NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

METHODOLOGY AND DESIGN OF ADAPTIVE AGENT-BASED SIMULATION ARCHITECTURES FOR BAMBOO OR VISUAL C++

by

Mark A. Boyd
Todd A. Gagnon

March 1999

Thesis Advisor: Michael Zyda
Thesis Co-Advisor: Rudolph Darken

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
March 1999

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
METHODOLOGY AND DESIGN OF ADAPTIVE AGENT-BASED SIMULATION ARCHITECTURES FOR BAMBOO OR VISUAL C++

5. FUNDING NUMBERS

6. AUTHOR(S)
Boyd, Mark A. and Gagnon, Todd A.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(maximum 200 words)***

Zero-sum budgeting, downsizing, and increased mission requirements make it more challenging for U.S. Navy leaders to understand the short and long-term consequences of their decisions. An enterprise model of the Navy could provide decision-makers with a tool to study how their decisions might affect the Navy's ability to conduct worldwide operations. Agent-based simulation technology provides a flexible platform to model the complex relationships between the Navy's many components. Agent-based modeling uses software agents to define each relevant entity of the system. These agents have the ability to interact with their environment and learn or adapt their behaviors while trying to achieve their goals. The aggregate of these interactions results in identifiable behavior patterns known as emergent behaviors. This thesis looks at two methods of designing the underlying architecture for a simple agent-based simulation. A classic predator-prey relationship is modeled using a Windows/C++ implementation and a dynamically extensible Bamboo implementation. While the Windows/C++ implementation is straightforward, it requires definition of all agents before run-time. Bamboo is more challenging to implement, but allows the introduction of agents on the fly, and can easily be extended for distributed implementation. Both appear to be viable implementation architectures for an enterprise model of the Navy.

14. SUBJECT TERMS
Agent-Based Simulation, Autonomous Agents, Bamboo, Emergent Behavior, Adaptive Agents

15. NUMBER OF PAGES
109

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

METHODOLOGY AND DESIGN OF ADAPTIVE AGENT-BASED SIMULATION ARCHITECTURES FOR BAMBOO OR VISUAL C++

Mark A. Boyd
Major, United States Army
B.S., Oregon State University, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING VIRTUAL ENVIRONMENTS AND SIMULATION

Todd A. Gagnon
Lieutenant, United States Navy
B.S., United States Naval Academy, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
and
MASTER OF SCIENCE IN MODELING VIRTUAL ENVIRONMENTS AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
March 1999

Authors: Mark A. Boyd   Todd A. Gagnon

Approved by:

Michael Zyda, Thesis Advisor

Rudolph Darken, Thesis Co-Advisor

Michael Zyda, Academic Associate
Modeling Virtual Environments and Simulation Academic Group

Dan Boger, Chairman
Department of Computer Science

ABSTRACT

Zero-sum budgeting, downsizing, and increased mission requirements make it more challenging for U.S. Navy leaders to understand the short and long-term consequences of their decisions. An enterprise model of the Navy could provide decision-makers with a tool to study how their decisions might affect the Navy's ability to conduct worldwide operations. Agent-based simulation technology provides a flexible platform to model the complex relationships between the Navy's many components. Agent-based modeling uses software agents to define each relevant entity of the system. These agents have the ability to interact with their environment and learn or adapt their behaviors while trying to achieve their goals. The aggregate of these interactions results in identifiable behavior patterns known as emergent behaviors. This thesis looks at two methods of designing the underlying architecture for a simple agent-based simulation. A classic predator-prey relationship is modeled using a Windows/C++ implementation and a dynamically extensible Bamboo implementation. While the Windows/C++ implementation is straightforward, it requires definition of all agents before run-time. Bamboo is more challenging to implement, but allows the introduction of agents "on-the-fly", and can easily be extended for distributed implementation. Both appear to be viable implementation architectures for an enterprise model of the Navy.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

The authors would like to express our appreciation to our thesis committee members, Dr. Mike Zyda and Dr. Rudy Darken for their assistance, direction, and dedication throughout our course of study.

Also, for his guidance, we are indebted to John Hiles, who introduced us to agent-based modeling, and showed great patience through many meetings.

We are grateful to Kent Watsen who encouraged and guided the Bamboo implementation to include porting the simulation to the latest version of Bamboo.

For his technical support in the graphics lab we must thank Jimmy Liberato.

Finally, for their love and support we thank our families, especially our wives, Lauren and Kim, and our kids, Courtney, Morgan, and Keegan.

I. INTRODUCTION

A. MOTIVATION

Every day the Navy's top leaders make key decisions affecting the flow of money from its sources down to its resources. These decisions have certain consequences that impact the Navy's overall warfare capability, which is a direct measure of the Navy's ability to meet the global needs of the nation. Today, with the current trend of military downsizing and zero-sum budgeting, each decision made has a greater effect on the Navy's various components and their abilities to maintain the levels of readiness needed for a strong, effective force. Often, the effects of budget decisions may not be felt for a number of years. Under the current process, budget planners regularly make key decisions with neither the time nor ability to fully model how these decisions might affect the Navy in the future. An enterprise model of the U.S. Navy that contained the proper relationships between the Navy's budget allocation and its warfare capability could assist leaders in understanding the potential consequences of various decisions. This insight would help those individuals make more informed decisions in the future.

For years, the entertainment industry has developed modeling and simulation technology that in some ways surpassed comparable technology developed by the Department of Defense (DoD). The DoD normally develops modeling and simulation technology that differs greatly in use from that of the entertainment industry, but has realized that much of what the entertainment industry produces can replace, or enhance DoD technology with significant cost savings. A recent study published by the National Research Council (NRC), "Modeling and Simulation: Linking Entertainment and Defense," calls for the DoD to work with and learn from entertainment companies to better meet the DoD modeling and simulation requirements of the future [1]. As a result of this study, the Director of Naval Training (N7) requested an enterprise model of the U.S. Navy be developed that leveraged expertise from the entertainment industry.

The first decision required in the process was to determine what type of modeling technology existed in the entertainment industry that would provide the best approach for modeling the U.S. Navy. The Navy is a constantly evolving, complex system made up of many entities with sometimes-conflicting goals. To model this system requires an architecture that supports that evolution and the intricate interactions of the various components. After some consideration, it was determined that agent-based modeling, which has been used in the private and commercial sectors to successfully model large-scale, complex systems, would provide the best capabilities with which to develop an enterprise model of the U.S. Navy. This thesis explores some of the fundamental issues associated with developing an architecture for agent-based simulations.

B. BACKGROUND

Simulations are used to explore outcomes without having to become involved in expensive, time-consuming, or sometimes dangerous activities. Within this framework, simulations provide a way to answer questions, practice skills, or rehearse actions. Simulations also provide a platform to manipulate things in ways that are impossible to do with real systems. They can be started, stopped, restarted with new assumptions, and allow the introduction of entities that do not exist in the real world. Various techniques for modeling systems have been around as long as humanity. They have evolved from arranging stones to model the passing of the seasons, as seen at Stonehenge [2], to highly complex computer models like the flight simulators used to train pilots.

The fidelity built into a model depends on the kinds of questions the model needs to answer. The spectrum of fidelity ranges from aggregated or high-level models that might be used to study a military corps-level, force-on-force battle, to high-resolution or low-level models that might be used to study the human interactions of a peacekeeping operation. The ability to increase the fidelity of models has paralleled the development of high-speed computers. As processors and memory have gotten bigger, faster, and less expensive, modelers have been able to build simulations that are more intricate. Although this capability exists, high-resolution models are not appropriate in every circumstance. They are, however, particularly applicable to modeling systems where representation down to the entity level is pertinent.

Not only is capturing entity level interaction important to the result, but so is studying how these entities adapt and adjust based on these interactions. The resulting complexity of these kinds of simulations led to the development of agent-based simulations. Because agent-based simulations represent the dynamics of non-linear interactions and adaptive behaviors, they provide an outstanding environment to practice decision-making skills, and conduct training and rehearsals [3].

C. AGENT-BASED MODELING

Complex natural environments or complex systems present researchers trying to model and study them with many difficult issues. Many real world systems, often referred to as complex adaptive systems, include individual or local entities that have the ability to adapt to their environment and change their techniques for interacting with other local entities. A perfect example of this is the Earth, which has thousands of types (species) of individuals each with its own rules for interacting with and adapting to its environment. Over time, species adapt to ensure they accomplish their goal, which for most, is simply the survival of the species. The adaptive properties of the individuals often affect the system as a whole in variable and unpredictable ways; basically, the behavior of the whole system does not equal the sum of the individual components' behaviors. This phenomenon is known as *emergent behavior*, and when modeling certain systems tends to render traditional deterministic or stochastic modeling techniques inferior.

A common method of studying complex adaptive systems is through the use of computer simulations - called adaptive, agent-based simulations. Researchers trying to model their system can develop adaptive software agents that represent individual entities each with its own rules that describe how it should interact with its environment. What makes the agent adaptive is that it can revise its rules of behavior based on what it has learned from previous interactions. Adjusting its rules as it learns means the agent ensures that similar or repeated interactions will certainly produce different outcomes each time. Provided each agent is properly studied and modeled, the system as a whole will exhibit the same emergent behaviors as would be found in the real world providing the researcher with many insights to the behaviors of the entire system.

Agent-based simulations are most commonly used for entertainment and training. They provide an environment where a player, or person using the simulation, can view the potential consequences of their decision. Perhaps the most widely recognized entertainment applications are the simulation games produced by Maxis, in particular, SimCity Classic and SimCity 2000, which together have sold nearly six million copies, making them among of the best selling computer games of all time [4]. While gaming is a big market for agent-based models, the same technology is gaining popularity for training people on the dynamics of everything from budgeting to crowd control.

In the SimCity games, a player is "given a plot of barren land to zone into industrial, residential, and commercial areas. As the city grows, the player must deal with crime, education, and health issues by strategically placing police stations, schools, and hospitals. Manage traffic, the budget, and the needs of the constituents, or face riots, ridicule in the press, and eventual impeachment!" [5] The entity level interactions are controlled through an agent-based implementation; agents are the constituents. If a residential zone is provided water and electricity, people will build homes there. Population growth will stagnate unless industrial and commercial zones are designated facilitating the growth of schools, police, fire and medical protection, jobs and leisure opportunities. If an area becomes too crowded or is not properly balanced; agents interact causing riots, shifting the populations to more attractive locations, and possibly leaving the city altogether. Much like a real city, these simulated cities persist while there is constant change taking place.

Although SimCity is an entertainment application, the use of similar agent-based technology can provide city managers useful insight into the dynamics of city planning where they are able to view potential consequences of their decisions. For example, "What happens if we raise property taxes by 5%?", "What happens if we cut the police force budget or remove some police stations?" or "What happens if we build a zoo on the North end of the city?" While these simulations will not provide direct answers to the questions, they do provide the city manager with possible results of his actions. As the city manager runs through many iterations of one scenario, the new zoo for instance, he can identify possibilities of how the new zoo might affect the city as a whole - he can experiment. The zoo may bring in more tourists, cause nearby developments to increase, decrease, or stagnate, cause traffic problems, or have little effect at all. The bottom line is the simulation can identify potential issues the city manager might not have considered otherwise.

An example of a simulation that could easily be adapted for military purposes is CACTUS, an agent-based simulation developed to train senior police officers in the dynamics of crowd control [6]. The training simulation used before CACTUS consisted of a manual, pseudo-control room with incidents story-boarded before executing an expensive, time-consuming, and inflexible training exercise. Additionally, after action reviews were very limited, consisting mostly of discussion based on what people could remember and what few notes had been taken. An agent-based simulation was introduced because it provided a platform for more realistic incidents to develop, was less expensive to develop, was very flexible, and could be recorded for playback [6].

The methodology behind CACTUS is easily transferable to training military participants in the nuances of peacekeeping operations such as those now being conducted in the republics of the former Yugoslavia. These types of simulations provide key players the opportunity to plan for and rehearse actions to unexpected situations that were not realistically represented in the previous planning and training cycle.

An important note on adaptive agent-based simulations is that they do not predict the future because as events occur, there are infinitely many new states to which the current state of the environment may transition. These types of simulations only suggest individual states as possibilities and therefore do not guarantee the real world would produce the same output. Agent-based simulations simply provide a more abstract level of output that should help the researcher observe and understand complex cause/effect relationships.

D. BAMBOO

The academic and commercial sectors have developed many agent-based simulations over time; SimCity and CACTUS are two examples. Each of these allow runtime interactions where users can introduce new agents, modify agents' interaction rules, adjust behavior parameters, increase or decrease the numbers of agents, etc. These interactions, although occurring at runtime, are based on a static implementation of the simulation where all possible future capabilities were decided before the final compilation of the executable. This technique is reasonable if the simulation is modeling a system or environment whose limits are well understood and static. But, since agent-based simulations are often used to model highly complex, unfamiliar systems, a static implementation can cause certain limitations. Bamboo is a programming environment that allows users to overcome this limitation by providing a means to dynamically add functionality to a simulation at runtime. Users can create new functionality and dynamically link it to the current simulation executing without having to stop or recompile the whole system.

To illustrate the limitations of a statically implemented agent-based simulation, consider the scenario where a citrus farmer in southern California wishes to model an orange grove to help understand the effects of weather, farming techniques, and local flora and fauna on future crop yield. The farmer gathers facts, statistics, characteristics, and other pertinent information relating to the local environment, which, for his study, consists of typical weather in the area and all other plants, animals, and insects that might affect the orange crop yield. He must consider all known enemies and benefactors in the environment of the particular orange tree he wishes to grow. This is important because, like other processes that occur in nature, an orange grove is a very complex system and the omission of one small detail may cause the simulation to produce output far from reality.

Once the farmer has collected the information needed, he can design agents for each entity needed to populate the simulation. For this illustration, assume the year is 1975, and although the Mediterranean fruit fly (Medfly) has been trapped in the United States before, California has had no confirmed captures of the pest [7]. Because of this, the farmer never considers the Medfly as a potential threat to his orange grove, and therefore does not design an agent to represent it in the simulation. After spending months researching the environment where he plans to grow his oranges, and many more months designing and implementing a very robust agent-based simulation to model this environment, the farmer begins his simulation.

The simulation runs for months and begins to provide great insight to potential patterns in crop yield and tree survivability based on the interactions of all agents in the simulation. Now the farmer begins to see patterns that aid in planning the real world orange grove that he may never have considered otherwise. Assume that it is now late 1975 and the Los Angeles Times announces the first confirmed capture of a Medfly in the southern California [6]. The farmer must now reconsider attempting to grow his oranges in this area because the Medfly poses a serious threat and must be factored into his strategy. Because the simulation was originally statically implemented, the farmer must stop the simulation, design an agent to represent the Medfly, recompile the entire simulation, and run it all over again.

Had the farmer implemented his simulation using a dynamically extensible executable like that provided by Bamboo, he would have been able to design a Medfly agent and load it into the simulation while it was still running. The agents in the simulation would have been able to interact with new Medfly agent and vice-versa. These new interactions would begin to produce new behaviors or patterns that might assist the farmer in his strategic planning. This would have saved the farmer a great deal of time and money and provided more timely feedback.

Another example to highlight potential drawbacks of statically compiled agent-based simulations that may be more pertinent to a military audience is a combat simulation designed to provide insight on the expected success of various warfare tactics. Consider a scenario where forces are to be deployed on a peacekeeping operation to a war-torn country. Before actually committing forces in harms way it would be very productive to run a simulation that might provide some insight as to the potential outcomes of the operation. This would provide the peacekeepers with a platform to view potential consequences of their actions and allow them to practice reacting to various scenarios that might arise. This pre-mission training would hopefully limit the number of unexpected events during the execution of the actual mission.

As with the orange farmer example, the first thing the modeler of the peacekeeping scenario must do is gather the pertinent data. He must discover all possible information about all forces that may be involved in the operation and the environments where these operations might take place. "Who are the leaders?", "What kinds of tactics do the forces employ?", "What is the composition of the forces?", "Will they typically fight in built-up areas or open terrain?", "What are their goals?", "What are their constraints?" (especially pertinent to the peacekeeping force), etc., are all questions that need to be answered to build an accurate model.

Agents are then developed to represent entities, aggregate or individual, in the simulation. After running many iterations of the scenario, the modeler begins to notice certain behaviors emerge. He may begin to see the warring parties adapt certain tactics because of the introduction of peacekeepers. The warring parties may band together against the peacekeepers, they may remain separate but all act hostile towards the peacekeepers, some may disband or go into hiding and wait things out. The modeler now begins to experiment with ways to counter the new threats.

For this example, consider that the warring parties have banded together against the peacekeepers. The peacekeepers deploy to conduct a mission that turns into a full-blown conflict with the warring parties. The peacekeeping force commander calls for assistance - armored jeeps and five-ton trucks loaded with soldiers deploy to assist. (Requests by the commander to have tanks and infantry fighting vehicles available were denied before the initial operation ever began, so they were not built into the simulation.) The situation continues to escalate with the peacekeeping forces being divided and their reinforcements being blocked. As the scenario continues the peacekeepers begin taking heavy casualties.

The simulation has shown that there is potential for a violent conflict, something neither the commander nor his superiors anticipated. It has also shown that resources currently available to the peacekeeping force commander are potentially not adequate to handle extreme situations. The commander has the simulation run again, this time with a reaction force of tanks and infantry fighting vehicles. Since the simulation was restarted under different conditions, a conflict similar to the one witnessed in the previous run may or may not emerge. The commander does not know if this is simply a new outcome or the result of the introduction of new resources. What he really needed to know was how the employment of the tank and infantry fighting vehicle reaction force might have affected the outcome of that scenario. He needed the ability to introduce them as he saw the situation develop. If the operation had been developed using a Bamboo implementation, the tanks and infantry fighting vehicles could have been introduced "on the fly", thereby allowing the commander to see behavior patterns develop based on the introduction of new resources.

The simulation provides the commander with a tool to view situations as they arise that he may not have even considered. He can view potential outcomes, and with a Bamboo implementation, see how weapons not originally included in the simulation might actually impact the outcome of the mission. At that point he can either come up with new courses of action or go back to his superiors and request additional resources, because he has seen the potential for the mission to evolve into more than a peacekeeping operation.

The last two examples are fictional and contrived, but hopefully serve to illustrate that agent-based simulations can benefit a great deal from the dynamic extensibility that Bamboo offers. Bamboo provides the mechanisms where users or systems themselves can modify the executables on the fly without having to stop the simulation and recompile. Bamboo was originally designed to facilitate the development of real-time, networked virtual environments, and one can immediately see the potential for developing networked agent-based simulations where users from around the world could design and introduce their own agents into a commonly shared virtual environment through the Internet.

E. SUMMARY OF CHAPTERS

The remainder of the thesis is organized as follows:

- Chapter II: Agent-Based Modeling. Discusses a definition for agent-based models to include: the purpose of agent-based models, what makes an agent-based model different than other models, and what constitutes an agent-based model.
- Chapter III: Bamboo. Discusses the current implementation of Bamboo and how its capabilities are suited for dynamically extending virtual environments and simulations.
- Chapter IV: Architecture. Describes the development of a basic agent-based simulation architecture, modeling the predator-prey relationship, using both a Windows/C++ and Bamboo implementation.

- Chapter V: Conclusions. Discusses the limitations discovered during development and provides ideas as to future work that might be completed in this area.

II. AGENT-BASED MODELING

A. INTRODUCTION

Agent-based models, known by many different names to include bottom-up models, individual-base models, artificial social systems, or behavior-based models, are used to study everything from the stock market to ant colonies to the human immune system [2]. Regardless of their name, their purpose is to allow users to gain an understanding, through analysis, of the processes that appear in different complex systems [8].

At the core of agent-based simulations are independent software agents that represent the model down to the entity level. These agents populate an environment and interact with each other and the environment. Each agent has the ability to adapt or learn from these interactions - they evolve over time. While each agent has a relatively small number of possible behaviors, the sheer number of possible interactions and outcomes greatly increases the complexity of these simulations. The complexity is further increased by the inherent non-linearity of those interactions and typically produces unpredictable large-scale effects. These large-scale effects are known as emergent behaviors [8]. Agents, their interactions and adaptability, and emergence are what differentiate agent-based simulations from other types of simulations that typically aggregate behaviors instead of track individuals through time [9].

B. AGENTS

An agent is simply a software object with internal states and a set of associated behaviors [10]. Examples of what agents can represent include atoms, fish, organizations, people, vehicles, or nations [8]. A state represents attributes or properties of an agent such as identification number, sex, age, or geographic location. Some states, such as identification, are fixed for the life of the agent, while others, such as energy level, may change over time as the agent interacts with its environment [10]. An agent's behaviors provide a set of rules that describe how it should interact with its environment. These rules are often represented as a set of stimulus-response combinations, and are usually coded as IF-THEN statements [2]. An agent typically has an underlying goal such as food, survival, or wealth, and must navigate through the environment, modifying its behaviors based on interactions, in an attempt to attain that goal. The two major characteristics of agents found in agent-based simulations are their ability to interact with their environment, and through learning, their ability to adapt future behavior based on these interactions.

1. Interaction

An agent interacts with its environment and coordinates with other agents in an attempt to attain its underlying goal(s) and achieve a progressively better fit to the requirements of the environment. Their interactions may consist of many things to include mating, communication, combat or partnership [8].

Many steps must occur for a single interaction to take place. First, an agent must sense its surroundings, or environment, in order to determine whether or not there are any other agents with which to interact. Sensing is limited to a set range based on the expected real-world sensing limitations of the agent. An agent's sensors may be programmed explicitly so that each sensor has its own functionality. Another approach is to implement sensing in an abstract manner where the agent simply knows, or can access information about everything within its sensing range, but has no physical sensors to do so. This abstraction is useful when "how an agent senses" is not important compared to simple fact that it does sense because it allows developers to aggregate many sensors that an agent might actually use in the real world into one sensing capability. For example, humans use the five basic senses of touch, smell, sight, hearing, and taste to sense their environment and decide what action to take next. Rather than implement all five senses separately, it is often easier to provide a human agent with the ability to simply sense, and therefore know, everything about all other agents within its sensing range.

Once an agent has sensed its environment, it must gather information about each agent within its range to determine what course of action is required next. Gathering the information is usually accomplished through one of two ways; broadcast reception and direct interrogation. In the first method, an agent broadcasts its own state information to all other agents within range. This means that an agent within sensing range of the broadcasting agent will receive that information whether it needs it or not. For example, if two humans are within sensing range of one another. If one agent speaks, its "voice" is broadcast to any agent within "hearing" range. The second agent will hear that information whether it needs it or not. In the second method, an agent is allowed to interrogate another agent for specific information. Normally, the level of information available through direct interrogation is limited by the designer to match the level of available information that would be expected in the real world. For example, in the real world, when a herd of antelope are in mating season, a male antelope can sense whether a female antelope has already been impregnated. It makes sense then, that a male antelope agent in a simulation should be able to interrogate a female agent for pregnancy information and expect a valid reply. It is possible to combine both broadcast and interrogation techniques in an agent-based simulation since information is normally passed both ways in the real world.

Once an agent has gathered all the needed information about other agents within its vicinity, it must then determine what, if any, interactions it should attempt. Interactions may include attempts to mate, flee, or form alliances. An interaction normally affects two or more agents, therefore the outcome of that interaction must be determined fairly and equitably for all those involved. While the outcomes of some interactions are straightforward and easily determined, others, such as combat, can result in a large number of potential outcomes. To simplify the process, the outcome of a single interaction is usually determined by a referee in the simulation. A referee has access to all pertinent information needed to decide how an interaction should affect each agent involved. Once an interaction has occurred and the referee has decided the outcome, the agents involved must update their states and possibly revise their behavior rules. Referring to the mating example above, once two agents have successfully mated, the female's state value for pregnant would become true, and her behavior might be modified. She may become territorial and avoid other agents instead of moving towards them or she may require more food and therefore feed more. The level to which behavior is modified after an interaction again depends on the designer of the simulation.

2. Adaptability

The ability to adapt, or adjust, to their environment is one of the essential components of agents that distinguishes agent-based simulations from other traditional simulation techniques. Agents adapt by modifying their rules of behavior and strategies based on what they have learned from previous interactions. This adaptability greatly increases the level of complexity that can be modeled. Most agents modeled in a simulation will use two forms of adaptability, short-term and long-term, in order to attain their desired goals.

Short-term adaptability allows an agent to adjust its behaviors to satisfy some immediate requirement in the environment. It normally requires the temporary integration or switching between specific behaviors [11]. A simple example of this might be an autonomous robot agent that encounters a physical object in its path while attempting to relocate to a new location. If the robot has no prior knowledge of the object, and no generic avoidance behavior, it may collide with the obstacle. Once the collision has occurred, the robot will adjust its behavior by changing direction as needed to get around the object. The robot may alternate its behaviors between move forward and move sideways until it has cleared the object at which time it can resume its original goal of relocating. Switching between these two specific behaviors during the sequence of interactions is what makes this a short-term adaptation.

Long-term adaptation represents a higher level of learning and normally takes place over the life of the agent [11]. From the example above, the robot has learned that the object with which it collided is something it should avoid in the future. It can also remember basic information about the object, such as size and the most efficient way to avoid the object in the future. This means the next time the robot encounters the object while relocating, it will be able to avoid the object while minimizing the delay from its original goal of relocating. Over time, the robot will develop a new behavior called obstacle avoidance that represents a higher level of motion control compared to simply moving forward or sideways.

Agents that do not adapt will not be able to find their niche in the environment or achieve their goal(s). They are the ones that will perish, whether they are stock market agents trying to buy stock at a certain break point, military tactics agents trying to detect a vulnerability in an enemy's defense, or agents representing animals in the wild just trying to survive. As individual agents interact and adapt, group behaviors begin to emerge. These emergent behaviors are what provide the modeler with a platform to carry out the what-if scenarios and observe various outcomes.

C. EMERGENT BEHAVIORS

Entity level agents that learn from, and adapt to their environment by interacting with each other, provide researchers with realistic and useful views of behavior patterns that might emerge in real-world systems. These patterns, typically referred to as emergent behaviors, result from the aggregate interactions among, and adaptive nature of, individual agents [12]. They "... are often surprising because it can be hard to anticipate the full consequences of even simple forms of interaction" [8].

A good example of emergent behaviors is an ant colony as described by D. R. Hofstadter [2, 13].

> Individual ants are remarkably automatic (reflex driven). Most of their behavior can be described in terms of the invocation of one or more of about a dozen rules of the form "grasp object with mandibles, " " follow a pheromone trail (scents that encode 'this way to food,' 'this way to combat,' and so on) in the direction of an increasing (decreasing gradient," "test any moving object for 'colony member' scent," and so on. (To actually perform computer simulation of an ant following these rules, the description of the rules would have to be somewhat more detailed, but these phrases give the gist.) This repertoire, though small, is continually invoked as the ant moves through its changing environment. The individual ant is at high risk whenever it encounters situations not covered by the rules. Most ants, worker ants in particular, survive at most a few weeks before succumbing to some situation not covered by the rules.
>
> The activity of an ant colony is totally defined by the activities and interactions of its constituent ants. Yet the colony exhibits a flexibility that goes far beyond the capabilities of its individual constituents. It is aware of and reacts to food, enemies, floods, and many other phenomena, over a large area; it reaches out over long distances to modify it surroundings in ways that benefit the colony; and it has a life-span orders of magnitude longer than that of its constituents (though for some species the life-span of the queen may approximate the life-span of the colony). To understand the ant, we must understand how this persistent, adaptive organization emerges from the interactions of its numerous constituents.

While an individual ant's behavior rules are fairly small and simplistic, a complex colony emerges from the large number of ants and their interactions with the environment. The colony is much more than just the sum of the individual ants. The emergent behaviors displayed by the ant colony are the types of behaviors that modelers are looking for when they build agent-based simulations. They can model the individual entities with very basic states and behavior rules and from that alone, observe many complex patterns as they emerge.

D. SUMMARY

Agent-based models are very useful for simulating many types of systems. They are particularly appropriate for modeling realistic environments that consist of many agents interacting in a non-linear fashion. In an attempt to achieve a better fit with the environment, agents adapt future behaviors based on these interactions, resulting in complexity that is typically difficult to model using stochastic or deterministic processes. It is often more realistic and useful to provide agents with initial behaviors, let them interact, and then observe the behaviors that emerge. Agent-based modeling provides a platform where those unexpected behaviors can emerge and provide analysts with greater insight into the complexity of their models.

III. BAMBOO

A. INTRODUCTION

Bamboo is a toolkit that provides an application programmer's interface (API) for the development of real-time, networked, virtual environments (VE). Its primary focus is to provide a means for users to create dynamically extensible code. This means that applications programmed in Bamboo have the ability to dynamically reconfigure themselves by adding to or altering their functionality during runtime. It contains a series of functional modules that extend its basic execution core. Users can further extend the execution core by adding application specific modules that provide the VE with the desired capabilities. Although Bamboo was designed to facilitate the development of networked VEs, its unique features can greatly enhance traditional agent-based simulations as well. Dynamic extensibility is the most significant feature of Bamboo that will provide the greatest benefit to agent-based simulations.

B. DYNAMIC EXTENSIBILITY

Dynamic extensibility was the single most influential design issue for the creator of Bamboo [14]. Bamboo accomplishes this by implementing a plug-in metaphor much like that popularized by commercial software companies such as Netscape Navigator [15]. The biggest difference between Bamboo and traditional plug-ins is the fact that Bamboo does not require an application or system re-start in order to function. Each Bamboo module represents a plug-in that can extend the existing execution core. It further extends the plug-in metaphor by adding inter-module dependencies. Bamboo uses the plug-in concept along with the simple but robust mechanisms of callbacks and event handling to provide dynamic extensibility.

1. Dependency

Not only does Bamboo support the plug-in metaphor by allowing additional functionality be added to the executable through external modules, it utilizes modules itself to create the Bamboo runtime environment. The core executable, or "main" routine, contains only enough logic to page modules and provide the framework into which plug-ins may hook. The remainder of the functionality in the Bamboo runtime environment is provided through additional separate modules.

Developers who wish to use Bamboo to create a simulation simply need to create modules that further extend the capabilities of the existing Bamboo runtime environment. User application modules are paged into memory and become part of the current executable. Figure 3.1 provides an abstract view of the Bamboo core with external modules attached to it. This approach ensures that the programmer makes all decisions regarding an application's capabilities and that no decisions are forced by restrictions in Bamboo itself. All capabilities of an application are defined at runtime when the application is loaded into Bamboo.

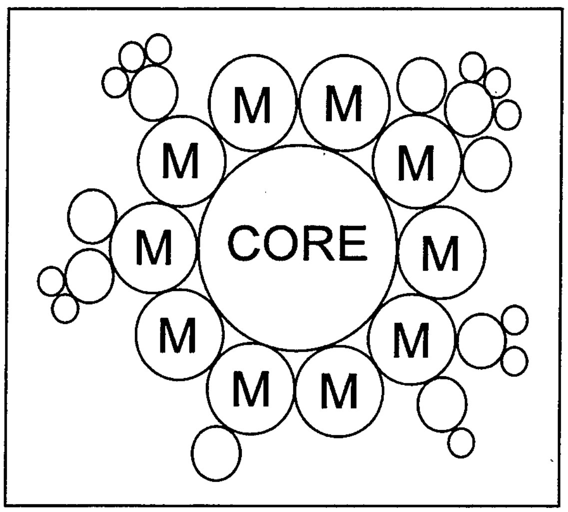

Figure 3.1: Bamboo runtime view

One of the main benefits of using a plug-in concept, is that it allows applications that need certain functionality not already in memory to load the needed modules. This is done through a dependency list where a module specifies all other modules on which it depends. Modules in the dependency list that are not active in memory are simply loaded before the application without any user interaction. A great advantage to this approach is that functionality that is not needed to run the current application, is not loaded into memory thereby saving valuable resources and enhancing system performance.

Specifying every possible module on which an application depends would be complex and difficult, so Bamboo simplifies the process by requiring an application to list only the immediate modules that it needs in memory. Bamboo then manages the system of dependencies to ensure that all required modules are loaded into memory in the correct order. Figure 3.2 depicts an example where module four (M4) is being loaded into memory. For the example, assume that the numbered modules are the application specific modules and that M3 has already been loaded into memory. As the system tries to load M4, it must first verify that M2 is in memory. Since M2 is not already in memory, the system must load M2. In the process of loading M2, the system must verify that M1 is already in memory, which it is because it was loaded when M3 was loaded. Having verified the required modules for M2, the system then loads M2, after which it can finish loading M4 [14].

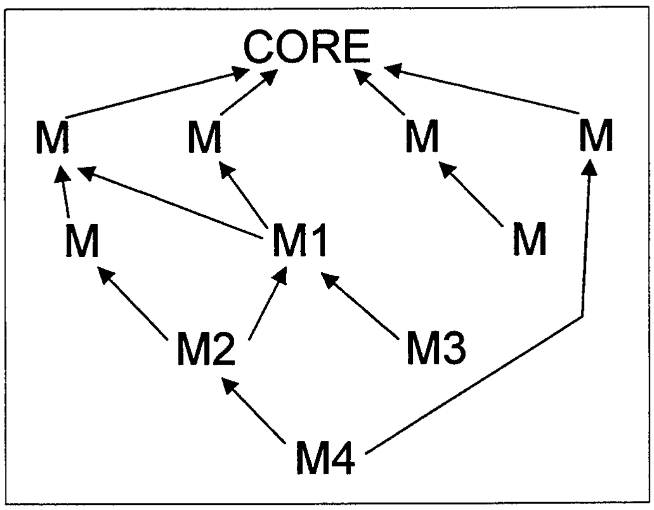

Figure 3.2: Module dependency view

2. Callbacks

The plug-in concept of Bamboo does help facilitate dynamic extensibility of a simulation, but the ability to extend the executable actually comes from the callback and callback handler. The callback is a very simple yet powerful component of Bamboo. It provides the framework to which new code can attach itself and be brought into the same address space as the executable. A callback enters the execution loop by attaching itself to a callback handler. A callback handler is a thread in the Bamboo runtime environment which shares execution time with the "main" routine and other callback handlers. A callback handler is responsible for sequentially executing each of its attached callbacks every time it itself is executed. Figure 3.3 illustrates how individual callbacks attach themselves to a callback handler.

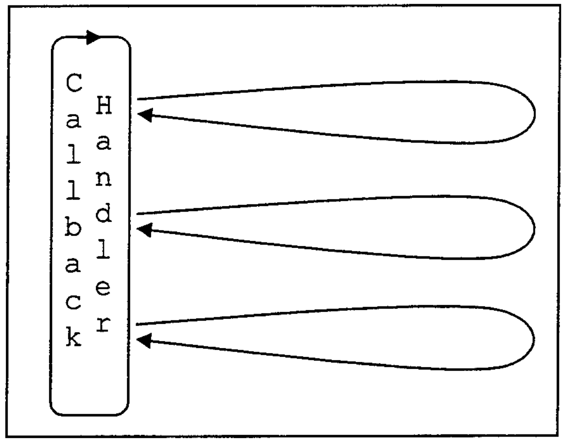

Figure 3.3: The Callback Handler

3. Event Handling

The event handler provides a useful abstraction for handling system and user generated events. It does so by using the callback handler to notify registered parties of an event via callbacks. Since Bamboo uses a callback handler for notification delivery, multiple callbacks may be executed in response to a single event.

C. SUMMARY

Bamboo breaks the paradigm of statically defined virtual environments and simulations by providing simple mechanisms to dynamically extend an executable. It accomplishes this by specifying a convention for defining new program modules, allowing those new modules to link into the executable through the use of callbacks and callback handlers, and by loading required modules for any new application without user interaction. As mentioned earlier, the ability to dynamically extend a simulation during runtime could greatly increase the utility of traditional statically defined agent-based simulations.

IV. ARCHITECTURE

A. INTRODUCTION

It is very challenging to describe the interactions among agents, especially when the agents can modify their behaviors thereby changing their rules of interaction with other agents. Developing an architecture that supports this methodology is also a daunting task. Object-oriented programming (OOP) languages, such as C++, seem to provide the best environment to program agent-based simulations. OOP provides many mechanisms that greatly facilitate the construction of agent-based models; the most significant of these include encapsulation, inheritance, and polymorphism. Agents and the environment in which they exist can all be implemented as objects; structures that hold data and procedures.[10] An agent's state is comprised of instance variables, while its behaviors are defined through methods. Inheritance provides a mechanism for defining a base class and letting modelers define agent specific routines, whereas polymorphism allows the modeler to redefine or extend the functionality of the base class if needed.

One of the drawbacks to this type of implementation is the requirement to have everything set before run time. If a modeler wishes to add a new type of agent - one not defined at run time - they must stop the simulation, update the code where appropriate, and then recompile. Bamboo appears to offer an attractive alternative to this because it affords the modeler the opportunity to define and add new agents "on the fly".

The goal of this thesis was to look at the issues associated with building architectures for agent-based adaptive simulations. We first designed the architecture using the Windows/C++ programming environment because of our familiarity with this programmer interface. As we conducted research and the architecture began to develop, we realized that the ability to add agents during a run could be very beneficial to the modeler. Discussions with Mike Zyda [16], Rudy Darken [17], and Kent Watsen [18], encouraged us to build an architecture using Bamboo, which provides the ability to implement this new paradigm.

With this in mind, we decided to model a simple predator-prey relationship to see how speed affects their interactions and the survivability of each species. This scenario afforded us the opportunity to fully exercise and view the core fundamentals of agent-based modeling, namely - agents, interactions, adaptability, and emergent behaviors. The agents are Cheetah, Antelope, and grassy feeding areas. Interactions between agents included: mating, killing, avoiding, herding, fleeing, chasing, and feeding. Through these interactions, we were able to observe how both the Cheetah and the Antelope adapt their behaviors to achieve their overall goal; which in this simulation was simply survival of the species. These interactions also lead to some emergent behaviors that we will discuss later.

The predator-prey model is called Savannah after the African Savannah where these real-world interactions take place each day. Much like the real Savannah, the simulated Antelope roam an open range in herds looking for food and potential mates, while trying not to fall prey to any predators. They may also die from infant mortality or age. The Cheetah, being solitary animals, typically avoid each other while hunting for prey in their own territory. The only time Cheetah come together is during mating season, when they will seek a mate and then return to their independent lifestyle. Like the Antelope, they can die from age or infant mortality and also starvation. Both Cheetah and Antelope have simple sets of rules to govern their behavior.

As is common with many other models, this simulation does not attempt to intricately model every detail. To attempt to model the predator-prey relationship exactly as it occurs in nature is unrealistic and is not the focus of this thesis. The normal practice, when deciding how much detail to include in the model, is to determine what is needed in the model and implement that to a sufficient level of detail. Since we were mainly interested in looking at architectural issues of agent-based modeling, only a few aspects of this relationship along with a few major components of each animal were modeled. For instance, the interactions between the Cheetah and Antelope were modeled in terms of the hunt-chase-kill cycle that exists for the Cheetah or the watch-flee-escape cycle that exists for the Antelope. As far as modeling the survivability of each species, other relationships were modeled such as mating, infant mortality, and aging. To further simplify the model, some capabilities or conditions were aggregated such as sensing ability and infant mortality.

It is also important to note that other species, which could affect the output a great deal, were not modeled in the main simulation. Again, this is because the main purpose of this thesis was not to model a Cheetah-Antelope relationship in the wild, but to discover architectural development issues of agent-based modeling.

B. WINDOWS/C++ IMPLEMENTATION

1. Introduction

The windows version of Savannah was developed on an Intergraph TDZ 2000, 400 MHz personal computer (PC) running the Microsoft Windows NT 4.0 Operating System (OS) using Microsoft Visual C++ 5.0. Visual C++ and Microsoft Foundation Class (MFC) libraries provided a straightforward programming environment to produce a two-dimensional 640 x 480-dpi display of Savannah. Although the simulation was developed on Windows NT, the precompiled version may be run on any Windows PC.

2. Interface

The user interface for Savannah was developed using Microsoft Developer Studio 97. The display provides the user with a simple, single-document window from which to view simulation runs. Making changes to the simulation requires Visual C++ and the MFC libraries. Figure 4.1 shows a typical screen shot of the interface during a simulation run.

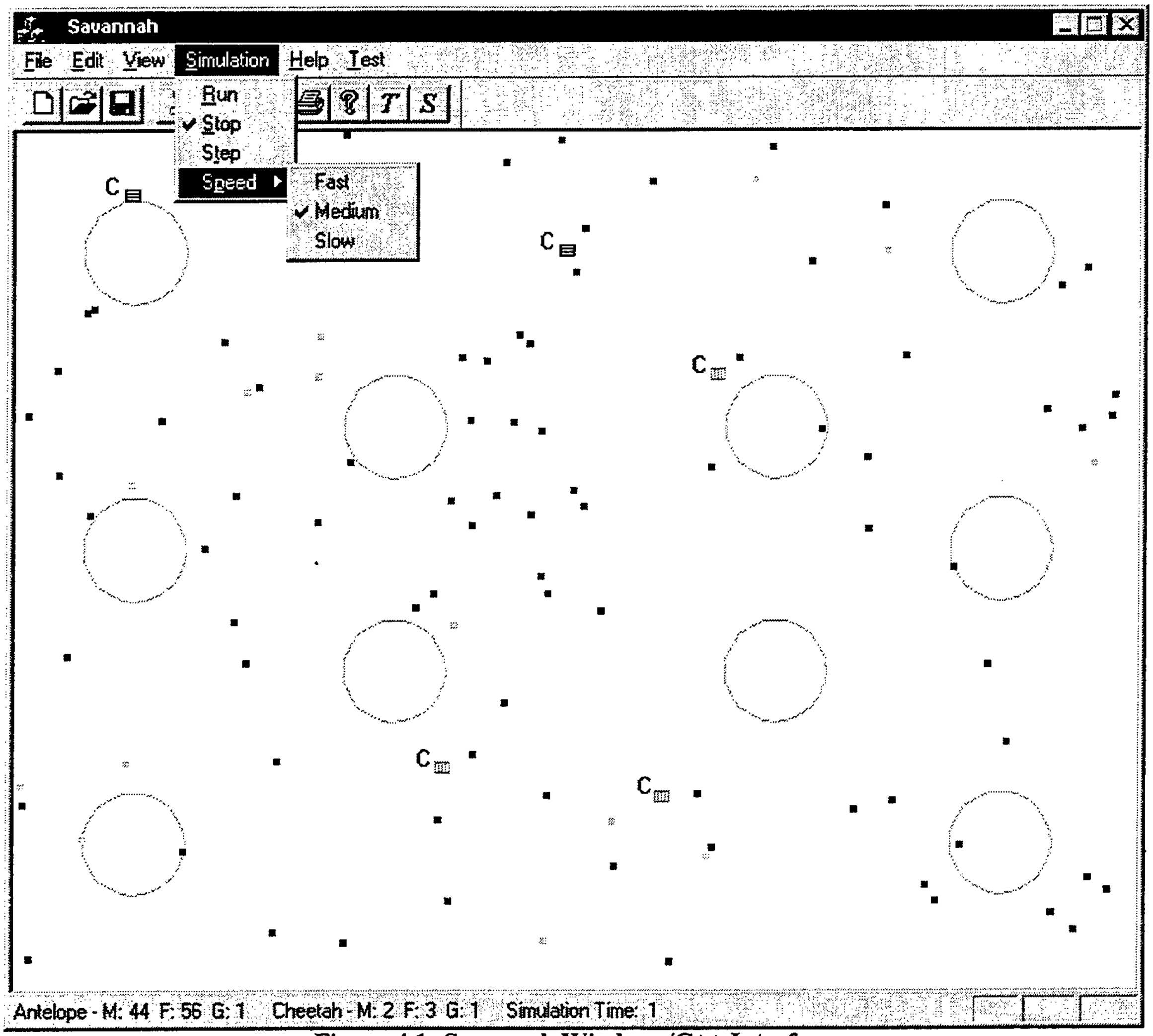

Figure 4.1: Savannah Windows/C++ Interface

The environment is initially populated with 100 randomly located Antelope and 5 randomly located Cheetah. The Antelope are color-coded in five increments, based on speed, using the Red-Green-Blue (RGB) spectrum. Red are the slowest Antelope, and blue are the fastest. The Cheetah are colored according to gender; male being black, and female being gray. For ease of identification, Figure 4.1 also identifies Cheetah with a "C".

The simulation can be started by either using the simulation pull-down menu or clicking on the "T" toggle button. The toggle button allows the user to start and stop the simulation. The simulation pull-down menu not only provides start and stop options, but also allows the manipulation of the simulation speed from slow to medium to fast, and the ability to step through the simulation run. The "S" step button, on the toolbar, also provides this step-through capability. Once the simulation has been started, the agents interact according to the architecture that is described in the following sections.

3. Architecture

a. Overall Design

When designing any model, the first thing to accomplish is to decide what is to be studied and to what detail. Answering questions such as "What will the simulation be used for?", "How much detail is needed?", "What issues may need to be studied in the future?", and "Who will use this simulation?" are often very helpful in determining an implementation structure.

In the case of Savannah, we wanted to see how speed affects the Antelope-Cheetah relationship and overall survivability of each species. To develop the architecture to support this, we initially designed the simulation using four linked lists; one list each for the male and female Antelope and Cheetah. Because most of the interactions in the simulation are based on location and distance between agents, we quickly found the linked-list implementation to be computationally prohibitive. After some experimentation, we settled on a hash table implementation using the Map class from the Standard Template Library (STL). The Map class is one of the collection classes from the STL and provides a one-to-one mapping of a unique key value and some associated data. The key can be of any valid type and the data can be a simple element or a complex data structure. For this simulation, the agents were placed into the Map based on their unique xy location in the virtual world. This allowed us to easily pare the agents that were not within sensing range when animals executed their sensing loop.

With the linked list implementation, the sensing loop required $O(n^2)$ computations because each agent had to traverse the entire list to sense those other agents within range. The Map implementation required $1/x\ O(n^2)$ where the scalar $1/x$ was inversely proportional to the number of local groups in the simulation. Since the agents were able to calculate the xy boundaries of their sensing range, they could then hash into the Map and only view those records of agents within range. As an example, if the simulation had 300 agents active, the linked list implementation required each agent to loop through all 300 records in the list so the sensing loop required 300 x 300, or 90000 steps. In the map implementation, each agent only looped through the agents within its sensing range so if the 300 agents were divided into 10 Antelope herds and 10 Cheetah, then each agent would loop through an average of 30-40 agents. This would require only 300 x 40, or 12000 steps to complete the sensing loop. So even though the final cost may appear to be $O(n^2)$, the hash table implementation did drastically reduce computational costs.

The next thing we considered was how to sequence the agent behaviors and interactions. In initial versions of the simulation the Antelope would sense their environment in a sensing loop and decide on what action to take. They would then take this action in a move loop. After the Antelope finished both loops, the Cheetah would then sense and take actions in the same manner. This gave the Antelope a one-step advantage, which would have been unrealistic and produced improper results. Therefore, in later versions we implemented concurrent sensing and action loops for each species. This meant that all Cheetah and Antelope would sense their environment and decide on their next action before any agent was allowed to move. This resulted in interactions that were more realistic and better matched what we would expect to occur in the real world.

b. Agents

When developing the architecture for an agent-based simulation, it is important to keep it as simple and generic as possible. It must be simple so that people can understand the underlying structure. If they do not understand this, then it will be very difficult to explain or make believable the complex emergent behaviors that result from the simulation. Making the architecture generic leads to reusability and aids extensibility. A generic architecture allows modelers to easily develop other agents for smooth integration into the simulation. Once the basic architecture is understood, adding a new agent only requires the need to know what basic functionality must be included in an agent. Also, implementing a different scenario would only require subtle changes to or extensions of existing code. The easiest way to implement generic reusability appears to be through OOP techniques. Figure 4.2 shows the basic class structure that was used in Savannah and will be discussed in detail in the following section.

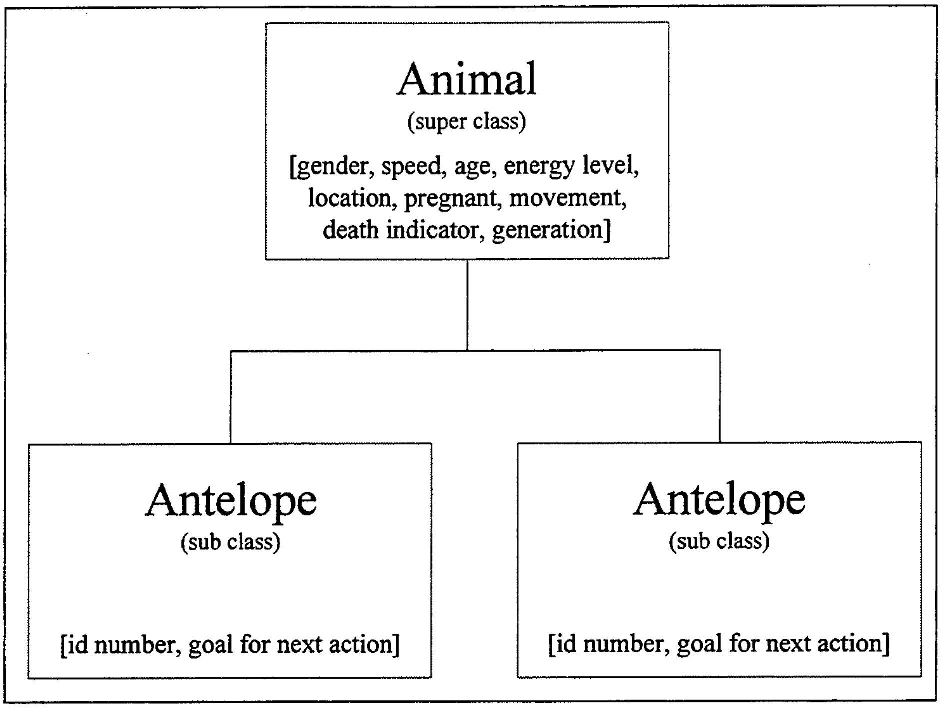

Figure 4.2: Savannah Class Structure

c. Base Class

The agents in Savannah are designed using an abstract base class with subclasses for each agent type. The use of a base class allows the identification of common characteristics and methods for all agents. We identified appropriate common attributes for animals and put them into the Animal base class. The Animal class state variables include: speed, age, generation, pregnancy state information, a mating season flag, location, a death indicator, and energy level. The Animal class includes methods that allow agents to move around their environment, avoid collisions with other agents, and virtual functions for mating and killing. The use of virtual functions ensures that modelers extending the base class include these functions in their specific subclass.

Speed is a statically implemented integer. Initially, each animal is assigned a random speed based on the minimum and maximum speed variables determined by the modeler. Cheetah are assigned an additional speed advantage to account for their sprinting ability when hunting. When a new animal agent is born, it is assigned the speed of either the mother or the father based on a random distribution.

The ***age*** and ***generation*** integer variables are used to track how old an animal is, and what generation it belongs to. Age is used to determine that an animal is old enough to mate, and at some point can cause it to die of old age. Initially, each animal is assigned a random age between a minimum and maximum age variable set by the modeler. When a new animal agent is born, it is assigned an age of zero. An animal's age increases by one unit during each simulation time step. The ***generation*** variable is simply a counter used to show how successful each species has been at reproduction. When an animal agent is born, it receives the generation value of its mother plus one.

The pregnancy state structure, ***pregPtr***, which is included with every female agent, is used as a way to carry genetic information about the father. When a new animal is born, there is the ability to numerically identify both parents, assign it the speed of either parent, tag a generation identifier to it, and assign it a sequential species identification number.

The mating season flag, ***inSeason***, simply notifies other agents that the agent is mate eligible. The modeler can control when a species is in mating season by setting the appropriate integer ranges before run time. The flag allows agents to determine if they should attempt to mate with other agents sensed during their sensing loop. The ability to manipulate the length of the mating season allows the modeler to see how shorter or longer seasons might affect the population sustainability of each species.

Location is an integer number that represents the current location of an agent in the environment. The use of a single integer number resulted in quicker position conflict detection and resolution than using an *xy* array. The number either conflicts or it does not, while in an array the agent would have to look at both elements of an array for all agents within its sensing range to determine if there is a location conflict. While location is a single integer, it does represent an *x* and *y* coordinate location. These coordinates can be returned through the ***getX*** and ***getY*** functions.

Figure 4.3 shows how a two dimensional *xy* location is converted and displayed as a single integer. The *x* position is the product of the maximum *x* value multiplied by the *y* offset plus the *x* offset for that row as annotated by the equation:

max(x)*offset(y)+offset(x). In this example max x = 640, offset(y) = 206, and offset(x) = 255. This results in an xy integer value of 16895.

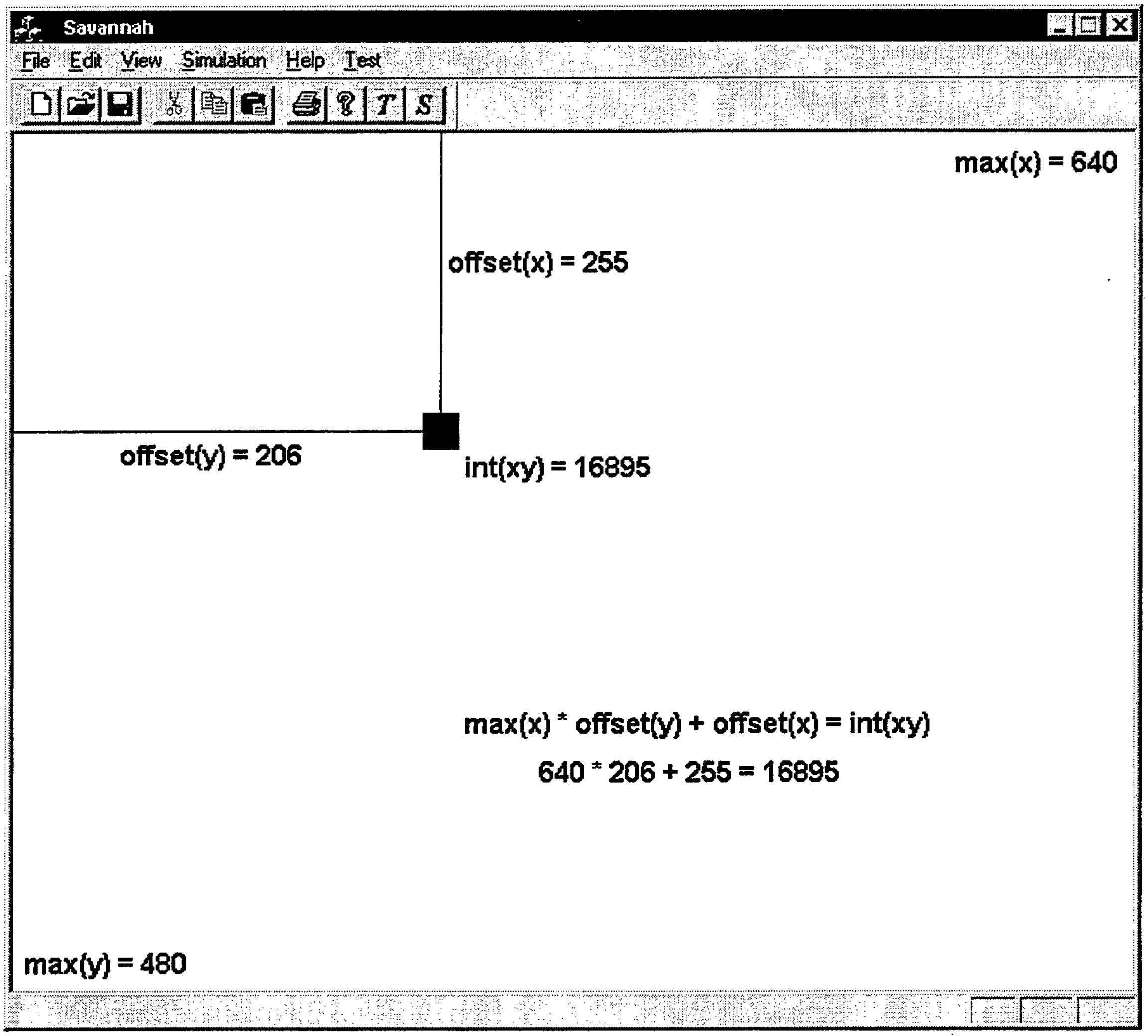

Figure 4.3: Computation of Integer xy Position

The death indicator value is a way to update an animals state indicating how it died. There are five legal entries to the ***deathIndicator*** field: ***age***, ***mortality***, ***starvation***, ***predator***, and the default value of ***not-dead***. When an agent dies its death indicator is set to the appropriate and the agent remains in the simulation for two time steps so other agents can sense it to determine how it died. When an agent reaches the maximum age it sets its death indicator to ***age***. Every agent created has the probability of dying from infant mortality. When this occurs, that agent's death indicator is set to ***mortality***. When an agent starves to death it sets its death indicator to ***starvation*** and when it is killed by a predator, the predator sets the agent's death indicator to ***predator***. After two simulation time steps the agent is removed from the simulation with the destructor method.

The ***energy level*** provides a way to put boundaries on agents actions. It can be used to trigger short-term goals in an agent, thereby dictating a sequence of possible actions. If the energy level is high enough, then the agent may not have to feed right away, but if it drops too low, the agent may have to hunt for food. Energy can also be used to force an agent to abandon a chase, if it expends too much energy, and rest. Another common use for energy is to determine if an animal has starved to death.

In Savannah, only the Cheetahs are modeled with an energy level. The integer-based energy level is used to control their hunting desires. In earlier versions, before the energy level was implemented, Cheetahs could eat a large population of Antelope in a short time; there was often no population balance. With energy implemented, every time a Cheetah kills an Antelope its energy level is boosted by a predetermined amount. The low energy level dictates the level at which the Cheetah must rest to regain strength. An intermediate level is set high enough so the Cheetah can start hunting again without immediately going below their low energy level. The high energy level acts as a hunting cut off, where once above this level, the Cheetah does not hunt, keeping it from decimating an entire Antelope population.

The animal base class also contains the methods needed to move agents around the environment. There are three move functions: ***move***, ***moveTo***, and ***moveFrom***. If an agent is not trying to move away from or towards another agent, the ***move*** function updates the agent's position based on the speed of its move: either ***rest*** or ***regular***. If the agent is moving away from another agent, such as when an Antelope is being chased, the ***moveFrom*** function is used to update its position. In this chase example, ***moveFrom*** uses the location of the Cheetah to move the Antelope in the opposite direction based on the Antelope's maximum speed. Similarly, the ***moveTo*** function is used to update an agent's position if it is moving toward a specific agent, i.e., when animals are attempting to mate.

While the agents are free to roam around the environment, the map implementation does not allow two agents to occupy the same location. Therefore, the base class also has a method to avoid collisions. This function simply checks to see if the agent is trying to move to an occupied position, and if so calls the appropriate move function until the agent has identified a position that is not currently occupied. This is also realistic in that most simulations need some kind of collision avoidance to maintain believability of interaction between agents.

As mentioned earlier, the use of virtual functions ensures that every developer of a subclass will define methods to describe actions needed to complete the model. In our implementation, we decided that determining if agents could kill or mate were not actions that should be generalized in the base class since they tend to require species specific attention. The killing tradeoff between every agent pair is different and should be decided by the developer of a specific agent.

While it may be considered an over simplification, we modeled the Cheetah's ability to kill Antelope based solely on proximity. The virtual function ***canKill*** provides the modeler with the ability to describe the agent-to-agent kill relationship in any way they would like. ***Mating*** was made a virtual function for the same reason. While cross species mating was not the main concern, we felt it was important to define the mating relationships within a specific species. This results in finer granularity than what could be provided in the base class.

d. Subclasses

The use of OOP in our simulation allowed us to develop a base class that implements attributes and behaviors common to all of the agents. In addition, the use of the class structure provided a way to implement species specific attributes. In Savannah, the Antelope subclass includes information on identification, mating, creation of new Antelope agents, and predator knowledge. The Cheetah subclass includes information on identification, mating, creation of new Cheetah agents, and killing.

Every agent in a simulation should have a unique identification number. There are many reasons why the modeler would want to know information about a specific agent. In a map or list implementation, agents must be able to identify themselves when iterating through loops. This prevents them from taking illegal actions on themselves. Additionally, specific identification numbers make it easy to track information such as; movement, mating, killing, herding, and offspring creation. In Savannah an integer identification number, ***idNum***, starting with one, is assigned to each agent based on species.

Mating is also handled in each subclass because even though mating could be described generically as either yes - they do, or no - they do not, it is more appropriate to have the flexibility to model species specific mating attributes. Defining mating as a virtual function in the base class, forces the modeler to determine if a yes or no style mating function is appropriate or if a more robust function is needed for their simulation. This provides greater flexibility and allows for agents that are more customizable. For example, while some species, such as Canadian Geese, pick one mate for life, others such as Elephant Seals mate in herds with a dominate alpha male spawning most of the offspring. A generic mating function could not account for the differences between both of these examples.

In Savannah, the mating routines are very similar in the Antelope and Cheetah subclasses. In both, the ***canMate*** function returns a Boolean expression on the ability of two agents to mate. Several things factor into determining the outcome of this Boolean logic to include: distance, age, season, species, sex, and whether or not the female is pregnant. If the function returns true, the agents then mate and the female agent begins her gestation period. Figure 4.4 shows the implementation of this logic.

```
bool Animal::canMate(Animal &potentialMate)
{
   bool mateFlag = false;
   if(this->getGender() == MALE)
   {
      mateFlag = ((!(potentialMate.isPregnant())) &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE) &&
                  (abs(this->getX() - potentialMate.getX()) <= MATE_DISTANCE) &&
                  (abs(this->getY() - potentialMate.getY()) <= MATE_DISTANCE));
   }
   else
   {
      mateFlag = ((!(this->isPregnant())) &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE) &&
                  (abs(this->getX() - potentialMate.getX()) <= MATE_DISTANCE) &&
                  (abs(this->getY() - potentialMate.getY()) <= MATE_DISTANCE));
   }
   return mateFlag;
}
```

Figure 4.4: Method to Determine if Two Animals Can Mate

If agents choose to mate, they will execute the ***mate*** function. The mate function is used to initialize the pregnancy information to include setting the females state to pregnant, recording the ***male id*** and ***speed*** for genetic information, and starting the gestation time counter. The ***gestation*** counter is simply an integer counter that increments each time step and can be set to account for species-specific gestation periods. When the gestation period ends, a series of functions determine how many agents will be born, and what attributes they will have.

The litter size is determined using a Normal distribution function. Species specific, minimum and maximum number born are entered and the conditional probably distribution function returns an integer for the number of agents created. To account for infant mortality, each potential agent is then tested to see if it dies as an infant in the ***diesAsInfant*** function. In Savannah, infant mortality includes any agent that would die within the first two years its life. Since infant mortality rates are also species specific, a floating point number from 0 to 1, representing the probability of infant mortality, must be entered for each subclass.

To streamline the simulation, only those agents that do not die of infant mortality are created. However, there are still methods to track the initial litter size and number of these that die as infants. New agents are created in the ***giveBirth*** function. This function assigns each new agent a species-specific integer identification number, an integer speed from either the mother or father, and an initial xy location near the mother.

The Cheetah subclass contains an additional method for killing, ***canKill***. This method simply tests if the Cheetah is close enough, and has the energy, to kill the Antelope. Figure 4.5 shows the implementation of ***canKill***. This results in the slower Antelope typically being killed off first, but also causes the slower Cheetah to eventually die of starvation. Successful kills result in the Cheetah manipulating the Antelope's state; setting its ***death indicator*** to ***Predator***.

```
bool Cheetah::canKill(Animal &prey)
{
	bool killFlag = false;

	if((abs(this->getX() - prey.getX()) <= KILL_RADIUS) &&
	   (abs(this->getY() - prey.getY()) <= KILL_RADIUS))
	   if(Animal::myRand() > .5)
	      killFlag = true;
	   else
	      killFlag = false;

	return killFlag;
}
```

Figure 4.5: Method to Determine if Cheetah Kills Prey

While the Antelope does not require a method for killing, the subclass does contain an additional method for acquiring predator knowledge. This is an attempt provide actual learning to the agent, thereby facilitating adaptation. It is a Boolean function that will be described in greater detail in the learning and adaptation subsection below.

e. Agents Summary

Taking advantage of the functionality offered by the C++ class structure appears to be an efficient methodology for representing agents. A generic base class offers the flexibility to extend it in order to meet almost any need. Once the agents have been correctly represented, their interactions need to be implemented in such a way as to produce believable, understandable results.

4. Interactions

Agent interactions are one of the essential characteristics of agent-based models. While the base architecture describes the state variables and methods of the agents, the methodology used to sequence interactions is also very important to the underlying implementation of these simulations. If events are not properly ordered, possible outcomes can be unrealistic, unbelievable, and very difficult to explain.

As stated above, Savannah is executed in two loops; a sensing loop and a movement loop. The underlying architecture to include the methods executed during these loops has already been described. The purpose of this section is to take a high-level look at how and why agent interactions were prioritized.

In the sensing loop, Antelope have five possible actions. They can flee, mate, move toward potential mates, herd, or feed. The priority of these events is very important to the outcome of the simulation. If the Antelope's first priority were always to mate, they would typically not be looking out for predators and could easily fall prey to the Cheetah. We chose to prioritize the Antelope's actions based on its current state, and what it sensed in its environment. The movement loop simply ensures that all agents simultaneously executed the proper move to achieve their current goal.

If an Antelope has predator knowledge and there is a Cheetah within its sensing range, it will always flee, no matter what the Cheetah is doing. If an Antelope is not fleeing and is in mating season, its next priority is to mate if it can. If there is not a mate in proximity and it is mating season, it will move toward the nearest mate eligible Antelope. If none of the above conditions exist, the Antelope will either try to move towards other Antelope or feed. The effect is the appearance of herding and searching for food simultaneously. This priority of actions seemed to result in the most realistic behaviors and outcomes.

Once the Antelope's desired actions are set, the Cheetah iterate through their sensing loop. This ensures that Cheetah are determining what action to take based on the current state of the environment. Cheetah have four possible actions to take including mating, moving towards a mate, avoiding other Cheetah and hunting. Since they have no predators in Savannah, Cheetah will always mate if in mating season and they are close enough to a potential mate. Otherwise, if it is mating season they will move toward the closest potential mate. When not in mating season, Cheetah try to avoid each other, and when their energy level becomes low enough, they are driven to hunt Antelope. Again, the Cheetah's actions are always constrained by their energy level.

Although we set the priorities of the agents, we believe it would be more appropriate for them to have the ability to set and adjust their own priorities. To do this, they must have the ability to interrogate other agents to determine other agents' states. For example, if an Antelope can sense a Cheetah, it should be able to tell if that Cheetah is hunting, mating, or taking some other action. Then the Antelope can make a more intelligent decision on what action to take in the presence of a Cheetah, rather than always fleeing when it senses one. Fleeing may cause it to be sensed when it may have remained undetected if it had rested.

The interactions also play a major role in determining what learning and adaptation the agents can accomplish. By setting the priorities for the agents, it appears we have constricted their ability to learn and therefore adapt. The learning and adaptation we see is Savannah is very basic. There appears to be a fine line as to how much guidance we should provide the agents. It is not only very difficult to hard code all interactions, but also limits the emergence of new behaviors. On the other hand, if they are just thrown in an environment with no guidance, they do nothing. A few basic rules to get and keep agents interacting seems to be the key to achieving true learning and adaptive behaviors.

5. Learning and Adaptation

Providing the agents with the ability to learn and adapt their behaviors is the most challenging component of agent-based modeling. Once they have been given a set of simple basic behaviors and interactions, how does one provide the agent with the ability to learn things that can not be anticipated? Then how can one tell if an agent is actually learning and adapting its behaviors? To look at these questions Savannah implements a simple learning routine that allows the adaptive behavior to be easily recognized. A more robust learning implementation methodology, developed for the Bamboo implementation, will be discussed in that section.

Savannah implements one learning routine based on a Boolean value. This method results in constrictive learning, because the agents appear to only have the ability to learn things that are determined by the modeler. However, some of the emergent behaviors discussed in the next section indicate learning and adaptations are occurring on levels that can not be directly traced. To test the Boolean flag method of learning, Antelope were provided with a "memory" field, called ***predatorKnowledge***, to learn and store knowledge about predators.

Predator knowledge is a Boolean that indicates the agent either does or does not have knowledge of predators. To test this method of learning, when Antelope agents are created their ***predatorKnowlege*** flag is either set to true, indicating they have the knowledge that Cheetah are predators, or false, indicating they do not have this knowledge. In the simulation loop, the predator knowledge field triggers the Antelope to flee if a Cheetah is within their sensing range.

Antelope who do not have predator knowledge are able to learn it during the sensing loop. During each loop an Antelope will sense all other Antelope within a specified range and if it senses a dead Antelope, it will look at the Antelope's state values to see how it died. If it died from a predator, the sensing Antelope's ***predatorKnowledge*** flag is set to true. The Antelope has learned that Cheetah are bad and from then on will adapt its behavior to flee from them if they are within its sensing range. This learning and adaptation cycle can be seen in figures 4.6, 4.7, 4.8, and 4.9. Figure 4.6 shows the entire Savannah environment.

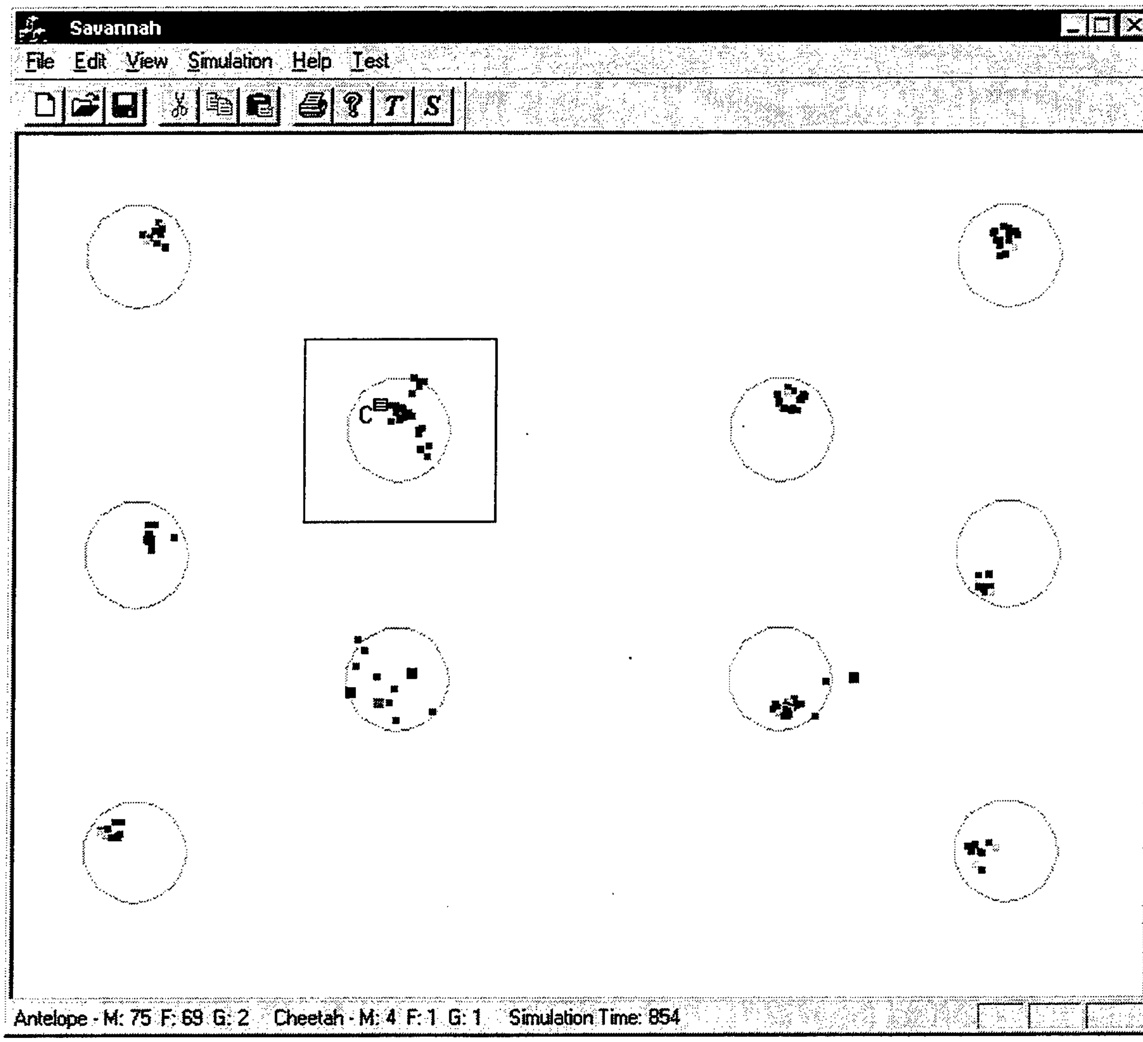

Figure 4.6. Learning and Adaptation in Savannah

Figure 4.7 through 4.9 are magnified views of where an interaction results in learning and behavior adaptation. Figure 4.7 shows a Cheetah that is able to intermingle with Antelope. Antelope this close to a Cheetah do not have predator knowledge and therefore do not flee. Figure 4.8 shows the same Cheetah just after it has killed an Antelope. The nearby Antelope will now observe this interaction the next time they sense. Figure 4.9 shows that the Antelope within sensing range of the kill are now attempting to move out of the area. They have learned that Cheetah are predators and have adapted their behaviors to flee from them.

Figure 4.7: No Predator Knowledge

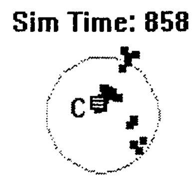

Figure 4.8: Cheetah Kills Antelope

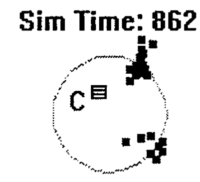

Figure 4.9: Antelope Learn and Flee

Both Antelope and Cheetah agents appear to learn and adapt their behaviors based on the emergent behaviors that are displayed in Savannah. These behaviors and what they might indicate are discussed in the following section.

6. Emergent Behaviors

Emergent behaviors are the result of the agent behaviors, interactions, learning and adaptation as described above. They are identifiable, believable occurrences of events that emerge from complex adaptive agents interacting in a given environment. The behaviors are present in virtually every run of the simulation, but when they appear they may vary drastically based on when the underlying events that cause them occur.

In Savannah, we identified several possible emergent behaviors. All of them make sense when compared to what is expected in the real world. Some emergent behaviors are obvious, while others are so subtle they are difficult to differentiate from behaviors that are programmed to occur. Emergent behaviors noticed in multiple runs of Savannah include: Antelope herd sizes, Antelope become faster as the species evolves over time, Cheetah appear to loiter around Antelope feeding sites, and Cheetah eventually resort to group tactics for hunting faster Antelope.

One of the actions Antelope in Savannah are programmed to do is to find other Antelope. This results in them eventually forming into herds. While this in itself is not an emergent behavior, the disposition of the herds appears to be. There is no algorithm to track or pare the herd size, yet the Antelope typically form into several herds of eight to twenty members. It is conceivable that all Antelope in Savannah could form one big herd, but the simple routines that require an Antelope to eat, mate, and flee from Cheetah all result in a moderation of herd sizes. Within these herds the Antelope populations typically get faster over time.

As seen in the real world, the slower Antelope in Savannah tend to be killed at a higher rate than the faster Antelope, although Antelope with no predator knowledge have the same chance of being killed regardless of their speed. The Cheetah does not have the ability to look at an Antelope's speed to determine which one to chase. They simply take up the case if they have the energy and can sense an Antelope. As the Antelope flee, the slower ones typically fall behind and are eaten by the Cheetah. As one might expect to see in the real world, it becomes more difficult for the Cheetah to successfully hunt as the Antelope population gets faster. Two behaviors appear to emerge to offset this phenomenon. First, Cheetah begin to remain closer to the Antelope feeding sites and secondly, they begin attacking in groups as they compete for food.

In Savannah, the Cheetah appear to quickly stake out their territory and typically remain within it as long as there are Antelope present. As the simulation progresses and the Antelope get faster, it takes more energy for the Cheetah to hunt. Patterns of loitering near Antelope feeding sites seem to develop. This is probably explained by the fact that the Cheetah need to conserve energy so they can continue to hunt and mate. Once they have identified a source of food, they do not need to roam as much to eat. If the Antelope population becomes fast enough or sparse enough, the Cheetah start to increase their roaming distance.

As the Cheetah begin to roam bigger areas, they tend to encounter more Cheetah. If the simulation progresses so that there becomes a competition for food, it appears as if the Cheetah begin to hunt in groups to corner the faster Antelope. This emergent behavior is in no way programmed or expected, but does make sense. The need for energy appears to cause them to modify their behavior in an attempt to corner Antelope, ensuring at least one of the Cheetah will receive an energy boost. If they were to remain apart, they would likely all die of starvation although there may be plenty of Antelope remaining.

Emergent behaviors also often seem to be in the eye of the beholder. What may appear as emergent to one may not even be recognized by someone else. What is consistent is that they are non-programmed phenomena that are explainable and identifiable when one considers all the low-level interactions that occur to make them emerge.

7. Windows/C++ Implementation Summary

An object-oriented architecture appears to be a very good way to implement agent-based simulations. It offers a structure that allows for easy, straightforward declaration and extension. Once the base class or classes have been identified, it becomes very simple for other users to modify the simulation. The Savannah implementation showed that with a simple, well-defined architecture, the basic elements of agent-based simulations can be achieved. While there are perhaps many ways the implement these simulations, it is important to note that believable outcomes will show whether or not the architecture has truly hit the mark.

Perhaps one weakness in this implementation is the requirement to have all agents defined at run time. The ability to dynamically extend a running simulation is very attractive for all the reasons discussed in previous chapters. To take a look at how such an architecture might be implemented, we next modeled Savannah using Bamboo. We named this implementation Savannah 3D.

C. BAMBOO IMPLEMENTATION

1. Introduction

The methodology behind the Bamboo architecture implementation is considerably different from the Windows/C++ implementation, although Bamboo still uses Microsoft Visual C++ to compile the code. The main difference stems from Bamboo itself, which is a toolkit that extends the functionality of the preexisting C++ libraries and then provides an execution layer above the Windows NT kernel. The code executed in Bamboo runs inside this layer. As a proof of concept for the Bamboo version, we built a predator-prey relationship very similar to our Savannah simulation and called it Savannah 3D, since its display window provides a three-dimensional (3D) representation of the simulation.

The following sections describe the Bamboo architecture implementation, but due to its similarity with the Windows version, we will only highlight the areas where Savannah 3D is different. For that reason, the emergent behaviors subsection seen in the Windows version will not be covered since this area did not change.

2. Interface

Running on Windows NT 4.0, Bamboo provides the user with a command-line interface through a DOS shell. Modules can be loaded into or removed from the execution core during run time with the *dynamicPageModule*. This is significant in three ways. First, it allows the modeler to create and load new agents. Second, modelers have the ability to remove an agent from the world. Third, which combines the first two, a modeler can remove an agent, redefine and reload it. What differentiates this from the traditional simulation methodology is that this can all be done without halting the simulation, providing greater flexibility. Using the *dynamicPageModule* also results in a smaller executable because users only load those modules necessary for a given simulation run.

To convert our Windows/C++ version over to Bamboo, we created five separate modules. The first module, *agentDisplayModule,* which simply creates a single-document, OpenGL window that represents an empty 3D world. Unlike the Windows version, the OpenGL window used to display the simulation is fully sizeable. The other four modules – *npsAgentModule*, *antelopeModule*, *cheetahModule*, and *grassModule* will be discussed in the following subsections. Figure 4.10 shows the agentDisplayModule with numerous instances of the Antelope, Cheetah, and Grass modules.

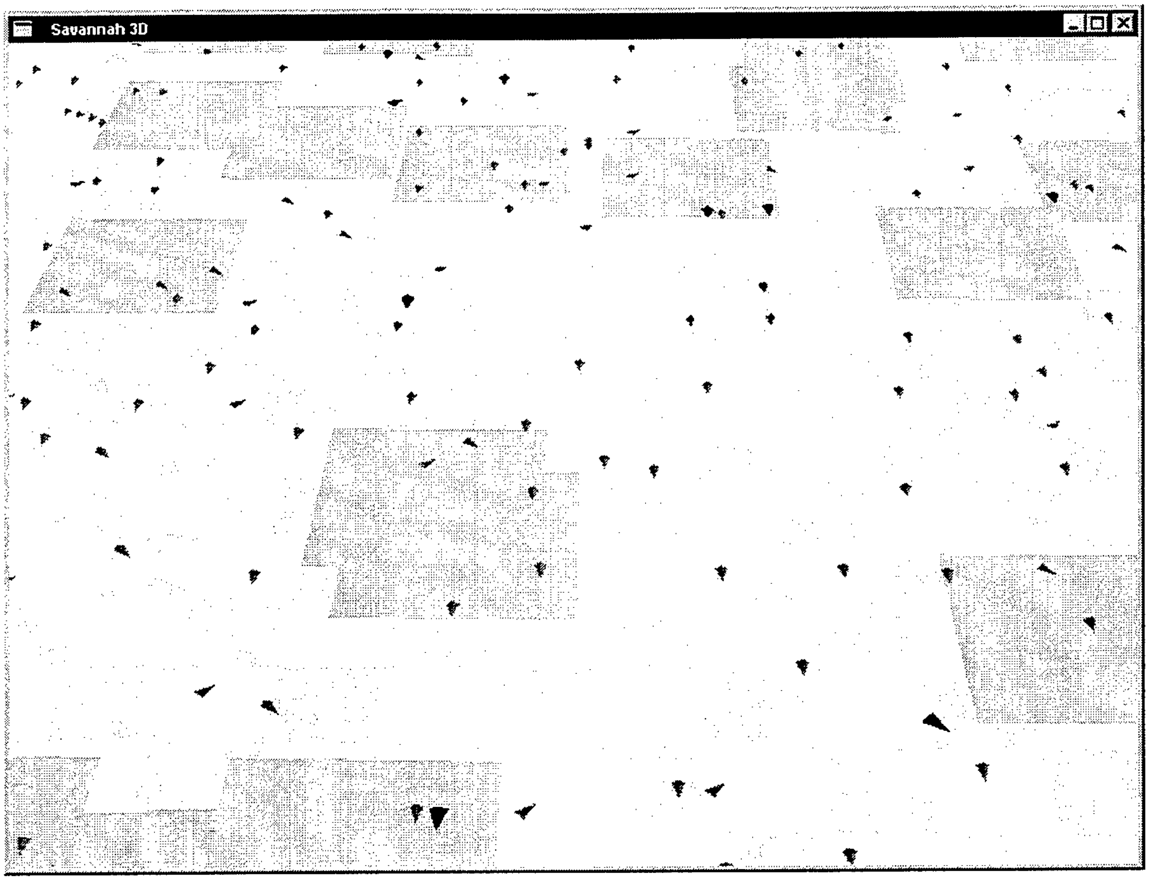

Figure 4.10: Savannah 3D with Loaded Modules

Users navigate through the world using the mouse to control 3D flight. The left mouse button controls forward flight, while the right mouse button controls backward flight. The world also has three predefined views that can be invoked using the keyboard. The first, invoked by the ***spacebar***, is on ground level at the origin looking in the direction of the negative z-axis. The second, invoked by the "***t***" key, is 200 units above the origin looking straight down, and the last, "***ctrl-t***" is located at x=50, y=100, and is looking back to the origin. As users develop modules for the simulation, they can define other keystrokes to invoke new camera viewpoints. If a user desires any type of output from the simulation, text can be written to the DOS shell. Functionality that will soon be implemented in Bamboo will provide a Graphical User Interface (GUI) where the user will be able to change agent attributes and view output from the simulation in a separate GUI window.

3. Architecture

a. Overall Design

Many of the issues we encountered implementing the Windows version regarding which data structure to use for object control and manipulation were eliminated by using Bamboo because it has this functionality built in. Every agent created in the simulation is based on an underlying object class in Bamboo called *bbListedClass*, and has an associated geometry, *npsGeometry*, that represents the agent in the virtual world. The *bbListedClass* automatically places the agent objects on a control list that can be traversed at any time by obtaining a handle to the list. Although this is a linked-list implementation, Bamboo is multi-threaded so we were able to fork a new thread to control the agents' move and sense loops. Separating computational requirements through the use of threads increased performance over our previous version by ensuring the graphics engine was given access to the processor in regular intervals and allowed to refresh the world at a decent rate. In Savannah, the graphics draw functions were executed sequentially in turn with the move and sense loops. This meant that it could only refresh the display window after all agents had completed one pass through their move and sense loops, which caused a noticeable screen flicker as more agents populated the world.

b. Agents

When developing Savannah 3D, most of the agent architecture was similar to our Savannah version although we did try to further develop the class structure. In the Bamboo version, the focus was to design a more generic agent-based simulation that would provide modelers greater flexibility in creating new agents that could seamlessly plug into a running simulation. To that end, we created a generic *npsAgent* class as the base class and then extended it to create our *Animal*, *Plant*, *Antelope*, *Cheetah*, and *Grass* classes. Figure 4.11 shows the class structure as it was implemented.

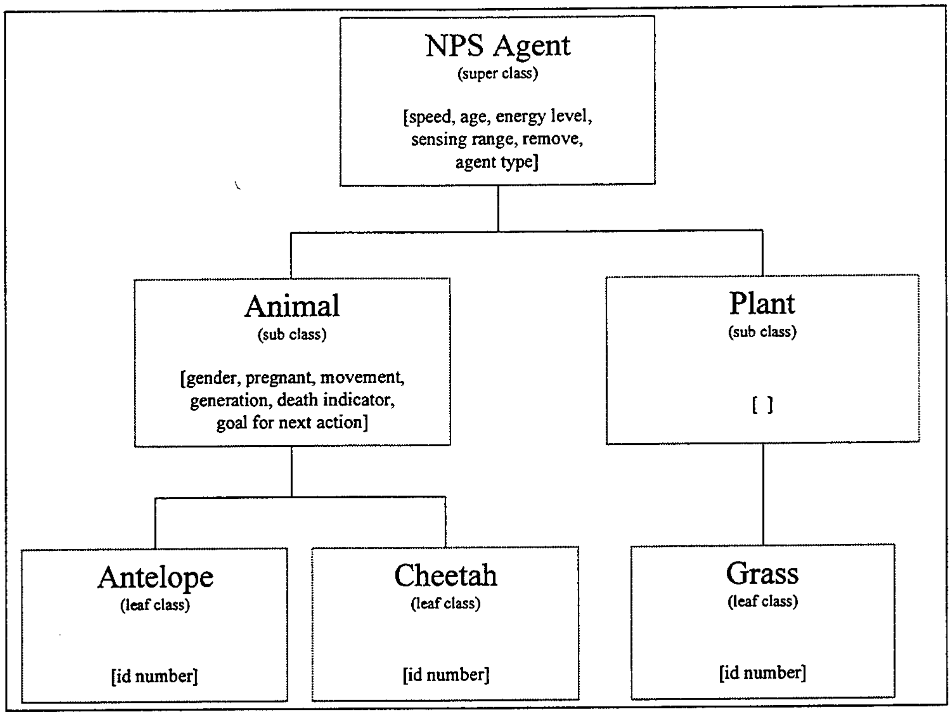

Figure 4.11: Savannah 3D Class Structure

As a rule, every agent class that would eventually be instantiated in the simulation had a module of its own, and resided as a leaf of that tree, so our simulation had *AntelopeModule*, *CheetahModule*, and *GrassModule*. Each of these modules contains the specific class and all application functionality needed to create the agent in Savannah 3D. All other abstract classes to include *npsAgent*, *Animal*, and *Plant* were included in the *npsAgentModule.*

c. *Base Class*

The *npsAgent* class was designed to implement only the basic state variables and functionality that might be needed by all future agents. To facilitate the addition of many different types of agents, we created a generic agent that implemented a base class with the following state variables – speed, age, sensing range, and energy level. We also included an agent-type attribute and a Boolean flag that can tell Bamboo to remove the agent once it is no longer needed in the simulation. To further develop the learning capabilities of the agents in Savannah 3D, *npsAgent* contains a set of vectors that allow an agent to store and remember class names of other agents it has discovered in the environment. The vectors include ***knownPredators***, ***knownFriends***, ***knownEnemies***,

knownEnergySources, and ***unknownAgents***. If an agent discovers a new agent and establishes a relationship with that agent, it can add the new one to the appropriate vector and use that information to guide how it interacts with the agent in the future. If it can not determine the relationship, then it must add the agent to its unknown vector. This implementation is a slightly more robust form of memory than the simple Boolean-flag mechanism used in Savannah. The extra memory allows an agent to develop a better database of information about its world and allows it to interact with the environment at more sophisticated level. The various benefits of this approach will be discussed later.

A location field was not required in the base class with this version because the Bamboo *npsGeometry* class included with each agent object contains a 3D-position field. Bamboo also provides a three-element vector class that can be used to pass or update the x, y, z coordinates of the geometry's position field.

Since we did not want the *npsAgent* to define the sensing and moving functions for all its subclasses, we included virtual functions for each to ensure that the modeler would implement these for every agent developed for a simulation. When a simulation runs in Bamboo, the only way it can track the agents and allow them to update their positions is by handling them all as *npsAgent* class objects. By including the ***sense*** and ***updatePosition*** virtual functions, Bamboo can loop through the list of active objects (which it recognizes as npsAgents) and call the two functions. Polymorphism allows the simulation to dynamically link to the correct definition of the sense and move methods by checking the derived class hierarchy until it finds where the methods are defined.

d. Subclasses

Savannah has five subclasses. The Animal and Plant are abstract classes that implement functionality common to all animals and plants respectively. The next two, *Antelope* and *Cheetah*, are subclasses of *Animal*, and the last, *Grass* is a subclass of *Plant*. Figure 4.8 shows how each of these classes contributed to our architecture. *Animal*, *Antelope*, and *Cheetah* remain virtually the same as they were in Savannah. The new subclasses included in Savannah 3D are the *Plant* and *Grass* classes. *Plant* is an abstract class that defined attributes and methods needed by all derived plant agents. *Grass* is a very simple class that extends *Plant* and implements a grass agent with no interaction or functionality. It was created only to add grass to the simulation to provide the Antelope with feeding areas.

4. Interactions

The interactions defined and witnessed in Savannah 3D did not differ significantly from those in Savannah. The one change was the elimination of the referee that was used in our Windows implementation. As mentioned in chapter II, the outcomes to interactions between agents is normally decided by a referee that has knowledge of the whole system including all agents. The referee must decide a fair outcome and indicate that to the agents. This is easy to accomplish in a statically developed simulation where all agents that will ever enter the world are known ahead of time.

In Savannah 3D, all agents are derived from the same base class which requires them to contain enough built-in logic to learn about other agents and determine the outcomes of interactions on their own. It would be impossible to program a referee that had knowledge of all potential agents that might enter the world, because Bamboo allows new agents to be implemented after the simulation has been created and compiled. The overhead associated with having a referee who could dynamically learn about every agent to ever enter the simulation would be too costly. Also, the referee would in fact be performing the interrogate-learn functions that all other agents would be doing, making the referee no better than any single agent. For these reasons, a referee in a Bamboo implementation is neither practical nor needed.

5. Learning and Adaptation

This is probably the most important, and, as we mentioned in the Windows architecture section, the most challenging part of creating an agent-based simulation. From Savannah, we determined that agents should have memory and corresponding logic that allowed them to make smarter decisions while navigating through the simulation. A desire to provide this prompted the creation of the dynamic vectors mentioned in subsection c above. Each vector allows the agent to store class-names of any agents it encounters, into groups based on its relationship with each agent. As the relationship develops or possibly changes over time, the agent can move or delete the reference to that agent to keep track of the appropriate relationship.

In order for the memory mechanisms to benefit the agent, each agent must have logic that takes advantage of them. While the functionality was included with *npsAgent* to manipulate the contents of each memory vector, no logic was provided to tell the agent what to do with the information. Since every agent pair will establish specific relationships with each other in ways that minimize cost and maximize payoff, it would not be possible for the base class to try to provide that logic.

To demonstrate how to implement logic that might complement the memory provided in Savannah 3D, we implemented the same learning for Antelope that we had done in the Windows version. In order to facilitate an Animal agent in learning about any predators or enemies it might have, we included a ***killer*** field in the Animal class. Now, if an Antelope agent is killed by a Cheetah, it will set the ***killer*** value to "Cheetah". Any other Antelope within sensing range will see the dead Antelope and be able to determine that the Cheetah agent was the killer. With this knowledge, the Antelope agents can then add "Cheetah" to their ***knownPredator*** list and act accordingly the next time they sense a Cheetah. Again this is a very simple example, but the goal was only to explore the possibility of a more robust learning and adaptation method.

6. Bamboo Implementation Summary

The Bamboo toolkit provides the basis for a very dynamic implementation of agent-based simulations. The predefined functionality hides many of the implementation details, so the modeler can concentrate on properly extending existing modules with well-defined agent models.

The real attractiveness of Bamboo though, is dynamic extensibility. The architecture implementation of Savannah 3D displayed a rudimentary version of this capability. As mentioned earlier, this greatly increases the flexibility of a simulation. For example, modelers often define entities to represent specific interactions or relationships in the real world. After observing simulation runs for a period of time, they begin to identify new aspects of the entities that should be studied. As they identify specific attributes of the agents that should have been included in the initial implementation, they can use Bamboo to unplug and redefine the agent to explore new relationships or interactions. Conversely, the Windows/C++ implementation would require modelers to stop simulation, redefine the agents, recompile the simulation, and then start the simulation again. Dynamic extensibility is a robust feature of Bamboo that provides modelers with unlimited options when deciding on how best to model a particular agent.

Another very nice feature of Bamboo is the ease with which simulations written in C++ for other applications can be ported over. Very little of the actual methodology behind Savannah had to be changed to build Savannah 3D. In fact, since Bamboo provides functionality not available with other programming libraries, some of the implementation can actually be streamlined during the conversion process. Again, we saw this when all of the location functionality of Savannah was removed on the conversion to Savannah.

Perhaps the biggest benefit to this type of implementation is the fact that modelers can create and execute simulations on multiple platforms, while the Windows/C++ implementation is constrained to the Windows OS.

D. SUMMARY

The preceding sections provide an overview of two different architectural implementations for agent-based simulations. The predator-prey models, Savannah and Savannah 3D, were built to explore issues associated with developing agent-based simulations. Both the Windows/C++ and Bamboo designs seem to be feasible options for building these simulations. Both implementations also take advantage of the functionality offered by OOP languages. These advantages include encapsulation, inheritance, polymorphism, and the STL.

The Windows/C++ version is a relatively straightforward implementation, in that it is a convention easily explained and understood. The class structure provides an ideal way to model agents and their behaviors. Allowing all agents to simultaneously sense and act on simple rule sets results in realistic interactions among agents, and often produce complex emergent behaviors that allow the researcher to conduct cognitive experiments.

The Bamboo version is more abstract, but in the long run, a much more attractive implementation. Not only does it offer all the advantages of the Windows/C++ version, but also provides the capability to dynamically add agents to a running simulation. This methodology makes programming very challenging; agents must not only interact and adapt to agents that are known at run time, they must also do so with agents that were not defined before run time.

V. CONCLUSIONS

A. CONCLUSION

Both the Windows/C++ and Bamboo agent-based simulation architectures appear to be appropriate for building an enterprise model of the Navy. Input from subject area experts will allow the proper agent functionality and inter-agent relationships to be accurately defined. Agents with the ability to learn and adapt in their pursuit of goals will provide a robust simulation that allows leaders to view the potential outcomes of their decisions through emergent behaviors.

While we have explored the issues of developing two types of agent-based simulation architectures, building an enterprise model of the Navy at such a low-level is probably not appropriate. It appears that the best approach to take when building SimNavy would be to create a modeling engine that contains the needed functionality to define specific agents through an easy-to-use interface. This would allow modelers to focus on developing accurate models of desired agents without having to concern themselves with code and implementation issues. It could be developed using many of the same ideas from the architectures we developed. Current students in the Naval Postgraduate School's Modeling, Virtual Environments, and Simulation Curriculum plan further research in this area.

B. FUTURE WORK

The following section lists future projects that could assist in further exploring the issues associated with using the agent-based simulation methodology to build an enterprise model of the Navy.

1. SimNavy Agents

When developing an enterprise model of the Navy, one of the first issues that needs to be addressed is to identify what components are required to be modeled. Once these components have been identified, the level to which they should be modeled, either as individual entities or aggregated systems, needs to be studied. Close coordination with all Navy agencies will help with the development of the logic and functionality of these various Navy agents. This in and of itself will be a very challenging task since it appears that most of the information currently available is stove piped, with very little cross talk between agencies.

2. Learning and Adaptation

It is very difficult to establish a generic solution for dynamic agent learning and adaptation. We explored two methods for providing agents with this capability. A very simple Boolean flag method was used to indicate knowledge of predetermined relationships. The status of the flag provided access to different functionality and triggered new behaviors, but was very limited. In the Bamboo implementation a more robust structure using dynamic arrays was implemented to assist in learning. Both seem to accomplish the goal, but is there a more efficient or dynamic way to do so? Is there a way to generalize learning even more? How can this learning be tied better to adaptation? Future work could entail a more detailed exploration of methods to provide agents with a robust learning ability that allows them to adapt their behaviors.

3. Networked Applications

The Savannah and Savannah 3D architectures were built to run on stand-alone computers. The Windows/C++ implementation is not directly portable to distributed applications. The Bamboo toolkit, however, is designed for networked virtual environments, and provides an outstanding platform to build networked agent-based simulations. This area is wide open for research. Probably one of the first and most critical areas that should be studied is how and in what format does agent data need to be passed across the network so as not to lose any of the functionality of an agent-based model?

4. SimNavy Engine

Savannah 3D is an attempt to generalize agent-based modeling enough so that it can be easily modified to execute many different variations of a simulation. To build a fully functional enterprise model of the Navy is going to require a generalized, yet very robust architecture. Research in this area is needed to determine what other functionality can be added to or implemented with Bamboo to begin building a SimNavy engine. Such an engine would provide a simple GUI that could be used to study the numerous dynamic relationships that exist throughout the Navy's structure.

APPENDIX A: IMPLEMENTATION CODE LISTINGS

Table of Contents for the Code Listings

WINDOWS/C++ IMPLEMENTATION

```
//*******************************************************************
//    EXECUTIVE SUMMARY
//File Name:    Animal.h
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  Package contains definition of Animal class and its
//              member functions to work in a larger simulation
//
//March 1999, Master Thesis
//*******************************************************************

#ifndef __ANIMAL_H__
#define __ANIMAL_H__

#include "stdafx.h"
#include <math.h>

struct Pregnancy{
   int partnerId;
   int maleSpeed;
   int gestationTime;
   int seasonCounter;
};

class Animal {

private:

   MOVE_SPEED   speedOfNextMove;

   DEATH_INDICATOR deathIndicator;

   char gender;

   int  maxSpeed,
        age,
        generation,
        location,
        moveToLocation,
        moveFromLocation,
        deathCounter,
        energyLevel;

   bool pregnant,
        inSeason,
        resting;


public:

   //Default Constructor
   Animal();

   //Newborn Initialization Constructor
   Animal (int s, int gn, int l);

   //Default Destructor - does nothing at this time
   ~Animal();

   //move the animal
   void move();
   void avoidCollision();
   void moveTo(int l);
   void moveFrom(int l);

   //test if two animals are within the provided range of each other
   bool inRange(Animal &secondAnimal, int testRange);

   //generate a random number
   double myRand ();

   //get and set the max speed of animal
   int getMaxSpeed();
   void setMaxSpeed(int s);

   //get gender of animal
   char getGender();
   void setGender(char g);

   //get and set energy level
   int getEnergyLevel();
   void setEnergyLevel(int el);

   //get and set xy coordinate (location) of animal
   int getLocation();
   int getX();
   int getY();
   void setLocation(int xy);
   void setLocation(int x, int y);
   void setRandomLocation();

   //get and set the choice of speed for next move
   MOVE_SPEED getSpeedOfNextMove();
   void setSpeedOfNextMove(MOVE_SPEED ms);

   //get and set reason for animals death
   DEATH_INDICATOR getDeathIndicator();
   void setDeathIndicator(DEATH_INDICATOR di);

   //get and set deathCounter
   int getDeathCounter();
   void setDeathCounter(int dc);

   //get age of animal
   int getAge();
   void setAge(int a);

   //get and set generation of animal
   int getGeneration();
   void setGeneration(int g);

   //get and set location to move to
   int getMoveToLocation();
   void setMoveToLocation(int mtl);

   //get and set location to move from
   int getMoveFromLocation();
   void setMoveFromLocation(int mfl);

   //can the animals mate
   virtual bool canMate(Animal &potentialMate);
   virtual void mate(Animal &mate){};

   //test to see if predator can kill the prey
   virtual bool canKill(Animal &prey);

   //get distance between two animals
   int getDistance(Animal &animal);
   int distanceFromFood(int l);

   //age animal one year
   void growOlder();

   //see if female is pregnant
   bool isPregnant();
   void setPregnant(bool p);

   //see if the animal is in season
   bool isInSeason();
   void setInSeason(bool is);

   //see if animal is dead
   bool isDead();

   //see if animal needs to rest
   bool isResting();
   void setRest(bool r);

   //pointer to Pregancy struct
   Pregnancy* pregPtr;

};


//*******************************************************************
// INLINE FUNCTIONS
//*******************************************************************

inline int Animal::getMaxSpeed()
{
   return maxSpeed;
}

inline void Animal::setMaxSpeed(int s)
{
   maxSpeed = s;
```

```
}

inline char Animal::getGender()
{
   return gender;
}

inline void Animal::setGender(char g)
{
   gender = g;
}

inline int Animal::getEnergyLevel()
{
   return energyLevel;
}

inline void Animal::setEnergyLevel(int el)
{
   energyLevel = el;
}

inline int Animal::getMoveToLocation()
{
   return moveToLocation;
}

inline void Animal::setMoveToLocation(int mtl)
{
   moveToLocation = mtl;
}

inline int Animal::getMoveFromLocation()
{
   return moveFromLocation;
}

inline void Animal::setMoveFromLocation(int mfl)
{
   moveFromLocation = mfl;
}

inline int Animal::getLocation()
{
   return location;
}

inline int Animal::getX()
{
   return (location % MAX_X);
}

inline int Animal::getY()
{
   return (location / MAX_X);
}
```

```
inline void Animal::setLocation(int xy)
{
   location = xy;
}

inline void Animal::setLocation(int x, int y)
{
   location = y * MAX_X + x;
}

inline void Animal::setRandomLocation()
{
   location = int(myRand() * MAX_X * MAX_Y);
}

inline MOVE_SPEED Animal::getSpeedOfNextMove()
{
   return speedOfNextMove;
}

inline void Animal::setSpeedOfNextMove(MOVE_SPEED ms)
{
   speedOfNextMove = ms;
}

inline int Animal::getAge()
{
   return age;
}

inline void Animal::setAge(int a)
{
    age = a;
}

inline int Animal::getGeneration()
{
   return generation;
}

inline void Animal::setGeneration(int g)
{
   generation = g;
}

inline bool Animal::isPregnant()
{
   return pregnant;

}

inline void Animal::setPregnant(bool p)
{
   pregnant = p;
}
```

```
inline bool Animal::isInSeason()
{
   return inSeason;
}

inline void Animal::setInSeason(bool is)
{
   inSeason = is;
}

inline bool Animal::isResting()
{
   return resting;
}

inline void Animal::setRest(bool r)
{
   resting = r;
}

inline DEATH_INDICATOR Animal::getDeathIndicator()
{
   return deathIndicator;
}

inline void Animal::setDeathIndicator(DEATH_INDICATOR di)
{
   deathIndicator = di;
}

inline int Animal::getDeathCounter()
{
   return deathCounter;
}

inline void Animal::setDeathCounter(int dc)
{
   deathCounter = dc;
}

inline bool Animal::isDead()
{
   return (deathIndicator != NOT_DEAD);
}

inline void Animal::growOlder()
{
   age++;
}

#endif

//end file animal.h
```

```
//*******************************************************************
//    EXECUTIVE SUMMARY
//File Name:    Animal.cpp
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  Package contains definition of Animal class and its
//              member functions to work in a larger simulation
//
//March 1999, Master Thesis
//*******************************************************************

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <ctime>
#include "Animal.h"

//*******************************************************************
//  DEFINES AND FILE SCOPE CONSTANTS
//*******************************************************************

//------------------------------------------------------------------
// Function:   Animal::Animal()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//------------------------------------------------------------------
Animal::Animal ()
:speedOfNextMove(REGULAR), deathIndicator(NOT_DEAD), pregPtr(NULL),
 pregnant(false), inSeason(false), generation(1), moveToLocation(0),
 moveFromLocation(0), deathCounter(0), energyLevel(800), resting(false)
{
   double genderRand = myRand();

   if (genderRand < 0.5)
      gender = MALE;
   else
      gender = FEMALE;

   //assign a random max speed for the animal between 5..10
   maxSpeed = myRand()*10;

   if (maxSpeed < 5)
      maxSpeed += 5;

   setRandomLocation();

   age = int(myRand() * ANIMAL_AGE);

}//end animal::animal()


//------------------------------------------------------------------
// Function:   Animal::Animal(int s, int x, int y)
```

```
// Return Val: None
// Parameter:  Speed, XY position
// Purpose:    Newborn initialization constructor
//-------------------------------------------------------------------
Animal::Animal(int s, int gn, int l)
: speedOfNextMove(REGULAR), deathIndicator(NOT_DEAD), maxSpeed(s),
  generation(gn), age(NEWBORN_AGE), pregnant(false), inSeason(false),
  pregPtr(NULL), location(l+1), moveToLocation(0), moveFromLocation(0),
  deathCounter(0), energyLevel(800), resting(false)
{
   double genderRand;

   genderRand = myRand();
   if (genderRand < 0.5)
   {
      gender = MALE;
   }
   else
   {
      gender = FEMALE;
   }

}//end Animal::Animal(int s, char g, int x, int y)


//-------------------------------------------------------------------
// Function:   Animal::~Animal()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//-------------------------------------------------------------------
Animal::~Animal()
{
   //do nothing at this point

}//end Animal::~Animal()


//-------------------------------------------------------------------
// Function:   Animal::move ()
// Return Val: void
// Parameter:  None
// Purpose: move animals that are not resting
//-------------------------------------------------------------------
void Animal::move()
{
   if(speedOfNextMove != REST)
   {
      int tempX = this->getX(),
          tempY = this->getY();

      double randX = myRand();
      double randY = myRand();

      if (randX <= 0.5)
         tempX--;
      else
         tempX++;

      if (randY <= 0.5)
         tempY--; // this moves the animal up one row
      else
         tempY++; // this moves the animal down one row

      if(tempX <= MIN_X)
         tempX = MIN_X + 1;//bring the animal back one unit
      if(tempX >= MAX_X)
         tempX = MAX_X-1;//bring the animal back one unit
      if(tempY <= MIN_Y)
         tempY = MIN_Y + 1; //move the animal down one row
      if(tempY >= MAX_Y)
         tempY = MAX_Y - 1; //move the animal up one row

      this->setLocation(tempX, tempY);

   }//end if not at REST

   return;

}//end Animal::move()


//-------------------------------------------------------------------
// Function:   Animal::avoidCollision ()
// Return Val: void
// Parameter:  None
// Purpose:  keep animals from occuping the same grid space
//-------------------------------------------------------------------
void Animal::avoidCollision()
{
   int tempX = this->getX(),
       tempY = this->getY();

   double randX = myRand();
   double randY = myRand();

   if (randX <= 0.5)
      tempX--;
   else
      tempX++;
   if (randY <= 0.5)
      tempY--; // this moves the animal up one row
   else
      tempY++; // this moves the animal down one row

   if(tempX <= MIN_X)
      tempX = MIN_X + 1;//bring the animal back one unit
   if(tempX >= MAX_X)
      tempX = MAX_X-1;//bring the animal back one unit
   if(tempY <= MIN_Y)
      tempY = MIN_Y + 1; //move the animal down one row
   if(tempY >= MAX_Y)
      tempY = MAX_Y - 1; //move the animal up one row

   this->setLocation(tempX, tempY);

   return;

}//end Animal::avoidCollision()


//-------------------------------------------------------------------
// Function:   Animal::moveTo ()
// Return Val: void
// Parameter:  int location
// Purpose:  returns destination for animals next move
//-------------------------------------------------------------------
void Animal::moveTo(int l)
{
   int tempX = l % MAX_X,
       tempY = l / MAX_X,
       thisTempX = this->getX(),
       thisTempY = this->getY();

   if(speedOfNextMove == REGULAR)
   {
      if((thisTempX - tempX) > 0)
         thisTempX -= UNIT_MOVEMENT;
      else if((thisTempX - tempX) < 0)
         thisTempX += UNIT_MOVEMENT;

      if((thisTempY - tempY) > 0)
         thisTempY-= UNIT_MOVEMENT;
      else if((thisTempY - tempY) < 0)
         thisTempY += UNIT_MOVEMENT;
   }//end if
   else//RUN
   {
      if((thisTempX - tempX) > 0)
         thisTempX -= this->getMaxSpeed()/2;
      else if((thisTempX - tempX) < 0)
         thisTempX += this->getMaxSpeed()/2;

      if((thisTempY - tempY) > 0)
         thisTempY-= this->getMaxSpeed()/2;
      else if((thisTempY - tempY) < 0)
         thisTempY += this->getMaxSpeed()/2;
   }//end else
   if(thisTempX <= MIN_X)
      thisTempX = MIN_X + 1;//bring the animal back one unit
   if(thisTempX >= MAX_X)
      thisTempX = MAX_X-1;//bring the animal back one unit
   if(thisTempY <= MIN_Y)
      thisTempY = MIN_Y + 1; //move the animal down one row
   if(thisTempY >= MAX_Y)
      thisTempY = MAX_Y - 1; //move the animal up one row
   this->setLocation(thisTempX, thisTempY);

   return;

}//end Animal::moveTo()


//-------------------------------------------------------------------
// Function:   Animal::moveFrom()
// Return Val: void
// Parameter:  int location
// Purpose:  returns destination for animals next move
//-------------------------------------------------------------------
void Animal::moveFrom(int l)
{
   int tempX = l % MAX_X,
       tempY = l / MAX_X,
       thisTempX = this->getX(),
       thisTempY = this->getY();

   if(speedOfNextMove == REGULAR)
   {
      if((thisTempX - tempX) > 0)
         thisTempX += UNIT_MOVEMENT;
      else if((thisTempX - tempX) < 0)
         thisTempX -= UNIT_MOVEMENT;
      if((thisTempY - tempY) > 0)
         thisTempY += UNIT_MOVEMENT;
      else if((thisTempY - tempY) < 0)
         thisTempY -= UNIT_MOVEMENT;
   }//end if
   else//RUN
   {
      if((thisTempX - tempX) > 0)
         thisTempX += this->getMaxSpeed()/2;
      else if((thisTempX - tempX) < 0)
         thisTempX -= this->getMaxSpeed()/2;
      if((thisTempY - tempY) > 0)
         thisTempY += this->getMaxSpeed()/2;
      else if((thisTempY - tempY) < 0)
         thisTempY -= this->getMaxSpeed()/2;
   }//end else
   if(thisTempX <= MIN_X)
      thisTempX = MIN_X + 1;//bring the animal back one unit
   if(thisTempX >= MAX_X)
      thisTempX = MAX_X-1;//bring the animal back one unit
   if(thisTempY <= MIN_Y)
      thisTempY = MIN_Y + 1; //move the animal down one row
   if(thisTempY >= MAX_Y)
      thisTempY = MAX_Y - 1; //move the animal up one row

   this->setLocation(thisTempX, thisTempY);

   return;

}//end Animal::moveFrom()
```

```
//-------------------------------------------------------------------
// Function:   Animal::myRand ()
// Return Val: double - a pseudorandom number between 0.0 and 1.0
// Parameter:  none
// Purpose:    return random number
//-------------------------------------------------------------------
double Animal::myRand ()
{
   double randomNumber;

   randomNumber = rand()/double(RAND_MAX);

   return randomNumber;

}//end Animal::myRand()


//--------------------------------------------------------------------
//Function:   Animal::canMate ()
//Return Val: true / false
//Parameter:  potentialMate
//Purpose:    evaluate whether two like animals can mate
//--------------------------------------------------------------------
bool Animal::canMate(Animal &potentialMate)
{
   bool mateFlag = false;

   if(this->getGender() == MALE)
   {
      mateFlag = ((!(potentialMate.isPregnant())) &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE) &&
                  (abs(this->getX() - potentialMate.getX()) <=
                     MATE_DISTANCE) &&
                  (abs(this->getY() - potentialMate.getY()) <=
                     MATE_DISTANCE));
   }
   else
   {
      mateFlag = ((!(this->isPregnant())) &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE) &&
                  (abs(this->getX() - potentialMate.getX()) <=
                     MATE_DISTANCE) &&
                  (abs(this->getY() - potentialMate.getY()) <=
                     MATE_DISTANCE));
   }

   return mateFlag;

}//end function Animal::canMate(Animal &potentialMate)


//--------------------------------------------------------------------
//Function:  Animal::canKill ()
//Return Val:
//Parameter:
//Purpose:  evaluate whether two like animals can mate
//--------------------------------------------------------------------
bool Animal::canKill(Animal &prey)
{
   return false;

}//end bool Animal::canKill(Animal &prey)


//--------------------------------------------------------------------
//Function:  Animal::getDistance()
//Return Val:  int distance between two animals
//Parameter:  animal
//Purpose:  determine distance between two animals
//--------------------------------------------------------------------
int Animal::getDistance(Animal &animal)
{
   int xSquare, ySquare, answer;
   xSquare = (this->getX() - animal.getX()) * (this->getX() -
      animal.getX());
   ySquare = (this->getY() - animal.getY()) * (this->getY() -
      animal.getY());
   answer = (sqrt(xSquare + ySquare));

   return answer;

}//end Animal::getDistance()


//--------------------------------------------------------------------
//Function:  Animal::distanceFromFood()
//Return Val:  int between animal and food
//Parameter:  int location
//Purpose:  determine distance between animal and food
//--------------------------------------------------------------------
int Animal::distanceFromFood(int l)
{
   int tempX = l % MAX_X,
       tempY = l / MAX_X,
       xSquare,
       ySquare,
       answer;

   xSquare = (this->getX() - tempX) * (this->getX() - tempX);
   ySquare = (this->getY() - tempY) * (this->getY() - tempY);
   answer = (sqrt(xSquare + ySquare));

   return answer;
}//end Animal::distanceFromFood()

//end file Animal.cpp


//*********************************************************************
//    EXECUTIVE SUMMARY
```

```
//File Name:    Antelope.h
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  Package contains definition of Antelope class and its
//              member functions to work in a larger simulation
//
//March 1999, Master Thesis
//*********************************************************************

#ifndef __ANTELOPE_H__
#define __ANTELOPE_H__

#include "animal.h"

class Antelope: public Animal{

private:

   ANTELOPE_DESIRED_ACTION nextAction;

   int idNum,
       herdSize;

   bool predatorKnowledge;

public:

   //Default Constructor
   Antelope();

   //Newborn Initialization Constructor
   Antelope (int s, int gn, int l);

   //Default Destructor - does nothing at this time
   ~Antelope();

   //produce a newborn Antelope from a male/female pair
   Antelope* Antelope::giveBirth(int speedOne, int speedTwo,
      int motherGeneration, int motherLocation);

   //get antelope identification number
   int getIdNum();

   //set and get herd size
   int getHerdSize();
   void setHerdSize(int hs);

   //get and set the desired next action for the antelope
   ANTELOPE_DESIRED_ACTION getNextAction();
   void setNextAction (ANTELOPE_DESIRED_ACTION na);

   //return antelope litter size
   int litterSize();

   //return true if Antelope dies as infant
   bool diesAsInfant();

   //can the Antelope mate
   bool canMate(Antelope &potentialMate);
   void mate(Antelope &mate);

   //are the Antelope mate eligible
   bool mateEligible(Antelope &potentialMate);

   //does a Antelope know Cheetahs are bad?
   bool getPredatorKnowledge();
   void setPredatorKnowledge(bool pk);

   //print Antelope info
   void printAntelopeInfo();

};


//*********************************************************************
// INLINE FUNCTIONS
//*********************************************************************

inline int Antelope::getIdNum()
{
   return idNum;
}

inline ANTELOPE_DESIRED_ACTION Antelope::getNextAction()
{
   return nextAction;
}

inline void Antelope::setNextAction (ANTELOPE_DESIRED_ACTION na)
{
   nextAction = na;
}

inline bool Antelope::getPredatorKnowledge()
{
   return predatorKnowledge;
}

inline void Antelope::setPredatorKnowledge(bool pk)
{
   predatorKnowledge = pk;
}

inline int Antelope::getHerdSize()
{
   return herdSize;
}

inline void Antelope::setHerdSize(int hs)
{
```

```
   herdSize = hs;
}

#endif

//end file Antelope.h


//*******************************************************************
//    EXECUTIVE SUMMARY
//File Name:    Antelope.cpp
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  Package contains definition of Antelope class and its
//              member functions to work in a larger simulation
//
//March 1999, Master Thesis
//*******************************************************************

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <ctime>
#include "Antelope.h"

//*******************************************************************
// DEFINES AND FILE SCOPE CONSTANTS
//*******************************************************************

static int numAntelope = 0;

//-------------------------------------------------------------------
// Function:   Antelope::Antelope()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//-------------------------------------------------------------------
Antelope::Antelope ()
   :Animal(), idNum(numAntelope++), nextAction(A_NOTHING),
    herdSize(1)
{
   if(Animal::myRand() < PREDATOR_KNOWLEDGE)
      predatorKnowledge = true;
   else
      predatorKnowledge = false;

}//end Antelope::Antelope()


//-------------------------------------------------------------------
// Function:   Antelope::Antelope(int s, int gn, int x, int y)
// Return Val: None
// Parameter:  Speed, generation, and XY position
// Purpose:    Initialization constructor for newborn Antelopes
```

```
//-------------------------------------------------------------------
Antelope::Antelope(int s, int gn, int l)
: Animal(s, gn, l), idNum(numAntelope++), nextAction(A_NOTHING),
  herdSize(1)
{
   if(Animal::myRand() < PREDATOR_KNOWLEDGE)
      predatorKnowledge = true;
   else
      predatorKnowledge = false;

}//end Antelope::Antelope(int s, int x, int y)


//-------------------------------------------------------------------
// Function:   Antelope::~Antelope()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//-------------------------------------------------------------------
Antelope::~Antelope ()
{
   //do nothing at this point
}//end Antelope::~Antelope()


//-------------------------------------------------------------------
//Function:   Antelope::giveBirth ()
//Return Val: Antelope
//Parameter:  male speed, female speed
//Purpose:    make a new Antelope
//-------------------------------------------------------------------
Antelope* Antelope::giveBirth(int speedOne, int speedTwo,
                              int motherGeneration, int motherLocation)
{
   int newSpeed,
       nextGeneration = (motherGeneration + 1);

   Antelope *newBorn;

   if(Animal::myRand() < .5)
      newSpeed = speedOne;
   else
      newSpeed = speedTwo;

   newBorn = new Antelope(newSpeed, nextGeneration, motherLocation+1);

   #ifdef SPEED_COUT
      cout << "ANTELOPE" << "     " << "speedOne:  " << speedOne << "     "
           << "speedTwo:  " << speedTwo << "     " << "newborn speed   "
           << newSpeed << endl;
   #endif

   return newBorn;

}//end animal::mate()
```

```
//-------------------------------------------------------------------
//Function:   Antelope::mate ()
//Return Val: true / false
//Parameter:  potentialMate
//Purpose:    evaluate whether two like animals can mate
//-------------------------------------------------------------------
void Antelope::mate(Antelope &mate)
{
   if(this->getGender() == MALE)
   {
      mate.setPregnant(true);
      mate.pregPtr->partnerId = this->getIdNum();
      mate.pregPtr->maleSpeed = this->getMaxSpeed();
      mate.pregPtr->gestationTime = 0;
   }
   else
   {
      this->setPregnant(true);
      this->pregPtr->partnerId = mate.getIdNum();
      this->pregPtr->maleSpeed = mate.getMaxSpeed();
      this->pregPtr->gestationTime = 0;
   }

   return;

}//end function Antelope::mate()


//-------------------------------------------------------------------
// Function: void Antelope::printAntelopeInfo()
// Return Val:  void
// Parameter:  none
// Purpose:    Print Antelope information
//-------------------------------------------------------------------
void Antelope::printAntelopeInfo()
{
   if(getGender() == 'M')
   {
      cout <<"  Male Antelope " << idNum;
   }
   else
   {
      cout << "Female Antelope " << idNum;
   }

   cout <<" => spd = " << getMaxSpeed() .
       <<" gndr = " << getGender()
       <<" age = "   << getAge()
       <<" x = " << getX()
       <<" y = " << getY()
       << endl;
   return;

}//end Antelope Antelope::printAntelopeInfo()
```

```
//-------------------------------------------------------------------
// Function: int Antelope::litterSize()
// Return Val:  int number in litter
// Parameter:  none
// Purpose:    return a random number of antelope in a litter
//-------------------------------------------------------------------

int Antelope::litterSize()
{

   int litter = 1;

   if(Animal::myRand() >= 0.9)
      litter = 2;

   return litter;

}//end Antelope::litterSize()


//-------------------------------------------------------------------
// Function:  bool Antelope::diesAsInfant()
// Return Val:  true for dies; false for lives
// Parameter:  none
// Purpose:  return whether infant dies or not
//-------------------------------------------------------------------
bool Antelope::diesAsInfant()
{
   double randNum = Animal::myRand();

   return (randNum < ANTELOPE_MORTALITY_RATE);

}//end Antelope::mortality()


//-------------------------------------------------------------------
// Function:   bool Antelope::canMate()
// Return Val:  true for yes; false for no
// Parameter:  potential mate
// Purpose:    return whether Antelope can mate or not
//-------------------------------------------------------------------
bool Antelope::canMate(Antelope &potentialMate)
{
   bool mateFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate.getGender() == FEMALE))
   {
      mateFlag = ((!(potentialMate.isPregnant())) &&
                  (this->getNextAction() == A_MATE) &&
                  (potentialMate.getNextAction() == A_MATE)  &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE) &&
                  (abs(this->getX() - potentialMate.getX()) <=
                     MATE_DISTANCE) &&
```

```
               (abs(this->getY() - potentialMate.getY()) <=
                  MATE_DISTANCE));
   }
   else if((this->getGender() == FEMALE) &&
      (potentialMate.getGender() == MALE))
   {
      mateFlag = ((!(this->isPregnant())) &&
                 (this->getNextAction() == A_MATE) &&
                 (potentialMate.getNextAction() == A_MATE)  &&
                 (potentialMate.getAge() >= MATE_AGE) &&
                 (this->getAge() >= MATE_AGE) &&
                 (abs(this->getX() - potentialMate.getX()) <=
                    MATE_DISTANCE) &&
                 (abs(this->getY() - potentialMate.getY()) <=
                    MATE_DISTANCE));
   }

   return mateFlag;
}//Antelope::canMate(Antelope &potentialMate)


//--------------------------------------------------------------------
// Function:   bool Antelope::mateEligible()
// Return Val:  potential mate
// Parameter:  true for yes; false for no
// Purpose:    return whether Antelope is eligible to mate
//--------------------------------------------------------------------
bool Antelope::mateEligible(Antelope &potentialMate)
{
   bool mateEligibleFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate.getGender() == FEMALE))
   {
      mateEligibleFlag = (!(potentialMate.isPregnant()) &&
                 (potentialMate.isInSeason()) &&
                 (potentialMate.getAge() >= MATE_AGE) &&
                 (this->getAge() >= MATE_AGE));
   }
   else if((this->getGender() == FEMALE) &&
      (potentialMate.getGender() == MALE))
   {
      mateEligibleFlag = (!(this->isPregnant()) &&
                 (this->isInSeason()) &&
                 (potentialMate.getAge() >= MATE_AGE) &&
                 (this->getAge() >= MATE_AGE));
   }

   return mateEligibleFlag;

}//end Antelope::mateEligible

//end file Antelope.cpp

//*******************************************************************
```

```
//    EXECUTIVE SUMMARY
//File Name:    Cheetah.h
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  Package contains definition of Cheetah class and its
//              member functions to work in a larger simulation
//
//March 1999, Master Thesis
//*******************************************************************

#ifndef __CHEETAH_H__
#define __CHEETAH_H__

#include "animal.h"

class Cheetah: public Animal{

private:

   CHEETAH_DESIRED_ACTION nextAction;
   int idNum;

public:

   //Default Constructor
   Cheetah();

   //Newborn initialization constructor
   Cheetah (int s, int gg, int l);

   //Default Destructor - does nothing at this time
   ~Cheetah();

   //produce a newborn Cheetah from a male/female pair
   Cheetah* Cheetah::giveBirth(int speedOne, int speedTwo,
      int motherGeneration, int motherLocation);

   //get Cheetah identification number
   int getIdNum();

   //return Cheetah litter size
   int litterSize();

   //get and set the desired next action for the antelope
   CHEETAH_DESIRED_ACTION getNextAction();
   void setNextAction (CHEETAH_DESIRED_ACTION na);

   //return whether or not the Cheetah dies in infancy
   bool diesAsInfant();

   //check to see if cheetah can kill Antelope
   bool canKill(Animal &prey);

   //can the Cheetah mate
   bool canMate(Cheetah &potentialMate);
   void mate(Cheetah &mate);

   //are the Cheetahs mate eligible
   bool mateEligible(Cheetah &potentialMate);

   //print Cheetah info
   void printCheetahInfo();

};


//*******************************************************************
// INLINE FUNCTIONS
//*******************************************************************

inline int Cheetah::getIdNum()
{
   return idNum;
}

inline CHEETAH_DESIRED_ACTION Cheetah::getNextAction()
{
   return nextAction;
}

inline void Cheetah::setNextAction (CHEETAH_DESIRED_ACTION na)
{
   nextAction = na;
}

#endif

//end file Cheetah.h


//*******************************************************************
//    EXECUTIVE SUMMARY
//File Name:    Cheetah.cpp
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  Package contains definition of Cheetah class and its
//              member functions to work in a larger simulation
//
//March 1999, Master Thesis
//*******************************************************************

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <ctime>
#include "Cheetah.h"

//*******************************************************************
```

```
//  DEFINES AND FILE SCOPE CONSTANTS
//*******************************************************************

static int numCheetah = 0;

//--------------------------------------------------------------------
// Function:   Cheetah::Cheetah()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//--------------------------------------------------------------------
Cheetah::Cheetah ()
   :Animal(), idNum(numCheetah++), nextAction(C_NOTHING)
{

   int speed = (rand()/double(RAND_MAX))*10;
   if (speed < 5)
   {
      speed += 5;
   }

   this->setMaxSpeed(speed + CHEETAH_SPEED_ADVANTAGE);

}//end Cheetah::Cheetah()


//--------------------------------------------------------------------
// Function:   Cheetah::Cheetah(int s, int gn, int l)
// Return Val: None
// Parameter:  Speed, generation, and XY position
// Purpose:    Initialization constructor for newborn Cheetahs
//--------------------------------------------------------------------
Cheetah::Cheetah(int s, int gn, int l)
: Animal (s, gn, l), idNum(numCheetah++), nextAction(C_NOTHING)
{
}//end Cheetah::Cheetah(int s, int gn, int l)


//--------------------------------------------------------------------
// Function:   Cheetah::~Cheetah()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//--------------------------------------------------------------------
Cheetah::~Cheetah ()
{
   //do nothing at this point

}//end Cheetah::~Cheetah()


//--------------------------------------------------------------------
//Function:   Cheetah::mate ()
//Return Val: Cheetah
//Parameter:  speed of two cheetah, female generation and location
//Purpose:  create new cheetah
```

```
//-----------------------------------------------------------------------
Cheetah* Cheetah::giveBirth(int speedOne, int speedTwo,
                            int motherGeneration, int motherLocation)
{
   int newSpeed,
       nextGeneration = (motherGeneration + 1);

   Cheetah *newBorn;

   if(Animal::myRand() < .5)
      newSpeed = speedOne;
   else
      newSpeed = speedTwo;

   newBorn = new Cheetah(newSpeed, nextGeneration, motherLocation + 1);

   return newBorn;

}//end animal::mate()


//-----------------------------------------------------------------------
//Function:   Cheetah::mate ()
//Return Val: none
//Parameter:  mate
//Purpose:  mate the cheetah
//-----------------------------------------------------------------------
void Cheetah::mate(Cheetah &mate)
{
   if(this->getGender() == MALE)
   {
      mate.setPregnant(true);
      mate.pregPtr->partnerId = this->getIdNum();
      mate.pregPtr->maleSpeed = this->getMaxSpeed();
      mate.pregPtr->gestationTime = 0;
   }
   else
   {
      this->setPregnant(true);
      this->pregPtr->partnerId = mate.getIdNum();
      this->pregPtr->maleSpeed = mate.getMaxSpeed();
      this->pregPtr->gestationTime = 0;
   }

   return;

}//end function Cheetah::mate()


//-----------------------------------------------------------------------
// Function:  void Cheetah::printCheetahInfo()
// Return Val:  void
// Parameter:  none
// Purpose:    Print Cheetah information
//-----------------------------------------------------------------------
void Cheetah::printCheetahInfo()
{
   if(getGender() == MALE)
   {
     cout <<"  Male Cheetah  " << idNum;
   }
   else
   {
     cout << "Female Cheetah  " << idNum;
   }

   cout <<" => spd = " << getMaxSpeed()
            <<" gndr = " << getGender()
            <<" age = "  << getAge()
            <<" x = " << getX()
            <<" y = " << getY()
            <<endl;
   return;

}//end void Cheetah::printCheetahInfo()


//-----------------------------------------------------------------------
// Function: int Cheetach::litterSize()
// Return Val:  int number in litter
// Parameter:  none
// Purpose:    return a random number of Cheetah in a litter
//-----------------------------------------------------------------------
int Cheetah::litterSize()
{

   int litter = 1;

   double randNum = Animal::myRand();

   if(randNum <= 0.05)
      litter = 1;
   else if((randNum > 0.05) && (randNum <= 0.15))
      litter = 2;
   else if((randNum > 0.15) && (randNum <= 0.3))
      litter = 3;
   else if((randNum > 0.3) && (randNum <= 0.7))
      litter = 4;
   else if((randNum > 0.7) && (randNum <= 0.85))
      litter = 5;
   else if((randNum > 0.85) && (randNum <= 0.95))
      litter = 6;
   else //if randNum > .95
      litter = 7;

   return litter;

}//end Antelope::litterSize()


//-----------------------------------------------------------------------
// Function: bool Cheetach::diesAsInfant()
// Return Val:  number of cheetah that die
// Parameter:  none
// Purpose:    return whether infant dies or not
//-----------------------------------------------------------------------
bool Cheetah::diesAsInfant()
{
   double randNum = Animal::myRand();

   return (randNum < CHEETAH_MORTALITY_RATE);

}//end Cheetah::mortality()


//-----------------------------------------------------------------------
//Function:  Cheetah::canKill ()
//Return Val:  true if yes; false if no
//Parameter:  potential prey
//Purpose:  evaluate whether two like animals can kill
//-----------------------------------------------------------------------
bool Cheetah::canKill(Animal &prey)
{
   bool killFlag = false;

   if(//(this->getMaxSpeed() > prey.getMaxSpeed()) &&
            (abs(this->getX() - prey.getX()) <= KILL_RADIUS) &&
            (abs(this->getY() - prey.getY()) <= KILL_RADIUS))
      if(Animal::myRand() > .5)
         killFlag = true;
      else
         killFlag = false;

   return killFlag;

}//Cheetah::canKill(Animal &prey)


//-----------------------------------------------------------------------
// Function:   bool Cheetah::canMate()
// Return Val:  true if yes; false if no
// Parameter:  potential mate
// Purpose:    return whether Cheetah can mate or not
//-----------------------------------------------------------------------
bool Cheetah::canMate(Cheetah &potentialMate)
{
   bool mateFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate.getGender() == FEMALE))
   {
      mateFlag = ((!(potentialMate.isPregnant())) &&
                  (this->getNextAction() == C_MATE) &&
                  (potentialMate.getNextAction() == C_MATE)  &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE) &&
                  (abs(this->getX() - potentialMate.getX()) <=
                     MATE_DISTANCE) &&
                  (abs(this->getY() - potentialMate.getY()) <=
                     MATE_DISTANCE));
   }
   else if((this->getGender() == FEMALE) &&
      (potentialMate.getGender() == MALE))
   {
      mateFlag = ((!(this->isPregnant())) &&
                  (this->getNextAction() == C_MATE) &&
                  (potentialMate.getNextAction() == C_MATE)  &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE) &&
                  (abs(this->getX() - potentialMate.getX()) <=
                     MATE_DISTANCE) &&
                  (abs(this->getY() - potentialMate.getY()) <=
                     MATE_DISTANCE));
   }

   return mateFlag;
}//Cheetah::canMate(Cheetah &potentialMate)


//-----------------------------------------------------------------------
// Function:   bool Cheetah::mateEligible()
// Return Val:  true if yes; false if no
// Parameter:  potential mate
// Purpose:    return whether Cheetah is eligible to mate
//-----------------------------------------------------------------------
bool Cheetah::mateEligible(Cheetah &potentialMate)
{
   bool mateEligibleFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate.getGender() == FEMALE))
   {
      mateEligibleFlag = ((!(potentialMate.isPregnant())) &&
                          (potentialMate.isInSeason()) &&
                          (potentialMate.getAge() >= MATE_AGE) &&
                          (this->getAge() >= MATE_AGE));
   }
   else if((this->getGender() == FEMALE) &&
      (potentialMate.getGender() == MALE))
   {
      mateEligibleFlag = (!(this->isPregnant()) &&
                  (this->isInSeason()) &&
                  (potentialMate.getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE));
   }

   return mateEligibleFlag;
}//Cheetah::mateEligible(Cheetah &potentialMate)

//end file Cheetah.cpp

//*********************************************************************
//    EXECUTIVE SUMMARY
```

```
//File Name:    StdAfx.h
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  Package contains standard Windows MFC settings and
//              simulation globals.  include file for standard system
//              include files, or project specific include files that are
//              used frequently, but are changed infrequently
//
//March 1999, Master Thesis
//*******************************************************************

#if!defined(AFX_STDAFX_H__99A28497_8631_11D2_889B_0000F8092715__INCLUDED_)
#define AFX_STDAFX_H__99A28497_8631_11D2_889B_0000F8092715__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define VC_EXTRALEAN    // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC OLE automation classes
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT

//*******************************************************************
//  Global Constants used to initialize and/or scale the simulation
//*******************************************************************

enum MOVE_SPEED {REST, REGULAR, RUN};
enum DEATH_INDICATOR {INFANT_MORTALITY, OLD_AGE, PREDATOR, STARVATION,
      NOT_DEAD};
enum ANTELOPE_DESIRED_ACTION {A_NOTHING, A_MATE, HERD, FLEE, FEED};
enum CHEETAH_DESIRED_ACTION {C_NOTHING, C_MATE, AVOID, CHASE};

#define HIGH_NUM

#ifdef HIGH_NUM
#define CHEETAH_KILLS_COUT
#define SPEED_COUT
#define MAX_TIME                        10000  //maximum time steps
#define NUM_ANTELOPE                      100  //# antelope to create
#define NUM_CHEETAH                         5  //# cheetah to create
#define MIN_X                               0
//X coordinates range from 0 to MAX_X
#define MAX_X                             640
//Y coordinates range from 0 to MAX_Y#define MIN_Y

#define MAX_Y                             480
#define ANIMAL_AGE                       1825
#define MAX_ANTELOPE_AGE                 2000
#define MAX_CHEETAH_AGE                  3650  //8 - 12 years - 10 years
```

```
#define MALE                              'M'
#define FEMALE                            'F'
#define KILL_RADIUS                         5
#define MATE_DISTANCE                       5
#define MATE_AGE                          660  //20-23 months->22 months
#define NEWBORN_AGE                         1  //1 day old
#define CHEETAH_SPEED_ADVANTAGE             1
#define ANTELOPE_GESTATION_PERIOD         171  //171 day gestation period
#define CHEETAH_GESTATION_PERIOD           95  //95 day gestation period
#define ANTELOPE_WAIT_TO_MATE_TIME        365  //wait for a litter to lve
#define CHEETAH_WAIT_TO_MATE_TIME         700  //wait for a litter to lve
#define CHEETAH_INFANT_AGE                700  //days until mom moves out
#define ANTELOPE_INFANT_AGE               365  //days until mom moves out
#define CHEETAH_MORTALITY_RATE           0.90//90% die in first 2 years
#define ANTELOPE_MORTALITY_RATE          0.3//30% die in first 2 years
#define STOP_SPEED                          0  //stop speed of any animal
#define CHEETAH_CRUISE_SPEED                3  //medium speed of Cheetah
#define ANTELOPE_CRUISE_SPEED               3  //medium speed of Antelope
#define ANTELOPE_REST_SENSING_RANGE        50
#define ANTELOPE_REGULAR_SENSING_RANGE     30
#define ANTELOPE_RUN_SENSING_RANGE         15
#define CHEETAH_REST_SENSING_RANGE        175
#define CHEETAH_REGULAR_SENSING_RANGE     175
#define CHEETAH_RUN_SENSING_RANGE         100
#define CHEETAH_AVOID_RANGE               150
#define CHEETAH_STARVATION_LEVEL            2
#define PREDATOR_KNOWLEDGE                0.5  //% know predator
#define UNIT_MOVEMENT                       1  //how many pixels to move
#define FAST_SIMULATION_SPEED              50
#define MEDIUM_SIMULATION_SPEED           250
#define SLOW_SIMULATION_SPEED             500
#define FRIEND_STANDOFF_DISTANCE           15
#define CHEETAH_ENERGY_BOOST              400
#define CHEETAH_HIGH_ENERGY_LEVEL         800
#define CHEETAH_STOP_HUNTING_LEVEL        200
#define CHEETAH_RESUME_HUNTING_LEVEL      600
#define CHEETAH_LOW_ENERGY_LEVEL            0
#define CHEETAH_REST_ENERGY_GAIN            4
#define CHEETAH_REGULAR_ENERGY_PENALTY      4
#define CHEETAH_RUN_ENERGY_PENALTY         10
#define ANTELOPE_START_IN_SEASON           15
#define ANTELOPE_STOP_IN_SEASON            60
#define CHEETAH_START_IN_SEASON            30
#define CHEETAH_STOP_IN_SEASON            140
#define FOOD_RANGE                         25
const int FOOD_LOCATION[]={44870, 45370, 109020, 109240, 153670, 154170,
   198620, 198840, 262470, 262970};

#endif

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations
// immediately before the previous line.

#endif
```

```
//*******************************************************************
//    EXECUTIVE SUMMARY
//File Name:    AgentGUIView.h
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  interface of the CAgentGUIView class
//
//March 1999, Master Thesis
//*******************************************************************

#if!defined(AFX_AGENTGUIVIEW_H__99A2849D_8631_11D2_889B_0000F8092715
   __INCLUDED_)
#define AFX_AGENTGUIVIEW_H__99A2849D_8631_11D2_889B_0000F8092715
   __INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif

class CAgentGUIView : public CView
{
protected: // create from serialization only
   CAgentGUIView();
   DECLARE_DYNCREATE(CAgentGUIView)

// Attributes
public:
   CAgentGUIDoc* GetDocument();

// Operations
public:

public:
   virtual void OnDraw(CDC* pDC);  // overridden to draw this view
   virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
   protected:
   virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
   virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
   virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

// Implementation
public:
   virtual ~CAgentGUIView();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:

   int loopSpeed,
      simulationTime;
```

```
   CPen pPenMaleAntelope1,
        pPenMaleAntelope2,
        pPenMaleAntelope3,
        pPenMaleAntelope4,
        pPenMaleAntelope5,
        pPenMaleCheetah,
        pPenFemaleCheetah,
        pPenFood;

   CBrush brushMaleAntelope1,
          brushMaleAntelope2,
          brushMaleAntelope3,
          brushMaleAntelope4,
          brushMaleAntelope5,
          brushMaleCheetah,
          brushFemaleCheetah;

   bool simulationOn,
        statisticsOn;

   void printAntelopeStatistics (CDC *pDC);
   void printCheetahStatistics (CDC *pDC);

   void updateStatusBar(int numMaleAntelope, int numFemaleAntelope,
                int antelopeGenerations,  int numMaleCheetah,
                int numFemaleCheetah, int cheetahGenerations,
                int simTime);
   void integerToString(int num, CString &numbers);

// Generated message map functions
protected:
   afx_msg void OnTimer(UINT nIDEvent);
   afx_msg void OnRunSimulation();
   afx_msg void OnStopSimulation();
   afx_msg void OnStepSimulation();
   afx_msg void OnToggle();
   afx_msg void OnSpeedFast();
   afx_msg void OnSpeedMedium();
   afx_msg void OnSpeedSlow();
   afx_msg void OnUpdateRunSimulation(CCmdUI* pCmdUI);
   afx_msg void OnUpdateStopSimulation(CCmdUI* pCmdUI);
   afx_msg void OnUpdateSpeedFast(CCmdUI* pCmdUI);
   afx_msg void OnUpdateSpeedMedium(CCmdUI* pCmdUI);
   afx_msg void OnUpdateSpeedSlow(CCmdUI* pCmdUI);
   afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
   afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
   DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in AgentGUIView.cpp
inline CAgentGUIDoc* CAgentGUIView::GetDocument()
   { return (CAgentGUIDoc*)m_pDocument; }
#endif

#endif
```

```
//*******************************************************************
//     EXECUTIVE SUMMARY
//File Name:     AgentGUIView.cpp
//
//Authors:       Mark A. Boyd; maboyd@bigfoot.com
//               Todd A. Gagnon; todd@gagnon.com
//
//Description:  implementation of the CAgentGUIView class
//
//March 1999, Master Thesis
//*******************************************************************

#include "stdafx.h"
#include "AgentGUI.h"
#include "MainFrm.h"

#include "AgentGUIDoc.h"
#include "AgentGUIView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

static int numAntelope = 0;
static int numCheetah  = 0;

IMPLEMENT_DYNCREATE(CAgentGUIView, CView)

BEGIN_MESSAGE_MAP(CAgentGUIView, CView)
   ON_WM_TIMER()
   ON_COMMAND(ID_RUN_SIMULATION, OnRunSimulation)
   ON_COMMAND(ID_STOP_SIMULATION, OnStopSimulation)
   ON_COMMAND(ID_STEP_SIMULATION, OnStepSimulation)
   ON_COMMAND(SIMULATION_TOGGLE, OnToggle)
   ON_COMMAND(SET_SPEED_FAST, OnSpeedFast)
   ON_COMMAND(SET_SPEED_MEDIUM, OnSpeedMedium)
   ON_COMMAND(SET_SPEED_SLOW, OnSpeedSlow)
   ON_UPDATE_COMMAND_UI(ID_RUN_SIMULATION, OnUpdateRunSimulation)
   ON_UPDATE_COMMAND_UI(ID_STOP_SIMULATION, OnUpdateStopSimulation)
   ON_UPDATE_COMMAND_UI(SET_SPEED_FAST, OnUpdateSpeedFast)
   ON_UPDATE_COMMAND_UI(SET_SPEED_MEDIUM, OnUpdateSpeedMedium)
   ON_UPDATE_COMMAND_UI(SET_SPEED_SLOW, OnUpdateSpeedSlow)
   ON_WM_LBUTTONDOWN()
   ON_COMMAND(SIMULATION_STEP, OnStepSimulation)
   ON_WM_LBUTTONUP()
   ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

// CAgentGUIView construction/destruction
CAgentGUIView::CAgentGUIView()
:loopSpeed(MEDIUM_SIMULATION_SPEED), simulationOn(false),
   statisticsOn(false), simulationTime(0)
{
   brushMaleAntelope1.CreateSolidBrush(RGB (200, 0, 0));
   pPenMaleAntelope1.CreatePen(PS_SOLID, 1, RGB (200, 0, 0));

   brushMaleAntelope2.CreateSolidBrush(RGB (200, 200, 0));
   pPenMaleAntelope2.CreatePen(PS_SOLID, 1, RGB (200, 200, 0));

   brushMaleAntelope3.CreateSolidBrush(RGB (0, 140, 0));
   pPenMaleAntelope3.CreatePen(PS_SOLID, 1, RGB (0, 140, 0));

   brushMaleAntelope4.CreateSolidBrush(RGB (0, 140, 200));
   pPenMaleAntelope4.CreatePen(PS_SOLID, 1, RGB (0, 140, 200));

   brushMaleAntelope5.CreateSolidBrush(RGB (0, 0, 255));
   pPenMaleAntelope5.CreatePen(PS_SOLID, 1, RGB (0, 0, 255));

   brushMaleCheetah.CreateSolidBrush(RGB (0, 0, 0));
   pPenMaleCheetah.CreatePen(PS_SOLID, 1, RGB (0, 0, 0));

   brushFemaleCheetah.CreateSolidBrush(RGB (150, 150, 150));
   pPenFemaleCheetah.CreatePen(PS_SOLID, 1, RGB (150, 150, 150));

   pPenFood.CreatePen(PS_SOLID, 1, RGB (150, 200, 150));

}

CAgentGUIView::~CAgentGUIView()
{
}

BOOL CAgentGUIView::PreCreateWindow(CREATESTRUCT& cs)
{
   return CView::PreCreateWindow(cs);
}

// CAgentGUIView drawing

void CAgentGUIView::OnDraw(CDC* pDC)
{
   CAgentGUIDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   static int simTime = 0;
   static numAntelopeStarved    = 0;
   static numAntelopeKilled     = 0;
   static numAntelopeDieOfAge   = 0;
   static numAntelopeDieAsInfant = 0;

   simulationTime = ++simTime;

   //four counters to keep track and report how many of each type are
   //still alive during this time step
   int numMaleAntelope      = 0,
       numFemaleAntelope    = 0,
       numMaleCheetah       = 0,
       numFemaleCheetah     = 0,
       antelopeGeneration   = 0,
       cheetahGeneration    = 0;

   if (statisticsOn)
   {
      printAntelopeStatistics (pDC);
      printCheetahStatistics (pDC);
   }

      pDC->SelectObject(&pPenFood);

      for(int ix = 0; ix < 10; ix++)
      {
         pDC->Ellipse(FOOD_LOCATION[ix] % MAX_X - 30, FOOD_LOCATION[ix] /
                      MAX_X - 30,
                    FOOD_LOCATION[ix] % MAX_X + 30, FOOD_LOCATION[ix] /
                      MAX_X + 30);
      }

      //Paint current male and female Antelope positions on screen
      for (pDoc->aix = pDoc->antelopeMap.begin(); pDoc->aix !=
           pDoc->antelopeMap.end(); ++(pDoc->aix))
      {
         if(pDoc->aix->second.getGeneration() > antelopeGeneration)
           antelopeGeneration = pDoc->aix->second.getGeneration();

         if (pDoc->aix->second.getDeathIndicator() == NOT_DEAD)
         {
            switch(pDoc->aix->second.getMaxSpeed())
               {
                 case  5 :
                 {
                    pDC->SelectObject(&pPenMaleAntelope1);
                    pDC->SelectObject(&brushMaleAntelope1);
                    break;
                 }
                 case  6 :
                 {
                    pDC->SelectObject(&pPenMaleAntelope2);
                    pDC->SelectObject(&brushMaleAntelope2);
                    break;
                 }
                 case  7 :
                 {
                    pDC->SelectObject(&pPenMaleAntelope3);
                    pDC->SelectObject(&brushMaleAntelope3);
                    break;
                 }
                 case  8 :
                 {
                    pDC->SelectObject(&pPenMaleAntelope4);
                    pDC->SelectObject(&brushMaleAntelope4);
                    break;
                 }
                 default://9 and 10
                 {
                    pDC->SelectObject(&pPenMaleAntelope5);
                    pDC->SelectObject(&brushMaleAntelope5);
                    break;
                 }
              }//end switch getSpeed()

            if (pDoc->aix->second.getGender() == MALE)
            {
              numMaleAntelope++;
            }
            else
            {
              numFemaleAntelope++;
            }
            pDC->Rectangle(pDoc->aix->second.getX()-2,
                           pDoc->aix->second.getY()-2,
                           pDoc->aix->second.getX()+2,
                           pDoc->aix->second.getY()+2);
         }//end if NOT_DEAD

      }//end for (aix)

      //Paint current male and female Cheetah positions on screen
      for (pDoc->cix = pDoc->cheetahMap.begin();
           pDoc->cix != pDoc->cheetahMap.end(); ++(pDoc->cix))
      {
         if(pDoc->cix->second.getGeneration() >cheetahGeneration)
           cheetahGeneration = pDoc->cix->second.getGeneration();

         if (pDoc->cix->second.getDeathIndicator() == NOT_DEAD)
         {
           if (pDoc->cix->second.getGender() == MALE)
           {
             numMaleCheetah++;
             pDC->SelectObject(&pPenMaleCheetah);
             pDC->SelectObject(&brushMaleCheetah);
           }
           else
           {
             numFemaleCheetah++;
             pDC->SelectObject(&pPenFemaleCheetah);
             pDC->SelectObject(&brushFemaleCheetah);
           }
           pDC->Rectangle(pDoc->cix->second.getX()-3,
                          pDoc->cix->second.getY()-3,
                          pDoc->cix->second.getX()+3,
                          pDoc->cix->second.getY()+3);
         }//end if NOT_DEAD

      }//end for (cix)

   updateStatusBar(numMaleAntelope, numFemaleAntelope,
      antelopeGeneration, numMaleCheetah, numFemaleCheetah,
      cheetahGeneration, simTime);
```

```
}

// CAgentGUIView printing

BOOL CAgentGUIView::OnPreparePrinting(CPrintInfo* pInfo)
{
   return DoPreparePrinting(pInfo);
}

void CAgentGUIView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
}

void CAgentGUIView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
}

// CAgentGUIView diagnostics

#ifdef _DEBUG
void CAgentGUIView::AssertValid() const
{
   CView::AssertValid();
}

void CAgentGUIView::Dump(CDumpContext& dc) const
{
   CView::Dump(dc);
}

CAgentGUIDoc* CAgentGUIView::GetDocument() // non-debug version is inline
{
   ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CAgentGUIDoc)));
   return (CAgentGUIDoc*)m_pDocument;
}
#endif //_DEBUG

// CAgentGUIView message handlers

void CAgentGUIView::OnTimer(UINT nIDEvent)
{
   CRect rect;
   GetClientRect(&rect);
   CAgentGUIDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   pDoc->moveAllAnimals();
   pDoc->antelopeSensing(simulationTime);
   pDoc->cheetahSensing(simulationTime);

   InvalidateRect(rect);
   CView::OnTimer(nIDEvent);

}

void CAgentGUIView::OnRunSimulation()
```

```
{
   #if !defined MY_TIMER
      #define MY_TIMER
      SetTimer (0, loopSpeed, NULL);
   #endif

   simulationOn = true;

   return;

}

void CAgentGUIView::OnStopSimulation()
{
   KillTimer(0);

   simulationOn = false;

   return;
}

void CAgentGUIView::OnStepSimulation()
{
   #if defined MY_TIMER
      KillTimer(0);
   #endif

   simulationOn = false;

   CRect rect;
   GetClientRect(&rect);

   CAgentGUIDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   pDoc->moveAllAnimals();
   pDoc->antelopeSensing(simulationTime);
   pDoc->cheetahSensing(simulationTime);

   InvalidateRect(rect);

   return;
}


void CAgentGUIView::OnToggle()
{
   if(simulationOn)
   {
      OnStopSimulation();
      simulationOn = false;
   }
   else
   {
      OnRunSimulation();
      simulationOn = true;
```

```
   }
}

void CAgentGUIView::OnSpeedFast()
{
   KillTimer(0);
   loopSpeed = FAST_SIMULATION_SPEED;
   OnRunSimulation();
}

void CAgentGUIView::OnSpeedMedium()
{
   KillTimer(0);
   loopSpeed = MEDIUM_SIMULATION_SPEED;
   OnRunSimulation();
}

void CAgentGUIView::OnSpeedSlow()
{
   KillTimer(0);
   loopSpeed = SLOW_SIMULATION_SPEED;
   OnRunSimulation();
}

void CAgentGUIView::OnUpdateRunSimulation(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(simulationOn == true);
}

void CAgentGUIView::OnUpdateStopSimulation(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(simulationOn == false);
}

void CAgentGUIView::OnUpdateSpeedFast(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(loopSpeed == FAST_SIMULATION_SPEED);
}

void CAgentGUIView::OnUpdateSpeedMedium(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(loopSpeed == MEDIUM_SIMULATION_SPEED);
}

void CAgentGUIView::OnUpdateSpeedSlow(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(loopSpeed == SLOW_SIMULATION_SPEED);
}

void CAgentGUIView::OnLButtonDown(UINT nFlags, CPoint point)
{
   statisticsOn = (statisticsOn == false);

   CView::OnLButtonDown(nFlags, point);
}
```

```
void CAgentGUIView::OnLButtonUp(UINT nFlags, CPoint point)
{
   CView::OnLButtonUp(nFlags, point);
}

//--------------------------------------------------------------------
// Method:      updateStatusBar()
// Parameters: none
// Return val: none
// Purpose:     Updates the numbers of each animal indicated in the Status
//              bar (lower left hand side of the window
//--------------------------------------------------------------------
void CAgentGUIView::updateStatusBar(int numMaleAntelope,
                    int numFemaleAntelope, int antelopeGenerations,
                    int numMaleCheetah, int numFemaleCheetah,
                    int cheetahGenerations, int simTime)
{
   //get a pointer to the window using the global AfxGetApp()
   //function
   CMainFrame* p_mFrame = (CMainFrame*)AfxGetApp()->m_pMainWnd;

   //CString object is required to print in the status bar - we will
   //convert int's to a string of ints
   CString numbers;

   //start developing the coordinates string with number of male Antelope
   numbers = "Antelope - M: ";

   //convert the numMaleAntelope to a string and append it to numbers
   integerToString(numMaleAntelope, numbers);

   //add the Female count to numbers string
   numbers += "F: ";
   integerToString(numFemaleAntelope, numbers);

   //add the Antelope generation count to numbers string
   numbers += "G: ";
   integerToString(antelopeGenerations, numbers);

   //add the Cheetah Male and Female counts to numbers string
   numbers += "   Cheetah - M: ";
   integerToString(numMaleCheetah, numbers);

   numbers += "F: ";
   integerToString(numFemaleCheetah, numbers);

   //add the Cheetah generation count to numbers string
   numbers += "G: ";
   integerToString(cheetahGenerations, numbers);

   numbers += "   Simulation Time: ";
   integerToString(simTime, numbers);

   //call MainFrame's SetPaneText() method, passing the pane# (0)
   //we want to change, and the new value it should reflect
   p_mFrame->SetPaneText(0, numbers);
```

```
}


//-----------------------------------------------------------------------
// Method:      integerToString()
// Parameters: int number - number of which ever animal is passed
//              CString numbers - string representation of our population

// Return val: none
// Purpose:     Converts our animal numbers into a string of integers for
//              use in the status bar.
//-----------------------------------------------------------------------

void CAgentGUIView::printAntelopeStatistics (CDC *pDC)
{
   CAgentGUIDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   //CString object is required to print in the status bar - we will
   //convert int's to a string of ints
   CString numbers;

   //start developing the coordinates string with number of male Antelope
   numbers = "Antelope die:  A: ";

   //convert the numMaleAntelope to a string and append it to numbers
   integerToString(pDoc->antelopeDieOfAge, numbers);

   //add the Female count to numbers string
   numbers += "P: ";
   integerToString(pDoc->antelopeKilled, numbers);

   //add the Female count to numbers string
   numbers += "IM: ";
   integerToString(pDoc->antelopeDiesAsInfant, numbers);

   pDC->SetTextColor (RGB (0,0,0));
   pDC->TextOut (200, 20, numbers);

   //add the Female count to numbers string
   numbers = "Antelope Born: ";
   integerToString(pDoc->numAntelopeCreated, numbers);

   pDC->SetTextColor (RGB (0,0,0));
   pDC->TextOut (200, 35, numbers);
}


//-----------------------------------------------------------------------
// Method:      integerToString()
// Parameters: int number - number of which ever animal is passed
//              CString numbers - string representation of our population

// Return val: none
// Purpose:     Converts our animal numbers into a string of integers for
```

```
//              use in the status bar.
//-----------------------------------------------------------------------
void CAgentGUIView::printCheetahStatistics (CDC *pDC)
{
   CAgentGUIDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   //CString object is required to print in the status bar - we will
   //convert int's to a string of ints
   CString numbers;

   //start developing the coordinates string with number of male Antelope
   numbers = "Cheetah die:  A: ";

   //convert the numMaleAntelope to a string and append it to numbers
   integerToString(pDoc->cheetahDieOfAge, numbers);

   //add the Female count to numbers string
   numbers += "S: ";
   integerToString(pDoc->cheetahDieOfStarvation, numbers);

   //add the Female count to numbers string
   numbers += "IM: ";
   integerToString(pDoc->cheetahDiesAsInfant, numbers);

   pDC->SetTextColor (RGB (0,0,0));
   pDC->TextOut (200, 440, numbers);

   //add the Female count to numbers string
   numbers = "Cheetah Born: ";
   integerToString(pDoc->numCheetahCreated, numbers);

   pDC->SetTextColor (RGB (0,0,0));
   pDC->TextOut (200, 455, numbers);

   //add the Female count to numbers string
   numbers = "Failed Chases: ";
   integerToString(pDoc->numUnsuccessfulChase, numbers);

   pDC->SetTextColor (RGB (0,0,0));
   pDC->TextOut (200, 470, numbers);
}


//-----------------------------------------------------------------------
// Method:      integerToString()
// Parameters: int number - number of which ever animal is passed
//              CString numbers - string representation of our population

// Return val: none
// Purpose:     Converts our animal numbers into a string of integers for
//              use in the status bar.
//-----------------------------------------------------------------------
void CAgentGUIView::integerToString(int num, CString &numbers)
{
   int divisor = 1;
```

```
   int digits  = 1;
   int quotient;

   if (num / FIVE_DIGITS > 0)
   {
      divisor = FIVE_DIGITS;
      digits = 5;
   }
   else if (num / FOUR_DIGITS > 0)
   {
      divisor = FOUR_DIGITS;
      digits = 4;
   }
   else if (num / THREE_DIGITS > 0)
   {
      divisor = THREE_DIGITS;
      digits = 3;
   }
   else if (num / TWO_DIGITS > 0)
   {
      divisor = TWO_DIGITS;
      digits = 2;
   }
   else if (num / ONE_DIGIT > 0)
   {
      divisor = ONE_DIGIT;
      digits = 1;
   }

   for (int ix=0; ix<digits; ix++){
      switch (quotient = (int) num/divisor){
         case 0:
            numbers += '0';
            break;
         case 1:
            numbers += '1';
            break;
         case 2:
            numbers += '2';
            break;
         case 3:
            numbers += '3';
            break;
         case 4:
            numbers += '4';
            break;
         case 5:
            numbers += '5';
            break;
         case 6:
            numbers += '6';
            break;
         case 7:
            numbers += '7';
            break;
         case 8:
```

```
            numbers += '8';
            break;
         case 9:
            numbers += '9';
            break;
      }

      num -= quotient*divisor;
      divisor /= 10;
   }

   numbers += "  ";
}//end integerToString()


//**********************************************************************
//     EXECUTIVE SUMMARY
//File Name:      AgentGUIDoc.h
//
//Authors:        Mark A. Boyd; maboyd@bigfoot.com
//                Todd A. Gagnon; todd@gagnon.com
//
//Description:  interface of the CAgentGUIDoc class
//
//March 1999, Master Thesis
//**********************************************************************

#include <map>
#include "Animal.h"
#include "Antelope.h"
#include "Cheetah.h"

#if
!defined(AFX_AGENTGUIDOC_H__99A2849B_8631_11D2_889B_0000F8092715
   __INCLUDED_)
#define AFX_AGENTGUIDOC_H__99A2849B_8631_11D2_889B_0000F8092715
   __INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

using namespace std;

typedef map<int, Antelope> POSITION2ANTELOPE;
typedef map<int, Cheetah> POSITION2CHEETAH;

#define FIVE_DIGITS   10000
#define FOUR_DIGITS   1000
#define THREE_DIGITS  100
#define TWO_DIGITS    10
#define ONE_DIGIT     1

class CAgentGUIDoc : public CDocument
{
```

```
protected: // create from serialization only
   CAgentGUIDoc();
   DECLARE_DYNCREATE(CAgentGUIDoc)

// Attributes
public:

   POSITION2ANTELOPE antelopeMap;
   POSITION2CHEETAH  cheetahMap;

   POSITION2ANTELOPE tempAntelopeMap;
   POSITION2CHEETAH  tempCheetahMap;

   POSITION2ANTELOPE::iterator aix;
   POSITION2CHEETAH::iterator cix;

   POSITION2ANTELOPE::iterator taix;
   POSITION2CHEETAH::iterator tcix;

   int antelopeDiesAsInfant,
       antelopeDieOfAge,
       antelopeKilled,
       numAntelopeCreated,
       cheetahDiesAsInfant,
       cheetahDieOfAge,
       cheetahDieOfStarvation,
       cheetahKilled,
       numCheetahCreated,
       numUnsuccessfulChase;

// Operations
public:

   public:
   virtual BOOL OnNewDocument();
   virtual void Serialize(CArchive& ar);

// Implementation
public:

   void moveAllAnimals();
   void antelopeSensing(int simTime);
   void cheetahSensing(int simTime);

   virtual ~CAgentGUIDoc();
#ifdef _DEBUG
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

protected:


// Generated message map functions
protected:
   DECLARE_MESSAGE_MAP()
```

```
};

#endif


//**********************************************************************
//    EXECUTIVE SUMMARY
//File Name:    AgentGUIDoc.cpp
//
//Authors:      Mark A. Boyd; maboyd@bigfoot.com
//              Todd A. Gagnon; todd@gagnon.com
//
//Description:  implementation of the CAgentGUIDoc class
//
//March 1999, Master Thesis
//**********************************************************************

#include <ctime>
#include "stdafx.h"
#include "AgentGUI.h"
#include "MainFrm.h"

#include "AgentGUIDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

IMPLEMENT_DYNCREATE(CAgentGUIDoc, CDocument)

BEGIN_MESSAGE_MAP(CAgentGUIDoc, CDocument)

END_MESSAGE_MAP()

// CAgentGUIDoc construction/destruction

CAgentGUIDoc::CAgentGUIDoc()
{

   srand((unsigned)time(NULL));
   rand();

   //Create a group of initial Antelope
   for (int antelopeNum = 0; antelopeNum < NUM_ANTELOPE; antelopeNum++)
   {
      Antelope tempAntelope;

      if (tempAntelope.getGender() == FEMALE)
         tempAntelope.pregPtr = new Pregnancy;
      while(antelopeMap.find(tempAntelope.getLocation()) !=
         antelopeMap.end())
      {
         tempAntelope.avoidCollision();
```

```
      }//end while
      antelopeMap.insert(POSITION2ANTELOPE::value_type
         (tempAntelope.getLocation(), tempAntelope));

   }//end for (antelopeNum)

   //Create a group of initial Cheetah
   for (int cheetahNum = 0; cheetahNum < NUM_CHEETAH; cheetahNum++)
   {
      Cheetah tempCheetah;

      if (tempCheetah.getGender() == FEMALE)
         tempCheetah.pregPtr = new Pregnancy;
      while(cheetahMap.find(tempCheetah.getLocation()) !=
         cheetahMap.end())
      {
         tempCheetah.avoidCollision();
      }//end while
      cheetahMap.insert(POSITION2CHEETAH::value_type(
         tempCheetah.getLocation(), tempCheetah));

   }//end for (cheetahNum)

   antelopeDiesAsInfant   = 0;
   antelopeDieOfAge       = 0;
   antelopeKilled         = 0;
   numAntelopeCreated     = 0;
   cheetahDiesAsInfant    = 0;
   cheetahDieOfAge        = 0;
   cheetahDieOfStarvation = 0;
   cheetahKilled          = 0;
   numCheetahCreated      = 0;
   numUnsuccessfulChase   = 0;

}

CAgentGUIDoc::~CAgentGUIDoc()
{
}

BOOL CAgentGUIDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

      return TRUE;
}


// CAgentGUIDoc serialization
void CAgentGUIDoc::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
   }
   else
```

```
   {
   }
}

// CAgentGUIDoc diagnostics
#ifdef _DEBUG
void CAgentGUIDoc::AssertValid() const
{
   CDocument::AssertValid();
}

void CAgentGUIDoc::Dump(CDumpContext& dc) const
{
   CDocument::Dump(dc);
}
#endif //_DEBUG

// CAgentGUIDoc commands

//---------------------------------------------------------------------
// Function:   CAgentGUIDoc::moveAllAnimals
// Return Val: None
// Parameter:  None
// Purpose:    Steps through the list of alive animals
//             and updates their position, checks age...
//---------------------------------------------------------------------
void CAgentGUIDoc::moveAllAnimals()
{
// MOVE ANTELOPE
      for (aix = antelopeMap.begin(); aix != antelopeMap.end(); ++aix)
      {
         //Checks to see what the agent's move goal is and moves
         switch(aix->second.getNextAction())
         {
            case A_MATE :
            {
               aix->second.setSpeedOfNextMove(REGULAR);
               aix->second.moveTo(aix->second.getMoveToLocation());
               break;
            }
            case A_NOTHING :
            {
               if(rand() < RAND_MAX/2)
                  aix->second.setSpeedOfNextMove(REGULAR);
               else
                  aix->second.setSpeedOfNextMove(REST);
               aix->second.move();
               break;
            }
            case FEED :
            {
               aix->second.setSpeedOfNextMove(REGULAR);
               aix->second.moveTo(aix->second.getMoveToLocation());
               break;
            }
            case HERD :
```

```
         {
            aix->second.setSpeedOfNextMove(REGULAR);
            aix->second.moveTo(aix->second.getMoveToLocation());
            break;
         }
         default:// FLEE
         {
            aix->second.setSpeedOfNextMove(RUN);
            aix->second.moveFrom(aix->second.getMoveFromLocation());
            break;
         }

      }//end switch getNextAction()

      //test for collisions of agents and adjust one
      while(tempAntelopeMap.find(aix->second.getLocation()) !=
         tempAntelopeMap.end())
      {
         aix->second.avoidCollision();
      }//end while
      tempAntelopeMap.insert(POSITION2ANTELOPE::value_type
         (aix->second.getLocation(), aix->second));

   }//end for (aix) Antelope Move Loop

   antelopeMap = tempAntelopeMap;
   tempAntelopeMap.clear();

// MOVE CHEETAH

   for (cix = cheetahMap.begin(); cix != cheetahMap.end(); ++cix)
   {
      //Checks to see what the agent's move goal is and moves accordingl
y
      switch(cix->second.getNextAction())
      {
         case C_MATE :
         {
            cix->second.setEnergyLevel(cix->second.getEnergyLevel() -
               CHEETAH_REGULAR_ENERGY_PENALTY);
            cix->second.setSpeedOfNextMove(REGULAR);

            cix->second.moveTo(cix->second.getMoveToLocation());
            break;
         }
         case C_NOTHING :
         {
            if(cix->second.isResting())
            {
               cix->second.setEnergyLevel(cix->second.getEnergyLevel()
                  + 4*CHEETAH_REST_ENERGY_GAIN);
               cix->second.setSpeedOfNextMove(REST);
            }
            else //if not rest then move normally
            {
               if(rand() < RAND_MAX/2)
               {
                  cix->second.setEnergyLevel
                     (cix->second.getEnergyLevel() -
                        CHEETAH_REGULAR_ENERGY_PENALTY);
                  cix->second.setSpeedOfNextMove(REGULAR);
               }
               else
               {
                  cix->second.setEnergyLevel
                     (cix->second.getEnergyLevel() +
                        CHEETAH_REST_ENERGY_GAIN);
                  cix->second.setSpeedOfNextMove(REST);
               }
            }
            cix->second.move();
            break;
         }
         case AVOID :
         {
            cix->second.setEnergyLevel(cix->second.getEnergyLevel() -
               CHEETAH_REGULAR_ENERGY_PENALTY);
            cix->second.setSpeedOfNextMove(REGULAR);
            cix->second.moveFrom(cix->second.getMoveFromLocation());
            break;
         }
         default://CHASE
         {
            cix->second.setEnergyLevel(cix->second.getEnergyLevel() -
               CHEETAH_RUN_ENERGY_PENALTY);
            cix->second.setSpeedOfNextMove(RUN);
            cix->second.moveTo(cix->second.getMoveToLocation());
            break;
         }
      }//end switch getNextAction()

      //test for collisions of agents and adjust one
      while(tempCheetahMap.find(cix->second.getLocation()) !=
         tempCheetahMap.end())
      {
         cix->second.avoidCollision();
      }//end while
      tempCheetahMap.insert(POSITION2CHEETAH::value_type
         (cix->second.getLocation(), cix->second));

   }//end for (cix) Cheetah Move Loop

   cheetahMap = tempCheetahMap;
   tempCheetahMap.clear();

}//end CAgentGUIDoc::moveAllAnimals()

//----------------------------------------------------------------------
// Function:    CAgentGUIDoc::antelopeSensing()
// Return Val: None
// Parameter:  None
// Purpose:     Steps through the list of alive male and female Antelope
//              and allows them to sense their environment and decide what
//              to do for their next action
//----------------------------------------------------------------------
void CAgentGUIDoc::antelopeSensing(int simTime)
{
   for (aix = antelopeMap.begin(); aix != antelopeMap.end(); ++aix)
   {
      if ((simTime%365 > ANTELOPE_START_IN_SEASON) && (simTime%187 <
         ANTELOPE_STOP_IN_SEASON))
         aix->second.setInSeason(true);
      else
         aix->second.setInSeason(false);

      int upper = MAX_X * MAX_Y,
          lower = MIN_X * MIN_Y,
         currentSensingRange = 0;

      switch(aix->second.getSpeedOfNextMove())
      {
         case REST :
         {
            currentSensingRange = ANTELOPE_REST_SENSING_RANGE;
            lower = aix->second.getLocation() - MAX_X *
               ANTELOPE_REST_SENSING_RANGE -
               ANTELOPE_REST_SENSING_RANGE;
            upper = aix->second.getLocation() + MAX_X *
               ANTELOPE_REST_SENSING_RANGE +
               ANTELOPE_REST_SENSING_RANGE;
            break;
         }
         case REGULAR :
         {
            currentSensingRange = ANTELOPE_REGULAR_SENSING_RANGE;
            lower = aix->second.getLocation() - MAX_X *
               ANTELOPE_REGULAR_SENSING_RANGE -
               ANTELOPE_REGULAR_SENSING_RANGE;
            upper = aix->second.getLocation() + MAX_X *
               ANTELOPE_REGULAR_SENSING_RANGE +
               ANTELOPE_REGULAR_SENSING_RANGE;
            break;
         }
         default://case RUN
         {
            currentSensingRange = ANTELOPE_RUN_SENSING_RANGE;
            lower = aix->second.getLocation() - MAX_X *
               ANTELOPE_RUN_SENSING_RANGE -
               ANTELOPE_RUN_SENSING_RANGE;
            upper = aix->second.getLocation() + MAX_X *
               ANTELOPE_RUN_SENSING_RANGE +
               ANTELOPE_RUN_SENSING_RANGE;
            break;
         }
      }//end switch

      if (lower < MIN_X * MIN_Y)
         lower = MIN_X * MIN_Y;
      if (upper > MAX_X * MAX_Y)
         upper = MAX_X * MAX_Y;

      int partnerMoveToDistance  = 100;
      int friendMoveToDistance = 100;
      bool mated          = false;
      bool foundPartner = false;
      bool foundFriend  = false;

      for (taix = antelopeMap.lower_bound(lower); taix !=
         antelopeMap.upper_bound(upper); ++taix)
      {
         if(abs(aix->second.getX() - taix->second.getX()) <=
            currentSensingRange)
         {
            if(aix->second.getIdNum() != taix->second.getIdNum())
            {
               if(taix->second.isDead() &&
                  (taix->second.getDeathIndicator() == PREDATOR))
                  aix->second.setPredatorKnowledge(TRUE);

               if (aix->second.canMate(taix->second) && !mated)
               {
                  mated = true;
                  aix->second.mate(taix->second);
                  aix->second.setNextAction(FEED);
               }//end if
               else if(aix->second.mateEligible(taix->second))
               {
                  foundPartner = true;
                  if(aix->second.getDistance(taix->second) <
                     partnerMoveToDistance)
                  {
                     partnerMoveToDistance =
                        aix->second.getDistance(taix->second);
                     aix->second.setMoveToLocation
                        (taix->second.getLocation());
                  }//end if
                  aix->second.setNextAction(A_MATE);
               }//end else if mateEligible(taix)
               else if(!foundPartner)
               {
                  if((aix->second.getDistance(taix->second) <
                     friendMoveToDistance) &&
                     (aix->second.getDistance(taix->second) >
                        FRIEND_STANDOFF_DISTANCE))
                  {
                     foundFriend = true;
                     friendMoveToDistance =
                        aix->second.getDistance(taix->second);
                     aix->second.setMoveToLocation
                        (taix->second.getLocation());
                     aix->second.setNextAction(HERD);
                  }//end if
               }//end else if
```

```
         }//end if getIdNum()

       }//end if (aix->getX)

    }//end for taix Antelope Map - sense other antelope

    int foodDistance = 10000,
        foodLocation;

    for(int ix = 0; ix < 10; ix++)
    {
       while(aix->second.distanceFromFood(FOOD_LOCATION[ix]) <
          foodDistance)
       {
          foodDistance = aix->second.distanceFromFood
             (FOOD_LOCATION[ix]);
          foodLocation = FOOD_LOCATION[ix];
       }//end while
    }//end for

    //if your not going to mate, if you haven't found found
    //   a friend or 1/2 the time
    //   when you have found a friend antelope will move to food
    if(!foundPartner)
    {
       if(!foundFriend || (rand() < RAND_MAX/2))
       {
          if(foodDistance < FOOD_RANGE)
             aix->second.setNextAction(A_NOTHING);
          else
          {
             aix->second.setMoveToLocation(foodLocation);
             aix->second.setNextAction(FEED);
          }//end else
       }//end if
    }//end if

    int tempMoveDistance = 100;

  //check for predators if predator knowledge = true
  if(aix->second.getPredatorKnowledge())
  {
    for (tcix = cheetahMap.lower_bound(lower); tcix !=
       cheetahMap.upper_bound(upper); ++tcix)
    {
       if(abs(aix->second.getX() - tcix->second.getX()) <=
          currentSensingRange)
       {
          if(aix->second.getDistance(tcix->second)< tempMoveDistance)
          {
             tempMoveDistance = aix->second.getDistance
                (tcix->second);
             aix->second.setMoveFromLocation
                (tcix->second.getLocation());
          }//end if

             aix->second.setNextAction(FLEE);
          }//end if aix->getX()

     }//end for tcix - CheetahMap - sense cheetahs
  }//end if (aix->second.getPredatorKnowledge()

  //NOW THAT THEY HAVE SENSED AND DECIDED WHAT TO DO, DO IT
  //ANTELOPE ACTION
  if(aix->second.isPregnant())
  {
     if(aix->second.pregPtr->gestationTime== ANTELOPE_GESTATION_PERIOD)
     {
        int litter = aix->second.litterSize();

        numAntelopeCreated += litter;

        Antelope *babyAntelope;
        for (int ix = 1; ix <= litter; ix++)
        {
           babyAntelope = aix->second.giveBirth
              (aix->second.pregPtr->maleSpeed,
                      aix->second.getMaxSpeed(),
                         aix->second.getGeneration(),
                      aix->second.getLocation());

           if(babyAntelope->diesAsInfant())
           {
              babyAntelope->setDeathIndicator(INFANT_MORTALITY);
              antelopeDiesAsInfant++;
           }
           if (babyAntelope->getGender() == FEMALE)
              babyAntelope->pregPtr = new Pregnancy;
           while(tempAntelopeMap.find(babyAntelope->getLocation()) !=
              tempAntelopeMap.end())
           {
              babyAntelope->avoidCollision();
           }//end while
           tempAntelopeMap.insert(POSITION2ANTELOPE::value_type
              (babyAntelope->getLocation(), *babyAntelope));
        }//end for (ix) - create litter of size litter

        aix->second.setPregnant(false);
     }//end if pregancy gestation time > ANTELOPE_GESTATION_TIME
     else
     {
     aix->second.pregPtr->gestationTime++;
     }//end else
  }//end if aix->isPregnant()

     aix->second.growOlder();

     //check age and if over MAX_AGE then set deathIndicator to OLD_AGE
     if(aix->second.getAge() == MAX_ANTELOPE_AGE)
     {
        aix->second.setDeathIndicator(OLD_AGE);

        antelopeDieOfAge++;
     }
     //Last thing we do is check to make sure the antelope didn't die
     //two time steps ago.  if so take out of world otherwise increment

     //counterthis allows the other animals to sense this one and
     //learn how it died

     if((aix->second.getDeathIndicator() != NOT_DEAD) &&
        (aix->second.getDeathCounter() < 2))
     {
        aix->second.setDeathCounter(aix->second.getDeathCounter() + 1);
          while(tempAntelopeMap.find(aix->second.getLocation()) !=
             tempAntelopeMap.end())
          {
             aix->second.avoidCollision();
          }//end while
        tempAntelopeMap.insert(POSITION2ANTELOPE::value_type
           (aix->second.getLocation(), aix->second));
     }//end if getDeathIndicator()
     else if(aix->second.getDeathIndicator() == NOT_DEAD)
     {
          while(tempAntelopeMap.find(aix->second.getLocation()) !=
             tempAntelopeMap.end())
          {
             aix->second.avoidCollision();
          }//end while
        tempAntelopeMap.insert(POSITION2ANTELOPE::value_type
           (aix->second.getLocation(), aix->second));
     }//end if else getDeathIndicator()

  }//end for (aix) Antelope Sensing Loop

  // antelopeMap.clear();
  antelopeMap = tempAntelopeMap;
  tempAntelopeMap.clear();

}//end CAgentGUIDoc::antelopeSensing()

//----------------------------------------------------------------------
// Function:    CAgentGUIDoc::cheetahSensing()
// Return Val: None
// Parameter:  None
// Purpose:     Steps through the list of alive male and female Cheetah
//              and allows them to sense their environment and decide what
//              to do for their next action
//----------------------------------------------------------------------
void CAgentGUIDoc::cheetahSensing(int simTime)
{
  for (cix = cheetahMap.begin(); cix != cheetahMap.end(); ++cix)
  {
     if(cix->second.isResting())
     {
        if(cix->second.getEnergyLevel() > CHEETAH_RESUME_HUNTING_LEVEL)

        {
           numUnsuccessfulChase++;
           cix->second.setRest(false);
        }
     }
     else
     {

        if ((simTime%365 > CHEETAH_START_IN_SEASON) && (simTime%365 <
              CHEETAH_STOP_IN_SEASON))
           cix->second.setInSeason(true);
        else
           cix->second.setInSeason(false);

        int upper,
            lower,
            currentSensingRange;

        switch(cix->second.getSpeedOfNextMove())
        {
           case REST :
           {
              currentSensingRange = CHEETAH_REST_SENSING_RANGE;
              lower = cix->second.getLocation() - MAX_X *
                 CHEETAH_REST_SENSING_RANGE -
                 CHEETAH_REST_SENSING_RANGE;
              upper = cix->second.getLocation() + MAX_X *
                 CHEETAH_REST_SENSING_RANGE +
                 CHEETAH_REST_SENSING_RANGE;
              break;
           }
           case REGULAR :
           {
              currentSensingRange = CHEETAH_REGULAR_SENSING_RANGE;
              lower = cix->second.getLocation() - MAX_X *
                 CHEETAH_REGULAR_SENSING_RANGE -
                 CHEETAH_REGULAR_SENSING_RANGE;
              upper = cix->second.getLocation() + MAX_X *
                 CHEETAH_REGULAR_SENSING_RANGE +
                 CHEETAH_REGULAR_SENSING_RANGE;
              break;
           }
           default://case RUN
           {
              currentSensingRange = CHEETAH_RUN_SENSING_RANGE;
              lower = cix->second.getLocation() - MAX_X *
                 CHEETAH_RUN_SENSING_RANGE -
                 CHEETAH_RUN_SENSING_RANGE;
              upper = cix->second.getLocation() + MAX_X *
                 CHEETAH_RUN_SENSING_RANGE +
                 CHEETAH_RUN_SENSING_RANGE;
              break;
           }
        }//end switch getSpeedofNextMove

        //ensure we aren't trying to sense outside the world
```

```
    if (lower < MIN_X * MIN_Y)
       lower = MIN_X * MIN_Y;
    if (upper > MAX_X * MAX_Y)
       upper = MAX_X * MAX_Y;

    int partnerMoveToDistance = 1000;
    int avoidDistance = CHEETAH_AVOID_RANGE;
    bool foundPartner = false;
    bool mated = false;
    bool avoidCheetah = false;

    //sense other cheetah
    //still only sensing inside range
    for (tcix = cheetahMap.begin(); tcix != cheetahMap.end(); ++tcix)
    {
          if(cix->second.getIdNum() != tcix->second.getIdNum())
          {
             if (cix->second.canMate(tcix->second) && !mated)
             {
                mated = true;
                cix->second.mate(tcix->second);
                cix->second.setNextAction(C_NOTHING);
             }//end if canMate()
             else if(cix->second.mateEligible(tcix->second))
             {
                foundPartner = true;
                if(cix->second.getDistance(tcix->second) <
                   partnerMoveToDistance)
                {
                   partnerMoveToDistance =
                      cix->second.getDistance(tcix->second);
                   cix->second.setMoveToLocation
                      (tcix->second.getLocation());
                }//end if getDistance(tcix)
                cix->second.setNextAction(C_MATE);
             }//end else if mateEligible(tcix)
             else if (!mated && !foundPartner &&
                (cix->second.getDistance
                   (tcix->second) < avoidDistance))
             {
                avoidCheetah = true;
                avoidDistance = cix->second.getDistance
                   (tcix->second);
                cix->second.setMoveFromLocation
                   (tcix->second.getLocation());
                cix->second.setNextAction(AVOID);
             }//end else if (!tempMateFlag)

          }//end if getIdNum()

       if(!foundPartner && !avoidCheetah)
          cix->second.setNextAction(C_NOTHING);

    }//end for (tcix) cheetahMap - sense other cheetahs

    int preyMoveToDistance = 1000;

          if(!foundPartner && !(cix->second.isResting()) &&
             (cix->second.getEnergyLevel() < CHEETAH_HIGH_ENERGY_LEVEL))
          {
             //now sense antelope in world
             for (taix = antelopeMap.lower_bound(lower); taix !=
                antelopeMap.upper_bound(upper); ++taix)
             {
                if((abs(cix->second.getX() - taix->second.getX()) <=
                   currentSensingRange) &&
                   (taix->second.getDeathIndicator() == NOT_DEAD))
                {
                   if(cix->second.canKill(taix->second))
                   {
                      cix->second.setEnergyLevel
                         (cix->second.getEnergyLevel() +
                         CHEETAH_ENERGY_BOOST);
                      taix->second.setDeathIndicator(PREDATOR);
                      antelopeKilled++;
                      cix->second.setNextAction(C_NOTHING);
                   }//end if canKill()
                   else
                   {
                      if(cix->second.getDistance(taix->second) <
                         preyMoveToDistance)
                      {
                         preyMoveToDistance =
                            cix->second.getDistance(taix->second);
                         cix->second.setMoveToLocation
                            (taix->second.getLocation());
                      }//end if getDistance

                      cix->second.setNextAction(CHASE);
                   }//end if else canKill(taix)
                }//end if cix->getX()

                //check to make sure still has enough energy to hunt
                if(cix->second.getEnergyLevel() <
                   CHEETAH_STOP_HUNTING_LEVEL)
                {
                   cix->second.setRest(true);
                   cix->second.setNextAction(C_NOTHING);
                }//end if

             }//end for taix AntelopeMap - cheetah sensing antelope

          }//end if (!foundPartner)

       }//end else if isResting()

       //PUT IN CHEETAH ACTION CODE

       if(cix->second.isPregnant())
       {
          if(cix->second.pregPtr->gestationTime==CHEETAH_GESTATION_PERIOD)
          {

       int litter = cix->second.litterSize();

       Cheetah *babyCheetah;
       for (int ix = 1; ix <= litter; ix++)
       {
          numCheetahCreated++;

          babyCheetah = cix->second.giveBirth
                         (cix->second.pregPtr->maleSpeed,
                         cix->second.getMaxSpeed(),
                         cix->second.getGeneration(),
                         cix->second.getLocation());

          if(babyCheetah->diesAsInfant())
          {
             babyCheetah->setDeathIndicator(INFANT_MORTALITY);
             cheetahDiesAsInfant++;
          }
          if (babyCheetah->getGender() == FEMALE)
             babyCheetah->pregPtr = new Pregnancy;
          while(tempCheetahMap.find(babyCheetah->getLocation()) !=
                tempCheetahMap.end())
          {
             babyCheetah->avoidCollision();
          }//end while
          tempCheetahMap.insert(POSITION2CHEETAH::value_type
             (babyCheetah->getLocation(), *babyCheetah));

       }//end for (ix) create litter of size litter

       cix->second.setPregnant(false);

    }//end if pregPtr->gestationTime() == CHEETAH_GESTATION_TIME
    else
    {
       cix->second.pregPtr->gestationTime++;
    }//end else pregPtr->gestationTime() != CHEETAH_GESTATION_TIME

 }//end if cix->ifPregnant()

 cix->second.growOlder();

 //check age and if over MAX_AGE then set deathIndicator to OLD_AGE
 if(cix->second.getAge() == MAX_CHEETAH_AGE)
 {
    cix->second.setDeathIndicator(OLD_AGE);
    cheetahDieOfAge++;
 }

 if((cix->second.getEnergyLevel() < CHEETAH_STARVATION_LEVEL) &&
    (cix->second.getDeathIndicator() == NOT_DEAD))
 {
    cix->second.setDeathIndicator(STARVATION);
    cheetahDieOfStarvation++;
 }

       //last thing we do is check to make sure the antelope didn't die two
       //time steps ago.  if so take out of world otherwise increment counte
r
       //this allows the other animals to sense this one and learn how it di
ed

       if((cix->second.getDeathIndicator() != NOT_DEAD) &&
          (cix->second.getDeathCounter() < 2))
       {
          cix->second.setDeathCounter(cix->second.getDeathCounter() + 1);
                while(tempCheetahMap.find(cix->second.getLocation()) !=
                   tempCheetahMap.end())
                {
                   cix->second.avoidCollision();
                }//end while
          tempCheetahMap.insert(POSITION2CHEETAH::value_type
             (cix->second.getLocation(), cix->second));
       }//end if getDeathIndicator()
       else if (cix->second.getDeathIndicator() == NOT_DEAD)
       {
                while(tempCheetahMap.find(cix->second.getLocation()) !=
                   tempCheetahMap.end())
                {
                   cix->second.avoidCollision();
                }//end while
          tempCheetahMap.insert(POSITION2CHEETAH::value_type
             (cix->second.getLocation(), cix->second));
       }//end if else getDeathIndicator()

    }//end for (aix) Antelope Sensing Loop

    cheetahMap = tempCheetahMap;
    tempCheetahMap.clear();

}//end CAgentGUIDoc::cheetahSensing();
```

BAMBOO IMPLEMENTATION

```
// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: npsAgent.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Declaration for the npsAgent class. This abstract class
//              implements the base functionality used by all agents

// March 1999 Master Thesis
// ******************************************************************

#ifndef _npsAgent_h
#define _npsAgent_h

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "bbThread.h"
#include "npsGeometry.h"
#include "bbSafeClass.h"
#include "bbNamedObject.h"
#include "bbListedClass.h"
#include "npsVisualApi.h"
#include "vector.h"
#include "npsAgentApi.h"
#include <math.h>


// ******************************************************************
//   DEFINES
// ******************************************************************

#define X      0
#define Y      1
#define Z      2
#define MIN_X  -50
#define MAX_X  50
#define MIN_Y  -50
#define MAX_Y  50
#define MIN_Z  -50
#define MAX_Z  50

enum AGENT_RELATIONSHIPS {PREDATOR, ENEMY, FRIENDLY, FOOD, UNKNOWN};
typedef vector<char*> agentRelationsVector;

class npsAgent;

#ifdef _npsAgent_c
  ACE_EXPORT_SINGLETON_DECLARATION(bbSafeClass<npsAgent>);
  ACE_EXPORT_SINGLETON_DECLARATION(bbListedClass<npsAgent>);
#else
  ACE_IMPORT_SINGLETON_DECLARATION(bbSafeClass<npsAgent>);
  ACE_IMPORT_SINGLETON_DECLARATION(bbListedClass<npsAgent>);
#endif

// ******************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

class AGENT_API npsAgent : public npsGeometry,
                           public bbSafeClass<npsAgent>,
                           public bbListedClass<npsAgent>,
                           public bbNamedObject
{
  private:
   char           *agentType;
   int            speed,
                  age,
                  sensingRange,
                  energyLevel;
   bool           remove;

   agentRelationsVector knownPredators,
                        knownPrey,
                        knownFriends,
                        knownEnemies,
                        knownFood,
                        unknownAgents;

    npsAgent();
    // disable default construtor

  protected:
    // Never create protected member variables as they may
    // be corrupted by threads accessing them via objects
    // instantiated from derived classes.  Note that even
    // internal routines must be concious when accessing
    // them directly.

    npsAgent(bbCallbackFunc *callbackFunc);
      // contructor for derived agent

  public:

    virtual ~npsAgent() = 0;
    // Destruct a device object


   //get distance between two agents
   float getDistance(npsAgent &agent);
   float getDistanceFromLocation(npsVec3f location);

   void setAgentType(char *at);
   char* getAgentType();
   bool isSameAgentType(npsAgent *agent);

   void addToPredators(char *name);
   void addToFriends(char *name);
   void addToEnemies(char *name);
   void addToFood(char *name);
   void addToUnknown(char *name);

   void setRandomPosition();

   void setSpeed (int s);
   int getSpeed ();

   void setAge (int a);
   int getAge();
   void growOlder();

   void setSensingRange(int sr);
   int getSensingRange();

   void setEnergyLevel(int el);
   int getEnergyLevel();

   void setRemove();
   bool getRemove();

   AGENT_RELATIONSHIPS npsAgent::getRelationship (npsAgent *agent);

   virtual void updatePosition(int time) = 0;
   virtual void sense(int time) = 0;
   virtual bool isKilled(npsAgent &agent) = 0;

   //general utilities that might be useful
   double myRand ();
   char* npsAgent::integerToString(int inNum);

};

// ******************************************************************
//   INLINED MEMBER FUNCTIONS
// ******************************************************************

inline void npsAgent::setAgentType(char *at)
{
   agentType = at;
}

inline char* npsAgent::getAgentType()
{
   return(agentType);
}

inline void npsAgent::setSpeed (int s)
{
   speed = s;
}

inline int npsAgent::getSpeed ()
{
   return speed;
}

inline void npsAgent::setAge (int a)
{
   age = a;
}

inline int npsAgent::getAge()
```

```
{
   return age;
}

inline void npsAgent::growOlder()
{
   age++;
}

inline void npsAgent::setSensingRange(int sr)
{
   sensingRange = sr;
}

inline int npsAgent::getSensingRange()
{
   return sensingRange;
}

inline void npsAgent::setEnergyLevel(int el)
{
   energyLevel = el;
}

inline int npsAgent::getEnergyLevel()
{
   return energyLevel;
}

inline void npsAgent::setRemove()
{
   remove = true;
}

inline bool npsAgent::getRemove()
{
   return(remove);
}

#endif // _npsAgent_h


// *****************************************************************
//   EXECUTIVE SUMMARY
// Module Name: npsAgent.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Implementation for the npsAgent class. This abstract class
//              implements the base functionality used by all agents

// March 1999 Master Thesis
// *****************************************************************

// *****************************************************************
//   INCLUDES AND EXTERNS
// *****************************************************************
```

```
#define _npsAgent_c
#include "npsAgent.h"
#include <GL/gl.h>


// *****************************************************************
//   DEFINES & FILE SCOPE VARIABLES
// *****************************************************************

bbThread  *thread;

// *****************************************************************
//   CODE
// *****************************************************************

void updateFunc(bbThread *thread, bbData *data)
{
   static int time = 0;
   npsAgent *agent, *sensedAgent;

   int numAgents = bbListedClass<npsAgent>::getNumObjects();
   for (int i = 0; i < numAgents; i++)
   {
      agent = bbListedClass<npsAgent>::getObject(i);
      agent->sense(time);
   }

   for (int j = 0; j < numAgents; j++)
   {
      agent = bbListedClass<npsAgent>::getObject(j);
      agent->updatePosition(time);
      if (agent->getRemove())
      {
         delete agent;
         //decrement counters to account for deleted object
         j--; numAgents--;
      }
   }

   if (time%100 == 0)
   {
      cout<<"simulation time = " << time<<endl;
      cout<<"num Agents = "<<numAgents<<endl;
   }
   time++;
}


npsAgent::npsAgent(bbCallbackFunc *_callbackFunc)
: npsGeometry(_callbackFunc), speed(-1), age(-1), sensingRange(-1),
  energyLevel(-1), agentType("npsAgent")
{
   static bool firsttime = 1;

   // first check/set this class's type
   if (firsttime)
   {
      srand((unsigned)time(NULL));  //seed the random number generator
```

```
      rand();
      thread = new bbThread(updateFunc, 0, CYCLE_RATE, 100.0);
      firsttime = 0;
   }
}


npsAgent::~npsAgent()
{
   //do nothing
}


//-----------------------------------------------------------------
// Function:   npsAgent::getRelationship (bbType)
// Return Val: AGENT_RELATIONSHIPS
// Parameter:  bbType
// Purpose:    Allows an agent to get the class type of another
//             agent and determine what relationship it has with the
//             new agent.  If
//-----------------------------------------------------------------

AGENT_RELATIONSHIPS npsAgent::getRelationship (npsAgent *agent)
{
   AGENT_RELATIONSHIPS relationship = UNKNOWN;
   agentRelationsVector::iterator it;

   if(!knownPredators.empty())
   {
      for (it = knownPredators.begin(); it != knownPredators.end(); it++)
      {
         if(!(strcmp(agent->getAgentType(),*it)))
            relationship = PREDATOR;
      }
   }

   if(!knownFriends.empty())
   {
      for (it = knownFriends.begin(); it != knownFriends.end(); it++)
      {
         if(!(strcmp(agent->getAgentType(),*it)))
         {
            relationship = FRIENDLY;
         }
      }
   }

   if(!knownEnemies.empty())
   {
      if(!(strcmp(agent->getAgentType(),*it)))
      {
         if(agent->getAgentType() == *it)
            relationship = ENEMY;
      }
   }

   if(!knownFood.empty())
   {
```

```
      for (it = knownFood.begin(); it != knownFood.end(); it++)
      {
         if(!(strcmp(agent->getAgentType(),*it)))
            relationship = FOOD;
      }
   }

   if(!unknownAgents.empty())
   {
      for (it = unknownAgents.begin(); it != unknownAgents.end(); it++)
      {
         if(!(strcmp(agent->getAgentType(),*it)))
            relationship = UNKNOWN;
      }
   }

   return (relationship);
}


//-----------------------------------------------------------------
//Function:    getDistance(npsAgent)
//Return Val: int distance between two agents
//Parameter:
//Purpose:     determine distance between two agents
//-----------------------------------------------------------------
float npsAgent::getDistance(npsAgent &agent)
{
   npsVec3f thisPosition, agentPosition;
   int      xSquare, zSquare;
   float    answer;

   this->getPosition(thisPosition);
   agent.getPosition(agentPosition);

   xSquare = (thisPosition[X] - agentPosition[X]) * (thisPosition[X] -
             agentPosition[X]);
   zSquare = (thisPosition[Z] - agentPosition[Z]) * (thisPosition[Z] -
             agentPosition[Z]);
   answer  = (sqrt(xSquare + zSquare));

   return (answer);

}//end npsAgent::getDistance()


//-----------------------------------------------------------------
//Function:    distanceFromLocation()
//Return Val: int between animal and food
//Parameter:
//Purpose:     determine distance between animal and food
//-----------------------------------------------------------------
float npsAgent::getDistanceFromLocation(npsVec3f location)
{
   npsVec3f thisPosition;
   int      xSquare, zSquare;
   float    answer;

   this->getPosition(thisPosition);
```

```
   xSquare = (thisPosition[X] - location[X]) * (thisPosition[X] -
             location[X]);
   zSquare = (thisPosition[Z] - location[Z]) * (thisPosition[Z] -
             location[Z]);
   answer = (sqrt(xSquare + zSquare));

   return (answer);

}//end npsAgent::distanceFromLocation()


//---------------------------------------------------------------------
//Function:   isSameAgentType(npsAgent*)
//Return Val: bool
//Parameter:  npsAgent
//Purpose:    return true if the passed agent is the same type as this
//---------------------------------------------------------------------
bool npsAgent::isSameAgentType(npsAgent *agent)
{
   return(this->getAgentType() == agent->getAgentType());

}//end npsAgent::isSameType()


//---------------------------------------------------------------------
//Function:   setRandomLocation()
//Return Val:
//Parameter:
//Purpose:    allows an agent to be placed in a random location.  The
//            altitude remains constant at three so if you want to set
//            a random altitude as well then you must overload this func
//---------------------------------------------------------------------
void npsAgent::setRandomPosition()
{
   npsVec3f position;

   float x, y, z = 0.0;

   x = MIN_X + myRand() * (MAX_X - MIN_X);
   z = MIN_Z + myRand() * (MAX_Z - MIN_Z);
   //use constant altitude for now
   y = 1;

   position.set(x,y,z);
   this->setPosition(position);
}


//---------------------------------------------------------------------
//Function:   addToPredator(char*)
//Return Val:
//Parameter:  char* - agentType
//Purpose:    will add the agentType to vector of known predators
//---------------------------------------------------------------------
void npsAgent::addToPredators(char *name)
{
   knownPredators.insert(knownPredators.end(), name);
}


//---------------------------------------------------------------------
//Function:   addToFriends(char*)
//Return Val:
//Parameter:  char* - agentType
//Purpose:    will add the agentType to vector of known Friends
//---------------------------------------------------------------------
void npsAgent::addToFriends(char *name)
{
   knownFriends.insert(knownFriends.end(), name);
}


//---------------------------------------------------------------------
//Function:   addToEnemies(char*)
//Return Val:
//Parameter:  char* - agentType
//Purpose:    will add the agentType to vector of known enemies
//---------------------------------------------------------------------
void npsAgent::addToEnemies(char *name)
{
   knownEnemies.insert(knownEnemies.end(), name);
}


//---------------------------------------------------------------------
//Function:   addToFood(char*)
//Return Val:
//Parameter:  char* - agentType
//Purpose:    will add the agentType to vector of known Food
//---------------------------------------------------------------------
void npsAgent::addToFood(char *name)
{
   knownFood.insert(knownFood.end(), name);
}


//---------------------------------------------------------------------
//Function:   addToUnknown(char*)
//Return Val:
//Parameter:  char* - agentType
//Purpose:    will add the agentType to vector of known unknown agents
//---------------------------------------------------------------------
void npsAgent::addToUnknown(char *name)
{
   unknownAgents.insert(unknownAgents.end(), name);
}


//---------------------------------------------------------------------
// Function:   npsAgent::myRand ()
// Return Val: double - a pseudorandom number between 0.0 and 1.0
// Parameter:
// Purpose:    return random number between 0.0 and 1.0
//---------------------------------------------------------------------
double npsAgent::myRand ()
{
   double randomNumber;

   randomNumber = rand()/double(RAND_MAX);

   return randomNumber;

}//end Animal::myRand()


//---------------------------------------------------------------------
// Method:      integerToString()
// Parameters:  int number - number id which ever animal is passed
// Return val:  char*
// Purpose:     Can be used by agent classes to convert integers to string
//              values
//---------------------------------------------------------------------
char* npsAgent::integerToString(int inNum)
{
   int divisor = 1;
   int digits  = 1;
   int quotient;

   if (inNum / 10000 > 0)
   {
      divisor = 10000;
      digits = 5;
   }
   else if (inNum / 1000 > 0)
   {
      divisor = 1000;
      digits = 4;
   }
   else if (inNum / 100 > 0)
   {
      divisor = 100;
      digits = 3;
   }
   else if (inNum / 10 > 0)
   {
      divisor = 10;
      digits = 2;
   }
   else if (inNum / 1 > 0)
   {
      divisor = 1;
      digits = 1;
   }

   char outNum [64];
   strcpy(outNum, "");
   for (int ix=0; ix<digits; ix++){
      switch (quotient = (int) inNum/divisor){
        case 0:
      strcat(outNum, "0");
          break;
        case 1:
      strcat(outNum, "1");
          break;
        case 2:
      strcat(outNum, "2");
          break;
        case 3:
      strcat(outNum, "3");
          break;
        case 4:
      strcat(outNum, "4");
          break;
        case 5:
      strcat(outNum, "5");
          break;
        case 6:
      strcat(outNum, "6");
          break;
        case 7:
      strcat(outNum, "7");
          break;
        case 8:
      strcat(outNum, "8");
          break;
        case 9:
      strcat(outNum, "9");
          break;
      }

      inNum -= quotient*divisor;
      divisor /= 10;
   }
      strcat(outNum, "  ");

      return (&outNum[0]);
}//end integerToString()


// ********************************************************************
//   EXECUTIVE SUMMARY
// Module Name: agentDisplayApp.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Declaration of class that creates an openGL window to
//              display agents in world
//
// March 1999 Master Thesis
// ********************************************************************

#ifndef _agentDisplayApp_h
#define _agentDisplayApp_h

// ********************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ********************************************************************

void initAgentDisplayApp();

#endif // _agentDisplayApp_h
```

```
// *******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: agentDisplayApp.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Implementation of class that creates an openGL window to
//              display agents in world
//
// March 1999 Master Thesis
// *******************************************************************

// *******************************************************************
//   INCLUDES AND EXTERNS
// *******************************************************************


#include "agentDisplayApp.h"
#include "bbGlobals.h"
#include "npsVisual.h"
#include "npsWindow.h"
#include "npsViewport.h"
#include "npsFlyingCamera.h"
#include "npsKeyboard.h"
#include "bbEventResponse.h"
#include "bbCallback.h"
#include "npsGeometry.h"

#include <math.h>
#include <GL/gl.h>


// *******************************************************************
//   DEFINES & FILE SCOPE VARIABLES
// *******************************************************************

  bbCallback        *callback;
  npsWindow         *window;
  npsCamera         *camera;
  npsVec3f           position;
  npsQuaternion      rotation;
  npsViewport       *viewport;
  npsGeometry       *boid1;


// *******************************************************************
//   CODE
// *******************************************************************

void initAgentDisplayApp()
{
  void initKeyboardModule();
  void initVisualModule();

  initKeyboardModule();
  initVisualModule();
}//end initAgentDisplayApp()

void initKeyboardModule()
{
  void              escFunc(void *object, bbData *data);
  void              resetFunc(void *object, bbData *data);
  void              sideViewFunc(void *object, bbData *data);
  void              topViewFunc(void *object, bbData *data);
  npsKeyboard      *keyboard;
  bbEventResponse  *eventResponse;
  bbCallback       *callback;

  // get the keyboard device
  keyboard = npsKeyboard::getInstance();

  // set up exit key
  eventResponse = new bbEventResponse(npsKeyboard::KEY_ESC |
                                      npsKeyboard::UP_TRANS);
  callback = new bbCallback();
  callback->setFunc(escFunc);
  eventResponse->addCallbackLast(callback);
  keyboard->addEventResponse(eventResponse);

  // set up reset key
  eventResponse = new bbEventResponse(npsKeyboard::KEY_SPACE |
                                      npsKeyboard::UP_TRANS);
  callback = new bbCallback();
  callback->setFunc(resetFunc);
  eventResponse->addCallbackLast(callback);
  keyboard->addEventResponse(eventResponse);

  // set up side view looking down from above key
  eventResponse = new bbEventResponse(npsKeyboard::KEY_T |
                                      npsKeyboard::UP_TRANS);
  callback = new bbCallback();
  callback->setFunc(sideViewFunc);
  eventResponse->addCallbackLast(callback);
  keyboard->addEventResponse(eventResponse);

  // set up top down view key
  eventResponse = new bbEventResponse(npsKeyboard::KEY_T |
                                      npsKeyboard::CTRL_MASK |
                                      npsKeyboard::UP_TRANS);
  callback = new bbCallback();
  callback->setFunc(topViewFunc);
  eventResponse->addCallbackLast(callback);
  keyboard->addEventResponse(eventResponse);
}//end initKeyboardFunc()


void initVisualModule()
{
  void initCheckerboardFunc(void *object, bbData *data);

  // init terrain geometry
  new npsGeometry(initCheckerboardFunc);

  // open a window, viewport, and camera/ownship
  window = new npsWindow(800, 600);
    viewport = new npsViewport(0.0f, 1.0f, 0.0f, 1.0f);
      camera = new npsFlyingCamera(npsFlyingCamera::MOUSE);
      camera->setName("camera1");
      camera->setFarClip(400.0f);
      camera->setGeometry(boid1);
      position.set(0.0f, 3.0f, -10.0f);
      camera->setPosition(position);
      rotation.setEulers(NPS_DEG2RAD(180.0f),0.0f,0.0f);
      camera->setOrientation(rotation);
      camera->setClearColor(0.66f, 0.66f, 1.0f, 1.0f);
    viewport->setCamera(camera);
  window->addViewport(viewport);
}//end initCheckerboardFunc()


void escFunc(void *object, bbData *data)
{
  exit(0);
}//end escFunc()


void resetFunc(void *object, bbData *data)
{
  npsVec3f        initPosition;
  npsQuaternion   initRotation;

  initPosition.set(0.0f, 3.0f, -10.0f);
  initRotation.setEulers(NPS_DEG2RAD(180.0f),0.0f,0.0f);
  camera->setPosition(initPosition);
  camera->setOrientation(initRotation);
}//end resetFunc()


void sideViewFunc(void *object, bbData *data)
{
  npsVec3f        initPosition;
  npsQuaternion     initRotation;

  initPosition.set(0.0f, 50.0f, 100.0f);
  initRotation.setEulers(0.0f,NPS_DEG2RAD(-30.0f),0.0f);
  camera->setPosition(initPosition);
  camera->setOrientation(initRotation);
}//end sideViewFunc()


void topViewFunc(void *object, bbData *data)
{
  npsVec3f        initPosition;
  npsQuaternion   initRotation;

  initPosition.set(0.0f, 150.0f, 0.0f);
  initRotation.setEulers(0.0f,NPS_DEG2RAD(-90.0f),0.0f);
  camera->setPosition(initPosition);
  camera->setOrientation(initRotation);
}//end topViewFunc()


void initCheckerboardFunc(void *object, bbData *data)
{
  npsGeometry   *geometry;
  u_int          displayListNum;
  const float    CELL_LENGTH = 5.0;
  const u_int    NUM_CELLS_LONG = 25;
  const u_int    NUM_CELLS_WIDE = 25;
  const u_int    NUM_VERTS_LONG = NUM_CELLS_LONG + 1;
  const u_int    NUM_VERTS_WIDE = NUM_CELLS_WIDE + 1;
  const u_int    TOTAL_NUM_VERTS = NUM_VERTS_LONG * NUM_VERTS_WIDE;
  u_int       i, j, currVert;
  bool        colorToggle;

  GLfloat     coords[TOTAL_NUM_VERTS][3];

  // init Vals
  colorToggle = 0;
  for (i=0; i<NUM_VERTS_LONG; i++)
    {
    for (j=0; j<NUM_VERTS_WIDE; j++)
      {
      currVert = i*NUM_VERTS_WIDE + j;

      coords[currVert][0] = (CELL_LENGTH * i) -
                    (NUM_CELLS_LONG*CELL_LENGTH*0.5f);
      coords[currVert][1] = 0.0f;
      coords[currVert][2] = (-CELL_LENGTH * j) +
                    (NUM_CELLS_WIDE*CELL_LENGTH*0.5f);
      }
    }

    displayListNum = glGenLists(1);
    glNewList(displayListNum, GL_COMPILE);
    {
    glShadeModel(GL_FLAT);

    colorToggle = 0;
    for (i=0; i<NUM_CELLS_LONG; i++)
    {
      glBegin(GL_TRIANGLE_STRIP);
    {
    for (j=0; j<NUM_VERTS_WIDE; j++)
    {
      if (colorToggle)
      {
        colorToggle = 0;
        glColor3f(0.8f, 0.9f, 0.8f);
      }
      else
      {
        colorToggle = 1;
        glColor3f(0.85f, 0.95f, 0.85f);
      }

      currVert = i*NUM_VERTS_WIDE + j;
      glVertex3fv(coords[currVert]);

      currVert = (i+1)*NUM_VERTS_WIDE + j;
      glVertex3fv(coords[currVert]);
      }
      if ( NUM_CELLS_WIDE & 0x1 )
      {
```

```
        if (colorToggle)
          colorToggle = 0;
        else
          colorToggle = 1;
      }
    }
    glEnd();
    }
    glShadeModel(GL_SMOOTH);
    }
  glEndList();


  // set displaylist and remove callback func
  geometry = (npsGeometry*)object;
  geometry->setDisplayListNum(displayListNum);
  geometry->setCallbackFunc(0);

}//end initCheckerboardFunc()


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: animal.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Definition of the animal class agent for use in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

#ifndef _animal_h
#define _animal_h

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "npsAgent.h"
#include "npsAgentApi.h"
#include "npsVec3f.h"

// ******************************************************************
//   DEFINES
// ******************************************************************

enum DESIRED_ACTION {NOTHING, MATE, FEED, GATHER, AVOID, CHASE, FLEE};
enum MOVE_SPEED {REST, REGULAR, RUN};
enum DEATH_INDICATOR {INFANT_MORTALITY, OLD_AGE, PREDATION, STARVATION,
                      NOT_DEAD};

#define MALE            'M'
#define FEMALE          'F'
#define MATE_AGE         2
#define MATE_DISTANCE   .5
#define MOVE_INCREMENT  0.25

struct Pregnancy{
   int partnerId;
   int maleSpeed;
   int gestationTime;
   int seasonCounter;
};

class Animal;
#ifdef _Animal_c
  ACE_EXPORT_SINGLETON_DECLARATION(bbSafeClass<Animal>);
  ACE_EXPORT_SINGLETON_DECLARATION(bbListedClass<Animal>);
#else
  ACE_IMPORT_SINGLETON_DECLARATION(bbSafeClass<Animal>);
  ACE_IMPORT_SINGLETON_DECLARATION(bbListedClass<Animal>);
#endif

// ******************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

class AGENT_API Animal: public npsAgent{

private:

   DESIRED_ACTION  nextAction;
   MOVE_SPEED      speedOfNextMove;
   DEATH_INDICATOR deathIndicator;

   int generation,
       mateAge,
       deathCounter;

   npsVec3f moveToLocation,
            moveFromLocation;

   char gender,
        *killer;

   bool pregnant,
        inSeason,
        resting;

protected:

    //Constructor
    Animal(bbCallbackFunc *callbackFunc);


public:

    //Default Destructor - does nothing at this time
    ~Animal();

   //default move methods provided to all animals: X-Z planar
   void move();
   void moveTo(npsVec3f _position);
   void moveFrom(npsVec3f _position);

   //get and set the desired next action for the Animal
   DESIRED_ACTION getNextAction();
   void setNextAction (DESIRED_ACTION na);

   //get and set the choice of speed for next move
   MOVE_SPEED getSpeedOfNextMove();
   void setSpeedOfNextMove(MOVE_SPEED ms);

   //get and set reason for animals death
   DEATH_INDICATOR getDeathIndicator();
   void setDeathIndicator(DEATH_INDICATOR di);

   //return random Animal litter size based on upper and lower bounds
   int randomLitterSize(int lower, int upper);

   //return true if Animal dies as infant based on mortality rate provided
   bool diesAsInfant(double mortalityRate);

   //can the Animal mate
   bool canMate(Animal *potentialMate);
   void mate(Animal *mate);
   bool mateEligible(Animal *potentialMate);

   //get and set deathCounter
   int getDeathCounter();
   void setDeathCounter(int dc);

   //get and set generation of animal
   int getGeneration();
   void setGeneration(int g);

   //get and set location to move to
   npsVec3f  getMoveToLocation();
   void setMoveToLocation(npsVec3f mtl);

   //get and set location to move from
   npsVec3f getMoveFromLocation();
   void setMoveFromLocation(npsVec3f mfl);

   //get and set the mate age variable
   int getMateAge();
   void setMateAge(int ma);

   //test to see if one animal can kill another
   virtual bool isKilled(npsAgent &agent) = 0;

   //get and set the gender for an animal
   char getGender();
   void setGender(char g);

   //see if female is pregnant
    bool isPregnant();
   void setPregnant(bool p);

   //see if the animal is in season
   bool isInSeason();
   void setInSeason(bool is);

   //see if animal is dead
   bool isDead();

   //see if animal needs to rest
   bool isResting();
   void setRest(bool r);

   //main methods to let agents interact
   virtual void updatePosition(int time) = 0;
   virtual void sense(int time) = 0;

   void setKiller(char *k);
   char *getKiller();

   //pointer to Pregancy struct
   Pregnancy* pregPtr;

};


// ******************************************************************
//   INLINED MEMBER FUNCTIONS
// ******************************************************************

inline DESIRED_ACTION Animal::getNextAction()
{
   return nextAction;
}

inline void Animal::setNextAction (DESIRED_ACTION na)
{
   nextAction = na;
}

inline MOVE_SPEED Animal::getSpeedOfNextMove()
{
   return (speedOfNextMove);
}

inline void Animal::setSpeedOfNextMove(MOVE_SPEED ms)
{
   speedOfNextMove = ms;
}

inline npsVec3f  Animal::getMoveToLocation()
{
   return (moveToLocation);
}

inline void Animal::setMoveToLocation(npsVec3f mtl)
{
   moveToLocation = mtl;
}

inline npsVec3f Animal::getMoveFromLocation()
{
   return (moveFromLocation);
}

inline void Animal::setMoveFromLocation(npsVec3f mfl)
```

```
{
   moveFromLocation = mfl;
}


inline DEATH_INDICATOR Animal::getDeathIndicator()
{
   return (deathIndicator);
}

inline void Animal::setDeathIndicator(DEATH_INDICATOR di)
{
   deathIndicator = di;
}

inline int Animal::getDeathCounter()
{
   return (deathCounter);
}

inline void Animal::setDeathCounter(int dc)
{
   deathCounter = dc;
}

inline int Animal::getGeneration()
{
   return (generation);
}

inline void Animal::setGeneration(int g)
{
   generation = g;
}

inline int Animal::getMateAge()
{
   return (mateAge);
}

inline void Animal::setMateAge(int ma)
{
   mateAge = ma;
}

inline char Animal::getGender()
{
   return (gender);
}

inline void Animal::setGender(char g)
{
   gender = g;
}


inline bool Animal::isPregnant()
{
   return (pregnant);
}
inline void Animal::setPregnant(bool p)
{
   pregnant = p;
}

inline bool Animal::isInSeason()
{
   return (inSeason);
}

inline void Animal::setInSeason(bool is)
{
   inSeason = is;
}

inline bool Animal::isDead()
{
   return (deathIndicator != NOT_DEAD);
}

inline bool Animal::isResting()
{
   return (resting);
}

inline void Animal::setRest(bool r)
{
   resting = r;
}

inline void Animal::setKiller(char *k)
{
   killer = k;
}

inline char * Animal::getKiller()
{
   return(killer);
}

#endif // _Animal


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: animal.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Implementation of the animal class agent used in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

#define _animal_c
```

```
// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "Animal.h"
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <ctime>

//******************************************************************
// DEFINES AND FILE SCOPE CONSTANTS
//******************************************************************

static int  numAnimal = 0;

//------------------------------------------------------------------
// Function:   Animal::Animal()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//------------------------------------------------------------------
Animal::Animal (bbCallbackFunc *_callbackFunc)
   :npsAgent(_callbackFunc ), nextAction(NOTHING),
   speedOfNextMove(REGULAR), deathIndicator(NOT_DEAD), pregPtr(NULL),
   pregnant(false), inSeason(false), generation(1), resting(false)
{

}//end Animal::Animal()


//------------------------------------------------------------------
// Function:   Animal::~Animal()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//------------------------------------------------------------------
Animal::~Animal ()
{
   //do nothing at this point
}//end Animal::~Animal()



//------------------------------------------------------------------
// Function:   Animal::move ()
// Return Val: void
// Parameter:  None
// Purpose:    Provides the basic movement in the XZ plane which should
//             suffice for most animals.  This can be overloaded in a sub-
//             class if needed
//------------------------------------------------------------------
void Animal::move()
{
   int changeX = 0,
       changeZ = 0;

   float  hpr[3];
   npsVec3f        tempPosition;
   npsQuaternion   tempRotation;

   this->getPosition(tempPosition);
   tempRotation.getEulers(hpr);

   if(this->getSpeedOfNextMove() != REST)
   {
      float tempX = tempPosition[X],
            tempY = tempPosition[Y],
            tempZ = tempPosition[Z];

      double randX = npsAgent::myRand();
//    double randY = npsAgent::myRand(); //don't need to change altitude
      double randZ = npsAgent::myRand();

      if (randX <= 0.5)
      {
         tempX -= MOVE_INCREMENT;
         changeX = -1;
      }
      else
      {
         tempX += MOVE_INCREMENT;
         changeX = 1;
      }

      if (randZ <= 0.5)
      {
         tempZ -= MOVE_INCREMENT; // this moves the animal up one row
         changeZ = -1;
      }
      else
      {
         tempZ += MOVE_INCREMENT; // this moves the animal down one row
         changeZ = 1;
      }

      if(tempX <= MIN_X)
         tempX = MIN_X + 1;//bring the animal back one unit
      if(tempX >= MAX_X)
         tempX = MAX_X - 1;//bring the animal back one unit
      if(tempZ <= MIN_Z)
         tempZ = MIN_Z + 1; //move the animal down one row
      if(tempZ >= MAX_Z)
         tempZ = MAX_Z - 1; //move the animal up one row


      if(changeX > 0)
      {
         if(changeZ > 0)
            hpr[0] = NPS_DEG2RAD(45.0f); //+
         else if(changeZ < 0)
            hpr[0] = NPS_DEG2RAD(45.0f); //-
         else
            hpr[0] = 0;
      }
      else if (changeX < 0) //else
```

```
    {
       if(changeZ > 0)
          hpr[0] = NPS_DEG2RAD(135.0f); //+
       else if(changeZ < 0)
          hpr[0] = NPS_DEG2RAD(135.0f); //-
       else
          hpr[0] = NPS_DEG2RAD(180.0f);
    }
    else
    {
       if (changeZ > 0)
          hpr[0] = NPS_DEG2RAD(90.0f);
       else if (changeZ < 0)
          hpr[0] = NPS_DEG2RAD(90.0f);
       else
          hpr[0] = 0;
    }
    tempRotation.setEulers(hpr);
    this->setOrientation(tempRotation);

    tempPosition.set(tempX, tempY, tempZ);
    this->setPosition(tempPosition);
  }//end if not at REST

  return;

}//end Animal::move()


//----------------------------------------------------------------------
// Function:   Animal::moveTo ()
// Return Val: void
// Parameter:  None
// Purpose:    Provides the basic movement to a position in the XZ plane
//             which should suffice for most animals.  This can be
//             overloaded in a subclass if needed
//----------------------------------------------------------------------
void Animal::moveTo(npsVec3f position)
{
  int changeX = 0,
      changeZ = 0;

  float  hpr[3];

  npsVec3f         tempPosition;
  npsQuaternion    tempRotation;

  this->getPosition(tempPosition);
  tempRotation.getEulers(hpr);

  float moveToX = position[X],
        moveToY = position[Y],
        moveToZ = position[Z],
        thisTempX = tempPosition[X],
        thisTempY = tempPosition[Y],
        thisTempZ = tempPosition[Z];

  if(this->getSpeedOfNextMove() == REGULAR)
  {
     if((thisTempX - moveToX) > 0)
     {
        thisTempX -= MOVE_INCREMENT;
        changeX = -1;
     }
     else if((thisTempX - moveToX) < 0)
     {
        thisTempX += MOVE_INCREMENT;
        changeX = 1;
     }

     if((thisTempZ - moveToZ) > 0)
     {
        thisTempZ -= MOVE_INCREMENT;
        changeZ = -1;
     }
     else if((thisTempZ - moveToZ) < 0)
     {
        thisTempZ += MOVE_INCREMENT;
        changeZ = 1;
     }
  }//end if
  else//RUN
  {
     if((thisTempX - moveToX) > 0)
     {
        thisTempX -= (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeX = -1;
     }
     else if((thisTempX - moveToX) < 0)
     {
        thisTempX += (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeX = 1;
     }

     if((thisTempZ - moveToZ) > 0)
     {
        thisTempZ -= (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeZ = -1;
     }
     else if((thisTempZ - moveToZ) < 0)
     {
        thisTempZ += (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeZ = 1;
     }
  }//end else

  //check for going out of bounds
  if(thisTempX <= MIN_X)
     thisTempX = MIN_X + 1;//bring the animal back one unit
  if(thisTempX >= MAX_X)
     thisTempX = MAX_X - 1;//bring the animal back one unit
  if(thisTempZ <= MIN_Z)
     thisTempZ = MIN_Z + 1; //move the animal down one row
  if(thisTempZ >= MAX_Z)
     thisTempZ = MAX_Z - 1; //move the animal up one row

     //set orientation
  if(changeX > 0)//plusX)
  {
     if(changeZ > 0)//plusZ) //x=1,z=1
        hpr[0] = NPS_DEG2RAD(45.0f); //+
     else if(changeZ < 0)//x=1,z=-1
        hpr[0] = NPS_DEG2RAD(45.0f); //-
     else
        hpr[0] = 0;
  }
  else if (changeX < 0) //else
  {
     if(changeZ > 0) //plusZ) //x=-1,z=1
        hpr[0] = NPS_DEG2RAD(135.0f); //+
     else if(changeZ < 0) //x=-1,z=-1
        hpr[0] = NPS_DEG2RAD(135.0f); //-
     else//x=-1,z=0
        hpr[0] = NPS_DEG2RAD(180.0f);
  }
  else//x=0
  {
     if (changeZ > 0)//x=0,z=1
        hpr[0] = NPS_DEG2RAD(90.0f);
     else if (changeZ < 0)//x=0,z=-1
        hpr[0] = NPS_DEG2RAD(90.0f);
     else
        hpr[0] = 0;
  }

  tempPosition.set(thisTempX, thisTempY, thisTempZ);
  tempRotation.setEulers(hpr);
  this->setPosition(tempPosition);
  this->setOrientation(tempRotation);

  return;

}//end Animal::moveTo()


//----------------------------------------------------------------------
// Function:   Animal::moveFrom()
// Return Val: void
// Parameter:  None
// Purpose:    Provides the basic movement from a position in the XZ plane
//             which should suffice for most animals.  This can be
//             overloaded in a subclass if needed
//----------------------------------------------------------------------
void Animal::moveFrom(npsVec3f _position)
{
  int changeX = 0,
     changeZ = 0;

  float  hpr[3];

  npsVec3f        tempPosition;
   npsQuaternion tempRotation;

  this->getPosition(tempPosition);
  tempRotation.getEulers(hpr);

  float moveToX = _position[X],
        moveToY = _position[Y],       //don't need for now
        moveToZ = _position[Z],
        thisTempX = tempPosition[X],
        thisTempY = tempPosition[Y],  //don't need for now
        thisTempZ = tempPosition[Z];

  if(this->getSpeedOfNextMove() == REGULAR)
  {
     if((thisTempX - moveToX) > 0)
     {
        thisTempX += MOVE_INCREMENT;
        changeX = 1;
     }
     else if((thisTempX - moveToX) < 0)
     {
        thisTempX -= MOVE_INCREMENT;
        changeX = -1;
     }

     if((thisTempZ - moveToZ) > 0)
     {
        thisTempZ += MOVE_INCREMENT;
        changeZ = 1;
     }
     else if((thisTempZ - moveToZ) < 0)
     {
        thisTempZ -= MOVE_INCREMENT;
        changeZ = -1;
     }
  }//end if
  else//RUN
  {
     if((thisTempX - moveToX) > 0)
     {
        thisTempX += (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeX = 1;
     }
     else if((thisTempX - moveToX) < 0)
     {
        thisTempX -= (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeX = -1;
     }

     if((thisTempZ - moveToZ) > 0)
     {
        thisTempZ += (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeZ = 1;
     }
     else if((thisTempZ - moveToZ) < 0)
     {
        thisTempZ -= (MOVE_INCREMENT + 0.05*this->getSpeed());
        changeZ = -1;
     }
  }//end else

  //check for going out of bounds
  if(thisTempX <= MIN_X)
     thisTempX = MIN_X + 1;//bring the animal back one unit
  if(thisTempX >= MAX_X)
```

```
      thisTempX = MAX_X - 1;//bring the animal back one unit
   if(thisTempZ <= MIN_Z)
      thisTempZ = MIN_Z + 1; //move the animal down one row
   if(thisTempZ >= MAX_Z)
      thisTempZ = MAX_Z - 1; //move the animal up one row

   //set orientation
   if(changeX > 0)//plusX)
   {
      if(changeZ > 0)//plusZ)
         hpr[0] = NPS_DEG2RAD(45.0f); //+
      else if(changeZ < 0)
         hpr[0] = NPS_DEG2RAD(45.0f); //-
      else
         hpr[0] = 0;
   }
   else if (changeX < 0) //else
   {
      if(changeZ > 0) //plusZ)
         hpr[0] = NPS_DEG2RAD(135.0f); //+
      else if(changeZ < 0)
         hpr[0] = NPS_DEG2RAD(135.0f); //-
      else
         hpr[0] = NPS_DEG2RAD(180.0f);
   }
   else
   {
      if (changeZ > 0)
         hpr[0] = NPS_DEG2RAD(90.0f);
      else if (changeZ < 0)
         hpr[0] = NPS_DEG2RAD(90.0f);
      else
         hpr[0] = 0;
   }

   tempPosition.set(thisTempX, thisTempY, thisTempZ);
   tempRotation.setEulers(hpr);
   this->setPosition(tempPosition);
   this->setOrientation(tempRotation);

   return;

}//end Animal::moveFrom()


//-------------------------------------------------------------------
// Function:    bool Animal::mateEligible()
// Return Val:
// Parameter:
// Purpose:     return whether Animal is eligible to mate
//-------------------------------------------------------------------
bool Animal::mateEligible(Animal &potentialMate)
{
   bool mateEligibleFlag = false;

   if (this->getAgentType() == potentialMate.getAgentType())
   {
      if((this->getGender() == MALE) &&
         (potentialMate.getGender() == FEMALE))
      {
         mateEligibleFlag = (!(potentialMate.isPregnant()) &&
                  (potentialMate.isInSeason()) &&
                  (potentialMate.getAge() >= potentialMate.getMateAge())&&
                  (this->getAge() >= this->getMateAge()));
      }
      else if((this->getGender() == FEMALE) &&
            (potentialMate.getGender() == MALE))
      {
         mateEligibleFlag = (!(this->isPregnant()) &&
                  (this->isInSeason()) &&
                  (potentialMate.getAge() >= potentialMate.getMateAge())&&
                  (this->getAge() >= this->getMateAge()));
      }//end if else getGender()...

   }//end if getAgentType()

   return mateEligibleFlag;

}//end Animal::mateEligible


//-------------------------------------------------------------------
// Function:    bool Animal::canMate()
// Return Val:
// Parameter:
// Purpose:     return whether Animal can mate or not
//-------------------------------------------------------------------
bool Animal::canMate(Animal &potentialMate)
{
   bool mateFlag = false;

   if (this->getAgentType() == potentialMate.getAgentType())
   {
      if((this->getGender() == MALE) &&
         (potentialMate.getGender() == FEMALE))
      {
         mateFlag = ((!(potentialMate.isPregnant())) &&
                  (this->getNextAction() == MATE) &&
                  (potentialMate.getNextAction() == MATE)  &&
                  (potentialMate.getAge() >= potentialMate.getMateAge())&&
                  (this->getAge() >= this->getMateAge()) &&
                  (this->getDistance(potentialMate) <= MATE_DISTANCE));
      }
      else if((this->getGender() == FEMALE) &&
            (potentialMate.getGender() == MALE))
      {
         mateFlag = ((!(this->isPregnant())) &&
                  (this->getNextAction() == MATE) &&
                  (potentialMate.getNextAction() == MATE)  &&
                  (potentialMate.getAge() >= potentialMate.getMateAge())&&
                  (this->getAge() >= this->getMateAge()) &&
                  (this->getDistance(potentialMate) <= MATE_DISTANCE));
      }//end if getGender()

   }//end if getAgentType()

   return (mateFlag);
}


//-------------------------------------------------------------------
//Function:    Animal::mate ()
//Return Val:
//Parameter:   mate
//Purpose:     begin pregnancy once two animals mate
//-------------------------------------------------------------------
void Animal::mate(Animal &mate)
{
   if(this->getAgentType() == mate.getAgentType())
   {
      if(this->getGender() == MALE)
      {
         mate.setPregnant(true);
         mate.pregPtr->maleSpeed = this->getSpeed();
         mate.pregPtr->gestationTime = 0;
      }
      else
      {
         this->setPregnant(true);
         this->pregPtr->maleSpeed = mate.getSpeed();
         this->pregPtr->gestationTime = 0;
      }
   }

   return;
}//end function Animal::mate()


//-------------------------------------------------------------------
// Function:    randomLitterSize(int lower, upper)
// Return Val:  int - number in litter
// Parameter:   lower, upper
// Purpose:     return a random number of Animals in a litter bounded by
//              the upper and lower bounds provided
//-------------------------------------------------------------------

int Animal::randomLitterSize(int lower, int upper)
{
   return (npsAgent::myRand() * upper + lower);

}//end Animal::litterSize()


//-------------------------------------------------------------------
// Function:    diesAsInfant(double)
// Return Val:  bool
// Parameter:   mortalityRate
// Purpose:     return whether infant dies or not
//-------------------------------------------------------------------
bool Animal::diesAsInfant(double mortalityRate)
{
   double randNum = npsAgent::myRand();

   return (randNum < mortalityRate);

}//end Animal::mortality()
```

```
//end file Animal.c

// ******************************************************************
//    EXECUTIVE SUMMARY
// Module Name: antelope.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Definition of the anntelope class agent used in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

#ifndef _antelope_h
#define _antelope_h

// ******************************************************************
//    INCLUDES AND EXTERNS
// ******************************************************************

#include "npsAgentApi.h"
#include "animal.h"


// ******************************************************************
//    DEFINES
// ******************************************************************

#define INFANT_MORTALITY_RATE    0.50
#define REST_SENSING_RANGE       20
#define REGULAR_SENSING_RANGE    15
#define RUN_SENSING_RANGE        10
#define FRIEND_STANDOFF_DISTANCE 1.5
#define FOOD_RANGE               5
#define MAX_AGE                  3650
#define BEGIN_SEASON             30
#define END_SEASON               75
#define ONE_YEAR                 365
#define KILLED_RADIUS            0.15
#define KILL_PROBABILITY         0.7
#define ANTELOPE_GESTATION_PERIOD 60

// ******************************************************************
//    FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

class Antelope: public Animal{

private:

   int idNum,
       herdSize;

public:

    //Constructor
    Antelope(bbCallbackFunc *callbackFunc);
```

```
   //Constructor
   Antelope();

   //Default Destructor - does nothing at this time
   ~Antelope();

   //produce a newborn Antelope from a male/female pair
   Antelope* giveBirth(int motherSpeed, int fatherSpeed,
                       int motherGeneration, npsVec3f motherLocation);

   //get antelope identification number
   int getIdNum();

   //set and get herd size
   int getHerdSize();
   void setHerdSize(int hs);

   //can the Antelope mate
   bool canMate(Antelope *potentialMate);
   void mate(Antelope *mate);

   //are the Antelope mate eligible
   bool mateEligible(Antelope *potentialMate);

   //test to see if antelope is killed by predator
   bool isKilled(npsAgent &agent);

   //allow antelope to move through the world
   void updatePosition(int time);

   //allow the antelope to sense the world
   void sense(int time);//npsAgent *sensedAgent);

   int litterSize();
   bool diesAsInfant();

   void sensePredators(npsAgent *agent, float &closestPredator);
   void senseFriendly(npsAgent *agent, float &closestFriend,
              float &closestPartner);
   void senseFood(npsAgent *agent, npsVec3f &closestFoodPosition);
   void senseEnemy(npsAgent *agent, float &closestUnknown);
   void senseUnknown(npsAgent *agent, float &closestUnknown);
};

// *******************************************************************
//   INLINED MEMBER FUNCTIONS
// *******************************************************************

inline int Antelope::getIdNum()
{
   return idNum;
}

inline int Antelope::getHerdSize()
{
   return herdSize;
}

inline void Antelope::setHerdSize(int hs)
{
   herdSize = hs;
}

#endif // _antelope_h


// *******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: antelope.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Implementation of the antelope class agent for use in
//              npsAgent
//
// March 1999 Master Thesis
// *******************************************************************


// *******************************************************************
//   INCLUDES AND EXTERNS
// *******************************************************************

#include "antelope.h"
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <ctime>
#include <math.h>
#include <GL/gl.h>

//*******************************************************************
//  DEFINES AND FILE SCOPE CONSTANTS
//*******************************************************************

void initGeomFunc(void *object, bbData *data);

static int  numAntelope = 0;

//-------------------------------------------------------------------
// Function:   Antelope::Antelope()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//-------------------------------------------------------------------
Antelope::Antelope ()
   :Animal(initGeomFunc), idNum(numAntelope++), herdSize(1)
{
   this->setAgentType("Antelope");

   double genderRand = npsAgent::myRand();

   if (genderRand < 0.5)
      this->setGender(MALE);
   else
   {
      this->setGender(FEMALE);
      this->pregPtr = new Pregnancy;
   }

   //assign a random max speed for the animal between 5..10
   int maxSpeed = myRand()*10;
   if (maxSpeed < 5)
      maxSpeed += 5;
   this->setSpeed(maxSpeed);

   this->setAge(int(myRand() * MAX_AGE));
   this->setMateAge(MATE_AGE);
   this->setSensingRange(REGULAR_SENSING_RANGE);
}//end Antelope::Antelope()

//-------------------------------------------------------------------
// Function:   Antelope::~Antelope()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//-------------------------------------------------------------------
Antelope::~Antelope ()
{
   //do nothing at this point
}//end Antelope::~Antelope()


//-------------------------------------------------------------------
// Function:   initGeomFunc()
// Return Val:
// Parameter:
// Purpose:    provides OpenGL calls from which Babmoo will draw antelope
//-------------------------------------------------------------------
void initGeomFunc(void *object, bbData *data)
{
  GLfloat   coords[4][3] = { { 0.0f, 0.0f, -0.5f}, // front
             {-0.3f, 0.0f, 0.5f},   // back left
             { 0.3f, 0.0f, 0.5f},   // back right
             { 0.0f, 0.4f, 0.5f} // top
               };

  glShadeModel(GL_FLAT);
  glBegin(GL_TRIANGLES);
   {
   glColor3f(0.75f, 0.5f, 0.75f);  // bottom
   glVertex3fv(coords[0]);
   glVertex3fv(coords[2]);
   glVertex3fv(coords[1]);

   glColor3f(0.75f, 0.5f, 0.5f);  // left
   glVertex3fv(coords[0]);
   glVertex3fv(coords[1]);
   glVertex3fv(coords[3]);

   glColor3f(0.5f, 0.5f, 0.75f);  // right
   glVertex3fv(coords[0]);
   glVertex3fv(coords[3]);
   glVertex3fv(coords[2]);

   glColor3f(0.75f, 0.5f, 0.75f);  // back
   glVertex3fv(coords[1]);
   glVertex3fv(coords[2]);
   glVertex3fv(coords[3]);
   }
  glEnd();
  glShadeModel(GL_SMOOTH);
}//end Antelope::initGeom()


//-------------------------------------------------------------------
//Function:   Antelope::giveBirth ()
//Return Val: Antelope
//Parameter:  male speed, female speed
//Purpose:    make a new Antelope with speed the average of it's parents
//-------------------------------------------------------------------
Antelope* Antelope::giveBirth(int motherSpeed, int fatherSpeed,
                    int motherGeneration, npsVec3f motherLocation)
{
   int   newSpeed;
   char  name[64];

   Antelope *newBorn;

   if(npsAgent::myRand() < .5)
      newSpeed = motherSpeed;
   else
      newSpeed = fatherSpeed;

   newBorn = new Antelope();

   strcpy(name, "Antelope");
   strcat(name, this->integerToString(newBorn->getIdNum()));
   newBorn->addToFriends(newBorn->getAgentType());
   newBorn->setName(name);

   //set values of newborn based on parents' information
   newBorn->setSpeed(newSpeed);
   newBorn->setPosition(motherLocation);
   newBorn->setGeneration(motherGeneration + 1);

   return newBorn;
}//end antelope::mate()


//-------------------------------------------------------------------
// Function:   bool Antelope::canMate()
// Return Val:
// Parameter:
// Purpose:    return whether Antelope can mate or not
//-------------------------------------------------------------------
bool Antelope::canMate(Antelope *potentialMate)
{
   bool mateFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate->getGender() == FEMALE))
```

```
   {
      mateFlag = ((!(potentialMate->isPregnant())) &&
                 (this->getNextAction() == MATE) &&
                 (potentialMate->getNextAction() == MATE)  &&
                 (this->getDistance(*potentialMate) <= MATE_DISTANCE));
   }
   else if((this->getGender() == FEMALE) &&
          (potentialMate->getGender() == MALE))
   {
      mateFlag = ((!(this->isPregnant())) &&
                 (this->getNextAction() == MATE) &&
                 (potentialMate->getNextAction() == MATE)  &&
                 (this->getDistance(*potentialMate) <= MATE_DISTANCE));
   }

   return mateFlag;
}//end canMate()


//--------------------------------------------------------------------
// Function:   bool Antelope::mateEligible()
// Return Val:
// Parameter:
// Purpose:    return whether Antelope is eligible to mate
//--------------------------------------------------------------------
bool Antelope::mateEligible(Antelope *potentialMate)
{
   bool mateEligibleFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate->getGender() == FEMALE))
   {
      mateEligibleFlag = (!(potentialMate->isPregnant()) &&
                  (potentialMate->isInSeason()) &&
                  (potentialMate->getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE));
   }
   else if((this->getGender() == FEMALE) &&
          (potentialMate->getGender() == MALE))
   {
      mateEligibleFlag = (!(this->isPregnant()) &&
                  (this->isInSeason()) &&
                  (potentialMate->getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE));
   }

   return mateEligibleFlag;
}//end Antelope::mateEligible()


//--------------------------------------------------------------------
//Function:    Antelope::mate ()
//Return Val:  true / false
//Parameter:   mate
//Purpose:     begin pregnancy once two antelope mate
//--------------------------------------------------------------------
void Antelope::mate(Antelope *mate)
{
   if(this->getGender() == MALE)
```

```
   {
      mate->setPregnant(true);
      mate->pregPtr->maleSpeed = this->getSpeed();
      mate->pregPtr->gestationTime = 0;
   }
   else
   {
      this->setPregnant(true);
      this->pregPtr->maleSpeed = mate->getSpeed();
      this->pregPtr->gestationTime = 0;
   }

   return;
}//end function Antelope::mate()


//--------------------------------------------------------------------
// Function:   updatePosition()
// Return Val:
// Parameter:
// Purpose:    allow the antelope to update position
//--------------------------------------------------------------------
void Antelope::updatePosition(int time)
{
   switch(this->getNextAction())
   {
      case MATE :
      {
         this->setSpeedOfNextMove(REGULAR);
         this->moveTo(this->getMoveToLocation());
         break;
      }
      case NOTHING :
      {
         if(rand() < RAND_MAX/2)
            this->setSpeedOfNextMove(REGULAR);
         else
            this->setSpeedOfNextMove(REST);
         this->move(); .
         break;
      }
      case FEED :
      {
         this->setSpeedOfNextMove(REGULAR);
         this->moveTo(this->getMoveToLocation());
         break;
      }
      case GATHER :
      {
         this->setSpeedOfNextMove(REGULAR);
         this->moveTo(this->getMoveToLocation());
         break;
      }
      case AVOID :
      {
         this->setSpeedOfNextMove(REGULAR);
         this->moveTo(this->getMoveFromLocation());
         break;
      }
```

```
      case FLEE :
      {
         this->setSpeedOfNextMove(RUN);
         this->moveFrom(this->getMoveFromLocation());
         break;
      }
      default :  //CHASE
      {
         this->setSpeedOfNextMove(RUN);
         this->moveFrom(this->getMoveToLocation());
         break;
      }

   }//end switch getNextAction()

   if(this->isPregnant())
      {
         if(this->pregPtr->gestationTime == ANTELOPE_GESTATION_PERIOD)
         {
            int litter = this->litterSize();
            npsVec3f tempLocation;
            this->getPosition(tempLocation);

            Antelope *babyAntelope;
            for (int ix = 1; ix <= litter; ix++)
            {
               babyAntelope = this->giveBirth(this->getSpeed(),
                              this->pregPtr->maleSpeed,
                              this->getGeneration(), tempLocation);

               if(babyAntelope->diesAsInfant())
               {
                  babyAntelope->setDeathIndicator(INFANT_MORTALITY);
               }
         else
         {
            //do nothing at this time
         }
               if (babyAntelope->getGender() == FEMALE)
                  babyAntelope->pregPtr = new Pregnancy;
            }//end for (ix) - create litter of size litter

            this->setPregnant(false);
         }//end if pregancy gestation time > ANTELOPE_GESTATION_TIME
         else
         {
         this->pregPtr->gestationTime++;
         }//end else
      }//end if aix->isPregnant()

      this->growOlder();

      //check age and if over MAX_AGE then set deathIndicator to OLD_AGE
      if(this->getAge() == MAX_AGE)
      {
         this->setDeathIndicator(OLD_AGE);
      }

      //Last thing we do is check to make sure the antelope didn't die two
```

```
      //time steps ago.  if so take out of world elsee increment counter
      //this allows other animals to sense this one and learn how it died

      if((this->getDeathIndicator() != NOT_DEAD) &&
         (this->getDeathCounter() < 2))
      {
         this->setDeathCounter(this->getDeathCounter() + 1);
      }//end if getDeathIndicator()
      else if((this->getDeathIndicator() != NOT_DEAD) &&
             (this->getDeathCounter() >=2))
      {
         this->setRemove();
      }//end if else getDeathIndicator()
}//end updatePosition()


//--------------------------------------------------------------------
// Function:   sense()
// Return Val:
// Parameter:
// Purpose:    allow the antelope to sense environment and decide which
//             action to take next
//--------------------------------------------------------------------
void Antelope::sense(int time)//npsAgent *sensedAgent)
{
   if((time%ONE_YEAR > BEGIN_SEASON) && (time%ONE_YEAR < END_SEASON))
      this->setInSeason(true);
   else
      this->setInSeason(false);

   int currentSensingRange   = 0;

   float closestPartner      = 100,
         closestFriend       = 100,
         closestEnemy        = 100,
         closestUnknown      = 100,
         closestPredator     = 100;
   bool  sensedFood          = false;

   npsVec3f moveToLocation, closestFoodPosition;

   //initialize to large value to start with
   closestFoodPosition.set(MAX_X*5.0f, MAX_Y*5.0f, MAX_Z*5.0f);

   switch(this->getSpeedOfNextMove())
   {
      case REST :
      {
         currentSensingRange = REST_SENSING_RANGE;
         break;
      }
      case REGULAR :
      {
         currentSensingRange = REGULAR_SENSING_RANGE;
         break;
      }
      default://case RUN
      {
         currentSensingRange = RUN_SENSING_RANGE;
```

```
         break;
      }
   }//end switch

   this->setNextAction(NOTHING); //reset this for tracking

   int numAgents = bbListedClass<npsAgent>::getNumObjects();

   for (int j = 0; j < numAgents; j++)
   {
      npsAgent *sensedAgent = bbListedClass<npsAgent>::getObject(j);

      if ((this->getDistance(*sensedAgent) <= currentSensingRange) &&
         (this->getName() != sensedAgent->getName()))
      {
         switch (this->getRelationship(sensedAgent))
         {
         case PREDATOR:
            {
               sensePredators(sensedAgent, closestPredator);
               break;
            }
         case ENEMY:
            {
               if(this->getNextAction() != FLEE)
                  senseEnemy(sensedAgent, closestEnemy);
               break;
            }
         case FRIENDLY:
            {
               if(this->getNextAction() != FLEE)
               {
                  senseFriendly(sensedAgent, closestFriend,
                                closestPartner);
               }
               break;
            }
         case FOOD:
            {
               sensedFood = true;
               if(this->getNextAction() != FLEE)
               {
                  senseFood(sensedAgent, closestFoodPosition);
               }
               break;
            }
         default:  //case UNKNOWN:
            {
               senseUnknown(sensedAgent, closestUnknown);
               break;
            }
         }//end switch
      }//end if (distanceToAgent...)

   }//end for (j<numAgents)

   //if your not going to mate, if you haven't found found a
   //friend or 1/2 the time when you have found a friend antelope
   //will move to food anyway
```

```
   if((this->getNextAction() != MATE) &&
      (this->getNextAction() != FLEE) && sensedFood)
   {
      if((this->getNextAction() != GATHER) || (npsAgent::myRand() < 0.5))
      {
         if (this->getDistanceFromLocation(closestFoodPosition) < FOOD_RANGE)
            this->setNextAction(NOTHING);
         else
         {
            this->setMoveToLocation(closestFoodPosition);
            this->setNextAction(FEED);
         }//end if else getDistance
      }//end if nextAction != Gather
   }//end if nextAction != MATE...

}//end Antelope::sense()


//-------------------------------------------------------------------
// Function:   sensePredators()
// Return Val:
// Parameter:  npsAgent, float
// Purpose:    allow agent to sense a known predator and decide what to
//             next
//-------------------------------------------------------------------
void Antelope::sensePredators(npsAgent *agent, float &closestPredator)
{
   npsVec3f moveFromLocation;

   if(this->isKilled(*agent))
   {
      this->setRemove();
      this->setDeathIndicator(PREDATION);
      this->setKiller(agent->getAgentType());
   }
   else if(this->getDistance(*agent) < closestPredator)
   {
      closestPredator = this->getDistance(*agent);
      agent->getPosition(moveFromLocation);
      this->setMoveFromLocation(moveFromLocation);
      this->setNextAction(FLEE);
   }//end if

}//end sensePredator


//-------------------------------------------------------------------
// Function:   senseFriendly()
// Return Val:
// Parameter:  npsAgent, float, float
// Purpose:    allow agent to sense a known predator and decide what to
//             next
//-------------------------------------------------------------------
void Antelope::senseFriendly(npsAgent *agent, float &closestFriend,
                             float &closestPartner)
{
   float distanceToAgent = this->getDistance(*agent);

   npsVec3f moveToLocation;
```

```
   if (this->isSameAgentType(agent))
   {
      if (this->canMate((Antelope *)agent))
      {
         this->mate((Antelope *)agent);
         this->setNextAction(FEED);  //should feed after mating
      }//end ifcanMate()
      else if((this->mateEligible((Antelope *)agent)) &&
              (distanceToAgent < closestPartner))
      {
         closestPartner = distanceToAgent;
         agent->getPosition(moveToLocation);
         this->setMoveToLocation(moveToLocation);
         this->setNextAction(MATE);
      }//end else if mateEligible(sensedAgent)
      else if(this->getNextAction() != MATE)
      {
         if((distanceToAgent < closestFriend) &&
            (distanceToAgent > FRIEND_STANDOFF_DISTANCE))
         {
            closestFriend = distanceToAgent;
            agent->getPosition(moveToLocation);
            this->setMoveToLocation(moveToLocation);
            this->setNextAction(GATHER);
         }//end if(getDistance()
      }//end else if (getNextAction)
   }//end if isSameAgent()
   else
   {
      //no interaction defined for other friendly agents at this time
   }//end if/else(isSameAgentType)
}//end senseFriendly()


//-------------------------------------------------------------------
// Function:   senseFood()
// Return Val:
// Parameter:  npsAgent, npsVec3f
// Purpose:    allow agent to sense a known food source and remember if
//             it is the closest one
//-------------------------------------------------------------------
void Antelope::senseFood(npsAgent *agent, npsVec3f &closestFoodPosition)
{
   if (this->getDistance(*agent) <
       this->getDistanceFromLocation(closestFoodPosition))
   {
      agent->getPosition(closestFoodPosition);
   }//end if

}//end senseFood


//-------------------------------------------------------------------
// Function:   senseEnemy()
// Return Val:
// Parameter:  npsAgent, int
// Purpose:    allow agent to sense a known predator and decide what to
//             next
```

```
//-------------------------------------------------------------------
void Antelope::senseEnemy(npsAgent *agent, float &closestEnemy)
{
   //do nothing for Enemies at this time
}//end sensePredator


//-------------------------------------------------------------------
// Function:   senseUnknown()
// Return Val:
// Parameter:  npsAgent, int
// Purpose:    allow agent to sense an unknown agent and decide how to
//             interact with it
//-------------------------------------------------------------------
void Antelope::senseUnknown(npsAgent *agent, float &closestUnknown)
{
   if(!(strcmp(agent->getAgentType(),"Grass")))
   {
      this->addToFood(agent->getAgentType());
   }
   else if(!(strcmp(agent->getAgentType(),"Cheetah")))
   {
      this->addToPredators(agent->getAgentType());
   }
   else
   {
      this->addToUnknown(agent->getAgentType());
   }

}//end sensePredator


//-------------------------------------------------------------------
// Function: int Antelope::litterSize()
// Return Val:  int number in litter
// Parameter:
// Purpose:    return a random number of antelope in a litter
//-------------------------------------------------------------------

int Antelope::litterSize()
{
   int litter = 1;

   if(npsAgent::myRand() >= 0.9)
      litter = 2;

   return litter;

}//end Antelope::litterSize()


//-------------------------------------------------------------------
// Function:   bool Antelope::diesAsInfant()
// Return Val:
// Parameter:
// Purpose:    return whether infant dies or not
//-------------------------------------------------------------------
bool Antelope::diesAsInfant()
{
```

```
   double randNum = npsAgent::myRand();

   return (randNum < INFANT_MORTALITY_RATE);

 }//end Antelope::mortality()


//-------------------------------------------------------------------
// Function:   isKilled(Animal)
// Return Val: bool
// Parameter:
// Purpose:    every animal must be able to determine if it has been
//             killed by another animal
//-------------------------------------------------------------------
bool Antelope::isKilled(npsAgent &agent)
{
   bool killFlag = false;

   if((this->getDeathIndicator() == NOT_DEAD) &&
      (this->getRelationship(&agent) == PREDATOR) &&
      (this->getDistance(agent) <= KILLED_RADIUS) &&
      (npsAgent::myRand() < KILL_PROBABILITY))
   {
      cout<<"testing antelope isKilled by "<<agent.getName()<<endl;
      killFlag = true;
   }
   return killFlag;
}
//end file Antelope.c


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: antelopeApp.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: The application class for the Antelope Module - used
//              to instantiate Antelope agents when requested
//
// March 1999 Master Thesis
// ******************************************************************

#ifndef _antelopeApp_h
#define _antelopeApp_h

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "antelope.h"


// ******************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

void initAntelopeApp();
```

```
void exitAntelopeApp();

// ******************************************************************
//   INLINED MEMBER FUNCTIONS
// ******************************************************************

#endif // _AntelopeApp_h


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: antelopeApp.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: The application class for the Antelope Module - used
//              to instantiate Antelope agents when requested
//
// March 1999 Master Thesis
// ******************************************************************

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "antelopeApp.h"
#include "bbModule.h"
#include "bbThread.h"
#include "bbCallback.h"
#include "npsKeyboard.h"
#include "bbEventResponse.h"


#include <math.h>
#include <GL/gl.h>

// ******************************************************************
//   DEFINES & FILE SCOPE VARIABLES
// ******************************************************************

bbThread    *thread;
static int   numAntelope = 0;

// ******************************************************************
//   CODE
// ******************************************************************

void initKeyboardModule();
void initAntelopeFunc(int numAntelope);
void initBoidFunc(void *object, bbData *data);
char* integerToString(int inNum);


void initAntelopeApp()
{
   int numAntelope = 0;

   initKeyboardModule();
```

```
}//end initAntelopeApp()

void exitNpsAgentApp()
{
   //do nothing for now - should remove antelope if desired
}//end exitAntelopeApp()

void initKeyboardModule()
  {
  void               getNumFunc(void *object, bbData *data);
  npsKeyboard        *keyboard;
  bbEventResponse    *eventResponse;
  bbCallback         *callback;

  // get the keyboard device
  keyboard = npsKeyboard::getInstance();

  // set up get number of antelope by using the 'a' key
  eventResponse = new bbEventResponse(npsKeyboard::KEY_A |
                                      npsKeyboard::UP_TRANS);
  callback = new bbCallback();
  callback->setFunc(getNumFunc);
  eventResponse->addCallbackLast(callback);
  keyboard->addEventResponse(eventResponse);
  }

void  getNumFunc(void *object, bbData *data)
{
   cout <<"How many antelope would you like to create? " << endl;
   cin >> numAntelope;

   initAntelopeFunc(numAntelope);
}

void initAntelopeFunc(int numAntelope)
{
   Antelope *myAntelope;
   npsVec3f     position;

   for (int i = 0; i < numAntelope; i++)
   {
      char  name[64];

      myAntelope = new Antelope();

      strcpy(name, "Antelope");
      strcat(name, myAntelope->integerToString(myAntelope->getIdNum()));

      myAntelope->setRandomPosition();
      myAntelope->addToFriends(myAntelope->getAgentType());

      myAntelope->setName(name);
      cout<<"name = "<<name<<"type "<<myAntelope->getAgentType()<<endl;
   }
}
//end antelopeApp.c

// ******************************************************************
```

```
//   EXECUTIVE SUMMARY
// Module Name: cheetah.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Definition of the cheetah agent for use in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

#ifndef _cheetah_h
#define _cheetah_h

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "npsAgentApi.h"
#include "animal.h"

// ******************************************************************
//   DEFINES
// ******************************************************************

#define INFANT_MORTALITY_RATE    .80
#define REST_SENSING_RANGE       50
#define REGULAR_SENSING_RANGE    35
#define RUN_SENSING_RANGE        20
#define AVOID_DISTANCE           150
#define MAX_AGE                  3650
#define BEGIN_SEASON             30
#define END_SEASON               75
#define ONE_YEAR                 365
#define GESTATION_PERIOD         60
#define ENERGY_BOOST             200
#define HIGH_ENERGY_LEVEL        1200
#define STOP_HUNTING_LEVEL       200
#define RESUME_HUNTING_LEVEL     800
#define REGULAR_ENERGY_PENALTY   4
#define RUN_ENERGY_PENALTY       10
#define REST_ENERGY_GAIN         4
#define AVOID_RANGE              10
#define KILLED_RADIUS            0.10
#define KILL_PROBABILITY         0.5


// ******************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

class Cheetah: public Animal{

private:

   int idNum;

protected:
```

```
public:

    //Constructor
    Cheetah(bbCallbackFunc *callbackFunc);

    //Constructor
    Cheetah();

    //Default Destructor - does nothing at this time
    ~Cheetah();

   //produce a newborn Cheetah from a male/female pair
   Cheetah* giveBirth(int motherSpeed, int fatherSpeed,
                   int motherGeneration, npsVec3f motherLocation);

    //get cheetah identification number
   int getIdNum();

   //can the Cheetah mate
   bool canMate(Cheetah *potentialMate);
   void mate(Cheetah *mate);

   //are the Cheetah mate eligible
   bool mateEligible(Cheetah *potentialMate);

   //test to see if cheetah is killed by another agent
   bool isKilled(npsAgent &agent);

   //allow cheetah to move through the world
   void updatePosition(int time);

   //allow the cheetah to sense the world
   void sense(int time);

   int litterSize();
   bool diesAsInfant();

   //sense various types of agents
   void sensePredators(npsAgent *agent, float &closestPredator);
   void senseFriendly(npsAgent *agent, float &closestFriend,
              float &closestPartner);
   void senseFood(npsAgent *agent, float &closestFoodPosition);
   void senseEnemy(npsAgent *agent, float &closestUnknown);
   void senseUnknown(npsAgent *agent, float &closestUnknown);

};

// *****************************************************************
//   INLINED MEMBER FUNCTIONS
// *****************************************************************

inline int Cheetah::getIdNum()
{
   return idNum;
}

#endif // _cheetah_h
```

```
// *****************************************************************
//   EXECUTIVE SUMMARY
// Module Name: cheetah.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Implementation of the cheetah agent for use in npsAgent
//
// March 1999 Master Thesis
// *****************************************************************

// *****************************************************************
//   INCLUDES AND EXTERNS
// *****************************************************************

#include "cheetah.h"
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <ctime>
#include <math.h>
#include <GL/gl.h>


//*****************************************************************
//  DEFINES AND FILE SCOPE CONSTANTS
//*****************************************************************
void initGeomFunc(void *object, bbData *data);

static int  numCheetah = 0;

//*****************************************************************
//  CODE
//*****************************************************************

//-----------------------------------------------------------------
// Function:   Cheetah::Cheetah()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//-----------------------------------------------------------------
Cheetah::Cheetah ()
   :Animal(initGeomFunc), idNum(numCheetah++)
{
   this->setAgentType("Cheetah");

   double genderRand = npsAgent::myRand();

   if (genderRand < 0.5)
      this->setGender(MALE);
   else
   {
      this->setGender(FEMALE);
      this->pregPtr = new Pregnancy;
   }

   //assign a random max speed for the animal between 5..10
   int maxSpeed = myRand()*12;
   if (maxSpeed < 7)
      maxSpeed += 7;
   this->setSpeed(maxSpeed);

   this->setEnergyLevel(1200);

   this->setAge(int(myRand() * MAX_AGE));
   this->setMateAge(MATE_AGE);
   this->setSensingRange(REGULAR_SENSING_RANGE);

}//end Cheetah::Cheetah()


//-----------------------------------------------------------------
// Function:   Cheetah::~Cheetah()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//-----------------------------------------------------------------
Cheetah::~Cheetah ()
{
   //do nothing at this point
}//end Cheetah::~Cheetah()


//-----------------------------------------------------------------
// Function:   initGeomFunc()
// Return Val:
// Parameter:
// Purpose:    provides OpenGL calls from which Babmoo will draw  cheetah
//-----------------------------------------------------------------
void initGeomFunc(void *object, bbData *data)
{
  GLfloat    coords[4][3] = { { 0.0f, 0.0f, -0.6f}, // front
             {-0.4f, 0.0f, 0.6f},   // back left
             { 0.4f, 0.0f, 0.6f},   // back right
             { 0.0f, 0.4f, 0.6f} // top
                };

  glShadeModel(GL_FLAT);
  glBegin(GL_TRIANGLES);
    {
    glColor3f(0.0f, 0.0f, 0.0f);  // bottom
    glVertex3fv(coords[0]);
    glVertex3fv(coords[2]);
    glVertex3fv(coords[1]);

    glColor3f(0.0f, 0.0f, 0.0f);  // left
    glVertex3fv(coords[0]);
    glVertex3fv(coords[1]);
    glVertex3fv(coords[3]);

    glColor3f(0.0f, 0.0f, 0.0f);  // right
    glVertex3fv(coords[0]);
    glVertex3fv(coords[3]);
    glVertex3fv(coords[2]);

    glColor3f(0.0f, 0.0f, 0.0f);  // back
    glVertex3fv(coords[1]);
    glVertex3fv(coords[2]);
    glVertex3fv(coords[3]);
    }
  glEnd();
  glShadeModel(GL_SMOOTH);
}//end Cheetah::initGeomFunc()


//-----------------------------------------------------------------
//Function:   Cheetah::giveBirth ()
//Return Val: Cheetah
//Parameter:  male speed, female speed
//Purpose:    make a new Cheetah with speed of one of it's parents
//-----------------------------------------------------------------
Cheetah* Cheetah::giveBirth(int motherSpeed, int fatherSpeed,
                      int motherGeneration, npsVec3f motherLocation)
{
   int newSpeed;

   char  name[64];

   Cheetah *newBorn;

   if(npsAgent::myRand() < 0.5)
      newSpeed = motherSpeed;
   else
      newSpeed = fatherSpeed;

   newBorn = new Cheetah();

   strcpy(name, "Cheetah");
   strcat(name, this->integerToString(newBorn->getIdNum()));
   newBorn->addToFriends(newBorn->getAgentType());
   newBorn->setName(name);

   //set values of newborn based on parents' information
   newBorn->setSpeed(newSpeed);
   newBorn->setPosition(motherLocation);
   newBorn->setGeneration(motherGeneration + 1);

   return newBorn;
}//end cheetah::giveBirth()


//-----------------------------------------------------------------
// Function:   bool Cheetah::canMate()
// Return Val:
// Parameter:
// Purpose:    return whether Cheetah can mate or not
//-----------------------------------------------------------------
bool Cheetah::canMate(Cheetah *potentialMate)
{
   bool mateFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate->getGender() == FEMALE))
   {
```

```
      mateFlag = ((!(potentialMate->isPregnant())) &&
                  (this->getNextAction() == MATE) &&
                  (potentialMate->getNextAction() == MATE)  &&
                  (this->getDistance(*potentialMate) <= MATE_DISTANCE));
   }
   else if((this->getGender() == FEMALE) &&
           (potentialMate->getGender() == MALE))
   {
      mateFlag = ((!(this->isPregnant())) &&
                  (this->getNextAction() == MATE) &&
                  (potentialMate->getNextAction() == MATE)  &&
                  (this->getDistance(*potentialMate) <= MATE_DISTANCE));
   }

   return mateFlag;
}//end Cheetah::canMate()


//----------------------------------------------------------------------
// Function:   bool Cheetah::mateEligible()
// Return Val:
// Parameter:
// Purpose:    return whether Cheetah is eligible to mate
//----------------------------------------------------------------------
bool Cheetah::mateEligible(Cheetah *potentialMate)
{
   bool mateEligibleFlag = false;

   if((this->getGender() == MALE) &&
      (potentialMate->getGender() == FEMALE))
   {
     mateEligibleFlag = (!(potentialMate->isPregnant()) &&
                  (potentialMate->isInSeason()) &&
                  (potentialMate->getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE));
   }
   else if((this->getGender() == FEMALE) &&
           (potentialMate->getGender() == MALE))
   {
     mateEligibleFlag = (!(this->isPregnant()) &&
                  (this->isInSeason()) &&
                  (potentialMate->getAge() >= MATE_AGE) &&
                  (this->getAge() >= MATE_AGE));
   }

   return mateEligibleFlag;

}//end Cheetah::mateEligible()


//----------------------------------------------------------------------
//Function:   Cheetah::mate ()
//Return Val: true / false
//Parameter:  mate
//Purpose:    begin pregnancy once two cheetah mate
//----------------------------------------------------------------------
void Cheetah::mate(Cheetah *mate)
{
   if(this->getGender() == MALE)
```

```
   {
      mate->setPregnant(true);
      mate->pregPtr->partnerId = this->getIdNum();
      mate->pregPtr->maleSpeed = this->getSpeed();
      mate->pregPtr->gestationTime = 0;
   }
   else
   {
      this->setPregnant(true);
      this->pregPtr->partnerId = mate->getIdNum();
      this->pregPtr->maleSpeed = mate->getSpeed();
      this->pregPtr->gestationTime = 0;
   }

   return;

}//end function Cheetah::mate()


//----------------------------------------------------------------------
// Function:   updatePosition()
// Return Val:
// Parameter:
// Purpose:    allow the cheetah to update position
//----------------------------------------------------------------------
void Cheetah::updatePosition(int time)
{
   //check to make sure still has enough energy to keep hunting
   if(this->getEnergyLevel() < STOP_HUNTING_LEVEL)
   {
      this->setRest(true);
      this->setNextAction(NOTHING);
   }//end if

   switch(this->getNextAction())
   {
      case MATE :
      {
         this->setSpeedOfNextMove(REGULAR);
         this->setEnergyLevel(this->getEnergyLevel() - REGULAR_ENERGY_PENALTY);
         this->moveTo(this->getMoveToLocation());
         break;
      }
      case NOTHING :
      {
         if(this->isResting() || (rand() < RAND_MAX/2))
         {
            this->setSpeedOfNextMove(REST);
            this->setEnergyLevel(this->getEnergyLevel() + REST_ENERGY_GAIN);
         }
         else
         {
            this->setSpeedOfNextMove(REGULAR);
            this->setEnergyLevel(this->getEnergyLevel() - REGULAR_ENERGY_PENALTY);
         }
         this->move();
         break;
      }
      case FEED :
```

```
      {
         this->setSpeedOfNextMove(RUN);
         this->setEnergyLevel(this->getEnergyLevel() - RUN_ENERGY_PENALTY);
         this->moveTo(this->getMoveToLocation());
         break;
      }
      case GATHER :
      {
         this->setSpeedOfNextMove(REGULAR);
         this->setEnergyLevel(this->getEnergyLevel() - REGULAR_ENERGY_PENALTY);
         this->moveTo(this->getMoveToLocation());
         break;
      }
      case AVOID :
      {
         this->setSpeedOfNextMove(REGULAR);
         this->setEnergyLevel(this->getEnergyLevel() - REGULAR_ENERGY_PENALTY);
         this->moveFrom(this->getMoveFromLocation());
         break;
      }
      case FLEE :
      {
         this->setSpeedOfNextMove(RUN);
         this->setEnergyLevel(this->getEnergyLevel() - RUN_ENERGY_PENALTY);
         this->moveFrom(this->getMoveFromLocation());
         break;
      }
      default :  //CHASE
      {
         this->setSpeedOfNextMove(RUN);
         this->setEnergyLevel(this->getEnergyLevel() - RUN_ENERGY_PENALTY);
         this->moveTo(this->getMoveToLocation());
         break;
      }

   }//end switch getNextAction()


   if(this->isPregnant())
      {
         if(this->pregPtr->gestationTime == GESTATION_PERIOD)
         {
            int litter = this->litterSize();
            npsVec3f tempLocation;
            this->getPosition(tempLocation);

            Cheetah *babyCheetah;
            for (int ix = 1; ix <= litter; ix++)
            {
               babyCheetah = this->giveBirth(this->getSpeed(), this->pregPtr->maleSpee
===> d,
                              this->getGeneration(), tempLocation);

               if(babyCheetah->diesAsInfant())
               {
                  babyCheetah->setDeathIndicator(INFANT_MORTALITY);
               }
               else
```

```
               {
                  cout<<"newborn cheetah"<<endl;
               }
               if (babyCheetah->getGender() == FEMALE)
                  babyCheetah->pregPtr = new Pregnancy;
            }//end for (ix) - create litter of size litter

            this->setPregnant(false);
         }//end if pregancy gestation time > GESTATION_TIME
         else
         {
         this->pregPtr->gestationTime++;
         }//end else
      }//end if aix->isPregnant()


      this->growOlder();

         //check age and if over MAX_AGE then set deathIndicator to OLD_AGE
         if(this->getAge() == MAX_AGE)
         {
            this->setDeathIndicator(OLD_AGE);
         }

         //Last thing we do is check to make sure the cheetah didn't die two
         //time steps ago.  if so take out of world otherwise increment counter
         //this allows the other animals to sense this one and learn how it died

         if((this->getDeathIndicator() != NOT_DEAD) && (this->getDeathCounter() < 2))
         {
            this->setDeathCounter(this->getDeathCounter() + 1);
         }//end if getDeathIndicator()
         else if((this->getDeathIndicator() != NOT_DEAD) && (this->getDeathCounter() >
===> =2))
         {
            this->setRemove();
         }//end if else getDeathIndicator()

}//end updatePosition()


//----------------------------------------------------------------------
// Function:   sense()
// Return Val:
// Parameter:
// Purpose:    allow the cheetah to sense environment and decide which
//             action to take next
//----------------------------------------------------------------------
void Cheetah::sense(int time)//npsAgent *sensedAgent)
{
   if(this->isResting())
   {
      if(this->getEnergyLevel() > RESUME_HUNTING_LEVEL)
      {
         this->setRest(false);
      }
   }
   else
```

```
   {

      if((time%ONE_YEAR > BEGIN_SEASON) && (time%ONE_YEAR < END_SEASON))
         this->setInSeason(true);
      else
         this->setInSeason(false);

      int currentSensingRange   = 0;

      float closestPartner    = 100,
            closestFriend      = 100,
            closestEnemy      = 100,
            closestUnknown    = 100,
            closestFood       = 100,
            closestPredator    = 100;
      bool  sensedFood        = false;

      npsVec3f moveToLocation;

      switch(this->getSpeedOfNextMove())
      {
         case REST :
         {
            currentSensingRange = REST_SENSING_RANGE;
            break;
         }
         case REGULAR :
         {
            currentSensingRange = REGULAR_SENSING_RANGE;
            break;
         }
         default://case RUN
         {
            currentSensingRange = RUN_SENSING_RANGE;
            break;
         }
      }//end switch

      this->setNextAction(NOTHING); //reset this for tracking

      int numAgents = bbListedClass<npsAgent>::getNumObjects();

      for (int j = 0; j < numAgents; j++)
      {
         npsAgent *sensedAgent = bbListedClass<npsAgent>::getObject(j);

         if ((this->getDistance(*sensedAgent) <= currentSensingRange) &&
             (this->getName() != sensedAgent->getName()))
         {

            switch (this->getRelationship(sensedAgent))
            {
            case PREDATOR:
               {
                  sensePredators(sensedAgent, closestPredator);
                  break;
               }
            case ENEMY:
               {
                  senseEnemy(sensedAgent, closestEnemy);
                  break;
               }
            case FRIENDLY:
               {
                  if((this->getNextAction() != FLEE) ||
                     (this->getNextAction() != MATE))
                     senseFriendly(sensedAgent, closestFriend, closestPartner);
                  break;
               }
            case FOOD:
               {
                  if((this->getNextAction() != FLEE) ||
                     (this->getNextAction() != MATE))
                     senseFood(sensedAgent, closestFood);
                  break;
               }
            default:  //case UNKNOWN:
               {
                  senseUnknown(sensedAgent, closestUnknown);
                  break;
               }
            }//end switch
         }//end if (distanceToAgent...)

      }//end for (j<numAgents)
   }//end ifelse(isResting)

}//end Cheetah::sense()


//-------------------------------------------------------------------
// Function:   sensePredators()
// Return Val:
// Parameter:  npsAgent, float
// Purpose:    allow agent to sense a known predator and decide what to
//             next
//-------------------------------------------------------------------
void Cheetah::sensePredators(npsAgent *agent, float &closestPredator)
{
   //nothing defined for predators at this time

}//end sensePredator


//-------------------------------------------------------------------
// Function:   senseFriendly()
// Return Val:
// Parameter:  npsAgent, float, float
// Purpose:    allow agent to sense a known predator and decide what to
//             next
//-------------------------------------------------------------------
void Cheetah::senseFriendly(npsAgent *agent, float &closestToAvoid,
                            float &closestPartner)
{
   float distanceToAgent = this->getDistance(*agent);

   npsVec3f moveToLocation, moveFromLocation;

   if (this->isSameAgentType(agent))
   {
      if (this->canMate((Cheetah *)agent))
      {
         this->mate((Cheetah *)agent);
         this->setNextAction(NOTHING);
      }//end if canMate()
      else if((this->mateEligible((Cheetah *)agent)) &&
              (distanceToAgent < closestPartner))
      {
         closestPartner = distanceToAgent;
         agent->getPosition(moveToLocation);
         this->setMoveToLocation(moveToLocation);
         this->setNextAction(MATE);
      }//end else if mateEligible(tcix)
      else if((this->getNextAction() != MATE) &&
              (this->getNextAction() != CHASE))
      {
         if((distanceToAgent < closestToAvoid) &&
            (distanceToAgent < AVOID_RANGE))
         closestToAvoid = distanceToAgent;
         agent->getPosition(moveFromLocation);
         this->setMoveFromLocation(moveFromLocation);
         this->setNextAction(AVOID);
      }//end else if (!MATE)

   }//end if isSameAgent()
   else
   {
      //no interaction defined for other friendly agents at this time
   }//end if/else(isSameAgentType)

}//end senseFriendly()


//-------------------------------------------------------------------
// Function:   senseFood()
// Return Val:
// Parameter:  npsAgent, npsVec3f
// Purpose:    allow agent to sense a known food source and remember if
//             it is the closest one
//-------------------------------------------------------------------
void Cheetah::senseFood(npsAgent *agent, float &closestFood)
{
   npsVec3f moveToLocation;

   if((this->getNextAction() != MATE) && !(this->isResting()) &&
      (this->getEnergyLevel() < HIGH_ENERGY_LEVEL))
   {
      if(this->getDistance(*agent) < closestFood)
      {
         if(agent->isKilled(*this))
         {
            this->setEnergyLevel(this->getEnergyLevel() + ENERGY_BOOST);
            this->setNextAction(NOTHING);
            cout<<"Antelope killed"<<endl;
         }//end if canKill()
         else
         {
            if(this->getDistance(*agent) < closestFood)
            {
               closestFood = this->getDistance(*agent);
               agent->getPosition(moveToLocation);
               this->setMoveToLocation(moveToLocation);
               this->setNextAction(CHASE);
            }//end if getDistance

         }//end if else agent->isKilled()
      }//end if cix->getX()

   }//end if (!foundPartner)

}//end senseFood


//-------------------------------------------------------------------
// Function:   senseEnemy()
// Return Val:
// Parameter:  npsAgent, int
// Purpose:    allow agent to sense a known predator and decide what to
//             next
//-------------------------------------------------------------------
void Cheetah::senseEnemy(npsAgent *agent, float &closestEnemy)
{
   //do nothing for Enemies at this time

}//end sensePredator


//-------------------------------------------------------------------
// Function:   senseUnknown()
// Return Val:
// Parameter:  npsAgent, int
// Purpose:    allow agent to sense an unknown agent and decide how to
//             interact with it
//-------------------------------------------------------------------
void Cheetah::senseUnknown(npsAgent *agent, float &closestUnknown)
{
   if(!(strcmp(agent->getAgentType(),"Antelope")))
   {
      this->addToFood(agent->getAgentType());
   }
   else
   {
      //do nothing at this time
   }

}//end sensePredator


//-------------------------------------------------------------------
// Function:   int litterSize()
// Return Val: int number in litter
// Parameter:
// Purpose:    return a random number of cheetah in a litter
//-------------------------------------------------------------------
```

```
int Cheetah::litterSize()
{
   int litter = 1;

   double randNum = npsAgent::myRand();

   if(randNum <= 0.05)
      litter = 1;
   else if((randNum > 0.05) && (randNum <= 0.15))
      litter = 2;
   else if((randNum > 0.15) && (randNum <= 0.3))
      litter = 3;
   else if((randNum > 0.3) && (randNum <= 0.7))
      litter = 4;
   else if((randNum > 0.7) && (randNum <= 0.85))
      litter = 5;
   else if((randNum > 0.85) && (randNum <= 0.95))
      litter = 6;
   else //if randNum > .95
      litter = 7;

   return litter;

}//end Cheetah::litterSize()


//---------------------------------------------------------------------
// Function:   bool Cheetah::diesAsInfant()
// Return Val:
// Parameter:
// Purpose:    return whether infant dies or not
//---------------------------------------------------------------------
bool Cheetah::diesAsInfant()
{
   return (npsAgent::myRand() < INFANT_MORTALITY_RATE);

}//end Cheetah::mortality()


//---------------------------------------------------------------------
// Function:   isKilled(agent)
// Return Val: bool
// Parameter:
// Purpose:    every animal must be able to determine if it can kill
//             another animal
//---------------------------------------------------------------------
bool Cheetah::isKilled(npsAgent &agent)
{
   bool killFlag = false;

   if((this->getDeathIndicator() == NOT_DEAD) &&
      (this->getRelationship(&agent) == PREDATOR) &&
      (this->getDistance(agent) <= KILLED_RADIUS) &&
      (npsAgent::myRand() < KILL_PROBABILITY))

      killFlag = true;

   return killFlag;

}//end Cheetah::isKilled()

//end file Cheetah.c


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: cheetahApp.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: The application class for the Cheetah Module - used
//              to instantiate Cheetah agents when requested
//
// March 1999 Master Thesis
// ******************************************************************

#ifndef _cheetahApp_h
#define _cheetahApp_h

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "cheetah.h"

// ******************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

void initCheetahApp();
void exitCheetahApp();

#endif // _CheetahApp_h


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: cheetahApp.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: The application class for the Cheetah Module - used
//              to instantiate Cheetah agents when requested
//
// March 1999 Master Thesis
// ******************************************************************

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "cheetahApp.h"
#include "bbModule.h"
#include "bbThread.h"
#include "bbCallback.h"
#include "npsKeyboard.h"
#include "bbEventResponse.h"

#include <math.h>
#include <GL/gl.h>

// ******************************************************************
//   DEFINES & FILE SCOPE VARIABLES
// ******************************************************************

bbThread    *thread;
static int   numCheetah = 0;

void initKeyboardModule();
void initCheetahFunc(int numCheetah);
void initBoidFunc(void *object, bbData *data);
char* integerToString(int inNum);

// ******************************************************************
//   CODE
// ******************************************************************


void initCheetahApp()
{
   initKeyboardModule();
}


void exitNpsAgentApp()
{

}


void initKeyboardModule()
{
   void            getNumFunc(void *object, bbData *data);
   npsKeyboard     *keyboard;
   bbEventResponse   *eventResponse;
   bbCallback      *callback;

   // get the keyboard device
   keyboard = npsKeyboard::getInstance();

   // set up get number of cheetah by using the 'a' key
   eventResponse = new bbEventResponse(npsKeyboard::KEY_C |
                                       npsKeyboard::UP_TRANS);
   callback = new bbCallback();
   callback->setFunc(getNumFunc);
   eventResponse->addCallbackLast(callback);
   keyboard->addEventResponse(eventResponse);
}

void  getNumFunc(void *object, bbData *data)
{
   cout <<"How many cheetah would you like to create? " << endl;
   cin >> numCheetah;

   initCheetahFunc(numCheetah);
}


void initCheetahFunc(int numCheetah)
{
   Cheetah    *myCheetah;
   npsVec3f   position;

   for (int i = 0; i < numCheetah; i++)
   {
      char  name[64];

      myCheetah = new Cheetah();

      strcpy(name, "Cheetah");
      strcat(name, myCheetah->integerToString(myCheetah->getIdNum()));

      myCheetah->setRandomPosition();
      myCheetah->addToFriends(myCheetah->getAgentType());

      myCheetah->setName(name);
      cout<<"name = "<<name<<"type "<<myCheetah->getAgentType()<<endl;
   }
}


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: plant.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Definition of the Plant agent for use in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

#ifndef _plant_h
#define _plant_h

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "npsAgent.h"
#include "npsAgentApi.h"
#include "npsVec3f.h"


// ******************************************************************
//   DEFINES
// ******************************************************************

class Plant;
#ifdef _Plant_c
  ACE_EXPORT_SINGLETON_DECLARATION(bbSafeClass<Plant>);
```

```
  ACE_EXPORT_SINGLETON_DECLARATION(bbListedClass<Plant>);
#else
  ACE_IMPORT_SINGLETON_DECLARATION(bbSafeClass<Plant>);
  ACE_IMPORT_SINGLETON_DECLARATION(bbListedClass<Plant>);
#endif

// ******************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

class AGENT_API Plant: public npsAgent{

private:

   //nothing specific for plants was implemented for this
   //project.  Attributes and behaviors much like those in
   //the Animal class could be implemented here.

protected:

    //Constructor
    Plant(bbCallbackFunc *callbackFunc);


public:

    //Default Destructor - does nothing at this time
    ~Plant();

    //update position or sense any other agents
    virtual void updatePosition(int time) = 0;
    virtual void sense(int time) = 0;

    //check for being killed
    virtual bool isKilled(npsAgent &agent) = 0;

};

#endif // _Plant


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: plant.C
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Definition of the Plant agent for use in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

#define _plant_c

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************
```

```
#include "Plant.h"

//******************************************************************
//  DEFINES AND FILE SCOPE CONSTANTS
//******************************************************************

static int  numPlant = 0;

//------------------------------------------------------------------
// Function:   Plant::Plant()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//------------------------------------------------------------------
Plant::Plant (bbCallbackFunc *_callbackFunc)
   :npsAgent(_callbackFunc )
{
   //nothing to construct at this time

}//end Plant::Plant()



//------------------------------------------------------------------
// Function:   Plant::~Plant()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//------------------------------------------------------------------
Plant::~Plant ()
{
   //do nothing at this point
}//end Plant::~Plant()
//end file Plant.c


// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: grass.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Definition of the grass agent for use in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

#ifndef _grass_h
#define _grass_h

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "npsAgentApi.h"
#include "Plant.h"
```

```
// ******************************************************************
//   DEFINES
// ******************************************************************

const float PATCH_SIZE = 6.0f;

// ******************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ******************************************************************

class Grass: public Plant{

private:

   int idNum;

protected:


public:

    //Constructor
    Grass(bbCallbackFunc *callbackFunc);

    //Constructor
    Grass();


    //Default Destructor - does nothing at this time
    ~Grass();

   int getIdNum();

   //will not update position or sense any other agents
   void updatePosition(int time);
   void sense(int time);
   bool isKilled(npsAgent &agent);

};

// ******************************************************************
//   INLINED MEMBER FUNCTIONS
// ******************************************************************

inline Grass::getIdNum()
{
   return(idNum);
}

inline void Grass::updatePosition(int time){}
inline void Grass::sense(int time){}
inline bool Grass::isKilled(npsAgent &agent){return false;}

#endif // _grass_h
```

```
// ******************************************************************
//   EXECUTIVE SUMMARY
// Module Name: grass.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Definition of the grass agent for use in npsAgent
//
// March 1999 Master Thesis
// ******************************************************************

// ******************************************************************
//   INCLUDES AND EXTERNS
// ******************************************************************

#include "grass.h"
#include <GL/gl.h>

//******************************************************************
//  DEFINES AND FILE SCOPE CONSTANTS
//******************************************************************
void initGeomFunc(void *object, bbData *data);

static int  numGrass = 0;

//------------------------------------------------------------------
// Function:   Grass::Grass()
// Return Val: None
// Parameter:  None
// Purpose:    Default constructor
//------------------------------------------------------------------
Grass::Grass ()
   :Plant(initGeomFunc), idNum(numGrass++)
{
   this->setAgentType("Grass");

}//end Grass::Grass()

//------------------------------------------------------------------
// Function:   Grass::~Grass()
// Return Val: None
// Parameter:  None
// Purpose:    Default destructor
//------------------------------------------------------------------
Grass::~Grass ()
{
   //do nothing at this point
}//end Grass::~Grass()

//------------------------------------------------------------------
// Function:   initGeomFunc()
// Return Val:
// Parameter:
// Purpose:    provides OpenGL calls from which Babmoo will draw the grass
//------------------------------------------------------------------
void initGeomFunc(void *object, bbData *data)
{
  GLfloat   coords[4][3] = { { -PATCH_SIZE, -0.5f, -PATCH_SIZE},// front
```

```
                { PATCH_SIZE, -0.5f,-PATCH_SIZE},   // back left
                { PATCH_SIZE, -0.5f, PATCH_SIZE},   // back right
                {-PATCH_SIZE, -0.5f, PATCH_SIZE} // top
                };

   glShadeModel(GL_FLAT);
   glBegin(GL_POLYGON);
     {
     glColor3f(0.5f, 0.7f, 0.5f);  // bottom
     glVertex3fv(coords[0]);
     glVertex3fv(coords[1]);
     glVertex3fv(coords[2]);
     glVertex3fv(coords[3]);
     }
   glEnd();
   glShadeModel(GL_SMOOTH);
}
//end file Grass.c


// *****************************************************************
//   EXECUTIVE SUMMARY
// Module Name: grassApp.h
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
// Description: Aplication file for the grass agent for use in npsAgent -
//              used to instantiate grass agents
//
// March 1999 Master Thesis
// *****************************************************************

#ifndef _grassApp_h
#define _grassApp_h

// *****************************************************************
//   INCLUDES AND EXTERNS
// *****************************************************************

#include "grass.h"

// *****************************************************************
//   FUNCTION PROTOTYPE SPECIFICATIONS
// *****************************************************************

void initGrassApp();
void exitGrassApp();

#endif // _GrassApp_h


// *****************************************************************
//   EXECUTIVE SUMMARY
// Module Name: grassApp.c
//
// Authors:     Mark A. Boyd  maboyd@bigfoot.com
//              Todd A. Gagnon todd@gagnon.com
//
```

```
// Description: Aplication file for the grass agent for use in npsAgent -
//              used to instantiate grass agents
//
// March 1999 Master Thesis
// *****************************************************************

// *****************************************************************
//   INCLUDES AND EXTERNS
// *****************************************************************

#include "grassApp.h"
#include "bbModule.h"
#include "bbThread.h"
#include "bbCallback.h"
#include "npsKeyboard.h"
#include "bbEventResponse.h"

#include <math.h>
#include <GL/gl.h>

// *****************************************************************
//   DEFINES & FILE SCOPE VARIABLES
// *****************************************************************

static int   numGrass = 0;

void initKeyboardModule();
void initGrassFunc(int numGrass);

// *****************************************************************
//   CODE
// *****************************************************************

void initGrassApp()
{
   int numGrass = 0;

   initKeyboardModule();
}//end initGrassApp()


void exitNpsAgentApp()
{
   //do nothing for now
}//end exitGrassApp()


void initKeyboardModule()
{
   void             getNumFunc(void *object, bbData *data);
   npsKeyboard      *keyboard;
   bbEventResponse   *eventResponse;
   bbCallback       *callback;

   // get the keyboard device
   keyboard = npsKeyboard::getInstance();

   // set up get number of grass by using the 'a' key
   eventResponse = new bbEventResponse(npsKeyboard::KEY_G |
```

```
                                       npsKeyboard::UP_TRANS);
   callback = new bbCallback();
   callback->setFunc(getNumFunc);
   eventResponse->addCallbackLast(callback);
   keyboard->addEventResponse(eventResponse);
}//end initKeyboardModule()


void  getNumFunc(void *object, bbData *data)
{
   cout <<"How many grass agents would you like to create? " << endl;
   cin >> numGrass;

   initGrassFunc(numGrass);
}//end getNumFunc()


void initGrassFunc(int numGrass)
{
   Grass    *myGrass;
   npsVec3f     position;

   float sqrFtTotal    = (MAX_X-MIN_X)*(MAX_Z-MIN_Z),
         sqrFtPerPatch = sqrFtTotal/numGrass,
         xOffset       = sqrt(sqrFtPerPatch),
         zOffset       = xOffset;

   int currentX = MIN_X,
       currentZ = MIN_Z;


   for (int i = 0; i < numGrass; i++)
   {
      char  name[64];

      if((currentX+xOffset) <= MAX_X)
         currentX += xOffset;
      else
      {
         currentX = MIN_X + xOffset;
         if((currentZ+zOffset) <= MAX_Z)
            currentZ += zOffset;
         else
            currentZ = MIN_Z + zOffset;
      }

      myGrass = new Grass();

      strcpy(name, "Grass");
      strcat(name, myGrass->integerToString(myGrass->getIdNum()));

      myGrass->setRandomPosition();

      myGrass->setName(name);

      myGrass->getPosition(position);
      cout<<"grass "<<myGrass->getIdNum()<<"in position "<<position<<endl;
   }
```

```
}//end initGrassFunc()

//end file grassApp.c
```

APPENDIX B: GLOSSARY

- adaptability
  - Modify rules of behavior and strategies based on interactions.
- agent
  - Software object with internal states and a set of associated behaviors.
- Bamboo
  - Cross platform, dynamically extensible, virtual environment toolkit.
- emergent behavior
  - Behavior patterns that emerge from the interactions of agents but are not inherent to the agents themselves.
- dynamic extensibility
  - Applications have the ability to dynamically reconfigure themselves by adding to or altering their functionality during runtime.
- event
  - A change of object attribute value, an interaction between objects, an instantiation of a new object, or a deletion of an existing object.
- interaction
  - An explicit action taken by an agent that can optionally be directed toward other agents including the environment.
- model
  - A physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process.
- simulation
  - A method for implementing a model over time. Also, a technique for testing, analysis, or training in which real-world systems are used, or where real-world and conceptual systems are reproduced by a model.

LIST OF REFERENCES

[1] Zyda, M. and Sheehan, J. (1997). Modeling and Simulation: Linking Entertainment & Defense. Washington, D.C.: National Academy Press.

[2] Holland, J. H. (1998). Emergence. Reading, MA: Helix Books.

[3] Thinking Tools (1999). Agent Based Adaptive Simulation Technology. [On-line] (2 Feb. 98). Available at URL: http://www.thinkingtools.com/html/technology.html

[4] Maxis. (1999). The SimCity Story. [On-line] (27 Jan. 99). Available at URL: http://www.simcity.com/3000/general.html

[5] Maxis. (1999). SimCity 2000. [On-line] (27 Jan. 99). Available at URL: http://www.maxis.com

[6] Williams, R. J. (1995). Using Agent Based Simulations in a Training Environment. [On-line] (28 Jan. 99). Available at URL: http://cbl.leeds.ac.uk/rodw/papers/eurosim-95/

[7] California Department of Food and Agriculture (1998). The Mediterranean Fruit Fly Fact Sheet. [On-line] (2 Feb. 99). Available at URL: http://www.cdfa.ca.gov/pests/medfly/mediterranean_fly.html

[8] Axtelrod, R. (1997). The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration. Princeton, NJ: Princeton University Press.

[9] Reynolds, C. (1997). Individual-Based Models. [On-line] (20 Jan. 99) Available at URL: http://hmt.com/cwr/ibm.html

[10] Axtell, R., and Epstein, J. M. (1996). Growing Artificial Societies: Social Science for the Bottom Up. Washington, D.C.: The Brookings Institute.

[11] Ziemke, T. (1998). Adaptive Behavior in Autonomous Agents, Presence, volume 7, number 6, December 1998.

[12] Casti, J. L. (1997). Would-be Worlds: How Simulation is Changing the Frontiers of Science. New York, NY: John Wiley & Sons, Inc.

[13] Hofstadter, D. R. (1979). Gödel, Escher, Bach: An Eternal Golden Braid. New York, NY: Basic Books.

[14] Watsen, K. and Zyda, M. (1998). Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments. 1998 IEEE Virtual Reality Annual International Symposium (VRAIS - 98), Atlanta, GA.

[15] Watsen, K. and Zyda, M. (1998). Bamboo - Supporting Dynamic Protocols for Virtual Environments. 1998 IMAGE Conference, Scottsdale, AZ.

[16] Zyda, M. (1999). Academic Associate and Chair of the Modeling, Virtual Environments, and Simulation Academic Group, Naval Postgraduate School, Monterey, CA.

[17] Darken, R. (1999). Assistant Professor of Computer Science and Chair of the Modeling, Virtual Environments, and Simulation Human-Computer Interaction Track, Naval Postgraduate School, Monterey, CA.

[18] Watsen, K. (1999). Senior Development Architect for Bamboo, Naval Postgraduate School, Monterey, CA.

BIBLIOGRAPHY

Axtell, R., and Epstein, J. M. (1996). Growing Artificial Societies: Social Science from the Bottom Up. Washington, D.C.: The Brookings Institute.

Axtelrod, R. (1997). The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration. Princeton, NJ: Princeton University Press.

California Department of Food and Agriculture (1998). The Mediterranean Fruit Fly Fact Sheet. [On-line] (2 Feb. 99). Available at URL: http://www.cdfa.ca.gov/pests/medfly/mediterranean_fly.html

Campos, A. M. C. and Hill, D. R. C. Web-Based Simulation of Agents Behaviors. [On-line] (15 Jan. 99). Available at URL: http://www.isima.fr/scs/wbms/d4/Websim.html

Casti, J. L. (1997). Would-be Worlds: How Simulation is Changing the Frontiers of Science. New York, NY: John Wiley & Sons, Inc.

Deitel, H. M. and Deitel, P. J. (1994). C++ How to Program. Englewood Cliffs, NJ: Prentice Hall.

Hofstadter, D. R. (1979). Gödel, Escher, Bach: An Eternal Golden Braid. New York, NY: Basic Books.

Holland, J. H. (1995). Hidden Order: How Adaption Builds Complexity. Reading, MA: Perseus Books.

Holland, J. H. (1998). Emergence. Reading, MA: Helix Books.

Honegger, B. (1999). VR Project to Simulate Whole Navy. Campus News, volume 6, issue 8. February 26, 1999.

Jennings, N. R. and Wooldridge, M. (1995). Intelligent Agents: Theory and Practice. Knowledge Engineering Review, October 1994.

Laird, J. E. (1998). Knowledge-based Multiagent Coordination, Presence, volume 7, number 6, December 1998.

Liles, S. W., Watsen, K. and Zyda, M. (1998). Dynamic Discovery of Simulation Entities Using Bamboo and HLA. 1998 Simulation Interoperability Workshop, Orlando, FL.

Lock, J. D. (1999). To Fight with Intrepidity ... The Complete History of the U.S. Army Rangers 1622 to Present. New York, NY: Pocket Books.

Maxis. (1999). The SimCity Story. [On-line] (27 Jan. 99). Available at URL: http://www.simcity.com/3000/general.html

Maxis. (1999). SimCity 2000. [On-line] (27 Jan. 99). Available at URL: http://www.maxis.com

Minar, N., Burkhart, R., Langton, C., and Askenazi, M. (1996). The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations. [On-line] (21 Jan. 99) Available at URL: http://www.santafe.edu/projects/swarm/intro-material.html

Prosise, J. (1996). Programming Windows 95 with MFC. Redmond, WA: Microsoft Press.

Resnick, M. (1998). Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. Cambridge, MA: The MIT Press.

Reynolds, C. (1997). Individual-Based Models. [On-line] (20 Jan. 99) Available at URL: http://hmt.com/cwr/ibm.html

Thinking Tools (1999). Agent Based Adaptive Simulation Technology. [On-line] (2 Feb. 98). Available at URL: http://www.thinkingtools.com/html/technology.html

Watsen, K. and Zyda, M. (1998). Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments. 1998 IEEE Virtual Reality Annual International Symposium (VRAIS - 98), Atlanta, GA.

Watsen, K. and Zyda, M. (1998). Bamboo - Supporting Dynamic Protocols for Virtual Environments. 1998 IMAGE Conference, Scottsdale, AZ.

Williams, R. J. (1995). Simulation for Public Order Training and Preplanning. [On-line] (28 Jan. 99). Available at URL: http://cbl.leeds.ac.uk/rodw/papers/eurosim-95/

Williams, R. J. (1995). Using Agent Based Simulations for Training. [On-line] (15 Jan. 99). Available at URL: http://cbl.leeds.ac.uk/rodw/papers/eurosim-95/

Williams, R. J. (1995). Using Agent Based Simulations in a Training Environment. [On-line] (28 Jan. 99). Available at URL: http://cbl.leeds.ac.uk/rodw/papers/eurosim-95/

Wooldridge, M. and Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. Knowledge Engineering Review, October 1994.

Ziemke, T. (1998). Adaptive Behavior in Autonomous Agents, Presence, volume 7, number 6, December 1998.

Zyda, M. and Sheehan, J. (1997). Modeling and Simulation: Linking Entertainment & Defense. Washington, D.C.: National Academy Press.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Capt. Steve Chapman, USN 1
   N6M
   2000 Navy Pentagon
   Room 4C445
   Washington, DC 20350-2000

4. George Phillips 1
   CNO, N6M1
   2000 Navy Pentagon
   Room 4C445
   Washingon, DC 20350-2000

5. Mike Macedonia 1
   Chief Scientist and Technical Director
   US Army STRICOM
   12350 Research Parkway
   Orlando, FL 32826-3276

6. National Simulation Center (NSC) 1
   ATTN:ATZL-NSC (Jerry Ham)
   410 Kearney Avenue --- Building 45
   Fort Leavenworth, KS 66027-1306

7. Director 1
   Office of Science & Innovation
   OSI, MCCDC
   3300 Russell Road
   Quantico, VA 22134-5021

8. Capt. Dennis McBride, USN 1
   Office of Naval Research (341)
   800 No. Quincy Street
   Arlington, VA 22217-5660

9. Col. Crash Konwin, USAF 1
DMSO
1901 N. Beauregard St.
Suite 504
Alexandria, VA 22311

10. Sid Kissen 1
National Security Agency
Attn: S312
9800 Savage Road
Fort George G. Meade, MD 20755

11. Mark A. Boyd 1
39062 White Fir Lane
Corvallis, Oregon 97330

12. Todd A. Gagnon 1
1278 North Main Street
Brewer, Maine 04412

13. John Hiles 1
22 Deer Stalker Path
Monterey, California 93940

14. Commanding Officer 1
Attn: Code 30
Navy Information Warfare Activity
9800 Savage Road
Fort Meade, Maryland 20755-6000

15. Paul Chatelier 1
Office of Science and Technology Policy
Education and Training Initiative
1901 North Beauregard Street, Suite 510
Alexandria, VA 22311