An interactive wildlife development framework for 3D racing game

Tng, John Cheun Hou

2009

Tng, J. C. H. (2009). An interactive wildlife development framework for 3D racing game.
Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/15461>

<https://doi.org/10.32657/10356/15461>

NANYANG TECHNOLOGICAL UNIVERSITY

An Interactive Wildlife Development Framework for 3D Racing Game

**A thesis for the Master of Engineering program of Nanyang
Technological University**

School of Computer Engineering

By

Tng Cheun Hou John

2009

Introduction

Acknowledgements

This is to acknowledge my industrial supervisor Mr. Parimal Aswani, my academic supervisor Asst. Prof. Tay Joc Cing and my co-supervisor Assoc. Prof. Chaudhari Narendra Shivaji for their valuable guidance and involvements throughout this project. Special thanks to TQ Global for sponsoring this master program.

CONTENT

1 INTRODUCTION	1
1.1 MOTIVATION	1
1.2 OBJECTIVE.....	2
1.3 PROBLEMS	3
1.4 METHODOLOGY.....	4
1.5 ORGANIZATION OF THIS THESIS	6
 2 BACKGROUND AND RELATED WORKS	 7
2.1 AUTONOMOUS AGENTS AND ARTIFICIAL LIFE	7
2.2 NAVIGATION AND OBSTACLE AVOIDANCE IN GAMES	11
2.3 ANIMATION	12
2.4 3D SOUND EFFECTS	16
2.5 POPULAR RACING GAMES	17
2.6 OTHER GENRE	24
 3 DESIGN OF A WILDLIFE ANIMATION SYSTEM.....	 27
3.1 ARCHITECTURE AND DESIGN	28
3.2 ANIMAL BEHAVIOUR STATE MACHINE	32
3.3 ANIMAL MOVEMENT.....	38
3.4 VEHICLE AVOIDANCE	43
3.5 ANIMAL ANIMATION / BLENDING AND LOD	47
3.6 3D SOUND EFFECTS	50
3.7 INTEGRATION INTO GAME ENGINE	52
3.8 SPAWNING	54
3.9 OPTIMISATION	58

Introduction

3.10DEBUGGING AND DEVELOPMENT TOOLS.....	59
4 ANALYSIS OF ALGORITHMS.....	63
4.1 GENERAL MATHEMATICS	63
4.1.1 CROSS PRODUCT	63
4.1.2 DOT PRODUCT.....	64
4.2 ANIMAL BEHAVIOUR STATE FUNCTION.....	65
4.3 AVOID FORBIDDEN ZONE	66
4.4 AVOID STEEP SLOPE.....	67
4.5 AVOID VEHICLE.....	68
4.6 AVOID OBJECT.....	69
4.7 AVOID WILDLIFE	70
4.8 FLOCKING	72
4.9 ANIMATION LOD	75
4.10SPAWNING SYSTEM	76
5 EMPIRICAL STUDY: SCENARIOS	78
5.1 THE BEGINNING	78
5.2 BIRDS	80
5.3 GROUND ANIMALS.....	84
5.4 SPECIAL EFFECTS.....	88
5.5 COMPARISON WITH OTHER GAMES	94
6 SUMMARY AND CONCLUSION	96
6.1 GAME PLAY	96
6.2 PERFORMANCE	96
6.3 WILDLIFE DESIGN	97
7 FUTURE WORK: WILDLIFE SYSTEM.....	99

FIGURES

Figure 1: Architecture of autonomous agent.	8
Figure 2: Screen shot of MotorStorm downloaded from GameSpot.	18
Figure 3: Screen shot of Dirt downloaded from GameSpot.	18
Figure 4: Screen shot of Burnout Paradise downloaded from GameSpot.	19
Figure 5: Screen shot of Rally Trophy downloaded from GameSpot.	20
Figure 6: First screen shot of animal crossing road in Rally Trophy.	21
Figure 7: Second screen shot of animal crossing road in Rally Trophy.	21
Figure 8: Third screen shot of animal crossing road in Rally Trophy.	22
Figure 9: Sega Rally screenshot with birds flying.	22
Figure 10: Sega Rally screenshot with cows beside the race track.	23
Figure 11: Screenshot of animals in Age of Empire 3.	24
Figure 12: Screenshot of animal in Wildlife Park.	26
Figure 13: Screenshot of animal in Petz Rescue Wildlife Vet.	26
Figure 14: Prototype of wildlife system.	27
Figure 15: Hierarchy of system.	28
Figure 16: Architecture of wildlife system.	30
Figure 17: Wildlife system interface.	31
Figure 18: Subsystems of wildlife system.	32
Figure 19: Animal states transitions.	33
Figure 20: Group states transitions.	33
Figure 21: Diagrammatic representation of FSM logic.	34
Figure 22: Path of animal avoiding different objects and areas.	39

Introduction

Figure 23: Obstacle avoidance needed.	39
Figure 24: Avoidance direction.	40
Figure 25: Obstacle avoidance not needed.	40
Figure 26: Graphical illustration of how flocking is done.....	41
Figure 27: Illustration of banking for bird.	42
Figure 28: Keys for movement elaboration figures.....	44
Figure 29: Vehicle in danger zone.....	44
Figure 30: Animal desired escape direction.	45
Figure 31: Vehicle in critical zone.....	46
Figure 32: Animal desired escape direction.	46
Figure 33: State transition after escaping.	46
Figure 34: Animation blend types.	48
Figure 35: Animation blending system.....	48
Figure 36: Phase shifted problem.	49
Figure 37: Animation LOD regions.....	50
Figure 38: Illustration of animal sound.	52
Figure 39: Example of game loop.	53
Figure 40: Desired spawn location.	56
Figure 41: Player sees animals within field of view.....	56
Figure 42: Player behind the animals now.	57
Figure 43: Animals respawn in front of player.....	57
Figure 44: Full spawning process.	57
Figure 45: Animal allocated frame.	58
Figure 46: Wildlife creation tool.	61
Figure 47: Behaviour state and transition creation tool.....	61
Figure 48: The beginning of the race.....	79

Introduction

Figure 49: Along the race.	80
Figure 50: A group of birds flock together.	81
Figure 51: A hawk flapping wings in the sky.	82
Figure 52: Hawks gliding in the sky.	83
Figure 53: A hawk banks as it turns.	84
Figure 54: Animals spawn in front of the animal.	85
Figure 55: Animals running away from vehicle.	86
Figure 56: Another animal running away from vehicle.	86
Figure 57: Animal chased into the forest.	87
Figure 58: Closed up animal chased into the forest.	88
Figure 59: Visual effect I.	89
Figure 60: Visual effect II.	89
Figure 61: Visual effect III.	90
Figure 62: Visual effect IV.	90
Figure 63: Quails escape effect I.	91
Figure 64: Quails escape effect II.	92
Figure 65: Quails escape effect III.	92
Figure 66: Quails escape effect IV.	93
Figure 67: Quails escape effect V.	93
Figure 68: Another quails escape scenario.	94
Figure 69: Mind map of wildlife system.	98

Introduction

TABLES

Table 1: Description of animation principles	12
Table 2: Key-framed animation Versus IK/FK animation	15
Table 3: Different FSM and their pros and cons	34
Table 4: Dynamic spawn VS static spawn	57

APPENDIX

Appendix 1: DirectX SDK animation shader code.....	100
Appendix 2: Example of the logic of the state machine.	103
Appendix 3: New state logic.....	104
Appendix 4: Example of scripting behaviour and transition for a hare.	106
Appendix 5: Example of sound effect setting for quail.	109
Appendix 6: Banking formula from aeromechanics for birds.	110
Appendix 7: Publications.....	111

Introduction

Abstract

An Interactive Wildlife Development Framework for 3D Racing Game

Tng Cheun Hou John

Master of Engineering

School of Computer Engineering

Nanyang Technological University

2009

This thesis describes the development and framework of an interactive wildlife system for the next generation of highly interactive computer games. Current games have racing themes that include animated animals which are non-interactive. This means that animals in these games do not react to a player's action. For example, players in such games cannot pursue an animal.

This wildlife system creates interactive animals for racing games where players can observe animals' behaviour through high resolution animations, hear sounds they produced or chase them around. This is a new kind of game play experience that has not been implemented before. Interactivity with the animals lets players feel that these animals are part of the game and not merely decorations. This enhances the realism of the game and makes the game more interesting. This wildlife system shows that with the use of graphics hardware and optimisation techniques, it is possible to implement a cost and resource efficient interactive and realistic wildlife system for 3D racing games.

CHAPTER 1

1 Introduction

This project explores how to use an interactive wildlife system to enhance the game play of 3D racing games. The contribution of this project is how to create an interactive wildlife system for racing game and what are the techniques and components used to achieve this objective. This thesis documents the research and development process of a Wildlife Animation System (WAS) that simulates wild animals in an open environment for 3D car racing games on the Xbox and PS3 console platforms. The objective of this wildlife system is to create animals (agents) that can navigate around the environment and show interactivity and reactivity by avoiding oncoming racing vehicles driven by players. This wildlife system is implemented using the C++ language.

1.1 Motivation

Animals with high quality animations are added to a virtual game environment to make it more interesting and exciting for players. Doing so also adds “life” and realism to the game environment. Imagine yourself as a racer driving along a road with realistic animals by the roadside.

Other than being mere “decorations”, these animals can contribute to the game play. To achieve this, interactivity is very important. For example, interactivity can be attained by vehicle avoidance. Animation (visual) and sound effects (audio) will demonstrate the interactivity properties of these animals. There is no racing game currently in the market that has interactive animals where players (drivers of the racing vehicle) can interact with,

Introduction

for example chasing them around in the environment. Having interactive animals in the race will provide a unique gaming experience for the players. Apart from racing with other racers, players can also explore the environment while looking at different animals, observing their behaviours and animations, and hearing the sounds they make.

1.2 Objective

The objective of this project is to create a WAS for a 3D car racing game title on Xbox and PS3 console platforms. Animals will be seen roaming around the environment with realistic animations and 3D sound effects. They will flee when players drive near them, giving the player an exciting “animal-chasing” game experience.

There needs to be two types of animals in the WAS from the player’s point of view; animals on land and animals in the air. Animals that live in the water are not needed as racing vehicles cannot race in water. There are some racing titles with birds in the games. However, these birds are not interactive and merely fly across the screen randomly. The animals in this WAS should be interactive and react to racing vehicles in the game. They should flee from oncoming vehicles, and at the same time avoid steep slopes, forbidden zones and objects while escaping.

Each type of animal should have appropriate behaviours known as states. For animals on land, they should have walk state, run state and idle state. Depending on the type of land animals, idle state includes behaviours looking around and feeding. Animals flying in the air are more complicated. They have idle state, fly state, take-off state and land state. Certain types of animals will gather as a group, a pack or a flock. For instance, birds flock together, cats and dogs spawn as a group.

Rendering and animating the animals are very important. They affect what the players are supposed to see. Whether an animal looks and behaves realistically depends very much

Introduction

on the rendering and animating process. Every state of an animal will involve high quality animation achieved through blending. Animation blending ensures smooth transitions between the different states of an animal. Lightings and shadows also allow the animals to blend into the environment nicely. 3D sound effects are played according to the animals' states. For example, quails chip and chirp when they are feeding, and the sound of them flapping their wings gets softer from left to right as they take off.

From the programmer's point of view, the WAS should also be optimised in terms of memory and performance. Optimisation is a major work for all game programmers. The WAS should not take more than 5 megabytes of memory and 1.2 milliseconds of execution time during the game at any moment. These bounds are allocated to the WAS to ensure that the whole game runs at 30 frames per second. It should be modular and portable for inclusion in other games. It should also be flexible and extensible so that it is easy to add new animals. A set of creation tools, tuning tools and debugging tools must be developed so that new animals can be created, tuned and debugged easily. To ensure that WAS can run efficiently in future generation games, it should also be multi-thread supported. Graphics hardware should be used for animation instead of the CPU so as to free up CPU resources for other game systems.

1.3 Problems

There is currently no racing game in the market with animals that players can interact with. Why would developers not want such a system in their games? First of all, every programmer would be concerned about the overhead of the wildlife system. Such interactive animals will take up a lot of CPU and memory resources. This is especially so if they want 3D models with lots of polygons, high quality textures, realistic animations and 3D sound effects for the animals.

Introduction

Another problem faced is with regards to collision detections and physics models. What kind of bounding areas should be used for animals; boxes, spheres, convex hulls or triangle lists? Triangle list is the most accurate bounding area. However, it is also the most expensive. On the other hand, the cheapest but least accurate bounding area is a sphere. Likewise for the physics model, the more accurate the physics model is, the more expensive the computation. A game programmer can omit the use of collision detection and physics model. But what happens when large animals run through objects? It will break the realism of the game.

A more scientific question to ask is to what extend should the animals' intelligence be modelled and simulated. Should each animal be an autonomous agent? Can the animals sense, think and act? A truly intelligent and interactive animal will sense, think and act by itself. However, developing such an agent will be very difficult and requires a lot of CPU and memory resources. All these questions discourage programmers from trying to create an interactive wildlife system for their games.

Regardless of the difficulties faced, it is valuable to implement an interactive system because it gives players a new kind of gaming experience. Each of the problems mentioned above has a trade-off between quality and resources. Hence, some trade-offs are made to achieve our goal. For example, there can be fewer animals with higher quality animation at one time. In addition, techniques can be employed to work around difficult problems such as collision detections.

1.4 Methodology

Based on the research done on animals in different games, a list of desired requirements and functions is generated. The researched games include Wildlife Tycoon: Venture Africa, Zoo Tycoon, Wildlife Park, etc. A demo wildlife system is then built based on the

Introduction

list. Building the wildlife demo in an early stage ensures that the project and its requirements and functions are feasible. It is implemented with the “big bang” software development strategy. Big bang strategy is adopted because the need to proof the wildlife system concept in a tight time constraints. The development stopped when the demo is functional and presentable.

However, the wildlife system cannot be directly ported into the game engine. Hence, it is redesigned and recoded to comply with the game engine architectures and standards. From this point, the software development strategy used is changed to “iterative”.

Additional requirements and functions are defined and added into the system in an iterative way to ensure that the fall back version is always correct and working. During the implementation, debugging tools are developed to ease debugging processes and to make sure that the system is working properly. Lines and boxes are drawn to visualise and verify the correctness of the wildlife system. The system is also checked for memory leakage. Each memory allocation is tracked by the memory system. When the game exits, the memory reports any memory that is not deleted. Models, animations and artworks of the animals are created by artists. Each model is carefully inspected for its quality and checked for its correctness.

Optimisation, the final stage is done to ensure that the system meets the CPU usage and memory requirements of the game. To do so, an external program PIX (performance investigator for Xbox) is used to find portions of the system that use unacceptable CPU and memory resources.

Lastly, game play tests ensure that the wildlife system gives players the desired game play experience. This is the part when the parameters of the wildlife system are tuned, such that the animals look nicer in the game.

Introduction

1.5 Organization of this thesis

The following chapter, Chapter 2 will be split into two parts. The first part is about some background and related works. The later part of the chapter consists of game reviews. Chapter 3 illustrates the whole wildlife system from its design to the implementation of each requirement and function. All major algorithms implemented in the wildlife system will be analysed in Chapter 4. The functions and game play will be described along with a game scenario in Chapter 5. Chapter 6 and Chapter 7 are the conclusion and future works of this project respectively.

CHAPTER 2

2 Background and Related Works

This chapter gives an introduction to some background fields. It is closely related to autonomous agents and artificial life, obstacle avoidance and animations. The animals are seen as agents and they navigate around the environment presenting different animations and sound effects to the players.

2.1 Autonomous agents and artificial life

The term “autonomous agents” is mentioned a few times in this thesis without clear definition. Many people have their own definition of “autonomous agents”. One of the better definitions is given by Stan Franklin.

Franklin [1, 2] states that “An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda so as to effect what it senses in the future”. A simplified definition of an autonomous agent stated by Blumberg [3] is “An autonomous agent is a software system with a set of goals which it tries to satisfy in a complex and dynamic environment.” Thalmann [4] described what makes up autonomy and listed some standard approach for developing autonomous agents. However, a full autonomous agent is much more complex. For instance, through learning and experience, a human being can avoid sharp pointed objects like needles, pins and spikes.

Background and Related Works

Figure 1 from [1] shows the architecture of an autonomous agent. The architecture shows that an autonomous agent is built up of many different subsystems such as learning mechanism, perception, action and memory.

Figure 1: Architecture of autonomous agent.

Franklin [1, 2] pointed out that this architecture is far from complete. It is very difficult to develop a full autonomous agent. Each subsystem is a complex architecture of its own. For example, the perception subsystem of a human includes the five senses: taste, touch, hearing, smell and sight. The human learning mechanism is too complex to be represented even with software programs in our current technology. Hans Peter Moravec, a research professor in the Robotics Institute of Carnegie Mellon University has some discussion on this in his publication [5]. Maes [6] suggested that the ultimate goal is to create an autonomous agent that is lifelike, fast, reactive, adaptive and robust. In order to achieve that, an architecture that embraces all lifelike functionalities such as learning, sensing, reasoning, thinking, feeling, interacting, etc. would be required.

Regardless of all the difficulties faced while trying to model an actual living thing with autonomous agents, the research is not discouraged. This is because autonomous agents

Background and Related Works

represent real world living things better and therefore produce more realistic simulations. With autonomous agents, emergent behaviours can be revealed naturally. In addition, agents can learn to live in virtual environments without further human interferences. However, the question of how to create an autonomous agent is still not fully answered. It remains an ambitious challenge for all researchers.

Autonomous agent is closely related to artificial life (in computer context). The field of artificial life tries to recreate biological life in computer software. Autonomy is more concerned with self behaving agents in the virtual world or environment. Artificial life is a broader field which involves autonomy and in-depth life creation and evolution. Usually, artificial life includes some form of genetic or evolution algorithms and neural networks.

It is not the goal of this project to create super autonomous agents as it takes up too much resource. However, animals in the wildlife system will have some similarities with other autonomous agent definitions. They can sense, reason and act. For instances, the animals will sense oncoming vehicles, think of an escape route and escape away from danger. The definition of autonomous agents in this thesis will be: “An autonomous agent is one that has some input sensor that can sense the environment, have some decision process that can create some output actions.”

Funge [7, 8] built a remarkable behavioural animation framework in dynamic environment for autonomous agents using a cognitive model. His agents have the ability to sense, reason and act accordingly. Basically the agents will sense the environment for the current situation, make use of this current situation information and plan for their future actions. Funge suggest that agents should only sense when necessary to prevent wastage of processing power. The environment’s uncertainty is modelled and checked by

Background and Related Works

the agent. In this way, the agent can decide when it needs to re-sense and re-plan for future actions. The agents sense the environment by querying the game engine. Using sorted first-order logic known as situation calculus, the agents extract important information and plan their actions. The agents also have memory systems developed to store necessary information. Funge showed that his framework allows users to specify high level directions or behaviours to control the agents, yet at the same time, the agents can govern their own low-level directions or behaviours with their reactive systems.

Xiaoyuan [9, 10] modelled some realistic artificial fishes which use temperature and vision perception sensors based on ray casting to capture environmental information in a virtual environment. The fishes have memory systems and behavioural models that allow them to have three mental states, namely hunger, libido and fear. In addition, each fish have their own characteristics such as colour and size, as well as being a predator or a prey. The realistic locomotion of the fishes is governed by several physics laws.

Funge and Xiaoyuan are very famous researchers in the field of autonomous agents and artificial life. Their objectives were to create virtual agents as close to real life agents as possible. Unlike WAS, the objective is to create animals that give players fun game play experience. The animals have to be believable but do not require a real in-depth simulation. Animals do not need to have memory systems because the players will not notice even if the animals have memory systems. However, to achieve believability, the animals need to have minimal artificial intelligence models such as behaviour states and reflex actions. Therefore the animals have simple mental and behaviour states such as run (when animals are threatened), walk and idle states. The animals also do not have any cognitive model, but they have simple reflex actions such as “fear = run” and “safe = idle or walk”. Some randomness are implanted into the animals so that their behaviours appear more natural to the players rather than fixed and scripted. The main reason for not

Background and Related Works

having full animal simulation with cognitive model and memory is to reduce the computation resources as much as possible. The animals must be believable and efficient.

2.2 Navigation and obstacle avoidance in games

Navigation forms a huge component of computer animation and games. Believable navigation for agents can be achieved with a lot of work and effort. One of the main challenges for achieving believable navigation is obstacle avoidance. Agents must be able to avoid obstacles in their ways. Just like in real life, humans and animals try to avoid obstacles in their paths. They seldom collide with objects or obstacles unless it was intentional or under special conditions such as under medication or psychological impairment.

When objects collide, real world physics rules govern the collision and return a response caused by the collision. The priority when navigating is to avoid obstacles rather than knocking into obstacles and wait for a response. This is similar for games and animations, where it is better to avoid than collide when navigating.

In early days, Reynolds [11-13] simulated flocking behaviours and obstacle avoidances of birds based on a distributed behavioural model. Each bird is equipped with a sphere perception system that allows it to avoid colliding with neighbours and match the velocity of other birds. Reynolds achieved realistic flocking animation of birds by applying simple rules. He is the inventor of many flocking algorithms and obstacle avoidance algorithms known as steering behaviour. Steering behaviour is a set of well-defined and widely used algorithms for agents' navigation. Although his inventions were about 20 years ago, they are still widely used in games today. Steering behaviour is used mainly to find directions or velocities. It is very suitable for low level navigation implementation. Different algorithms from the steering behaviour set are combined to form more complex

Background and Related Works

behaviours. It has no memory or planning, and the direction or velocity computed depends only on the previous frame. The future direction or velocity cannot be computed in advance. Many people have enhanced and optimised his algorithms. Recently, he managed to implement his flocking algorithm on the PS3 [14] hardware and showed excellent results.

Reynolds's flocking and obstacle avoidance algorithms are used in this wildlife system with some customisation and optimisation. Air animals that flock in groups use an optimised flocking algorithm, while land animals use an optimised obstacle avoidance algorithm of the steering behaviour. Details of the flocking algorithm and steering behaviour algorithm and their respective modifications will be explained in Section 4.

For completeness, there is another class of navigation approach known as pathfinding. A widely used pathfinding algorithm is the A* (A star) algorithm. Rabin's book [15-18] illustrates many articles on A* pathfinding algorithms. Unlike steering behaviour, pathfinding has planning and memory. Given the start and end position, pathfinding algorithm finds the path between them. The whole path and all the directions are stored in memory for future usage.

2.3 Animation

Animation is important in this wildlife system because it is the medium to show the behaviours and interactivity of animals to the players. Whether the animals look intelligent, realistic or believable depends mainly on the animation. Lasseter wrote some interesting articles about animations in [19, 20]. He mentioned that for an animation to look real, it depends on 11 traditional principles. The 11 traditional principles are described in Table 1 taken from [19].

Table 1: Description of animation principles

Background and Related Works

Number	Principles	Description
1.	Squash and Stretch	Defining the rigidity and mass of an object by distorting its shape during an action
2.	Timing and Motion	Spacing actions to define the weight and size of objects and the personality of characters
3.	Anticipation	The preparation for an action
4.	Staging	Presenting an idea so that it is unmistakably clear
5.	Follow through and overlapping action	The termination of an action and establishing its relationship to the next action
6.	Straight ahead action and pose-to-pose action	The two contrasting approaches to the creation of movement
7.	Slow in and out	The spacing of the in-between frames to achieve subtlety of timing and movement
8.	Arcs	The visual path of action for natural movement
9.	Exaggeration	Accentuating the essence of an idea via the design and the action
10.	Secondary Action	The action of an object resulting from another action
11.	Appeal	Creating a design or an action that the audience enjoys watching

Background and Related Works

The models and animations for this wildlife system are created by the artists and checked based on these 11 traditional principles to ensure that the model and animations are of good quality.

Each animated model has “bones”. By changing the position of the bones, it will affect the skin, just like our human body. For each bone, there are 2 parameters associated with it. They are the “influence” and the “weight”. These 2 parameters affect how the skin deforms according to the bone. The computation of the bone’s effect on the vertices of the skin of the model is extremely expensive. Natively, it needs to be done for every bone and every vertex using the CPU. Graphics Processing Unit (GPU – found in graphics cards) is a parallel processing unit. Therefore, it is much faster to do the computations on the GPU using shaders instead.

This WAS uses vertex shader of the GPU to translate or deform the vertices of the model to achieve skinning effect for animations. Vertex shaders are software programs that make use of the GPU found in graphics cards to do vertex processing. The different implementations of animation using vertex shaders (with Microsoft DirectX) are found in two good books by Jim Adams [21] and Frank D. Luna [22]. DirectX is a set of APIs for Windows system that handles low level multimedia functions such as rendering. More detail definitions and usages of DirectX and vertex shaders can be found in the books [21, 22]. The complete shader code found in Appendix 1 is taken from DirectX SDK.

There are two main kinds of animation systems. They are key-framed animation system and skeletal IK/FK (inverse kinematics and forward kinematics) system. Details on how to implement a skeletal IK/FK animation system can be found in the books mentioned before [21, 22]. Our game engine uses the skeletal key-framed animation system. The main reason why the game engine uses the key-framed animation system instead of the

Background and Related Works

IF/FK system is key-framed animation is computationally cheaper. IF/FK animation system is more flexible and can dynamically produce procedural animations by moving the skeletal bones of a model during the game. Procedural animation is produced by functions or mathematic calculations. However, because of the flexibility of an IF/FK system, an animation may go wrong if the bone's constraints are not set correctly. For example, a hand may be over twisted. This is not a problem for key-framed animation because the artists will model the animation properly before putting them into the game. The table below compares the pros and cons of a key-framed and IK/FK animation system.

Table 2: Key-framed animation Versus IK/FK animation

Properties	Key-framed animation	IK/FK animation
Computational cost	Computationally cheaper.	More expensive because more calculations due to the skeletal bones.
Flexibility	Fix animation after export.	Flexible and can dynamic produce procedural animation.
Correctness	Error free. Visual tested before putting into the game.	May have animation artefacts because of unexpected situations such as an over-twisted arm.

The other component of animation in this wildlife system is animation blending. Each of the behaviour of an animal has a different key-framed animation. For example, when an animal is in the walking state, the walk animation is being played. When the animal transits to the running state, the run animation is played. Sudden animation change will cause “popping” effects. Players can see the discontinuity of the animations. Therefore, there is a need to blend different animations together to produce smooth animations for the animals.

Background and Related Works

2.4 3D Sound effects

3D sounds are directional sounds. For example, if the sound is coming from the left of the player, the player will hear a louder sound from the left speaker and a softer sound from the right speaker. Based on the sound amplitude we hear using our ears, our brains are able to locate the sound source accurately.

Other than animation, 3D sound is the next important medium to show the behaviours and interactivity of the animals to the players. Very few games have 3D sound features for their animals. This is because many programmers do not find it worthy to invest time to program such features, or memory and CPU resources to get such sound system running in the games just for their animals. So why is this project doing so? This is a new type of game play experience that the project is trying to bring forward to the players. As the wildlife system is part of the game play, animals in the game can catch players' attention by their animations and the sounds they produce. Therefore it is definitely worth it to invest resources to develop a 3D sound system for the wildlife system.

From the physiology of eye movements [23] by Rayner and Sereno, a human eye can only focus on a 2 degree of visual angle (180 degrees) around the point of fixation. This means that when a player is staring at the screen while playing the game, the player only focus on a 2 degrees angle of his visual angle. Therefore, it is not easy to catch sight of small animals. As for sound, we are able to hear 360 degrees around us. Details can be found in [24] by Ruggero. Humans can hear and determine where sounds come from. This allows us to search around visually for the position of the object that made the sound. Hence, if a player cannot see an animal, the sound will help the player locate the animal. More details about focus attention based on gaze and sound can be found in [25] by Stiefelhagen. Another important reason to include 3D sound effects for animals is to enhance the realism of the game.

Background and Related Works

Basically, 3D sound effects add realism to the wildlife system and to the game. 3D Sound effects help players to spot animals while they are racing. In addition, some animals produce sounds when they change states. This will exhibit some basic reflex intelligence of the animals as well.

2.5 Popular racing games

There are a few good racing game titles published in the recent two years. Each of them has its specialties and features that draw players to them. This section will only focus on comparing and reviewing their animal system if they have one. Details of the games and reviews can be found in GameSpot (<http://www.gamespot.com>), one of the best game review sites.

MotorStorm, a big racing game title developed by Evolution Studios was released in March 2007. It is rated by GameSpot as one of the best racing games, with a score of 7.9 out of 10.0. This game does not have a wildlife system. Nevertheless, it is still fun to play. Many gamers are looking forward to MotorStorm 2. Figure 2 shows a screenshot of the game MotorStorm.

Background and Related Works

Figure 2: Screen shot of MotorStorm downloaded from GameSpot.

Three months later, another big title called Dirt was released (June 2007). This game does not have a wildlife system as well. It was rated 8.3 out of 10.0 by GameSpot as its graphics is more realistic than MotorStorm. Figure 3 shows a screenshot of the game Dirt.

Figure 3: Screen shot of Dirt downloaded from GameSpot.

Burnout Paradise is the latest and highest rated racing game (9.0/10.0 by GameSpot) released just a few months ago in January 2008, by Electronic Arts (EA). Its racing environment is mainly in the city; hence it does not feature any animal. Figure 4 shows a screenshot of Burnout Paradise downloaded from GameSpot.

Background and Related Works

Figure 4: Screen shot of Burnout Paradise downloaded from GameSpot.

Rally trophy is an old game released in November 2001. The graphics cannot compare to current games like Dirt or MotorStorm. But at that time, it was rated 8.7 out of 10.0 by GameSpot. Figure 5 shows the screenshot of the game Rally Trophy.

Background and Related Works

Figure 5: Screen shot of Rally Trophy downloaded from GameSpot.

Rally Trophy is one of the first games that have wildlife in them. While racing, players can witness wildlife or livestock crossing the road. However, these animals will spawn very far ahead of the racing vehicle such that they will be gone before the racing vehicle reaches them. This means that the animals do not have any actual interaction with the vehicles. Furthermore, the animals do not make any 3D sound. Compared to our wildlife system, vehicles can chase animals around in the game, and the animals make 3D sounds effects. The three figures below show an animal (circled in red) crossing the road in Rally Trophy.

Background and Related Works

Figure 6: First screen shot of animal crossing road in Rally Trophy.

Figure 7: Second screen shot of animal crossing road in Rally Trophy.

Background and Related Works

Figure 8: Third screen shot of animal crossing road in Rally Trophy.

A new game released in October 2007, Sega Rally has some nice wildlife in it. The two screenshots below show the game with some animals. The screenshots are taken from RapidGames homepage: (http://games.rapidshare.com/games/2007/09/sega_rally/en/).

Figure 9: Sega Rally screenshot with birds flying.

Background and Related Works

Figure 10: Sega Rally screenshot with cows beside the race track.

In Sega Rally, birds fly around occasionally. There are no interaction between the racing vehicles and the birds. Bigger land animals such as cows or giraffes are not reachable by the racing vehicle. This is so as there is an invisible wall that forbids the vehicle from going off the race track.

In conclusion, most good racing games do not have animals in them. Those with animals in the games merely used them as decorations. This means that players cannot interact with the animals. In comparison, players in this wildlife system can chase the animals around. When being chased, the animals change their behaviours and animations. In addition, the animals also make 3D sounds that enhance the realism of the game. This is a new kind of game play experience that has never been implemented before in racing games.

Background and Related Works

2.6 Other Genre

For completeness, there is need to mention that there are other genres of games with wildlife. For example, a real time strategy (RTS) game Age of empire 3 (AoE 3) has wildlife. Land animals in AoE 3 play different roles. Animals like sheep and cows server as food sources in the game. When the player's army attack passive animals like sheep and cows, they flee. Aggressive animals like tigers are guardians of treasure in the game. Aggressive animals such as guardians will battle against the player's army if they are engaged. This similar animal behaviour and AI is found in Warcraft 3, another RTS game. There are birds in AoE 3 too. The birds fly and flock around in the air. However, the birds do not have banking and auto pilot effects as they turn as compared to this wildlife system. Figure 11 shows a screenshot of some animals in AoE 3.

Figure 11: Screenshot of animals in Age of Empire 3.

Background and Related Works

As can be seen from the screenshot, animals have different states. One of the deer is feeding, some are looking around and others are moving around. There are two types of animals in the screenshot; a group of deer and a group of tigers. The two types of animals are grouped separately. This wildlife system has both the features mentioned above. Each animal of this wildlife system has different states; and different kind of animals belong to different groups. Animals in AoE 3 make different sounds. However, they are not 3D sounds as compared to this wildlife system.

Another popular genre of games with wildlife is simulation. A few examples are Wildlife Tycoon, Wildlife Zoo, Wildlife Park, Petz Rescue Wildlife Vet, etc. The animals of these games usually have more interactions with other animals instead of the player's in-game character. Some of these games do not even have an in-game player character, like Wildlife Tycoon. Wildlife Tycoon is a pure simulation game where the objective is for players to set up an Africa safari. The game play of these games comes from the animals themselves. Therefore, the animals in such games are expected to be smarter and have more interactions with the game environments and other animals.

Figure 12 shows a screenshot of animals in Wildlife Park. Wildlife Park is also a simulation game. The objective of the game is to set up a wildlife park for visitors. Figure 13 shows a screenshot of an animal in Petz Rescue Wildlife Vet under medical care. This is an interesting game. Players find wounded, injured or sick animals along the storyline. The players are supposed to give medical treatments such as bandages, operations or other forms of medical care to animals.

Background and Related Works

Figure 12: Screenshot of animal in Wildlife Park.

Figure 13: Screenshot of animal in Petz Rescue Wildlife Vet.

III

CHAPTER 3

3 Design of a Wildlife Animation System

Figure 14 below shows a screenshot of the wildlife system prototype rendered using OpenGL. The demo has some birds and deer. The birds will fly off the ground and the deer will run away if a vehicle is too near. Deer will be grouped together and birds will flock together.

Figure 14: Prototype of wildlife system.

As mentioned in Chapter 1.4, the prototype is built before the wildlife system is redesigned, re-implemented and ported into the game. The following section of this

Design of a Wildlife Animation System

chapter consists of the architecture and the design process of the wildlife system. It is broken into different sub-sections based on the wildlife functionalities and requirements.

3.1 Architecture and design

This wildlife system adopts the idea by Maes [6]. It is made up of connected subsystems each capable of solving small and localized problems. By doing so the wildlife system becomes more modular. Any subsystem can be removed affecting the rest of the system. In addition, it simplifies the thinking process of the agent. Putting a lot of information to the main brain to process will make the thinking or decision process expensive and complicated.

The architecture is designed to allow real-time adding and changing of parameters of objects in the game environment. This saves developers from the traditional time-consuming method of modifying, restarting and evaluating of games. Having this structure, the wildlife system is easy to maintain and easy to create. Figure 15 below shows an example of the wildlife system.

Figure 15: Hierarchy of system.

Design of a Wildlife Animation System

The wildlife system is designed using simple hierarchy structure. Wildlife Pool is a collection of wildlife groups, while wildlife group is a collection of wildlife. As seen from Figure 15 , the system consists of three levels. The different levels are called pool level, group level and animal level. The boxes with numbers in them symbolise “**consist of**”.

The Wildlife Pool has a total of 40 animals, consisting of:

- 2 adult Deer groups, each consisting of 1 adult deer,
- 3 mix Deer groups, each consisting of 2 adult deer and 3 young deer,
- 2 young Deer groups, each consisting of 2 young deer,
- 3 adult Hare groups, each consisting of 5 adult hares, and
- 1 young Hawk group, each consisting of 4 young hawks.

The wildlife system linked all settings and information to a centralised base. When settings in the centralised base are changed, all animals using these settings will be changed appropriately in real-time. This feature is useful to tune all the parameters and settings of the wildlife system without restarting the whole game. To speed up the development of wildlife in games, an interactive and user-friendly graphical-user-interface (GUI) layer is built to modify information in the centralized base. The GUI layer will be described in Section 3.10. The architecture of the wildlife system is shown in Figure 16.

Design of a Wildlife Animation System

Figure 16: Architecture of wildlife system.

This wildlife system handles data in an efficient way different from the norm. For usual systems, each animal or group has its own set of settings and information. For example, hawks have a maximum running speed of 30mph. So every instance of hawk will have a variable containing the maximum speed, resulting in duplicated data. Furthermore, if the maximum speed of the hawks needs to be changed real-time, the speed variable of every hawk will have to be changed. This system groups same information together and places them in the centralized information base. In this way, the speed of the hawks only needs to be changed once in the main information base. Small speed variation (for randomness) for the same kind of animals can be set individually if needed. Creator can then view and compare the changes to the animals real-time in the game environment.

This architecture also empowers the creator to create a wildlife system from scratch quickly. The dotted-grey-bordered boxes in Figure 15 represent animal configurations, implying that the whole wildlife system only consist of 10 configurations: 1 pool configuration, 5 group configurations and 4 animal configurations. For a usual system, it will probably take up to 52 configurations (40 animal configurations, 11 group configurations and 1 pool configuration). This greatly reduced the number of configuration creation and time and effort in wiring up the configurations files.

Design of a Wildlife Animation System

The wildlife system is designed to be modular and portable to other games and multi-thread supported. It interfaces with the main game engine by three main functions. Create(), Tick() and Render() functions. As the name implies, Create() is the initial stage to set up the wildlife system. Tick() is called on every frame by the engine to do any processing of the wildlife system. Render() is also called every frame by the game engine. However, Render() only collects the model, sort them according to depth and render them. The Tick() and Render() interface functions can run on a same thread or two different threads. Figure 17 shows a figure of how the wildlife system is interfaced to the game engine.

Figure 17: Wildlife system interface.

This wildlife system is made up of different subsystems. Figure 18 shows the main subsystems of wildlife system.

Design of a Wildlife Animation System

Figure 18: Subsystems of wildlife system.

The behaviour of each animal is controlled by a state machine and the states are designed using scripts so that it can be flexibly changed without recompiling the game. The animals can navigate around the environment avoiding objects, steep slopes, forbidden areas and oncoming vehicles. Each animal has an audio system that allows them to produce 3D sound effects. Animals like birds have flocking and grouping behaviour. The blending supported animation system plays different animations for the animals according to their behaviours. A Level-Of-Detail (LOD) system is implemented for animation to reduce the computation cost of the wildlife system as a form of optimisation. A spawning system helps to spawn animals in front of the racing vehicles and it also helps to reduce the total number of animals in the game as a form of optimisation. The subsystems will be explained in detail in Sections 3.2 to 3.10.

3.2 Animal behaviour state machine

The wildlife behaviour system is implemented using two layers of Finite-State-Machine (FSM). FSM is a device that controls state transitions. The first layer controls the state behaviours of the animals. An example is illustrated in Figure 19.

Design of a Wildlife Animation System

Figure 19: Animal states transitions.

The second layer is the group layer. It controls the higher-level behaviour of a whole group of animals; for example, flocking behaviour. An example is illustrated in Figure 20.

Figure 20: Group states transitions.

When in a group escape state, the state machine will cause the list of animals in the group to transit to run state. In this way, hierarchical states control is achieved.

State transitions of the state machine are event triggered. For instance, a deer walking around in the environment will switch to run state when it encounters danger. Danger is the event and it triggers the state transition. The state machine can also trigger itself to change state. For example, the deer can change from walk state to run state whenever it desires. Similarly, it can change from run state back to walk state. This randomness is represented in the state. An example of the logic of the state machine can be found in Appendix 2.

State transitions now look more realistic, with state transition logic being made up of “if” statements and probability. This simple logic structure facilitates easy implementation in programming language.

However, executing the state machine logic for every frame in the game loop will pose a problem. With the logic above, the deer may choose to run, walk, and then run again

Design of a Wildlife Animation System

within very short intervals. This causes the deer to jitter between states. Conversely, the deer may choose to run forever. To overcome this problem, a time control module is implemented. There is a minimum and maximum time for a deer to stay in a particular state. No state transition is possible before the minimum time, and after the maximum time, the state must change. The state machine also contains the animation of the state to be rendered. Figure 21 below shows the diagrammatic representation of the new FSM logic above. MTS is the maximum time in state while mTS is the minimum time in state. The new state logic can be found in Appendix 3.

Figure 21: Diagrammatic representation of FSM logic.

State machines are powerful and flexible tools used in AI systems. However, states must be finite. Table 3 shows the different types of FSM and their pros and cons.

Table 3: Different FSM and their pros and cons

Type	Description	Pros	Cons
------	-------------	------	------

Design of a Wildlife Animation System

Data-driven FSM	FSM states and behaviours can be dynamically created from a FSM configuration file.	<ul style="list-style-type: none"> • Do not need to recompile the code when changing states and behaviours. • Dynamically create FSM based on configurations files. 	<ul style="list-style-type: none"> • Slower in loading and execution.
Stacked-based FSM	A FSM enhanced by a stack structure.	<ul style="list-style-type: none"> • Able to keep track of the previous states in an organised way. 	
Macro-based FSM	A FSM where the states and transitions can be designed by using macros of the compiler.	<ul style="list-style-type: none"> • Fast and easy to implement. 	<ul style="list-style-type: none"> • Codes not object oriented structured.
Basic FSM	Basic state machine that uses “switch-case” statements.	<ul style="list-style-type: none"> • Easiest to implement 	<ul style="list-style-type: none"> • Becomes messy when states are huge. • Codes not object oriented structured.

Design of a Wildlife Animation System

Hierarchical FSM	A Hierarchy-structured FSM that allows many layers of FSM to be used.	<ul style="list-style-type: none"> • Can handle complex FSM with many states in an organised way. 	<ul style="list-style-type: none"> • A hierarchical framework or system has to be build
------------------	---	--	--

More in-depth discussion about the different types of FSM and their pros and cons are found in [15-17, 26, 27].

Data-driven FSM is used in WAS. The advantages offered by data-driven FSM suits the WAS completely. Using this type of FSM, each type of animal can have different behaviours easily. All the states, transitions and conditions are stored in the animal setting file. Changing the behaviours of animals does not need to recompile the whole game.

To make the state machine programmable and flexible; it cannot be static or hard-coded into the game engine. It is represented as a behaviour state graph. The nodes are the states and the edges are transitions. During loading time, the animals' behaviour graphs are created from setting files. Nodes and edges can be added into the behaviour graph dynamically. Behaviour functions are bind to the nodes and the edges. Each node or edge holds a pointer that contains user-defined data. This implementation allows the behaviour graph to be flexible and extensible. Examples of user-defined data are AnimationType, AnimationTimeScale, Probability, etc.

On top of that, a simple scripting system is developed. The scripting system enables behaviour and transition logics to be changed without recompiling the code. However, the position and orientation calculation routines for individual states such as “walk”, “run” and “fly” are all precompiled. In addition, the scripting system also binds the animations file to the animals. This really improves development and debugging efficiency. An

Design of a Wildlife Animation System

animal's animation can be changed in the setting without recompiling the game as well.

An example of the full script used in the final version of the game is found in Appendix 4.

```
[Behavior1]

Name = "Idle"
AnimationType = "Idle"
AnimationFileName = "WlHare_Idle"
AnimationTimeScale = 1.000000
EnterScript = ""
ExecuteScript = "{ Idle; }"
ExitScript = ""
InitialBehavior = true


[Behavior2]

Name = "Walk"
AnimationType = "Walk"
AnimationFileName = "WlHare_Walking"
AnimationTimeScale = 1.000000
EnterScript = "{ DebugState; }"
ExecuteScript = "{ Walk; }"
ExitScript = ""


[Transition1]

Name = "IdleToWalk"
From = "Idle"
To = "Walk"
AnimationBlendDuration = 0.500000
Probability = 0.005000
TransitionLogic = "WildlifeTimeInState > 3.0 &&
Probability { BlendAnimation; PlayTransitionSound; }"
```

Above shows part of the script used for a hare. The hare has 2 behaviour states; the idle behaviour and the walk behaviour. Each behaviour state is bound to an animation file called AnimationFileName. There is a time scale for each animation. This controls the speed of the animation. EnterScript, ExecuteScript and ExitScript are functions bindings of the game engine. It binds the behaviour state function to the game engine function. For example, the hare will use the “Walk” function in the game engine as its walk algorithm. To change the walk function used, just specify another function instead. For example ExecuteScript = “{ HareWalk }”. This scripting system has debug capability as well. In

Design of a Wildlife Animation System

[behaviour 2], the “EnterScript” has a “DebugState” function. This function will be called whenever there is a state change to “Walk”. This function prints out useful debug information of the animal such as its velocity, previous state, animation, etc. If InitialBehavior is set to true, it will set that behaviour as the initial behaviour.

The transition block above shows the state transition information. There is a valid transition from idle state to walk state. The AnimationdBlendDuration is the time to blend the idle animation to the walk animation. If a hare is in the idle state, it has 0.5% chance of changing to the walk state every frame. The TransitionLogic are the conditions and the functions binding to the game engine. In the above case, a hare must be in the idle state for more than 3 seconds before it can change state. This is to prevent the jittery state change mentioned previously.

3.3 Animal movement

Instead of following physics and collision detection rules, the animals use collision avoidance. The difference is that collision avoidance attempts to avoid objects before a collision occurs, while collision detection occurs when a body collides into objects. This section touches on the movement system of animals. Animal movements are made up of different things, like random movement, slope avoidance, object avoidance, forbidden area avoidance, wildlife avoidance (wildlife avoid each other) and vehicle avoidance. Vehicle avoidance is explained in next section.

Figure 22 shows the path of “animal A” navigating the environment avoiding different objects and areas. The arrows indicate the direction of “animal A”. The animal first avoids an obstacle, and then it avoids another animal. Next, it avoids a forbidden area and lastly, it avoids a steep slope.

Design of a Wildlife Animation System

Figure 22: Path of animal avoiding different objects and areas.

Obstacle avoidance, animal avoidance and navigation is governed by the concept of Reynolds' [12, 28, 29] steering behaviour. A brief illustration of obstacle avoidance using animal's body width is shown in the diagrams below. Figure 23 shows a situation where obstacle avoidance is needed.

Figure 23: Obstacle avoidance needed.

The animal is on the right side of the obstacle. Therefore it should turn right to avoid the obstacle. Figure 24 shows the direction that the animal should head.

Design of a Wildlife Animation System

Figure 24: Avoidance direction.

Lastly, Figure 25 shows an animal with its body width at a distance from the obstacle.

Therefore avoidance is not required.

Figure 25: Obstacle avoidance not needed.

The forbidden area avoidance is simple. When an animal detects a forbidden area in front, it will turn around and walk towards the opposite direction for 2 seconds. When the 2 seconds are up, the animal will find a new random direction again. This simple algorithm is not robust, but it is efficient and sufficient for the game. When there is no avoidance needed, the animal will roam around randomly.

Flocking effect is achieved by Reynolds' flocking concept. Detail explanations and algorithms on both the steering behaviour and flocking algorithms are provided in [15-17,

Design of a Wildlife Animation System

26, 27]. The actual algorithms used for steering behaviour and flocking in this wildlife system are slightly different due to the required optimisation.

Figure 26 shows how flocking is done for the birds. The separation, centre of mass (COM) and flock heading direction are all 2D vectors. They are added together to get the final resultant desired direction vector.

Figure 26: Graphical illustration of how flocking is done.

From the above figure, flock heading direction is provided randomly. It will change after every few seconds. This is to simulate the changing directions of the flock of birds. The COM direction is for the birds to get close to the centre of the group. This prevents birds from flying too away from the group. Separation direction forces the birds to fly away from neighbouring birds if they are too near to prevent collision. By adding these 3 vectors, the final desired direction is produced. The above figure only illustrates a single flocking bird. This same procedure is done for all the birds. Combining all the birds, they form a realistic and optimised flock of birds. The flocking algorithms used in the wildlife system are shown and analysed in the next chapter with more details.

To improve and extend the movement system, acceleration and deceleration parameters are introduced. There is also a parameter to control the turning of animal so as to prevent it from changing direction suddenly.

Design of a Wildlife Animation System

To improve the realism of birds' behaviour, a banking system with auto pilot capability is introduced. Birds can fly and bank freely based on a formula. This formula is derived from aeromechanics. See Appendix 6 on how the formula is used in WAS. Reynolds' [28] article also contain some information about banking.

$$\omega V = g \tan \Phi$$

ω – Angular rate

V – Velocity

g – Gravity

Φ – Bank angle

A bird has 2 types of directions. First being the current direction (2D projected direction). Second being the desired direction (also 2D projected direction). A bird turns when the desired direction change. This causes the current direction to turn towards and align with the desired direction with the bird's turn rate. Turn rate is a property that controls how many degrees a bird can turn in 1 second. This turn rate together with the velocity of the bird will be used in the above formula to compute the banking angle of the bird. Figure 27 illustrates the turning of a bird.

Figure 27: Illustration of banking for bird.

The angle difference is used to compute the turn rate. As mentioned above, the turn rate together with the velocity of the bird will be used to compute the banking angle. If the angle difference is bigger, the bird will bank more, if the angle difference is smaller, the bird will bank less.

Design of a Wildlife Animation System

A more accurate and realistic model to represent body parts movement is biomechanics [9]. The biomechanics model describes locomotion as different muscle groups coordinating with each other to bring balance and movement to a living body. The auto pilot capability allows the bird to control its banking nicely. For example, in order to turn 90 degrees to the left, the bird has to bank at a certain angle for a certain time, and then bank back to upright position before the angle is reached, so that it does not overshoot the turn.

In short, the movement system makes use of procedural methods to find all necessary paths and directions and even banking angles. There is absolutely no scripted paths or directions. Animals have speeds, accelerations and decelerations to make them look believable. They use collision avoidance instead of collision detection and response to avoid objects and forbidden areas before they collide. A body width test explained previously is used to determine whether there is a need to avoid any objects.

3.4 Vehicle avoidance

Vehicle avoidance is one of the main features to show interactivity of the animals. Therefore, it is an important component of this system. The animals of this wildlife system are able to escape from danger unharmed in a logical way. For example, if a driver in a vehicle is chasing an animal, the animal is able to escape gracefully without getting hurt. The animals also have another interesting feature. Assuming a first person view (driver), the animal will try to be in the viewing frustum of the driver. This enables the game to showcase the animal system and allows the player to have more fun chasing the animals around in the game environment. The following figures show the technique used to achieve the above goals. Figure 28 shows the keys used in the next few figures.

Design of a Wildlife Animation System

Figure 28: Keys for movement elaboration figures.

Animals move differently based on which danger zone they are in. Dangers can be predators, humans or even vehicles. Figure 29 shows the three different zones, where the danger is a vehicle.

Figure 29: Vehicle in danger zone.

If the animal is in the safe zone, it will not escape. It will roam around or feed in the environment. However, if the animal is in the danger zone, it needs to escape to the safe zone while achieving a minimum angle between the direction of the vehicle and the

Design of a Wildlife Animation System

relative position of the vehicle to itself. Hence as the animal escapes, it turns gradually towards a direction within the two solid arrows and not its initial direction, based on its maximum turn rate to ensure the animal escapes gracefully. The final direction of the animal is shown in Figure 30.

Figure 30: Animal desired escape direction.

Now the animal is escaping in a direction in the acceptable range. It will continue to change its direction based on the relative direction of the vehicle and itself, while maintaining the minimum angle until the vehicle is out of the danger zone. After which, the animal will return to its normal lifestyle. This provides for a graceful escape while keeping the animal in the viewing frustum most of the time.

If the vehicle reaches the critical zone, the animal will adapt a new set of properties. It will now escape perpendicularly to the vehicle direction. Meanwhile, the speed will also increase. In this manner, the animal can escape unharmed. Point to note is that the turning is still bounded by animal's turn rate. The next two figures show the animal's change of direction when the vehicle enters its critical zone.

Design of a Wildlife Animation System

Figure 31: Vehicle in critical zone.

Figure 32: Animal desired escape direction.

After escaping, the animal will change its state. An example of the state change is shown in Figure 33, after an animal escapes from a vehicle.

Figure 33: State transition after escaping.

Design of a Wildlife Animation System

3.5 Animal animation / blending and LOD

Animation and blending are very important as the outlook of the animals must look realistic. Their colours, sizes, lighting, shadows and behaviours must blend into the environment nicely. Fortunately, lighting and shadowing are handed by the game engine.

Animations are done by interpolation of key frames. After which, these key frames are passed down to the shader and rendered out through vertex shading. Each key frame comprises of three components: vector for position, scalar for scale, and quaternion for orientation. Instead of key frame animation system, more complex systems such as inverse-kinematics (IK) system or skeletal system [30] can be used to perform animation. Both systems produce more realistic animation at the expense of taking up more resources from the hardware.

With the current setup, the state machine controls and chooses the animation to render. However, sudden animation changes occur when state changes. To produce high quality and smooth animations, a blending system must be employed. The blending system takes in two animations, interpolates them and produces an output animation for rendering. The speed of blending depends on the blend duration, while the smoothness depends on the blend type.

Design of a Wildlife Animation System

Figure 34: Animation blend types.

Three different ways of blending animations is shown in Figure 34. The blue and black arrows represent two different animations. The red arrow is the intermediate animation produced by linear interpolation of the two animations. There are other more complex blend types that can produce better results. The interpolation function (red arrow) may not be linear. Figure 35 below shows how the blender works. The black arrow shows the direction of animation data.

Figure 35: Animation blending system.

Even with this blend system, it does not completely eliminate the problem of sudden changes in animation. Figure 36 illustrates a potential problem that causes this. It is known as the phase shifted problem. The red arrow shows state transition. The blue arrow shows which animation to render.

Design of a Wildlife Animation System

Figure 36: Phase shifted problem.

As shown in the above diagram, phase shift problem exists when two animations have different animation time. The solution is to maintain the same animation time for both animations. In this wildlife system, when a new animation (right walk) blend starts, it will start from the animation time of the old animation (left walk). The wildlife system defines a blend output as presented in the segment below (found in Appendix 4).

```
[Transition1]

Name = "IdleToWalk"
From = "Idle"
To = "Walk"
AnimationBlendDuration = 0.500000
Probability = 0.005000
TransitionLogic = "WildlifeTimeInState > 3.0 &&
Probability { BlendAnimation; PlayTransitionSound; }"
```

To reduce the trouble of coding which animation to render, the FSM is equipped with the capability of searching for appropriate animation automatically. This allows for animations to be added to the wildlife system at real-time in the game environment. AnimationBlendDuration (in seconds) is the time to blend from “idle” animation to “walk” animation. Probability is the state transition probability per frame.

Design of a Wildlife Animation System

The animation system has a seamless (Level-Of-Detail) LOD system that optimises the animation computation. When animals are far away, they are reduced to only a few pixels on the screen. These animals can do accumulated animation computation. For example, an animal that is far away can compute the animation every 1 second instead of every frame. This saves a lot of CPU recourses. Animals that are nearer will compute animation more frequently. The number of computation per second reduces linearly to the distance away from the player. Figure 37 shows the animation LOD region. Within the circle radius of the player, there will be no animation LOD for the animals. Outside the circle, animation LOD applies.

Figure 37: Animation LOD regions.

3.6 3D sound effects

The game engine audio player is operated by “Miles Sound System (MSS)”. The processing and the mixing of 3D sounds are done by MSS. However, the sound system for wildlife is designed and implemented within the scope of this project.

To make sure that the animals are interactive, each animal has three types of sound; they are “Random”, “Transition” and “State” sound. Random sounds are randomly played based on probability in a given animal behaviour state. Transition sounds are played once when an animal goes through a behaviour state transition. State sounds are played repeatedly when an animal is in a given state. Each type of sound is tagged to an animal

Design of a Wildlife Animation System

behaviour or its state transition during load time. This is to avoid any overhead needed to search for the sound effect to play. The settings below shows the sound effect setting of a quail used in the game. A similar setting can be found in Appendix 5 as well.

```
[Header]
NumSounds = 3

[Sound1]
Filename = "QuailTakeOffTransition.mp3"
MinDistance = 1.0
MaxDistance = 500
Is3D = true
Volume = 0.5
Type = Transition
FromState = Idle
ToState = Fly

[Sound2]
Filename = "QuailIdleRandom.mp3"
MinDistance = 1.0
MaxDistance = 500
Is3D = true
Volume = 0.5
Probability = 0.2
Type = Random
State = Idle

[Sound3]
Filename = "QuailFlyState.mp3"
MinDistance = 1.0
MaxDistance = 500
Is3D = true
Volume = 0.5
Type = State
State = Fly
```

From the setting, the quail has 3 sounds. It has a transition sound from idle state to fly state. Whenever the quail changes from idle to fly, it will play the sound effect “QuailTakeOffTransition.mp3” once. The random sound “QuailIdleRandom.mp3” will be played randomly with a possibility of 0.2 in idle state. When the quail is in fly state, it will play the “QuailFlyState.mp3” sound repeatedly, which is the wing flapping sound.

Design of a Wildlife Animation System

Figure 38 shows the portion where sounds are played. When the quail is in idle state, it will randomly “chip”. When vehicle is near, it will fly off yelling. Lastly, when the quail is flying, the wing’s flapping sound is played repeatedly.

Figure 38: Illustration of animal sound.

Is3D is set true because the sound effects are 3D sounds. MinDistance and MaxDistance are the clipping distance for the sound in feet. Volume directly controls the volume of the sound effects. For transition sound, it is tagged to the transition from “FromState” to “ToState”, whereas for random and state sounds, they are only tagged to a “State”.

It is expensive to play 3D audio because of the mixing and processing. The game engine allows only 64 audios to play at a time. Therefore, not every animal can play sound effects together. To solve this problem, the animals are sorted according to the distance from the camera. Through experiments, it was found that the ideal number of sound effects played is those made by the three nearest animals. Players can hardly hear the difference of the sounds made by the fourth or further animals as they will be too far away to be audible.

3.7 Integration into game engine

The wildlife system provides two functions, Create and Destroy. The create function is called before any Tick. It initializes the whole wildlife system. Destroy function is called when the game shuts down. It deletes all allocated memory to prevent any memory

Design of a Wildlife Animation System

leakage. These two functions reduce significant programming needed by the wildlife creator.

Figure 39: Example of game loop.

In most games, the different subsystems of the game engine are processed first before rendering. An example of the game loop is presented above in Figure 39. The wildlife system provides two other main functions, “Tick” and “Render”. “Tick” function is the entry point for the game loop or game thread to enter the wildlife system for processing. “Render” function is the entry point for animations to render. The system separates the processing section from the rendering section. As seen from Figure 39, this facilitates the

Design of a Wildlife Animation System

integration into the game engine instead of messing up the game loop. Game thread and render thread are two different threads in the game engine.

3.8 Spawning

The wildlife system offers two spawning methods: static location spawn and dynamic location spawn to allow animals to have their desired positions in the game environment.

Static location spawning places each animal at a fixed desired position in the environment when the animals first spawn. The animals are then free to roam around in the environment if the creator allows. This method requires the creator to specify the initial positions in the configuration.

The second method is dynamic location spawn. As the name implies, the animals do not spawn at fixed positions. Their locations are determined by the viewing frustum where players can see. Needless to say, it is waste of effort to spawn an animal at a position where the player cannot see. Hence, when an animal is out of the viewing frustum of the player, it will spawn in front of the player again. To add additional flexibility to the wildlife system, the spawn frequency of the animals is controlled by a probability parameter.

The dynamic spawn approach is ideal for cinematic purposes. An example is a driver looking out of the windscreen while driving through a safari. This method creates the effect of endless stretch of animals along the roadside in the environment while in actual fact; there are only a few animals. Compared to the static spawn method, many more animals have to be placed in the environment to achieve the same effect. In a way, dynamic spawning optimises the use of hardware resources. Optimisation will be discussed in the next section.

Design of a Wildlife Animation System

The dynamic spawn approach is not perfect and brings about some problems. Animals will suddenly appear out of nowhere. In order to reduce this effect, animal spawning can be done at a further distance from the player. From far, the animals look smaller, making their sudden appearance less obvious. To improve the situation further, an additional approach used is to have an animal size blend. That is, the animals are smaller in size when they first spawn, slowly growing to their actual size. This method is most suitable for birds. The birds will appear to be flying towards the player from afar.

A third method is to implement alpha blend for the animations. Using this method, animals can spawn at any distance from the camera. This is similar to size blending, where the animal's animation is transparent when it first spawns. Then it slowly becomes opaque. This method gives the player a virtual world effect. All three solutions mentioned are provided in the system.

However, there is still a need to check for the animals' spawn locations before they are spawned. For example, a deer cannot spawn on a very steep slope and in the middle of the sea. These checks can be easily performed by functions provided by game engines to get the different ground types and their gradient.

For multiplayer or split screen games, the location to spawn the animals becomes tricky. An animal is considered out of the viewing frustum if and only if it is not in all of the cameras. Hence the position for spawning depends on the furthest camera. Figure 40 illustrates the situation.

Design of a Wildlife Animation System

Figure 40: Desired spawn location.

The red spot determines the location that the animals should spawn. It is chosen so that the maximum number of cameras can see the animals in their viewing frustum. The distance from the cameras is based on the furthest camera. This is a “safest priority” approach to ensure that the animals will not spawn too near to any camera.

Dynamic location spawning has another limitation. Once an animal is out of the viewing frustum, it is “pulled-out” of its location. Hence, if a player who passed by an animal decides to turn back and look at it again, it will be gone! One solution is to “pull-out” the animal only if the animal is out of the frustum for more than five seconds (or any amount of time depending on the game), and when the distance is far enough.

The figures below explain the procedure of dynamic spawning. In Figure 41, the player sees a group of animals.

Figure 41: Player sees animals within field of view.

When the player moves behind the animals as shown in Figure 42 for a period of time (about five seconds and about 100 feet away), the animals disappear and respawn in front of the player far away (about 400 feet) with size blending as shown in Figure 43.

Design of a Wildlife Animation System

Figure 42: Player behind the animals now.

Figure 43: Animals respawn in front of player.

Figure 44 below shows the complete process in a single diagram.

Figure 44: Full spawning process.

Table 4 summarizes the advantages of both spawning methods.

Table 4: Dynamic spawn VS static spawn

Dynamic	Static
<ul style="list-style-type: none"> • Hardware resource saving • Simulate many animals 	<ul style="list-style-type: none"> • No extra checks and spawn algorithms needed

Design of a Wildlife Animation System

<ul style="list-style-type: none"> Generated real-time 	
---	--

3.9 Optimisation

This wildlife system has to be very efficient. The memory cap is 5 megabytes (run time memory) and the CPU computation cap is 1.2 milliseconds. Functions are implemented to keep track of CPU time and memory. This is done to zoom in to the most costly functions or sub-systems of the wildlife system for optimisation. There are several areas of the system that can be optimised. Scheduling is done to prevent frame hits. Collision test with the ground, collision avoidances, direction calculation or even spawning test are not done for every frame. Instead, they are performed every few frames. In the wildlife system, test and calculations are staggered for each animal.

A window is first specified. It consists of frames for each animal to perform ground test, collision avoidance and direction calculations. Within each frame interval, computations are staggered. The diagram below shows how optimisation is done by spreading out the computation of 10 animals with a window size of 6.

Figure 45: Animal allocated frame.

In Figure 45, the first row is the frame number and alphabets are the animals. For example, C and I are in frame 3. This means that animal C and I only do the necessary computations in frame 3. A more complex way of scheduling AI is found in [31].

Design of a Wildlife Animation System

Mathematical calculations of directions and angles are optimised using an efficient math library. Cross products, dot products are calculated based on 2D only. Trigonometry such as sin, cos and tan are all pre-computed. Some mathematical functions are implemented in the assembly language to improve the performance. For Xbox, some mathematical calculations are vectorised to make use of the VMX in Xbox 360 vector processing hardware. Detail of VMX and Xbox architecture can be located in [32] by Andrews.

Flocking algorithm and steering behaviour are optimised as well. In addition, animation LOD reduces the computation cost of animation. To reduce computation further, only the three nearest animals will play 3D sound effects at any moment. Dynamic spawn is one of the optimisation techniques employed. The wildlife system is also equipped with frustum culling capability. This means that when an animal is out of the viewing frustum of the camera, it will not be rendered or processed at all. Scalable dynamics [33-36] can be used for further optimisation.

Memory is optimised by sharing geometry and animation data of the same animal models in the graphics hardware vertex buffer. All animals are limited to have 15 bones each. The number of polygons for each animal cannot exceed 1000. The resolution of the animal texture is 256 X 256 compressed into DXT5 texture file format. All sound effect files are compressed to less than 100 kilobytes.

3.10 Debugging and development tools

Development and debugging tools are extremely important for game development. To develop a huge piece of software like a game, there are many unforeseen situations and conditions. Bugs are not rare and they must be solved. This is where debugging tools come in handy. During the wildlife system development, debugging functions like debug print text, debug draw lines, draw circles, draw bounding boxes are built. These

Design of a Wildlife Animation System

debugging functions are used extensively to debug mathematics calculations. In the scripting system of the wildlife, there is an inbuilt debugging system to debug the behaviours and the transitions of an animal. Good debugging tools also help to save time and effort in debugging. A reload function is also implemented to reload the wildlife system during the game. This saves time to restart the game just to see the effect of any changes made to the wildlife settings.

To make the wildlife system more user-friendly and to speed up wildlife creation, a wildlife development Graphical User-Interface (GUI) tool known as “Wildlife Editor” is developed. This tool is created by C#. Artists and other programmers can make use of this tool to create or edit a wildlife system. This tool will generate setting files similar to those in Appendix 4 where the game engine can load it. This means that apart from typing the wildlife setting file, this tool can also generate the file. Figure 46 shows the screenshot of the development tool.

Design of a Wildlife Animation System

Figure 46: Wildlife creation tool.

As mentioned in the beginning of the chapter, there are 3 levels in this wildlife system; Wildlife Pool, Wildlife Group and Wildlife. Therefore in this tool, there are 3 “tabs”. Figure 46 shows the Wildlife Pool tab. As an example, the wildlife pool has 7 types of animals (displayed in the circle) as shown in the middle panel. The side panel shows the properties of the wildlife pool that the user can change and edit.

Figure 47 below shows the Wildlife tab of the same wildlife development tool.

Figure 47: Behaviour state and transition creation tool.

In this tab, users are allowed to design the wildlife behaviours and state transitions. The left panel (the panel with graphics) shows the topology properties of the animal. The big square nodes are the behaviours and the smaller rectangle nodes are the transition logics.

Design of a Wildlife Animation System

The edges are the links between the behaviours and transitions. The small arrows on the edges show the direction of the transitions. The right panel shows the properties of each node or edge when user selects it from the left panel.

IV

CHAPTER 4

4 Analysis of Algorithms

There are some important or costly algorithms in this wildlife system that should be addressed and optimised. This section provides analyses for those algorithms. The functions shown here are not the actual functions used in the wildlife system. Instead, they are modified to be more readable.

4.1 General mathematics

The trigonometry functions such as sin, cos and tan are pre-computed into a lookup table. The computation of steering behaviour is first done on 2D only (left/right and forward/backward). After that, the height of animal is gotten from the terrain height map. This technique is commonly known as 2.5D (two and a half Dimensions). It reduces calculations for mathematic functions such as cross-products and dot products.

4.1.1 Cross product

The code below shows a 3D cross product function. It has 6 multiplications and 3 subtractions.

```
inline Vector3 CrossProduct(const Vector3 &v0, const Vector3 &v1)
{
    Vector3 result;

    result.x = v0.y*v1.z - v0.z*v1.y;
    result.y = v0.z*v1.x - v0.x*v1.z;
    result.z = v0.x*v1.y - v0.y*v1.x;

    return result;
}
```

Analysis of Algorithms

There is no real cross product for 2D. However, the z dimension can be set to 0. That will be a 2D cross product actually. When z is set to 0, the 3D cross product becomes the following:

```
inline Vector3 CrossProduct(const Vector3 &v0, const Vector3 &v1)
{
    Vector3 result;

    result.x = 0 - 0;
    result.y = 0 - 0;
    result.z = v0.x*v1.y - v0.y*v1.x;

    return result;
}
```

This means that a 2D cross product is only “ $z = v0.x*v1.y - v0.y*v1.x$ ”. This function is useful for finding the clockwise or anti-clockwise rotation direction of an angle. The 2D cross product function only takes 2 multiplications and 1 subtraction. It reduces computation to 33% of the original 3D cross product. This is a huge optimisation for the wildlife system because many cross products calculations are needed.

4.1.2 Dot product

The code below shows a 3D dot product and a 2D dot product functions respectively.

```
inline float DotProduct(const Vector3 &v0, const Vector3 &v1)
{
    return (v0.x*v1.x + v0.y*v1.y + v0.z*v1.z);
}
```

```
inline float DotProduct(const Vector2& v0, const Vector2& v1)
{
    return (v0.x*v1.x + v0.y*v1.y);
}
```

The 3D dot product takes 3 multiplications and 2 additions whereas the 2D dot product takes 2 multiplications and 1 addition. Using the 2D dot product reduces the computation to about less than 67% of the original computation.

4.2 Animal behaviour state function

There are many state functions for the wildlife to choose from, such as Walk, Run, Fly, Idle, etc. These state functions are bound to the behaviour graph defined in the setting file.

For example, the “ExecuteScript = { Walk; }” in the following setting.

```
[Behavior2]

Name = "Walk"
AnimationType = "Walk"
AnimationFileName = "WLHare_Walking"
AnimationTimeScale = 1.000000
EnterScript = ""
ExecuteScript = "{ Walk; }"
ExitScript = ""
```

All of these state functions are organized in a neat and maintainable way. The code below shows a Walk function.

```
void Wildlife::Walk()
{
    // if this is Fly function, it will simulate flocking effect.
    // SimulateFlock();

    // Find the walking direction.
    ComputeDirection();

    // Find the walking speed.
    ComputeSpeed(mSpeed, mMaxWalkSpeed);

    // Update to new position.
    mPosition += mLookVector*mSpeed;

    // Put the animal on the ground nicely.
    PerformGroundTest();

    // Update the animation.
    UpdateAnimation();
}
```

For the best quality, the behaviour function is executed once per animal every frame. However, due to scheduling optimisation, it may execute once every few frames. The upper bound is $O(n)$ where n = number of animals in the wildlife system.

Analysis of Algorithms

In the ComputeDirection function, it is again broken down into several functions such as AvoidForbiddenZone, AvoidSteepSlope, AvoidVehicle, AvoidObjects, and AvoidWildlife. Each will be explained in the following parts. The code below shows the ComputeDirection function.

```
bool Wildlife::ComputeDirection()
{
    // Test all avoidance in priority ordered.
    if(
        !AvoidForbiddenZone()
        && !AvoidSteepSlope()
        && !AvoidVehicle()
        && !AvoidObject()
        && !AvoidWildlife()
    )
    {
        // No avoidance at all.
        return false;
    }

    // Hit any one avoidnace.
    return true;
}
```

The functions are placed in a “if statement” to illustrate the priority of avoidance. The highest priority goes to avoiding forbidden zones such as water. The next priority goes to steep slope and so on. At any point in time when an avoidance function is fired, it will stop checking for lower priority avoidance. This is also an optimisation, so that each animal only avoids one type of avoidance at one time. The upper bound of executing this function is $O(n)$ where n = number of animals in the wildlife system.

4.3 Avoid forbidden zone

This has the highest avoidance priority because it will look funny for land animals to run into the sea or river. Hence, this should be avoided at all costs. The following code shows the AvoidForbiddenZone function.

Analysis of Algorithms

```
bool Wildlife::AvoidForbiddenZone()
{
    // Predict 30 feet in front.
    mPredictedPosition = mPosition + mLookVector*30.0f;

    // Check that predicted position if it is in a forbidden zone.
    if(InForbiddenZoneAvoidance(mPredictedPosition))
    {
        // If it is, turn around.
        mTurnAround = true;
        return true;
    }

    // If it is not, return false.
    return false;
}
```

As this function has the highest priority, it will be tested and executed as long as the animal is scheduled to execute. Hence, the upper bound for this algorithm is $O(n)$, where n = number of animals.

4.4 Avoid steep slope

This is the second priority avoidance algorithm. If the animal is not predicted to hit the forbidden zone, it will execute this function. Below shows the code for avoid steep slope function. This slope test is done by testing the normal of the slope at a predicted point. This algorithm will execute a maximum time of $O(n)$ if the zone test always fail.

Analysis of Algorithms

```
bool Wildlife::AvoidSteepSlope()
{
    // Variable to hold the normal of a terrain point.
    Vector3 upVector;

    // Predict a distance ahead.
    // Get the normal of that predicted position in upVector.
    GetTerrainIntersection(mPredictedPosition, upVector);

    // Test the normal of that point.
    // If that predicted point is > mMaxInclination, turn around.
    if(upVector.z < mMaxInclination)
    {
        // Turn around and move in the downslope direction.
        mTurnAround = true;

        // Make into 2D and normalise it.
        upVector.z = 0.0f;
        Normalize(&upVector);

        // Assign the downslope direction.
        mFinalDirection = Normalize(upVector);
        return true;
    }

    return false;
}
```

4.5 Avoid vehicle

This is the most important algorithm in the wildlife system as the core feature of this system is to avoid oncoming vehicles. The upper bound for executing this function is $O(n)$ times where n = number of animals of the wildlife system.

Analysis of Algorithms

```

bool Wildlife::AvoidVehicle()
{
    // Search for nearest vehicle.
    if(SearchForVehicle(mRadius))
    {
        // If vehicle is in critical range.
        if(mVehicleInCriticalRange)
        {
            // If vehicle is in critical range.
            // Always run at the mCriticalRunAngle
            mFinalDirection = FixHeadingAngle(mCriticalRunAngle);

            // Slowly rotate the animal with its TurnRate.
            // So that the animal don't suddenly pop.
            mLookVector = RepairHeading(mCriticalRunTurnRate);
        }
        else
        {
            // If vehicle is not in critical range.
            // Run at a minimum angle of mRunMinimumAngle
            mFinalDirection = FixHeadingToMinAngle(mRunMinimumAngle);

            // Slowly rotate the animal with its TurnRate.
            // So that the animal don't suddenly pop.
            mLookVector = RepairHeading(mRunTurnRate);
        }

        return true;
    }

    return false;
}

```

4.6 Avoid object

The SearchForObject function searches for the nearest object in a radius. If there is an object with the radius of the animal, the object avoidance algorithm will be fired. The worse case complexity of this algorithm is $O(n)$ where n = number of animals in the wildlife system.

Analysis of Algorithms

```

bool Wildlife::AvoidObject()
{
    // Search for nearest object in a radius.
    if(SearchForObject(mRadius))
    {
        // Get right vector of the animal in 2D.
        Vector3 right(mLookVector.y, - mLookVector.x, 0.0f);

        // Compute the avoidance vector.
        Vector3 avoidObjectVector;
        avoidObjectVector = mPosition - mpNearestObject->Position;

        // Make the avoidance vector 2D.
        avoidObjectVector.z = 0.0f;

        // Compute the magnitude of the avoidance vector.
        float magnitude = FastMagnitude(avoidObjectVector);

        // Find the positive dot product.
        float absDP = abs(DotProduct2D(avoidObjectVector, right));

        // This is a distance test and a body width test.
        // The concept is explained in chapter 3.3.
        if(    magnitude < mOffset + 30.0f
            && absDP < mBoundingBox + mOffset)
        {
            // A factor use to compute the final direction.
            float factor = 3.0f/(magnitude - mOffset);

            // Make factor always positive.
            GetPositive(factor);

            // Clamp it to between 0.5 and 3.
            Clamp(factor, 0.5f, 3.0f);

            // Normalise the look vector
            Normalize(&mLookVector);

            // The maths to calculate the direction vector.
            mFinalDirection = mLookVector + avoidObjectVector*factor;

            // Normalise the direction.
            Normalize(&mFinalDirection);
            return true;
        }
    }

    return false;
}

```

4.7 Avoid wildlife

This wildlife avoidance algorithm is exactly the same as object avoidance algorithm.

Except that instead of searching for objects, it searches for animals. The code below

Analysis of Algorithms

shows the algorithm for avoiding wildlife. The worse case complexity is also $O(n)$, where n = number of animals in the wildlife system.

```
bool Wildlife::AvoidWildlife()
{
    // Search for nearest wildlife in a radius.
    if(SearchForWildlife(mRadius))
    {
        // Get right vector of the animal in 2D.
        Vector3 right(mLookVector.y, - mLookVector.x, 0.0f);

        // Compute the avoidance vector.
        Vector3 avoidObjectVector;
        avoidObjectVector = mPosition - mpNearestWildlife->Position;

        // Make the avoidance vector 2D.
        avoidObjectVector.z = 0.0f;

        // Compute the magnitude of the avoidance vector.
        float magnitude = FastMagnitude(avoidObjectVector);

        // Find the positive dot product.
        float absDP = abs(DotProduct2D(avoidObjectVector, right));

        // This is a distance test and a body width test.
        // The concept is explained in chapter 3.3.
        if(    magnitude < mOffset + 30.0f
            && absDP < mBoundingBox + mOffset)
        {
            // A factor use to compute the final direction.
            float factor = 3.0f/(magnitude - mOffset);

            // Make factor always positive.
            GetPositive(factor);

            // Clamp it to between 0.5 and 3.
            Clamp(factor, 0.5f, 3.0f);

            // Normalise the look vector
            Normalize(&mLookVector);

            // The maths to calculate the direction vector.
            mFinalDirection = mLookVector + avoidObjectVector*factor;

            // Normalise the direction.
            Normalize(&mFinalDirection);
            return true;
        }
    }

    return false;
}
```

Analysis of Algorithms

What is interesting in this algorithm is the function SearchForWildlife. This function searches through all animals for the corresponding nearest animal. Hence, the complexity of this algorithm is $O(n^2)$. The algorithm is shown below.

```
void SearchForWildlife(float radius)
{
    // Declare some variables.
    float d;
    float nearestDistance = FLOAT_MAX;
    unsigned int i;
    Wildlife *pOther;

    // Loop through all animals.
    for(i = 0; i < mWildlifeList.NumberOfItems; i++)
    {
        pOther = mWildlifeList[i];

        // Skip itself.
        if(pOther != pCurrent)
        {
            // Compute the distance between animals.
            d = FastMagnitude(pCurrent->Position - pOther->Position);

            // Radius test and nearest animal test.
            if(d < radius && d < nearestDistance)
            {
                // Update nearest animal.
                nearestDistance = d;
                mpNearestWildlife = pOther;
            }
        }
    }
}
```

4.8 Flocking

The usual flocking algorithm requires three components: cohesion, alignment and separation. Cohesion controls the closeness of a bird to the centre of the flock, alignment controls the direction of the flock and separation controls the distance between each bird. Cohesion, alignment and separation are $O(n^2)$ complexities. In total, the flocking algorithm is $O(3n^2)$.

Analysis of Algorithms

The original way to calculate the centre of mass (COM) component in cohesion is to add up all the positions of the birds excluding itself and divide by $n-1$ (total – itself). However, in this wildlife system, the COM is calculated as all the position of the birds divide by total number. All the birds will use this COM to compute its flight path. Therefore, the COM component in cohesion is computed once at $O(1)$ complexity with a small accuracy trade off. In addition, alignment is totally replaced with a random direction. This removed one of the $O(n^2)$ components. The complexity of the algorithm after optimisation becomes $O(n^2 + 1)$. This optimises to about 33% of the original flocking algorithm.

The code below shows the SimulateFlock algorithm that is called by the birds when flocking is activated. The complexity of this function is $O(n)$ where n = number of birds. Take note that in the first part, the complexity is about the flocking algorithm, whereas in the later part, the complexity is about the function calls.

```
void Wildlife::SimulateFlock()
{
    // Variables to store flocking properties.
    // By experimentation, 1.2*separation and 1.4*cohesion gives nice
    // flocking effect.
    Vector3 sForce      = 1.2f*ComputeSeparation();
    Vector3 aForce      = mpWildlifeGroup->mFinalDirection;
    Vector3 cForce      = 1.4f*ComputeCohesion();

    mFinalDirection = sForce + aForce + cForce;
}
```

The code below shows the ComputeCOM algorithm that is called by the birds group once every execution. The complexity of this function is $O(n)$ where n = number of groups.

Analysis of Algorithms

```
void WildlifeGroup::ComputeCOM()
{
    // Initialise the mCOM variable.
    mCOM = Vector3(0.0f, 0.0f, 0.0f);

    // Loop through all the birds.
    for(unsigned int i = 0; i < mWildlifeList.NumberOfItems; i++)
    {
        // Add all their positions.
        mCOM += mWildlifeList[i]->Position;
    }

    if(mWildlifeList.NumberOfItems > 0)
    {
        // Divide by number of birds to get the COM.
        mCOM /= (float)mWildlifeList.NumberOfItems;
    }
}
```

For each bird to get the cohesion force, it needs to minus their position and normalise it.

The complexity of this function is $O(n)$ where n = number of birds.

```
Vector3 Wildlife::ComputeCohesion()
{
    return Normalize(mpGroup->mCOM - mPosition);
}
```

The code below shows the ComputeSeparation function. Basically, each bird has to add up all the forces acting against it in opposite direction. Therefore, this algorithm is in $O(n^2)$ complexity. In the wildlife system, this separation computation is the most expensive component of the flocking feature. However, it is still extremely fast for less than 20 birds, and the game will only have about 15 birds each time.

```

void Wildlife::ComputeSeparation(Vector3 &separation)
{
    // Create some variables.
    Vector3 toAgent(0.0f, 0.0f, 0.0f);
    List<Wildlife *> *pList = mpGroup->GetWildlifeList();
    float length;

    // Initialise the separation force to 0.
    separation = Vector3(0.0f, 0.0f, 0.0f);

    // Loop through the list.
    for (unsigned int i = 0; i < pList->NumberOfItems; i++)
    {
        // Exclude itself.
        if(pList[i] != this)
        {
            // Compute the vector between the other bird and itself.
            toAgent = mPosition - pList[i]->Position;

            // Compute the distance between the 2 birds.
            length = FastMagnitude(toAgent);

            // Error check.
            if(length != 0.0f)
            {
                // Normalise the direction.
                Normalize(&toAgent);

                // Add the separation force together.
                // Nearer birds have greater separation force.
                separation += toAgent/length;
            }
        }
    }

    // Normalise the separation force.
    Normalize(&separation);
}

```

4.9 Animation LOD

The code below shows the animation Level-Of-Detail (LOD) algorithm. If the animal is within a certain distance, no LOD is activated. This is to ensure that animals that are nearest to the camera are animated at their highest quality. If the animals are not within the certain distance, then LOD is tested based on LODDistance (distance where LOD starts), mRadius (the radius of the animal) and the LODFactor (factor used to control the quality of the LOD). LODFactor is inversely proportional to the quality of the animation

Analysis of Algorithms

LOD. This variable cannot be 0. This LOD system tries to reduce the TickAnimation from executing frequently so as to reduce the computation time. UpdateAnimation is $O(n)$ where n = number of animals. The worse case scenario happens when all animals are within the no LOD range. This is where TickAnimation is also $O(n)$ complexity.

```
void Wildlife::UpdateAnimation()
{
    // If distance within no LOD distance.
    if(mDistanceFromCamera < mLODDistance)
    {
        // Tick the animation every frame using frame time.
        // Frame time is the time between the last 2 frames.
        // It is also known as delta seconds.
        TickAnimation(mAnimTickDeltaSeconds);
    }
    else
    {
        // Check if the animation should tick.
        if(mAnimTick > (int)(mLODDistance/(mRadius*mLODFactor)))
        {
            // Tick the animation every frame using frame time.
            TickAnimation(mAnimTickDeltaSeconds);
        }
        else
        {
            // Accumulate the ticks and the delta seconds.
            mAnimTickDeltaSeconds += mTickDeltaSeconds;
            mAnimTick++;
        }
    }
}
```

4.10 Spawning system

The code below shows the spawning algorithm. Spawning is done based on each animal, which means that the algorithm runs in $O(n)$ complexity where n = number of animals. The spawning test is based on priority. The cheapest test is conducted first so as to prevent expensive functions from executing as much as possible. This strategy is used widely in games.

Analysis of Algorithms

```
void WildlifePool::DynamicSpawn()
{
    for(unsigned int i = 0; i < mWildlifeList.NumberOfItems; i++)
    {
        Wildlife *pWildlife = mWildlifeList[i];
        pGroup = pWildlife->GetGroup();

        // Probability to spawn.
        if(    SpawnProbability(pWildlife)

            // Out of screen > 3 seconds and behind player?
            && ValidOutOfScreenTime(pWildlife)

            // Inside frustum?
            && InFrustum(pWildlife->Position)

            // Within camera clipping distance?
            && WithinClipDistance(pWildlife->Position)

            // Not spawn on slope and not spawn in forbidden area?
            && ValidSpawnPosition(pWildlife))
        {
            // All test pass. Animal respawn in front of driver.
            Respawn(pWildlife);
        }
    }
}
```

CHAPTER 5

5 Empirical Study: Scenarios

This section describes the wildlife system in the game play. While the players drive, they encounter different animals and gain game play experience through the wildlife system. The subsections below describe each stage or feature of the game and the wildlife system. In addition, this empirical study is also a test of the wildlife system to make sure that it fulfils all the requirements and objectives.

5.1 The beginning

Figure 48 shows a screenshot of the game at the starting line of the race. Players are given some time to get ready before the race actually starts. The two convertible vehicles in front are AI vehicles and the focused vehicle is the player vehicle. In the same screenshot, some birds are flying far away in the sky. At such a far distance, animation LOD is activated. Since the birds are only a few pixels, their animations are not computed every frame. This saves CPU resources for other functions of the game.

Empirical Study: Scenarios

Figure 48: The beginning of the race.

As the vehicle drives through the race, animals spawn in front of the vehicle as shown in Figure 49. The spawning system makes sure that the animals are spawned in valid places, and that players will not notice the animals disappearing behind them. To ensure this, the vehicle must pass the animal for at least five seconds and at least a hundred feet away from them before they are allowed to respawn.

The animals are spawned with size blending as well. This means that the newly spawned animals will grow in size to the desired size. This is done so that players will not perceive animals suddenly popping out of the environment. With size blending, the spawning is done seamlessly without players noticing as it is natural that when people drive near an object, the object becomes bigger due to changing viewing perspectives. The change in size is not noticeable as long as the distance is reasonably far as shown in the screenshot.

Empirical Study: Scenarios

In addition, this creates a very nice effect for birds. It will give a visual impression that the birds are flying nearer towards the vehicle, as the vehicle is driving nearer to the birds.

Figure 49: Along the race.

5.2 Birds

Figure 50 shows a group of birds flocking together in the sky. These birds are spawned as a single group, flying in the same direction to show that they are flocking. They fly at a reasonable distance and height such that the player can see. Occasionally, they make 3D sound effects like birds chipping. This makes the environment more realistic and lively.

Empirical Study: Scenarios

Figure 50: A group of birds flock together.

However, not all birds flock together. For example, hawks do not flock together. Figure 51 shows a screenshot of a hawk in the sky. The two smaller birds behind the hawk are not hawks. They belong to the same kind of bird in Figure 50.

Empirical Study: Scenarios

Figure 51: A hawk flapping wings in the sky.

Birds like hawks have different behaviour states. They flap their wings and they may hold their wings to glide. Figure 52 shows two hawks gliding across the screen. The hawks make screeching sounds in between some reasonable time intervals as they glide. Again, this adds some realism to the environment. The hawks are very good scene decoration elements because they are big birds. Players can see their animations, behaviours, the transition between gliding and the flapping of their wings, and hear the sounds they make.

Empirical Study: Scenarios

Figure 52: Hawks gliding in the sky.

Figure 53 shows a hawk banks as it turns along its flight path. The banking and turning is controlled by the auto pilot system. The auto pilot system allows the bird to have a nice, logical and natural turn. In the same screenshot, the sun at the top left corner cast a shadow of the hawk on the bottom right corner as shown in Figure 53. This shadow blends in to the environment nicely, allowing the hawk to be perceived as part of the environment.

Empirical Study: Scenarios

Figure 53: A hawk banks as it turns.

5.3 Ground animals

Ground animals have almost the same features as compared to the birds. They have shadows, different behaviours, 3D sounds, spawning, etc. On top of all the features, ground animals are more interactive. This is because racing vehicles cannot fly in the air to interact with the birds. However, there is an example in the next section that shows that birds are equally interactive in this wildlife system.

Figure 54 shows some animals in front of the vehicle. Some animals are walking around in the environment while others are in their idle state looking around. As the vehicle goes nearer to the animals, they change their state to running state and start to run away from the vehicle. There is no animation popping effect when the animals change their states because of the seamless animation blending system that controls the animations. The

Empirical Study: Scenarios

speeds of the animals are governed by accelerations and thus these animals will not suddenly change their speeds. In addition, some animals like cats meow when they transit to the run state. The interactivity of the animals comes from the change of states, the sounds they made, the animations as well as the intelligent ways they avoid the vehicle when the vehicle gets near them.

Figure 54: Animals spawn in front of the animal.

Figure 55 shows the situation of the animals run at their critical speeds and almost perpendicular to the velocity of the vehicle when the vehicle gets very near to the animals. This is to ensure that the animals escape unharmed. The colours, shadows and lightings of the animals blend correctly into the environment. Figure 56 shows another scenario where an animal escapes from the vehicle into the forest.

Empirical Study: Scenarios

Figure 55: Animals running away from vehicle.

Figure 56: Another animal running away from vehicle.

Empirical Study: Scenarios

Figure 57 and Figure 58 are sequential screenshots. The two figures show more interactivity and intelligence of the animal. When the player starts to chase the running animal, it runs deeper into the forest, and faster.

Figure 57: Animal chased into the forest.

Figure 58 is the closed up screenshot of the vehicle chasing the animal. Players can choose to chase animals and have some fun trying to run them over. Especially in an off-road racing game where players can roam around the environment freely, chasing animals becomes a completely new game play experience. However, the animals will always escape safely. It is one of the requirements of the game to be non-violent. Therefore, animals should avoid vehicles as much as possible, except in special situations when forbidden area avoidance and steep slope avoidance take precedence.

Empirical Study: Scenarios

Figure 58: Closed up animal chased into the forest.

5.4 Special effects

There are two special effects in the wildlife system to make the game more interesting. Figure 59 to Figure 62 show the first special effect. Before the race starts, the camera rotates around the vehicle for a few seconds to let players get ready. During this time, birds fly across the screen chipping away. This creates a nice wild jungle effect.

Empirical Study: Scenarios

Figure 59: Visual effect I.

Figure 60: Visual effect II.

Empirical Study: Scenarios

Figure 61: Visual effect III.

Figure 62: Visual effect IV.

Empirical Study: Scenarios

Figure 63 to Figure 67 show the second special effect. A group of quails spawn in front of the vehicle. They are in idle state, feeding and looking around while making 3D sound effects to catch the player's attention. When the vehicle is near, the quails will scatter and fly off in different directions. The figure below shows a group of idle quails.

Figure 63: Quails escape effect I.

As the vehicle gets near, the quails begin to flee. They make more chipping sounds and change into flying states. This situation is shown in the figure below. As the birds fly higher, their shadows become bigger. This lets the player know that the quails are flying higher.

Empirical Study: Scenarios

Figure 64: Quails escape effect II.

Figure 65: Quails escape effect III.

Empirical Study: Scenarios

Figure 66: Quails escape effect IV.

Figure 67: Quails escape effect V.

Empirical Study: Scenarios

Figure 68 below shows another quail fleeing effect.

Figure 68: Another quails escape scenario.

5.5 Comparison with other games

Vehicle can get nearer to animals in this game as compared to Rally Trophy (Previously reviewed in Section 2.5). This is because the animals in this wildlife system are interactive. They can escape without getting hurt. It allows players to enjoy chasing animals around. Banking and auto pilot feature of birds in this wildlife system make the birds look more realistic as compared other games mentioned in Section 2.5 and 2.6. The two special effects mentioned in the previous section are not found in any game mentioned in the game reviews as well. Another cutting edge feature that this wildlife system provides is the ability for the animals to make 3D sounds to attract the player's attention. The wildlife system is optimised with many different techniques. Therefore, it

Empirical Study: Scenarios

has enough resources to keep the animals' model and animation quality high. And thus, the quality of model and animation are better than other reviewed games.

CHAPTER 6

6 Summary and conclusion

This chapter summarises the results of the implementation of the wildlife system, which was successfully created, meeting all the specifications and requirements. The final version of the wildlife system will be published with the game.

6.1 Game play

This is a new kind of game play for racing games that has not been implemented in the past. Game testers from the publisher gave good feedbacks about the wildlife system. They were impressed with its interactivity, and were driving around chasing animals; looking at their behaviours and listening to the 3D sound effects the wildlife made.

6.2 Performance

This wildlife system has several optimisation mechanisms. The wildlife spawning method reduces number of animals. Animation LOD reduces the animation calculation time. 2.5D is used instead of 3D calculations to optimise the computation time for finding positions, angles, and directions of the animals. Scheduling reduces the frequency of decision making and computation of positions, angles and directions of the animals. Special hardware VMX is used for the maths library for optimisation. Animation skinning is done with GPU instead of CPU. Textures are compressed to DXT5 format.

One of the cutting edges of this wildlife system is that it can produce reasonably high quality graphics. The reason behind it is because the software optimisations are explored deeply and implemented. With all the above mentioned optimisations, number of

Summary and conclusion

polygons can remain reasonably large, texture quality and animation quality can remain high. The approach here is to optimise the software extensively so that the quality of the model and animation still remains high.

The final wildlife system operates using 0.98 milliseconds of the CPU time and 4.5 megabytes of runtime memory on an Xbox 360 machine with 25 animals (5 types of animals). The CPU time excludes sorting, rendering and audio time because they are done using another thread which is not the bottle neck of the game. In general, the wildlife system is linearly scalable in $O(n)$. The only two features that have $O(n^2)$ complexity are wildlife avoidance and flocking. The wildlife system is also tested and works with 500 animals with a frame rate of 20 fps.

6.3 Wildlife design

The wildlife system is properly designed to facilitate development and maintenance. It is portable, modular and multi-thread supported. It is already ported and tested on another game demo. The design is suitable for games that run on PC, Xbox and PS3. Figure 69 below shows the full wildlife system mind map.

Summary and conclusion

Figure 69: Mind map of wildlife system.

VII

CHAPTER 7

7 Future work: Wildlife system

One possible direction for this wildlife system is to be implemented into new next generation games that run on next generation consoles and hardware. Better animal animations with more polygons and more bones can be put into the game. More complex behaviours can be designed. Animals can also have personalities and characteristics, as well as social status. Some games already have such features. Those games are usually character-based games with a few characters. The reason is because it is expensive to have these artificial intelligence features.

Another direction to go is to implement an inverse kinematics and forward kinematics (IK/FK) system. With the IK/KF system, the wildlife system can be made into a character system where the characters support dynamic animations and behaviours. It can then be used in character-based games such as role-playing-games (RPG). The objective is to make the wildlife system into an agent system suitable for both complex and simple agent simulation. In this way, it can be used in every genre of games.

More tools can also be built for the wildlife system. The current wildlife editor can be enhanced into a better what-you-see-is-what-you-get (WYSIWYG) tool. A wildlife viewer can be incorporated into the wildlife editor so that designers or artists can view the animals' behaviours immediately from the tool itself.

APPENDIX - 1

Appendix 1: DirectX SDK animation shader code.

```
//
// Animated Skinned Mesh Effect file
// Copyright (c) Microsoft Corporation. All rights reserved.
//

float4 lhtDir = {0.0f, 0.0f, -1.0f, 1.0f};    //light Direction
float4 lightDiffuse = {0.6f, 0.6f, 0.6f, 1.0f}; // Light Diffuse
float4 MaterialAmbient : MATERIALAMBIENT = {0.1f, 0.1f, 0.1f, 1.0f};
float4 MaterialDiffuse : MATERIALDIFFUSE = {0.8f, 0.8f, 0.8f, 1.0f};

// Matrix Palette
static const int MAX_MATRICES = 26;
float4x3 mWorldMatrixArray[MAX_MATRICES] : WORLDMATRIXARRAY;
float4x4 mViewProj : VIEWPROJECTION;

////////////////////////////////////////
struct VS_INPUT
{
    float4 Pos          : POSITION;
    float4 BlendWeights : BLENDWEIGHT;
    float4 BlendIndices : BLENDINDICES;
    float3 Normal       : NORMAL;
    float3 Tex0         : TEXCOORD0;
};

struct VS_OUTPUT
{
    float4 Pos      : POSITION;
    float4 Diffuse  : COLOR;
    float2 Tex0     : TEXCOORD0;
};

float3 Diffuse(float3 Normal)
{
    float CosTheta;

    // N.L Clamped
    CosTheta = max(0.0f, dot(Normal, lhtDir.xyz));

    // propogate scalar result to vector
    return (CosTheta);
}

VS_OUTPUT VShade(VS_INPUT i, uniform int NumBones)
{
    VS_OUTPUT o;
    float3 Pos = 0.0f;
    float3 Normal = 0.0f;
```

Future work: Wildlife system

```

float      LastWeight = 0.0f;

// Compensate for lack of UBYTE4 on Geforce3
int4 IndexVector = D3DCOLORtoUBYTE4(i.BlendIndices);

// cast the vectors to arrays for use in the for loop below
float BlendWeightsArray[4] = (float[4])i.BlendWeights;
int   IndexArray[4]        = (int[4])IndexVector;

// calculate the pos/normal using the "normal" weights
// and accumulate the weights to calculate the last weight
for (int iBone = 0; iBone < NumBones-1; iBone++)
{
    LastWeight = LastWeight + BlendWeightsArray[iBone];

    Pos += mul(i.Pos, mWorldMatrixArray[IndexArray[iBone]]) *
BlendWeightsArray[iBone];
    Normal += mul(i.Normal, mWorldMatrixArray[IndexArray[iBone]]) *
BlendWeightsArray[iBone];
}
LastWeight = 1.0f - LastWeight;

// Now that we have the calculated weight, add in the final
influence
Pos += (mul(i.Pos, mWorldMatrixArray[IndexArray[NumBones-1]]) *
LastWeight);
Normal += (mul(i.Normal, mWorldMatrixArray[IndexArray[NumBones-1]])
* LastWeight);

// transform position from world space into view and then projection
space
o.Pos = mul(float4(Pos.xyz, 1.0f), mViewProj);

// normalize normals
Normal = normalize(Normal);

// Shade (Ambient + etc.)
o.Diffuse.xyz = MaterialAmbient.xyz + Diffuse(Normal) *
MaterialDiffuse.xyz;
o.Diffuse.w = 1.0f;

// copy the input texture coordinate through
o.Tex0 = i.Tex0.xy;

return o;
}

int CurNumBones = 2;
VertexShader vsArray[4] = { compile vs_1_1 VShade(1),
                           compile vs_1_1 VShade(2),
                           compile vs_1_1 VShade(3),
                           compile vs_1_1 VShade(4)
                           };

////////////////////////////////////
// Techniques specs follow
////////////////////////////////////
technique t0
{
    pass p0

```

Future work: Wildlife system

```
{  
    VertexShader = (vsArray[CurNumBones]);  
}  
}
```

A2 *APPENDIX - 2*

Appendix 2: Example of the logic of the state machine.

- If in walk state:
 - If hungry, go to feed state
 - 10% chance go to run state
 - 90% chance remain in walk state
- If in run state:
 - 10% chance go to walk state
 - 90% chance remain in run state
- If in feed state:
 - If not hungry
 - go to walk state
 - If hungry, remain in feed state

APPENDIX - 3

Appendix 3: New state logic.

- If in walk state:
 - Render walk state
 - If less than minimum time
 - Remain in walk state
 - If more than maximum time
 - Change to run state
 - If more than minimum time
 - If hungry, go to feed state
 - 10% chance go to run state
 - 90% chance remain in walk state
- If in run state:
 - Render run state
 - If less than minimum time

Future work: Wildlife system

- Remain in run state
- If more than maximum time
 - Change to walk state
- If more than minimum time
 - 10% chance go to walk state
 - 90% chance remain in run state
- If in feed state:
 - Render feed state
 - If less than minimum time
 - Remain in feed state
 - If more than maximum time
 - Change to walk state
 - If more than minimum time
 - If not hungry
 - Go to walk state
 - If hungry, remain in feed state

APPENDIX - 4

Appendix 4: Example of scripting behaviour and transition for a hare.

```
; -----
; Start -> Wildlife Behaviour

[Behavior1]

Name = "Idle"
AnimationType = "Idle"
AnimationFileName = "WlHare_Idle"
AnimationTimeScale = 1.000000
EnterScript = ""
ExecuteScript = "{ Idle; }"
ExitScript = ""
InitialBehavior = true

[Behavior2]

Name = "Walk"
AnimationType = "Walk"
AnimationFileName = "WlHare_Walking"
AnimationTimeScale = 1.000000
EnterScript = ""
ExecuteScript = "{ Walk; }"
ExitScript = ""

[Behavior3]

Name = "Run"
AnimationType = "Run"
AnimationFileName = "WlHare_Running"
AnimationTimeScale = 1.500000
EnterScript = ""
ExecuteScript = "{ Run; }"
ExitScript = ""

[Behavior4]

Name = "RunCoolDown"
AnimationType = "Run"
AnimationFileName = "WlHare_Run"
AnimationTimeScale = 1.000000
EnterScript = ""
ExecuteScript = "{ Run; }"
ExitScript = ""
```

Future work: Wildlife system

```

; End -> Wildlife Behaviour
; -----

; -----
; Start -> Wildlife Transition

[Transition1]

Name = "IdleToWalk"
From = "Idle"
To = "Walk"
AnimationBlendDuration = 0.500000
Probability = 0.005000
TransitionLogic = "WildlifeTimeInState > 3.0 && Probability
{ BlendAnimation; PlayTransitionSound; }"

[Transition2]

Name = "IdleToRun"
From = "Idle"
To = "Run"
AnimationBlendDuration = 0.500000
Probability = 0.000000
TransitionLogic = "VehicleInAlertRange { BlendAnimation;
PlayTransitionSound; }"

[Transition3]

Name = "WalkToRun"
From = "Walk"
To = "Run"
AnimationBlendDuration = 0.500000
Probability = 0.000000
TransitionLogic = "VehicleInAlertRange { BlendAnimation;
PlayTransitionSound; }"

[Transition4]

Name = "WalkToIdle"
From = "Walk"
To = "Idle"
AnimationBlendDuration = 0.500000
Probability = 0.005000
TransitionLogic = "WildlifeTimeInState > 3.0 && Probability
{ BlendAnimation; PlayTransitionSound; }"

[Transition5]

Name = "WalkToWalk"
From = "Walk"
To = "Walk"
AnimationBlendDuration = 0.000000
Probability = 0.005000
TransitionLogic = "!Swift && WildlifeTimeInState > 3.0 && Probability
{ WildlifeFindRandomDirection; BlendAnimation; PlayTransitionSound; }"

```

Future work: Wildlife system

```
[Transition6]

Name = "RunCoolDownToRun"
From = "RunCoolDown"
To = "Run"
AnimationBlendDuration = 0.000000
Probability = 0.000000
TransitionLogic = "VehicleInAlertRange { BlendAnimation;
PlayTransitionSound; }"

[Transition7]

Name = "RunCoolDownToWalk"
From = "RunCoolDown"
To = "Walk"
AnimationBlendDuration = 0.500000
Probability = 0.000000
TransitionLogic = "WildlifeTimeInState > 4.0 { BlendAnimation;
PlayTransitionSound; }"

[Transition8]

Name = "RunToRunCoolDown"
From = "Run"
To = "RunCoolDown"
AnimationBlendDuration = 0.500000
Probability = 0.000000
TransitionLogic = "!VehicleInAlertRange { BlendAnimation;
PlayTransitionSound; }"

; End -> Wildlife Transition
; -----
```

A5 *APPENDIX - 5*

Appendix 5: Example of sound effect setting for quail.

```
[Header]
NumSounds = 2

[Sound1]
Filename = "QuailTakeOffTransition.mp3"
MinDistance = 1.0
MaxDistance = 500
Is3D = true
LoopCount = 1
Volume = 0.1
IsStreaming = false
Type = Transition
FromState = Idle
ToState = FlyUp

[Sound2]
Filename = "QuailIdleRandom.mp3"
MinDistance = 1.0
MaxDistance = 500
Is3D = true
LoopCount = 1
Volume = 0.1
IsStreaming = false
Probability = 0.2
Type = Random
State = Idle

[Sound3]
Filename = "QuailFlyState.mp3"
MinDistance = 1.0
MaxDistance = 500
Is3D = true
LoopCount = 1
Volume = 0.1
IsStreaming = false
Type = State
State = Fly
```

A6 *APPENDIX - 6*

Appendix 6: Banking formula from aeromechanics for birds.

$$\omega V = g \tan \Phi$$

ω – Angular rate

V – Velocity

g – Gravity

Φ – Bank angle

To find the banking angle, make Φ the subject of the equation: $\Phi = \tan^{-1}(\omega V/g)$

g is a constant = 9.81 m/s^2 .

ω is the rotation angle (yaw angle or rotation about the up bird's up direction) per second of a bird in radian per second.

V is the velocity of the bird in m/s.

The resulting Φ is the roll angle or rotation about the bird's look direction.

APPENDIX - 7

Appendix 7: Publications.

- [1] J. C. Tay, C. H. Tng, and C. S. Chan, "Environmental effects on the coevolution of pursuit and evasion strategies," *Genetic Programming and Evolvable Machines*, vol. 9, pp. 5-37, 2008.
- [2] Tng C. H. John, Edmond C. Prakash, and N. S. Chaudhari., "Strategic Team AI Path Plans: Probabilistic Pathfinding," *International Journal of Computer Game Development*, vol. 2008, Article ID 834616, 6 pages, 2008.
doi:10.1155/2008/834616, 2008.
- [3] Tng C. H. John, "Wildlife Animation System," *GameScape December 2007 Seminal, NTU, GameLab*.

REFERENCE

-
- [1] S. Franklin, "Autonomous Agents as Embodied AI," *Cybernetics and Systems*, vol. 28, pp. 499-520, 1997.
 - [2] S. Franklin and A. Graesser, "Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents," *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, Springer-Verlag, pp. 21-35, 1996.
 - [3] B. M. Blumberg and T. A. Galyean, "Multi-level direction of autonomous creatures for real-time virtual environments," *Proceedings of SIGGRAPH*, vol. 95, pp. 47-54, 1995.
 - [4] D. Thalmann, C. Hery, S. Lippman, H. Ono, S. Regelous, and D. Sutton, "Crowd and group animation," *Proceedings of the conference on SIGGRAPH 2004 course notes*, 2004.
 - [5] H. Moravec, "When will computer hardware match the human brain," *Journal of Evolution and Technology*, vol. 1, pp. 1-12, 1998.
 - [6] P. Maes, "Artificial life meets entertainment: lifelike autonomous agents," *Communications of the ACM*, vol. 38, pp. 108-114, 1995.
 - [7] J. D. Funge, "Making Them Behave: Cognitive Models for Computer Animation," University of Toronto, 1998.
 - [8] J. D. Funge, *AI for Games and Animation: A Cognitive Modeling Approach*: AK Peters, Ltd., 1999.
 - [9] X. Tu, *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception and Behavior*: Springer, 1999.
 - [10] X. Tu and D. Terzopoulos, "Artificial fishes: physics, locomotion, perception, behavior," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*: ACM Press, 1994.
 - [11] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM SIGGRAPH Computer Graphics*, vol. 21, pp. 25-34, 1987.
 - [12] C. W. Reynolds, "Not Bumping into Things," *Notes for the SIGGRAPH*, vol. 88, 1988.
 - [13] C. W. Reynolds, "Steering behaviors for autonomous characters," *Game Developers Conference*, vol. 1999, 1999.
 - [14] C. Reynolds, "Big fast crowds on PS3," *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pp. 113-121, 2006.
 - [15] S. Rabin, *AI Game Programming Wisdom*: Charles River Media, 2002.
 - [16] S. Rabin, *AI Game Programming Wisdom 2*: Charles River Media, 2003.
 - [17] S. Rabin, *AI Game Programming Wisdom 3*: Charles River Media, 2006.
 - [18] S. Rabin, *AI Game Programming Wisdom 4*: Charles River Media, 2008.
 - [19] J. Lasseter, "Principles of traditional animation applied to 3D computer animation," *ACM SIGGRAPH Computer Graphics*, vol. 21, pp. 35-44, 1987.
 - [20] J. Lasseter, "Tricks to animating characters with a computer," *ACM SIGGRAPH Computer Graphics*, vol. 35, pp. 45-47, 2001.

Future work: Wildlife system

- [21] J. Adams, *Advanced Animation with DirectX*: Thomson Course Technology, 2003.
- [22] F. D. Luna, *Introduction to 3D Game Programming with DirectX 9.0 c: A Shader Approach*: Wordware Publishing, Inc., 2006.
- [23] K. Rayner and S. C. Sereno, "Eye movements in reading psycholinguistic studies," *Language*, pp. 57-81, 1994.
- [24] M. A. Ruggero, "Physiology and Coding of Sound in the Auditory Nerve," *The Mammalian Auditory Pathway: Neurophysiology*, 1992.
- [25] R. Stiefelhagen, J. Yang, and A. Waibel, "Estimating focus of attention based on gaze and sound," *Proceedings of the 2001 workshop on Percetive user interfaces*, pp. 1-9, 2001.
- [26] M. Buckland, *Programming Game AI by Example*: Wordware Pub., 2005.
- [27] D. M. Bourg and G. Seemann, *AI for Game Developers*: O'Reilly, 2004.
- [28] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pp. 25-34, 1987.
- [29] C. W. Reynolds, "Steering behaviors for autonomous characters," 1999.
- [30] J. Daniel, "Realtime Character Animation Blending Using Weighted Skeleton Hierarchies," 2000.
- [31] I. Wright, J. Marshall, W. House, and N. Street, "Egocentric AI Processing for Computer Entertainment: A Real-Time Process Manager for Games," 2000.
- [32] J. Andrews and N. Baker, "Xbox 360 System Architecture," *IEEE Micro*, vol. 26, pp. 25-37, 2006.
- [33] S. Chenney, O. Arikan, and D. Forsyth, "Proxy simulations for efficient dynamics," *Proceedings of Eurographics 2001, Short Presentations*, 2001.
- [34] S. Chenney and D. Forsyth, "View-dependent culling of dynamic systems in virtual environments," *Proceedings of the 1997 symposium on Interactive 3D graphics*, pp. 55-58, 1997.
- [35] S. Chenney, J. Ichnowski, and D. Forsyth, "Dynamics modeling and culling," *Computer Graphics and Applications, IEEE*, vol. 19, pp. 79-87, 1999.
- [36] M. Sung, M. Gleicher, and S. Chenney, "Scalable behaviors for crowd simulation," *Computer Graphics Forum*, vol. 23, pp. 519-528, 2004.