FIT3169 Assignment3

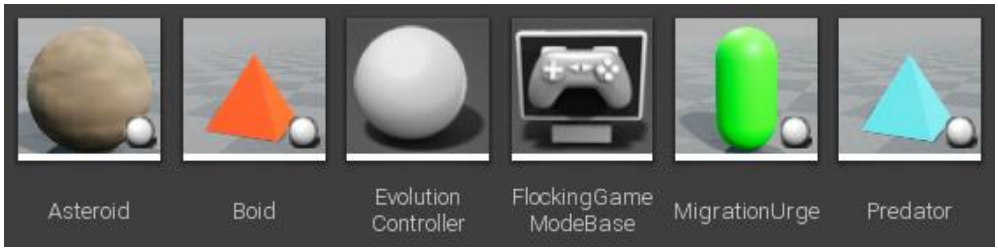
Qixuan Xin

28732138

Table of Contents

Introduction

All the agents' visual representations are shown above.



The velocity of the boids is adjusted by weight. Therefore, in this assignment I'm trying to use evolutionary algorithm to find the optimized solution of the weight of the avoid, velocity matching and flocking centre. When a boid can survive 300 seconds or more, the program terminated and the weights of this boid is the optimized solution.

```
velocityVector += avoidVector * avoidWeight + alignVector * alignWeight + tendencyVector * trendWeight;
```

The boids implement the following behaviour:

- Avoid other boids, asteroids and predators
- Matching the velocity of other boids and asteroids
- Trying to go to the centre of the surrounding boids
- When reach the target location, move to next target location
- Boid will not migrate at the start of the game, but every 30 second they will migrate once.

Pre-defined Genome Type

There are four genome types I defined. The four pre-defined types are shown below:

```
case INIT_GENOME::Normal:
    avoid = 1;
    align = 1;
    trend = 1;
```

```
case INIT_GENOME::AvoidFocus:
    avoid = 2;
    align = 0.5;
    trend = 0.5;
```

```
case INIT_GENOME::AlignFocus:
    avoid = 0.5;
    align = 2;
    trend = 0.5;
```

```
case INIT_GENOME::TrendFocus:
    avoid = 0.5;
    align = 0.5;
    trend = 2;
```

The range of the weights is 0.5 – 2. Therefore, all the weights of the normal genome type are 1. The focus type means this type only focus on the specific weight and careless other weights. The reason of doing so is because I do not know which weight should care more in the boids' action sequence. If the avoid weight is more important, the genome with higher avoid weight will survive more time and has more possibility to breed the offspring. And the offspring will inherit this feature and repeat this loop. This is why I make them only focus on one of the weights. Therefore, in each generation the genome will crossover and mutate and finally show the optimized solution like this: Avoid: 1.6, Align: 0.8, Trend 1.3

Parent Selection

The parent selection method I chosen is “Roulette Wheel”. This method allows the genome with higher fitness to have higher possibility to be chose as parents. This feature can help the evolutionary algorithm to generate the global optimized solution rather than the local optimized solution. If we select parents only by the magnitude of the fitness, the algorithm can generate local optimized solution but may not be the global optimized solution.

```
ABoid* AEvolutionController::ParentalSelection()
{
    // select the parent by roulette wheel selection
    ABoid* parent = nullptr;

    while (!parent) {
        float selectPointer = FMath::FRandRange(0.f, 100.f);

        for (TMap<ABoid*, float>::TIterator it = rouletteWheel.CreateIterator(); it; ++it) {
            selectPointer -= it->Value;
            if (selectPointer < 0) {
                parent = it->Key;
                break;
            }
        }
    }

    return parent;
}
```

Population Selection

The replacement rule of the population selection is only replacing the failed population (the dyed boids). I set this rule because I think the alive boids in the odd generation still have chance to reach the final goal. Therefore, they are going to be counted as the new generation offspring rather than the odd generation.

```
void AEvolutionController::SurvivorSelection()
{
    // loop through all died solution and reset them with the new genome
    // the alive boids continue their work
    int childrenIndex = 0;
    for (int i = 0; i < population.Num(); i++) {
        if (population[i]->isDied) {
            if (childrenIndex < children.Num() - 1) {
                population[i]->Breed(children[childrenIndex]);
                childrenIndex++;
            }
            else {
                population[i]->Breed(children[childrenIndex]);
            }
        }
    }
}
```

Problems Encountered

In the parent selection part, sometimes the parent selection will return nullptr. I do not know it is because the proportion is wrongly calculated or some subtle code execution order problems. What I did to solve this problem is, add a while loop outside of the selection part. If the parent is a nullptr, select the parent again until it successfully selects a parent.

The second problem is that two collided boids, but only one of them died. This is because I use a flag called *IsDied* to indicate if the boid can still collide with other boids, move to the target location, etc. However, as the execution order, the boid which is firstly get executed died first. And the *IsDied* flag will be set as true. When the second boid handle this event, it will regard the first boid as died and will not trigger the death event of itself. To solve this problem, I directly call the death function of the second boid in the first boid's death event. This way the second boid will not check if the first boid is died or not but just directly execute the death function.

Known Bugs

No obvious bug found in this case.