AI game design in VR hunting

Zhao, Kangqiao

2021

Zhao, K. (2021). AI game design in VR hunting. Master's thesis, Nanyang Technological University, Singapore. <https://hdl.handle.net/10356/153784>

<https://hdl.handle.net/10356/153784>

<https://doi.org/10.32657/10356/153784>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 15:57:18 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

AI GAME DESIGN IN VR HUNTING

KANGQIAO ZHAO

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2021

AI GAME DESIGN IN VR HUNTING

KANGQIAO ZHAO

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Master of Engineering (Meng)

2021

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

17/08/2021
.....
Date

NTU NTU NTU NTU NTU NTU NTU NTU
NTU NTU NTU NTU NTU NTU NTU NTU
Kangqiao Zhao
NTU NTU NTU NTU NTU NTU NTU NTU
.....
Kangqiao Zhao

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

17/08/2021

.....
Date

NTU NTU NTU NTU NTU NTU NTU NTU
NTU NTU NTU NTU NTU NTU NTU NTU
NTU NTU NTU NTU NTU NTU NTU NTU
NTU NTU NTU NTU NTU NTU NTU NTU
.....

Supervisor: Dr Lin Feng

Authorship Attribution Statement

This thesis contains material from three papers accepted at conferences in which I am listed as an author.

- Chapter 3 is published as Kangqiao Zhao, Feng Lin, and Hock Soon Seah. "Steering autonomous animals in VR hunting." International Workshop on Advanced Imaging Technology (IWAIT) 2021. Vol. 11766. International Society for Optics and Photonics, 2021.
- Chapter 4 is published as Kangqiao Zhao, Feng Lin, and Hock Soon Seah. "Collective Intelligence of Autonomous Animals in VR Hunting." 2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW). IEEE, 2021.
- Chapter 5 is published as Kangqiao Zhao, Feng Lin, and Hock Soon Seah. "Reinforcement Learning for Quadruped Locomotion." Computer Graphics International, Geneva Switzerland (Online), 6-10 September 2021.

The contributions of the co-authors are as follows:

- Prof. Lin provided the initial project direction.
- The manuscript was revised by Prof. Lin and Prof. Seah.

17/08/2021

.....
Date

NTU NTU NTU NTU NTU NTU NTU NTU
NTU NTU NTU NTU NTU NTU NTU NTU
Kangqiao Zhao
NTU NTU NTU NTU NTU NTU NTU NTU
.....
Kangqiao Zhao

Abstract

In the scenario of a VR hunting game, behaviour and autonomy of in-game animals are crucial in minimizing users' uncomfortableness and maximizing their experience while gaming. AI design of a successful Non-Player Character(NPC) mainly depends on four major behaviour layers which include: Decision, Steering, Navigation and Locomotion. In this study, the objective is to focus on designing and implementing novel algorithms for each of these four layers to achieve an integrated behaviour system in order to allow these animal characters to interact with the environment in an intelligent and realistic way.

Specific and generic animal behaviour trees are designed and implemented on a typical predator animal with intelligent and cooperative consciousness in decision-making when interacting with a complex and ever-changing game environment. A novel physics-based rig simplification model: Constrained Ellipsoid Model (CEM) is proposed which allows steering inputs to generate force and torque driven movements. New adaptive Autonomous Animal Behaviour (AAB) algorithms are designed for animal agents to navigate around the environment, with a research focus on collective intelligence and tactical actions. Advanced strategies for a group of autonomous animals are developed in order to simulate a more realistic forest environment. Computational experiments and comparisons with animation results are presented, which show significant advantages over previous work. Forward and Backward Reaching Inverse Kinematics (FABRIK) is extended and used as the inverse kinematic solver to handle the locomotion of the autonomous animals. This Generic Quadruped Inverse Kinematic Model (GQIKM) presented is seamlessly combined with AAB algorithms so that the animals can perform all sorts of autonomous actions efficiently, supported with realistic and smooth animations. Finally, this inverse kinematic embedded motion control system is further improved

by extending Proximal Policy Optimization (PPO) and designing specific reinforcement learning algorithms in order to bring animal agents' level of intelligence further deeper down into the actual skeletal motion of bones and joints.

All the algorithms in each behaviour layer presented in this research can be used individually or integrally for autonomous behaviour and motion control of other general, bipedal, quadrupedal and multipedal characters not only in computer graphics but also in fields such as vehicle navigation or robotic steering etc.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisor: Prof. Lin Feng, for his patient guidance throughout my two years of Master's study. I would also like to thank Prof. Seah Hock Soon for his kind assistance on my research as well as preparation of the papers. Lastly and most importantly, I want to thank my beloved father and mother in China for always providing me with enormous financial and mental support even at such a great distance that separates us.

Contents

Abstract	iv
Acknowledgments	vi
List of Figures	x
List of Tables	xii
List of Abbreviations	xiii
List of Publications	xv
1 Introduction	1
1.1 Motivation & Research Objective	1
1.2 VR Hunting Game	2
1.3 Thesis Organization	3
2 Decision Layer - An Implementation of Behaviour Tree	5
2.1 Introduction	5
2.2 Background	6
2.3 Methodology	6
2.3.1 Finite State Machine	6
2.3.2 Behaviour Tree	8
2.4 Experimental Results	12
2.5 Summary	16
3 Steering Layer - A Constrained Ellipsoid Model	17
3.1 Background	17
3.2 Methodology - Constrained Ellipsoid Model	19
3.3 Summary	23

4	Navigation Layer - Autonomous Animal Behaviours	25
4.1	Basic Behaviours	25
4.1.1	Navigation	25
4.1.2	Pursuit and Fleeing	26
4.1.3	Hovering	28
4.2	Collective Behaviours	30
4.2.1	Group Animal Pursuit and Fleeing	30
4.2.2	Force Combination	34
4.3	Experimental Results	37
4.3.1	Navigation	37
4.3.2	Pursuit	37
4.3.3	Hovering	39
4.3.4	Group Hovering with Leader	39
4.4	Summary	42
5	Locomotion Layer - Generic Quadruped Inverse Kinematics Model Reinforcement Learning with Autonomous Animal Agent	43
5.1	Introduction	43
5.2	Background	44
5.2.1	Inverse Kinematics in Computer Graphics	44
5.2.2	Reinforcement Learning	45
5.3	A Generic Quadruped Inverse Kinematics Model	47
5.3.1	Model Basics	47
5.3.2	Constraints for a Walking Gait Cycle	48
5.3.3	Tuning and Generalization	53
5.4	Reinforcement Learning with Autonomous Animal Target	55
5.4.1	Proximal Policy Optimization (PPO)	55
5.4.2	Autonomous Animal Target Training	56
5.4.3	Experimental Results and Analysis	59
5.5	Summary	61

6	Conclusions and Future Directions	62
6.1	Conclusions	62
6.2	Future Directions	63
A	Source Code	67
A.1	AutoAnimal.cs	67
A.2	Navigate.cs	68
A.3	Pursuit.cs	68
A.4	Flee.cs	69
A.5	Hover.cs	69
A.6	Radar.cs	70
A.7	CollisionAvoidance.cs	70
A.8	LeaderFollowing.cs	71
	References	73

List of Figures

1.1	Four major components of an integrated motion behaviour system	2
1.2	Gameplay and Oculus Rift	3
1.3	Environment and initial animals	3
2.1	Decision layer	5
2.2	Finite State Machine of a NPC predator	7
2.3	A generic sub-tree - Flee	10
2.4	Behaviour tree - Wolf	11
2.5	Experiment scenario	13
2.6	Hovering wolf and deer	13
2.7	Pursuing wolf and fleeing deer	14
2.8	Hovering wolf and fleeing deer	14
2.9	Pursuing wolf and flanking wolf	15
3.1	A generic Constrained Ellipsoid Model of a terrestrial animal	20
4.1	Hovering mechanism	29
4.2	Group pursuit	30
4.3	Fleeing based on an average positional vector	32
4.4	Detective radar and repulsive force	35
4.5	An autonomous animal approaching its target	38
4.6	Pursuing the fleeing target with different distances	38
4.7	Hovering path with randomly located destination	40
4.8	Hovering path with Autonomous Animal Behaviour	40
4.9	Force combination: hovering with leader	41

5.1	Generic Quadruped Inverse Kinematics Model	47
5.2	Joint positions and types	48
5.3	Two phases of one leg during walking gait cycle	49
5.4	Swing: lift and fall	52
5.5	Quadruped walking gait cycle	53
5.6	Hover + Obstacle Avoidance + Flee	57
5.7	Trajectory of wolf agent and target agent during a training episode . . .	57
5.8	Cumulative reward: Vanilla Dynamic Target vs Autonomous Animal Target	59
5.9	Episode length: Vanilla Dynamic Target vs Autonomous Animal Target .	60
5.10	Episode length comparison	60

List of Tables

2.1	State transition matrix	7
3.1	Parameters of Constrained Ellipsoid Model	21
5.1	PPO training hyperparameters	59

List of Abbreviations

AAB	Autonomous Animal Behaviours
AI	Artificial Intelligence
BT	Behaviour Tree
CCD	Cyclic Coordinate Descent
CEM	Constrained Ellipsoid Model
CPMM	Constrained Point Mass Model
DDPG	Deep Deterministic Policy Gradient
DPG	Deterministic Policy Gradient
DoF	Degree of Freedom
DQN	Deep Q Network
FABRIK	Forward and Backward Inverse Kinematics
FK	Forward Kinematics
FSM	Finite State Machine
GD	Gradient Descent
GP	Gaussian Processes
GQIKM	Generic Quadruped Inverse Kinematics Model
IK	Inverse Kinematics
MANN	Mode Adaptive Neural Network
MERL	Maximum Entropy Reinforcement Learning
ML	Machine Learning
MoE	Mixture of Experts
MORL	Multi-Objective Reinforcement Learning
NN	Neural Network
NPC	Non-Player Character
NPCs	Non-Player Characters
PCA	Principle Component Analysis
PFNN	Penalty-Free Neural Network
PPO	Proximal Policy Optimization
RBF	Radial-Basis Functions
RL	Reinforcement Learning
RNN	Recurrent Neural Networks
RPD	Random Prioritized Dithering
RRT	Rapidly-Exploring Random Trees

SAC	Soft Actor-Critic
TRPO	Trust Region Policy Optimization
VDT	Vanilla Dynamic Target
VR	Virtual Reality
WGC	Walking Gait Cycle

List of Publications

International Conferences

- Kangqiao Zhao, Feng Lin, and Hock Soon Seah. "Reinforcement Learning for Quadruped Locomotion." Computer Graphics International, Geneva Switzerland (Online), 6-10 September 2021.
- Kangqiao Zhao, Feng Lin, and Hock Soon Seah. "Collective Intelligence of Autonomous Animals in VR Hunting." 2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW). IEEE, 2021.
- Kangqiao Zhao, Feng Lin, and Hock Soon Seah. "Steering autonomous animals in VR hunting." International Workshop on Advanced Imaging Technology (IWAIT) 2021. Vol. 11766. International Society for Optics and Photonics, 2021.

Chapter 1

Introduction

1.1 Motivation & Research Objective

In recent years, Virtual Reality (VR) hardware and technology are developing towards an increasingly mature stage. Compared with traditional 3D applications which mostly run on PC, general VR-based applications use a set of double-handed controllers for interactions between the user and the virtual environment. These controllers will be able to realize real-time tracking of the position of user's both hands and arms. Therefore, impacts of quality VR-based applications can be extended to novel fields such as patient body rehabilitation in health care, training of military, medical, manufacturing staffs and athletes, in addition to the traditional market of game industry. The fact that VR applications are highly immersive poses a greater challenge for computer graphics and Non-Player Characters (NPCs) control which are crucial in minimizing users' uncomfortableness and maximizing their experience during their time feeling the virtual world. Animation and motion control, behaviour adaptation and artificial intelligence of the NPCs should be studied and improved especially when implemented on a VR application [1].

An integrated motion behaviour system for any NPC (Non-Player Character) can be subdivided into four specific layers: decision, navigation, steering and locomotion. The first layer at the top of the hierarchy is for decision making. It generates a reasonable decision and goal [2, 3] for the animals based on the status of the environment and propagates the outputs down to the following layers for execution. The navigation layer is responsible for path finding of a trajectory that connects the animal's current location to

the targeted location, and avoiding collisions with any of the static terrain obstacles [4,5]. The steering layer is to move the character along that path, avoiding collisions with other dynamic obstacles due to environmental change and appearance of other NPCs and the player [6–8]. The final layer of the motion behaviour hierarchy is for locomotion, which converts the outputs of the steering layer into actual skeletal motion of bones and joints based on an inverse kinematics model to generate a set of smooth and realistic animations [9–14].

Figure 1.1: Four major components of an integrated motion behaviour system

The objective of this study is to generate a universal integrated motion behaviour system for NPCs with intelligence, awareness of team cooperation and realistic body joints movements, by achieving innovations and improvements based on previous algorithms in each of the four layers described above. The concept of Artificial Intelligence in Game Design is separately reflected on the autonomy of NPC agents in these four major layers. Compared with hard-coded behaviours, the proposed methods give these agents the ability to simulate real-time intelligence of a real animal.

1.2 VR Hunting Game

Based on this NPC motion behaviour system, a VR Hunting game is designed with the help of third-party application Unity3D (www.unity.com) and core tool packages such as SteamVR, VRTK, and MLAgent. VR headset and controllers: Oculus Rift, are used to run the game. Figure 1.2 shows a student playing the Hunting game using Oculus Rift VR headset and controllers.

This game is situated at a virtual forest environment with a large variety of wildlife NPCs. The player is equipped with a bow, unlimited arrows and able to freely move around the walkable areas of the terrain. The objective of the game is that player needs to hunt down as many animals as possible by firing his arrows. During this process, the player also needs to avoid being attacked by carnivore animals such as wolves and bears. Both the players and animal NPCs have respective health bars. Health bar decreasing

Figure 1.2: Gameplay and Oculus Rift

to zero will result in death and respawn. Figure 1.3 shows the game environment and all the types of initial animals.

Figure 1.3: Environment and initial animals

Some theoretical concepts introduced in this thesis can also be implemented on a traditional 3D application. When these methods are applied on VR Hunting, behaviour vraisemblance of NPCs becomes even more critical and apparent due to the highly immersive characteristic of VR environments.

1.3 Thesis Organization

The remaining part of the thesis is organized as follows. Chapter 2 describes the implementation of behaviour tree on designing the decision layer of the animal agents. Chapter

3 proposes a novel constrained ellipsoid model that allows animal NPCs to steer based on a physical approach. Following the steering layer described in Chapter 3, Chapter 4 combines this approach with steering behaviour algorithms so that animal agents can autonomously navigate themselves based on different tasks given. Chapter 5 completes the entire motion behaviour system by translating these movements into bones and joints of a generic quadruped animal agent using inverse kinematics algorithms and reinforcement learning. Finally, Chapter 6 concludes this integrated behaviour system and poses relevant discussions and future improvements.

Chapter 2

Decision Layer - An Implementation of Behaviour Tree

2.1 Introduction

The purpose of the Decision Layer is to give NPC the ability to generate its following decisions based on the information previously collected from exterior (environment) and interior (self-state). For the case of a wild forest hunting environment, NPCs' interactions with the player, its surroundings, and other NPCs will constantly trigger changes of its states. Some behaviours such as seeing the prey, colliding with an obstacle, getting shot by the player etc. will change the exterior states of the NPC while its interior states may include mood, fatigue, hungriness etc. The Decision layer plays a critical role in the integrated behaviour system simply because it is the first layer above all other layers. Behaviours will not make sense whatsoever if the incorrect decision is passed down for execution. Figure 2.1 illustrates the basic task of the decision layer.

Figure 2.1: Decision layer

2.2 Background

In artificial intelligence, robotics and game applications, the ability to make decisions autonomously is very critical in designing intelligent agents [15]. Finite State Machine (FSM) was a major approach addressing this issue. However, with the fast improvement of artificial intelligence, more complex decision structures of the agent have largely raised the demand for algorithm efficiency. Therefore, FSM was quickly obsoleted because of its deficiency in scaling, reusing, extension and adaptation.

The idea of Behaviour Tree (BT) was first raised in the field of game development. The core concepts of BT were dispersedly shared in partially documented blog posts and conference presentations. Until nowadays, it is still unclear who first proposed this algorithm. After Michael Mateas, Andrew Stern and Damián Isla [16, 17] sorted the previous appearances and standardized the key concepts, BT was since widely spread and rapidly refined into a proven technique. Since then, the first journal paper concerning BT was published [18] and the idea was quickly brought into applications of other fields such as game AI, combat training simulation, robotic manipulators, mobile ground robots, aerial and underwater vehicles [19–22].

Some major milestone papers in development of BT can be sorted as follow: Establishment of BT core concepts, comparisons, convergence analysis, stochastic analysis, parallel node analysis, multi-agent BTs [23–27] etc. In specific AI game types, BT has also been widely used in: real-time strategy games, first person shooting games, platform games, dialogue games [28–31] etc.

2.3 Methodology

2.3.1 Finite State Machine

Finite State Machines consist of a set of states, inputs, and functions which take previous state as input and output a new state. By utilizing a Finite State Machine, we can easily describe the decision layer of the agent by a set of states and conversion rules between them (FSM diagram or state conversion matrix).

Figure 2.2 and Table 2.1 describe an example decision layer of a typical NPC predator.

FSM is a very convenient approach toward designing simple decision layers of a certain NPC. It can be easily realized by *switch & case* statements.

Figure 2.2: Finite State Machine of a NPC predator

Table 2.1: State transition matrix

Current State	Input	Output State
Attack	Sees prey	Pursuit
Attack	Lost prey	Hover
Attack	Low health	Flee
Pursuit	Reachable	Attack
Pursuit	Lost prey	Hover
Pursuit	Low health	Flee
Hover	Sees prey	Pursuit
Hover	Low health	Flee
Flee	Healthy	Hover
Flee	No health	Die

2.3.2 Behaviour Tree

Although FSM is a very convenient and easy way to deal with designing easy decision layers, coding workload becomes very tedious when facing more complex decision logics. The most critical issues of FSM are that it can hardly be reused, maintained and modularized.

Behaviour Tree is a widely used method to handle plan execution in fields such as AI games design, robotics, control systems etc. Instead of using *States* to describe in-game logic, Behaviour Tree uses *Tasks* as nodes. When NPCs need to decide which actions it should take on the next step, it will traverse the Behaviour Tree from the **root** node. During this process, parent nodes will decide which and how child nodes are executed based on their own types. After execution, child nodes will return the result back to the parent node. Traversal sequence of the Behaviour Tree follows the rule of a depth-first spanning tree. There are 3 general types of node in a Behaviour Tree: **root** node, **composite** node and **execution** node. The **root** node is the first node with no parent node. Nodes with no child node are called the **execution** node and the rest are called **composite** nodes.

Execution node is subdivided into *condition* node and *action* node. *Condition* node, which is equivalent to an *if* statement, is used to verify whether certain condition is fulfilled. In the case of a predator NPC in the Hunting game, these conditions can be: “is prey in sight range?”, “is prey in reach range?”, “is myself healthy?” etc. Based on the verification, the *condition* node will output “*Success*” or “*Failure*” to its parent node. *Action* nodes complete actual tasks such as: “play animation”, “play sound”, “steer character”, “restore health” etc. In this section, all the *condition* nodes are painted in red with condition text followed by “?” and *action* nodes painted in green.

Composite nodes are used to control the traversal rule of the Behaviour Tree. The two basic types of **composite** node are *selector* node and *sequence* node. The *selector* node will execute its child nodes from left to right. If the child node returns “*Failure*”, the *selector* node will proceed to execute the next child node until one of the child nodes returns “*Success*” or “*Running*”. If so, it will also return “*Success*” or “*Running*” to its parent node. If all the child nodes return “*Failure*”, it will also return “*Failure*” to its parent node. In contrast, Sequence node traverses its child nodes in a similar way

except the “*Success*” and “*Failure*” logic is reversed. Algorithm 1 and 2 describe this key concept.

Algorithm 1 Selector

```

1: for  $i$  from 1 to  $N$  do  ▷ Traverse all  $N$  child nodes starting from the leftmost child
2:    $state \leftarrow Tick(child(i))$ 
3:   if  $state = Running$  then
4:     return Running
5:   if  $state = Success$  then
6:     return Success      ▷ If one child returns Success, Selector returns Success
7:   end
8: end
9: return Failure      ▷ Since all the child returned Failure, Selector returns Failure

```

Algorithm 2 Sequence

```

1: for  $i$  from 1 to  $N$  do  ▷ Traverse all  $N$  child nodes starting from the leftmost child
2:    $state \leftarrow Tick(child(i))$ 
3:   if  $state = Running$  then
4:     return Running
5:   if  $state = Failure$  then
6:     return Failure      ▷ If one child returns Failure, Sequence returns Failure
7:   end
8: end
9: return Success      ▷ Since all the child returned Success, Sequence returns Success

```

Taking the Hunting game for instance, if an in-game predator character has found its prey (player or prey in sight range), it will have multiple action choices to take such as: seek for comrades for a collective group pursuit or solo pursuit. The predator should first check if there are comrades in sight range, if the *condition* node returns “*Success*”, then proceed to seek for its comrades. Otherwise, the predator will start to proceed solo pursuit. In this case, *selector* node will be able to define the priority rule between these possible actions. The leftmost node has the highest priority and the rightmost node has the lowest priority (Algorithm 1). In the chapter, all the *selector* node will be drawn as a circle with a “?” inside.

Sequence node will try to execute all the leaf nodes from left to right. If a leaf node returns “*Success*”, *sequence* node proceeds to execute the next leaf node until a

node returns “*Failure*”, which will instantly lead to *sequence* node returning “*Failure*”. If all the leaf nodes are successfully executed, *sequence* node then returns “*Success*” (Algorithm 2). In this chapter, all the *sequence* node will be drawn as an arrow pointing toward right.

Instead of defining all the complex conditional transactions between each individual state, behaviour tree connects all the conditions and states by only using *sequence* and *selector* nodes. The biggest advantage of defining the decision layer using behaviour trees is that coding becomes much more modularized and self-consistent. Comparing to only one type of decision layer design, Hunting game consists of different types of agents such as wolf, deer, bear etc. who share many similar attributes. Instead of rewriting a finite state machine for every single type of NPC, workload can be largely decreased by designing and sharing generic sub-trees. Figure 2.3 shows a generic fleeing sub-tree that can be reused to design different types of agent.

Figure 2.3: A generic sub-tree - Flee

The behaviour tree shown in Figure 2.4 describes the decision layer of the most typical intelligent NPC in hunting game: Wolf. Algorithm 3 describes the traversal sequence of its decision BT.

Figure 2.4: Behaviour tree - Wolf

Algorithm 3 Wolf agent decision BT traversal sequence

```
1: if Target in sight then
2:   if Health level low then
3:     if Target is player then
4:       FLEE()
5:     else
6:       HOVER()
7:   else
8:     if Other wolf agents are pursuing the target then
9:       FLANK()
10:    else
11:      PURSUIT()
12: else
13:   if Wolf leader is in sight then
14:     FOLLOW()
15:   else
16:     HOVER()
```

2.4 Experimental Results

In this section, some typical interactions between agents are shown to demonstrate the basic operating principles of the behaviour tree of the wolf agent.

As shown in Figure 2.5 and Figure 2.6, the scene consists of a wolf agent and a deer agent. The yellow sector demonstrates the sight range of the wolf agent, and the yellow circle demonstrates the detection range of the deer agent. At the beginning, both agents execute the hovering behaviour and randomly choose their following destinations. The track of the wolf agent is coloured in red, and the track of the deer agent is coloured in blue.

When the deer agent enters the sight range of the wolf agent, this event immediately triggers the *sequence* node of the wolf agent to traverse to the next **execution** node, which drives the wolf agent to set the deer agent as the new target and start pursuing. This procedure is similar for fleeing execution of the deer agent. Figure 2.7 shows this triggered event.

The next experiment is a demonstration of how an injured wolf agent react to a deer agent under the same case scenario. As shown on Figure 2.8, although the deer agent

Figure 2.5: Experiment scenario

Figure 2.6: Hovering wolf and deer

Figure 2.7: Pursuing wolf and fleeing deer

has entered the sight range of the wolf agent, its behaviour tree stays in hovering state while the behaviour tree of the deer agent traverses to fleeing state. The track of the wolf agent is coloured in red, and the track of the deer agent is coloured in blue.

Figure 2.8: Hovering wolf and fleeing deer

In this last experiment, the flanking decision of the wolf agent is demonstrated. This scene consists of two wolf agents and one deer agent. The first wolf agent is under pursuit state with the fleeing deer agent as its target. When these two agents enter the sight range of the second wolf agent, instead of directly heading toward the current position of the deer agent, the second wolf agent chooses to flank its target by predicting the future position of the deer agent since it observes that another wolf agent is currently in pursuit of the same target. It should be noted that in this case scenario, once the wolf agent is running pursuit state or flanking state, the behaviour tree will continue to execute the current state until “*Success*” or “*Fail*” of the current task is returned (deer agent caught or fled) because these states here are set to be completion-prioritized. Therefore, even when the flanking wolf agent enters the sight range of the pursuing wolf agent, the pursuing wolf agent will continue to run the pursuit state instead of also switching to flanking state. The Figure 2.9 shows a demonstration of the collective intelligence of two wolf agents cooperating with each other while hunting their target. The tracks of the wolf agents are coloured in red, and the track of the deer agent is coloured in blue.

Figure 2.9: Pursuing wolf and flanking wolf

2.5 Summary

The first step toward a truly intelligent NPC is that the agent is required to be able to make the correct decisions under any circumstances by itself. The motivation of defining Decision layer is to give autonomous agents the ability to generate its decisions based on information collected from the exterior environment and its interior states. This is critical because correct decisions need to be passed down to the following layer to generate reasonable behaviours. The contributions in this chapter include: different types of existing decision-making methods are compared and evaluated, general and specific decision structures of different types of animal agents is defined and implemented, experimental results are given and proven to be concise, efficient and accurate.

In all the experiments from this chapter, the steering and navigation mechanisms applied on NPCs to achieve movements and tracks are designed based on a very basic translation of coordinates with brainless NPCs not knowing what sort of action it is performing while steering and navigating around the environment. In the next chapter, new steering algorithms will be introduced that allow NPCs to self-drive themselves when given specific tasks such as: seek, hover, pursuit etc.

Chapter 3

Steering Layer - A Constrained Ellipsoid Model

3.1 Background

Navigation and Steering has long been a popular research topic in computer graphics and robotics. The goal of which is to find the most suitable algorithm for the character of study to generate an optimal path between its current location and the destination location while steering the character along that trajectory [32]. In this thesis, the proposed methods will be separately introduced in Chapter 3 and Chapter 4.

One of the popular approaches towards path planning is based on environment modelling. The pre-processing step of modelling the environment is required before an optimal route can be found for the character. During this process, the walkable area of the terrain is represented in navigation meshes and visibility graphs. In the earlier days, E. W. Dijkstra [33] introduced the first leading graph traversal algorithm that finds the shortest path between two nodes on the graph. Based on that, Hart later introduced the famous A* pathfinding algorithm for further improvement [34]. Nowadays, frequently used algorithms that apply this kind of graph traversal approach are based on these two algorithms and their variants [35]. They are most likely able to find an optimal solution and easy to implement. However, the downsides of this approach are: relatively slow execution time, inefficient when dealing with dynamic obstacles and other agents thus unsuitable for real-time simulation, requires pre-processing of an appropriate mesh graph.

The work in this chapter has been published in "Steering Autonomous Animals in VR hunting".

Similar to A^* and Dijkstra that directly searches for the entire optimal path before steering, the RRT (Rapidly-exploring Random Trees) algorithm that is first introduced by Steven LaValle takes a different approach by not relying on environment modelling [36]. Pathfinding and terrain construction is processed simultaneously by expanding a tree from the starting position based on randomly sampled data of the space until the distance between the current position and the target position reaches the threshold. RRT based approaches generally has a very short execution time and efficient in finding the optimal solution. One critical downside of this algorithm is that it is not able to quickly react to unreachable target.

Recently, more and more self-adaptive approaches that orient towards autonomous and intelligent agents have been introduced. Path planning using reinforcement learning has been one major approach toward navigation of the agent. The agent is first trained with a set of reward functions that tell him whether the action took in each state receives a positive or negative feedback from the environment. For simple tasks such as moving towards the target while avoiding the obstacles, such reward functions can be set based on whether the distance between the agent and the target is shortened or lengthened, or whether the agent hit an obstacle during his exploration and training in the environment. For more complex tasks, advanced methods need to be applied such as Multi-Objective Reinforcement learning (MORL) for agents that need to achieve a set of tasks which sometimes may even be mutual conflicting [37, 38]. Other methods such as social choice theory is used for training a group of agents. The preferred action of the entire group in the current state is determined similarly to holding an election where each individual agent vote for its preferred action [39].

An interesting approach toward navigating and steering self-adaptive agents, which is based on rigid body dynamics, is by exerting a force. For instance, many researchers has studied the influence of "social force" on simulating a realistic crowd evacuation scenario [40, 41]. Reynold introduced the concept of steering force on autonomous vehicles to achieve a set of various behaviours [8]. The advantages of these rule-based approaches is that they are very efficient and easy to implement, does not require any pre-processing steps such as environment modelling and agent training, and suitable for real-time simulation. However, since the agent is assimilated as a particle with mass to incorporate

with force-driven displacement, in certain scenarios, the movement and heading direction of the agent will seem to be unrealistic [42].

3.2 Methodology - Constrained Ellipsoid Model

A physical object has a non-zero radius, mass and moment of inertia. The shape of an ellipsoid can approximate the rig of most terrestrial animals, such as bears, as illustrated in Figure 3.1. Of the ellipsoid model, only linear motion due to external force and angular displacement due to steering torque need to be considered, and Newton's laws of motion become applicable [43]. Fundamental properties of a rigid body are its position $\mathbf{P}$ (centre of mass), mass $\mathbf{m}$, velocity $\mathbf{v}$ and acceleration $\mathbf{a}$. The only external force in this study is the steering force $\mathbf{SF}$ which is generated with specific algorithms. In reality, the steering force $\mathbf{SF}$ is mainly provided by the friction of the ground and the magnitude of the friction is dependent on how much thrust the animal itself is able to generate. Hence, it is limited by the animal's current physiological state. In addition to limitation on the steering force, there is also a limitation on the maximum speed the animal can reach because in reality besides the ground friction that provide finite acceleration, animals are also subjected to air friction and momentum of its reciprocating parts resulting in deceleration. In order to embody these limitations, a rational way is to define the constraint parameters *maxForce* and *maxSpeed*. The range of these parameters varies based on different animal species [44].

The position $\mathbf{P}$ of the centre of mass of a rigid body with respect to a global orthogonal reference frame $O-xyz$ can be denoted as a 3-basis vector: $\mathbf{p} = [P_x, P_y, P_z]^T$. Similarly, velocity $\mathbf{v}$, acceleration $\mathbf{a}$ and steering force $\mathbf{SF}$ can be similarly denoted as 3-basis vectors with respect to a local dynamic orthogonal frame $P-xyz$ with its centre always attached to the current position of the centre of mass. The position of the centre of mass can thus be updated by applying the kinematic rules.

Let the update interval be Δt . A new steering force is calculated at the beginning of each iteration and applied to the point mass. This force will generate a corresponding acceleration and change in velocity which finally translates into an update on the rigid body's position:

Figure 3.1: A generic Constrained Ellipsoid Model of a terrestrial animal

$$a_t = \frac{SF_t}{m}, (SF_t < F_{max}) \quad (3.1)$$

$$v_t = v_0 + a_t \Delta t, (v_t < v_{max}) \quad (3.2)$$

$$p_t = p_0 + \frac{v_0 + v_t}{2} \Delta t \quad (3.3)$$

To represent the actual animal rig, a new vector **Heading** is defined to indicate the current orientation of the animal character so that the rig can be attached to the ellipsoid.

As a rigid body with a fixed volume has a moment of inertia which determines the torque needed for it to generate an angular acceleration about a fixed axis, the model should conform to its moment of inertia for the fixed axis rotations. In our case, the moment of inertia of the animal character only concerns rotations about Z axis since there will be no significant rotation on X and Y axes. The larger the size and mass of the animal, the harder the animal can generate angular acceleration while turning [45]. The moment of inertia about the Z axis is given by:

$$I_{zz} = \frac{m(a^2 + b^2)}{5} \quad (3.4)$$

Similarly, the angular velocity and acceleration of the ellipsoid is constrained by ST_{max} and ω_{max} . When a new steering torque is determined at the beginning of each iteration and applied to the ellipsoid, it generates a corresponding angular acceleration and change in angular velocity and finally translates into an update on the rigid body's heading direction:

$$\alpha_t = \frac{ST_t}{I}, (ST_t < T_{max}) \quad (3.5)$$

$$\omega_t = \omega_0 + \alpha_t \Delta t, (\omega_t < \omega_{max}) \quad (3.6)$$

$$h_t = \begin{bmatrix} \cos(\frac{\omega_0 + \omega_t}{2} \Delta t) & -\sin(\frac{\omega_0 + \omega_t}{2} \Delta t) & 0 \\ \sin(\frac{\omega_0 + \omega_t}{2} \Delta t) & \cos(\frac{\omega_0 + \omega_t}{2} \Delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} h_0 \quad (3.7)$$

So far, all the parameters of Constrained Ellipsoid Model is summarized in Table 3.1.

Table 3.1: Parameters of Constrained Ellipsoid Model

Parameter Name	Data Type
Mass (m)	scalar
Position (p)	vector
Velocity (v)	vector
Acceleration (a)	vector
Steering Force (SF)	vector
Max Force (F_{max})	scalar
Max Speed (v_{max})	scalar
Moment of Inertia (I)	scalar
Heading (h)	vector
Angular Velocity (ω)	vector
Angular Acceleration (α)	vector
Steering Torque (ST)	vector
Max Torque (τ_{max})	scalar
Max Angular Speed (ω_{max})	scalar

A problem with the rotational displacement is that sometimes a large angle between velocity v and heading h will lead to animal characters not heading toward their current moving direction, resulting in foot-sliding issues. One solution to deal with this issue

is to limit and constantly adjust the value of v_{max} based on the current included angle between v and h . Supposing that this angle and the value of v_{max} at time t are $\beta(t)$ and $v_{max}(t)$, the relation between $v_{max}(t)$ and $\beta(t)$ (h_t and v_t) is set as:

$$v_{max}(t) = v_{max}(t) * \cos(\frac{\beta(t)}{2}), \beta(t) \in (0, \pi) \quad (3.8)$$

Since h_t is a unit vector:

$$\cos(\frac{\beta(t)}{2}) = \sqrt{\frac{1 + h_t \cdot \frac{v_t}{|v_t|}}{2}} \quad (3.9)$$

As shown in the expression above, when the included angle between heading and velocity is zero, which means that they are perfectly aligned, there will be no adjustment. As β increases, v_{max} will decrease so that the animal will slow down to give time for the model to adjust its heading direction toward the current velocity direction. When β is equal to π , that is, heading and velocity are in opposite direction, v_{max} will be set to zero and the animal will stop moving.

Algorithm 4 Locomotive Dynamics: The position and heading direction are updated by respectively updating navigation force SF , acceleration a , velocity v and torque ST , angular acceleration α and angular velocity ω . The mass and moment of inertia of the animal are denoted as m and I . The displacement and angular displacement of the animal are denoted as p and h . In addition, update interval is denoted as Δt

```

1:  $t = 0$ 
2: while simulation is running do
3:   for each animal do
4:     Update Navigation Force  $SF$  ▷  $SF < F_{max}$ 
5:     Update Steering Torque  $ST$  ▷  $ST < ST_{max}$ 
6:      $v'_{max} = v_{max} * \sqrt{\frac{1 + h_t \cdot \frac{v_t}{|v_t|}}{2}}$ 
7:      $a_t = \frac{SF}{m}$ 
8:      $\alpha_t = \frac{ST}{I}$ 
9:      $v_t = v_0 + a_t \Delta t$  ▷  $v_t < v'_{max}$ 
10:     $\omega_t = \omega_0 + \alpha_t \Delta t$  ▷  $\omega_t < \omega_{max}$ 
11:     $p_t = p_0 + \frac{v_0 + v_t}{2} \Delta t$ 
12:     $h_t = \begin{bmatrix} \cos(\frac{\omega_0 + \omega_t}{2} \Delta t) & -\sin(\frac{\omega_0 + \omega_t}{2} \Delta t) & 0 \\ \sin(\frac{\omega_0 + \omega_t}{2} \Delta t) & \cos(\frac{\omega_0 + \omega_t}{2} \Delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} h_0$ 
13:   end for
14:    $t = t + \Delta t$  ▷ Begin new interval
15: end while
    
```

To sum up, Algorithm 4 describes an entire locomotive dynamics iteration. At the beginning of each update iteration, a new steering force SF and steering torque ST are determined and applied to the model. Before resulting in a corresponding update on its velocity and position, v_{max} is also adjusted based on the included angle between the previous velocity and heading vector to make sure that they stay aligned. The new heading vector h_t is updated by multiplying the previous heading vector h_0 by a rotation matrix of the angular displacement during time interval Δt about Z axis.

3.3 Summary

Mainly, the updated inputs during the dynamics procedure are the steering force SF and steering torque ST . With different ways how these two variables are defined, we can

generate a rich vocabulary of different autonomous motions. Comparative advantages include:

- Compared with heuristic and data-driven steering approaches, CEM is much more efficient, and without the need of pre-procedure of data gathering and model training. Thus, it is free from the burden of unavailability of representative data and loss of specificity to comply for a monolithic model.
- Compared with other rule-based steering approaches, CEM is more flexible, extendible and general in dealing with various terrains and environments.
- Compared with static grid-based representation of the terrain, CEM can deal with unpredictable paths, dynamic obstacles and targets. This is a key to success in pathfinding and navigation of autonomous animals.
- Autonomous character steering using Cellular Automata models is computationally efficient. However, these models do not allow contact between characters thus are unable to produce realistic group-level behaviours and to reveal the collective intelligence of the characters. CEM can effectively solve these problems.
- Finally, compared with traditional particle dynamics, CEM is derived based on a more precisely defined rigid body, thus it can generate more precise kinematic results with a relatively low computational cost.

Chapter 4

Navigation Layer - Autonomous Animal Behaviours

4.1 Basic Behaviours

4.1.1 Navigation

For an autonomous animal to approach its target, an *expected velocity* of the animal needs to be defined first, that is, the velocity at which the animal reaches its target under an ideal state. This vector is in the direction from the animal's current location to the target location and its magnitude is the value of *maxSpeed*. Supposing that the mass of the animal is m , then we can further derive the *expected momentum* of the animal, $p_e = mv_e$. In particular, the initial momentum, $p_0 = mv_0$. According to the impulse-momentum theorem: For an object to generate a change in momentum, a resultant force must be applied to the object for a period of time. Supposing that the update interval is Δt . The impulse J_F needed can be expressed as:

$$J_F = \int_t^{t+\Delta t} SF dt = mv_e - mv_0 \quad (4.1)$$

The steering force SF applied during time interval Δt is a constant force, therefore:

$$SF = \frac{mv_e - mv_0}{\Delta t} = \frac{m}{\Delta t} \cdot \Delta v \quad (4.2)$$

The work in this chapter has been published in "Collective Intelligence of Autonomous Animals in VR Hunting".

Similarly, for the steering torque ST with initial angular momentum $L_0 = I\omega_0$ and expected angular momentum $L_e = I\omega_e$ ($\omega_e = \omega_{max}$), the value of angular impulse J_T and steering torque ST is:

$$J_T = \int_t^{t+\Delta t} ST dt = I\omega_e - I\omega_0 \quad (4.3)$$

$$ST = \frac{I\omega_e - I\omega_0}{\Delta t} = \frac{I}{\Delta t} \cdot \Delta\omega \quad (4.4)$$

Due to the constraints of **maxforce** and **maxtorque**, it usually takes multiple iterations before the current velocity of the animal aligns with its expected velocity. Therefore, the steering force and torque given to the animal continuously change based on its new position and velocity after each iteration. Algorithm 5 (Appendix A.2) describes the basic navigation behaviour.

Algorithm 5 Navigation toward a target: Compute a navigation force and torque based on animal's current velocity, current angular velocity, and the position of the target. The position of the animal and the target are respectively denoted as p_i and p_t . The maximum velocity and expected velocity of the animal are respectively denoted as v_{max} and v_e . The current angular velocity, expected angular velocity and maximum angular velocity are respectively denoted as ω_i , ω_e and ω_{max} . In addition, time frame interval is Δt , the mass and moment of inertia of the animal are m and I .

```

1: for each animal do
2:    $\omega_e = \omega_{max}$ 
3:    $\Delta\omega = \omega_e - \omega_i$ 
4:    $T = \frac{I}{\Delta t} * \Delta\omega$  ▷ Update new Torque
5:    $v_e = (p_t - p_0).normalized * v_{max}$ 
6:    $\Delta v = v_e - v$ 
7:    $F = \frac{m}{\Delta t} * \Delta v$  ▷ Update new Force
8: end for
    
```

4.1.2 Pursuit and Fleeing

For the pursuit (and fleeing) behaviour, a predictor for the predator is set up by defining a new parameter T representing the prediction interval. The predicted displacement of the prey is computed by multiplying the value of its current velocity by T . The predicted

position of the prey is updated by adding this displacement on its current position and it is set as the target for the predator during the pursuit process.

The value of T can be defined in several ways. Instead of a constant, for the purpose of some diversification, T should be proportional to the distance between the animal and its target and inversely proportional to the sum of the velocity of both the predator and the prey. This is reasonable because if the distance between the two animals is very large, the pursuer should reserve more distance for prediction, and the same for velocity. In case that the prey is almost in front of the predator, the predator should directly head toward the current position of prey. The positional relationship between them can be acquired by calculating the dot product of their heading vectors $a = (a_1, a_2, a_3)$ and $b = (b_1, b_2, b_3)$, and the included angle θ between these two vectors is obtained by:

$$\theta = \cos^{-1}\left(\frac{a_1b_1 + a_2b_2 + a_3b_3}{|a||b|}\right) \quad (4.5)$$

Based on experimental results, it is stipulated that when $\theta < 20^\circ$, the predator will navigate directly toward the current position of the prey. Algorithm 6 (Appendix A.3) describes the pursuit strategy.

Algorithm 6 Pursuit: Compute a pursuit force for the pursuer based on the position of the prey. The current velocity of the animal is denoted as v and the current velocity of the target is denoted as v_t . Their respective positions and heading vectors are p_i , p_t and h_i , h_t . In addition, T represents the predicted interval and v_{max} represent the maximum velocity of the animal.

```

1: for each animal  $i$  in pursuit do
2:   //Check directional relationship between the two heading vectors, if the included
   angle is less than 20 degrees, set current position of the prey as target
3:   if  $a \cdot b < 0.95$  then
4:      $v_e = (p_t - p_i).normalized * v_{max}$  ▷ Target: prey position
5:      $\Delta v = v_e - v$ 
6:      $F = \frac{m}{\Delta t} * \Delta v$ 
7:   else
8:      $T = \frac{\|p_t - p_i\|}{v_{max} + v_t}$  ▷ Calculate interval
9:      $v_e = ((p_t + v_t * T) - p_i).normalized * v_{max}$  ▷ Target: predicted position
10:     $\Delta v = v_e - v$ 
11:     $F = \frac{m}{\Delta t} * \Delta v$ 
12:   end if
13: end for

```

Closely related to pursuit is the fleeing algorithm. The strategy is opposite to pursuit, thus it is the prey that navigates away from the predicted position of the predator. The expected velocity is opposite to the target instead of heading toward it. Algorithm 7 describes (Appendix A.4) this strategy.

Algorithm 7 Flee: Compute a fleeing force for prey based on the position of the pursuer. The current velocity of the animal is denoted v and the current velocity of the target is denoted v_t . Their respective positions are p_i and p_t . In addition, T represents the predicted interval and v_{max} represent the maximum velocity of the animal.

```

1: for each animal  $i$  in flee do
2:    $T = \frac{\|p_t - p_i\|}{v_{max} + v_t}$  ▷ Calculate interval
3:    $v_e = (p_i - (p_t + v_t * T)).normalized * v_{max}$ 
4:    $\Delta v = v_e - v$ 
5:    $F = \frac{m}{\Delta t} * \Delta v$ 
6: end for

```

4.1.3 Hovering

In the VR hunting game, when the animals are not assigned to any tasks, they are supposed to be idling and hovering randomly around the scene, waiting for an event to trigger. When these idling animals enter the sight of the VR player, we would like them to behave naturally. The objective of random hovering is to make the autonomous animals animate while moving around.

A solution to the hovering behaviour is proposed, with reference to the working principle of the rotation of internal combustion engine's crankshaft. As shown in Figure 4.1, the animal is linked to the crankshaft through a connecting rod and the destination is locked onto a circular path. The destination is given a random displacement after each update interval and its position keeps changing around the circle. By limiting the destination to this circular path, it can be assured that the animal rationally changes its orientation to the opposite direction. If the animal is currently heading toward east, in the next iteration it will remain heading towards east with a slight angular difference.

The customizable variables are the length of the rod l (hovering length), rotation radius r (hovering radius), and the random displacement of the destination after each update interval. Different values will produce various kinds of interesting hovering actions. Algorithm 8 (Appendix A.5) describes the hovering behaviour.

Algorithm 8 Hovering: Compute a hovering force for the animal. The hovering length is denoted as l , the hovering radius is denoted as r , the random jitter is denoted as j and the random displacement is denoted as d . The difference should be noted that t_c is the position of the target on the local coordinate and p_t is the final destination position on the global coordinate.

```
1: for each animal  $i$  in hovering do
2:   //Compute a random displacement
3:    $d = (\text{random}(-j, j), \text{random}(-j, j), \text{random}(-j, j))$ 
4:   //Add this displacement on the current target
5:    $t_c = t_c + d$ 
6:   //The new position of the target may not be on the circle
7:    $t_c = r * t_c.\text{normalized}$ 
8:   //Convert into global coordinate
9:    $p_t = v_i.\text{normalized} * l + t_c + p_i$ 
10:  //Compute the navigation force
11:   $v_e = (p_t - p_i).\text{normalized} * v_{max}$ 
12:   $\Delta v = v_e - v_i$ 
13:   $F = \frac{m}{\Delta t} * \Delta v$ 
14: end for
```

Figure 4.1: Hovering mechanism

4.2 Collective Behaviours

4.2.1 Group Animal Pursuit and Fleeing

In nature, collective intelligence of animals is applied in order to increase the chance of successfully chasing down their prey. Predators such as wolves usually prefer more strategies of working in groups; in addition to individual wolf's prediction, they are aware of the roles they are playing and the tasks they are assigned as a group's member. In recognition of this, a group chasing strategy is proposed to embody the collective intelligence of these autonomous animals. The wolf-deer pursuit and fleeing scenario is used again for explanation of the algorithm.

Refer to Figure 4.2, instead of each wolf targeting its predicted position of the prey, some wolves will try to flank harder, blocking all possible escape routes and aiming to gradually corner the prey. In the illustration, different tasks and targets of the team will determine the individual pursuit behaviours. Note that wolf 2 is nearer to the deer and has a shorter *offset 1*; whereas wolf 3 is further away from the deer, and has a larger *offset 2*.

Figure 4.2: Group pursuit

This flanking strategy is described by Algorithm 9. Each wolf will be randomly assigned to chase after one of the 3 targets. The first target is directly the predicted position of the prey that, if the included angle between the two heading directions is less than 20 degrees, is the current position of the prey. The two other targets are virtual targets that are offsets, either to the left or right, from the prey. This offset is inversely proportional to the distance between the predator and the prey.

Algorithm 9 Group Flanking: Compute different pursuit forces for every group member based on the position and heading direction of the prey. The previously introduced individual pursuit algorithm is denoted as *pursuit()*, whose input variable is the target of the pursuit. *rd* is a random variable between 0 and 1, which determines the task to be assigned. $\vec{l}$ and $\vec{r}$ are two directional vector pointing toward the left and right side of the prey. Cross product of the forward vector and up vector is used.

```

1: for each animal i in group do
2:   rd = random(0, 1)
3:   if rd < 0.2 then
4:     Directly target the predicted position / current position
5:     pursuit(pt)
6:   else if rd < 0.6 then
7:     //Target at left offset of the prey
8:     //Find out the vector directing toward the left of the prey
9:      $\vec{l} = t.transform.forward \times t.transform.up$ 
10:    //Set the offset as new target
11:    pursuit(pt +  $\vec{l} * \|p_t - p_i\| * 0.7$ )
12:   else
13:     //Target right offset of the prey
14:     //Find out the vector directing toward the left side of the prey
15:      $\vec{l} = t.transform.up \times t.transform.forward$ 
16:     //Set the offset as new target
17:     pursuit(pt +  $\vec{l} * \|p_t - p_i\| * 0.7$ )
18:   end if
19: end for

```

Meanwhile, to keep the in-game balance, the preys should have the same level of intelligence, resulting in real time chase-flee action between the animals. The prey is supposed to be fleeing based on the predicted position of a single pursuer (nearest). However, with group flanking, this fleeing strategy will quickly get the prey cornered and killed. Therefore, the fleeing strategy can be improved by dynamically determining

the fleeing directions based on every observable predator within the prey's sight range. By doing so, the prey will determine the best fleeing direction to maximize the distance away from every observable predator. The problem turns to finding an average positional vector putting every predator's position into consideration as the new target for flee, as shown in Figure 4.3. Supposing that there are currently n observable predators, finding the centroid of a polygon with n vertices can be done through Eigen analysis (where P contains positional vectors of all the vertices and M_n is the average matrix in a x -dimensional coordinate):

$$p^{(\infty)} = \lim_{k \rightarrow \infty} p^k = (\lim_{k \rightarrow \infty} (M_n)^k) P = \left(\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n} \right)_{1 \times n} P_{n \times x} \quad (4.6)$$

Figure 4.3: Fleeing based on an average positional vector

Algorithm 10 Group Flee: Compute a fleeing force for the prey based on the position and velocity of every observable predator. The previously introduced individual flee algorithm is denoted as *flee()*, whose input variable is the target of the flee (*C* as target's position and *V* as target's velocity). The total distance between every predator and the prey is denoted as *d*.

```

1: for each animal i in group flee do
2:   d = 0
3:   C = (0, 0)
4:   V = (0, 0)
5:   for each predator n detected do
6:      $d_n = \|p_i - p_n\|$ 
7:     d = d + dn
8:   end for
9:   for each predator n do
10:     $\omega_n = \frac{d_n}{d}$ 
11:    C = C +  $\omega_n v_n$ 
12:    V = V +  $\omega_n v_n$ 
13:   end for
14:   flee(C, V)
15: end for
```

Using the geometric centroid of the polygon with observable predators as its vertices to represent an integrated position of all the predators means that every predator's threat toward the prey are been treated equally during the pursuit process. To embody a difference in level of threat between the predators, a more appropriate approach would be applying the weighted centroid instead of the geometric centroid. The predators with a closer distance toward the prey will be more threatening to the prey and thus have a higher weight during computation. The same weight values are used to compute the average weighted velocity for all the predators, as follows:

$$\omega_i = \frac{d_i}{d_1 + d_2 + \dots + d_k} \quad (4.7)$$

$$C = p_1 \omega_1 + p_2 \omega_2 + \dots + p_k \omega_k \quad (4.8)$$

$$v = v_1 \omega_1 + v_2 \omega_2 + \dots + v_k \omega_k \quad (4.9)$$

where k is the number of observable predators, d_i is the distance between the i th predator and the prey and ω_i is the weight of the i th predator. C is the weighted centroid and v is the weighted average velocity which serve as the final inputs for *flee()*. Algorithm 10 describes this fleeing strategy.

This group fleeing algorithm is also suitable for "one predator chasing a group of preys" scenario, where prey with a closer distance has a higher weight.

4.2.2 Force Combination

Animals like wolf, deer and bear mostly act as a group. A leader guides the team in behaviours such as navigation, pursuit, flee etc. To incorporate this collective intelligence into the VR hunting game, an appropriate following target for the members needs to be defined. This target can be determined based on the velocity vector of the leader, which gives information on both the location, and the heading direction of the leader.

$$tv = leader.velocity$$

$$tv = normalized(tv) * leaderBehindDist$$

$$behind = leader.position - tv$$

The larger the value of *leaderBehindDist*, the larger the distance between the following target and the leader. The coordinate of *behind* is set as the navigation target for the members. Hereafter, we must make sure that the squad would not be crowded while members are performing the following action. This can be done by setting an invisible radar around the animal for detection of other nearby animals. For each nearby animal detected, a repulsive force will be exerted to make sure that animals keep a certain distance between each other. The detection radar centred on the point mass of the animal is determined with a certain radius, as shown in Figure 4.4. When another animal is detected, it is considered as one of animal's neighbour, otherwise ignored.

For each neighbour detected, the vector r from the animal to the neighbour is first computed. The direction of the repulsive force is obtained by normalizing vector r . The smaller the distance is from the animal to its neighbour, the larger the magnitude of

the repulsive force is; therefore, it is inversely proportional to the distance between the animal and neighbour, $r/\|r\|$. This is described by Algorithm 11.

Finally, after obtaining all the forces required for the leader following motion, these forces need to be combined. Algorithm 12 is for finding the resultant force of all the individual forces exerted on the animal.

Figure 4.4: Detective radar and repulsive force

Algorithm 11 Separation: Compute a separation force for all the team members so that the formation would not be too crowded. The position vector from the target to the character is denoted as r .

```

1: for each animal  $i$  do
2:   //Retrieve all the neighbour targets inside the detection radar
3:   for each neighbour targets inside the detection radar do
4:     //Compute the distance between the target and the character
5:      $r = p_i - p_s$ 
6:      $R = r.normalized$ 
7:      $l = \|r\|$ 
8:     //Compute separation provided by this target
9:      $F+ = R/l$ 
10:  end for
11:  //Make sure the resultant force does not exceed maxForce
12:  if  $F > F_{max}$  then
13:     $F = F_{max}$ 
14:  end if
15: end for

```

Algorithm 12 Resultant force: Compute a resultant force of all the forces in the force list.

```
1: for each animal  $i$  do
2:   for force  $f_k$  in forcelist do
3:      $F = F + f_k * f_k.weight$ 
4:   end for
5:   //Make sure the resultant force does not exceed maxForce
6:   if  $F > F_{max}$  then
7:      $F = F_{max}$ 
8:   end if
9: end for
```

Note that all the individual forces are weighted since there is a priority relationship between these forces. For instance, obstacle avoidance should be the first priority among all other motions. Therefore, the weight of obstacle avoidance is relatively higher.

While this solution does handle the motion combination in most of the cases, there are still scenarios when animal would behave bizarrely. For instance, when the animal is currently beside an obstacle without any other tasks assigned, it will be pushed away by a huge repulsive force since the weight is so high. Therefore, an alternative solution is to use random prioritized dithering, as described in Algorithm 13.

Algorithm 13 Random prioritized dithering: Generate the steering force based on a prioritized random value. The weight of force k is denoted as w_k and its probability in computing steering force is denoted as p_k . n is the total number of forces in the *forcelist*.

```
1: for each animal  $i$  in group flee do
2:   for force  $f_k$  in forcelist do
3:      $p_k = \frac{w_k}{\sum w_n}$ 
4:   end for
5:    $r = random(0, 1)$ 
6:   for force  $f_k$  in forcelist do
7:     if  $r < \sum p_k$  then
8:        $F = f_k$ 
9:     end if
10:  end for
11: end for
```

The alternative algorithm associates a probability to all the input forces and forces

with higher priority given larger probability. At each update iteration, the algorithm generates a new random value to determine the force to apply and forces to dismiss. Therefore, it combines all the motions with prioritized forces not too influential while less prioritized forces still have some influence.

4.3 Experimental Results

4.3.1 Navigation

The proposed algorithm is advantageous over the existing force-driven methods, which simply apply to the NPC an attractive force toward the target, such as a centripetal force, which only produces an orbital trajectory around the target instead of approaching the target.

Figure 4.5 illustrates the new navigation behaviour of an autonomous animal in our experimental study. The initial velocity v_0 , expected velocity v_e and their difference Δv at the starting point of a wolf approaching its target are accordingly given. Its trajectory during the procedure is generated in red. Instead of heading directly toward the position of its target, which will result in sudden and unrealistic rig rotation with an easily predictable straight trajectory, the wolf is gradually adjusting its orientation during its approach toward the target which is more in line with a realistic scenario. Another advantage of this algorithm is that, the NPCs can be assigned to any tasks at any time under any circumstances since a new “task” only exerts a new virtual external “force” and “torque” on the rigid body without needing to conform to its previous state of motion.

4.3.2 Pursuit

Figure 4.6 shows the generated prediction intervals and the values of T of three predators in our experiments. The three predators navigate with respective directions toward their targets, considering the prey as a moving target. (Note that the actual pursuit paths will be generated based on the CEM in Chapter 3, illustrated in Figure 3.1.) Compared with existing games in which the pursuit target is set as the target for navigation, the resulting movements of the animals are more realistic, exhibiting the first level of intelligence: not

Figure 4.5: An autonomous animal approaching its target

Figure 4.6: Pursuing the fleeing target with different distances

directly to chase toward the current position of its target but its predicted position of the target so that it can catch up with the target with a minimum amount of time.

4.3.3 Hovering

If the movement of hovering NPC is defined based on some fixed pre-defined paths, then its trajectory would be easily predictable when the player examines them closely. Other existing systems take a brutal approach to set a new randomly located destination for the animal around the scene after a certain time interval. Although this approach seems acceptable at first glance, its performance is not that satisfying. When the new randomly located destination appears behind the animal, it will cause the animal to suddenly turn around as if they just found a prey or alerted by a predator. Though, in fact, there is no specific purpose for it to do so during the hovering process until a new event is triggered. Several such cases are shown in our experiments, along the trajectory in Figure 4.7.

When the animal is hovering with its destination locked onto a circular path, it can be assured that the animal can rationally change its orientation to the opposite direction. As shown in our experiment in Figure 4.8, the hovering wolf animates along a trigonometric shaped path. Furthermore, the algorithm is highly customizable by simply adjusting the values of the hovering parameters (hovering length and radius). The parameters can be set to different values for different types of animals, environments and circumstances. These values can also be set as variables during the hovering process instead of constants to generate a more unpredictable trajectory and heterogeneous behaviour. The random hovering mechanism once again reflects the simplicity and diversity of the model.

4.3.4 Group Hovering with Leader

Figure 4.9 shows a comprehensive experiment result of the force combination mechanism. The leader wolf (track in red) is navigating toward a certain direction while performing the hovering motion. This gives the leader the sense of a vague direction of the prey without being too sure thus hovering toward that direction. The member wolves (tracks in blue) trust the senses of the leader thus follow it while separating with each other. The experiment result demonstrates a fairly natural scenario of “hovering with leader”,

Figure 4.7: Hovering path with randomly located destination

Figure 4.8: Hovering path with Autonomous Animal Behaviour

Figure 4.9: Force combination: hovering with leader

with the NPC trajectories being smooth in transitions, team member positions keeping a certain distance with each other, and staying behind the leader.

In this comprehensive experiment, significant advantages of the new algorithms over previous works are demonstrated, mainly in:

- Excellent performance of the model in handling with interactions between individual NPCs, and in generating group-level behaviours. These are the most critical requirements of an integrated steering system and embodiment of NPCs' collective intelligence.
- True feasibility of behavioural combination with the model by applying the weighted resultant force algorithms and the random prioritized dithering algorithms.
- Realizable collective intelligence by combining different behaviours and generating new behaviours. This feature will enable further customization and extension for a wide spectrum of autonomous motion control systems.

4.4 Summary

The motivation of defining Steering and Navigation Layer is to generate a simulative agent model and path-finding algorithms so that the commands given by the Decision layer can convert into agent movements on the global coordinate. The contributions in this chapter can be briefly concluded as:

- Previous related methods are compared and evaluated
- A novel rig simplification model is introduced
- A set of new Autonomous Steering Behaviours are proposed
- Collective behaviours and strategies are studied
- Computational experiments and performance comparisons are presented

New adaptive steering algorithms are designed for the autonomous animals to navigate around the environment. Several individual behaviours including navigation, pursuit, fleeing and hovering are introduced; the corresponding experiment results are presented, and the comparative advantages over previous works are highlighted.

Research objective is focused on collective intelligence of animal agents in decision making and tactical thinking. Advanced strategies for a group of autonomous animals are developed in order to simulate a more realistic forest environment. Group behaviours and strategies of the autonomous animals embody their collective intelligence in group cooperation scenarios. Finally, force combination algorithms are explained to reflect the diversity and flexibility of this approach.

Chapter 5

Locomotion Layer - Generic Quadruped Inverse Kinematics Model Reinforcement Learning with Autonomous Animal Agent

5.1 Introduction

In this Chapter, we will focus on design of the locomotion layer for an articulated quadruped model using two different approaches.

First, a new Generic Quadruped Inverse Kinematics Model (GQIKM) is proposed. The model details and constraints are introduced and illustrated. Secondly, algorithms that describe an integrated walking gait cycle of a typical quadruped animal are explained in detail. Within this method, the connection between autonomous animal behaviours and GQIKM is achieved. Previously introduced algorithms that drive agents with script motion can now be applied on an articulated locomotive model for more realistic movement behaviour.

Lastly, a novel approach toward environmental configuration is proposed by combining Proximal Policy Optimization (PPO) with Autonomous Animal Behaviours for creating neural network models and learning agents with more efficiency and robustness. Experiments are conducted on a specific user case where a wolf agent need to chase down

The work in this chapter has been published in "Reinforcement Learning for Quadruped Locomotion".

its prey in the shortest time possible and can be extended to other scenarios where target simulation can help to improve the performance of the trained model.

5.2 Background

5.2.1 Inverse Kinematics in Computer Graphics

The objective of Inverse Kinematics (IK) algorithms is to find a solution to determine joint parameters of a manipulator so that the end effector moves to the desired position using numerical equations and methods. Researchers has been placing a high value on studies of IK solution since 20th century. Until now, IK methods are wildly applied in fields including computer graphics, game design, robotics etc [46].

In game design, IK is mostly used in dealing with locomotion of articulated models such as humanoids, animals, multipedal characters. In the earlier days, in order to animate the motion of the character, designers need to adjust all Degrees of Freedom (DoFs) by tuning the rotation variables on each joint to realize the locomotion of these characters, which is also called “forward kinematics”. With the fast development of computer software and hardware, locomotion of in-game agents has become more and more complex following workload become more and more tedious. Since then, IK algorithms have slowly replaced traditionally used forward kinematics methods and become the fundamental solution towards this issue until now. The key concept of IK is that, instead of tuning rotation variables on each single joint, IK takes the desired target position of the end effector as input and outputs the rotation variables of each joint by using kinematic equations and numerical computations. By applying IK solutions, the workload of designing locomotion of the in-game character has been largely deduced.

Research and solutions toward IK can generally be classified into four kinds of approaches: Analytical, Numerical, Data-driven and Hybrid. Analytical approaches try to solve a function of joints starting position, length of the mechanism and rotational constraints [47]. Analytically, the solution can be computed by trying out all the possibilities that the joints can be placed. The advantage of dealing IK with analytical approaches is that they would not suffer from singularity problems thus wildly used in robotics with a low DoF. Secondly, they are more computation efficient when solving simple and specific problems comparing to other IK solvers. However, their scalability is very poor due

to the non-linear nature of kinematic equations, which means that these solutions can hardly handle IK with high amount of DoFs, redundant and repetitive problems such as computer graphics and game design, thus nowadays only applicable theoretically. Numerical approaches are widely explored by researchers thus include a huge number of different algorithms. Some key methods using numerical approaches include: Jacobian methods [48], Newton methods [49], Heuristic methods such as Forward and Backward Inverse Kinematics (FABRIK) [50], Cyclic Coordinate Descent (CCD) [51] and their variants. These methods find IK solutions by repeating a set of numerical iterations. The final objective becomes minimization of a cost function problem. In recent years, as development of machine learning and motion capture technology gets more mature, data-driven IK approaches has also become applicable in practical scenarios. Referring to collected data from the real world, the articulated humanoid or animal model learns to match the end effector positions to a feasible posture. A correlation between joint rotation and control parameters can be found through deep learning and convolutional neural networks [52–55].

5.2.2 Reinforcement Learning

Recently, more and more self-adaptive approaches that orient towards autonomous and intelligent agents have been introduced. Path planning using reinforcement learning has been one major approach towards navigation of the agent. The earliest concept of reinforcement learning comes from Chris Watkins, who invented the term of “Q-learning” [56]. “Q-learning” is a very simple value-based, off-policy algorithm. The objective of this algorithm is to learn and find out a “Q-table” where the values of maximum discounted future reward (Q-value) are stored for each corresponding state and actions by modelling the value function as a Bellman equation. After obtaining the “Q-table”, the agent can then choose to make the best move towards a maximum reward at every current state. Later on, the concept of “Q-learning” is also further extended towards other reinforcement learning algorithms with neural networks such as deep “Q-learning” [57]. However, the biggest disadvantage of “Q-learning” is also very apparent. When the task is becoming more complex and concerns issues such as large number of states, intractability problems, unguaranteed value function, continuous states rather than discrete states,

requirement of long-term rewards rather short-term, “Q-learning” algorithms generally behave poorly and inefficiently. Nowadays, with the success of machine learning and deep neural networks, more and more reinforcement learning algorithms are leaning towards policy gradient methods which optimize a parameterized policy using gradient descent with long-term cumulative rewards rather than optimizing values. This is a fundamentally different approach towards the issue since policy gradient algorithms use probability to choose actions to take which frees the requirement of maintaining a “Q-table” to store a one-to-one correspondence between states and actions. When the agent received a positive reward by performing a certain action, the probability of it repeating this kind of actions will be increased in the future and vice-versa. Some of the major approaches toward model-free RL: Deep Q Network (DQN) [58], Deep Deterministic Policy Gradient (DDPG) [59], Trust Region Policy Optimization (TRPO) [60], Proximal Policy Optimization (PPO) [61] and Soft Actor-Critic (SAC) [62]. Inspired by traditional Q-learning, DQN is the beginning of a successful combination of convolutional NN and RL. It uses a convolutional NN to represent the value function and provide target value to the NN based on Q-learning in RL. Although DQN can now successfully handle tasks with high observation spaces, it is still not suited for environment with continuous action spaces. DDPG which is based on Deterministic Policy Gradient (DPG) extends DQN ideas through an actor-critics approach. Comparing to DQN, DDPG is able to operate in continuous action spaces using a deterministic policy. Similar to DDPG, TRPO also uses two NN instances for both actor and critics. The major contribution of this algorithm is that the update size during training is constrained by a “trust region” to avoid new policy diverging too much relative to existing policy which makes the training process easier. PPO is a new approach towards policy gradient algorithms. A novel objective function is proposed and is updated during the training procedure in multiple epochs of mini-batches per data sample. TRPO is also meant to determine an appropriate step size, but implementation of PPO is much more simple and easier to solve. SAC, which is also using an actor-critics architecture, tries to create policies in continuous spaces, combining the concept of exploring maximum entropy (MERL).

5.3 A Generic Quadruped Inverse Kinematics Model

5.3.1 Model Basics

With the combination of CEM and AAB Algorithms, an animal character is already able to freely navigate around a plane terrain using pre-defined animations like “walking” and “running”. Script motion is more computationally efficient than root motion and is acceptable when characters are moving in large terrains. However, the biggest disadvantage of script motion is also very obvious, which is the foot-sliding issue. No matter how preciously the relation between the current velocity and the animation play speed is defined, the feet sliding problem is still inevitable. To solve the problem, a Generic Quadruped Inverse Kinematics Model will be introduced, as illustrated in Figure 5.1, where a wolf character is used to represent the GQIKM; it can easily be extended to other types of quadrupedal animal characters.

Figure 5.1: Generic Quadruped Inverse Kinematics Model

The Generic Quadruped Inverse Kinematics Model consists of a total number of 14 bones and 20 joints. One joint (joint 1) serves as an imaginary root joint which keeps track of the position and orientation of the Constrained Ellipsoid Model and determines the position of the 5 sub-root joints of the body. A total number of nine joints serve as

possible end effectors, respectively located at the four sub-root joints above the upper legs (joint 5, 9, 13, 17), four joints on the feet (joint 8, 12, 16, 20) and the joint on the head (joint 4). The main objective of the model is to translate the displacement of position p and the orientation vector *heading* from the Constrained Ellipsoid Model into new positions of the end effectors in order to apply IK algorithms to drive the entire model for the desired animations.

A general spherical joint has three degrees of freedom in terms of rotational motion: roll (rotation around x axis), pitch (rotation around y axis) and yaw (rotation around z axis). A ball-and-socket joint has all three degrees of freedom; on the other hand, a hinge joint only has one (pitch). Each leg consists of two hinge joints (knee) and two ball-and-socket joints (elbow and ankle). The head consists of two ball-and-socket joints. An illustration of joint IDs and types is shown in Figure 5.2

Figure 5.2: Joint positions and types

5.3.2 Constraints for a Walking Gait Cycle

Forward and Backward Reaching Inverse Kinematics (FABRIK) [50] is an algorithm which plays with displacements of points and lines. Compared with other IK solvers, FABRIK is more suitable in handling singularity problems and generating smooth armature motions; this makes it an ideal choice for real-time computation [63].

For Quadrupedal Inverse Kinematics Model like the motion of a wolf character, an entire iteration of a waking gait cycle can be divided into two phases for each of the four legs: the “**swing**” and “**stance**” phases [64]. In “swing” phase, the foot joint (e.g. joint 8) swings in the air and the sub-root joint (e.g. joint 5) follows the motion of the root joint (joint 1). In “stance” phase, the position of the foot joint is fixed and the sub-root joint follows the motion of the root joint. During the entire simulation process, the distance between the root joint and the four sub-root joints, the mutual distance between the four sub-root joints remains constant. Algorithm 14 for the “swing” phase and Algorithm 15 for “stance” phase are accordingly designed. Some visual demonstrations of the two phases are shown in Figure 5.3.

Figure 5.3: Two phases of one leg during walking gait cycle

Algorithm 14 A full iteration of the Swing phase: Before each iteration starts, a new positional displacement Δs_e and a new orientation vector *heading* is updated from the Ellipsoid model. The initial values for horizontal and vertical distances toward the target are set as $d_{dif} = 0.19$, $h_{dif} = 0.1$. (lift: 0.19 + fall: 0.19 + translation: 0.38 = one complete stride length: 0.76). As the simulation goes on, this value keeps on decreasing until reaches to 0 when the end effector reaches the target and looks for its next target. Note that after each iteration, the horizontal displacement of the end effector is always twice the horizontal displacement of the sub-root joint, root joint and the Ellipsoid. The new initial position of the end effector before each iteration starts is denoted as p_0 . The position of the end effector is denoted as p_e . The position of the target is denoted as t .

```

1: //1. IK motion
2: //Update the initial position of the end effector
3:  $p_0 = p_e$ 
4: if leg in lifting phase then
5:     //Set the position of the peak target
6:      $t = p_e + d_{dif} \cdot \text{heading} + h_{dif} \cdot (0, 0, 1)$ 
7: else
8:     //Set the position of the ground product
9:      $t = p_e + d_{dif} \cdot \text{heading} + h_{dif} \cdot (0, 0, -1)$ 
10: end if
11: FABRIK from  $p_e$  to  $t$ , root joint:  $p_r$ . Stop when the horizontal displacement of  $p_e$ 
    is equal to the displacement of the Ellipsoid  $\Delta s_e$  or when target is reached
12: while  $|p_e - p_0|_x < \Delta s_e$  do
13:     //Update new position of  $p_e$ 
14:     ...
15: end while
16: //When reached target:
17: if  $p_e = t$  then
18:     //If in lifting phase: reset distance toward target, switch to falling phase and
    continue simulation
19:     if leg in lifting phase then
20:          $p_0 = p_e$ ,  $d_{dif} = 0.38$ ,  $h_{dif} = 0.1$ 
21:          $phase = fall$ 
22:     //If in falling phase: switch to stance phase, simulation ends
23:     else
24:          $phase = stance$ 
25:     end if
26: else
27:     //Update horizontal and vertical distance toward target and continue simulation
    in lifting phase
28:      $d_{dif} = d_{dif} - |p_e - p_0|_x$ 
29:      $h_{dif} = h_{dif} - |p_e - p_0|_y$ 

```

```

30: end if
31: //2. Translational motion
32: for each joint  $i$  do
33:      $p_i = p_i + \Delta s_s \cdot heading$ 
34: end for

```

Algorithm 15 A full iteration of the Stance phase: The initial value for horizontal distance toward the target is set as $d_{dif} = 0.38$. As the simulation goes on, this value keeps on decreasing until reaches to 0 when the end effector reaches the target. The new initial position of the end effector before each iteration starts is denoted as p_0 . The position of the end effector is denoted as p_e . The position of the target is denoted as t .

```

1: //IK motion
2: //Update the initial position of the end effector
3:  $p_0 = p_e$ 
4:  $t = p_e + d_{dif} \cdot heading$ 
5: //FABRIK from  $p_e$  to  $t$ , root joint:  $p_r$ . Stop when the horizontal displacement of  $p_e$ 
   is equal to the displacement of the Ellipsoid  $\Delta s_e$  or when target is reached.
6: while  $|p_e - p_0|_x < \Delta s_e$  do
7:     //Update new position of  $p_e$ 
8:     ...
9: end while
10: //When reached target:
11: if  $p_e = t$  then
12:     //Switch to swing phase, simulation ends
13:      $phase = swing$ 
14: else
15:     //Update horizontal and vertical distance toward target and continue simulation
       in stance phase
16:      $d_{dif} = d_{dif} - |p_e - p_0|_x$ 
17: end if

```

While the animal character is moving, the right-side legs and the left-side legs take turn in executing one of the two phases. When legs of one side are executing the “swing” phase, the legs of the other side execute the “stance” phase and vice versa. It should be noted that the final displacements of the sub-root joints are the same after both phases, which maintains the positional and motional synchronization of the four leg chains from both sides, the four sub-root joints, and the root joint during the entire simulation process.

During the “swing” phase, the motion of the leg is further divided into two sub-phases: “lift” and “fall”. According to research on gait pattern of animals, one stride length of a grown wolf is approximately around 30 inches (0.76m) and while the wolf is running at its full speed, the distance between groups of tracks can reach up to 6 feet (1.8m) [65]. To simulate a realistic animation, when the horizontal displacement of the foot joint reaches half of the stride length (0.38m), the vertical displacement reaches to its peak at the same time. Therefore, this moment is the point-in-time that separate the two phases. The final leg chain movement is a combination of translational motion and IK motion. The translational motion moves the bones toward the heading direction without changing any of the inter-bone angles. The IK motion moves the foot joint (end effector) to the peak target position during the “lift” phase and back to ground target position during the “fall” phase. Figure 5.4 illustrates a full iteration of the IK motion of a leg chain. The new positions of the joints after each time iteration are updated after completing both the translational and the IK motion.

Figure 5.4: Swing: lift and fall

On the other hand, the “stance” phase is more straight forward. To synchronize body motion with “swing” phase, the sub-root joints (joint 5) serve as end effectors which move along with the CEM while the foot joints (joint 8) serve as fixed roots for IK motion.

A full iteration of the right-side legs walking gait cycle starting from the beginning of the “swing” phase to the beginning of the “stance” phase is shown in Figure 5.5. Note that these screenshots are taken in one fixed global coordinate, the movement of the joints are attached to the movement of the root bone.

Figure 5.5: Quadruped walking gait cycle

5.3.3 Tuning and Generalization

The Generic Quadruped Inverse Kinematics Model is now able to translate outputs from the Constrained Ellipsoid Model into root motions of the leg chains and generate smooth walking animation. Tunings for the model are designed in the following aspects:

- **IK target reachability:** In Generic Quadruped Inverse Kinematics motion, it is important to make sure that the target is within the reach of the constrained bone chain during the entire simulation. The first layer of constraints comes from Constrained Ellipsoid Model. The inputs of Generic Quadruped Inverse Kinematics Model including the positional displacement of the root bone and the angular displacement of the heading vector are constrained by the maximum velocity v_{max} and the maximum angular velocity ω_{max} of the Constrained Ellipsoid Model. The second layer of constraints is within the constant parameters defining the target position in each type of animation. Taking the “walking” animation specifically defined in the above section for instance, the peak target position and the ground target position are defined with reference to stride length and height of a wolf in the real world. When there is no angular displacement on the Constrained Ellipsoid Model, it can be assured that the target is reachable. Since the sub-root joint above the upper leg is a ball-and-socket joint (with rotation around x -axis locked during “walking”), the model allows a limited amount of angular displacement at

each iteration. This amount depends on how much the joint can rotate around the z-axis. As long as the angular displacement of the heading vector does not exceed the limit of the sub-root joint rotation on z axis, the target position will always be reachable. When the target is found to be out of reach, there are two possible solutions: The *heading* vector is updated to fit the maximum reach range of the leg chain and feedback the new *heading* vector to Constrained Ellipsoid Model for further update; or, the current animation is terminated and the joint positions are updated based on definitions of other animations to keep the model in shape.

- **Frame rate and animation adjustment:** Having a positional displacement that is too large with respect to the value of one stride length can lead to missing of key frames and unsmooth animation. This can happen for two major reasons: the time interval between each iteration or the velocity of Constrained Ellipsoid Model is too large. For the first case, the solution is to decrease the value of time interval based on the current value of positional displacement and stride length for a higher number of frames per second. For the second case, the solution is similar to the *heading* vector: either by maintaining the walking animation and feedback the threshold velocity to Constrained Ellipsoid Model for further update, or by switching the current animation to “running” for larger stride length values in order to keep the model in shape.
- **Head chain motion:** The motion of the head chain is relatively simple compared with the leg chains. There is no inverse kinematics motion related with the movement of Constrained Ellipsoid Model and the translational motion works with exactly the same manner as the leg chain. However, some interesting rules can be defined to control the end effector target position of the head bone. For instance, instead of always setting the target position aligned with the *heading* vector, it can be set to align with the direction of the prey in a flanking scenario, which gives the VR player an impression that although the predator is not currently heading directly towards the prey, its sight is always on the prey while flanking. These simple rule-based approaches can largely improve user’s experience while gaming.

- **More animation and animals:** The modelling of more types of animal rigs and animations can be done in the similar way by applying the basic methods in modelling a quadrupedal wolf character and “walking” animation. Creating other animal rigs involves modifying the bone lengths, sub-root joint positions and joint types. The key to creating new animation such as “running”, “sneaking”, “idling” etc. is still the relations and translations between Constrained Ellipsoid Model variables and end effector target position. Taking “running” for example, the running gait cycle is similarly divided into two phases: ”swing” and ”stance”, except this time, it’s the two front legs that perform ”swing” phase when the two hind legs perform ”stance” phase simultaneously and vice versa. An increased stride length and lifting height will allow generation of more key frames during a single phase for a higher moving speed.

5.4 Reinforcement Learning with Autonomous Animal Target

5.4.1 Proximal Policy Optimization (PPO)

PPO was introduced by OpenAI in 2017, and soon became the baseline algorithm for general RL research. PPO may not be the best RL algorithm nowadays, but it can be said that PPO is the most widely applicable algorithm at present.

Policy optimization methods has started to become the mainstream of reinforcement method since tasks that we want the agents to achieve is getting more and more tedious and the environment being more complicated. Unlike in the early days, when the first generation of reinforcement learning methods are used to train agents with discrete action options in a 2-dimensional environmental space, learning algorithms must be also effective when dealing with continuous action spaces in 3D, VR, and real-world environments.

PPO is the method that was built upon Trust Region Policy Optimization (TRPO) to solve the problem. Instead of adding new constraint parameters, PPO aims to directly involve these constraints into the objective function itself to achieve better efficiency. In this chapter, we will only have a quick look at one type of variant of the PPO algorithm:

PPO – clip since it is the default variant used at OpenAI and also the variant we are going to use latter on in our experiment.

$$L^{clip}(\theta) = \widehat{\mathbb{E}}_t[\min(r_t(\theta)\widehat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\widehat{A}_t)]$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

As shown in the expression above, the objective function of *PPO – clip* consists of expectation of advantage function $\widehat{A}_t$ times $\min()$ between the default Policy Gradient objective and Clipped Policy Gradient objective. The key hyperparameter in this function is ϵ which defines the range constraining the new policy from diverging too much from the old. The most frequently used ϵ values are 0.1 and 0.2.

When the advantage is positive, $\min()$ serves as a ceiling to stop objective function from increasing at some point (defined by ϵ) so that the new policy will stop benefitting from going further away from the previous policy as vice versa for when the advantage is negative. Experimental results have shown that *PPO – clip* outperforms other mainstream Policy Gradient methods in multiple case scenarios.

5.4.2 Autonomous Animal Target Training

In this sub-section, we will study a specific user case where a wolf agent needs to learn how to chase down its prey in the shortest time possible

When an agent needs to know how to reach its target efficiently, the target configuration during the training process is very critical and directly influences the behaviour of the agent. Higher level of intelligence for the target means a higher level of robustness for the agent. The target to be introduced in this section is also a self-driven agent similar to the pre-trained agent. The target agent is driven by a combination of Autonomous Animal Behaviours including: *Flee()*, *Hover()* and *CollisionAvoidance()*.

At the beginning of a new episode, the target agent first chooses a random position on the terrain to respawn. After spawning, instead of staying on that position and wait till the wolf agent reaches it, it will first detect whether wolf agent is in its detection radius. If the wolf agent is not detected, then it will proceed to execute the *Hover()*

Figure 5.6: Hover + Obstacle Avoidance + Flee

behaviour. When the wolf agent is detected, the target agent will start executing the *Flee()* behaviour. During both *Hover()* and *Flee()* behaviour, the target will always be constrained by *CollisionAvoidance()* force to keep itself from hitting a wall and cornered. Figure 5.7 shows the movement of the target agent while been chased by the wolf agent in a training episode. (Wolf Agent: track in green, Target Agent: track in red)

Figure 5.7: Trajectory of wolf agent and target agent during a training episode

There are two major reasons and advantages why Autonomous Animal Behaviour should be applied on the target agent. Firstly, since it is the locomotive wolf agent who is our target for reinforcement learning, it is meaningless and less efficient to apply another locomotive model on the target agent. The most important function of the target agent during training is to just provide position coordinates and heading vector for evaluation of the reward function, which is exactly what a Constrained Ellipsoid Model with Autonomous Animal Behaviours can provide. Secondly, same as trained agents with NN model, autonomous animal agents are also self-guiding when given a specific task by the decision layer or the user. Proven by the behaviour algorithms that are introduced in Chapter 4, these animal agents are not simply just autonomous vehicles but NPCs with a high level of individual and collective intelligence. This rule-based method can provide a very decent adversary target during training of a RL-based locomotive agent with a very short amount of pre-training workload and improvement in training efficiency.

The basic training configurations are:

- **Objective:** Agent needs to catch the target as fast as possible while keeping its balance without falling to the ground. (Avoid its body mesh to collide with the ground)
- **Observations:** The agent has a vector observation space size of 204, containing ground contact information of body parts, body position, rotation, velocity, angular velocity of each limb, target position and velocity.
- **Rewards:**
 1. The product of how body velocity matches with the goal velocity and how body direction matches with the predicted direction toward the target. (a value between 0 and 1)
 2. Body mesh collides with the target: +1 (current episode ends)
 3. Body mesh collides with the ground: -1 (current episode ends)
- **Actions:** A total number of 36 continuous action corresponding to the rotation and strength of each limb.

Table 5.1: PPO training hyperparameters

batch_size	buffer_size	learning_rate	beta	epsilon	lamda	num_epoch
2048	20480	0.0003	0.005	0.2	0.95	3

- **Target:** In the beginning of each episode, a new position of the target is randomly chosen and generated within the range of the training platform. The target immediately execute a combination of Autonomous Animal Behaviours: *Hover()*, *Flee()* and *CollisionAvoidance()*. The target also has a detection radius which is a sphere collider with its centre at the centre of gravity of the target.
- **Hyperparameters:** see Table 5.1

5.4.3 Experimental Results and Analysis

The training environment consists of nine identical terrains and quadruped agents training simultaneously to speed up training. The entire neural network takes 243 floats as input and output 39 floats as continuous action for rotation and strength of each joint. The main policy network consists of 3 hidden layers and 512 hidden units.

Figure 5.8: Cumulative reward: Vanilla Dynamic Target vs Autonomous Animal Target

For comparison with existing methods, as shown in Figure 5.8 and Figure 5.9, after 15,000,000 steps of training with PPO and Soft Actor Critic (SAC), cumulative reward of both algorithms with an Autonomous Animal Target reached to around 2200 which surpassed the benchmark reward of 2000 training with vanilla dynamic target given by ML-Agents researchers [66].

Figure 5.9: Episode length: Vanilla Dynamic Target vs Autonomous Animal Target

Two timers recording the episode length for both trained agents are set for a performance comparison test. As shown in Figure 5.10, results are generated based on a total number of 100 episodes. Mean episode length for the Vanilla wolf agent is 8.1658(s); and that of the quadruped locomotive agent is 6.1141(s), which outperforms the Vanilla agent by approximately 25% faster in a more arbitrary and challenging environment.

Figure 5.10: Episode length comparison

5.5 Summary

The motivation of defining Locomotion layer is to convert the outputs of the Steering and Navigation layer into actual skeleton motion of joints and bones. The contributions in this chapter can be briefly concluded as:

- Previous methods concerning Inverse Kinematics and Reinforcement Learning are compared and evaluated
- A new Generic Quadruped IK Model is proposed
- Walking gait cycle of a quadruped animal is studied
- A novel AAB based Reinforcement Learning of quadruped locomotion is introduced
- Experimental results of different types of introduced approaches are presented and evaluated

In conclusion, locomotive behaviours of the NPC will largely affect the overall quality of adversarial games. Motion control, behaviour adaptation and RL of the NPC show significant improvement on user experience. Major technical contributions of this study are made to inverse kinematics embedded motion control, quadruped locomotion behaviour adaptation and dynamic environment informed reinforcement learning of the NPC agent. Instead of designing specific dynamics algorithms, the NPC agent can be trained with RL in the entire behaviour system.

The proposed concepts and techniques have been illustrated by a specific use case of in-game predator and prey interaction. Aimed at an intelligent locomotive predator to reach its autonomous animal prey as fast as possible in all the circumstances, the PPO learning scheme is combined with the autonomous animal target for a more challenging training environment. The target is also an intelligent agent that is capable of hovering, fleeing and obstacle avoiding. Such an autonomous animal target enables a variety of different behaviours. In fact, the same approach can also be used to train intelligent fleeing IK agent and much more.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

In this thesis, using the scenario of a VR hunting game, an integrated motion behaviour system of in-game animal NPC is introduced and explained in detail.

For Decision layer, behaviour tree of a typical predator animal is designed and implemented in Unity. Experiments are conducted based on a predator-prey interaction scenario. The results have shown that the wolf agent made different choices and actions based on his current status and the external environments. As external environment updates, the wolf agent is also able to continuously change its actions to tactically achieve its objective. Furthermore, the wolf agents also know how to make use of a cooperation mindset when group chasing a prey agent, which largely increases their chance to catch their target and at the same time the vraisemblance of the game.

For Steering and Navigation layer, a Constrained Ellipsoid Model is proposed as a rigid simplification model to derive a wide range of different motion behaviours. New adaptive behaviour algorithms are designed for the autonomous animals to navigate around the environment. Several individual behaviours including navigation, pursuit, fleeing and hovering are introduced; the corresponding experiment results are presented, and the comparative advantages over previous works are highlighted. Aiming to focus on collective intelligence in decision making and tactical thinking, advanced strategies for a group of autonomous animals are developed in order to simulate a more realistic forest environment. Group behaviours and strategies of the autonomous animals embody their collective intelligence in group cooperation scenarios. Finally, force combination

algorithms are explained to reflect the diversity and flexibility of this approach. Computational experiments and performance comparisons are also presented. The Combination of Constrained Ellipsoid Model and Autonomous Animal Behaviours can be easily modularized and reapplied into a wide range of NPC motion control use case scenarios.

For Locomotion layer, a new Generic Quadruped IK Model is introduced. The walking gait cycle of a quadruped animal character is studied to increase the vraisemblance of its motion. With the transition between Stance phase and Swing phase of each of the four legs, specific rule algorithms can be applied for basic animations such as walking and running. Combining the Generic Quadruped IK Model, Autonomous Animal Behaviour and FABRIK, foot-sliding issue of the Constrained Ellipsoid Model is solved and root motion with more realistic animations is achieved.

Finally, with initial thoughts to further increase the level of intelligence of the agents, the idea of Quadruped IK reinforcement learning is proposed and implemented. Different from existing approaches to train self-driving IK agent, the GQIKM is trained with a target agent who is capable of pursuing, fleeing and obstacle avoiding using AABs. The experiments are conducted on a target chasing scenario and the results have shown that learning agent trained with Autonomous Animal target fulfils the benchmark requirements and its performance surpasses agent trained with a vanilla dynamic target in a more challenging environment. With a variety of different behaviours that Autonomous Animal target can provide, this approach enables the possibilities of machine learning on IK agents with even more types of motion behaviours.

6.2 Future Directions

As concluded in the above section, an integrated motion behaviour system for any generic locomotive quadruped model is successfully achieved with innovations in each specific layers. However, since this is a very widely discussed topic in the field of computer graphics with huge research potentials, there are still many alternative approaches available and a large room for improvement. For further development in my field of research, I am planning on creating new compatible algorithms especially in Locomotive layer in order to achieve a more realistic visual effect to reach the requirements for industrialization and commercialization standard.

As mentioned above in my previous studies, two approaches are given to handle the locomotion of the quadruped animal agent. The first approach given is through directly giving the quadruped end effectors destination targets based on a general pre-defined walking gait cycle and move the agent's skeletons using a simple IK solver such as FABRIK. The advantage of this physics-based approach is obvious: its simplicity to implement and stability when handling with unseen environments. This method is seamlessly combined with the Autonomous Animal Behaviour algorithms which makes these behaviour algorithms' implementation on a rigged IK model possible instead of a simplified Constrained Ellipsoid Model. However, since both of these approaches are mostly rule-based, the pathfinding and locomotion of the agent largely suffer from animation stiffness and duplication issues.

The alternative approach presented is by training these rigged quadruped models using reinforcement learning. Although the performance of the agent is largely improved by using an adversarial autonomous animal opponent, it suffers from the same issue: unrealistic body movement. This approach may be very useful when applied on robotic navigation and pathfinding for a higher robustness on challenging terrains, but when used on simulating the locomotion of a virtual quadruped animal in computer graphics, this is still a fundamental issue remained unsolved. Therefore, my future research objective is to find a solution which can make the movement of the limbs and other body parts more natural and realistic while maintaining its robustness and advantages when achieving user given tasks.

To simulate a realistic locomotion behaviour of these quadruped agents as much as possible, there is a must-have prerequisite: collected data from the movement of a real quadruped animal. Data-driven character control methods, same as reinforcement learning approaches, are widely discussed and developed in today's related fields of study. To deal with noisy and unstructured motion capture data for training, there are a wide range of proposed methods in previous works, such as motion graphs [67, 68] and by utilizing finite state machines [69]. In the earlier days, researchers focus on synthesizing motion data using traditional machine learning method such as by Principle Component Analysis (PCA), Radial-Basis Functions (RBF), Gaussian Processes (GP) [70–73]. However, most of these traditional machine learning algorithms lack scalability which means that

they require high amount of captured motion data. Tedious pre-processing works need to be done before motion training.

More approaches direct toward learning captured motion data through Neural Networks. Distinguished methods include Recurrent Neural Networks (RNN) which predicts agent motion based on its previous state [74], conditional restricted Boltzmann Machines and their extensions [75, 76]. However, the capacity of these methods is only limited to predicting future moves of the agent based on previous motion states while incapable of taking in user's control inputs.

Recently, an interesting variety of approaches introduced facing toward this issue emerge very quickly and the results have proven to be very promising. The fundamentals of these methods are inspired by Mixture of Experts methods (MoE) [77, 78], which is a supervised learning architecture of experts. Each different expert is responsible for subdomain of a specific task. Beside the main network, the task of another gating network is to select specific experts to handle a certain input. Noteworthy approaches that utilize the combination of traditional deep learning with MoE gating network include Penalty-Free Neural Network (PFNN), Mode Adaptive Neural Network (MANN) etc. [55, 77, 79, 80].

Although there are a huge number of published papers which address motion control of general bipedal agents, the number of papers published which systematically address the issue with animal-like quadrupedal agent motion control is relatively limited since motion gait cycle of quadruped animals are way more complicated than human locomotion. Furthermore, unlike human motion data, classification and labelling of raw motion data of quadruped animals is very tedious since it is very hard to tell the animals to maintain a steady type of motion such as walk, run, canter, turning etc. during data collection thus resulting in very noisy and unorganized motion data. Therefore, I think that this research topic has a huge potential for further improvements and makes a perfect research topic for my PhD study based on what I have done so far during my Master's study.

Finally, to conclude this report, I would like to propose the following points for my possible research directions and objectives in the future:

- Collection and clustering of a set of noisy and unorganized raw quadruped motion data

- Novel data-driven neural network structure construction for quadruped motion control and prediction
- Imitation Learning and Reinforcement Learning performance comparison
- Novel hybrid methods design combining data-driven and physic-based approaches
- Tuning of the model to reach the requirement of quality motion control on variable and challenging terrains
- Adversarial and collective interaction design between predator and prey agent
- Implementation of the methods on NPC quadruped animal agent in an integrated motion behaviour system using Unity engine
- VR controllers and user body real-time tracking

Appendix A

Source Code

A.1 AutoAnimal.cs

```
using UnityEngine;
using System.Collections;
//using System.Collections.Generic;

public class AutoAnimals : MonoBehaviour {

    //forceList is the current set of different behaviour forces
    private Behaviour[] forceList;
    public float Vmax = 20;
    public float Fmax = 80;
    public float m = 2;
    public float inertia = 1;
    public float calcInt = 0.1f;
    public bool isPlanar = true;
    public Vector3 velocity;

    private Vector3 Force;
    protected Vector3 Acc;
    private CharacterController controller;
    private Rigidbody theRigidbody;
    private Vector3 moveDistance;
    private float timer;

    protected void Start ()
    {
        Force = new Vector3(0,0,0);
        moveDistance = new Vector3(0,0,0);
        forceList = GetComponents<Behaviour>();
        controller = GetComponent<CharacterController>();
        theRigidbody = GetComponent<Rigidbody>();
        timer = 0;
    }

    void Update ()
    {
        timer += Time.deltaTime;
        Force = new Vector3(0,0,0);

        //update force and acceleration
        if (timer > calcInt)
        {
            foreach (Behaviour f in forceList)
            {
                Force = Force + f.weight()*f.Force();
            }
            Force = Vector3.ClampMagnitude(Fmax, Force);
            acc = Force / m;
            timer = 0;
        }
    }

    void FixedUpdate()
    {
        //update position
        velocity += acceleration * Time.fixedDeltaTime;
```

```
velocity = velocity.normalized * maxSpeed;

moveDistance = velocity * Time.fixedDeltaTime;

if (isPlanar)
    moveDistance.y = 0;

if (controller != null)
    controller.SimpleMove(velocity);
else
    theRigidbody.MovePosition(theRigidbody.position + moveDistance);

if (velocity.sqrMagnitude > 0.00001)
{
    Vector3 Facing = Vector3.Slerp(transform.forward, velocity, inertia * Time.deltaTime);
    Facing.y = 0;
    transform.forward = Facing;
}
}
```

A.2 Navigate.cs

```
using UnityEngine;
using System.Collections;

public class Navigate : Steering {

    public GameObject target;
    private Vector3 desiredVelocity;
    private Vehicle m_vehicle;
    private float maxSpeed;
    private bool isPlanar;

    void Start () {
        m_vehicle = GetComponent<Vehicle>();
        maxSpeed = m_vehicle.maxSpeed;
        isPlanar = m_vehicle.isPlanar;
    }

    public override Vector3 Force()
    {
        desiredVelocity = (target.transform.position - transform.position).normalized * maxSpeed;
        if (isPlanar)
            desiredVelocity.y = 0;
        return (desiredVelocity - m_vehicle.velocity);
    }
}
```

A.3 Pursuit.cs

```
using UnityEngine;
using System.Collections;

public class Pursuit : Steering {

    public GameObject target;
    private Vector3 desiredVelocity;
    private Vehicle m_vehicle;
    private float maxSpeed;

    void Start () {
        m_vehicle = GetComponent<Vehicle>();
        maxSpeed = m_vehicle.maxSpeed;
    }

    public override Vector3 Force()
    {
        Vector3 toTarget = target.transform.position - transform.position;

        float relativeDirection = Vector3.Dot(transform.forward, target.transform.forward);

        if ((Vector3.Dot(toTarget, transform.forward) > 0) && (relativeDirection < -0.95f))
        {
            desiredVelocity = (target.transform.position - transform.position).normalized * maxSpeed;
            return (desiredVelocity - m_vehicle.velocity);
        }
    }
}
```

```
float lookaheadTime = toTarget.magnitude /
    (maxSpeed + target.GetComponent<Vehicle>().velocity.magnitude);

desiredVelocity = (target.transform.position +
    target.GetComponent<Vehicle>().velocity * lookaheadTime - transform.position).normalized * maxSpeed;
return (desiredVelocity - m_vehicle.velocity);
}
}
```

A.4 Flee.cs

```
using UnityEngine;
using System.Collections;

public class Flee : Steering {

    public GameObject target;
    public float fearDistance = 20;

    private Vector3 desiredVelocity;
    private Vehicle m_vehicle;
    private float maxSpeed;

    void Start () {
        m_vehicle = GetComponent<Vehicle>();
        maxSpeed = m_vehicle.maxSpeed;
    }

    public override Vector3 Force()
    {
        //if the target is in the fear range, then flee, else ignore it by return 0;
        Vector3 tmpPos = new Vector3(transform.position.x, 0, transform.position.z);
        Vector3 tmpTargetPos = new Vector3(target.transform.position.x, 0, target.transform.position.z);

        if (Vector3.Distance(tmpPos, tmpTargetPos) > fearDistance)
            return new Vector3(0,0,0);

        desiredVelocity = (transform.position - target.transform.position).normalized * maxSpeed;
        return (desiredVelocity - m_vehicle.velocity);
    }
}
```

A.5 Hover.cs

```
using UnityEngine;
using System.Collections;

//pay attention that this function is quite related with frame rate

public class Hover : Steering {

    public float wanderRadius;
    public float wanderDistance;
    public float wanderJitter;
    public bool isPlanar;
    //public GameObject targetIndicator;

    private Vector3 desiredVelocity;
    private Vehicle m_vehicle;
    private float maxSpeed;
    private Vector3 circleTarget;
    private Vector3 wanderTarget;

    void Start () {
        m_vehicle = GetComponent<Vehicle>();
        maxSpeed = m_vehicle.maxSpeed;
        isPlanar = m_vehicle.isPlanar;

        circleTarget = new Vector3(wanderRadius*0.707f, 0, wanderRadius * 0.707f);
    }

    public override Vector3 Force()
    {
        Vector3 randomDisplacement = new Vector3((Random.value-0.5f)*2*wanderJitter,
            (Random.value-0.5f)*2*wanderJitter, (Random.value-0.5f)*2*wanderJitter);
```

```
    if (isPlanar)
        randomDisplacement.y = 0;

    circleTarget += randomDisplacement;
    circleTarget = wanderRadius * circleTarget.normalized;

    wanderTarget = m_vehicle.velocity.normalized * wanderDistance + circleTarget + transform.position;

    desiredVelocity = (wanderTarget - transform.position).normalized * maxSpeed;
    return (desiredVelocity - m_vehicle.velocity);
}
}
```

A.6 Radar.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Radar : MonoBehaviour {

    private Collider[] colliders;
    private float timer = 0;
    public List<GameObject> neighbors;

    public float checkInterval = 0.3f;
    public float detectRadius = 10f;
    public LayerMask layersChecked;

    void Start () {
        neighbors = new List<GameObject>();
    }

    void Update () {
        timer += Time.deltaTime;

        //ticked
        if (timer > checkInterval)
        {
            neighbors.Clear();
            colliders = Physics.OverlapSphere(transform.position, detectRadius); //, layersChecked);
            for (int i=0; i < colliders.Length; i++)
            {
                if (colliders[i].GetComponent<Vehicle>())
                    neighbors.Add(colliders[i].gameObject);
            }

            timer = 0;
        }
    }
}
```

A.7 CollisionAvoidance.cs

```
using UnityEngine;
using System.Collections;

public class CollisionAvoidance : Steering
{
    public bool isPlanar;
    private Vector3 desiredVelocity;
    private Vehicle m_vehicle;
    private float maxSpeed;
    private float maxForce;
    public float avoidanceForce;
    public float MAX_SEE_AHEAD = 2.0f;
    private GameObject[] allColliders;

    void Start ()
    {
        m_vehicle = GetComponent<Vehicle>();
        maxSpeed = m_vehicle.maxSpeed;
        maxForce = m_vehicle.maxForce;
        isPlanar = m_vehicle.isPlanar;
        //avoidanceForce = 20.0f;
        if (avoidanceForce > maxForce)
            avoidanceForce = maxForce;
    }
}
```

```
//MAX_SEE_AHEAD = 20.0f;
allColliders = GameObject.FindGameObjectsWithTag("obstacle");
}

public override Vector3 Force()
{
    RaycastHit hit;
    Vector3 force = new Vector3(0,0,0);
    //Debug.DrawLine(transform.position, transform.position + transform.forward * 10);

    Vector3 velocity = m_vehicle.velocity;
    Vector3 normalizedVelocity = velocity.normalized;

    Debug.DrawLine(transform.position, transform.position + normalizedVelocity *
        MAX_SEE_AHEAD * (velocity.magnitude / maxSpeed));

    //if (Physics.Raycast(transform.position, normalizedVelocity, out hit, MAX_SEE_AHEAD))
    if (Physics.Raycast(transform.position, normalizedVelocity, out hit,
        MAX_SEE_AHEAD * velocity.magnitude / maxSpeed))
    {
        //Vector3 ahead = transform.position + normalizedVelocity * MAX_SEE_AHEAD;
        Vector3 ahead = transform.position + normalizedVelocity *
            MAX_SEE_AHEAD * (velocity.magnitude / maxSpeed);
        force = ahead - hit.collider.transform.position;
        force *= avoidanceForce;

        if (isPlanar)
            force.y = 0;

        //Debug.DrawLine(transform.position, transform.position + force);
        //change color use when there is only one AI in the scene, or multiple actions will conflict
        foreach (GameObject c in allColliders)
        {
            if (hit.collider.gameObject == c)
            {
                c.GetComponent<Renderer>().material.color = Color.green;
            }
            else
                c.GetComponent<Renderer>().material.color = Color.white;//Color.gray;
        }
    }
    else
    {
        foreach (GameObject c in allColliders)
        {
            c.GetComponent<Renderer>().material.color = Color.white;//Color.gray;
        }
    }
}

/*
if (Physics.Raycast(transform.position, transform.forward, out hit, maxSpeed))
{
    if (hit.collider.tag != "ground")
    {
        Vector3 ahead = transform.position + transform.forward * maxSpeed;
        force = ahead - hit.collider.transform.position;
        force *= maxForce;

        if (isPlanar)
            force.y = 0;

        Debug.DrawLine(transform.position, transform.position + force);
    }
}*/
return force;
}
}
```

A.8 LeaderFollowing.cs

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(SteeringForArrive))]

public class LeaderFollowing : Steering
{
    public Vector3 target;
    private Vector3 desiredVelocity;
    private Vehicle m_vehicle;
```

```
private float maxSpeed;
private bool isPlanar;
public GameObject leader;
private Vehicle leaderController;
private Vector3 leaderVelocity;
private float LEADER_BEHIND_DIST = 2.0f;
private SteeringForArrive arriveScript;
private Vector3 randomOffset;

void Start ()
{
    m_vehicle = GetComponent<Vehicle>();
    maxSpeed = m_vehicle.maxSpeed;
    isPlanar = m_vehicle.isPlanar;
    leaderController = leader.GetComponent<Vehicle>();
    arriveScript = GetComponent<SteeringForArrive>();
    arriveScript.target = new GameObject("arriveTarget");
    //arriveScript.target = leader;
    arriveScript.target.transform.position = leader.transform.position;//new Vector3(0,0,0);
    //randomOffset = new Vector3(0,0,0);//(Random.Range(0,3),Random.Range(0,3),Random.Range(0,3));
}

public override Vector3 Force()
{
    leaderVelocity = leaderController.velocity;
    target = leader.transform.position + LEADER_BEHIND_DIST * (-leaderVelocity).normalized;
    arriveScript.target.transform.position = target;// + randomOffset;

    return new Vector3(0,0,0);
}
}
```

Bibliography

- [1] A. Kumar, F. Lin, and J. C. Rajapakse, “Mixed spectrum analysis on fmri time-series,” *IEEE Transactions on Medical Imaging*, vol. 35, no. 6, pp. 1555–1564, 2016.
- [2] J. De, X. Zhang, F. Lin, and L. Cheng, “Transduction on directed graphs via absorbing random walks,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 7, pp. 1770–1784, 2017.
- [3] M. C. Leong, D. K. Prasad, Y. T. Lee, and F. Lin, “Semi-cnn architecture for effective spatio-temporal learning in action recognition,” *Applied Sciences*, vol. 10, no. 2, p. 557, 2020.
- [4] C. Lou, C. Pang, C. Jing, S. Wang, X. He, X. Liu, L. Huang, F. Lin, X. Liu, and H. Wang, “Dynamic balance measurement and quantitative assessment using wearable plantar-pressure insoles in a pose-sensed virtual environment,” *Sensors*, vol. 18, no. 12, p. 4193, 2018.
- [5] Q. Cheng, W. Song, F. Lin, Y. Liu, Y. Wang, and Y. Zheng, “Resolution enhancement of near-eye displays by overlapping images,” *Optics Communications*, vol. 458, p. 124723, 2020.
- [6] J. Cai, F. Lin, and H. S. Seah, *Graphical Simulation of Deformable Models*. Springer, 2016.
- [7] J. Cai, F. Lin, Y. T. Lee, K. Qian, and H. S. Seah, “Modeling and dynamics simulation for deformable objects of orthotropic materials,” *The Visual Computer*, vol. 33, no. 10, pp. 1307–1318, 2017.
- [8] C. W. Reynolds, “Steering behaviors for autonomous characters,” in *Game developers conference*, vol. 1999. Citeseer, 1999, pp. 763–782.
- [9] Z. Wu, H. S. Seah, and F. Lin, “Nurbs volume for modelling complex objects,” in *Volume graphics*. Springer, 2000, pp. 159–167.
- [10] W. Zhongke, L. Feng, S. H. Soon, and C. K. Yun, “Evaluation of difference bounds for computing rational bézier curves and surfaces,” *Computers & Graphics*, vol. 28, no. 4, pp. 551–558, 2004.
- [11] M. M. Movania and L. Feng, “Realtime physically-based deformation using transform feedback,” *The OpenGL Insights*, pp. 231–246, 2012.
- [12] M. M. Movania, W. M. Chiew, and F. Lin, “On-site volume rendering with gpu-enabled devices,” *Wireless Personal Communications*, vol. 76, no. 4, pp. 795–812, 2014.
- [13] M. M. Movania, F. Lin, K. Qian, W. M. Chiew, and H. S. Seah, “Coupling between meshless fem modeling and rendering on gpu for real-time physically-based volumetric deformation,” 2012.
- [14] T. Wang, Z. Jiang, Q. Kemao, F. Lin, and S. H. Soon, “Gpu accelerated digital volume correlation,” *Experimental Mechanics*, vol. 56, no. 2, pp. 297–309, 2016.
- [15] M. Iovino, E. Scukins, J. Styruud, P. Ögren, and C. Smith, “A survey of behavior trees in robotics and ai,” *arXiv preprint arXiv:2005.05842*, 2020.
- [16] M. Mateas and A. Stern, “A behavior language for story-based believable agents,” *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 39–47, 2002.
- [17] D. Isla, “Building a better battle,” in *Game developers conference, san francisco*, vol. 32, 2008.

BIBLIOGRAPHY

- [18] G. Flórez-Puga, M. A. Gomez-Martin, P. P. Gomez-Martin, B. Díaz-Agudo, and P. A. González-Calero, “Query-enabled behavior trees,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 298–308, 2009.
- [19] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.
- [20] G. Berthling-Hansen, E. Morch, R. A. Løvliid, and O. E. Gundersen, “Automating behaviour tree generation for simulating troop movements (poster),” in *2018 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA)*. IEEE, 2018, pp. 147–153.
- [21] B. Banerjee, “Autonomous acquisition of behavior trees for robot control,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 3460–3467.
- [22] K. Y. Scheper, S. Tijmons, C. C. de Visser, and G. C. de Croon, “Behavior trees for evolutionary robotics,” *Artificial life*, vol. 22, no. 1, pp. 23–48, 2016.
- [23] A. J. Champandard and P. Dunstan, “The behavior tree starter kit,” in *Game AI Pro 360*. CRC Press, 2019, pp. 27–46.
- [24] M. Colledanchise and P. Ögren, “How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees,” *IEEE Transactions on robotics*, vol. 33, no. 2, pp. 372–389, 2016.
- [25] M. Colledanchise, A. Marzinotto, and P. Ögren, “Performance analysis of stochastic behavior trees,” in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 3265–3272.
- [26] M. Colledanchise and L. Natale, “Analysis and exploitation of synchronized parallel executions in behavior trees,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 6399–6406.
- [27] M. Colledanchise, A. Marzinotto, D. V. Dimarogonas, and P. Oegren, “The advantages of using behavior trees in multi-robot systems,” in *Proceedings of ISR 2016: 47th International Symposium on Robotics*. VDE, 2016, pp. 1–8.
- [28] G. Robertson and I. Watson, “Building behavior trees from observations in real-time strategy games,” in *2015 International Symposium on Innovations in Intelligent Systems and Applications (INISTA)*. IEEE, 2015, pp. 1–7.
- [29] L. A. Ripamonti, S. Gratani, D. Maggiorini, D. Gadia, and A. Bujari, “Believable group behaviours for npcs in fps games,” in *2017 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2017, pp. 12–17.
- [30] D. Becroft, J. Bassett, A. Mejía, C. Rich, and C. Sidner, “Aipaint: A sketch-based behavior tree authoring tool,” in *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.
- [31] E. Coronado, F. Mastrogiovanni, and G. Venture, “Development of intelligent behaviors for social robots via user-friendly and modular programming tools,” in *2018 IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)*. IEEE, 2018, pp. 62–68.
- [32] O. Souissi, R. Benatitallah, D. Duvivier, A. Artiba, N. Belanger, and P. Feyzeau, “Path planning: A 2013 survey,” in *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)*. IEEE, 2013, pp. 1–8.
- [33] E. W. Dijkstra *et al.*, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [34] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
- [35] X. Cui and H. Shi, “A*-based pathfinding in modern computer games,” *International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125–130, 2011.
- [36] S. M. LaValle *et al.*, “Rapidly-exploring random trees: A new tool for path planning,” *Report No. TR 98-11, Computer Science Department, Iowa State University*, 1998.
- [37] K. Van Moffaert, M. M. Drugan, and A. Nowé, “Scalarized multi-objective reinforcement learning: Novel design techniques,” in *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE, 2013, pp. 191–199.

BIBLIOGRAPHY

- [38] R. Yang, X. Sun, and K. Narasimhan, “A generalized algorithm for multi-objective reinforcement learning and policy adaptation,” in *Advances in Neural Information Processing Systems*, 2019, pp. 14 636–14 647.
- [39] A. Sen, “Social choice theory,” *Handbook of mathematical economics*, vol. 3, pp. 1073–1181, 1986.
- [40] Q. Liu, “A social force model for the crowd evacuation in a terrorist attack,” *Physica A: Statistical Mechanics and its Applications*, vol. 502, pp. 315–330, 2018.
- [41] M. Xu, Y. Wu, P. Lv, H. Jiang, M. Luo, and Y. Ye, “misfm: On combination of mutual information and social force model towards simulating crowd evacuation,” *Neurocomputing*, vol. 168, pp. 529–537, 2015.
- [42] Z. Yao, G. Zhang, D. Lu, and H. Liu, “Data-driven crowd evacuation: A reinforcement learning method,” *Neurocomputing*, vol. 366, pp. 314–327, 2019.
- [43] R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2014.
- [44] T. M. Griffin, R. P. Main, and C. T. Farley, “Biomechanics of quadrupedal walking: how do four-legged animals achieve inverted pendulum-like movements?” *Journal of Experimental Biology*, vol. 207, no. 20, pp. 3545–3558, 2004.
- [45] D. Baraff, “Physically based modeling: Rigid body simulation,” *SIGGRAPH Course Notes, ACM SIGGRAPH*, vol. 2, no. 1, pp. 2–1, 2001.
- [46] A. Aristidou, J. Lasenby, Y. Chrysanthou, and A. Shamir, “Inverse kinematics techniques in computer graphics: A survey,” in *Computer Graphics Forum*, vol. 37, no. 6. Wiley Online Library, 2018, pp. 35–58.
- [47] J. U. Korein, *A geometric investigation of reach*. MIT press, 1985.
- [48] A. Liegeois *et al.*, “Automatic supervisory control of the configuration and behavior of multibody mechanisms,” *IEEE transactions on systems, man, and cybernetics*, vol. 7, no. 12, pp. 868–871, 1977.
- [49] R. Fletcher, *Practical methods of optimization*. John Wiley & Sons, 2013.
- [50] A. Aristidou and J. Lasenby, “Fabrik: A fast, iterative solver for the inverse kinematics problem,” *Graphical Models*, vol. 73, no. 5, pp. 243–260, 2011.
- [51] A. A. Canutescu and R. L. Dunbrack Jr, “Cyclic coordinate descent: A robotics algorithm for protein loop closure,” *Protein science*, vol. 12, no. 5, pp. 963–972, 2003.
- [52] C. F. Rose III, P.-P. J. Sloan, and M. F. Cohen, “Artist-directed inverse-kinematics using radial basis function interpolation,” in *Computer Graphics Forum*, vol. 20, no. 3. Wiley Online Library, 2001, pp. 239–250.
- [53] E.-J. Ong and A. Hilton, “Learnt inverse kinematics for animation synthesis,” *Graphical Models*, vol. 68, no. 5-6, pp. 472–483, 2006.
- [54] D. Holden, J. Saito, and T. Komura, “A deep learning framework for character motion synthesis and editing,” *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–11, 2016.
- [55] H. Zhang, S. Starke, T. Komura, and J. Saito, “Mode-adaptive neural networks for quadruped motion control,” *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–11, 2018.
- [56] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
- [57] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, “Continuous deep q-learning with model-based acceleration,” in *International conference on machine learning*. PMLR, 2016, pp. 2829–2838.
- [58] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, “Deep exploration via bootstrapped dqn,” *Advances in neural information processing systems*, vol. 29, pp. 4026–4034, 2016.
- [59] S. Li, Y. Wu, X. Cui, H. Dong, F. Fang, and S. Russell, “Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 4213–4220.
- [60] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust region policy optimization,” in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

BIBLIOGRAPHY

- [61] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.
- [62] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.
- [63] A. Aristidou, Y. Chrysanthou, and J. Lasenby, “Extending fabrik with model constraints,” *Computer Animation and Virtual Worlds*, vol. 27, no. 1, pp. 35–57, 2016.
- [64] T. Kikuchi, S. Tanida, K. Otsuki, T. Takehashi, and J. Furusho, “Development of intelligent ankle-foot orthosis (i-af) with mr fluid brake and control system for gait control,” in *Service robotics and mechatronics*. Springer, 2010, pp. 75–80.
- [65] P. C. Paquet and L. N. Carbyn, “Gray wolf,” *Wild mammals of North America: biology, management, and conservation*, pp. 482–510, 2003.
- [66] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar *et al.*, “Unity: A general platform for intelligent agents,” *arXiv preprint arXiv:1809.02627*, 2018.
- [67] L. Kovar, M. Gleicher, and F. Pighin, “Motion graphs,” in *ACM SIGGRAPH 2008 classes*, 2008, pp. 1–10.
- [68] O. Arikian and D. A. Forsyth, “Interactive motion generation from examples,” *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 483–490, 2002.
- [69] M. Lau and J. J. Kuffner, “Behavior planning for character animation,” in *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2005, pp. 271–280.
- [70] S. Wold, K. Esbensen, and P. Geladi, “Principal component analysis,” *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.
- [71] M. J. Orr *et al.*, “Introduction to radial basis function networks,” 1996.
- [72] L. E. Peterson, “K-nearest neighbor,” *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.
- [73] J. Quinonero-Candela and C. E. Rasmussen, “A unifying view of sparse approximate gaussian process regression,” *The Journal of Machine Learning Research*, vol. 6, pp. 1939–1959, 2005.
- [74] K. Fragkiadaki, S. Levine, P. Felsen, and J. Malik, “Recurrent network models for human dynamics,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 4346–4354.
- [75] J. Carreira, P. Agrawal, K. Fragkiadaki, and J. Malik, “Human pose estimation with iterative error feedback,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4733–4742.
- [76] Z. Li, Y. Zhou, S. Xiao, C. He, and H. Li, “Auto-conditioned lstm network for extended complex human motion synthesis. corr abs/1707.05363 (2017),” *arXiv preprint arXiv:1707.05363*, 2017.
- [77] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer,” *arXiv preprint arXiv:1701.06538*, 2017.
- [78] Y. H. Hu, S. Palreddy, and W. J. Tompkins, “A patient-adaptable ecg beat classifier using a mixture of experts approach,” *IEEE transactions on biomedical engineering*, vol. 44, no. 9, pp. 891–900, 1997.
- [79] D. Holden, T. Komura, and J. Saito, “Phase-functioned neural networks for character control,” *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–13, 2017.
- [80] C. Yang, K. Yuan, S. Heng, T. Komura, and Z. Li, “Learning natural locomotion behaviors for humanoid robots using human bias,” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 2610–2617, 2020.