Group Members: Brian Corpus

Project Type: Use Genetic Algorithm to Solve an Obstacle Course

GitHub Repository: bcc6912 (https://github.com/bcc6912/GameAI)

Video Link: https://youtu.be/yLVrlxS-OoE

How the Genetic Algorithm Works:

1) Initial population is spawned
    a) A sequence of 200 different floats is randomly generated (all fitting inside 360 degrees) which represents each direction the NPC will turn to
2) Each NPC moves forward in the direction its currently facing
    a) NPC moves in its forward direction for 0.2 seconds
    b) If NPC hits a wall while moving, it has crashed and will stop moving
    c) NPC will also stop moving if the red wall behind it passes it
3) Once all NPCs stop moving, the fitness of each NPC is checked
    a) Calculates how far the NPC is from the destination, how far the NPC is from the next checkpoint, how many checkpoints the NPC has passed, and if it has crashed into anything
    b) The NPC with the highest fitness of this generation is recorded and marked with a dark cyan pill and its path is drawn
        i) If this fitness is higher than the highest fitness recorded across all generations, it is marked with a purple pill and its path is drawn
    c) The NPC that is the closest to the goal is marked with a cyan pill and its path is drawn
4) Of the NPCs in the current generation, the top 10% in terms of highest fitness will be used for the next generation
    a) Locations of top 10% are marked with a yellow sphere
    b) The cutoff can be changed in inspector before starting simulation
    c) Lower 90% get deleted
5) Sequence of directions of surviving NPCs is then mixed together, with a chance of mutation
    a) Mutation: Current index of the sequence will change to a new randomly generated direction
    b) Mutation rate can change dynamically
    c) Mutation rate gradually increases if there is no progress for extended periods of time (if the NPC with the highest fitness does not surpass the currently highest recorded fitness for four generations, the mutation rate will increase. The mutation rate will go back to its original rate if it does surpass the highest recorded fitness level)
6) New population is spawned and the process repeats from step 2 until the goal (blue box) is reached

Best/Most Cleverly Implemented Parts:

- Discouraging using NPCs that crashed, or would crash if they went any further forward
    - NPCs that crashed would have their fitness score significantly reduced
    - If the NPC has not crashed, a line is projected forward in the NPC's last direction and it checks if a wall is a few units in front of the NPC. Its fitness score would be reduced if this was true
- In the event the generations are hitting something of a wall, the mutation rate gradually increases on its own as mentioned in Step 5 of how the genetic algorithm works
- To encourage the use of a more efficient path, the moving wall was added
    - Lowers the fitness of NPCs if they are hit
    - Also speeds up the simulation time in case NPCs are moving in a circle

Difficult Parts:

- NPCs would not go toward the path they need to take in order to get past the current obstacle
    - Checkpoints are set up at the end of each obstacle that are put into the calculation of the NPC's fitness
    - Each checkpoint grants +1 fitness to the fitness score of the NPC, and the distance between the NPC's current target checkpoint and the NPC's final location is also calculated into the NPC's fitness score

Good/Bad Parts:

- Good: The genetic algorithm works as intended and the "DNA" of the surviving members of the previous generation mix together correctly to create better NPCs for the next generation
- Good: Markings for the best recorded NPC of the last generation, best recorded NPC overall, and the furthest NPC of the last generation
- Bad: Despite the single Random.InitState() call, there seem to be no consistent results when running the simulation multiple times. This was also tested via debug logging a Random.Range call after calling Random.InitState(). This result always came up with the same float (658.4666).
- Bad: Some calculated fitnesses appear as though they should be the best recorded, but are not.

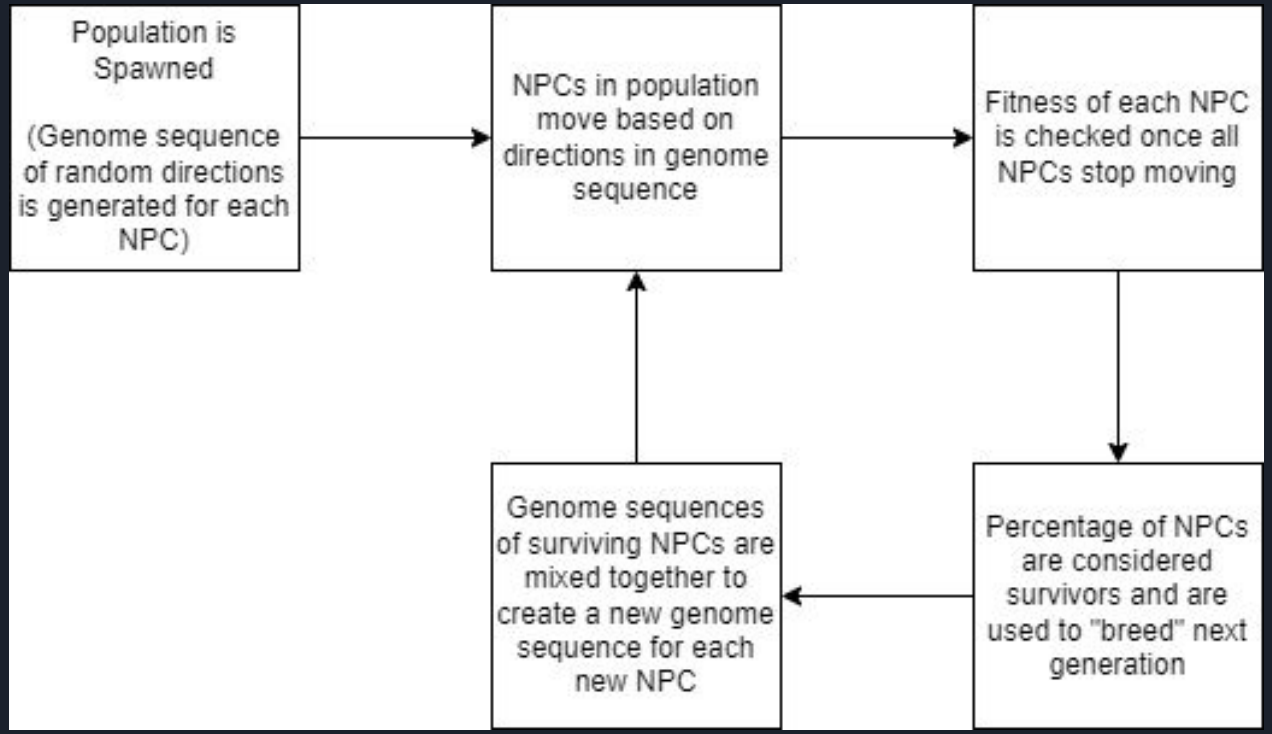
What Worked Well:

While using a separate script to instantiate the DNA worked well, the utilization of such a script seemed redundant as the results were the same if I did not. Rather than keeping the instances of the NPC through each generation and just changing the values within the GeneticPathfinder script, I found it easier to just destroy the current instances and create new ones. It also made for better generations as the simulation progressed.

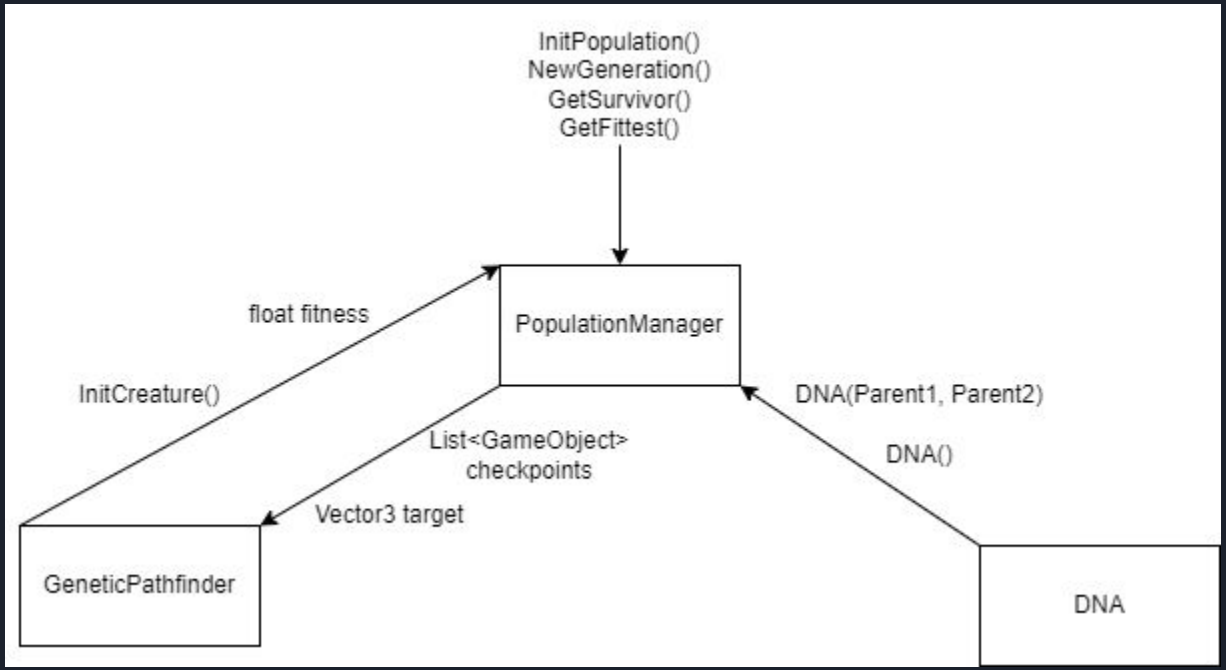
What I Could Do Better

- Better instantiate the NPCs so it can be more efficient
- Mark more NPCs ie. last moving NPC and its path
- Change collision detection so set random seeds will be consistent

Genetic Algorithm Diagram



Class Diagram



DNA Constructors

```
public DNA(int geneLength = 50)
{
    for (int i = 0; i < geneLength; i++)
    {
        genes.Add(Random.Range(0.0f, 360.0f));
    }
}

public DNA(DNA parent1, DNA parent2, float mutationRate = 0.01f, float mutationIncreaseCount = 0.0f)
{
    for (int i = 0; i < parent1.genes.Count; i++)
    {
        float mutationChance = Random.Range(0.0f, 1.0f);
        if (mutationChance <= mutationRate + mutationIncreaseCount)
        {
            genes.Add(Random.Range(0.0f, 360.0f));
        }
        else
        {
            int chance = Random.Range(0, 2);
            if (chance == 0)
            {
                genes.Add(parent1.genes[i]);
            }
            else
            {
                genes.Add(parent2.genes[i]);
            }
        }
    }
}
```

Spawning Next Generation

```
void NextGeneration()
{
    GetFarthest();

    int survivorCut = Mathf.RoundToInt(populationSize * survivorCutoff);
    List<GeneticPathfinder> survivors = new List<GeneticPathfinder>();

    for (int i = 0; i < survivorCut; i++)
    {
        survivors.Add(GetFittest());
    }

    GetRecord(survivors); // find out if mutation rate needs to increase

    for (int i = 0; i < population.Count; i++)
    {
        Destroy(population[i].gameObject);
    }
    population.Clear();

    while (popula[illegible].Count < populationSize)
    {
        for (int i = 0; i < survivors.Count; i++)
        {
            int parent2 = Random.Range(0, survivors.Count);
            while (parent2 == i)
            {
                parent2 = Random.Range(0, survivors.Count);
            }
            GameObject newNPC = Instantiate(pillPrefab, spawnPoint.position, Quaternion.identity);
            newNPC.GetComponent<GeneticPathfinder>().InitCreature(new DNA(survivors[i].dna, survivors[Random.Range(0, survivors.Count)].dna, mutationRate + mutationIncreaseCount), end.position, checkpoints);
            population.Add(newNPC.GetComponent<GeneticPathfinder>());
            if (population.Count >= populationSize)
            {
                break;
            }
        }
    }

    for (int i = 0; i < survivors.Count; i++)
    {
        Destroy(survivors[i].gameObject);
    }

    deathWall.Reset();

    generationNum++;
}
```

Mutation Rate Increase Code

```
if (maxFitness <= recordFitness + 0.0001f)
{
    rateIncreaseCounter++;
    if (rateIncreaseCounter >= rateIncreaseThreshold + 1)
    {
        mutationIncreaseCount += 0.001f;
        rateIncreaseCounter = 0;
    }
}
else
{
    recordFitness = maxFitness;

    recordNPC.ChangePosition(survivors[index].transform.position);

    survivors[index].path.Add(survivors[index].transform.position);

    recordNPC.DrawLine(survivors[index].path);

    // Debug.Log(survivors[index].path.Count);

    rateIncreaseCounter = 0;
    mutationIncreaseCount = 0.0f;
}
```

Fitness Calculation Code

```
public float fitness
{
    get
    {
        path.Add(transform.position);

        RaycastHit[] frontObstacles = Physics.RaycastAll(new Ray(transform.position, transform.forward), 5.0f, obstacles);

        obstaclesInRange = frontObstacles.Length;

        Vector2 currentPos = new Vector2(transform.position.x, transform.position.z);
        Vector2 targetPos = new Vector2(target.x, target.z);

        float targetDistance = Vector2.Distance(currentPos, targetPos);

        if (targetDistance == 0)
        {
            targetDistance = 0.0001f;
        }

        float checkpointDistance = 0.0f;
        float checkpointModifier = 0.0f;

        if (targetCheckpoint <= 5)
        {
            List<Vector2> targetCheckpointPos = new List<Vector2>();

            foreach (Checkpoint c in checkpoints)
            {
                if (c.checkpointNum == targetCheckpoint)
                {
                    targetCheckpointPos.Add(new Vector2(c.transform.position.x, c.transform.position.z));
                }
            }

            float shortestCheckpointDistance = float.MaxValue; // changes to distance to closest not reached checkpoint

            foreach (Vector2 v in targetCheckpointPos)
            {
                float tempDistance = Vector2.Distance(currentPos, v);
                if (tempDistance < shortestCheckpointDistance)
                {
                    shortestCheckpointDistance = tempDistance;
                }
            }

            checkpointDistance = shortestCheckpointDistance;

            checkpointModifier = (100 / checkpointDistance) * ((float)checkpointCount / 2.5f);
        }

        if (hasCrashed)
        {
            if (obstaclesInRange == 0)
            {
                Collider[] obstaclesInRadius = Physics.OverlapSphere(transform.position, viewRadius, obstacles);
                obstaclesInRange = obstaclesInRadius.Length;
            }

            float obstacleMultiplier = 1f - (0.15f * (obstaclesInRange));

            return (100 / (targetDistance + checkpointDistance)) *  0.6f * obstacleMultiplier + checkpointCount;

        }
        else
        {
            if (hitDeathWall)
            {
                float obstacleMultiplier = 1f - (0.15f * (obstaclesInRange));

                return (100 / (targetDistance + checkpointDistance)) * 0.601f * obstacleMultiplier + checkpointCount;
            }
            else
            {
                float obstacleMultiplier = 1f - (0.15f * (obstaclesInRange));

                return (100 / (targetDistance + checkpointDistance)) * obstacleMultiplier + checkpointCount;
            }
        }
    }
}
```

Distance Calculation

Crashed Calculation

Checkpoint Calculation