ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2022
D. L. Michels and S. Pirk
(Guest Editors)

Volume 41 (2022), Number 8

Digital Terrarium

Evelyn Ryan

Figure 1: *Replace me with a teaser image of your work!*

Abstract

The Digital Terrarium is a virtual world, crowd simulation, and behavioural simulator designed to provide a template for creating realistic animal herds with individualistic actors. In other words, each actor is meant to represent an animal within a flock. However, each actor has a behaviour engine that gives them individualistic drive. For instance, while flocking, an actor may identify a desired location to travel to. This behaviour leads to emergent behaviour within the flock, as each flock member is effected by their neighbours. Additionally, within the terrarium, there are two "species" of actor: prey and predator. Predators hunt their prey neighbours, while prey flee their predator neighbour. Collision detection is used to simulate a prey being "consumed", thus removing the captured prey from the terrarium.

The foundations of this system are Craig Reynolds's seminal work Flocks, Herds, and Schools: a Distributed Behaviour Model as well as Improv: A System for Scripting Interactive Actors in Virtual World by Ken Perlin and Athomas Goldberg. Simply put, Craig Reynolds steering system for Flocks was used as the basis for actor steering while the probabilistic Behaviour Engine describe in Improv is used to effect an agents behaviour. By default, the majority of the time, actors will be flocking.

CCS Concepts

• ***Computing methodologies*** → *Collision detection;* • ***Hardware*** → *Sensors and actuators; PCB design and layout;*

1. Introduction

Virtual worlds have captured the imagination of humanity long before the earliest digital computers existed. Now, in the modern day, we are able to create the worlds that were once exclusively in the realm of science fiction. Without movement, or a facimile of life, a virtual world feels lifeless and fake. The more populated the world is, the more real it feels to a human interacting with it.

Cinema, video games, and VR are all fields that strive to maintain the suspension of disbelief. Frequently, these artistic mediums rely on scripted behaviour for animal life in their virtual worlds. This method has flaws. For instance, within a video game, a user's suspension of disbelief may break if they notice an animal cycling through a predetermined path. This is a problem the Digital Terrarium seeks to fix.

The Digital Terrarium is a distributed behaviour model designed to give artists the power to create believable and unpredictable flocking animals within their virtual worlds. Individualistic behaviours are determined through chance, and will effect the overall behaviour of the group.

The probabilities for each behaviour, as well as the nature of a flock, are all easily controlled by creators, allowing them to create a variety of flocks and species. Additionally, these species can interact with each other. For instance, there can be predator flocks and prey flocks, with the predators hunting the prey. This can lead to further emergent behaviour within the terrarium that can be customized for any project.

One major limitation to this solution is the reliance on chance, which always has a chance to create unbelievable and immersion

breaking behaviour. By relying on established methods of steering, we can mitigate these risks while also adding parameters to allow artists fine control over the exact probabilities of emergent behaviours.

In this paper, the problem of creating a believable and varied ecological facsimile is laid out, with solutions discussed from my own implementation of the Digital Terrarium.

2. Related Work

In order to understand the role actors can play in virtual worlds, we must first understand what a virtual world is. While there are competing definitions of this term, for the context of this paper Girvan's definition suffices; a virtual world is a digital space that human users can view and/or act upon [Gir18]. Within this framework, the simple terrarium used for the Digital Terrarium is itself a virtual world. However, the self contained system implemented for this paper is applicable to larger virtual worlds with more actors and human participation. For the purpose of clarity, this particular description minimizes human interaction to emphasize the artificial life's implementation.

Similar to virtual worlds, the concept of artificial life has been in the human imagination long before such things were thought to be possible. We also need to establish what this term means for the purpose of this work. Artificial life, unlike artificial intelligence, is a facsimile of the behaviour of the lifeforms that surround us in the natural world [Ste95].

Like fluid simulations, life simulation are far to complex to simulate at efficient speeds for the purpose of interactive worlds or film making, so instead generalizations and simplifications are made so that simulated life appears real enough to the human observer. The purpose is not to perfectly mimic life but to create something life-like enough to be accepted by the observer. In contrast to simulating physical systems like the aforementioned fluid dynamics, artificial life has additional unknowns such as social pressures, freewill, and the question of the existence of the soul [Min86].

In order to simulate the unpredictable nature of free will or individualism in living creatures, the Digital Terrarium uses probability. This gives the actors within the world the appearance of individualistic quirks without the need to prove the existence of freewill. Who can say, perhaps what we call free will is compounding consequences of chance stimuli in our surroundings.

That being said, the Digital Terrarium is not the first artificial life simulation that uses probability to create this facsimile of individualism with a herd. The paper *Modeling Individual Behaviors in Crowd Simulation* implements a very similar method of individual behavior simulation within a artificial life simulation [Bra]. However, this model uses a human crowd simulation rather than an animalistic heard.

Additionally, the paper *Scalable Behaviors for Crowd Simulation* by Mankyu Sung also uses probability to create individualistic behaviour within human crowds. The behaviours within this paper, however, are scalable. This means the individual behaviour can scale to the larger crowd, which is also similar to the Digital Terrarium [Sun04].

As discussed later, the behaviour engines of the actors within the Digital Terrarium uses a behaviour tree prior to probability process. For instance, if an actor is currently seeking a goal, the range of actions they can take after they begin the goal seeking behaviour will be different than when engaged in another behaviour [LBC10]. Behaviour trees have been explored for many process, from video game enemies to robot controls [Jon18]. Behaviour trees can even be adaptive, and change as a simulation runs. That particular concept is beyond the scope of this paper though.

As virtual worlds and artificial life continue to be important parts of simulation and entertainment, it is important to consider what makes a virtual world believable. Virtual actors play an important role in this believable. Artificial human life is important. We all spend great lengths of time within crowds of other people. Likewise, artificial animal life is important too. While we may take the presence of animal life for granted in our day to day lives, without them the world would be a much more barren place. While simulating the quirks and habits of living creatures may border on the impossible, creating a believable simulation of living creatures can add a great deal of texture and immersion to a virtual world.

3. Overview

The Digital Terrarium integrates numerous well established animation principals. The goal of the terrarium is to run a large amount of digital actors within a digital world using computationally inexpensive strategies. As such, the methods used within the system are both well know and have low computational cost.

The underlying system of the Digital Terrarium is a particle system. Each of the actors are a particle, simply rendered as colored spheres. Since the emphasis of this particular project is a low cost system that can be applied to many scenarios, the simplest rendering was used. However, because the system is easily customized, the sphere's can be easily replaced with fully animated models.

The problem can be broken into two major sections: the high level behaviour determination and the behaviour itself. As previously mentioned, the behaviour is determined using a behaviour tree as well as a behaviour engine. This approach is used so that an actor can recognize what action it is currently doing so it may select a random action from a list of transition actions or continue its current action. In other words, each action has it's own behaviour engine within its corresponding branch of the behaviour tree.

3.1. Particle System

Particle systems utilize simple rendering to allow for large systems of particles to interact with each other. That is why a particle system is used for this project. The particle system itself has little utility on it's own. The main purpose of the particle system is to store the information each actor needs in order to act within each time step in the simulation. Therefore, while the particle system itself has little utility, it must be able to be updated from the time simulator.

When creating a simple particle simulator like what is used in the Digital Terrarium, the number of total particles can be quite high due to the low cost. For the implementation of the Digital

Terrarium for this project, there can be a total of 1000 particles in play at a time.

The Digital Terrarium has two sets particles: predator particles and prey particles. It is important to be able to differentiate between the two types of particles in someway because predator and prey actors behave in different ways in both flocking and individual behaviour.

3.2. Time Simulator

However, what good is a particle if it doesn't move? The time simulator remedies this. The time simulator acts on every time step of the simulation, allowing particles to move. This is all well and good, but how do we tell particles to move?

The time simulator extracts three pieces of information from the particle it is acting upon: vx, vy, and vz. These correspond to the x, y, and z values of the previous step's velocity. The particle can then be moved using the forward Euler method for solving Ordinary Differential Equations, as seen below:

$$x_p(t+\Delta t) = x_p(t) + v_p(t)\Delta t \tag{1}$$

Where $x_p(t+\Delta t)$ is the current position $x_p(t)$ is the previous position of the particle, $v_p(t)$ is the velocity of the particle, and Δt is the time step.

This ignores the fact that the simulation must update the velocity in every step based on either flocking or individual behaviour. This is where the behaviour tree comes in. The Behaviour Tree is a function within the time simulator that is called in every time step of the simulator.

3.2.1. Behaviour Tree

The Digital Terrarium has two behaviour trees: a prey behaviour tree and a predator behaviour tree. Both of the trees function in the same way. When a particle is put into the behaviour tree, a piece of information is extracted from the particle. The information pertains to the particle's current activity. For instance, if the particle is currently flocking, the information may equal a string "flock" or perhaps a number that corresponds to that activity. The behaviour tree parses the particle to a certain branch based on this number. Each behaviour tree has a number of branches equal to the total number of activities an actor can take.

Algorithm 1 Prey Behaviour Tree

```
state ← particleAction
if state == "flocking" then
    flockingBehaviourEngine()
else if state == "dancing" then
    dancingBehaviourEngine()
else if state == "goalSeeking" then
    goalSeekingBehaviourEngine()
else if state == "variation" then
    variationBehaviourEngine
end if
```

The next natural question to ask is how do the behaviour engines function?

3.2.2. Behaviour Engines

Each of the behaviour engines work more or less the same. Firstly, a random number generator is used to generate a value between 0 and 101. The value is filter through a series of if statements. The if statements represent the bounds of the probability for each behaviour [PA96].

Algorithm 2 Prey Flocking Behaviour Engines

```
state ← random(1, 101)
if state ≥ 1 OR state ≤ 70 then
    flocking()
else if state ≥ 71 OR state ≤ 80 then
    dancing()
else if state ≥ 81 OR state ≤ 90 then
    goalSeeking()
else if state ≥ 91 OR state ≤ 100 then
    variation()
end if
```

For the purpose of initialization, each actor begins in the "flocking" state. As such, flocking acts as the nexus state which particles will spend most of their time. The flocking behaviour engine can lead to all other behaviour. However, within the other behaviours there are only two options: continue the current behaviour or go back to flocking with the current behaviour being the most likely outcome.

This reveals the true purpose of having multiple behaviour engines. Early in development, predator and prey actors only had one behaviour engine each. Since flocking is the most weighted outcome, actors will occasional switch to a different behaviour, then almost immediately switch either to flocking or another behaviour. In order to have all the behaviour's pronounced, multiple behaviour engines were used in conjunction with behaviour trees.

Predator and prey actor necessarily behave differently. While their behaviour trees are largely the same, instead of goal seeking, predators hunt. While prey select a point in the terrarium to move towards, predators select a prey creature to move towards. Additionally, predator actors are solitary, a trait represented within their flocking behaviour that will detailed later.

Thus, the high level overview has been completed. We now understand the decision process each actor must take in order to act. However the question remains, how do the actors move in a low level view? For this we will discuss steering, the primary method of movement used in the Digital Terrarium.

3.3. Low level behaviour

The low level work in this particular project utilizes steering for the purpose of crowd simulation. In each time step, the last step's direction of each particle is added to the behaviour direction of the step's behaviour outcome. The speed of each particle remains constant.

In the following subsections, the steering information of the various behaviours is specified.

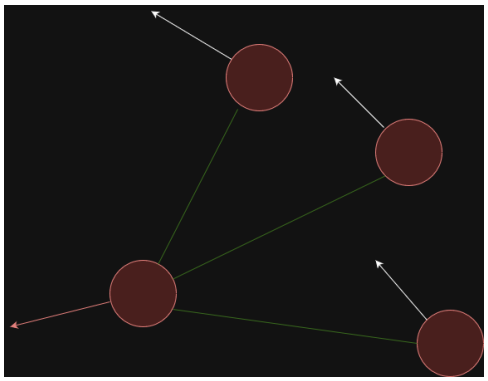

Figure 2: *The principle of separation for flocking steering*

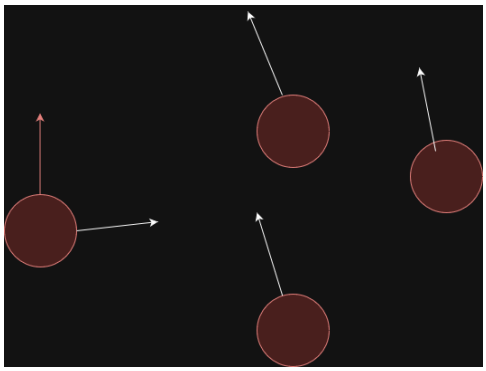

Figure 3: *The principle of alignment for flocking steering*

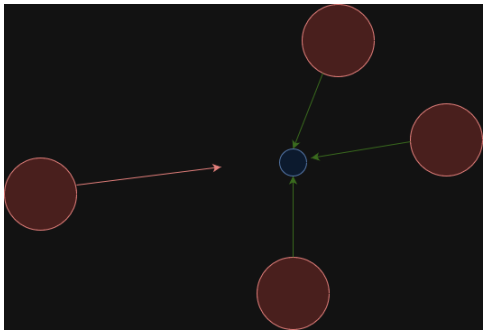

Figure 4: *The principle of cohesion for flocking steering*

3.3.1. Flocking Steering

Flocking steering can be broken down into three distinct parts: separation, alignment, and cohesion. The mechanics of these methods are simple, once calculated, the direction vectors can be added to the original direction vector of the actor.

An important concept within this steering is that of a neighbourhood. Generally, if the three rules are implemented without a neighbourhood in place, actors from across the terrarium will try to flock with one another with no regard for the distance between them. This is easily fixed by forming neighbourhoods. Actors will only flock with other actors in their neighbourhood. That is, actors learn what other actors are within a certain distance from themselves, and only flocks with those actors.

Separation is designed such that the flocking actors do not get too close to one another. If too crowded, actors will begin to phase in and out of one another unless a physical system is implemented. Separation removes the need for a physical system in this context.

As shown in Figure 2, separation functions by finding the current position of the actor's neighbours and creating vectors from the actor to its neighbours.The separation vector is the average of these relational vectors multiplied by negative 1 [Rey98].

Within predator populations, the separation vector will have a flat constant of 2 scaling it. Since all of the vectors within flocking should be normalized, this weights separation twice as strong as the other rules of flocking. Thus the solitary lifestyle of birds of prey are simulated.

Alignment is designed such that all the actors within the flock travel in a similar direction. This is also easily calculated as the average direction of the actor's neighbours. An illustration of this principle is seen in Figure 3.

Also important is the principle of cohesion. In a way, this principle is the antithesis to separation and indeed in cases of two actors, separation and cohesion are likely to cancel one another out. However, this will change in cases of larger number of particles due to how cohesion is calculated.

The cohesion vector is calculated by subtracting the coordinates of the center of mass of the actor's neighbours by the coordinates of the actor. Instead of taking the average of the vectors from the actor to it's neighbours like in separation, this method directs itself towards the center of mass of the neighbourhood. In Figure 4, the blue dot represents the center of mass. Thus, in cases of neighbourhoods of size greater than 2, separation and cohesion will not cancel each other out [Rey98].

The coordinates of the center of mass are calculated with the following equation:

$$R = \frac{\Sigma_{i=1}^{n} m_i * r_i}{\Sigma_{i=1}^{n} m_i} \tag{2}$$

Where r_i and m_i are the coordinates and mass of particle i.

3.3.2. Dancing

After flocking, the most complex behaviour is dancing. Dancing is an interaction between two individuals within a flock. The two dancers begin to circle around each other's center of mass and slowly close the distance between each other. This can be thought of as analogous to mating behaviour in birds.

In order for both actors to participate, there needs to be a information handshake. First, an actor must check if they are being invited to dance. If so, then they begin the dance with the invitee. Otherwise, then select their nearest neighbour and save the invited actors ID. Then, during the invited actor's next step. the invited ac-

tor check's it's neighbours for its own ID. If found, it will begin to dance with the invitee.

3.3.3. Goal Seeking

The next behaviour is relatively simple. Goal seeking generates a random point within the bounds of the digital terrarium. The direction of the actor then changes to be towards the goal. Once the actor reaches the goal, a new goal is selected. The process continues until the behaviour engine selects a new behaviour for the actor.

3.3.4. Variation

Variation is a simple behaviour that slightly adjusts the direction of an actor in the x,y, or z direction by a small amount. First, a random number from 1 to 3 is generated. These values correspond to x, y, and z respectively.

Then, a number between -3 and 3 is generated and replaces the current value in the x,y, or z place of the actor's direction vector. The direction should then be normalized and the actor's speed should be scalar multiplied with the now normalized direction in order to keep the speed constant.

3.3.5. Hunting

Hunting is similar to goal seeking with in the prey behaviour engines. The main difference is with predators, instead of choosing a random point within the terrarium, they instead choose a random prey actor.

Prey actor's must therefore check every step if a predator is colliding with them. When predator and prey collide, this marks the prey being captured and thus consumed. The prey's data is then overwritten with flags that mark the actor as "dead". Thus, the prey can either disappear, or remain motionless in the terrarium to be marked as dead.

The collision detection itself is quite simple. Each actor is a sphere with a predefined radius. Thus, if the distance between the predator and prey is equal to the size of the prey's radius added to the predator's radius, the particles have collided and the prey is caught.

3.3.6. Overrides

Regardless of the current behaviour, actors must still do certain actions. For instance, if there is a surface that must be avoided, actors must still avoid them regardless of their behaviour. This is simple enough to simulate.

As seen in Figure 5, the line normal of square or flat surfaces is trivial to find. In the figure, the purple line represents the line normal. All that must be done in order for the actor, represented by the red circle, to avoid the white line is to add the purple line to the green line. The green line represents the actor's direction vector.

Additionally, prey actors must always be avoiding predator actors. This is also simple. In order for the prey animals to not constantly huddled in a corner trying to flee predators, we must also use neighbourhoods in relation to predators. If a predator is in a prey's neighbourhood, a vector from the prey to the predator must be created. Then, this vector can be multiplied by -1 and normalized. Then the vector can be added to the current direction of prey. The prey will then try to move out of the predator's neighbourhood.

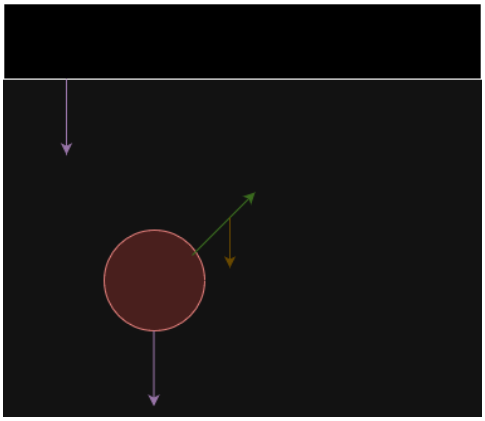

Figure 5: *Object Avoidance*

3.4. Implementation

For my own implementation, C++ was used. A template for CSC473 created by Professor Brandon Haworth provided the base code including rending and lighting. In the next session, the runtime speed, computer specs, ease of implementation, and places for the project to go next will be explored.

4. Evaluation

The simulation runs pretty well on my machine. Steering rarely requires expensive operations, the cost comes from the many operations that are required within the systems various behaviour operations. excess cost can also come prom outputs, and flawed loops. In order to avoid excessively large loops, the number of initialized actors is accessed within the time simulation.

The majority of the cost comes from the fact that every step has a nested loop. The worst case scenario is a flocking prey actor. Each loop iterates through every particles. The number of particles can be represented as N. Then, a naive approach was taken in assessing neighbour's in flocking, so all N-1 other prey must then be checked. Therefore we have now iterated N(N-1) times. Then, in the prey override, all N predator's must be checked for proximity. This takes another N iterations. Now, we have N(N-1 + N) = N(2N-1) = $2N^2 - 1$. Therefore, the big O cost is O(N^2). This runs well on my computer, but it could be worse on machines with less GPU or CPU power.

4.1. System Specifications

The computer the project was implemented on is a high end gaming PC, meaning it's above average specification may distort the effectiveness of the results. The PC has a 11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz processor, 16 GB of ram (15.7 usuable), the Intel Iris Xe Graphics GPU, the NVidia GeForce RTX 3060 laptop

GPU, a 500 GB SSD, and is running Windows 11 64 bit. The laptop is less than two years old (2023) and is capable of running up to 600 particle at extremely high FPS (upwards of 1000). The next section includes a detailed breakdown of the FPS performance in relation to particle numbers.

4.2. Performance Breakdown

While the performance of small scale systems of particle actors is quite strong, the performance as the number of particles increases begins to decrease sharply. The exponential cost of the algorithm is too expensive, even on high end computers.

As seen in Figure 6, the FPS performance of the program remains above 2000 FPS up until the 200 particle mark, then slowly decreases until 600 particles are reached, in which the FPS sharply decreases to single digit numbers for the particle counts upwards of 800. The final drop off is quite severe and makes the program practically unusable in such high numbers.

However, the results surpass the goal of the project, which was to run the program with 50 particles at an FPS greater than 60. This is a marked achievement in my eyes and supports the conclusion that the system would be viable for implementation in real time rendering projects such as game development. That being said, there is much room for improvement that could ensure the system works for a larger variety of machines.

Additionally, it should be noted that a variety of tests were done besides the parameterized particle number. Additionally, behaviours were tested individually, and while most of the behaviours had similar performances, behaviours that didn't require the nested for loop ran much faster, much to my expectation.

4.3. Room for Improvement

The biggest boon to performance would be to replace the nested for loop with a more effective search algorithm. The naive approach was taken for the implementation of flocking and hunting. This means that every single other particle is checked every time. There are a number of ways to reduce this runtime cost for each actor.

My first idea was to store particles in a graph format instead of an array. Then instead of checking every particle, the actor could only check the particles with an edge connecting to themselves. Then at the end of each step, an algorithm would be called that creates edges between nodes within each other's neighbourhoods. This could potentially bring the cost down to O(NlogN) depending on the algorithm used for the second part.

Another idea would be to store the particles in a hash map using their coordinates as the keys. Then, each actor only checks the hash entries associated with the coordinates within their neighbourhood. The actor would then adjust it's hash position after each movement.

Another room for improvement would be the ability to add obstacles to the terrarium. This was a part of the original plan of the project but it had to be dropped due to time constraints. More ideas for future paths the project could take are included in the conclusion.

4.4. Flaws in the algorithm

While the algorithm performs decently at relatively high particle numbers, there are some overarching problems with the simulation itself which emerged as a direct result of the implementation or assumptions that were made along the way.

The largest problem so far was an issue around the predator's hunt behaviour. When predator's hunt prey, the prey flees. Predators usually catch prey, however, if they do not, prey can "escape" by clinging to the bounds of the terrarium. The predators cannot penetrate as far into the boundary as prey can. The problem was left in because it allows the prey a method of escape.

Additionally, there is a problem with the use of the c++ call rand(). While it is the most accessible random number generator in the c++ library, it is not truly random and will often generate the next rand() call by iterating the previous by one. This is a significant flaw in the algorithm because it means a the length a behaviour lasts isn't variable.

The final flaw was another matter of time. Due to a tight course load in my final university term, I didn't have time to implement all the behaviours I wanted to. I finished what I could and made sure to get the essential predator/prey dynamic functional but some of the less important behaviours didn't get a chance to see the light of day.

4.5. Final Evaluation

When compared to the initial project proposal, a great deal of content had to be changed. As previously mentioned, my greatest road block was time. Time was extremely limited, especially time to complete this project. As such, the original concept was greatly reduced in scale in order to be completed in a manageable time. Originally, the concept included twice the number of species, including "land" species that had limited movement in the z-axis.

This ended up being far too ambitious. Many of the original behaviour had to be cut as well. What I was left with was the bare minimum I wanted to complete for this project. Even still, I managed to do it and the performance is genuinely pretty decent. I blew my original expectations for the number of particles I could run out of the water.

In the end, I feel like the project is a mixed bag. On one hand, I didn't get to complete everything I had hoped to. On the other hand, the runtime of what I did implement was much better than I expected and the result is pretty decent looking as well.

5. Conclusion

The Digital Terrarium is a animal crowd simulator meant to simulate the individual quirks and predator/prey relationship of living creatures. It utilizes behaviour trees and behaviour engines to create a variety of behaviours that are meant to correlate to the behaviour of living creatures. Some of the key behaviours include flocking, hunting, and object avoidance. The behaviour engines use probability to switch between behaviours, and the behaviour tree recognizes each actors current action to change which behaviour engine is being used. The system works pretty well with some key insights

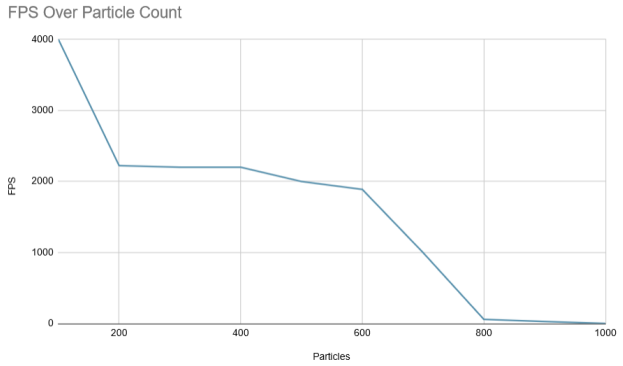

Figure 6: *Performance Evaluation Based on Parameterized Particle Numbers*

and findings that effect the quality of the simulations appearance and believability.

5.1. Key Findings

One of the key findings is the uncanny reaction time of agents in the simulation. Agents react to outside forces with startling speed. This is for a number of reasons. Firstly, actors don't have sight. Instead each actor is practically omniscient. They see every other actor in the terrarium but only react to actors within their neighbourhood. This causes one particular problem in the simulation. If predator and prey are moving the same speed, the prey's reaction speed is such that the predator can never catch the prey.

No matter how fast the predator reacts to the prey, no matter how many predators there are, if the prey starts ahead, the predators cannot catch them. There is one simple fix without adding a method for sight, prey drive. Prey drive is a scientific concept. When a predator has prey in their sights, adrenaline increases their physical capabilities such that their chances of catching their prey is greatly improved.

The simulated prey drive is similar. When a predator has a prey actor in sight, the speed of the predator actor increases such that its speed is greater than the speed of the prey. While this calculation could be challenging to make the prey drive functional, in my implementation all the speeds are consistent. Therefore, I just used a larger speed when calculating the hunt vector.

Another key finding is that the override method of having object and predator avoidance has a key flaw. The override vectors weigh heavily over the flocking vectors. This is because of the way vectors get normalized.

Each of the avoidance, cohesion, and separation vectors are normalized and added together. Then, their summed vector is normalized. Therefore, each of the vectors are a third of the weight of the total vector. The override vectors however, are one half of their vectors. As a result, overrides completely overpower flocking behaviour. This is somewhat acceptable. As the name suggests, override is supposed to be an actors last minute reaction, they are necessarily urgent. However, the result is somewhat unnatural as prey bob and weave around predators with a level of fine aerial control only seen in insects. Perhaps this particular simulation is a better simulation of insect flocks than bird flocks.

5.2. Challenges

The biggest challenge in this project was understanding the high level decision making of each actor. My initial proposal didn't include behaviour trees due to the fact that I didn't consider the fact that if a behaviour has a low probability of occurring, it will likely only be in that behaviour for one step before returning to flocking. Therefore, behaviour trees were added to allow actors to stay in a single action for a longer time.

This in of itself had some issues. In order to track the current behaviour of actors, I added another place to the particle arrays used to track actor information. The array is already 7 doubles long, adding another created strain on the system and made tracking each index even more convoluted. On top of that, it caused some issues with the way I initially had the particle system and time simulator interacting.

Initially, I passed the particle system to the time simulator as a base class, meaning I couldn't access the public functions and data structures of the particle system. I remedied this by instead passing the time simulator a class ref pointing to the particle system. This way I could directly access the functions and data structures in the particle system within the time simulator.

Another difficult aspect was debugging. It was very difficult to tell what was going wrong with a behaviour just by observing it. It took a lot of detailed diagnostics in order to find and diagnose the root cause of malfunctioning behaviours.

5.3. Limitations

Perhaps the biggest limitation on this project is the absence of a good random number generator in c++. The base random number generator has some major flaws that make it almost unusable. While there are other random number generators for C++, most of them would have required an additional library, something I didn't have time to install when I discovered this truth about the rand() call.

There is a way to make the random call slightly more effective. By using srand(), you can seed the random call. This effects the starting point of the call meaning it only effects the starting point. Subsequent rand calls will still simply iterate instead of create even an imitation of random numbers.

Another massive limitation was time. I had very little time to work on this project due to the very heavy workload of the term. Keeping my head above water was a startling challenge and it was definitely my most challenging term in my scholastic career. That being said, I am quite proud of the work I accomplished.

Another major limitation was the terrarium size. For most of the testing, the terrarium was limited to a 22x22x22 cube. As a result, certain behavioural issues that would require more space to diagnose would be missed.

5.4. Future Work

There is a great deal I would like to expand upon in this work for my own interest. I enjoy game development and I find this work very interest for developing involved and immersive NPC animals. As such, I wish to expand upon the behaviours listed to include interactive elements. Some actors may be friendly with the player, while others could hunt players.

Additionally, I would like to add heuristic object avoidance in this project. I want to create a system where actors can avoid objects that are near them while seeking a goal behind said object. This would also be a very useful skill to have for game development as it would allow me to create more dangerous enemies and better companion NPCs for players to engage with.

Additionally, I would like to expand the project to include the content I ended up having to cut from my original pitch. I would like to expand upon this with different kinds of species that have different methods of flocking. Two of the main ones I considered where bunnies and wolves. I especially want to implement wolves because I think it would be very cool to create artificial life that hunts with the pack dynamics of wolves. Bunnies are also something I want to implement because they are one of my favorite animals.

References

[Bra] BRAUN A.: Modeling individual behaviors in crowd simulation. In *Proceedings 11th IEEE International Workshop on Program Comprehension*, IEEE.

[Gir18] GIRVAN C.: What is a virtual world? definition and classification - educational technology research and development. In *SpringerLink* (2018), IEEE, pp. 487–501.

[Jon18] JONES S.: Evolving behaviour trees for swarm robotics. In *Distributed Autonomous Robotic Systems* (2018), IEEE, pp. 487–501.

[LBC10] LIM C.-U., BAUMGARTEN R., COLTON S.: Evolving behaviour trees for the commercial game defcon. In *Applications of Evolutionary Computation: EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I* (2010), Springer, pp. 100–110.

[Min86] MINSKY M.: Society of mind. In *MIT Press* (1986), IEEE.

[PA96] PERLIN K., ATHOMAS G.: Improv: a system for scripting interactive actors in virtual worlds. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (1996), IEEE, pp. 205–216.

[Rey98] REYNOLDS C. W.: Flocks, herds, and schools. In *Seminal Graphics* (1998), IEEE, pp. 273–282.

[Ste95] STEELS L.: The artificial life roots of artificial intelligence. In *Artificial Life* (1995), IEEE.

[Sun04] SUNG M.: Scalable behaviors for crowd simulation. In *Computer Graphics Forum* (2004), IEEE, pp. 519–528.