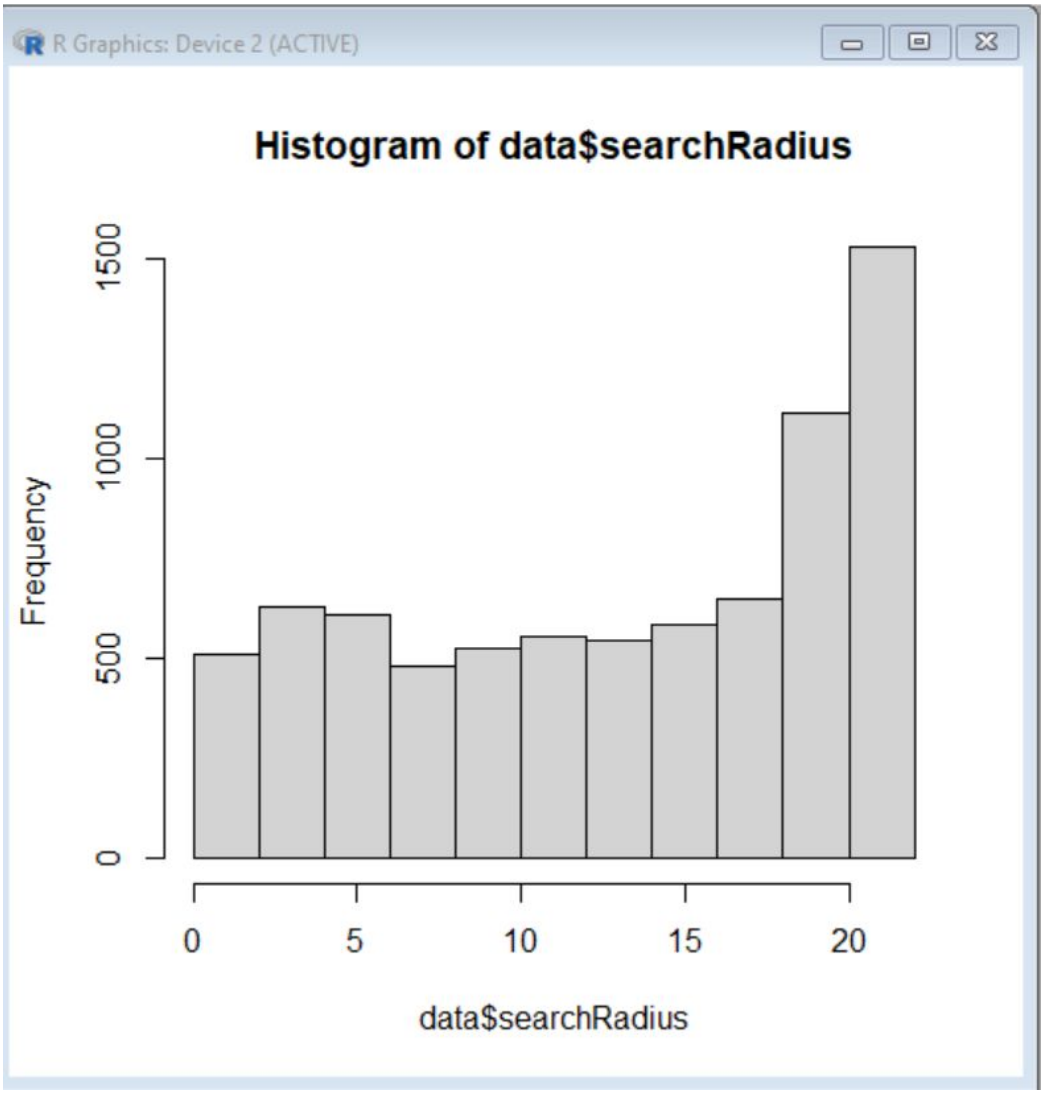
We had made the assumption that as resources were finite, as the population grew, there would be a large drop off in the number of agents, this would essentially weed out the less desirable, slow limited ranged agents, the ones less likely to grab food or water. Leaving only the fittest agents, this process would continue indefinitely as the population size grew and shrunk. However, in our tests we were unable to find that balance. Most of the time there would be a large initial drop off of weak agents followed by a slow inevitable climb to overpopulation.



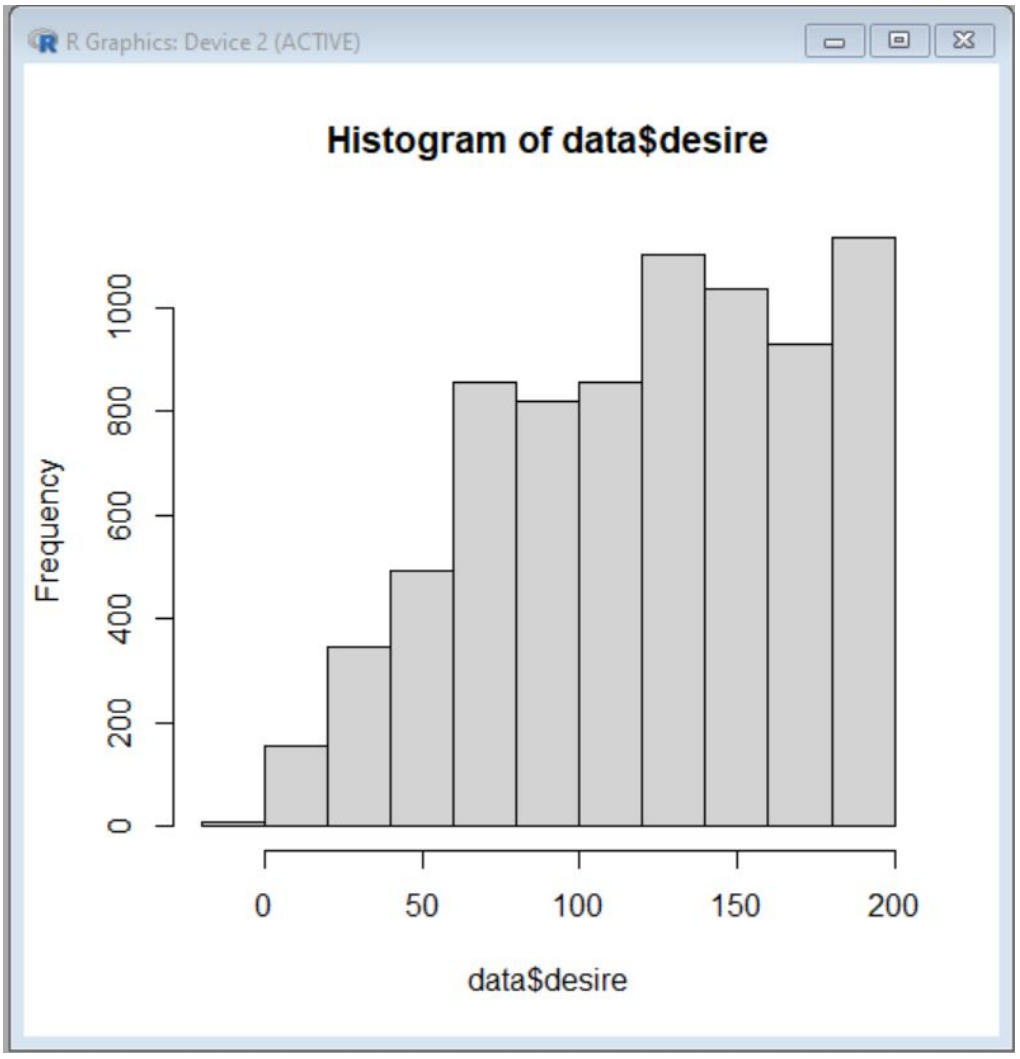
As you can see above there is the initial population followed by a small dip until a large climb to overpopulation. Examining our genetic data we could see that there was an evolution in certain traits.The figure below shows that over time the system favored agents with a high move speed. Likely due to the increasing number of agents in the scene and the growing competition for food.



The same can be said for the searchRadius of the agents.This showed us again, as competition was fierce it was more favorable, to have a higher search radius, this allowed agents to find food and water that was further away, it also allowed them to move greater distances which in turn let to a bigger chance of finding food want water or a potential mating partner.

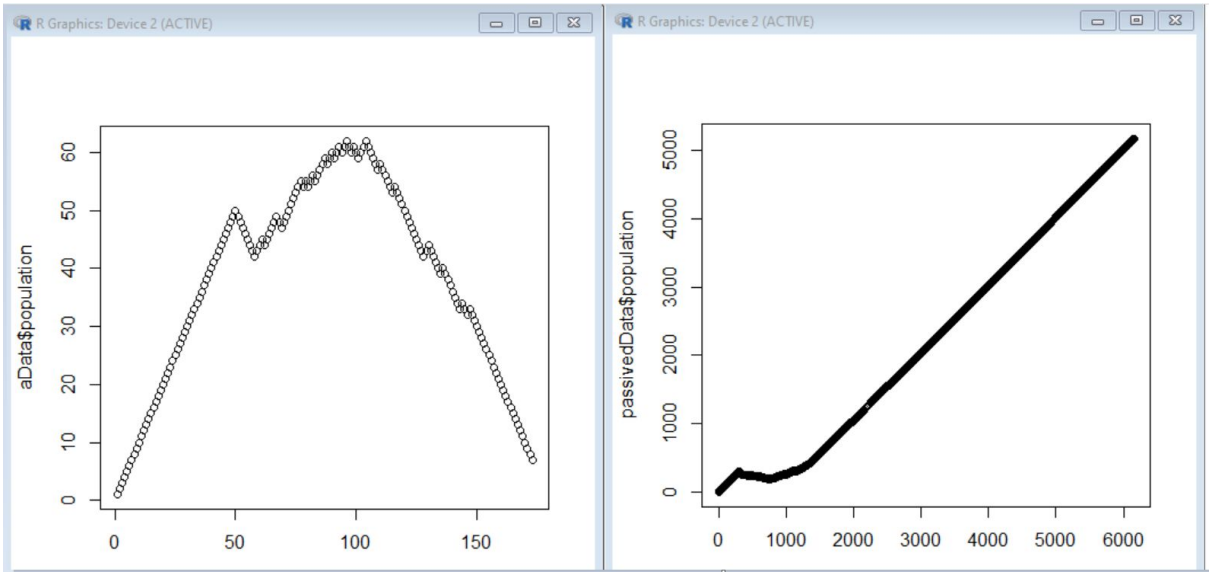


Other key traits like gestation, the time it takes to reproduce were generally stable. This can most likely be due to the lack of danger, thus the need to reproduce quickly to pass on their genes in the system as only the passive agents were in the scene, meaning it didn't matter how long it took for an agent to give birth. One of the biggest reasons there was a constant case of over population without any major dips in the population was the desire trait. The desired trait was a value initially set between 0-100 and was a probability of an agent being selected as a mating partner by a female agent. This value mutated over time in such a way that agents were always likely to find a suitable mating partner.



Testing the Scene with Passive and Aggressive Agents

Our initial dry run tests with both passive and aggressive agents were the same as the tests without aggressive agents. We spawned the same number of passive and aggressive agents, with a low amount of water tiles (generally 20-50 tiles). This quickly ended in the extinction of both species as the aggressive agent immediately wiped out the passive agents before they had a chance to properly procreate. This meant we had to spawn a low number of aggressive agents into the scene in order to ensure the passive agents had a chance to procreate.



The image above shows the aggressive agents on the left and the passive agents on the right, you can see as the initial population of the aggressive ages rises there is a dip in the number of passive agents.However, the passive agents are quickly able to recover and over populate the scene while the aggressive agents slowly die out. This was due to the aggressive agents over-populating a specific area of the landmass, the passive agents stayed away from that area and only procreated further away from the aggressive agents. While only a few aggressive agents actually found where the passive agents were.

Unfortunately we were unable to find a solution where the agents could co-exist. At every option we tried the aggressive agents quickly decimated the passive agents, or the passive agents survived long enough to completely over populate the dwindling aggressive agents, who were never able to recover. When examining possible causes for this, we found that there were a number of reasons why. The cost trait, which is how much energy an agent spends each time they move, plays a big factor in the success or failures of the species. As the cost trait grows by how fast and how far an agent can travel, it essentially also shortens their lifespan an incredible amount. It can be safe to assume that because the aggressive agents were too fit, they were constantly expending too much energy, and thus couldn't procreate at a fast enough rate to match the ever growing population of passive agents.

The landmass itself played a huge factor in the success/failure of both species, as the agents essentially conducted a ‘random walk’ until they found the food/water/mate they were searching for. There is a chance that an agent could never move in the right direction towards their desired resource. This was noticed multiple times when testing both species coexisting where the aggressive agents merely occupied one half of the landmass, while the passive agents occupied the other half this led to the dying out of the aggressive agents while the passive agents thrived.

Testing

Ad hoc

For ad hoc testing we made ample use of Unitys Scene functionality. This allowed us to create separate spaces where we could set up an environment to accurately test a specific functionality.

Food & Water search Test

To test for agents searching for food, we created a 5x5 grid for one agent to move on. A custom foodTest script was made. This script would be attached to a game object. Every time an agent collided with a piece of food their hunger was replenished and the food would immediately teleport to another location on the grid. This could be scaled up for any N number of agents or pieces of food to ensure that the mechanism for searching for food behaved correctly.

The same test was carried out for searching for pieces of water, and could be scaled to any N agents and N number of water sources.

Search for mate Test

This test was more involved, it was set up similarly to food and water search, however, we needed to test once two agents got close enough that the request for mate was handled correctly. Our agents have the chance to decline a request for mating if a certain agent's desire score is quite low. Upon our initial tests for this feature we found that if an agent was declined/accepted, they would constantly search for the agent that rejected them and ignore food and water sources this led to the scene becoming over populated with agents. To fix this we introduced a list of rejectedMates for the male agents. If a male was rejected by a female they would add that female to the list. Upon requesting to mate with a female a male would check to see if that female was already in the list of rejected mates, if they were they would stop advancing and go look for another mate. For mates that were successful we added a time to reproduce and a gestation period for females. Females would have to wait a specified time, governed by a trait in their genetics until they can give birth. They also have a time until they can reproduce again. This helped us prevent over population of the system at an alarming rate.

Unit Testing

Unit testing in Unity is widely known as a very difficult aspect of using Unity. Most developers don't unit test their games in Unity due to how difficult it generally is. This difficulty is usually caused by dealing with monobehaviours or interacting with a game object.

Monobehaviours make it difficult to access methods from a class while in a test script. To combat this issue, we would essentially need to refactor every monobehaviour in our project. This is a very time consuming endeavour and unfortunately, at the time of realisation, we lacked the necessary time to proceed with this.

Future Work

When considering working on this application in the future, or rebuilding it again from the ground up, here are some things we've outlined that could be improved, added or changed in order to make a better system.

The system demonstrates the ability to show evolution of a species overtime in accordance with the environment in which agents are placed into. However, a few key things keeps it from being as accurate as it could be when dealing with a research project. The random walk of the agents leaves a lot at random, this significantly increases the amount of time needed for running the system to find favorable results, and essentially adds a luck element to the system and is wholly undesirable. To remedy this a proper pathfinding algorithm could be introduced for the agents. Specifically a D* algorithm this stands for 'Dynamic A*' and is used for pathfinding for agents in a changing environment.

The game theory aspect of the project could have been developed more, with key examples being:

- A flocking system that allows passive agents to warn each other of prey that are off in the distance
- Aggressive agents being able to share food or fight each other for food, essentially simulating the prisoners dilemma problem
- Passive Agents could forage food and share it amongst other agents.

Although there is ample enough data being generated from the system, in a research setting more data could be extracted from the agents. A binary tree system where new agents born into the system keep a track of their parents. Essentially creating a family tree. Researchers could then follow these branches to see how specific agents genes were passed on and mutated throughout the life cycle of the program.

The system could also benefit from a UX standpoint. The system shows little user feedback and could be expanded upon by giving the user options to pause the system, change specific traits of a specific agent, modify the terrain in real time and so on. We feel like these additions would help show the variability of the genetic behaviors, and would increase the longevity of the system being used by researchers.

Unit Testing

Unit testing in Unity can be a very large job as refactoring of the code needs to take place. It is rare and difficult to create a unity project which is already set up for unit testing. With more time to prepare for unit testing in this project, we would make use of Nsubstitute. NSubstitute is a friendly substitute for .NET mocking libraries. It has a simple, succinct syntax to help developers write clearer tests. We have learned the importance of test driven development as a result of this project and will always remember to apply that where possible.

References

- Weimann, J. (2020). *Unit Testing in Unity3D - Testing Against Monobehaviors using Mocks*. *YouTube*. Available at: https://www.youtube.com/watch?v=r7VkbV0PRC8 [Accessed 6 April 2020].
- Yale.edu. (2020). *Game Theory | Open Yale Courses*. [online] Available at: https://oyc.yale.edu/economics/econ-159 [Accessed 16 November 2019].
- Morrall, D. (2003). Ecological Applications of Genetic Algorithms. *Ecological Informatics*, [online] pp.35–48. Available at: https://link.springer.com/chapter/10.1007/978-3-662-05150-4_3 [Accessed 12 March 2020].
- Unity Technologies (2019). *Unity - Manual: Unity User Manual (2019.3)*. [online] Unity3d.com. Available at: https://docs.unity3d.com/Manual/index.html [Accessed 16 May 2020].
- Lague, S. (2020). *Coding Adventure: Simulating an Ecosystem*. *YouTube*. Available at: https://www.youtube.com/watch?v=r_It_X7v-1E [Accessed 2 October 2019].
- Primer (2019). *Simulating Natural Selection*. *YouTube*. Available at: https://www.youtube.com/watch?v=0ZGbIKd0XrM [Accessed 2 October 2019].
- Blender (2019). *First Steps - Blender 2.80 Fundamentals*. *YouTube*. Available at: https://www.youtube.com/watch?v=MF1qEhBSfq4&list=PLa1F2ddGya_-UvuAqHAksYnB0qL9yWDO6 [Accessed 5 December 2019].

- DAWKINS, R. (1989). *The selfish gene*. Oxford, Oxford University Press.
- Jacksondunstan.com. (2017). *JacksonDunstan.com*. [online] Available at: https://jacksondunstan.com/articles/3746 [Accessed 26 January 2020].
- Blogspot.com. (2012). *Really Really Easy Multithreading in Unity3D.* [online] Available at: https://entitycrisis.blogspot.com/2012/08/really-really-easy-multithreading-in.html [Accessed 26 January 2020].
- Flick, J. (2020). *Procedural Grid, a Unity C# Tutorial*. [online] Catlikecoding.com. Available at: https://catlikecoding.com/unity/tutorials/procedural-grid/ [Accessed 15 March 2020].
- thoughtbot. (2017). *How to Git with Unity*. [online] Available at: https://thoughtbot.com/blog/how-to-git-with-unity [Accessed 20 November 2019].
- One Wheel Studio (2020). *1: Day Night Cycle - Sun Movement and Color*. *YouTube*. Available at: https://www.youtube.com/watch?v=babgYCTyw3Y [Accessed 10 February 2020].