Emergent flocking boid-based crowd behavior through generalization of the system rules in a 3D Reinforcement learning environment with predator satiation and foraging

Author:
Georgi Ivanov

Supervisor:
George Palamas

AALBORG UNIVERSITY
DENMARK

Copenhagen
June 2020

Aalborg University Copenhagen

Semester: 10^{th} semester, Medialogy (MSc)

Title: Emergent flocking boid-based crowd behavior through generalization of the system rules in a 3D Reinforcement learning environment with predator satiation and foraging

Project period: February 1^{st}, 2020 - June 4^{th}, 2020

Semester theme: Media innovation

Supervisor: George Palamas

Group members:
Georgi Nikolaev Ivanov

Finished: June 4^{th}, 2020
Pages: 65

Aalborg Univeristy Copenhagen.
A.C. Meyers Vænge 15, 2450.
Copenhagen SV, Denmark.
Secretary: Lisbeth Nykjær
Mail: lny@create.aau.dk
Phone: 99 40 24 70

Abstract

The purpose of this project was to create an emergent flocking boid-based crowd behavior, through generalization of the system rules, predatory threat and foraging for the virtual crowd. Extensive research was conducted on the topics of flocking boids, prey and predator behaviors in nature and emergent behaviors and self-organizing systems. 3D Reinforcement learning environment was implemented using Unity ML-agents. Finally, evaluation was conducted by comparing two different ML crowd implementations with a flocking boid control environment. The results showed a successful evolving of an emergent crowd behavior from one of the models, using a generalized set of system rules. Despite the suffered hardware limitations, the exhibited results satisfied the problem criteria, as well as providing interesting insights and findings regarding emergent behaviors.

Contents

List of Figures

1 Introduction

By definition, crowd simulating is the process of recreating the dynamics of a large group of entities, unified by a common purpose, environmental conditions or other external stimulus [40]. Crowd simulations are an all important part of video games and cinematography, but they are also used in more serious scenarios, such as crisis training, evacuations or general urban and architecture planning. Based on their purpose and application, there are different types of crowd simulations, each highlighting different key behaviors. For example, while in an evacuation simulation it might be beneficial to navigate the crowd using an as realistic as possible pathfinding algorithm, a virtual crowd meant to spark life on the streets of a video game city might benefit more from character and behavioral diversity and low run-time computational cost [40].

One of the earliest attempts at engineering an emergent group behavior is that of Craig Reynolds with his famous Flocking boids algorithm [34]. The reason why this work is so significant, is because it marks the beginning and becomes a basis of worldwide efforts in competing to create a robust and lightweight group behavior model for virtual crowds of large numbers. The release of the continued work of Reynolds [35] on his original flocking behavior, where he shows the applications of transferring the flock behavior approach to crowds instead of flying creatures, has further served as inspiration of many future projects aimed at achieving crowd behavior based on different emergent approaches. The reason why the emergent approach is so popular solution to this problem lies in the specifics of virtual crowd dynamics. To be exact, there are three main methods for designing virtual crowds [33]:

- **Flow-based approach**, where the emphasis is on the crowd as one singular entity instead on the individual agent [9].
- **Entity-based approach**, which is the opposite and focuses on the separate agents of the crowd. In this scenario all agents are bound by some overall governing rules that serve as the catalyst to some desired emergent behavior, making simulations like this useful for studying specific desired aspects of group dynamics. Basic flocking belongs to that category as well.
- **Agent-based approach** is the most advanced of the three methods, as it does not rely only on overall governing rules for the emergence of a certain behavior. Instead, each member of the crowd is an autonomous intelligent agent, that has the capacity for a much broader set of possible decisions, than the agents from the entity-based approach. This solution usually involves some type of ML model, used for navigating the agents. Furthermore, this method combines the benefits of the entity-based approach, with the capacity for much more complex and diverse resulting behaviours, than what a human developer can produce through a set of navigation rules [40].

As mentioned earlier, one of the biggest problems of virtual crowds is the simulation of groups of virtual entities with sufficient behavioral fidelity at a low computational cost [40]. A great example for this is the creation of the famous 'Lord of the Rings' movie trilogy [17], where in each of the three movies crowds of extremely large numbers of over 50 000 entities at a time had to be handled (see figure 1). What is more, due to the nature of the scenes where the crowds were utilized and the aim for high attention to every detail by the producers, it was a requirement that every single crowd member behaves naturally and in a unique way that does not give a feeling for uniformity and staleness of the virtual masses [17]. To achieve that, the producers used the 'MASSIVE' (Multiple Agent Simulation System In a Virtual Environment) crowd generation algorithm [8]. 'MASSIVE' utilizes the 'Miarmy' [19] human logic engine for Autodesk's Maya 3D modeling tool [27], implementing an AI capable of choosing from a large set of animation sequences based on the environmental conditions each entity is currently in [17]. This allowed for a high fidelity artificial crowd to be created for the movies, without the use of traditional techniques for handling crowd computation costs, that might reduce the quality of the visual experience. Some of these techniques are 3D culling, which discards objects from the virtual scene that are considered unimportant at the current moment and decreasing LOT (Level Of Detail), where the further away a crowd member or object is from the camera, the less attention to detail is invested into rendering it visually and behaviorally.

Figure 1: Virtual crowd in 'Lord of the Rings' [17].

As entity and agent-based systems have been proven to be great solutions to the virtual crowds problem, flocking boids find a large use in this field. One example for this is the super-massive crowd simulation by Passsos et. al. [32], who achieved a flocking boid simulation of over 1 million agents. Furthermore, Hartman et. al. [16] propose the usage of the flocking boids algorithm as a lightweight solution to simulating virtual crowds in modern games and other virtual simulations. Similar idea is realised by Dewi et. al. [10], who use flocking boid behavior for the simulation of a realistic crowd, in order to investigate certain crowd dynamics aspects. In addition,

Chiang et. al. [7], show that the emergent behavior of a flocking boid represents a viable solution for simulating a virtual crowd.

There are numerous both behavioral and visual high fidelity crowd simulation solutions nowadays, which make use of the ever-growing computational capacity of modern computers. However, the field of emergent behaviors also gives purpose to some much more lightweight solutions such as flocking boid-based crowd simulations, that combine technical ingenuity with a very respectable level of behavioral fidelity and large number of applications.

2 Analysis

2.1 State of the art

2.1.1 Flocking behaviour in simple ecosystems as a result of artificial evolution

One interesting work in the field of emergent behaviors and more specifically flocking, is the project by Kwasnicka et. al. [23], designed to observe the emergent behaviors in a open-ended evolutionary system of creatures, evolved by a genetic algorithm. Genetic algorithms (GA) are a subset of the field of evolutionary approaches, that are characterised with their operational simplicity (compared to a neural network approach) and effectiveness in the field of artificial evolutionary ecosystems [23].

The particular environment setup designed by Kwasnicka et. al. involves a simplistic two dimensional environment, combining not only the traditional for this type of setups large number of agents and predator, but also foraging capabilities for the agents as well as habitat artefacts, such as plants (serving as food) and life system for the occupants of the environment, serving as an additional tool for control of the artificial population (see figure 2).

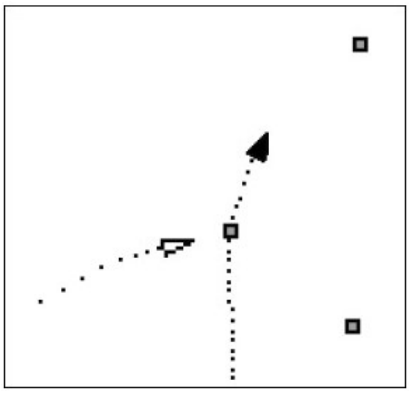


Figure 2: Two dimensional environment created by Kwasnicka et. al.[23]. The predator (black triangle) in this particular situation has decided to ignore the prey (white triangle) and rather continue to the next piece of food (rectangle), as it has learned that usually larger numbers of prey would gather around unpatrolled food.

Due to the versatility of the GA, the environment includes simultaneous evolution of both prey and predator, resulting in much higher number of possible outcomes. What is more interesting about this work however, is that it implements an *open-ended* evolution. Open-ended evolutionary systems, also known as *autoconstructive evolution* are unique in that they allow for the evolved subjects to choose in the time of production of their offspring, thus mimicking this aspect of natural life [23].

The findings of the paper constitute mixed opinions about the overall benefit of introducing such a high number of diversity of the elements of the environment, as it is speculated to increase the produced entropy to a level where no classifiable behaviors can be observed. Furthermore, the extensive research conducted by the authors on the topic of what initial rules and conditions facilitate flocking behavior in such artificial ecosystems, combined with the findings of their own experiment, lead to the conclusion that there is still no concrete explanation for the reason of the emergence of this behavior and any further research in the field will be highly beneficial for the creation of a universal theory [23].

2.1.2 Investigating and Modeling the Emergent Flocking Behaviour of Sheep Under Threat with Fear Contagion

Another interesting work in the field of emergent behaviors is the paper by Chang et. al. [6], aimed at investigating the behavior of agents that include the implementation of the fear-contagion system (see figure 3).

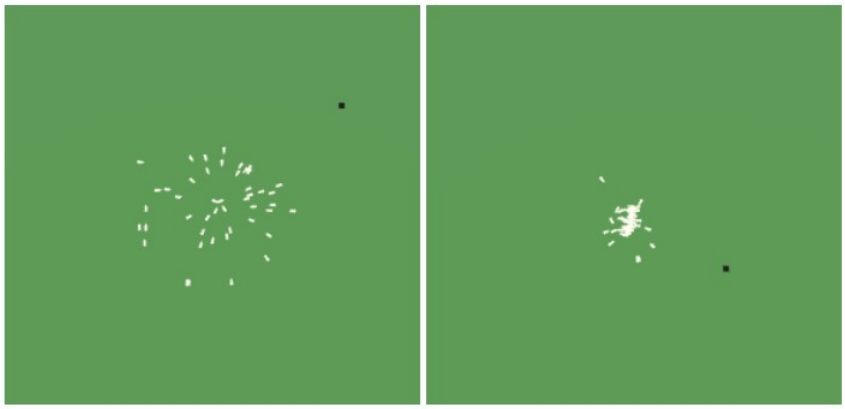


Figure 3: Environment design by Chang et. al.[6]. The image on the left shows the reaction of the heard (white triangles) without the implementation of fear-contagion right after the approach of the dog (black rectangle) has commenced. The image on the right depicts the same situation, but with the fear-contagion implementation.

The authors of the paper draw inspiration by the work of Delgado-Mata, Carlos, et al., that did extensive further development of the original flocking boid algorithm, subsequently after the introduction of the system by Craig Reynolds [34]. More specifically, Chang et. al. have used Unity engine [13], in order to create an evolutionary environment, the goal of which is to achieve control of a heard of sheep towards a desired goal, using a single control method, represented by a shepherding dog. The goal of the conducted experiment was to test the results of including a fear-contagion factor in the behavior of the heard, against a control environment with a generic flocking boid implementation. The fear-contagion implementation itself includes an additional force to the reaction of the sheep to the actions of the dog, based on the fear levels of the individual neighbours, as well as the distance to the dog.

The results from the experiment show that grouping behavior is achievable through the proposed model, but in a far less noticeable scale than what would be considered satisfactory (matching or exceeding the results from the control environment). Additionally, paper concludes that further investigation is needed, in order to formulate the exact causality relation between the rules of the environment and the produced outcome. Considering the conclusion drawn by the authors, it can be further speculated that the insufficiency in the results was produced by the reliance on an overly complicated reward system (such that includes indirect dependence on the fear factor). This conclusion would be an important consideration in the design of the current project.

2.1.3 Emergent Escape-based Flocking Behavior using Multi-Agent Reinforcement Learning

In their paper Hahn, Carsten, et al.[14] show how emergent flocking behavior can be achieved using an escape-based approach and a simple fitness function for the trained agents, all combined in their proposed 'SELFish' approach (Swarm Emergent Learning Fish). Similarly to this work, Hahn, Carsten, et al. draw their inspiration from Reynolds's [34] famous flocking boids implementation.

The goal of this paper is to show that a flocking boid behavior can be achieved in a simple reinforcement learning environment, where the only motivation for the agents is to stay alive as long as possible. The key element of the proposed implementation is the escape-based approach, where along the classic flock of cloned agents, a predator is introduced to the environment. To be more specific, the environment consists of a two dimensional, rectangular action field, where the agents and the predator are represented by circles with same radius for agents and predator (see figure 4).

Figure 4: Two dimensional environment created by Hahn, Carsten, et al.[14], with the agents in green and the singular predator in orange. The lines in the bodies of both agents and predator serve to show their current movement direction.

The environment sets no limitations for its occupants, as the rectangular field is implemented in a way that exiting on one side, immediately leads to entering back from the opposing side, creating the effect of a infinitely looping field. The action space of the agents consists from only two actions, that is whether to change their orientation by some degrees clockwise or counterclockwise. Using such a simple action vector is possible due to the fact that the agents never stop moving, taking one step per frame in the direction they are facing. Furthermore, this is important as

it significantly alleviates the processing load of the training algorithm, matching the overall simplistic theme of the environment and allowing the focus to be entirely on the exhibited behavior and not on visual fidelity. This is important because even with such computationally and graphically simplistic environment in its basic setup, these numbers increase drastically once training is initiated. To emphasize further, due to the fact that machine learning algorithms generally benefit proportionally to the size of the learning experiences, during training time the time step of the environment is usually significantly accelerated. This, combined with the fact that every frame multiple agent instances are maintained, vastly increases the overall computational load of the environment.

The observation space of the agents follows the same theme, consisting of each agent being aware of its own movement, the predator's and n nearest neighbours. While this also reduces the computation load of the algorithm, it is also inspired by the way flock members observe their surrounding environment in nature.

The predator shares the same physical characteristics as the rest of the agents and its purpose is to serve as a catalyst for the emerging of the flocking behavior. The main difference between the agents and the predator is in that while the agents are controlled by a neural network, the predator behavior is hard coded with a very specific idea in mind, that ties together all elements of the environment to the overall purpose of the project. More specifically, the predator has a predefined vision radius which determines which flock member will be targeted. If there are no flock members within the radius, the predator will go after the closest one from the entire population. However, if more than one flock members enter the radius, the predator has the chance to switch targets, by picking a random agent from the current number in his vicinity. This simple rule adds the key aspect in the the behavior of the predator that allows agents that have been targeted to escape, by rejoining the rest of the flock and potentially 'confusing' the predator. As the agents' fitness function rewards them for simply staying alive longer, the above described rule serves as an implicit reward mechanism, designed to induce swarming of the agents with the goal of prolonging their survival.

Hahn, Carsten, et al. evaluated the results of the created model, by doing an in-between comparison among several simulation setups. Particularly, the two main reinforcement learning setups were based on Deep Q Newtwork (DQN) and Deep Deterministic Policy Gradient (DDPG) algorithms. The results of those were compared with the original flocking boid implementation and one even simpler environment, where the main rule would be that upon facing with the predator, the agents will be directly repelled to the opposite direction. The results from these environments were compared in terms of similarity to the generic flocking boids implementation. For example, taking the DQN implementation, the average angular deviation of the agents in that environment was compared with that of the original boid. Other metrics such as density of the agents, average distance between them and visual inspection were also used.

By and large, the findings of the paper allow for the conclusion that the two reinforcement learning approaches lead to a significantly similar behavior to the original boids implementation. More importantly, the authors suggest the fact that the agents still evolve towards a grouping behavior, rather than selfish running away from the predator is a state of Nash equilibrium for the environment. To put this in other words, considering the used fitness function and the general rules of that environment, it can be inferred what is the optimal behavior for maximising the reward and that is simply turning away from the predator and running away in the opposite direction every time he is encountered. Despite of that however, as mentioned already the agents would still evolve towards the swarm-like formations. In that case, the evolved behavior can be explained with the 'Prisoner's dilemma' behavior [15]. This is a situation when multiple actors in some situation are acting in their own self interest, with the intention of achieving the best possible outcome, at the cost of the other participant's loss. The result in this paradox however is that both parties always fail to reach the highest personal output from the situation, resulting in a none-optimal solution. The authors propose that this is the case with the boid formations in the above described environment, but further investigation is required. Furthermore, it is suggested that future developments of that work, such as adding walls to the terrain, including obstacle or including food, hold the potential to even more interesting results in terms of emerging behavior. What is more, the experiment can be taken even further, by allowing the predator evolve a behavior of its own and then repeat the cycle, thus leading to co-evolution of both sides.

2.2 Flocking boids

Created in 1986 by Craig Reynolds [34], 'Flocking boids' is a work that introduces an algorithm for achieving emergent flocking behavior, inspired by real life grouping of large numbers of animals. What is fascinating about it is that it requires just 3 simple rules in order to achieve a complex behavioral pattern. More specifically, those rules are *cohesion*, *alignment* and *avoidance*. Each agent in the flock is guided by the combination of those 3 core behaviors, meaning that there is no requirement for unified control of the group. This leads to a complex swarming pattern of the group, at the expense of very low computation cost, due to every member being completely autonomous. This makes the flocking boid algorithm popular to this day, as it holds a great potential for further development and serves as a basis for many, more complex works.

2.2.1 Cohesion

Cohesion behavior is implemented by first defining a vision radius for the agents (see figure 5). Depending on the size of that radius, each agent will consider a certain number of neighbouring agents at any given time. The way the cohesion movement vector is calculated, is by taking the positions of all neighbouring agents and averaging

them into one point of interest. The movement vector simply points towards this point, ensuring that the agent remains in close proximity to his neighbours.

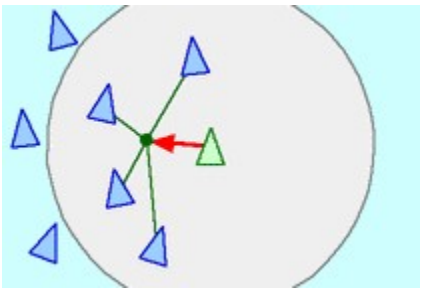


Figure 5: Visualisation of cohesion behavior [34]. The red arrow marks the direction vector for the observed agent towards the average position of the neighbouring members.

2.2.2 Alignment

The next component behavior called alignment works in a similar way to the cohesion, in that each agent considers the number of neighbours currently in his vision radius and ignores every other agent. The difference is in that in this case, the average is taken between the forward facing vectors of the neighbours and the agent's orientation is adjusted. This ensures that once part of a local group, all agents share similar directional orientation.

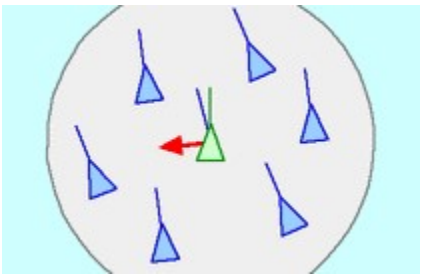


Figure 6: Visualisation of alignment behavior[34]. The red arrow shows the direction towards which the orientation angle of the observed agent should be changed in order to match the average orientation of the surrounding neighbours.

2.2.3 Avoidance

Avoidance is a bit different from the other two key behaviours in that it works in regards to a second predefined radius (usually smaller than the neighbour radius), called avoidance radius. Here, the agent monitors how many neighbours will get too

close to him in order to avoid collision. Once one or more neighbours are within the avoidance radius, the average position of those is calculated and the agent is redirected in the direction pointing away from that position.

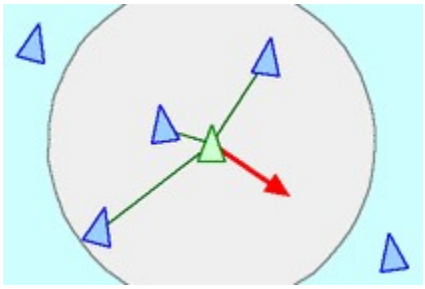


Figure 7: Visualisation of avoidance behavior[34]. The red arrow points in the direction of the movement vector opposite to the one in the direction of the average position of the neighbouring members, that are within the avoidance radius of the observed agent.

2.3 Emergent behaviors

As the technicalities of how the flocking boids algorithm operates have been already introduced, the following section is going to explore more in-depth the meaning of one of the most important aspects of this work, that is emergence.

When talking about emergent behaviors, it is not difficult to imagine scenarios from the natural and none-natural world, where the combination of simple composing parts into one, can lead to unforeseen outcomes where the composite system holds much greater capacity for complex behavior than its composing parts alone. Following from the previous section, an every day occurrence of an emergent behavior can be seen in the formations created by flocks of birds during flight, evolved as defence mechanism against predators. When grouped together, the feathered seek to appear as one large entity, confusing the predator and eventually achieving a significant reduction of hunt-down prey [5]. Similarly, certain species of fish will form schools while foraging, with the purpose of increasing their chances for survival [30]. Furthermore, emergent behaviors can be also observed in the insect world, where for example ants would form colonies, that are vastly more beneficial survival strategy for the survival of their species, than rejecting cooperation [38]. This behavior is known as swarming.

One of perhaps the most important examples of emergence is the human brain, where the combination of a large number of fairly simple in functionality parts known as neurons, leads to the manifestation of arguably the most complex system in the known world [21].

As suggested above, examples of emergent behaviors can be seen not only in

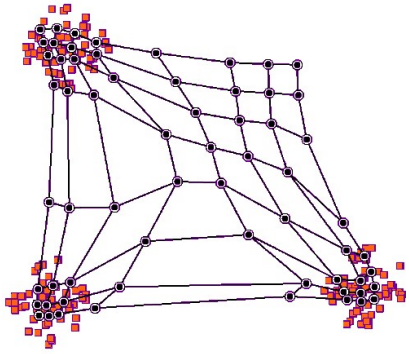
nature. In man-made systems such as the field of software, emergence can often be the result of bringing a new component into an already well-explored and known system, leading to unpredicted behavior, that can be either beneficial or leading to a failure [29].

By definition, an emergent behavior is something resulting from the creation of a composite system, that exhibits higher level of complexity than its sub-parts [25]. Emergence itself, is described as the occurrence of systems or entities, that posses properties which can not be observed in their singular parts. That is to say, when describing emergent behaviors in a system of multiple agents, the system is observed as one singular entity, despite the lack of overall governance. What is more, emergent systems as flocking boids for example are controlled on a local level, where every agent can be considered an autonomous piece of the whole, but lead to the emergence of higher complexity only once a larger group operates together.

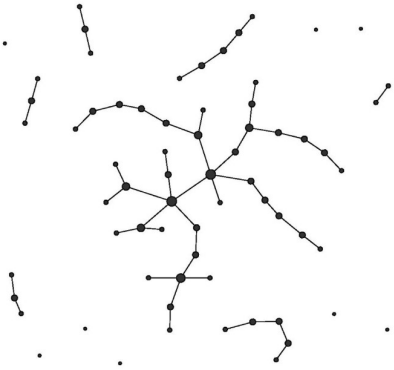
It is speculated that emergent behaviors occur primarily in systems prone to *none-linearity* [25]. That is to say, the output of the system does not match the input in proportion. Another way to define that is through the mathematical field known as *Chaos theory*. This branch of mathematics and statistics deals with systems that despite of being perceived as completely random in behavior, are actually governed by very deterministic laws, the hard-to-conceptualize nature of which leads to the uniqueness of the emergent properties. To emphasize further, it can be speculated that emergence is a property of systems rich on chaotic characteristics, otherwise referred to as entropy, that given enough time, lead to self-organization and are strongly related to the initial conditions of the system [25]. In other words, chaotic systems have the potential for *self-organization*.

Self-organizing systems are such systems that undergo a process known as *spontaneous order*, leading to the formation of organized structures or behaviors between the parts of the system, without the need for external intervention [4]. As suggested earlier, self-organization occurs in both man-made and natural systems. More specifically, in the field of computer science, examples of spontaneous order systems are random graphs, cellular automata, evolutionary algorithms, multi-agent systems, swarm intelligence etc. (see figure 8). [26].

(a) Conway's 'Game of life' cellular automata[1]



(b) Self organizing map[26]



(c) Random graphs[26]

(d) Karl Sims's procedurally evolved creatures[37]

Figure 8: Examples of self-organizing systems

The reason why this is important is because a known property of none-linear systems is that a very minor change to the overall composition, can often result in a significant change in the output. A well known example for that are the studies conducted on the *Lorenz attractor* in the movement of systems of planetary bodies. Even though that the system is by nature deterministic and therefore predictable, micro-changes in the input of the simulation that can be otherwise disregarded as rounding errors, have the potential to lead to extreme differences in the final results [25].

To summarize, this leads to the conclusion that when designing a setup meant to manifest emergent properties, a great deal of attention should be directed towards the initial conditions and overall governing rules of the system. It can be speculated that the key to achieving a certain desired behavior from the system, lies in the balance between a set of rules, that are specific enough in order to lead to the desired outcome, yet not limit the level of entropy of the system, thus allowing for a large set of possible outputs to manifest.

2.4 Predator satiation

This section is going to discuss one reemerging topic of interest in the works related to emergent behaviors, that is *predator satiation* and its relation to the topic of this project.

One way to classify animals in nature is by separating them to predators and prey. There are numerous observable tactics derived from the notion of prolonging survival, that can be observed in both sides. One such strategy is predator satiation, which is related to the efforts of mass groups of prey to reduce the probability for each singular member to be eaten by a predator [28] (see figure 9).

(a) A flock of starlings avoiding a hawk in a 'Sort sol' formation ('Black sun' from Danish)

(b) A school of fish avoiding a shark

Figure 9: Examples of predator satiation and 'safety in numbers' behaviors [39]

More specifically, predator satiation involves an anti-predator adaptation behavior in which the grouping prey aims to overwhelm the predator by appearing in much higher numbers and densities than the predator can consume. Once the predator is flooded with prey, it can consume only a certain maximum amount, which combined with the higher density of the prey, ultimately results in less individual casualties. Furthermore, this is speculated to have a relation to the *safety in numbers* hypothesis, which states that almost every type of animal would, in certain situations, show a tendency to a mass behavior of grouping with its self-similar. This is speculated to be induced by efforts to increase safety and general chances for survival [28].

Furthermore, the predator satiation hypothesis occurs in both animals (formations of flocks of starlings and schools of fish for example) and plants (mast seeding). However, the specific topic of interest for this project will be the instance of this behavior occurring in the non-vegetal part of the animal world.

The reason why it is important to realise the specifics behind this phenomena is because this is the exact type of behavior that has served as inspiration for Craig Reynolds in the creating of his flocking algorithm [34]. The grouping instinct observable in almost every animal in nature, is usually the result of a multitude of outside

stimuli and not just predator avoidance. The common occurrence of this behavior is also part of the reason why the flocking boid behavior itself appears as so natural and animal-like, despite of it being the result of such simple and crude rules [34]. Furthermore, there are numerous approaches that are speculated to serve as a catalyst for emergent grouping behavior of agents in virtual environments - using a predator, implementing a food seeking behavior, an escape based behavior to name a few [6]. Some of those approaches have even already been observed in practical application in some of the examples presented in beginning of the Analysis.

2.5 Reinforcement learning and proximal policy optimization

This chapter will describe in higher detail the specifics behind the machine learning method of choice for this project.

There are three main types of widely utilized machine learning (ML) algorithms nowadays - supervised, unsupervised and reinforcement learning. While the first two are paradigms usually related to solving problems that include non-dynamic environments, the later has become the state-of-the-art approach when it comes to dealing with intelligent agents in games [36]. More specifically, reinforcement learning is a ML approach that deals with teaching virtual agents, by relaying on a specific action-state-reward causality relationship, instead of manually labeled data. To emphasize further, this relation produced by the interaction between the actor (agent) and the space-time state of the environment is also known as the *Markov decision process* (MDP) [36]. What is characteristic about this approach is the *memorylessness* of the algorithm, also referred to as the *Markov property*.

To explain this into further detail, while the environment operates in predefined time-steps, at each step the decision maker (agent) finds itself in some state s. Going forward, the agent can choose between a number of possible actions A, dependent on the current state he is in. Once the agent proceeds with the chosen action, the process (environment) will respond by transitioning into a new state s' (considered random from the perspective of the agent), rewarding the agent with reward $R_a(s, s')$ based on the action he has chosen. Taking into consideration the causality of this cycle, it can be inferred that the new state s' is directly affected by the chosen action a, which on the other hand is a result of the previous state s. That is to say, the decision making process is conditionally independent of past states beyond s, thus satisfying the aforementioned Markov property.

However, reinforcement learning is far from perfect. Traditional reinforcement learning methods are synonymous for problems such as intractability of the unwanted behavior of the algorithm due to things like lack of certainty in the agents value function (how good would a certain state be in terms of future reward) and overwhelming complexity arising from continuous state/action space [36]. Some of these issues are

solvable by introducing *Policy gradient* (PG) methods, that make use of *gradient descent* optimization, which is essentially a mathematical method used to find the minimum of a given function, in this case the loss of the trained agent. Even with that however, PG methods are still prone to problems such as over-sensitivity to chosen step size or need of tremendously long learning times, resulting from poor sampling efficiency [36].

This is where *Proximal policy optimization* (PPO) steps in. PPO models are essentially an improved version of traditional RL with PG models in that they combine the relative ease of hyper parameters tuning of supervised models with PG update computation at every step, leading to minimized deviation from previously learned policies [36]. Furthermore, this improvement in performance is achieved through the use of *Stochastic gradient descent*, which is an alternative way for optimizing the agent's objective function, by relying on a smaller number of random samples from the collected data, instead of sampling the entire available data set as with traditional GD [36].

Considering the above described benefits of RL with PPO, this approach is further a beneficial choice due to its wide availability through Unity engine's newly introduced ML library, namely Unity ML-agents [36].

2.6 Problem statement

Trough the extensive research described in the Analysis, it was shown that virtual crowd simulations find great use in the video game, cinematography and other fields. Studying the capabilities of emergent systems, it was hypothesized that complex nature-like grouping behavior can be derived using only few simple initial rules and conditions. Furthermore, the research suggests that the output of the system can be greatly enhanced by the use of additional catalyst conditions, such as predatory threat or need for foraging. Several great examples of that were described in detail, both from the entertainment industry and the scientific field studying the dynamics of virtual emergent grouping behaviors. Taking all of this into consideration, an interesting direction for this project would be to create a complex emergent crowd behavior that also has practical applicability in video games, cinematography, etc. In addition, it was speculated upon the information gathered regarding self-organizing systems, that a proper choice of initial rules that does not diminish the level of entropy of the system, would lead to a much larger set of potential outcomes. Translated to the field of emergent behaviors, this would mean that in order to achieve a certain desired behavior, a balance must be set in the initial rules of the environment, such that they do not over-specify the desired outcome, yet still narrate towards it. In other words, more generalized set of conditions rather than too specific ones, have the potential for a much more complex emergent crowd behavior. Finally, in order to practically achieve that, a state of the art reinforcement learning approach is to be utilized. Combining its applicability in both video games and movie-making with its

machine learning capabilities, Unity engine and its ML-agents library are the perfect tool for the realisation of this project.

With this in mind, the following problem statement can be formulated:

Is it possible to achieve emergent flocking boid-based crowd behavior through generalization of the system rules, using a 3D Reinforcement learning environment with predator satiation and foraging of the virtual crowd?

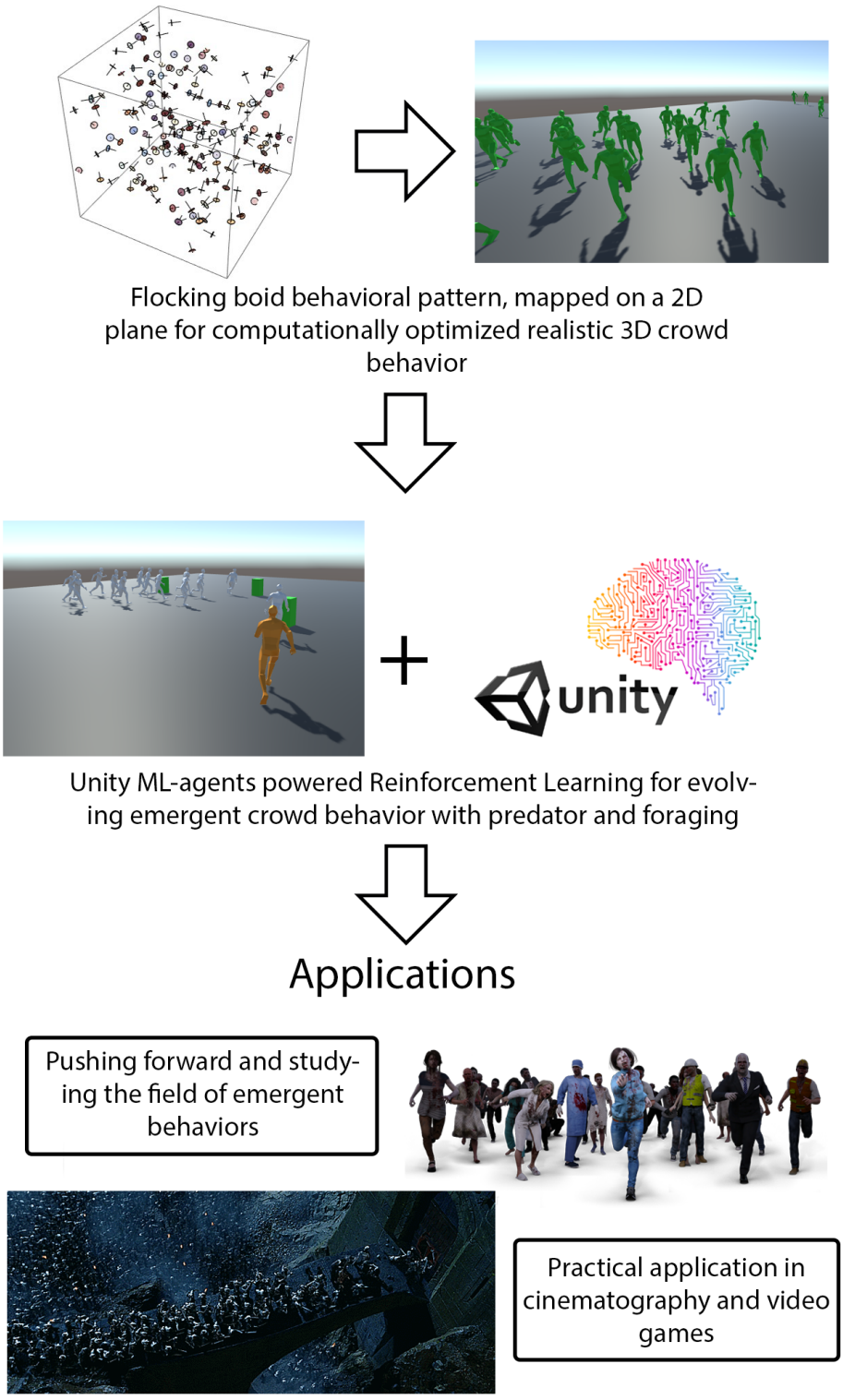


Figure 10: Flowchart visualizing the evolution of the project direction

3 Methods

This section is going to describe the techniques used to evaluate the problem statement of the project.

As described in the Problem statement section, the main goal of this project would be to create a Reinforcement learning PPO model, that can learn a flocking boid-based crowd behavior through generalization of the rules of the emergent system. In the context of RL, these rules can either be coming from the environment in which the ML model is situated or from the model itself. In terms of the ML model, these rules are represented by the fitness function, which aims at guiding the behavior of the model through defining some constraints. These constraints are realised in the form of different rewards, that are given to the agent based on the exhibited behavior. This is where the main focus for achieving the aforementioned balance will be, as apart from side factors such as neural network hyper parameters and overall structure of the environment, the fitness function is the main key to successful training of the any RL agent [36].

In terms of the environment, the external factors that will shape the behavior of the ML model are the inclusion of a predator and need for food. As shown earlier in the Analysis, both of these are proven to serve as catalysts for nature-like crowd behaviors of different virtual crowds.

Furthermore, it was described in the Problem Statement section that an additional goal would be that the product of this project can have further practical utility, beyond just empirical study of the dynamics of the emergent behavior. The majority of the examples of emergent flocking boid-based crowd behaviors studied in the Analysis implement only the minimal requirements in terms of visual fidelity of their products. The reason for that is because as already mentioned, most of these projects focus solely on studying the dynamics of the emergent system. In that case, a simple two dimensional environment with the agents and other inhibitors and objects represented by simple geometric shapes would suffice and it will even further benefit the performance of the application. The goal of the current project is to go a step further, by creating a product which upon completion of the training of the ML model, can be directly imported and used as an asset for a bigger video game project, movie or other. In order to accomplish that, the product will be implemented in a 3D environment in Unity engine, which is used both as a game engine and a film making tool [11], [12]. What is more, the agents inhabiting the environment will be represented by fully animated biped humanoids, thus further enhancing the visual fidelity of the created behavior.

Finally, the level of success of the presented product is to be evaluated. The way this will be done is by comparing the produced emergent behavior to the original flocking boid behavior. To achieve a 1 to 1 comparison where the additional elements of the environment do not create biases, a second visually and conditionally identical implementation will be made, with the only difference being that instead of a ML

model governing the virtual crowd, the original flocking boid rules will serve for navigation. The trained ML model will then be compared in terms of how close its learned behavior is to each of the 3 flocking behaviors (cohesion, alignment and avoidance).

The technicalities of the above described concepts will be discussed into further detail in the Design and Implementation chapter of the report. Furthermore, an in-depth description of the evaluation setting can be found in the Evaluation chapter.

4 Design and implementation

This section is going to describe the technical details behind the product of this project. The primary tools used to build the implementation are Unity engine [13], Microsoft Visual Studio [20] and Unity's Ml-agents library [22] with Anaconda 3 with Python 3.6 [2].

4.1 Flocking boid

As the purpose of this project is to compare the behavior of the trained agents to the original flocking boid implementation, first a control environment had to be created. The boid implementation consists of two main classes - 'Crowd Manager' and 'Crowd Member'.

4.1.1 Crowd Manager

The 'Crowd Manager' class is responsible for instantiating the flock with the appropriate initial parameters (see figure 11).

```
for (int i = 0; i < startingCount; i++)
{
   agentController newAgent = Instantiate(
        agentPrefab,
        new Vector3(Random.insideUnitCircle.x * agentDensity, yPos, Random.insideUnitCircle.y * agentDensity),
        Quaternion.Euler(Vector3.up * Random.Range(0f, 360f)),
        this.transform
        );
    newAgent.name = "Agent " + i;
    agents.Add(newAgent);
}
```

Figure 11: 'Crowd Manager' script main function.

The 'Instantiate' function holds the 'agentPrefab', which in turn contains all the visual details of the individual agent, as well the 'Crowd Member' script (see figure 12). Furthermore, the 'Instantiate' position field is overridden with a short function that spawns each new agent at a random position inside a unit circle, with radius controlled by a density variable. Same thing can be seen about the initial orientation field, where every agent has a new random initial orientation, making sure the entire flock does not face the same direction upon initialization. Finally, all agents are assigned a name and added to a list, from which are later accessed by the 'Crowd Member' script.

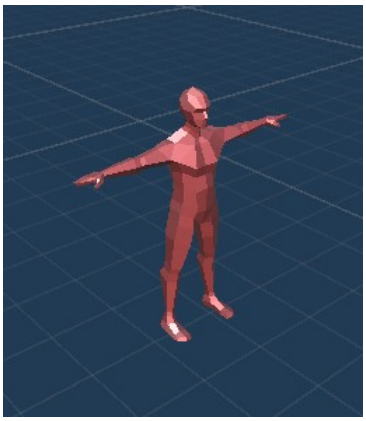


Figure 12: 'agentPrefab' character model

Character controller

The visuals of each individual agent, along with the animations applied to them, have been achieved through the use of advanced version of Unity's standard assets character controller model, which includes a biped humanoid character, fully animated for running, crouching and jumping. The reason why the control of the character here is more special, is related to the way the animation is applied to the 3D model. Usually, when creating a walking animation for example, the way of deciding what animation to play at every frame is by moving the character into the desired direction and then checking their speed. Then, depending on the speed of the character, the appropriate animation is run at that frame (see figure 13). That is to say, we first determine position P2 and then play the animation appropriate to the character movement to P2. As much as it is robust and simple to implement, this approach however fails once the goal is to have not only animation for running forward, but also such more complex one for turning around, changing direction at varying speeds and so on. The solution to that is to instead reverse the process by moving the character based on the played animation sequence. That is to say, once we determine the position P2 that we want the character to be in, we first play the animation appropriate for that transition and only then move the character, based on the played animation and not on the position we want them to be in.

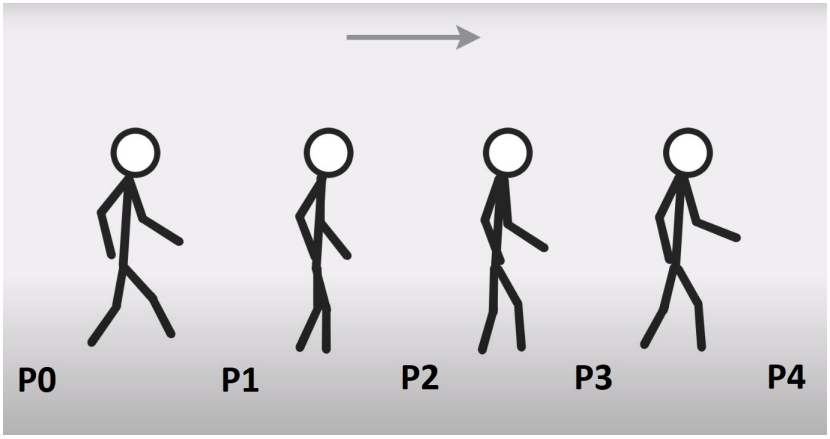


Figure 13: Animation positions [3]

4.1.2 Crowd Member

The 'Crowd Member' class deals with the navigation of each individual agent. As mentioned earlier in the analysis, one of the big advantages of Reynolds's boids algorithm is that there is no need for overall control of the flock. Instead, the behavior of each agent is governed locally. With that being said, the 'Crowd Member' class is the heart of the flocking boid part of the project, as it contains all the necessary ingredients for achieving the desired behavior of the agents.

The 'Crowd Member' class consists of 5 main functions, 4 of which are responsible for the separate flocking boid sub-behaviors Cohesion, Alignment and Avoidance, including one additional custom behavior called Obstacle Avoidance, which is unique to this project. The fifth main function of this class is simply called 'Move' and it serves as a weighting system for the aforementioned behaviors, combining them into a singe output vector.

Cohesion

The cohesion function of the 'Crowd Member' class serves to the same purpose as the already explained in the Analysis original cohesion behavior by Reynolds [34], with some small modifications, required as so to fit to this project scenario (see figure 14).

Firstly, another side function is called, that is GetNeighbours(), which returns a list of all agents that are currently within the predefined neighbour radius of the one of interest. Should there be no neighbouring agents at the current frame, the returned value e 0. If there are however any neighbours, their positions are taken and summed into a single composite 3D vector. Then, the summed positions are divided by their total number in order to produce an average value. Finally, the direction of the resulting vector is set to point away from the position of the current agent

of interest, by subtracting his position value from the whole. This logic can be also expressed in the form of the following equation, where C is the composite cohesion vector for the current agent, N is the total number of neighbouring agents A of that agent, and $A.pos$ is the position of each neighbour:

$$C = \frac{\Sigma_{i=1}^{N} A_i.pos}{N}$$

Additionally, the cohesion function has been altered by the original by using a 'SmoothDamp' function, in order to make the transition between two consecutive directions seamless in real time. This is important due to the fact that from all three boid sub-behaviors, the cohesion function has the largest reflection onto the actual movement of the agents and being called every frame will result in a jittery motion in the flock, due to the constant overall movement.

```
Vector3 Cohesion()
{
    Vector3 currentVelocity = new Vector3();
    float agentSmoothTime = 0.5f;
    Vector3 cohMove = Vector3.zero;
    if (GetNeighbors().Count == 0)
    {
        return Vector3.zero;
    }
    foreach (myCrowdMember a in GetNeighbors())
    {
        cohMove += a.transform.position;
    }
    cohMove /= GetNeighbors().Count;
    cohMove -= this.transform.position;
    cohMove = Vector3.SmoothDamp(this.transform.forward, cohMove, ref currentVelocity, agentSmoothTime);
    return cohMove;
}
```

Figure 14: 'Cohesion' function of the 'Crowd Member' class

Alignment

Similar to the cohesion function, the alignment function makes use of the total number of neighbouring agents, but instead of their positions, it sums the vectors pointing in the relative forward direction for each neighbour at the current frame (see figure 15). If there are no neighbours in the vicinity of the agent of interest, there is no change to the orientation of the agent. The logic of the function can be expressed in the following equation, where $Align$ is the composite alignment vector for the current agent, N is the total number of neighbouring agents A of that agent, and $A.forward$ is the local forward vector of each neighbour:

$$Align = \frac{\Sigma_{i=1}^{N} A_i.forward}{N}$$

```
Vector3 Alignment()
{
    Vector3 alignMove = Vector3.zero;
    if (GetNeighbors().Count == 0)
    {
        return this.transform.forward;
    }
    foreach (myCrowdMember a in GetNeighbors())
    {
        alignMove += a.transform.forward;
    }
    alignMove /= GetNeighbors().Count;
    return alignMove;
}
```

Figure 15: 'Alignment' function of the 'Crowd Member' class

Avoidance

The avoidance function is slightly different from the previous two in that it makes use of a second radius around the agent of interest, called avoidance radius (see figure 16). If there are any neighbours within the avoidance radius of the current agent of interest, their positions are used to calculate a vector pointing away from the current agent for each neighbour. Then, those vectors are summed together and divided by the total number of neighbours that are too close in order to average them. Should there be no neighbours stepping within the avoidance radius of interest at the current frame, no division is performed and a 0 value is returned in order to avoid errors. The logic of the function can be expressed in the following equation, where $Avoid$ is the composite avoidance vector for the current agent A_c, N is the total number of neighbouring agents A of that agent, and $A.pos$ is the position of each neighbour:

$$Avoid = \frac{\Sigma_{i=1}^{N} A_c.pos - A_i.pos}{N}$$

```
Vector3 Avoidance()
{
    Vector3 avoidMove = Vector3.zero;
    int nAvoid = 0;
    if (GetNeighbors().Count == 0)
    {
        return Vector3.zero;
    }
    foreach (myCrowdMember a in GetNeighbors())
    {
        float distance = Vector3.Distance(this.transform.position, a.transform.position);
        if (distance < avoidanceRadius)
        {
            avoidMove += this.transform.position - a.transform.position;
            nAvoid++;
        }
    }
    if (nAvoid > 0)
    {
        avoidMove /= nAvoid;
    }
    return avoidMove;
}
```

Figure 16: 'Avoidance' function of the 'Crowd Member' class

Obstacle avoidance

The main reason for the obstacle avoidance behavior is so that the agents can maneuver on a terrain surrounded by walls. Usually, when designing an obstacle avoidance function, the approach would be similar to that used for the flocking avoidance method. However, when dealing with objects such as walls, that can not be effectively described by a single position point, another options must be investigated. This lead the current obstacle avoidance function (see figure 17), which makes use of a second supplementary class called Fibonacci Rays (see figure 18). The way the supplementary class works, is by casting a sequence of rays, starting from the forward vector of the assigned object and pointing away from its center. If the entire sequence of rays is completed, the shape that they describe would be a sphere. This way, the ObstacleAvoidanceForce() method checks every next cast ray, if its direction is obstructed. Once an unobstructed direction is discovered, it is returned by the function as a moving direction.

```
Vector3 ObstacleAvoidanceForce()
{
    Vector3[] rayDirections = FibonacciRays.directions;

    for (int i = 0; i < rayDirections.Length; i++)
    {
        Vector3 dir = this.transform.TransformDirection(rayDirections[i]);
        Ray ray = new Ray(this.transform.position, dir);
        if (!Physics.SphereCast(ray, raySphereRadius, obstCollisionAvoidDst, obstacleMask))
        {
            return dir;
        }
    }
    return Vector3.zero;
}
```

Figure 17: 'Obstacle avoidance' function of the 'Crowd Member' class

```
static FibonacciRays()
{
    directions = new Vector3[FibonacciRays.numViewDirections];
    float goldenRatio = (1 + Mathf.Sqrt(5)) / 2;
    float angleIncrement = Mathf.PI * 2 * goldenRatio;

    for (int i = 0; i < numViewDirections; i++)
    {
        float t = (float)i / numViewDirections;
        float inclination = Mathf.Acos(1 - 2 * t);
        float azimuth = angleIncrement * i;

        float x = Mathf.Sin(inclination) * Mathf.Cos(azimuth);
        float y = 0;
        float z = Mathf.Cos(inclination);
        directions[i] = new Vector3(x, y, z);
    }
}
```

Figure 18: 'Fibonacci Rays' class

Predator avoidance and foraging

The way the flock agents avoid the predator and move towards food is quite straight forward. Firstly, the AvoidPredator() method works by simply checking if the predator is within a certain predefined avoidance radius. If yes, the function returns the direction vector pointing in the direction opposite of the predator (see figure 19).

Also similarly simple is the Forage() function, which sorts through all available food pieces and determines the closest one. Then the function returns the direction vector pointing towards that piece (see figure 19).

```
public Vector3 AvoidPredator()
{
    if (Vector3.Distance(this.transform.position, predator.transform.position) <= avoidanceRadius+1)
        return this.transform.position - predator.transform.position;
    else return Vector3.zero;
}

public Vector3 Forage()
{
    return (GameObject.FindGameObjectsWithTag("Food")
            .OrderBy(t => (t.transform.position - this.transform.position).sqrMagnitude).FirstOrDefault()
            .transform.position - this.transform.position).normalized;
}
```

Figure 19: Predator avoidance and foraging functions

Combined weighted movement

After all the other sub-behaviors have been sampled for the current frame, their resulting vectors are combined into one composite force that is used to move each agent (see figure 20). The Move() function does exactly that, by normalizing and weighting each vector. The weights for each sub-behavior have been chosen trough trial and error. Furthermore, the resulting vector is clamped to a predefined maximum value and scaled appropriately by a chosen speed. Finally, the product of this whole operation is scaled again by a fixed time update, in order to ensure that the movement of the agents is not out of sync with the rest of the environment during run time.

```
void Move()
{
    Vector3 move = Vector3.zero;
    int numOfBehavs = 4;
    Vector3[] behaviors = new Vector3[numOfBehavs];
    behaviors[0] = Cohesion();
    behaviors[1] = Alignment();
    behaviors[2] = Avoidance();
    behaviors[3] = ObstacleAvoidanceForce();
    float[] weights = new float[numOfBehavs];
    weights[0] = cohesionWeight;
    weights[1] = alignmentWeight;
    weights[2] = avoidanceWeight;
    weights[3] = obstacleAvoidanceForceWeight;
    for (int i = 0; i < behaviors.Length; i++)
    {
        Vector3 partialMove = behaviors[i] * weights[i];
        if (partialMove != Vector3.zero)
        {
            if (partialMove.sqrMagnitude > weights[i] * weights[i])
            {
                partialMove.Normalize();
                partialMove *= weights[i];
            }
            move += partialMove;
        }
    }
    move *= driveFactor;
    if (move.sqrMagnitude > squareMaxSpeed)
    {
        move = move.normalized * maxSpeed;
    }

    character.Move(move * Time.deltaTime, false, false);
}
```

Figure 20: 'Move' function of the 'Crowd Member' class

4.2 ML environment

The ML environment consists of 3 main pieces, that are the crowd of agents controlled by the ML model, the predator and the interactive environment in which those two dwell.

As mentioned earlier, this project makes use of Unity ML-agents library for implementing Reinforcement learning. This ML environment consists of several intercommunicating parts (see figure 21). The agent holds the implementation of the fitness function and is the learner in the system. To him is connected a brain, which is the neural network, which determines the behavior of the agent. As the agent interacts with the environment, the brain will decide the behavior of the agent accordingly. All of these are connected to the Academy class, which acts as an overseer of the entire process. Furthermore, the progression of the neural network is handled by Python ML library.

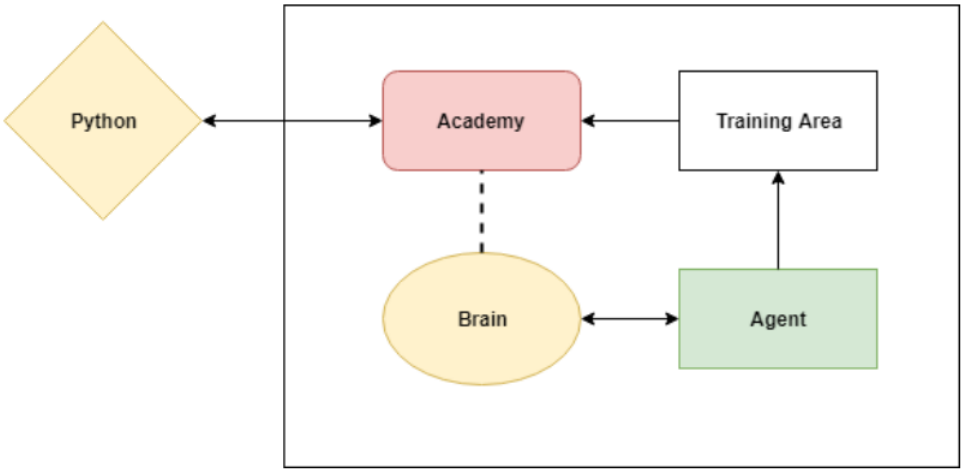


Figure 21: Unity ML-agents environment flowchart [31]

The hyper parameter values for the agent's brain can be seen in figure 22. The majority of these values have been chosen through trial and error, in order to provide the best possible learning for the current environment.

```
trainer: ppo
batch_size: 128
beta: 5.0e-3
buffer_size: 2048
epsilon: 0.2
hidden_units: 128
lambd: 0.95
learning_rate: 3.0e-4
learning_rate_schedule: linear
max_steps: 5.0e8
memory_size: 256
normalize: false
num_epoch: 3
num_layers: 2
time_horizon: 64
sequence_length: 64
summary_freq: 1000
use_recurrent: false
vis_encode_type: simple
reward_signals:
    extrinsic:
        strength: 1.0
        gamma: 0.99
```

Figure 22: NN hyper parameters

4.2.1 Training environment

The physical part of the environment consists of rectangular floor, surrounded by walls that limit the available field for both agents and predator (see figure 23). The

size of the usable area has been chosen with the total number of agents that are to be used in mind. That is to say, through trial and error in the initial iterations of the implementation, it was determined that a too small or too big area leads to either overpopulation by agents or inability to exploit the entire usable field effectively, thus leading to behavioral artefacts.

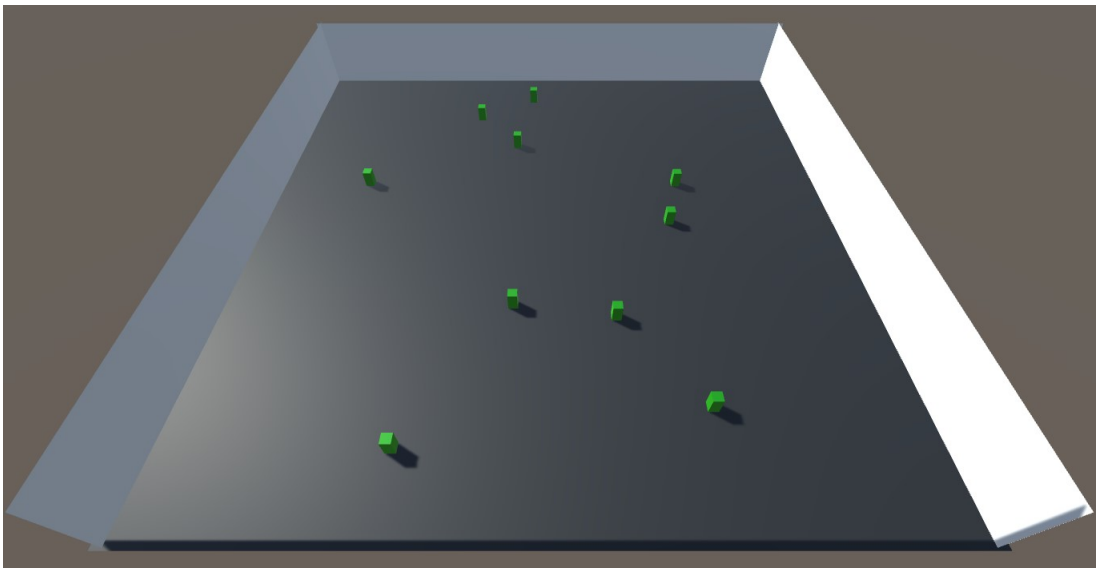


Figure 23: Environment with food represented by green cubes

In addition, the environment comes with a class called 'Food Caster', which is used for instantiating and keeping track of the food available for the agents. The main functionality of the 'Food Caster' class can be found in the GetRandomPos() function, which through recursion keeps track of the positions of the food pieces scattered around the map (see figure 24). Upon starting the simulation, a predefined number of food pieces are instantiated at random, non-repeating positions. Once an agent consumes a food piece, the 'Food Caster' will destroy it and replace it with a new one, at a new random position, thus preserving a constant number of food pieces on the map throughout the entire simulation.

```
Vector3 GetRandomPos()
{
    Vector3 tmpVector = new Vector3(Random.Range(floorCol.bounds.min.x + spawnAreaPadder, floorCol.bounds.max.x - spawnAreaPadder),
        floor.position.y + foodPrefab.transform.localScale.y / 2,
        Random.Range(floorCol.bounds.min.z + spawnAreaPadder, floorCol.bounds.max.z - spawnAreaPadder));

    if (usedInitPoses.Contains(tmpVector)) return GetRandomPos();
    return tmpVector;
}
```

Figure 24: Food positioning function

4.2.2 Crowd

The flock formation or the crowd, consists of identical instances of the same ML agent, that combines a Unity Ml-agent with the same character controller used for the flock members described above.

Firstly, the agent receives observations from the environment which serve as feedback for his actions. This is handled by the CollectObservations() method (see figure 25), which in this case passes 9 different feature vectors, equaling to a total of 17 observation values for the brain. These are the following:

- Agent position.
- Agent velocity on the x and z axis.
- Agent forward orientation vector.
- Predator position.
- Predator velocity on the x and y axis.
- The distance between the agent and the predator.
- Predator orientation vector.

```
public override void CollectObservations()
{
    AddVectorObs(this.transform.position);
    AddVectorObs(GetMyVelocity().x);
    AddVectorObs(GetMyVelocity().z);
    myLastPosition = this.transform.position;
    AddVectorObs(this.transform.forward.normalized);

    AddVectorObs(predator.transform.position);
    AddVectorObs(predatorRBody.velocity.x);
    AddVectorObs(predatorRBody.velocity.z);
    AddVectorObs(Vector3.Distance(this.transform.position, predator.transform.position));
    AddVectorObs(predator.transform.forward.normalized);
}
```

Figure 25: Agent observations

The second part of the agent observations are handled by a raycast function which can detect 4 different objects - other agents, predator, food and walls (see figure 26). The raycast of the agents consists of 18 vectors in total with a spread of 340 degrees. This ensures that the agents have no blind spots.

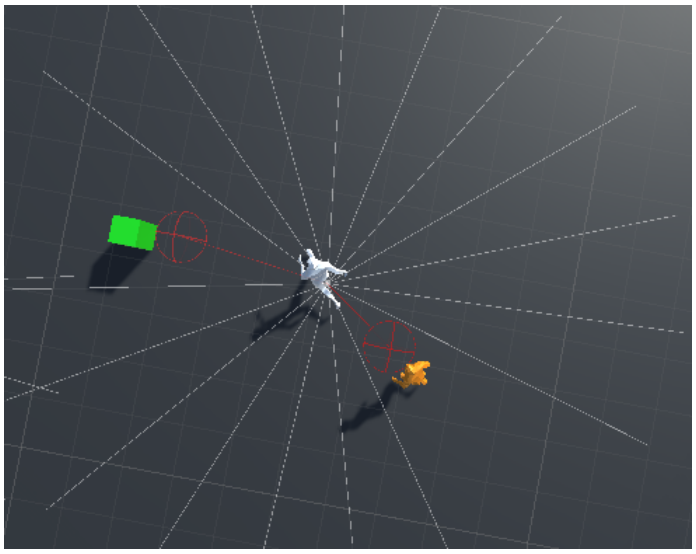


Figure 26: Agent raycast observations

The reason why the agent velocity is passed through a hand-made function instead of Unity's native Rigid body velocity method is because of the way the later is calculated. Essentially, in situations when the agent is facing an obstacle and running towards it, the native velocity method would still return a positive value, despite of the agent staying in one place. This is highly problematic, as the velocity values are essential observation for the learning of the agent. Therefore, if those values are false in certain moments, this can lead to the agent updating its policy for situations which are wrongly described by his experience.

Furthermore, the AgentAction() function is the most important from the agent class, as it defines the reward system for the agent (see figure 27). Firstly, the agent receives a control signal vector from the brain, which is used to move the avatar. Then three separate conditional checks are performed - whether the agent is colliding with food (see figure 29), whether he is colliding with the predator and whether he is colliding with any of the walls surrounding the training area. Upon collision with food, the agent is rewarded with a positive reward of 0.5. Upon collision with the predator, the reward is negative and twice as big. Additionally, the agent is rewarded a second negative reward of 0.5 for colliding with the walls. Upon collision with any of the food pieces, the food is removed and a new piece is positioned at a new random point on the field. The Done() method is called upon collision with wall or predator, as it is used for resetting the agent (see figure 28), which terminates the learning episode and re-instantiates the agent at a new random position, with a random facing direction. The reward values were chosen through numerous trials and errors and are aimed at balancing the behavior of the agent in terms of the different parts of the surrounding environment. The reason why the agents are penalized for

hitting the walls, is in order to define the boundaries of the movable area for the brain. During the initial iterations it was discovered that if no such penalty is introduced, the agents will inevitably regress towards a behavior of 'picking a corner' until the end of the learning episode, which lead to no significant progress in the policy updates.

```
public override void AgentAction(float[] vectorAction)
{
    Vector3 controlSignal = Vector3.zero;
    controlSignal.x = vectorAction[0];
    controlSignal.z = vectorAction[1];
    character.Move(Vector3.ClampMagnitude(controlSignal, 1.0f) * speed * Time.deltaTime, false, false);

    CheckIfCollidingWithFood();

    if (Physics.CheckSphere(this.transform.position, this.GetComponent<CapsuleCollider>().radius + 0.1f, wallMask))
    {
        SetReward(-0.5f);
        Done();
    }

    if (Vector3.Distance(this.transform.position, predator.position) <= 0.65f)
    {
        SetReward(-1.0f);
        Done();
    }

    Monitor.Log("reward", GetReward());
    FellOffThePlatform();
}
```

Figure 27: Agent Action function

```
public override void AgentReset()
{
    this.transform.position = new Vector3(
        Random.Range(floorCol.bounds.min.x + spawnAreaPadder,
        floorCol.bounds.max.x - spawnAreaPadder),
        0f,
        Random.Range(floorCol.bounds.min.z + spawnAreaPadder,
        floorCol.bounds.max.z - spawnAreaPadder)
        );
    this.transform.rotation = Quaternion.Euler(Vector3.up * Random.Range(0f, 360f));
}
```

Figure 28: Agent reset function

```
void CheckIfCollidingWithFood()
{
    Collider[] hitColliders = Physics.OverlapSphere(this.transform.position, 0.4f, foodMask);
    if (hitColliders.Length != 0)
    {
        SetReward(0.5f);

        for (int i = 0; i < hitColliders.Length; i++)
        {
            hitColliders[i].gameObject.transform.position = GetRandomPos();
        }
    }
}
```

Figure 29: Agent food collision function

4.2.3 Predator

The predator also consists of a character controller and a predator class, that determines his movement. While the character controller for the predator is essentially identical to the one used for the agents of the flocking boid implementation, the main functionality of this class is contained in the MoveDirection() function, that yields a direction vector for him every frame (see figure 30).

```
List<Transform> radiusMemmory = new List<Transform>();
Vector3 MoveDirection()
{
    List<Transform> preyWithinRadius = new List<Transform>();

    foreach (GameObject prey in GameObject.FindGameObjectsWithTag("Prey"))
    {
        if (Vector3.Distance(this.transform.position, prey.transform.position) <= predtorVisionRadius)
        {
            preyWithinRadius.Add(prey.transform);
            if (!radiusMemmory.Contains(prey.transform)) radiusMemmory.Add(prey.transform);
        }
        else
        {
            if (radiusMemmory.Contains(prey.transform)) radiusMemmory.Remove(prey.transform);
        }
    }

    if (preyWithinRadius.Count < 2)
    {
        radiusMemmory.Clear();
        return (GameObject.FindGameObjectsWithTag("Prey")
            .OrderBy(t => (t.transform.position - this.transform.position).sqrMagnitude).FirstOrDefault()
            .transform.position - this.transform.position).normalized;
    } else
    {
        if (radiusMemmory.Count > 0) return (radiusMemmory[radiusMemmory.Count - 1].position
                - this.transform.position).normalized;
        else return Vector3.zero;
    }
}
```

Figure 30: Predator main functionality

The key in the functionality of the predator is in that he does not just chase the closest pray. Instead, in order to create a more complex behavior, the predator will

chase the closest pray, only if there is no other prey in his predefined vision radius. In case there are more than one flock members within the predator's vision radius, he will go after the last one that has entered his vicinity. This ensures a complex behavior that provides the prey a chance to escape, by moving closer to other flock members, thus 'confusing' the predator. Furthermore, with the predator being the main moving force behind the emerging behavior of the pray, this way of implementing him predisposes the flock to seek 'safety in the numbers', ultimately making the predator a catalyst for the flocking behavior. This approach to the predator implementation has been inspired by the examples presented in the 'State of the Art' section of the Analysis.

The way this behavior is achieved is by first checking which flock members are within the predators vision radius. This is done on two levels - first, with a simple list that aims to check the members within radius for the current frame and second, with a global list the contents of which are independent from the update cycle, essentially serving as a memory for the predator. The reason why this type of implementation was necessary, is because of the way the dynamic between the predator and the prey work. More specifically, this addresses the problem two or more flock members entering the vicinity of the predator simultaneously, thus resulting in skips in the chain of targets for the predator. Furthermore, if the predator's direction function is dependent on the frame rate cycle, it becomes difficult to account for the flock members that have temporarily exited the predator's vision and then have reentered it again shortly after. This would result in undesired and more importantly inconsistent behavior in the predator, which can not be allowed, considering that the training of the flock depends on the interaction with him. The 'radius memory' of the predator is updated in three cases. First, if a flock member is noticed to be within the vision radius, but is not currently contained in the memory, then he is added. Second, if a flock member is exits the predator vision radius and has been in the memory so far, then he is removed from the memory. Finally, if the predator memory exceeds 20 flock members at a time, it is flushed completely and the information from the current frame is added again. The last step is executed by a 'fail-safe' function that is ran independently from the main direction calculation function and the reason for it is to prevent situations in which the predator, being essentially part of the environment, begins to slow down the entire simulation due to excessive calculations performed at each frame. Furthermore, the predator class is optimized by using the System.Linq (Language Integrated Query) library, native to C#, for sorting the lists of prey and comparing the distances between them. Additionally, further optimization has been achieved, by replacing the Vector3.Distance function for the comparison between distances, with a manual calculation that makes use of the much more lightweight in terms of computational load .sqrtMagnitude function. To explain this into further detail, this is done by comparing the squared magnitudes of the position vectors of interest, thus bypassing the expensive square root operation embedded into the original distance function. As explained earlier in the Analysis, when dealing with large

number of agent instances to process through, combined with an 100-fold increased environment time scale during training time, these small solutions for reducing the computational load result into a great improvement in the achieved performance.

5 Evaluation

This section is going to describe the evaluation setting and the results produced with it respectively.

5.1 Evaluation setting

In order to evaluate the problem statement of this project as accurately as possible, the test process was designed so that it utilizes both quantitative and qualitative data. To be more specific, the test consisted of comparison between 3 different crowd implementations - two ML implementations using the same RL model but with different settings and one crowd implementation identical to the original flocking boid implementation. For the quantitative part of the test, the three implementations were compared using various numerical statistics that describe the behavior of the crowds. For the qualitative part, visual inspection and comparison was done upon each solution, implementing additional visual ques that helped to underline behavioral artefacts and other less visible characteristics.

Throughout the numerous implementation iterations of this project, several RL model versions were deemed successful in terms of satisfying the conditions described in the Methods section. Each different version was designed with the purpose to test a different approach, hypothesized to be the solution to the problem statement. The main variation between the different implementations was expressed primarily in the reward function of the agents, the hyper parameters of the NN and the environmental conditions. As explained earlier in the report, due to the nature of the emergent approach, even the slightest variation in the initial conditions of the environment can potentially lead to an entirely different and unforeseen outcome. This is also the main reason why this project requires an implementation process with such a high emphasis on iterating through different possible solutions.

Out of those implementations, two were chosen for comparison with the original flocking boid-derived crowd implementation. To go into further detail, the two chosen solutions are the one that got the closest in terms of behavior to the control environment and the one that had the best results in terms of training the ML model (highest reward achieved, highest learning rate, etc.).

To be more specific the main difference between the two chosen ML implementations is in the fitness function used for each algorithm (see table 1).

	Vector observations	**Ray cast observations**
ML model 1 (best training performance)	- agent position - agent velocity - agent orientation	18 rays with a total spread of 340 degrees
Ml model 2 (best achieved behavior)	- agent position - agent velocity - agent orientation - predator position - predator velocity - distance from predator - predator orientation	18 rays with a total spread of 340 degrees

Table 1: Reward systems used for the two ML models

In order to compare the three implementations in terms of quantitative data, a separate program was built, so that it keeps track of the metrics of interest produced by the performance of the algorithms. More specifically, the program monitored the following values:

- Distance from the average center of the nearest group of neighbours.
- Distance from the predator.
- Number of neighbouring agents.
- The error in terms of angular difference between the forward vector of each agent, describing his movement at the current frame and the vector describing each of the three native flocking boid behaviors as well as the perfect desired behaviors for finding food and avoiding the predator:
  - Cohesion - the direction vector towards the average center of the nearest group of neighbours.
  - Avoidance - the direction vector pointing away from the average center of all neighbours that trespass the avoidance radius.
  - Alignment - the direction vector that is the average result between the forward vector of the nearest group of neighbours.
  - Foraging - the direction vector pointing towards the nearest piece of food.
  - Predator avoidance - the direction vector pointing away from the predator, if he is within the agent's immediate vicinity (predefined predator avoidance radius).

Furthermore, the predator behavior was also recorded for comparison between the 3 implementations. More specifically, two metrics were extracted from his behavior:

- Amount of times the predator caught a prey.
- The predator memory size.

As explained in the Implementation, the predator complex behavior is achieved by storing the number of agents that are within his vision radius in a frame rate independent memory. Therefore, the higher the size of that memory is, the more times an agent managed to avoid being eaten, because the predator got confused and targeted another crowd member in the middle of the chase. Additionally, the agents have been implemented with a maximum velocity that approximately 20% smaller than that of the predator, ensuring that they can not regress into a behavior of endless chasing in circles.

Upon collecting the above metrics, each of them was averaged for the total number of agents used for the given simulation, in order to gain representation of the entire crowd. Furthermore, in order to avoid biases, the values were noted 100 times, each time recording after 100 consecutive frames and finally averaged over the total number of recordings. This is done as to ensure that situations where the agents are showing behavioral artefacts are accounted for in the final measurements. Same was done with the predator data. Finally, for each of the three tested models two versions were tested - one with 15 and one with 60 members of the crowds. The above described evaluation setup can be seen summarized in table 2.

	Test version I	Test version II
ML model 1 (best training performance)	15 agents	60 agents
ML model 2 (best achieved behavior)	15 agents	60 agents
Flocking boid-based crowd (control implementation)	15 agents	60 agents

Table 2: Evaluation setup

Furthermore, the reason why only 15 agents per crowd were used during training, brings us to the hardware setup used for conducting all of the above described experiments (see table 3).

CPU	Intel Core i5-5200U (2 core, 2.2 - 2.7 GHz, 3 Mb cache)
GPU	nVidia GeForce GTX 950m 2GB
RAM	2 x 4 GB DDR3 (1600 MHz)

Table 3: Hardware setup used for developing and testing the product

Due to the highly insufficient computational power provided by the used setup, the maximum number of ML agents that could be trained at once without suffering too high performance losses was 15. Unfortunately, this is a number that is highly limiting for the overall scale of the project. The reason for that is because the crowd number during training, defines what relative number of neighbours each agent is trained to work with during inference. Of course it should be noted that provided a high enough robustness of the trained model is achieved, theoretically the agents can simply be cloned to get a higher crowd count during real-time performance and their learned group behavior should translate to this more numerous model. This is exactly what was done with this project, as to reach the higher number of 60 agents for the second version of the simulations. However, in a perfect situation, the total count of the crowd members during training should match that of the crowd during inference. The reason for that is simply in the total number of neighbours that can be at once in the immediate vicinity of each agent. Hypothetically, having such a low total count that less than half of the crowd members can be completely surrounded at once (find themselves in a group) during training, means that the overall experience from which the model learns is that the average crowd member moves completely alone or with only a few neighbours. Given the small learning experience to which the training times of the compared models translate due to the computational insufficiency, this would unavoidably reflect on the final learned behavior by the model and therefore on the evaluation results. However, taking into consideration the current scale of the project, this does not in any way devalue the findings of the performed experiments.

The overall reason why a second more numerous version of each model was tested, was based on the observation during initial testing that the exhibited behavior does not change linearly with the higher number of agents introduced. This is due to the complexity of the environment and the high number factors contributing to how the model learns. Furthermore, a higher crowd members count was shown to be the desired version of one such product, as the majority of studied examples in the Analysis either achieve the desired emergent behavior with much higher numbers, or show that the general application of a practical crowd simulation usually entails a very large crowd.

Finally, for the visual inspection of the crowds, the following visual ques were implemented (see figure 31). All of the color values have been chosen specifically to contrast between the different elements of the environment.

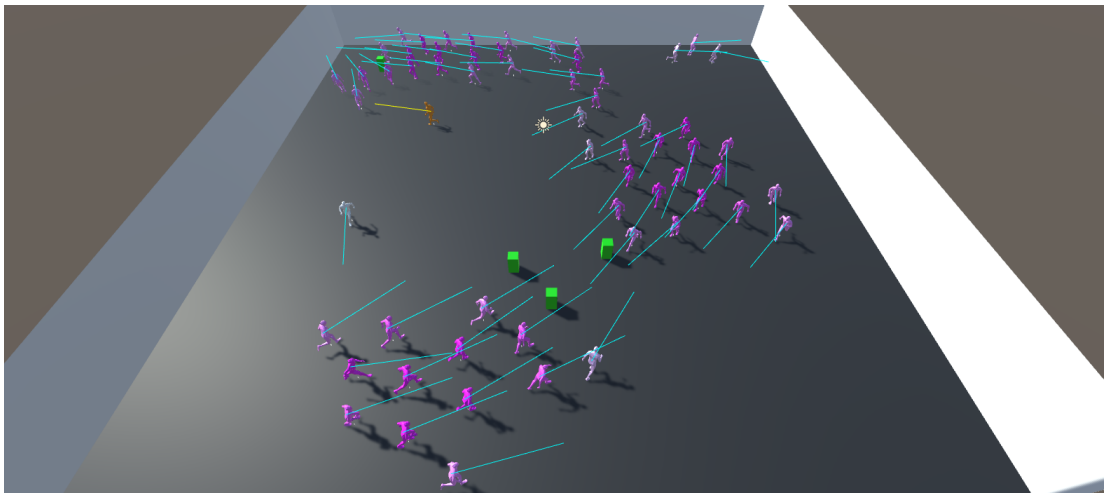


Figure 31: Visual ques used for the emphasizing the exhibited crowd behavior. Predator has both his avatar and forward vector orange. Crowd members are colored white or magenta, depending on the size of their local group and their forward vectors are colored in cyan. Food is colored green.

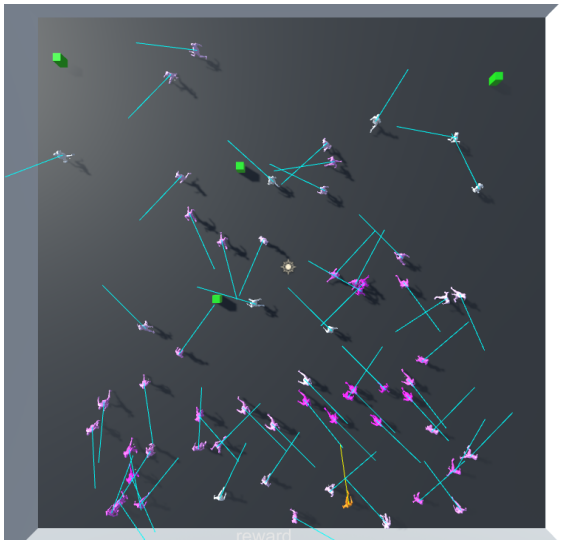
The purpose of the above described test setting is defined by the hypothesis that having the behavior rules essentially hard-coded, the flocking boid crowd implementation would have the most desirable results. It can then be used as a control environment to which the two ML implementations can be compared. The results from the above described experiment can be found in the next chapter.

5.2 Results

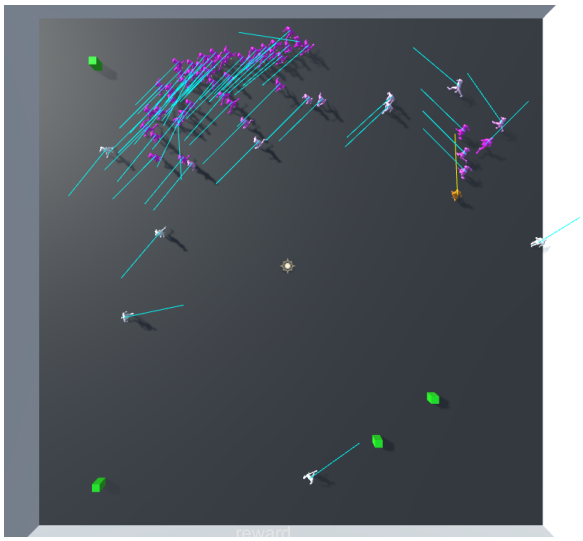
Each of the two ML models was trained for approximately 8 hours, equalling around 900 000 real-time steps for each agent. The training sessions were terminated after approximately the same time, in order to avoid differences in the training conditions. The number of food pieces used for the training and testing for each implementation was 4. This number was chosen through trial and error during the initial implementation iterations as well as using the information gathered throughout the research of other similar examples, as it provides the greatest balance between all of the other elements of the environment. The same process was used for the choice of terrain size, surface friction values, collider sizes, etc. Finally, both models used the same hyper parameters for their NNs.

The visual observation part of the test shows that ML model 2 was the one that was able to learn the best flocking boid-like behavior. However, the model seems to prioritize avoiding the predator, instead of balancing between foraging and avoidance. Due to that, the model fails to maximize its reward. Instead it reaches a state

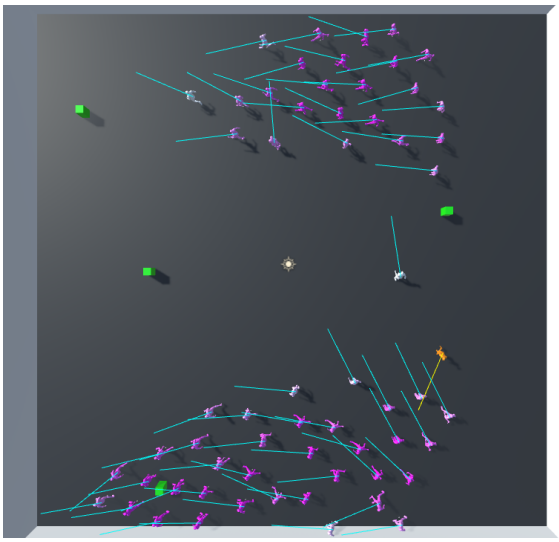
of dynamic equilibrium where the newly received negative rewards are minimized. ML model 1 shows the best training, meaning that the algorithm successfully maximizes its reward. Here, the model achieves a relative balance between foraging and predator avoidance. However, instead of prioritizing grouping, the individual agents of the crowd would prefer to deal with the predator on their own, mostly trying to outmaneuver him.



(a) ML model 1 (best learning performance)



(b) ML model 2 (best learned behavior)



(c) Control environment

Figure 32: Top-down view examples of behavioral performance of all three environments

Figure 33 shows the total average of the orientation angle errors for cohesion,

alignment, avoidance, foraging and predator avoidance for each of the three implementations tested with 15 members per crowd. ML model 1 (best trainer) shows to be the worst in terms of alignment, avoidance and foraging, followed by ML model 2 (perfect flocking behavior) and the control environment. This tendency however is reversed when it comes to the predator avoidance and cohesion. This can be further confirmed by the visual inspections of the three behaviors, showing that ML model 1 has the best maneuverability of the agents, therefore avoiding the predator with ease.

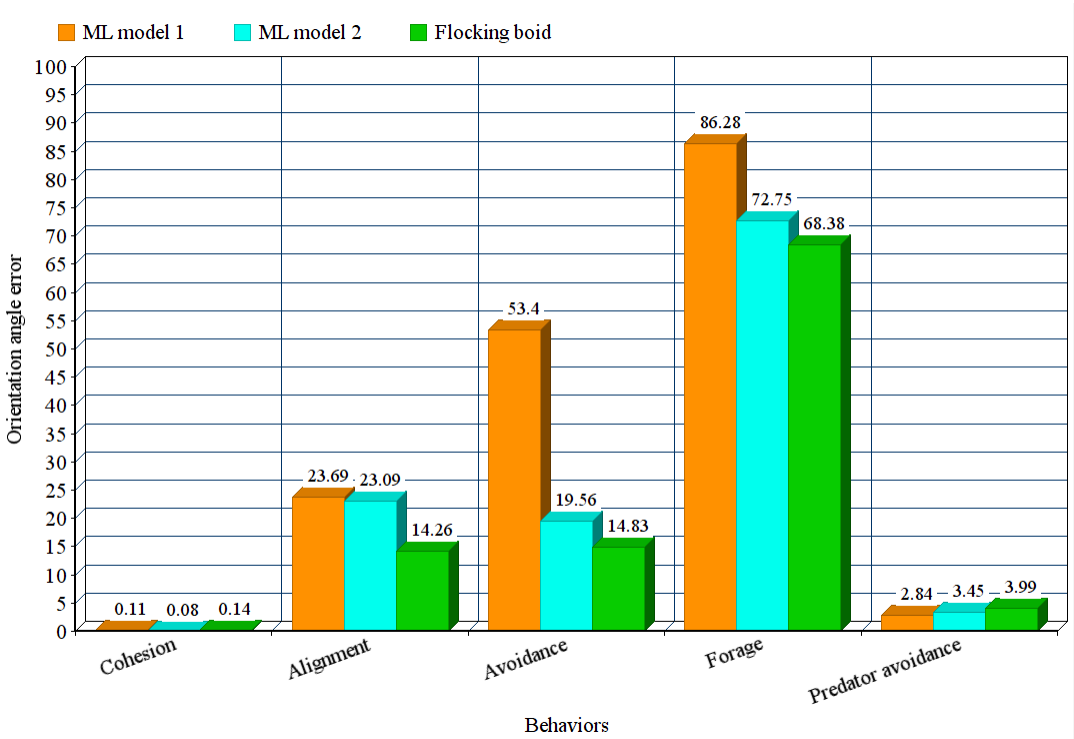


Figure 33: Orientation angle error for 5 behaviors with 15 agents per model - Cohesion, Alignment, Avoidance, Forage and Predator avoidance. The values for each behavior are listed for comparison between 3 models - ML model 1 (best learning model), ML model 2 (best flocking model) and original flocking boid

Figure 34 shows the same measurements, but for the test version with 60 agents. Here the control environment still performs best in terms of alignment, avoidance, foraging and predator avoidance, but the trend is shifted when comparing the two ML models. In avoidance, foraging and predator avoidance ML model 1 outperforms ML model 2, while the measurements for alignment show the opposite. Cohesion shows equal performance for ML model 1 and the control environment, while ML model 2 outperforms both.

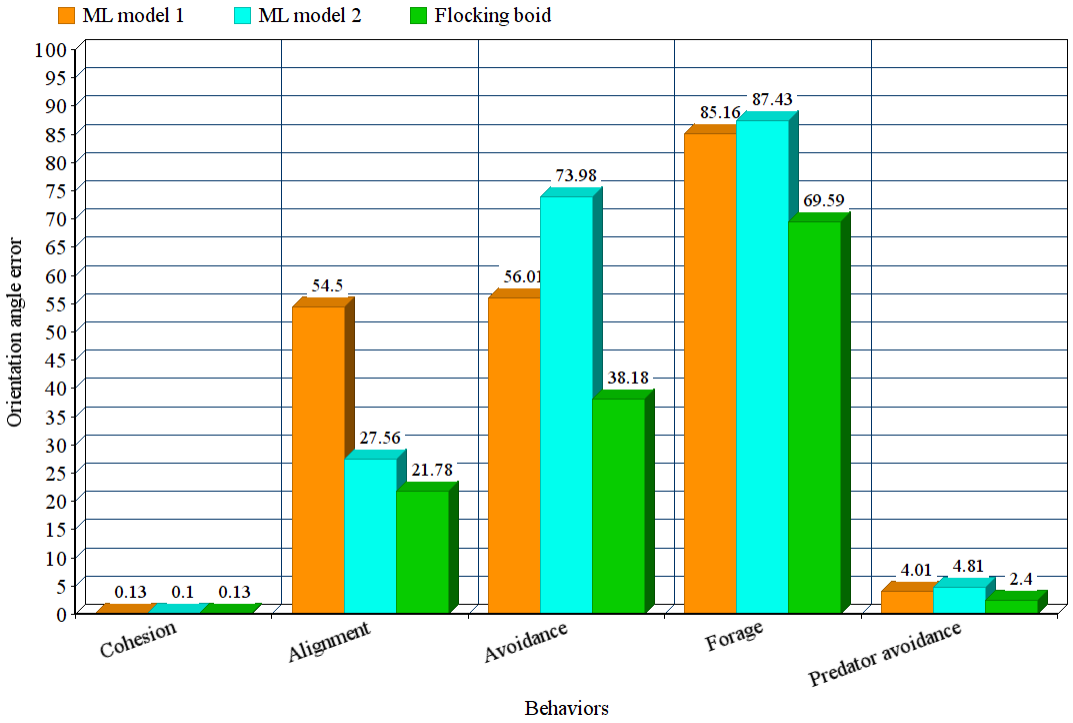


Figure 34: Orientation angle error for 5 behaviors with 60 agents per model

Figure 35 shows the grouping, predator distance and number of neighbours performance for the three implementations, in a test with 15 agents per group. Here ML model 2 outperforms both ML model 1 and the control environment in each category (notice that for grouping, a smaller distance means better performance, while for predator distance and number of neighbours, low values mean the opposite). ML model 1 does better than the control environment only in terms of predator distance, but the difference is too small to be considered significant.

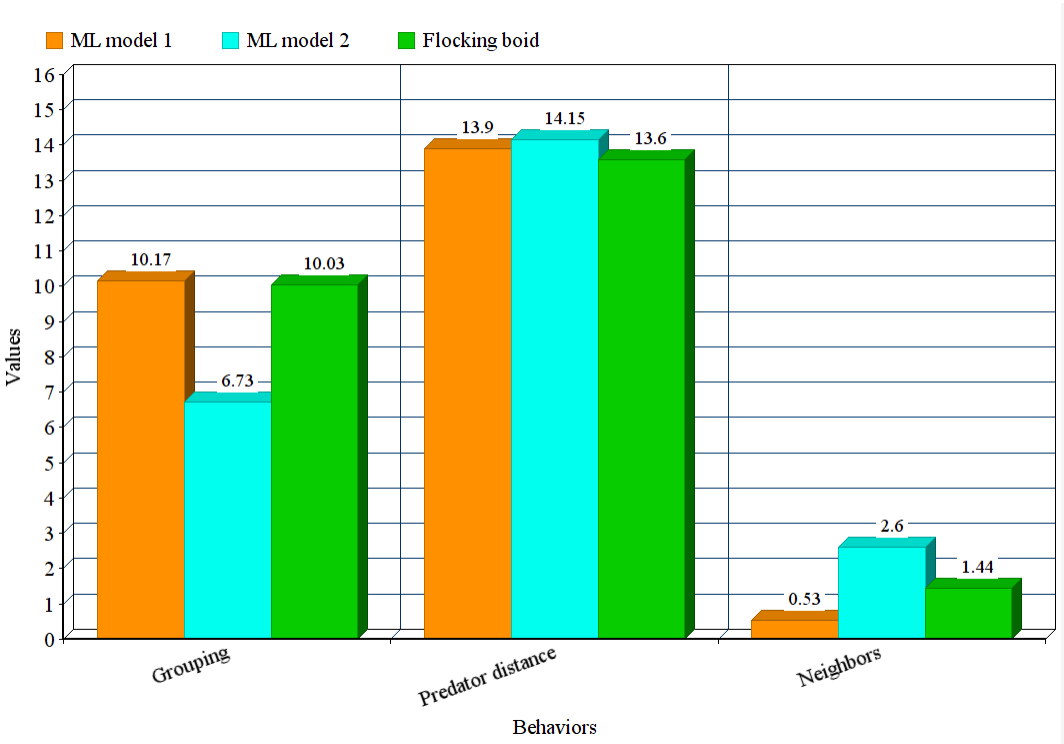


Figure 35: Additional metrics values with 15 agents per model. Presented metrics are Grouping, Predator distance, Number of neighbors

Figure 36 show the same but for the test with 60 agents. Here the exhibited trend is exactly the same as in the test with 15 crowd members, with the only difference being that in this case the control environment outperforms ML model 2.

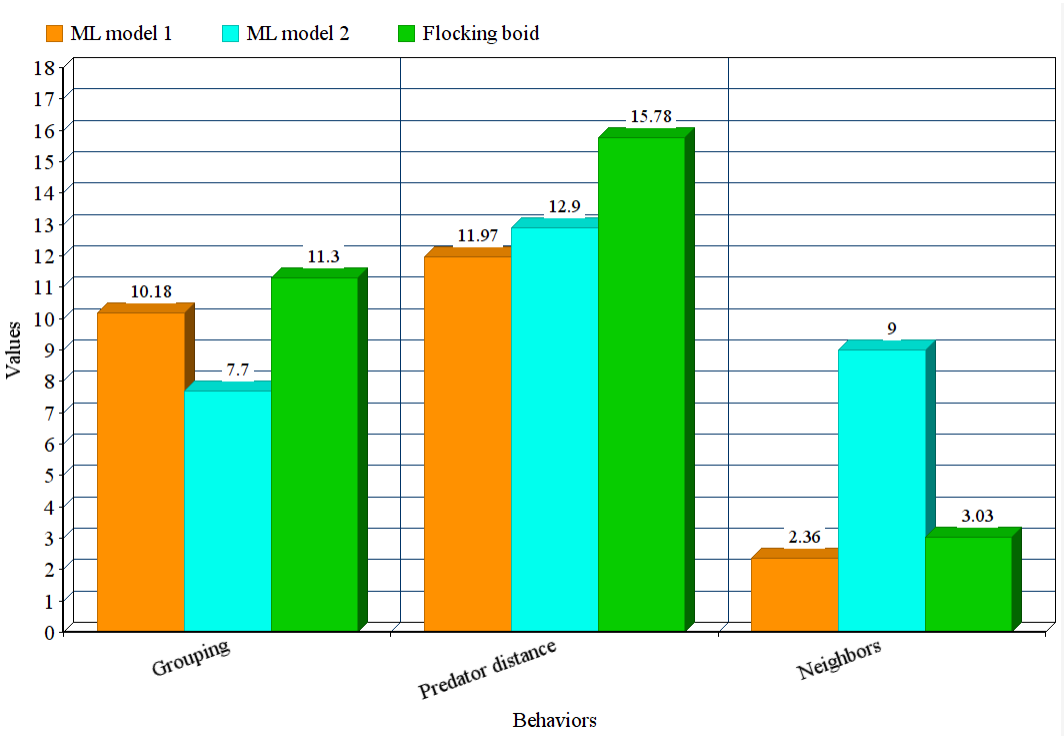


Figure 36: Additional metrics values with 60 agents per model. Presented metrics are Grouping, Predator distance, Number of neighbors

The following figure shows the cumulative reward of ML model 1's training session (best achieved training). Here the algorithm continues to increase its reward, stabilizing at around 1 000 000 steps. As it can be seen from the graph, this model shows relatively small fluctuation in performance, maintaining positive reward the entire session. However, as shown from the visual observations of the learned behavior, the model mostly fails to evolve toward grouping behavior.

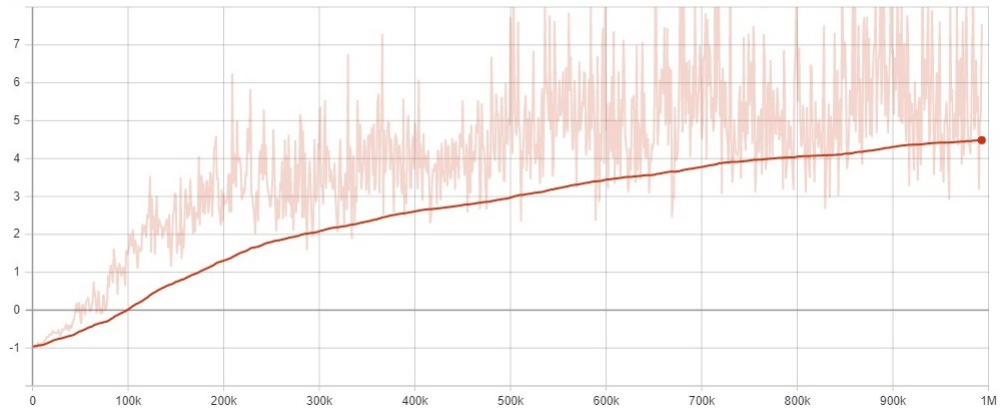


Figure 37: Cumulative reward over the total number of training steps for ML model 1 (best training session). Smoothing function has been applied due to high fluctuation

The next figure 38 shows the cumulative reward of ML model 2 for the entire training session. As mentioned earlier, despite of the near-perfect grouping behavior exhibited by the agents, this algorithm failed to as high of reward as the other presented ML model. With that being said however, the graph still shows some progress over time. The occasional spikes, followed by immediate drops indicate the efforts of the model to update its policy by attempting strategies that bring only short term success.

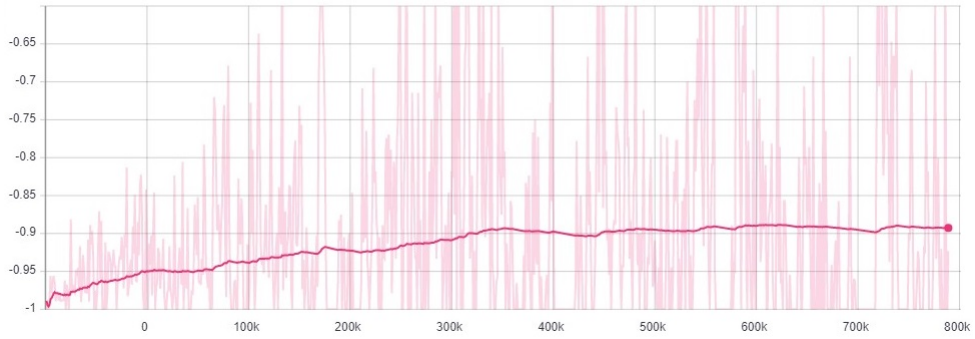


Figure 38: Cumulative reward over the total number of training steps for ML model 2 (best learned behavior). Smoothing function has been applied due to high fluctuation

The next figure shows the total number of times the predator caught an agent for the duration of each test session (see figure 39). As it can be seen from the graph, the difference in the 15 agents test version is quite small. However, the results show that ML model 1 has the least amount of casualties, followed by the control environment. The difference is increased in the more numerous version of the test, where the agents

from the control environment were the best at avoiding the predator and ML model 2 did the worst.

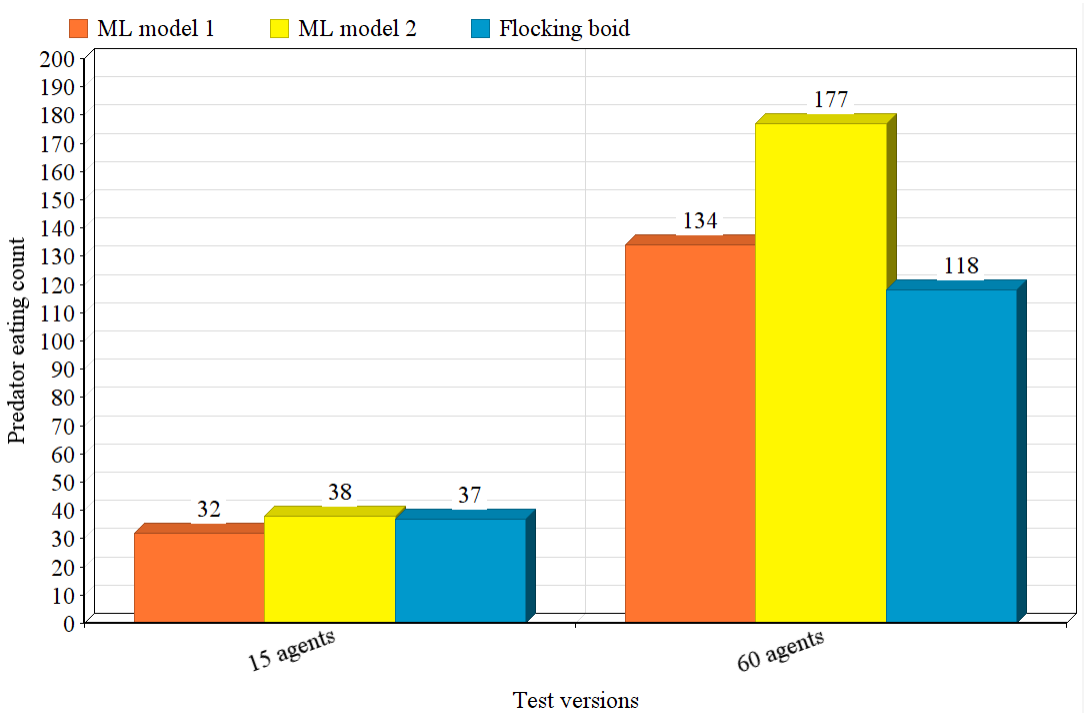


Figure 39: Amount of times the predator caught an agent for 10 000 steps, shown for each test version of the 3 tested implementations

Furthermore, figure 40 shows the average predator memory size for each test session. It can be seen from the graph that in the less numerous version of the test, the predator was most confused with the control environment and was mostly hunting single moving agents with ML model 1. This tendency is completely reversed in the second version of the test, where the predator was least confused in the control environment and most confused in ML model 2.

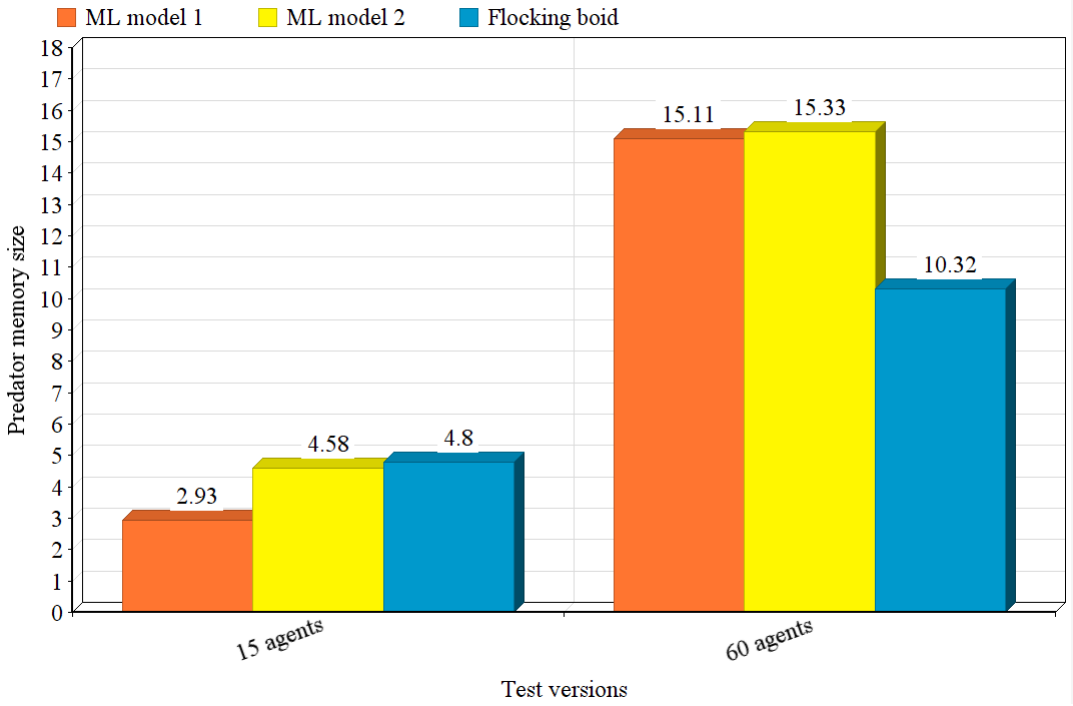


Figure 40: Average predator memory size, shown for each test version of the 3 tested implementations

Additionally, as the previous graphs as well as the visual observations show that ML model 2 solves the hypothesis by learning an excellent grouping behavior, the progress of the model has been sampled during training (see figures 41 and 42). The graphs show the progress for the orientation angle error of the model for cohesion, alignment, avoidance, foraging and predator avoidance. The way the presented data has been accumulated is by recording the performance of the algorithm every 50 000 steps of the training. Starting with alignment, it can be seen that the model increases the angular difference from the perfect behavior only once for a relatively short time during the entire session. The same can be said about the avoidance between neighbours, as the error peaks only once in the beginning of the training. It can be further observed that the model puts less priority in the foraging. Furthermore, predator avoidance is nearly perfect during the entire session. Cohesion is highly fluctuating, but note that due to the insignificance of the values, it is presented in another graph where it never goes above 1 degree error difference.

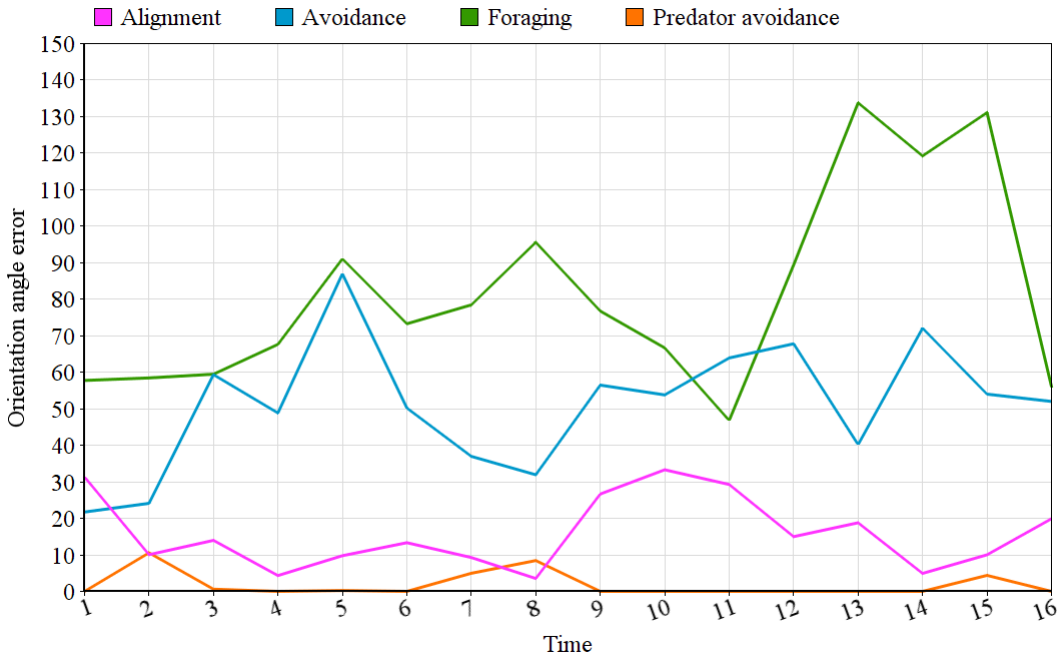


Figure 41: Progression during training of ML model 2 (best achieved behavior), shown for 4 orientation angle errors - Alignment, Avoidance, Foraging, Predator avoidance. One time step on the graph equals 50 000 training steps for the algorithm

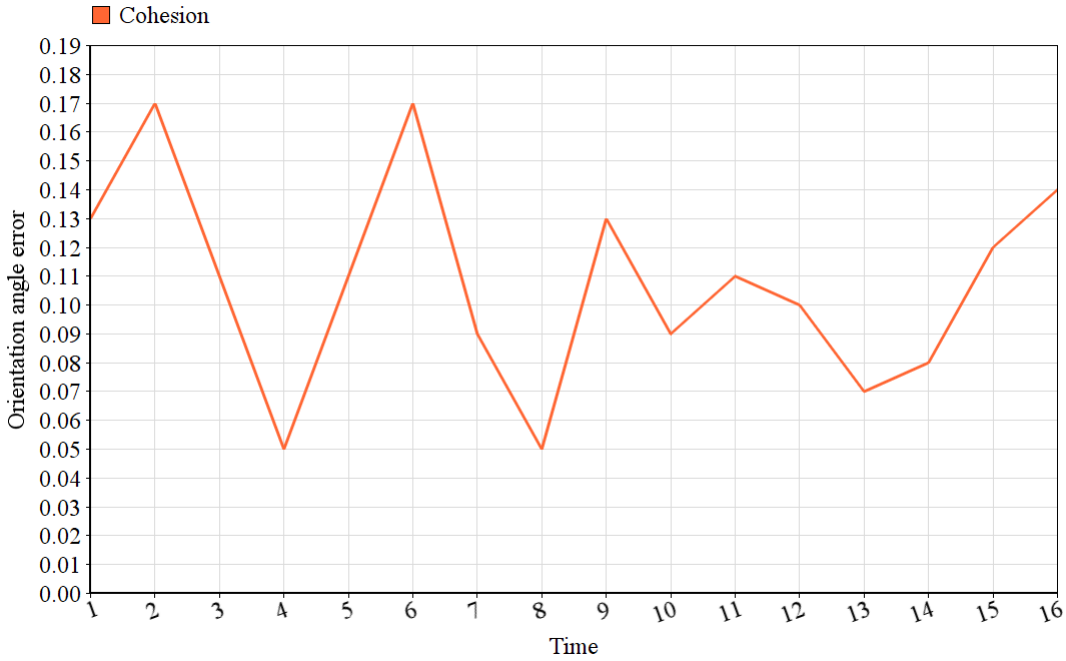


Figure 42: Progression during training of Cohesion orientation angle error. One time step on the graph equals 50 000 training steps for the algorithm

The last graph shows the progression of ML model 2 for 3 additional metrics - grouping, distance from predator and number of neighbours. While the distance from the predator is quite fluctuating, the distance from the local group center and the average number of neighbours are relatively stable throughout the session. Having an average number of neighbours of 3.5, shows that the algorithm keeps the majority of the agents surrounded in a group almost all of the time.

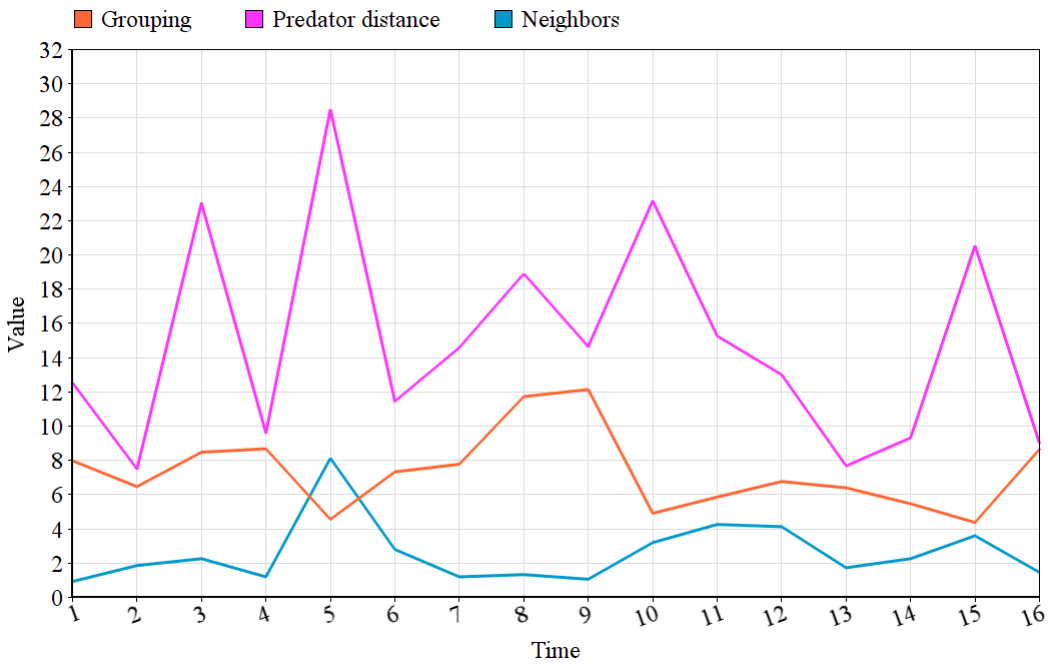


Figure 43: Progression during training of 3 additional metrics - grouping, distance from predator and number of neighbors. One time step on the graph equals 50 000 training steps for the algorithm

The findings from the above presented results will be described in the discussion.

6 Discussion

Based on the visual observations of the 3 implemented environments, it can be confirmed with certainty that ML model 2, which learned the best crowd-like behavior fulfils this requirement of the project's PS. During the research of the different key aspects utilized in these implementations (predatory threat, foraging, etc.), it was speculated that a natural evolution towards a grouping behavior will occur, provided the agents are given enough freedom of action through the rules of the environment. While this was also confirmed, the examination of the quantitative data gathered from the test, indicate that there are still some issues with the environments. For example, looking at the angular errors of the models in the two versions of the test, ML model 2 outperforms ML model 1 when dealing with 15 agents. Once the number is increased however, this tendency is shifted in the opposite direction. This, as speculated earlier, is due to the fact that the insufficient training caused by the hardware limitations is crippling for the performance of the less robust in terms of individual agent manoeuvrability ML model 2 when changes in the environment are introduced. However, even with these limitations, looking at the results for distance from local group center, distance from predator and average number of neighbours, ML model 2 outperforms ML model 1 even in the more numerous version of the test. What is more, in most of these criteria ML model 2 got better results even than the control environment, which can be further confirmed by the visual inspection.

For example, when looking at the average number of neighbours for the two test versions and comparing that to the neighbor avoidance behavior, it can be seen that the agents from ML model 2 became so proficient in navigating while in a densely packed group, that they no longer need to maintain the neighbour avoidance radius that is present in the control environment. This is important because each agent avatar is implemented with a certain amount of friction to its collider. Even though the friction values for all agents are minimized in order to avoid situations when they get stuck on each other, the way Unity handles friction is that when two colliders are moving while in contact with each other, they still experience a significant reduction in velocity. This was initially implemented with the idea to push forward the established tendency in the majority of other similar projects, which usually handle their crowd members as pass-through objects, independent of collisions. However, despite the danger of becoming slower and getting caught by the predator, the agents still prioritized the 'safety in numbers' approach. Looking at the predator distance this strategy can be confirmed to be beneficial, as it outperformed the other ML model in this category. However, when examining the amount of times the agents from the two models were caught by the predator, it can be seen that ML model 1 performed better. This can be further confirmed by the visual observations, as it is clear that here, despite of never evolving towards a group, the agents became really good at individually outmaneuvering the predator.

As described in the beginning of the Evaluation, the only difference between the

two ML implementations is in the amount of observations they receive. While the agents from ML model 1 were relying mostly on ray cast observations to perceive the world around them, the agents from ML model 2 had on their disposal several additional global vector observations. More specifically, the agents from ML model 2 could see the predator, even if he was not in the viewing distance of their observation rays. Furthermore, while ML model 1 learned to balance between foraging and predator avoidance by employing a strategy of independent behavior for each agent, ML model 2 reached a strategy that mostly prioritizes predator avoidance and individual agents rarely risk to break out from the group in search for food. Based on the results from the test, as well as visual observation of both the learned behaviors it can be hypothesized that this is due to the fact that agents that are completely surrounded in a group, have their ray cast vision obstructed by their immediate neighbours. In that situation, the agent has two options - either follow the movement of its neighbours, thus maintaining a group formation, or break out from the group in an effort to be able to see the potential threat from the predator and search for food. Therefore, the agents from ML model 1 who can rely only on their view radius, prefer to avoid their neighbours and move as much as possible, thus decreasing the chances of missing the incoming predator and increasing the chance to find food. On the other hand, the agents from ML model 2, even though having their vision obstructed, can still perceive the predator and can focus entirely on perfecting their movement in cohesion with the overall motion of the group.

In addition, it can be seen from the training progression statistics extracted for ML model 2, that the model excels in predator avoidance and alignment nearly throughout the entire training session. Even though the cohesion statistic shows high fluctuations, it should be noted that the reason why this behavior was presented in a separate graph is because the error there is really low. Therefore, these fluctuations in the margin of 0.20 degrees can be disregarded. One interesting observation from the progression graphs is that the food seeking behavior becomes increasingly worse over time, with the highest spike at the end of the training session. This confirms the so far collected observations, that this model prioritises predator avoidance and grouping, instead of looking for food. Furthermore, it can be hypothesized that had the training not suffered from the hardware limitations, the model can potentially learn to balance more efficiently between the two behaviors.

When looking at the predator statistics, one interesting finding to mention is that for almost all test sessions, the higher predator memory size does not translate to a smaller amount of agents eaten. This contradicts the initial speculations that the more the agents are able to confuse the predator, the safer they would be. However, the fact still remains that the agents form ML model 2 still prioritized grouping, despite of it being less efficient in the long-term. One way to explain this is that being in a group does prolong on average the life of each individual agent, without reducing the overall success of the predator. Inspecting the behavior of the agents, this can be further confirmed, as the most hunted down agents are primarily the

ones that break away from the group. Additionally, this is in correlation with the way predator satiation works in nature, as the 'safety in numbers' is a strategy that primarily benefits the individual group member and not the group as a whole.

As mentioned earlier, while ML model 2 is the implementation that got closest to the control environment, if looking only at the NN statistics of the model, it is not the one that performed best during training (for additional statistics from the training sessions of both models, see the Appendices section). However, given that the model still managed to reach the desired behavior, also shows the importance of the non-linear correlation between the design of the fitness function and the desired outcome. It can be hypothesized that this is mostly due to the relative unpredictability of ML models in general, especially when a larger emphasis is given on environmental stimuli (guiding the model's behavior through environmental catalysts such as food and predator) rather than extrinsic rewards. The reason why such large freedom was given to the ML models during training is exactly because of the main hypothesis of this project, that generalization instead of explicitness is the right approach towards an emergent behavior. What is more, it can be concluded that for the current project's scale, this hypothesis has been confirmed. That is to say, throughout the numerous initial iterations for finding the proper solution to the problem statement requirements, both explicit and implicit reward systems were tested. The final verdict is that over-specifying the fitness function in an attempt to narrow down the possible learning outcomes for the algorithm is not the right approach when looking to achieve emergent complexity out of given certain simple initial rules. Based on the numerous attempts, it can be further concluded that this is due to the very nature of how ML and more specifically RL models operate. To be exact, when designing an explicit reward system, the more specific its description is, the higher the risk for creating an exploitation hole. If such breakthrough opportunity exists, it can be said with nearly 100% certainty, that the agent will find a way to exploit it.

An example for that can be given with one of the earlier attempts for finding a solution to the PS. This attempt was based on the hypothesis that a singular agent can be thought the flocking behavior through imitation of the flock crowd. Then, the idea was that once this agent has perfected this behavior, he will be cloned, thus achieving an emergent crowd behavior. The fitness function for the agent was such that he would be rewarded the closer he follows the average center point of the nearest group of crowding neighbours. The way this was done was by first finding the distance between the position of interest and the position of the agent each frame and storing it. Then, upon making his next step, the agent would get rewarded either negatively or positively depending on whether or not he moved in the direction of the crowd center. Furthermore, the agent would be additionally penalized for every frame of not taking any action at all. In addition, the frequency of these reward signals also meant faster and more accurate learning, as the agent can rely on an immediate feedback for virtually every possible action he could take. This on theory leaves the agent with only one opportunity, which is to learn to follow the cohesion

center of his neighbours and align with their movement, thus maximizing his reward. However, upon several hundred thousand iterations of exploring the environment, the agent actually discovered a 'lazier' method for building up a positive reward, by predicting a sequence of positions where he can wait, while an approaching group of neighbours would 'bring' the position where he is supposed to go near him. Not only that, but the agent actually became so good at this, that he managed to maximize his reward by developing a highly optimized energy conservation strategy, mitigating the penalty for staying in one place by moving slightly from side to side. Despite of this being an impressive emergent behavior on its own, it is completely different from the desired behavior and an excellent example of how over-specifying can lead to undesired results.

Combining this experience with the findings from the current evaluation, it can be seen how the generalization of the initial rules of the system, translates to more opportunities for the development of nature-like strategies, the emergence of which in real life is not restricted by explicit set of guiding rules.

7 Conclusion

The purpose of this project was to create an emergent flocking boid-based crowd behavior through generalization of the system rules. Furthermore, this was to be achieved in a 3D Reinforcement learning environment, including predatory threat and foraging behavior of the implement crowd.

Summarizing the findings made upon the results from the evaluation, it can be concluded that the implemented ML model 2 fulfils the problem statement requirements. Both the quantitative and qualitative data indicate that the algorithm not only managed to evolve towards a crowding behavior, but it even surpassed the control environment in some of the measured criteria. However, the model still suffers certain limitations. As it was shown in the test results, the virtual crowd performs worse than the two other compared models in terms of seeking food, thus failing to maximize the explicit reward during training. Furthermore, an interesting finding was that despite the higher total suffered losses from the predator, the algorithm would still prefer the 'safety in numbers' survival strategy. This can be hypothesized to be a beneficial state of dynamic equilibrium between prolonging the life of the individual agent and the overall damage which the predator can inflict to the crowd as a whole. Finally, the performance exhibited by the presented crowd implementations show that complex emergent crowd behavior can be derived through generalization of the initial rules of the system and effectively guided through external environmental catalysts towards a certain desired result. However, further testing with longer training times is certain to show more satisfying results.

8 Future works

One area where there is certain need for future improvement is the hardware on which the proposed model was brought trough all the test iterations and eventually trained for the final evaluation. Due to the limitations imposed by the circumstances of the time of writing of this project, there was no availability for anything better than a low level consumer grade computer. It should be mentioned that the fact that the simulation was able to perform at its current level despite all the limitations, speaks even more well for the final product. However, it can also be stated with certainty that should the proper amounts of computational power has been available, the results would have been much more impressive. Furthermore, this is not only because of the fact that longer training times would have been achieved, but also because much more test iterations would have been possible. This is extremely important due to the high number of variables that can be tweaked in order to improve the performance of the system. Taking into consideration the possible amounts of fitness function and environment configurations, combined with the 23 available hyper parameters of the neural network, some of which taking an input range of over several thousands different values, achieving the desired outcome of the system through trial and error benefits from every single extra frame of performance.

Another aspect of the project that can be improved is related to optimization. As mentioned previously in the Implementation section, several optimization ploys have been utilized, in order to improve the performance of the simulation. However, there are more strategies that could have been implemented, should the project has taken place over a longer time period. One of these improvements is the division of the environment into smaller sectors and limiting the agents' input to the sector they are currently in. Such partitioning scheme would have definitely improved the performance of the simulation, as right now every single member of the crowd has to loop through all the other members.

In order to optimize the simulation even further, all repetitive calculations that slow down the system once multiplied with the number of agents, could be moved to a compute shader. The way compute shaders work, is by outsourcing calculations to the GPU, outside of the normal rendering pipeline. This allows for a large number of calculations that would otherwise be assigned to the CPU, to be done in parallel, thus significantly improving the overall performance of the given system. Unity engine does provide a way for compute shaders to be utilized, similar to the native to Microsoft Windows 'DirectX' API [24]. Combining these optimization techniques can vastly improve the run-time performance of the system during training, therefore allowing for far more experimental iterations to be conducted and better overall results to be achieved respectively.

In addition, the visual aspect of the simulation can be improved as well. While right now the program uses a fairly robust animation technique in combination with a 3D humanoid biped model for all of the characters present in the simulation, this

can be taken to a higher level. To be more specific, using a state of the art Motion Matching (MM) technique can vastly improve the visual fidelity of the animations, while also taking advantage of the already utilized ML-agents algorithm. To explain into further detail, MM is an alternative approach to standard virtual animations, developed by Ubisoft in 2015 for their new title at the time 'For Honor' [18]. The way MM works is by bypassing the traditional way of implementing animation through an animation state machine with predefined transitions and blends between clips and conditions. Instead, animations are constructed by choosing from the entire animation set at run-time, switching to any available pose based on the current one the character is in and the gameplay input. Each newly chosen pose is rapidly blended in with the ongoing sequence, thus leading to a much smoother and seamless transitions than with traditional animation. What is more, this technique can be improved even further, by utilizing machine learning for the choice of appropriate animation poses. This novel approach was proposed by Yousif Najim in his work 'Motion Matching with Proximal Optimization Policy' [31], where he used ML-agents in combination with PPO for achieving an innovative, far more optimized animations in Unity engine (see figure 44).

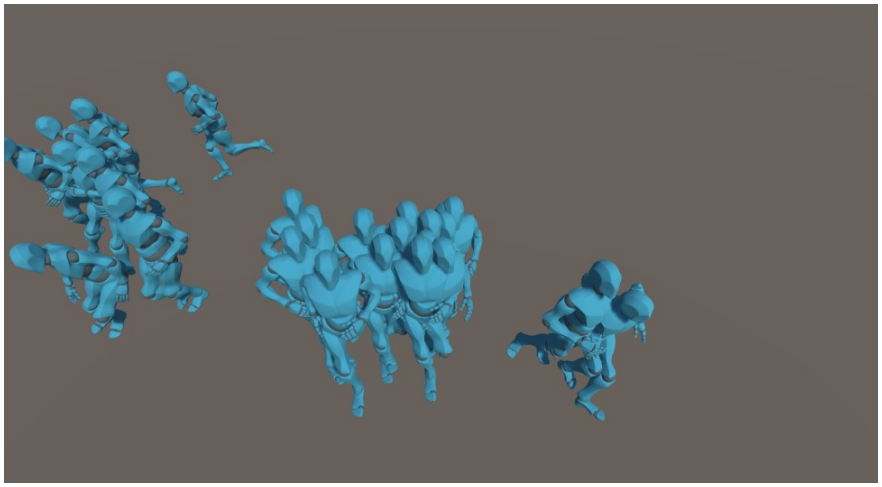


Figure 44: Multi-agent performance of Motion Matching model proposed by Najim[31]

Furthermore, one of the main differences between the product of this project and the state of the art examples presented in the Analysis, is that this emergent crowd is also implemented in a highly versatile 3D environment. This, as mentioned earlier in the report, creates a double utility for this application, as it can not only be used for scientific research purposes in the field of emergent behaviors, but also as a practical standalone utility that can find purpose in various industries. With that being said, one huge leap from all the other examples in the field of this project would be to

create a crowd that can navigate not only through all-flat areas, but also through terrains with varying altitudes, such as slopes, hills, mountains, etc. Incorporating the curvature of the terrain into the movement of the individual crowd members can potentially create an entirely different dimension of diversity to the overall crowd behavior.

References

[1] Andrew Adamatzky. *Game of life cellular automata*, volume 1. Springer, 2010.

[2] INC ANACONDA. Conda documentation, 2018.

[3] brackeys.com. Brackeys, 2020.

[4] Scott Camazine, Jean-Louis Deneubourg, Nigel R Franks, James Sneyd, Eric Bonabeau, and Guy Theraula. *Self-organization in biological systems*, volume 7. Princeton university press, 2003.

[5] Thomas Caraco, Steven Martindale, and H Ronald Pulliam. Avian flocking in the presence of a predator. *Nature*, 285(5764):400–401, 1980.

[6] Gabriel Chang and Michaela Stjerndal. Investigating and modeling the emergent flocking behaviour of sheep under threat with fear contagion, 2019.

[7] Ching-Shoei Chiang, Christoph Hoffmann, and Sagar Mittal. Emergent crowd behavior. *Computer-Aided Design and Applications*, 6(6):865–875, 2009.

[8] G Davis and B Far. Massive: Multiple agent simulation system in a virtual environment. *University of Calgary*, 2003.

[9] Pierre Degond, Laurent Navoret, Richard Bon, and David Sanchez. Congestion in a macroscopic model of self-driven particles modeling gregariousness. *Journal of Statistical Physics*, 138(1-3):85–125, 2010.

[10] Meilany Dewi, Moch Hariadi, and Mauridhi Hery Purnomo. Simulating the movement of the crowd in an environment using flocking. In *2011 2nd International Conference on Instrumentation, Communications, Information Technology, and Biomedical Engineering*, pages 186–191. IEEE, 2011.

[11] Veselin Efremov. Adam. In *ACM SIGGRAPH 2016 Real-Time Live!*, pages 17–17. 2016.

[12] Veselin Efremov and Adrian Lazar. Real-time procedural vfx characters in unity's real-time short film" the heretic". In *ACM SIGGRAPH 2019 Real-Time Live!*, pages 1–1. 2019.

[13] Unity Game Engine. Unity game engine-official site. *Online][Cited: October 9, 2008.] http://unity3d. com*, pages 1534–4320, 2008.

[14] Carsten Hahn, Thomy Phan, Thomas Gabor, Lenz Belzner, and Claudia Linnhoff-Popien. Emergent escape-based flocking behavior using multi-agent reinforcement learning. In *The 2018 Conference on Artificial Life: A Hybrid of the*

European Conference on Artificial Life (ECAL) and the International Conference on the Synthesis and Simulation of Living Systems (ALIFE), pages 598–605. MIT Press, 2019.

[15] Henry Hamburger. N-person prisoner's dilemma. *Journal of Mathematical Sociology*, 3(1):27–48, 1973.

[16] Christopher Hartman and Bedrich Benes. Autonomous boids. *Computer Animation and Virtual Worlds*, 17(3-4):199–206, 2006.

[17] Craig Hight. Making-of documentaries on dvd: The lord of the rings trilogy and special editions. *The Velvet Light Trap*, 56(1):4–17, 2005.

[18] https://forhonor.ubisoft.com/game/en us/home/. Ubisoft, 2020.

[19] http://www.basefount.com/miarmy.html. Miarmy, 2020.

[20] Brian Johnson, Marc Young, and Craig Skibo. *Inside Microsoft Visual Studio. NET*. Microsoft Press, 2002.

[21] Steven Johnson. *Emergence: The connected lives of ants, brains, cities, and software*. Simon and Schuster, 2002.

[22] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

[23] Halina Kwasnicka, Urszula Markowska-Kaczmar, and Marcin Mikosik. Flocking behaviour in simple ecosystems as a result of artificial evolution. *Applied Soft Computing*, 11(1):982–990, 2011.

[24] Frank Luna. *Introduction to 3D game programming with DirectX 11*. Stylus Publishing, LLC, 2012.

[25] Tyson Macaulay. *RIoT control: understanding and managing risks and the internet of things*. Morgan Kaufmann, 2016.

[26] Marco Mamei, Ronaldo Menezes, Robert Tolksdorf, and Franco Zambonelli. Case studies for self-organization in computer science. *Journal of Systems Architecture*, 52(8-9):443–460, 2006.

[27] Autodesk Maya. Autodesk maya, 2010.

[28] NJ Mills. Satiation and the functional response: a test of a new model. *Ecological Entomology*, 7(3):305–315, 1982.

[29] Jeffrey C Mogul. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.

[30] James E Morrow Jr. Schooling behavior in fishes. *The Quarterly review of biology*, 23(1):27–38, 1948.

[31] Yousif AH NAJIM. *Motion Matching with Proximal Optimization Policy*. PhD thesis, Aalborg University Copenhagen, 2019.

[32] E Passos, Mark Joselli, Marcelo Zamith, Jack Rocha, E Clua, Anselmo Montenegro, Aura Conci, and Bruno Feijo. Supermassive crowd simulation on gpu based on emergent behavior. In *Proceedings of the Seventh Brazilian Symposium on Computer Games and Digital Entertainment*, pages 70–75. Citeseer, 2008.

[33] Nuria Pelechano, Jan M Allbeck, and Norman I Badler. *Virtual crowds: Methods, simulation, and control*, volume 8. Morgan & Claypool Publishers, 2008.

[34] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

[35] Craig W Reynolds. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782. Citeseer, 1999.

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[37] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, 1994.

[38] Robert T Sullivan. Insect swarming and mating. *The Florida Entomologist*, 64(1):44–65, 1981.

[39] Yaniss Touahmi, Nikolay Burlutskiy, Kongwoo Lee, and Beomhee Lee. Congestion avoidance for multiple micro-robots using the behaviour of fish schools. *International Journal of Advanced Robotic Systems*, 9(3):67, 2012.

[40] Suiping Zhou, Dan Chen, Wentong Cai, Linbo Luo, Malcolm Yoke Hean Low, Feng Tian, Victor Su-Han Tay, Darren Wee Sze Ong, and Benjamin D Hamilton. Crowd modeling and simulation technologies. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 20(4):1–35, 2010.

Appendices

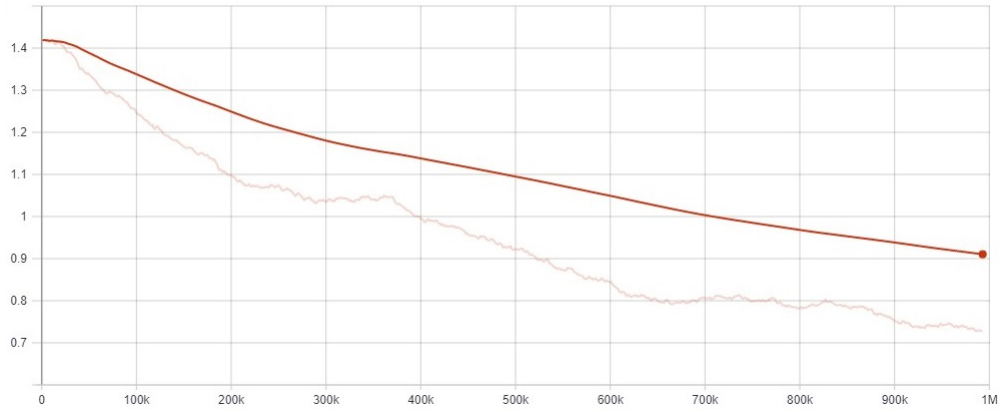


Figure 45: Entropy value over the total number of training steps for ML model 1

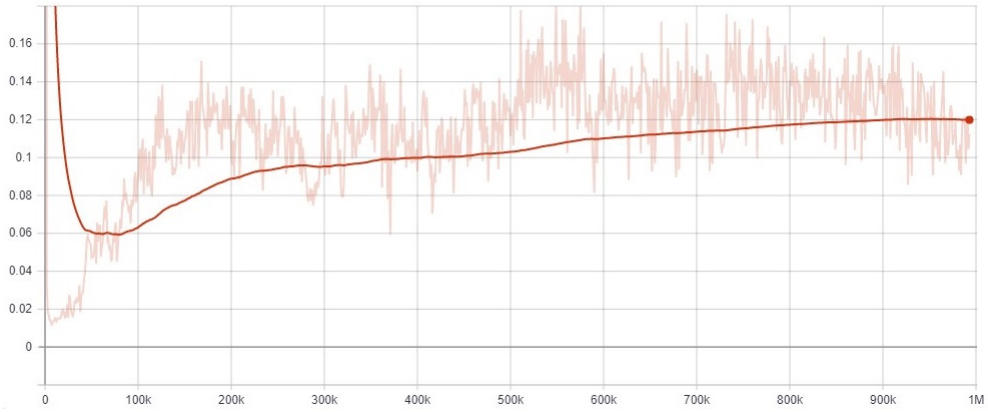


Figure 46: Value loss over the total number of training steps for ML model 1

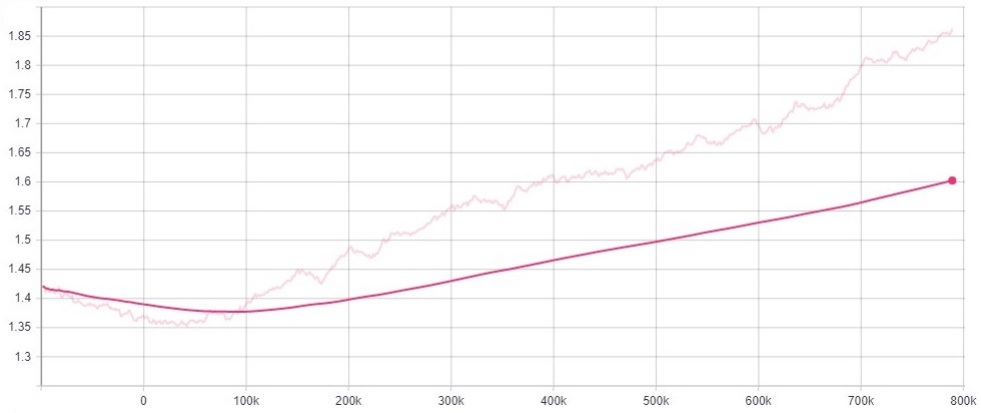


Figure 47: Entropy value over the total number of training steps for ML model 2

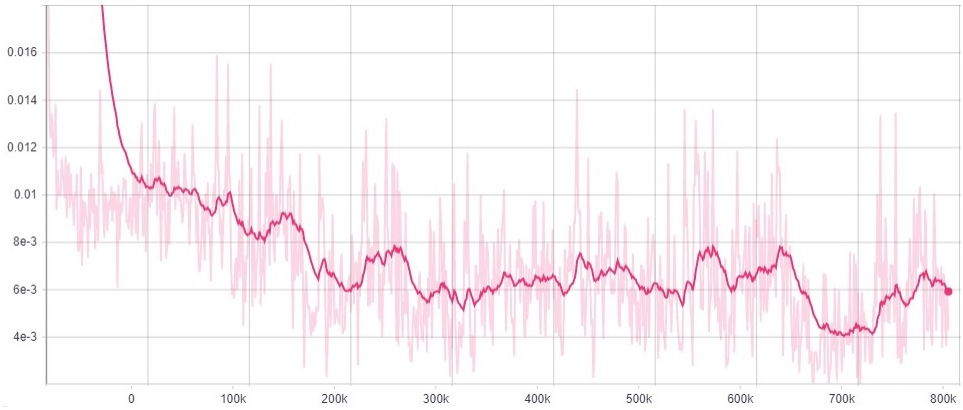


Figure 48: Value loss over the total number of training steps for ML model 2